

AD-A085 827

GENERAL ELECTRIC CO ARLINGTON VA INFORMATION SYSTEMS--ETC F/6 9/2
MEASUREMENT AND EXPERIMENTATION IN SOFTWARE ENGINEERING.(U)

APR 80 B CURTIS

N00014-79-C-0595

UNCLASSIFIED

TR-80-388200-1

NL

1 of 1

AD-A085 827

1 of 1

END

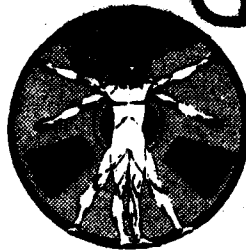
DATE

FILED

8-80

DTIC

ADA 085827



**Software
Management
Research**

LEVEL

**MEASUREMENT AND EXPERIMENTATION
IN SOFTWARE ENGINEERING**

BILL CURTIS

**DTIC
ELECTRONIC
JUN 30 1980**

**TR-80-388200-1
APRIL 1980**

This document has been approved
for public release and sale; its
distribution is unlimited.

DDC FILE COPY

**GENERAL ELECTRIC
INFORMATION SYSTEMS PROGRAMS
ARLINGTON, VIRGINIA**

80 6 27 104

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
	AD-A065 827	
4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED	
6 Measurement and Experimentation in Software Engineering.	9 Technical Report.	
7. AUTHOR(s)	14	6. PERFORMING ORG. REPORT NUMBER
10 Bill Curtis	15	TR-80-388200-1
8. PERFORMING ORGANIZATION NAME AND ADDRESS		7. CONTRACT OR GRANT NUMBER(s)
Information Systems Programs General Electric Company 1755 Jefferson Davis Hwy., Arlington, VA 22202		N00014-79-C-0595
9. CONTROLLING OFFICE NAME AND ADDRESS		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Engineering Psychology Programs, Code 455 Office of Naval Research Arlington, VA 22217	11	NR196 160
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE
same	1362	30 Apr 80
		13. NUMBER OF PAGES
		58
		15. SECURITY CLASS. (of this report)
		Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
same		
18. SUPPLEMENTARY NOTES		
Technical Monitor: Dr. John J. O'Hare		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
Software engineering, Measurement theory, Software experiments, Software science, Structured programming, Software complexity, Software metrics, Control flow, Experimental method, Modern programming practices.		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)		
The contributions of measurement and experimentation to the state-of-the-art in software engineering are reviewed. The role of measurement in developing theoretical models is discussed, and concerns for reliability and validity are stressed. Current approaches to measuring software characteristics are presented as examples. In particular, software complexity metrics related to control flow, module interconnectedness, and Halstead's Software Science are critiqued. The use of experimental methods in evaluating cause-effect		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

5/c 40744c

20. Continued

relationships is also discussed. Example programs of experimental research which investigated conditional statements and control flow are reviewed. The conclusion argues that advances in software engineering will be related to improvements in the measurement and experimental evaluation of software techniques and practices.

Accession For	
NTIS G&I	☒
DDC TAB	☐
Unannounced	☐
Justification	
By _____	
Distribution _____	
Availability _____	
Dist	
A	

MEASUREMENT AND EXPERIMENTATION
IN SOFTWARE ENGINEERING

Bill Curtis

Software Management Research
Information Systems Programs
General Electric Company
1755 Jefferson Davis Highway
Arlington, Virginia

Submitted to:

Office of Naval Research
Engineering Psychology Programs
Arlington, Virginia

Contract: N00014-79-C-0595
Work Unit: NR 196-160

April 1980

This report will appear in a special issue of the Proceedings of the IEEE concerning software engineering to be published in September 1980.

Approved for public release; distribution unlimited.
Reproduction in whole or in part is permitted for any
purpose of the United States Government.

ABSTRACT

The contributions of measurement and experimentation to the state-of-the-art in software engineering are reviewed. The role of measurement in developing theoretical models is discussed, and concerns for reliability and validity are stressed. Current approaches to measuring software characteristics are presented as examples. In particular, software complexity metrics related to control flow, module interconnectedness, and Halstead's Software Science are described. The use of experimental methods in evaluating cause-effect relationships is also discussed. Example programs of experimental research which investigated conditional statements and control flow are reviewed. The conclusion argues that many advances in software engineering will be related to improvements in the measurement and experimental evaluation of software techniques and practices.

Keywords: Software engineering, Measurement theory, Software experiments, Software science, Structured programming, Software complexity, Software metrics, Control flow, Experimental methods, Modern programming practices.

TABLE OF CONTENTS

Abstract.....	iii
Table of Contents.....	iv
Introduction.....	1
Science and measurement.....	2
Measurement of Software Characteristics.....	9
Uses for software metrics.....	9
Omnibus approaches to quantifying software.....	10
Software complexity.....	14
Control structures.....	17
Software science.....	19
Interconnectedness.....	23
Experimental Evaluation of Software Characteristics.....	25
Cause-effect relationships.....	25
Conditional statements.....	27
Control flow.....	35
Problems in experimental research.....	40
Summary.....	42
Acknowledgements.....	44
References.....	45

INTRODUCTION

The magnitude of costs involved in software development and maintenance magnify the need for a scientific foundation to support programming standards and management decisions. The argument for employing a particular software technique is more convincing if backed by experiments demonstrating its benefit. Rigorous scientific procedures must be applied to studying the development of software systems if we are to transform programming into an engineering discipline. At the core of these procedures is the development of measurement techniques and the determination of cause-effect relationships.

A commitment to measurement and experimentation hopefully begins by focusing on the phenomenon we are trying to explain. Rather than beginning by counting or experimentally manipulating various properties of software, we should first determine what software-related task we wish to understand. Modeling the processes underlying a software task helps identify properties of software that affect performance. Once the process is modeled, we can dissect it with all manner of scientific procedures.

The article on reliability by Musa⁶² in this issue presents a rigorous approach to modeling a software phenomenon. He specifies a set of assumptions about software failures that guide his development of a quantitative measure. Yet, Musa does not stop with a description of his measure. He takes the critically important step of validating his equation with actual data. Further, he does not define his measure on the basis of a one-shot study, but continues to test and refine his model against new data sets.

Statements that a software product has a mean-time-between-failure of 48 hours or satisfies specified timing constraints are grounded in the established measurement disciplines of reliability^{22,62} and performance evaluation. Other important attributes of software, such as its comprehensibility to programmers, have not been adequately defined. A model of how software characteristics affect programmer performance should underlie software engineering techniques which purport to make code more readable or

reduce the mental load of the programmer. The empirical study of such a model requires a disciplined application of measurement and experimental methods.

I will discuss several important principles in measurement and experimentation and review their application in research on how certain software characteristics make a program difficult for a programmer to understand and work with. I will begin with a discussion of how measurement is fundamental to the development of a scientific discipline.

Science and Measurement

Margenau⁵³ argues that the various scientific disciplines can be classified by the degree to which their analytical approach is theoretical rather than correlational. The correlational approach explains phenomena by the degree of relationship among observable events. The theoretical approach attempts to explain these relationships with principles and constructs which are often several levels of abstraction removed from relationships among empirical data.

Torgerson⁸⁸ believes that "the sciences would order themselves in largely the same way [as Margenau's ordering] if they were classified on the...degree to which satisfactory measurement of their important variables has been achieved" (p. 2). We know considerably more about measuring electricity or sound than we do about measuring the comprehensibility of software. Consequently, correlational studies are more characteristic of the behavioral than the physical sciences. According to Lord Kelvin⁴⁶:

When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science.

The development of scientific theory involves relating theoretical

constructs to observable data. Figure 1 illustrates two levels of theoretical modeling as discussed by Margenau and Torgerson. In a well-developed science constructs can be defined in terms of each other and are related by formal equations (e.g., $\text{force} = \text{mass} \times \text{acceleration}$). A model of relationships among constructs becomes a theory when at least some constructs can be operationally defined in terms of observable data.

In a less well-developed science, relationships between theoretical and operationally defined constructs are not necessarily established on a formal mathematical basis, but are logically presumed to exist. Such relationships among operationally defined constructs are often described by correlation or regression coefficients, while their relationships to non-operationally defined theoretical constructs are typically presented in verbal arguments. These presumed relationships are difficult to test, because negative results can be as easily attributed to a poor operational definition of the constructs as to an incorrect modeling of the relationships. In the next section, we will find presumed relationships existing between the hypothetical construct of program comprehensibility and its operational definition in software characteristics.

The development of an operational definition (i.e., the relating of a theoretical construct to observable data) requires a system of measurement. As described by Stevens⁸⁴:

... the process of measurement is the process of mapping empirical properties or relations into a formal model. Measurement is possible because there is a kind of isomorphism between (1) the empirical relations among properties of objects and events and (2) the properties of the formal game in which numerals are the pawns and operators the moves. (p. 24)

Measurement does not define a construct, rather it quantifies a property of the construct. The "brightness" of light and the "intelligence" of

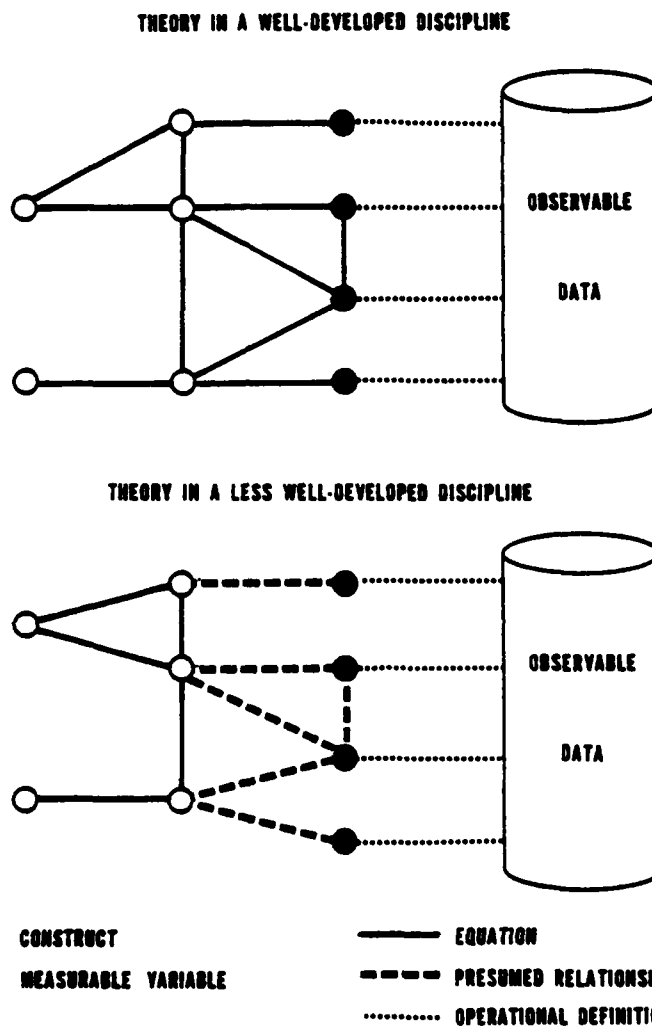


Figure 1. The structure of theory in science.

programmers are represented with a number system. Numbers are that fortunate development which relieves us of reporting the size of a software system with several hundred thousand pebbles.

