SOFTWARE COMPLEXITY MEASUREMENT

Inappropriate use of software complexity measures can have large, damaging effects by rewarding poor programming practices and demoralizing good programmers. Software complexity measures must be critically evaluated