The value of any empirical study will depend on the reliability and validity of the data. Reliability concerns the extent to which measures are accurate and repeatable⁶⁶. The less random error associated with a measurement, the more reliable it becomes. Two important factors underlying reliability are the internal consistency and the stability of the measure. If a measure is computed as the composite of several other measures, as in adding question scores to obtain an overall test score, then it is important to demonstrate that this composite is internally consistent. That is, all of the elementary measurements must be assessing the same construct and must be interrelated. If unrelated elements are added into a composite then it is difficult to interpret the resulting score.

The other aspect of reliability, stability, implies that an equivalent score would be obtained on repeated collections of data under similar circumstances. The reliability of a measure limits the strength of its relationships to other measures. However, a reliable measure may not be a valid measure of a construct.

Validity has many interpretations, and all seem to concern whether a measure represents what it was designed to assess. Generally, three types of validity are identified which differ in their implications for the measure's ultimate use. Often validity will depend on the thoroughness with which a domain of interest has been covered. This concern for content validity is important for the software quality metrics to be discussed in the next section. Content validity requires an inclusive definition of the domain of interest, such as a definition of the phenomena covered under software complexity. A measure is often said to be "face valid" if it appears to broadly sample the content domain.

Predictive validity involves using the measure to predict the outcome of some event. For instance, does knowing something about software complexity allow one to predict how difficult a program will be to modify?

Predictive validity is determined by a relationship between the measure and a criterion. A number of predictive validity studies will be reported in the next section.

In the less well-developed sciences the skill with which we operationally define constructs is critical in theory building. Construct validity concerns how closely an operational definition yields data related to an abstract construct. Construct validity is of immediate concern in developing software measurements, since many of our models do not rest on mathematical analysis.

Measurement consists of assigning numbers to represent the different states of a property belonging to the object under study. Relationships among these different states determine the type of measurement scale which should be employed in assigning numbers. Stevens⁸⁴ describes four types of scales which are presented in Table 1. The important consideration with scales is that we only operate on numbers in a way which faithfully represents potential events among the properties they measure. That is, in considering jersey numbers (a nominal scale) we would not expect to add a fullback from Texas (#20) to a tackle from Harvard (#79) and end up with a Yugoslavian placekicker (#99). The operation of addition is limited to interval and ratio scales.

The most desirable scales are those which possess ratio properties, because of the broader range of mathematical transformations which can be legitimately applied to such data. The type of measurement scale also limits the type of statistical operations that can be sensibly applied in analyzing data. It makes little sense to add up all the jersey numbers, divide by the total number of players, and then claim that the average player is a center (#51). Since statistical techniques make no assumptions about the type of scale employed, this problem is one in measurement rather than statistical theory.

TABLE 1
TYPES OF MEASUREMENT SCALES

SCALE	APPROPRIATE OPERATIONS*	DESCRIPTION	EXAMPLES
NOMINAL	$=, \neq$	CATEGORIES	SEX, RACE JERSEY NUMBERS
ORDINAL	$<, >$	RANK ORDERINGS	HARDNESS OF MINERALS RANK IN CLASS
INTERVAL	$+, -$	EQUIVALENT INTERVALS BETWEEN NUMBERS	TEMPERATURE (F° AND C°) CALENDAR TIME
RATIO	$+$	EQUIVALENT INTERVALS AND ABSOLUTE ZERO	TEMPERATURE (K°), HEIGHT LINES OF CODE

* THE OPERATIONS LISTED FOR EACH SCALE ARE APPROPRIATE
FOR ALL SCALES LISTED BENEATH IT.

Improved measurement will result from concentration on what we really ought to measure rather than what properties are readily countable. The more rigorous our measurement techniques, the more thoroughly a theoretical model can be tested and calibrated. Thus, progress in a scientific basis for software engineering depends on improved measurement of the fundamental constructs⁴⁵.

MEASUREMENT OF SOFTWARE CHARACTERISTICS

Uses for Software Metrics

Measurements of software characteristics can provide valuable information throughout the software life cycle. During development measurements can be used to predict the resources which will be required in future phases of the project. For instance, metrics developed from the detailed design can be used to predict the amount of effort that will be required to implement and test the code. Metrics developed from the code can be used to predict the number of errors that may be found in subsequent testing or the difficulty involved in modifying a section of code. Because of their potential predictive value, software metrics can be used in at least three ways:

1. Management information tools - As a management tool, metrics provide several types of information. First, they can be used to predict future outcomes as discussed above. Measurements can be developed for costing and sizing at the project level, such as in the models proposed by Freiman and Park³³, Putnam⁶⁹, and Wolverton⁹³. Other models have been developed for estimating productivity^{32,89}. Such metrics allow managers to assess progress, future problems, and resource requirements. If these metrics can be proven reliable and valid indicators of development processes, they provide an excellent source of management visibility into a software project.
2. Measures of software quality - Interest grows in creating quantifiable criteria against which a software product can be judged⁶⁰. An example criterion would be the minimally acceptable mean-time-between-failures. These criteria could be used as either acceptance standards by a software acquisition manager or as guidance to potential problems in the code during software validation and verification⁹⁰.

3. Feedback to software personnel - Elshoff²⁷ has used a software complexity metric to provide feedback to programmers about their code. When a section grows too complex they are instructed to redesign the code until metric values are brought within acceptable limits.

The three uses described above suggest a difference between measures of process and product. Measures of process would include the resource estimation metrics described as potential management tools. Measures of cost and productivity quantify attributes of the development process. However, they convey little information about the actual state of the software product. Measures of the product represent software characteristics as they exist as a given time, but do not indicate how the software has evolved into this state. Measures used for feedback to programmers, or as quality criteria, fall within this second category.

Belady⁵ argues that it will be difficult to develop a metric which can represent both process and product. Development of such a metric or set of metrics will require a model of how software evolves from a set of requirements into an operational program. Charting the sequential phases of the software life cycle will not provide a sufficient model. Some progress is being made on system evolution by Lehman and his colleagues at Imperial College in London^{7,9,47,49} and is discussed by Lehman⁴⁸ in this issue. In the remainder of this section, I will deal with measures of product rather than process.

Omnibus Approaches to Quantifying Software

There have been several attempts to quantify the elusive concept of software quality by developing an arsenal of metrics which quantify numerous factors underlying the concept. The most well-known of these metric systems are those developed by Boehm, Brown, Kaspar, Lipow, MacLeod, and Merritt¹¹, Gilb³⁵, and McCall, Richards, and Walters⁵⁵. The Boehm et al. and McCall et al. approaches are similar, although differing in some of the constructs and metrics they propose. Both of these systems have been developed from an intuitive clustering of software characteristics (Figure 2).

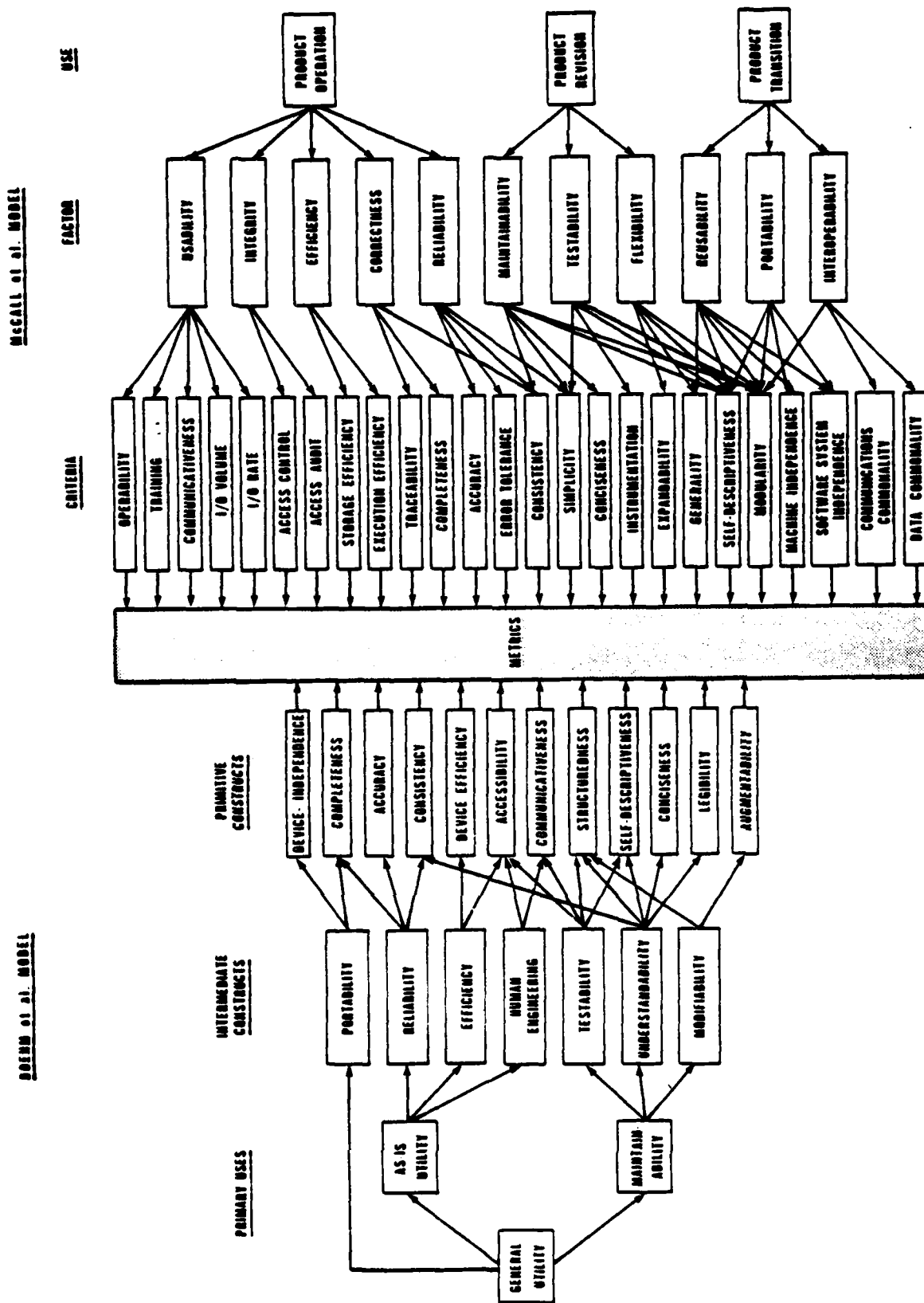


Figure 2. The Doehm et al. and McCall et al. software quality models.

The higher level constructs in each system represent 1) the current behavior of the software, 2) the ease of changing the software, and 3) the ease of converting or interfacing the system. From these primary concerns Boehm et al. develop seven intermediate constructs, while McCall et al. identify eleven quality factors. Beneath this second level Boehm et al. create twelve primitive constructs and McCall et al. define 23 criteria. For instance, at the level of a primitive construct or criterion both Boehm et al. and McCall et al. define a construct labeled "self-descriptiveness". For Boehm et al. this construct underlies the intermediate constructs of testability and understandability, both of which serve the primary use of measuring maintainability. For McCall et al. self-descriptiveness underlies a number of factors included under the domains of product revision and transition.

Primitive constructs and criteria are operationally defined by sets of metrics which provide the guidelines for collecting empirical data. The McCall et al. system defines 41 metrics consisting of 175 specific elements. Thus, the metrics themselves represent composites of more elementary measures. This proliferation of measures should ultimately be reduced to a manageable set which can be automated. Reducing their number will require an empirical evaluation of which metrics carry the most information and how they cluster. There are a number of multivariate statistical techniques available for such analyses⁶¹.

No software project can stay within a reasonable budget and maximize all of the quality factors. The nature of the system under development will determine the proper weighting of quality factors to be achieved in the delivered software. For instance, reliability was a critical concern for Apollo space flight software where human life was constantly at risk. For business systems, however, maintainability is typically of primary importance. In many real-time systems where space or time constraints are critical, efficiency takes precedence. However, optimizing code often lowers its quality as indexed by other factors such as maintainability and portability. Figure 3 presents a tradeoff analysis among quality factors performed by McCall et al.⁵⁵

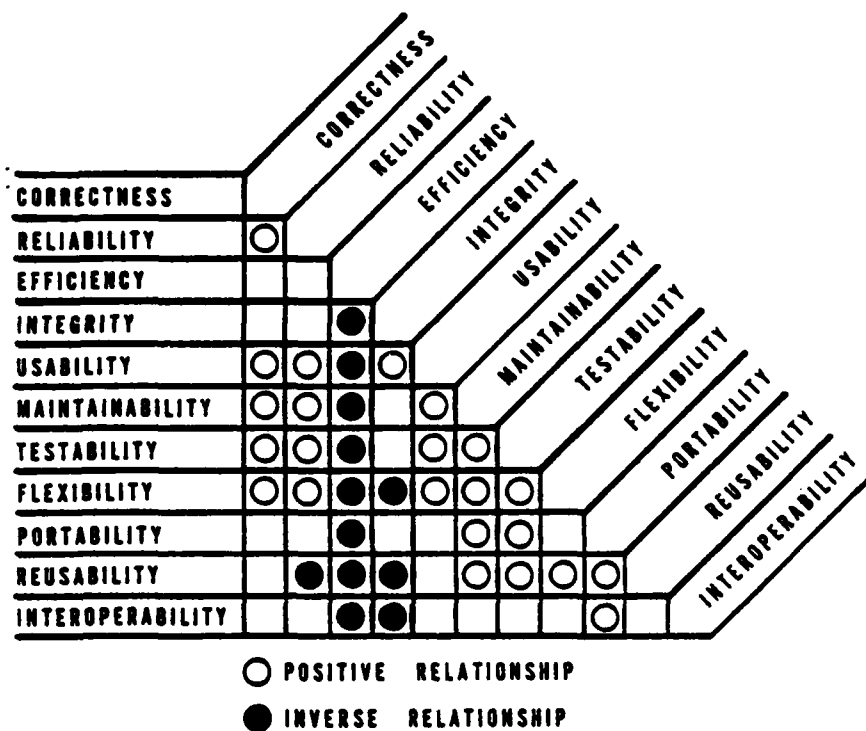


Figure 3. McCall et als.' tradeoff analysis among software quality factors.

The omnibus approach to metric development had its birth in the need for measures of software quality, particularly during system acquisition. However, the development of these metrics has not spawned explanatory theory concerning the processes affected by software characteristics. The value of these metric systems in focusing attention on quality issues is substantial. However, there is still a greater need for quantitative measures which emerge from the modeling of software phenomena. Much of the modeling of software characteristics has been performed in an attempt to understand software complexity.

Software Complexity

The measurement of software complexity is receiving increased attention, since software accounts for a growing proportion of total computer system costs¹⁰. Complexity has been a loosely defined term, and neither Boehm et al. nor McCall et al. included it among their constructs of software quality. Complexity is often considered synonymous with understandability or maintainability.

Two separate focuses have emerged in studying software complexity: computational and psychological complexity. Computational complexity relies on the formal mathematical analysis of such problems as algorithm efficiency and use of machine resources. Rabin⁷⁰ defines this branch of complexity as "the quantitative aspects of the solutions to computational problems" (p. 625). In contrast to this formal analysis, the empirical study of psychological complexity has emerged from the understanding that software development and maintenance are largely human activities⁹¹. Psychological complexity is concerned with the characteristics of software which affect programmer performance.

The investigation of computational and psychological complexity has been carried on without a unifying definition for the construct of software complexity. There do, however, seem to be common threads running through the complexity literature¹⁹ which suggest the following definition:

Complexity is a characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software. (p. 102)

Several important points are implied by this definition. First, the focus of complexity is not merely on the software, but on the software's interactions with other systems. Complexity has little meaning in a vacuum; it requires a point of reference. This reference takes meaning only when developed from other systems such as machines, people, other software packages, etc. It is these systems that are affected by the "complexity" of a piece of software. Worrying about software characteristics in the absence of other systems has merit only in an artistic sense, and measures of "artistic" software are quite arbitrary. However, when there is an external reference (criterion) against which to compare software characteristics, it becomes possible to operationally define complexity.

Second, explicit criteria are not specified. This definition allows mathematicians and psychologists to become strange bedfellows since it does not specify the particular phenomena to be studied. Rather, this definition steps back a level of abstraction and describes the goal of complexity research and the reference against which complexity takes meaning. Complexity is an abstract construct, and operational definitions only capture specific aspects of it.

The second point suggests the third: complexity will have different operational definitions depending on the criterion under study. Operational definitions of complexity must be expressed in terms which are relevant to processes performed in other systems. Complexity is defined as a property of the software interface which affects the interaction between the software and another system. To assess this interaction, we must quantify software characteristics which are relevant to it. A model of software complexity implies not only a quantification of software characteristics, but also a theory of processes in other systems. Thus, the starting point for developing a metric is not an ingenious parsing of software characteristics, but an understanding of how other systems function when they interact with software.

The following steps should be followed in modeling an aspect of software complexity:

- 1) Define (and quantify) the criterion the metric will be developed to predict.
- 2) Develop a model of processes in the interacting system which will affect this criterion.
- 3) Identify the properties of software which affect the operation of these processes.
- 4) Quantify these software characteristics.
- 5) Validate this model with empirical research.

The importance of this last point cannot be overemphasized. Nice theories become even nicer when they work. Preparing for the rigors of empirical evaluation will probably result in fewer metrics and tighter theories. Results from validation studies make excellent report cards on the current state-of-the-art.

Belady⁶ has categorized much of the existing software complexity literature. First, he distinguishes different software characteristics which are measured as an index of complexity: algorithms, control structures, data, or composites of structures and data. In a second dimension he describes the type of measurement employed: informal concept, construct counts, probabilistic/statistical treatments, or relationships extracted from empirical data. Most research has concerned counts of software characteristics, particularly control structures and composites of control structures and data. I will review some of the complexity research in these two areas and compare them to a system level metric.

Control Structures

A number of metrics having a theoretical base in graph theory have been proposed to measure software complexity by assessing the control flow^{8,17,36,54,71,94}. Such metrics typically index the number of branches or paths created by the conditional expressions within a program. McCabe's metric will be described as an example of this approach since it has received the most empirical attention.

M McCabe⁵⁴ 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. When combined these paths generate the complete control structure of the program. McCabe's $v(G)$ can be computed as the number of predicate nodes plus 1, where a predicate node represents a decision point in the program. It can also be computed as the number of regions in a planar graph (a graph in regional form) of the control flow. This latter method is demonstrated in Figure 4.

M McCabe argues that his metric assesses the difficulty of testing a program, since it is a representation of the control paths which must be exercised during testing. From experience he believes that testing and reliability will become greater problems in a section of code whose $v(G)$ exceeds 10.

Basili and Reiter⁴ and Myers⁶⁴ have developed different counting methods for computing cyclomatic complexity. These differences involved counting rules for CASE statements and compound predicates. Definitive data on the most effective counting rules have yet to be presented. Nevertheless, considering alternative counting schemes to those originally posed by the author of a metric is important in refining measurement techniques.

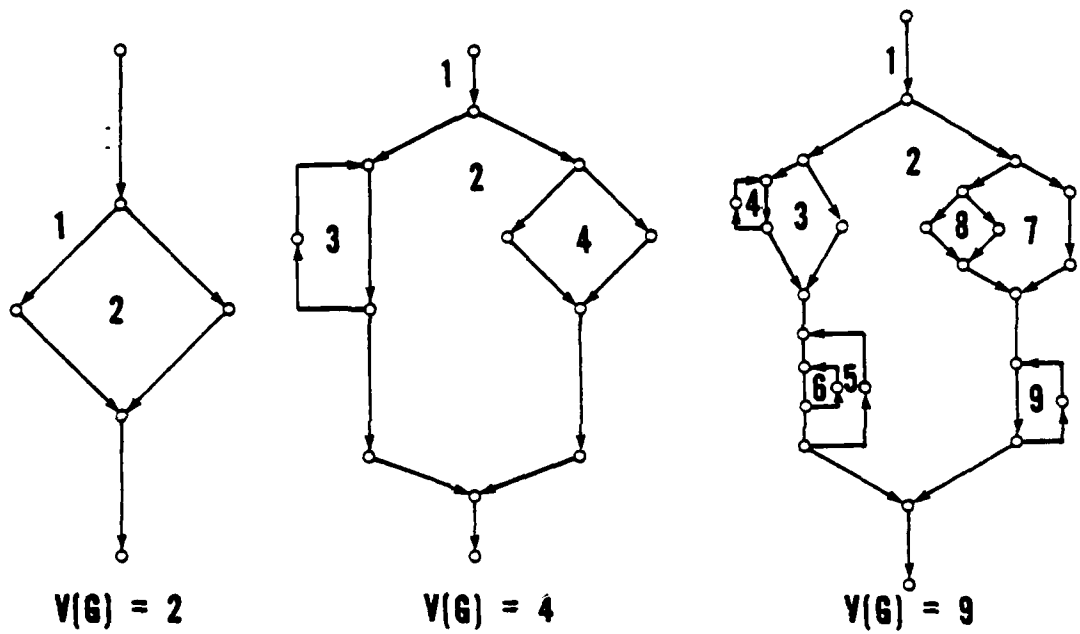


Figure 4. Computation of McCabe's $v(G)$.

Evidence continues to mount that metrics developed from graphs of the control flow are related to important criteria such as the number of errors existing in a segment of code and the time to find and repair such errors^{20,28,73}. Chen¹⁷ developed a variation of the cyclomatic number which indexed the nesting of IF statements and related this to the information-theoretic notion of entropy within the control flow. He reported data from eight programmers indicating that productivity decreased as the value of his metric computed on their programs increased. Thus, the number of control paths appears directly or indirectly related to psychological complexity.

Software Science

The best known and most thoroughly studied of what Belady⁶ classifies as *composite measures of complexity* has emerged from Halstead's theory of Software Science^{40,42}. In 1972, Maurice Halstead argued that algorithms have measurable characteristics analogous to physical laws. Halstead proposed that a number of useful measures could be derived from simple counts of distinct operators and operands and the total frequencies of operators and operands. From these four quantities Halstead developed measures for the overall program length, potential smallest volume of an algorithm, actual volume of an algorithm in a particular language, program level (the difficulty of understanding a program), language level (a constant for a given language), programming effort (number of mental discriminations required to generate a program), program development time, and number of delivered bugs in a system. Two of the most frequently studied measures are calculated as follows:

$$V = (N_1 + N_2) \log_2 (n_1 + n_2)$$

$$E = \frac{n_1 N_2 (N_1 + N_2) \log_2 (n_1 + n_2)}{2n_2}$$

where V is volume, E is effort, and

n_1 = number of unique operators

n_2 = number of unique operands

N_1 = total frequency of operators

N_2 = total frequency of operands

Halstead's theory has been the subject of considerable evaluative research³¹. Correlations often greater than .90 have been reported between Halstead's metrics and such measures as the number of bugs in a program^{8,18,30,34,67}, programming time^{39,75}, debugging time^{20,51}, and algorithm purity^{14,26,41}.

My colleagues and I have evaluated the Halstead and McCabe metrics in a series of four experiments with professional programmers. In the first two experiments²¹ problems in the experimental procedures, a limit on the size of programs studied, and substantial differences in performance among the 36 programmers involved in each suppressed relationships between the metrics and task performance. In fact it did not appear that the metrics were any better than the number of lines of code for predicting performance. However, in the third experiment²⁰ we used longer programs, increased the number of participants to 54, and eliminated earlier procedural problems. We found both the Halstead and McCabe metrics superior to lines of code for predicting the time to find and fix an error in the program.

In the final experiment⁷⁵, we asked nine programmers to each create three simple programs (e.g., find the maximum and minimum of a list of numbers) from a common specification of each program. The best predictor of the time required to develop and successfully run the program was Halstead's metric for program volume (Figure 5). This relationship was slightly stronger than that for McCabe's $v(G)$, while lines of code exhibited no relationship.

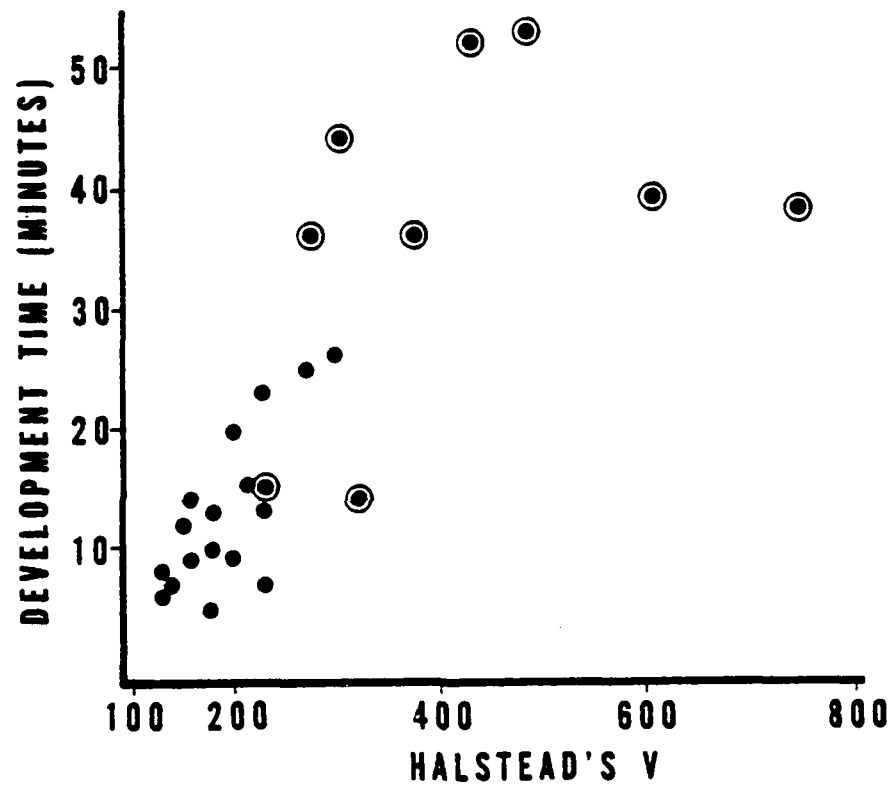


Figure 5. Scatterplot of Halstead's V against development time from Sheppard et al.

The datapoints circled in Figure 5 represent the data from a program whose specifications were less complete than those of the other two programs studied. The prediction of development time for this program was poor. We have observed in other studies that outcomes are more predictable on projects where a greater discipline regarding software standards and practices was observed^{58,59}. This experiment suggests that better prediction of outcomes may occur when more disciplined software development practices (e.g., more detailed program specifications) reduce the often dramatic performance differences among programmers.

In these experiments we found Halstead's and McCabe's metrics to be valid measures of psychological complexity, regardless of whether the program they were computed on was developed by the programmer under study or by someone else. We concluded that there is considerable promise in using complexity metrics to predict the difficulty programmers will experience in working with software. Similar conclusions have been reached by Baker and Zweben³ on an analytical rather than empirical evaluation of the Halstead and McCabe metrics.

Halstead's metrics have proven useful in actual practice. For instance, Elshoff²⁷ has used these metrics as feedback to programmers during development to indicate the complexity of their code. When metric values for their modules exceed a certain limit, programmers are instructed to consider ways of reducing module complexity. Bell and Sullivan⁸ suggest that a reasonable limit on the Halstead value for length is 260, since they found that published algorithms with values above this figure typically contained an error.

Regardless of the empirical support for many of Halstead's predictions, the theoretical basis for his metrics needs considerable attention. Halstead, more than other researchers, tried to integrate theory from both computer science and psychology. Unfortunately, some of the psychological assumptions underlying his work are difficult to justify for the phenomena to which he applied them. In general, computer scientists would do well to immediately purge from their memories:

- The magic number 7 ± 2
- The Stroud number of 18 mental discriminations per second.

These numbers describe cognitive processes related to the perception or retention of simple stimuli, rather than the complex information processing tasks involved in programming. Broadbent¹² argues that for complicated tasks (such as understanding a program) the magic number is substantially less than seven. These numbers have been incorrectly applied in too many explanations and are too frequently cited by people who have never read the original articles^{57,85}. Regardless of the validity of his assumptions, Halstead was a pioneer in attempting to develop interdisciplinary theory, and his efforts have provided considerable grist for further investigation.

Interconnectedness

Since the modularization of software has become an increasingly important concept in software engineering⁶⁸, several metrics have been developed to assess the complexity of the interconnectedness among the parts comprising a software system^{7,56,63,65,95}. For instance, Myers⁶³ models system complexity by developing a dependency matrix among pairs of modules based on whether there is an interface between them. Although his measure does not appear to have received much empirical attention, it does present two important considerations for modeling complexity at the system level⁶⁵. The first consideration is the strength of a module; the nature of the relationships among the elements within a module. The stronger, more tightly bound a module, the more singular the purpose served by the processes performed within it. The second consideration is the coupling between modules; the relationship created between modules by the nature of the data and control that is passed between them.

A primary principle of modular design is to achieve as much independence among modules as possible. This independence helps to localize the impact of errors or modifications to within one or a few modules. Thus, the complexity of the interface between modules may prove to be an excellent predictor of the difficulty experienced in developing and maintaining large

systems. Myers' measure identifies data flow as a critical factor in maintainability. Nevertheless, his measure has not been completely operationally defined, and its current value is primarily heuristic. Yau and his associates⁹⁵ are currently working on validating a model of this generic type. Unfortunately, little empirical evidence is available to assess the predictive validity of such metrics.

The focus of metrics measuring the interconnectedness among parts of a system is quite different from those which measure elementary program constructs or control flow. Metrics measuring the latter phenomena take a micro-view of the program, while interconnectedness metrics speak to a macro-level. An improved understanding of aggregating from the micro- to the macro-level needs to be achieved. For instance, summing the Halstead measures across modules leads to very different results than computing them once over the entire program⁵⁹.

Interconnectedness metrics may prove more appropriate parameters for macro-level models such as those which predict maintenance costs and resources. Macro-level metrics may prove better because factors to which micro-level metrics are more sensitive, such as individual differences among programmers, are balanced out at the macro- or project level. Macro-level metrics are less perturbed by these factors, increasing their benefit to an overall understanding of system complexity and its impact on system costs and performance.

Although we can quantify a software characteristic and demonstrate that it correlates with some criterion, we have not demonstrated that it is a causal factor influencing that criterion. An argument for causality requires the support of rigorous experimentation. The experimental evaluation of software characteristics is a small but growing research area.

EXPERIMENTAL EVALUATION OF SOFTWARE CHARACTERISTICS

Cause-Effect Relationships

Most of the studies reported in the previous section do not demonstrate cause-effect relationships between software characteristics and programmer performance. That is, there were a number of uncontrolled factors in the data collection environment which could have influenced the observed data. These alternate explanations dilute any statement of cause and effect. Although structural equation techniques^{25,44} allow an investigation of whether the data are consistent with one or more theoretical models, a causal test of theory will require a rigorously controlled experiment. According to Cattell¹⁶:

An experiment is a recording of observations . . . made by defined and recorded operations and in defined conditions followed by examination of the data . . . for the existence of significant relations. (p. 20)

Two important characteristics of an experiment are that its data collection procedures are repeatable and that each experimental event result in only one from among a defined set of possible outcomes⁴³. An experiment does not prove an hypothesis. It does, however, allow for the rejection of competing alternative explanations of a phenomenon.

The confidence which can be placed in a cause-effect statement is determined by the control over extraneous variables exercised in the collection of data. For instance, Milliman and I⁵⁸ reported a field study in which a software development project guided by modern programming practices produced higher quality code with less effort and experienced fewer system test errors when compared to a sister project developing a similar system in the same environment which did not observe these practices. Although many of the environmental factors were controlled, an

alternate explanation of the results was that the project guided by modern practices was performed by a programming team with more capable personnel.

An important characteristic of behavioral science experiments is the random assignment of participants to conditions²⁹. By removing any systematic variation in the ability, motivation, etc. of participants across experimental conditions, this method supposedly eliminates the hypothesis that experimental effects are due to individual differences among participants. Assigning a morning class to one condition and an afternoon class to another condition does not constitute random assignment, since students rarely choose class times on a random basis. However, if classes are the unit of study, the problem can be solved by randomly assigning a number of classes to each experimental condition. Random assignment has been a problem in testing causal relationships in field studies on actual software development projects.

There is often a conflict between what Campbell and Stanley¹⁵ describe as the internal and external validity of an experiment. Internal validity concerns the rigor with which experimental controls are able to eliminate alternate explanations of the data. External validity concerns the degree to which the experimental situation resembles typical conditions surrounding the phenomena under study. Thus, internal validity expresses the degree of faith in causal explanations, while external validity describes the generalizability of the results to actual situations.

In software engineering research, rigorous experimental controls are difficult to achieve on software projects and laboratory studies often seem contrived. External validity is probably a greater problem in studying process factors such as the organization of programming teams than in studying software characteristics. That is, the environmental conditions surrounding software development which are difficult to replicate in the laboratory would probably have a greater effect on the functioning of programming teams than on a programmer's comprehension of code.

Reviews of the experimental research in software engineering have been compiled by Atwood, Ramsey, Hooper, and Kullas² and Shneiderman⁷⁷. Topics which have been submitted to experimental evaluation include batch versus interactive programming, programming style factors (e.g., indented listings, mnemonic variable names, and commenting), control structures, documentation formats, code review techniques, and programmer team organization. In discussing experimental methods, I will focus on the evaluation of conditional statements and control flow. These topics were not chosen because they were believed to be more important than other subjects. Rather, they were chosen because several programs of research have developed around them and because the conditional statement has been a focus of argument since it was originally assailed by Dijkstra²³ in 1968. Control statements have been a concern of the structured programming movement, and the results reported here evaluate their most effective implementation.

The usability of control statements is important since they account for a large proportion of the errors made by programmers^{59,96}. Control structures are closely related to some of the metrics discussed in a previous section, such as McCabe's cyclomatic number. Research will be described here in a tutorial fashion to demonstrate the depth to which experimental programs can investigate a problem.

Conditional Statements

Sime, Green, and their colleagues at Sheffield University have been studying the difficulty people experience in working with conditional statements. In their first experiment Sime, Green, and Guest⁸¹ compared the ability of non-programmers to develop a simple algorithm with either nested or branch-to-label conditionals. Nesting implies the embedding of a conditional statement within one of the branches of another conditional. Nested structures are designed to make this embedding more visible and comprehensible to a programmer. Branch-to-label structures obscure the visibility of embedded conditions, since the "true" branch of a conditional statement sends the control elsewhere in the program to a statement with a specified label.

The conditional for the nested language was an IF-THEN-OTHERWISE construct similar to conditionals used in Algol, PL/I, and Pascal. This conditional construct is written:

```
IF [condition] THEN [process 1]
    OTHERWISE [process 2]
```

The branch-to-label conditional was the IF-GOTO construct of Fortran and Basic which Dijkstra²³ considered harmful. This conditional is written:

```
IF [condition] GOTO L1
    [process 2]
L1 [process 1]
```

In both examples, if the condition is true, process 1 is executed; if it is not true, process 2 is executed. A "condition" might be an expression such as " $X > 0$ ", while a process might be one or more statements such as " $X = X + 1$ ". Participants used one of these micro-languages to build an algorithm which organized a set of cooking instructions depending on the attributes of the vegetable to be cooked.

Sime et al. found that participants using the GOTO construct finished fewer problems, took longer to complete them, and made more semantic (e.g., logic) errors in building their algorithms than participants using the IF-THEN-OTHERWISE construct. They concluded that the GOTO construct placed a greater cognitive load on programmers by requiring that they both set and remember the label to which a conditional statement might branch. Further, programmers had to remember the various conditions which could branch to a particular label. These conditions are potentially more numerous for a GOTO than for an OTHERWISE statement.

In a further study Sime, Green, and Guest⁸² questioned whether the superiority of nested over branch-to-label construct would be maintained when multiple processes were performed under a single branch of a conditional. For instance, given a statement:

IF condition THEN process 1 AND process 2,

there are two ways a programmer may interpret its execution:

- 1) (IF condition THEN process 1) AND process 2, or
- 2) IF condition THEN (process 1 AND process 2).

In the first interpretation, process 2 is performed regardless of the state of the condition, while in the second it is performed only if the condition is true. Sime et al. believed it would be difficult for a programmer to retain the scope of the processes to be performed within each branch of a conditional statement. This difficulty would be especially acute when conditionals were nested within each other.

Sime et al.' second experiment investigated different techniques for marking the scope of the processes subsumed under each branch of a conditional statement. In addition to the IF-GOTO conditional, they defined a nested BEGIN-END and a nested IF-NOT-END representing two different structures for marking the scope of each branch in nested conditionals (Table 2). The BEGIN and END statements mark the scope of processes performed under one branch of a conditional statement, while the IF-NOT-END uses a more redundant scope marker by repeating the condition whose truth is being tested. In a strict sense, IF-NOT-END is the counterpart of the IF-THEN-ELSE construct, while the BEGIN-END markers could be used under either construct.

As in the previous experiment, non-programmers were asked to develop an algorithm for each of five problems within 2 hours total. Sime et al. found that more semantic (algorithmic) errors occurred in the IF-GOTO language, while errors in the nesting languages were primarily syntactic (grammatical). The BEGIN-END construct produced more syntactic errors and only half as many successful first runs as the other constructs. Errors were debugged ten times faster in the IF-NOT-END condition, which proved to be the most error free construct.

TABLE 2
CONDITIONAL STRUCTURES INVESTIGATED BY SIME, GREEN, AND GUEST

IF-GOTO	NESTED BEGIN-END	NESTED IF-NOT-END
IF (Condition 1) GOTO L1 IF (Condition 3) GOTO L2 (Process 4) STOP L1 IF (Condition 2) GOTO L3 (Process 2) STOP L2 (Process 3) STOP L3 (Process 1) STOP	IF (Condition 1) THEN BEGIN IF (Condition 2) THEN BEGIN (Process 1) END ELSE BEGIN (Process 2) END END ELSE BEGIN (Process 3) END END IF (Condition 3) THEN BEGIN (Process 3) END ELSE BEGIN (Process 4) END END	IF (Condition 1) IF (Condition 2) (Process 1) NOT (Condition 2) (Process 2) END (Condition 2) NOT (Condition 1) IF (Condition 3) (Process 3) NOT (Condition 3) (Process 4) END (Condition 3) END (Condition 1)

Results for semantic errors suggest that it is easier to keep track of the control flow in a nested language. However, when multiple processes are performed the use of scope markers often result in careless syntactic errors. Syntactic errors are more likely to occur with the BEGIN-END than in the IF-NOT-END constructs, because the redundancy of conditional expressions in the latter make the placement of markers more obvious.

Differential results for the two types of errors offer some validity for the syntactic/semantic model of programmer behavior developed by Shneiderman and Mayer⁷⁸. This model distinguishes between information which is language specific (syntactic) and language independent (semantic). The correct design of the algorithm is a semantic issue, while the grammatically accurate expression of that algorithm in a language is a syntactic issue. The structure of a language may simplify the design of an algorithm, but make its expression more difficult. Obviously, a language design should seek to simplify both the design and expression of an algorithm.

Based on the results of this second experiment, Sime, Green, and Guest⁸² proposed that their memory load explanation be replaced with an explanation that information is easier to extract from some languages than others. They distinguished two types of information: sequence and taxon. Sequence information involves establishing or tracing the flow of control forward through a program. Taxon information involves the hierarchical arrangement of conditions and processes within a program. Such information is important when tracing backward through a program to determine what conditions must be satisfied for a process to be executed. Sime et al. hypothesized that sequence information is more easily obtained from a nested language, while taxon information is more easily extracted from a nested language which also contains the redundant conditional expressions. Conceptually, taxon information can be determined without fully understanding the sequence of processes within a program, since not all branches have to be examined in a backwards tracing.

In two subsequent studies Green³⁷ validated these hypotheses in research with professional programmers. Sequence information was more easily determined from nested languages, although no differences were

observed between the BEGIN-END and IF-NOT-END constructs. Backwards tracing was performed much more easily with the IF-NOT-END construct. Green estimated that the time required for backwards tracing was 15% less in IF-NOT-END than in IF-GOTO conditionals.

It is important to recognize that program comprehension is not a uni-dimensional cognitive process. Rather, different types of human information processing are required by different types of software tasks. Green demonstrated that certain constructs were more helpful for performing certain software tasks. Software engineering techniques may differ in the benefits they offer to different programming tasks, since they differ in the types of human information processing that they assist.

Since the IF-NOT-END construct is not implemented in existing languages, Sime, Arblaster, and Green⁷⁹ investigated ways to improve the use of the BEGIN-END conditional markers. They developed a tool which would automatically build the syntactic portions of a conditional statement once the user chose the expression to be tested. In a second experimental condition, they developed an explicit writing procedure for helping participants develop the syntactic elements of a conditional statement. This procedure involved writing the syntax of the outermost conditional first, and then writing the syntax of conditionals nested within it. In the final condition participants were left to their own ways of using the conditional constructs.

Sime et al. found that participants solved more problems correctly on their first attempt using the automated tool, but that a writing procedure was almost as effective. The writing procedure reduced the number of syntactic errors, which had been the major problem with the BEGIN-END construct in earlier studies. Syntactic errors were not possible with the automated tool. The writing procedure and automated tools helped participants dispense with syntactic considerations quickly, so that they could spend more time concentrating on the semantic portion of the program (i.e., the function which was to be performed). However, once an error was made, it was equally difficult to correct regardless of the condition.

Thus, writing procedures and aids primarily increase the accuracy of the initial implementation.

Arblaster, Sime, and Green¹ reported on two further experiments which extended the results for writing rules to conditionals using GOTOs. Two groups of non-programmers, one of which had been taught the writing rules, constructed simple algorithms using the IF-GOTO conditional. The group trained in the writing procedures made fewer errors, semantic and syntactic, replicating the results obtained with the nested languages.

Arblaster et al. repeated this experiment using more sophisticated participants. They added two additional experimental conditions to those described above, both using an IF-IFNOT conditional structure. This conditional was written:

```
      IF [condition 1] GOTO L1 IFNOT GOTO L2
L1   [process 1]
L2   [process 2]
```

One of the two groups using this conditional was also trained in the use of writing rules. Use of the IF-IFNOT procedure resulted in fewer syntactic errors than observed with the IF-GOTO conditional. No differences were found for semantic errors.

Arblaster et al. pointed out that an early version of Fortran had a conditional format consistent with those found to be most effective in their experiments. However, this format was lost when further revisions of the language opted for an arithmetic IF conditional.

Shneiderman⁷⁶ compared the use of the arithmetic IF:

```
      IF [arithmetic condition] L1, L2, L3
L1   [process 1]
L2   [process 2]
L3   [process 3]
```

to the use of the logical IF:

```
      IF [boolean condition] GOTO L1
      [process 2]
L1   [process 1]
```

in Fortran. The arithmetic IF creates the possibility of a conditional with three branches (i.e., tests for >, =, and <). He found that novice programmers had more difficulty comprehending the arithmetic than the logical IF. However, no such differences were observed for more experienced programmers, who Shneiderman believed had adjusted to translating the more complex syntax of the arithmetic IF into their own semantic representation of the control logic.

In a recent experiment by Richards, Green, and Manton quoted by Green³⁸, the ordinary nesting of IF-THEN-ELSE conditionals was compared to the nesting of IF-NOT-END conditionals, and both were compared to a style of nesting in which an IF never directly follows a THEN. This last conditional would be written:

```
      IF [condition 1] THEN [process 1]
      ELSE IF [condition 2] THEN [process 2]
      ELSE IF [condition 3] THEN [process 3]
      etc.
```

and has the appearance of a CASE statement. Although this arrangement is contrary to the tenets of structured coding²⁴, some have argued that it may be easier for programmers to understand⁸⁷.

Green reports that for different forms of comprehension questions, this IF-THEN-ELSE-IF conditional was "never much better and sometimes much worse" than the other forms of nested conditionals. He argues that it is important to design computer languages so that perceptual cues such as indenting the levels of nesting can visually display the structure of the code. These perceptual cues can relieve the programmer of searching through the program text, a task which is distracting, time-consuming, and error-prone.

The extensive program of research by Sime, Green, and their colleagues has answered important questions about the structure of conditional statements. Their conclusions should be heeded in the design of future languages. They demonstrated:

- the superiority of nested over branch-to-label conditionals,
- the advantage of redundant expression of controlling conditions at the entrance to each conditional branch,
- that the benefits of a software practice may vary with the nature of the task, and
- that a standard procedure for generating the syntax of a conditional statement can improve coding speed and accuracy.

Overall, these results indicate that the more visible and predictable the control flow of a program, the easier it is to work with.

Sime, Green, and their associates have demonstrated how a preference among conditional constructs can be reduced to an empirical question which provides alternatives that were previously unconsidered (e.g., the IF-NOT-END construct). Their research has investigated structured coding at the construct level⁸⁰. The next step is to evaluate the reputed benefits of structured coding at the modular level, considering the various structured constructs in total.

Control Flow

Structured programming has become a catch-all term for programming practices related to system design, programming team organization, code reviews, configuration management, rules for program control flow, and myriad other procedures for software development and maintenance. This section will review only experiments related to structured coding: the enforcement of rules for program control flow. The control structures generally allowed under structured coding are displayed in Figure 6.

To evaluate reputed problems with the GOTO statement, Lucas and Kaplan⁵² instructed 32 students to develop a file update program in PL/C,

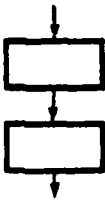
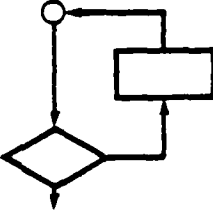
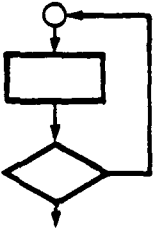
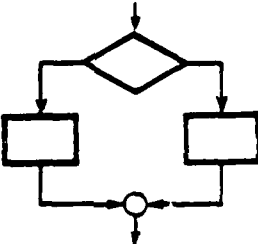
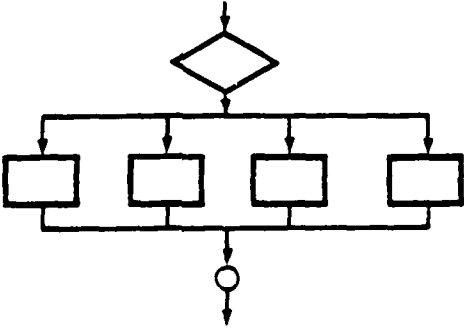
SEQUENCE		BEGIN process 1 process 2 END
REPETITION		WHILE condition DO process ENDDO
		REPEAT process UNTIL condition
SELECTION		IF condition THEN process 1 ELSE process 2 ENDIF
		CASE condition OF CASE 1: process 1 CASE 2: process 2 CASE n: process n ENDCASE

Figure 6. Constructs in structured control flow.

and half were further instructed to avoid the use of GOTOs. However, programmers in the GOTO-less condition were not trained in using alternate conditional constructs. Not surprisingly then, the GOTO-less group required more runs to develop their programs. In a subsequent task, all participants were required to make a specified modification to a structured program. Contrary to results on the earlier task, the group which had earlier struggled to write GOTO-less code made quicker modifications which required less compile time and storage space.

Weissman⁹² also investigated the comprehension of PL/I programs written in versions whose control flow was either 1) structured, 2) unstructured but simple, or 3) unstructured and complex. Participants were given comprehension quizzes and required to make modifications to the programs. Higher performance scores were typically obtained on the structured rather than unstructured versions, and participants reported feeling more comfortable with structured code. Love⁵⁰ subsequently found that graduate students could comprehend programs with a simplified control flow more easily than programs with a more complex control flow.

Recently, a series of experiments evaluating the benefits of structured code for professional programmers was conducted in our research unit by Sheppard, Curtis, Milliman, and Love⁷⁴. In the first experiment participants were asked to study a modular-sized Fortran program for 20 minutes, and then reconstruct it from memory. Three versions of control flow performing identical functions were defined for each of nine programs. One version was structured to be consistent with the principles of structured coding described by Dijkstra by allowing only three basic control constructs: linear sequence, structured selection, and structured iteration. Because structured constructs are sometimes awkward to implement in Fortran IV⁸⁶, a more naturally structured control flow was constructed which allowed limited deviations from strict structuring: multiple returns, judicious backward GOTOs, and forward mid-loop exits from a DO. Finally, a deliberately convoluted version was developed which included constructs that had not been permitted in the structured or naturally structured versions, such as backward exits from DO loops, arithmetic IFs, and unrestricted use of GOTOs.

As expected, control flow did affect performance. The convoluted control flow was the most difficult to comprehend (Figure 7). The difference in the average percent of reconstructed statements for naturally structured and convoluted control flows was statistically significant. Contrary to expectations, however, the strictly structured version did not produce the best performance. A slightly (although not significantly) greater percent of statements were recalled from naturally structured than from strictly structured programs.

In a second experiment Sheppard et al. instructed programmers to make specified modifications to three programs, each of which was written in the three versions of control flow described previously. A significantly higher percent of steps required to complete the modification was correctly implemented in the structured programs when compared to convoluted ones (Figure 7). There were no differences in the times required. No statistically significant differences appeared between the two versions of structured control flow, although performance was slightly better on strictly rather than naturally structured code. These results suggested that the presence of a consistent structured discipline in the code, either strict or natural, was beneficial and minor deviations from strict structuring did not adversely affect performance.

In a third experiment Sheppard et al. decided to compare the two versions of structured Fortran IV to Fortran 77, which contains the IF-THEN-ELSE, DO-WHILE, and DO-UNTIL constructs. They measured how long a programmer took to find a simple error embedded in a program. No differences were attributable to the type of structured control flow, replicating similar results in the first two experiments. The advantage of structured coding appears to reside in the ability of the programmer to develop expectations about the flow of control - expectations which are not seriously violated by minor deviations from strict structuring.

The research reviewed here indicates that programs in which some form of structured coding is enforced will be easier to comprehend and modify than programs in which such coding discipline is not enforced. It is not clear that structured coding will improve the productivity of programmers

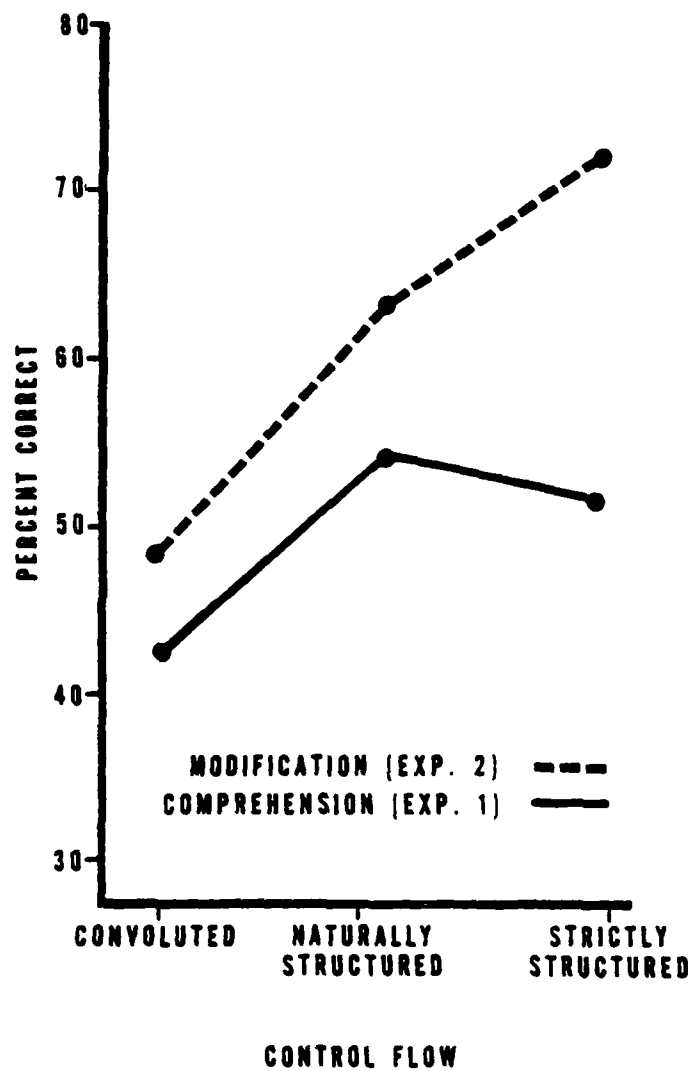


Figure 7. Performance by type of control flow from Sheppard et al.

during implementation. Some productivity improvements may be observed if less severe, more easily corrected errors are made using structured constructs, as suggested in the data of Sime and his colleagues. However, structured coding should reduce the costs of maintenance since such programs are less psychologically complex to work with. Experiments such as these can provide valuable guidance for decisions about an optimal mix of software standards and practices.

Problems in Experimental Research

It is important to recognize the benefits and limitations of controlled laboratory research. On the positive side, rigorous controls allow experimenters to isolate the effects of experimentally manipulated factors and identify possible cause-effect relationships in the data. On the other hand, the limitations of controlled research restrict the generalizations which can be made from the data. Laboratory research has an air of artificiality, regardless of how realistic researchers make the tasks.

Several problems attendant to most current empirical validation studies severely limit the generalizability of conclusions which can be drawn from them¹³. For instance, program sizes have frequently been restricted because of limitations in the research situation. This problem is characteristic of experimental research where time limitations do not allow participants to perform experimental tasks such as the coding or design of large systems. Also, since new factors come into play in the development of large systems (e.g. team interactions), the magnitude of a technique's effect on project performance may differ markedly from its effect in the laboratory.

The nature of the applications studied are often limited by the environments from which the programs are drawn (e.g., military systems, commercial systems, real-time, non-real-time, etc.). Further, there is frequently little assessment of whether results will hold up across programming languages. It is extremely difficult to perform evaluative research over a broad range of applications, especially when experimental

procedures are used. Thus, empirical results should be replicated over a series of studies on different types of programs in languages other than Fortran.

Another problem arises with what Sackman, Erickson, and Grant⁷² observed to be 25 or 30 to 1 differences in performance among programmers. This dramatic variation in performance scores can easily disguise relationships between software characteristics and associated criteria. That is, differences in the time or accuracy of performing some software task can often be attributed more easily to differences among programmers than to differences in software characteristics. Careful attention to experimental design is required to control this problem.

If generalizations are to be made about the performance of professional programmers, this is the population that should be studied rather than novices. As is true in most fields, there are qualitative differences in the problem-solving processes of experts and novices⁸³. However, the advantage of some techniques is the ease with which they are learned, and novices are the appropriate population for studying such benefits. Attempts to generalize experimental results must also be tempered by an understanding of how real-world factors affect outcomes. Data should be collected in actual programming environments to both validate conclusions drawn from the laboratory and determine the influence of real-world factors.

SUMMARY

Software wizardry becomes an engineering discipline when scientific methods are applied to its development. The first step in applying these methods is modeling the important constructs and processes. When these constructs have been identified, the second step is to develop measurement techniques so that the language of mathematics can describe relationships among them. The testing of cause-effect relationships in a theoretical model requires the performance of critical experiments to eliminate alternative explanations of the phenomena. Even when possessed of supportive experimental evidence, our sermonizing should be cautious until we have established limits for the generalizability of our data.

There are four major points I have stressed, some by implication, in this review. First, measurement and experimentation are complementary processes. The results of an experiment can be no more valid than the measurement of the constructs investigated. The development of sound measurement techniques is a prerequisite of good experimentation. Many studies have elaborately defined the independent variables (e.g., the software practice to be varied) and hastily employed a handy but poorly developed dependent measure (criterion). Results from such experiments, whether significant or not, are difficult to explain.

Second, results are far more impressive when they emerge from a program of research rather than from one-shot studies. Programs of research benefit from several advantages, one of the most important being the opportunity to replicate findings. When a basic finding (e.g., the benefit of structured coding) can be replicated over several different tasks (comprehension, modification, etc.) it becomes much more convincing. A series of studies also result in deeper explication of both the important factors governing a process and the limits of their effects. For instance, Sime, Green, and their colleagues identified the benefits and limitations of nested conditionals in an extensive program of research. Performing a series of studies also affords an opportunity to improve measurement and experimental

methods. Thus, the reliability and validity of results can be improved in succeeding studies.

Third, the rigors of measurement and experimentation require serious consideration of processes underlying software phenomena. Definitions should not be based on popular consensus. As energy is invested in defining constructs, a clearer picture of the process often emerges. Factors not thought to be a part of the process may present themselves as direct effects or limiting conditions. Alternate approaches to the techniques will also emerge, as in Sime et al's development of the IF-NOT-END nested conditional. The frustrations of scientific investigation are often the mothers of invention. Improved measurement techniques also provide better tools for management information systems. More reliable and valid measurement provides greater visibility and insight into project progress and product quality.

Finally, there is no substitute for sound experimental evidence in arguing the benefits of a particular software engineering practice or in comparing the relative merits of several practices. Managers often vacillate between their desire for proof and their impatience with the scientific approach. However, with understanding comes the possibility of greater control over outcomes, especially if causal factors and their limitations have been identified. The merchants of software elixirs can always quote case studies. Yet, such experiential evidence is no replacement for experimentation, and is frequently no better than a manager's intuition.

Measurement and experimentation are not intellectual diversions. They are the scientific foundations from which engineering disciplines continue to be built. Scientists must be sensitive to the most important questions they should tackle in software engineering, and should constantly reassess research priorities to keep pace with the state-of-the-art. Software professionals need to encourage the scientific investigation of their business if real improvements are to be made in software productivity and quality. The scientific study of software engineering is young, and its rate of progress will improve as measurement techniques and experimental methods mature.

ACKNOWLEDGEMENTS

I would like to thank Laszlo Belady and Sylvia Sheppard, and Drs. Elizabeth Kruesi and John O'Hare for their thoughts and comments. I would also like to thank Pat Belback, Noel Albert, Kathy Olmstead, and Lisa Grey for manuscript preparation and Lou Oliver for his support and encouragement. Work resulting in this paper was supported by the Office of Naval Research, Engineering Psychology Programs (Contract #N00014-79-C-0595). However, the opinions expressed in this paper are not necessarily those of the Department of the Navy. Reprints can be obtained from Dr. Bill Curtis; General Electric Company, Suite 200; 1755 Jefferson Davis Highway; Arlington, VA 22202.

REFERENCES

1. Arblaster, A.T., Sime, M.E., & Green, T.R.G. Jumping to some purpose. The Computer Journal, 1979, 22, 105-109.
2. Atwood, M.E., Ramsey, H.R., Hooper, J.N., & Kullas, D.A. Annotated Bibliography on Human Factors in Software Development (Tech. Report TR-P-79-1). Alexandria, VA: Army Research Institute, 1979. (NTIS No. AD A071 113).
3. Baker, A.L. & Zweben, S.H. A comparison of measures of control flow complexity. In Proceedings of COMPSAC '79. New York: IEEE, 1979, 695-701.
4. Basili, V.R. & Reiter, R.W. Evaluating automatable measures of software development. In Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE, 1980, 107-116.
5. Belady, L.A. Software complexity. In Software Phenomenology. Atlanta: AIRMICS, 1977, 371-383.
6. Belady, L.A. Complexity of programming: A brief summary. In Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost. New York: IEEE, 1980, 90-94.
7. Belady, L.A. & Lehman, M.M. A model of large program development. IBM Systems Journal, 1976, 15(3), 225-252.
8. Bell, D.E. & Sullivan, J.E. Further Investigations into the Complexity of Software (Tech. Rep. MTR-2874). Bedford, MA: MITRE, 1974.
9. Benyon-Tinker, G. Complexity measures in an evolving large system. In Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE, 1980, 117-127.
10. Boehm, B.W. Software and its impact: A quantitative assessment. Datamation, 1973, 19(5), 48-59.
11. Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., MacLeod, G.J., Merrit, M.J. Characteristics of Software Quality. Amsterdam: North Holland, 1978.
12. Broadbent, D.E. The magic number seven after fifteen years. In A. Kennedy & A. Wilkes (Eds.). Studies in Long Term Memory. New York: Wiley, 1975, 3-18.

13. Brooks, R. Studying programmer behavior experimentally: The problems of proper methodology. Communications of the ACM, 1980, 23, 207-213.
14. Bulut, N. & Halstead, M.H. Impurities found in algorithm implementation. SIGPLAN Notices, 1974, 9(3), 9-10.
15. Campbell, D. & Stanley, J.C. Experimental and Quasi-Experimental Designs for Research. Chicago: Rand-McNally, 1966.
16. Cattell, R.B. The principles of experimental design and analysis in relation to theory building. In R.B. Cattell (Ed.), Handbook of Multivariate Experimental Psychology. Chicago: Rand-McNally, 1966, 19-66.
17. Chen, E.T. Program complexity and programmer productivity. IEEE Transactions on Software Engineering, 1978, 3, 187-194.
18. Cornell, L.M. & Halstead, M.H. Predicting the Number of Bugs Expected in a Program Module (Tech. Rep. CSD-TR-205). West Lafayette, IN: Purdue University, Computer Science Department, 1976.
19. Curtis, B. In search of software complexity. In Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE, 1980, 95-106.
20. Curtis, B., Sheppard, S.B., & Milliman, P. Third time charm: Stronger prediction of programmer performance by software complexity metrics. In Proceedings of the Fourth International Conference on Software Engineering. New York: IEEE, 1979, 356-360.
21. Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., & Love, T. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. IEEE Transactions on Software Engineering, 1979, 5, 96-104.
22. Data Analysis Center for Software. Quantitative Software Models (SRR-1). Griffiss AFB, NY: Rome Air Development Center, 1979.
23. Dijkstra, E.W. GO TO statement considered harmful. Communications of the ACM, 1968, 11, 147-148.
24. Dijkstra, E.W. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, & C.A.R. Hoare (Eds.) Structured Programming. New York: Academic, 1972, 1-82.
25. Duncan, O.D. Introduction to Structural Equation Models. New York: Academic, 1975.

26. Elshoff, J.L. Measuring commercial PL/1 programs using Halstead's criteria. SIGPLAN Notices, 1976, 11, 38-46.
27. Elshoff, J.L. A review of software measurement studies at General Motors Research Laboratories. In Proceedings of the Second Software Life Cycle Management Workshop. New York: IEEE, 1978, 166-171.
28. Feuer, A.R. & Fowlkes, E.G. Some results from an empirical study of computer software. In Proceedings of the Fourth International Conference on Software Engineering. New York: IEEE, 1979, 351-355.
29. Fisher, R.A. The Design of Experiments. London: Oliver & Boyd, 1935.
30. Fitzsimmons, A.B. Relating the presence of software errors to the theory of software science. In A.E. Wasserman and R.H. Sprague (Eds.), Proceedings of the Eleventh Hawaii International Conference on Systems Sciences. Western Periodicals, 1978, 40-46.
31. Fitzsimmons, A.B. & Love, L.T. A review and evaluation of software science. ACM Computing Surveys, 1978, 10, 3-18.
32. Freburger, K. & Basili, V.R. The Software Engineering Laboratory: Relationship Equations (Tech. Rep. TR-764). College Park, MD: University of Maryland, Computer Science Department, 1979.
33. Freiman, F.R. & Park, R.E. PRICE software model - Version 3 an overview. In Workshop on Quantitative Software Models for Reliability, Complexity and Cost. New York: IEEE, 1980, 32-41.
34. Funami, Y. & Halstead, M.H. A software physics analysis of Akiyama's debugging data. In Proceedings of the MRI 24th International Symposium: Software Engineering. New York: Polytechnic Press, 1976, 133-138.
35. Gilb, T. Software Metrics. Cambridge, MA: Winthrop, 1977.
36. Green, T.F., Schneidewind, N.F., Howard, G.T., & Pariseau, R. Program structures, complexity and error characteristics. In Proceedings of the Symposium on Computer Software Engineering. New York: Polytechnic Press, 1976, 139-154.
37. Green, T.R.G. Conditional program statements and their comprehensibility to professional programmers. Journal of Occupational Psychology, 1977, 50, 93-109.
38. Green, T.R.G. Ifs and thens: Is nesting just for the birds. Software - Practice and Experience, 1980, 10, in press.

39. Gordon, R.D. & Halstead, M.H. An experiment comparing Fortran programming times with the software physics hypothesis. AFIPS Conference Proceedings, 1976, 45, 935-937.
40. Halstead, M.H. Natural laws controlling algorithm structure. SIGPLAN Notices, 1972, 7(2), 19-26.
41. Halstead, M.H. An experimental determination of the "purity" of a trivial algorithm. ACM SIGME Performance Evaluation Review, 1973, 2(1), 10-15.
42. Halstead, M.H. Elements of Software Science. New York: Elsevier North-Holland, 1977.
43. Hays, W.L. Statistics. New York: Holt, Rinehart, & Winston, 1973.
44. Heise, D.R. Causal Analysis. New York: Wiley, 1975.
45. Jones, T.C. Measuring programming quality and productivity. IBM Systems Journal, 1978, 17(1), 39-63.
46. Kelvin, W.T. Popular lectures and addresses, 1891-1894. Quoted by M.L. Shooman in the Introduction to Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE, 1980.
47. Lehman, M.M. Programs, cities, students - Limits to growth. In D. Gries (Ed.), Programming Methodology. New York: Springer-Verlag, 1978, 42-69.
48. Lehman, M.M. Programs, programming and the software life cycle. IEEE Proceedings, 1980, 68, xxx-xxx.
49. Lehman, M.M. & Parr, F.N. Program evolution and its impact on software engineering. In Proceedings of the Second International Conference on Software Engineering. New York: IEEE, 1976, 350-357.
50. Love, T. An experimental investigation of the effect of program structure on program understanding. SIGPLAN Notices, 1977, 12(3), 105-113.
51. Love, L.T. & Bowman, A. An independent test of the theory of software physics. SIGPLAN Notices, 1976, 11, 42-49.
52. Lucas, H.C. & Kaplan, R.B. A structured programming experiment. The Computer Journal, 1974, 19, 136-138.

53. Margenau, H. The Nature of Physical Reality. New York: McGraw-Hill, 1950.
54. McCabe, T.J. A complexity measure. IEEE Transactions on Software Engineering, 1976, 2, 308-320.
55. McCall, J.A., Richards, P.K., & Walters, G.F. Factors in Software Quality (Tech. Rep. 77CIS02). Sunnyvale, CA: General Electric, Command and Information Systems, 1977.
56. McClure, C.L. Reducing COBOL Complexity through Structured Programming. New York: Van Nostrand Reinhold, 1978.
57. Miller, G.A. The magic number seven, plus or minus two. Psychological Review, 1956, 63, 81-97.
58. Milliman, P. & Curtis, B. An evaluation of modern programming practices in an aerospace environment. In Proceedings of the Third Digital Avionics Systems Conference. New York: IEEE, 1979.
59. Milliman, P. & Curtis, B. A Matched Project Evaluation of Modern Programming Practices (RADC-TR-80-6, 2 vols.). Griffiss AFB, NY: Rome Air Development Center, 1980.
60. Mohanty, S.N. Models and measurements for quality assessment of software. ACM Computing Surveys, 1979, 11, 251-275.
61. Morrison, D.F. Multivariate Statistical Methods. New York: McGraw-Hill, 1967.
62. Musa, J. The measurement and management of software reliability. IEEE Proceedings, 1980, 68, xxx-xxx.
63. Myers, G.J. Software Reliability. New York: Wiley, 1976.
64. Myers, G.J. An extension to the cyclomatic measure of program complexity. SIGPLAN Notices, 1977, 12(10), 61-64.
65. Myers, G.J. Composite/Structured Design. New York: Van Nostrand Reinhold, 1978.
66. Nunnally, J.C. Psychometric Theory. New York: McGraw-Hill, 1967.
67. Ottenstein, L.M. Quantitative estimates of debugging requirements. IEEE Transactions on Software Engineering, 1979, 5, 504-514.
68. Parnas, D.L. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 1972, 15, 1053-1058.

69. Putnam, L.H. A general empirical solution to the macro software sizing and estimating problem. IEEE Transactions on Software Engineering, 1978, 4, 345-361.
70. Rabin, M.O. Complexity of computations. Communications of the ACM, 1977, 20, 625-633.
71. Richards, P. & Chang, P. Localization of Variables: A Measure of Complexity (Tech. Rep. 75CIS01). Sunnyvale, CA: General Electric, Command and Information Systems, 1975.
72. Sackman, H., Erickson, W.J., & Grant, E.E. Exploratory and experimental studies comparing on-line and off-line programming performance. Communications of the ACM, 1968, 11, 3-11.
73. Schneidewind, N. & Hoffman, H.M. An experiment in software error data collection and analysis. IEEE Transactions on Software Engineering, 1979, 5, 276-286.
74. Sheppard, S.B., Curtis, B., Milliman, P., & Love, T. Human factors experiments on modern coding practices. Computer, 1979, 12, 41-49.
75. Sheppard, S.B., Milliman, P. & Curtis, B. Experimental Evaluation of On-line Program Construction (Tech. Rep. TR-79-388100-6).
— Arlington, VA: General Electric, Information Systems Programs, 1979.
76. Shneiderman, B. Exploratory experiments in programmer behavior. International Journal of Computer and Information Sciences, 1976, 5, 123-143.
77. Shneiderman, B. Software Psychology: Human Factors in Computer and Information Systems. Cambridge, MA: Winthrop, 1980.
78. Shneiderman, B. & Mayer, B.R. Syntactic/semantic interaction in programmer behavior: A model and experimental results. International Journal of Computer and Information Sciences, 1979, 8, 219-238.
79. Sime, M.E., Arblaster, A.T., & Green, T.R.G. Reducing programming errors in nested conditionals by prescribing a writing procedure. International Journal of Man-Machine Studies, 1977, 9, 119-126.
80. Sime, M.E., Arblaster, A.T., & Green, T.R.G. Structuring the programmer's task. Journal of Occupational Psychology, 1977, 50, 205-216.

81. Sime, M.E., Green, T.R.G., & Guest, D.J. Psychological evaluation of two conditional constructions used in computer languages. International Journal of Man-Machine Studies, 1973, 5, 105-113.
82. Sime, M.E., Green, T.R.G., & Guest, D.J. Scope marking in computer conditionals - a psychological evaluation. International Journal of Man-Machine Studies, 1977, 9, 107-118.
83. Simon, H.A. Information processing models of cognition. In M.R. Rosenzweig & L.W. Porter (Eds.) Annual Review of Psychology (Vol. 30). Palo Alto, CA: Annual Reviews, 1979, 363-396.
84. Stevens, S.S. Measurement. In G.M. Maranell (Ed.), Scaling: A Sourcebook for Behavioral Scientists. Chicago: Aldine, 1974, 22-41.
85. Stroud, J.M. The fine structure of psychological time. New York Academy of Sciences Annals, 1967, 138(2), 623-631.
86. Tenny, T. Structured programming in Fortran. Datamation, 1974, 20 (7), 110-115.
87. Tracz, W.J. Computer programming and the human thought process. Software - Practice and Experience, 1979, 9, 127-137.
88. Torgerson, W.S. Theory and Methods of Scaling. New York: Wiley, 1958.
89. Walston, C.E. & Felix, C.P. A method of programming measurement and estimation, IBM Systems Journal, 1977, 16, 54-73.
90. Walters, G.F. Applications of metrics to a Software Quality Management (QM) program. In J.D. Cooper & M.J. Fisher (Eds.), Software Quality Management. New York: Petrocelli, 1979, 143-157.
91. Weinberg, G.M. The Psychology of Computer Programming. New York: Van Nostrand Reinhold, 1971.
92. Weissman, L.M. A Method for Studying the Psychological Complexity of Computer Programs (Tech. Rep. TR-CSRG-37). Toronto: University of Toronto, Department of Computer Science, 1974.
93. Wolverton, W.R. The cost of developing large scale software. IEEE Transactions on Computers, 1974, 23, 615-634.
94. Woodward, M.R., Hennell, M.A., & Hedley, D. A measure of control flow complexity in program text. IEEE Transactions on Software Engineering, 1979, 5, 45-50.

95. Yau, S.S. & Collofellow, J.S. Some stability measures for software maintenance. In Proceedings of COMPSAC '79. New York: IEEE, 1979, 674-679.
96. Youngs, E.A. Human errors in programming. International Journal of Man-Machine Studies, 1974, 6, 361-376.

OFFICE OF NAVAL RESEARCH

Code 455

TECHNICAL REPORTS DISTRIBUTION LIST

OSD

CDR Paul R. Catelier
Office of the Deputy Under Secretary
of Defense
OUSDRE (E&LS)
Pentagon, Room 3D129
Washington, D.C. 20301

CAPT John Duncan
Office of the Secretary of Defense
(C3I)
Pentagon, Room 3C200
Washington, D.C. 20301

Department of the Navy

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

Director
Analysis and Support Division
Code 230
Office of Naval Research
800 North Quincy Street
Arlington, Va 22217

Director
Naval Analysis Programs
Code 431
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

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

Department of the Navy

Special Assistant for Marine Corps
Matters
Code 100M
Office of Naval Research
800 North Quincy Street
Arlington, VA 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 Office
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, CA 96503

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

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

Scientific Advisor to DCNO (MPT)
OP 01T (Dr. Marshall)
Washington, D.C. 20370

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

Head
Aerospace Psychology Department
Code L5
Naval Aerospace Medical Research Lab
Pensacola, FL 32508

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

Navy Personnel Research and
Development Center
Code 305
San Diego, Ca 92152

Navy Personnel Research and
Development Center

Management Support Department
Code 210
San Diego, CA 92152

CDR P. M. Curran
Code 604
Human Factors Engineering Division
Naval Air Development Center
Warminster, PA 18974

Human Factors Engineering Branch
Code 1226
Pacific Missile Test Center
Point Mugu, Ca 93042

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

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

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

Dean of Research Administration
Naval Postgraduate School
Monterey, CA 93940

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

HQS, U.S. Marine Corps
Attn: CCA40 (Major Pennell)
Washington, D.C. 20380

Commanding Officer
MCTSSA
Marine Corps Base
Camp Pendleton, CA 92055

Chief, C³ Division
Development Center
MCDEC
Quantico, VA 22134

Mr. Arnold Rubinstein
Naval Material Command
NAVMAT 08D22
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 513
Washington, D.C. 20361

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

Naval Sea Systems Command
Personnel & Training Analyses Office
NAVSEA 074C1
Washington, D. C. 20362

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

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

L. Chmura
Naval Research Laboratory
Code 7503
Washington, D.C. 20375

Dr. N. Perrone
ONR
Code 474
800 North Quincy Street
Arlington, VA 22271

J. B. Blankenheim
NAVELEX
Code 47013
Washington, D.C. 20360

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

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

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

Mr. Merlin Malehorn
Office of the Chief of Naval
Operations (OP 102)
Washington, D.C. 20350

Department of the Army

Mr. J. Barber
HQS, Department of the Army
DAPE-MBR
Washington, D.C. 20310

Dr. Joseph Zeidner
Technical Director
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

ARI 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. Donald A. Topmiller
Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HGES
Wright-Patterson AFB, OH 45433

Air University Library
Maxwell Air Force Base, AL 36112

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

Foreign Addresses

North East London Polytechnic
The Charles Myers Library
Livingstone Road
Stratford
London E15 2LJ
England

Professor Dr. Carl Graf Hoyos
Institute for Psychology
Technical University
8000 Munich
Arcisstr 21
Federal Republic of Germany

Director, Human Factors Wing
Defense & Civil Institute of
Environmental Medicine
Post Office Box 2000
Downsview, Ontario M3M 3B9
Canada

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

Other Government Agencies

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

Dr. Craig Fields
Director, Cybernetics Technology
Office
Defense Advanced Research Projects
Agency
1400 Wilson Boulevard
Arlington, VA 22209

Other Organizations

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

Dr. J. Miller
National Oceanic and Atmospheric
Administration
11400 Rockville Pike
Rockville, MD 20852

Professor Douglas E. Hunter
Defense Intelligence School
Washington, D.C. 20374

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

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

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

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

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

Dr. Alphonse Chapanis
Department of Psychology
The Johns Hopkins University
Charles and 34th Streets
Baltimore, MD 21218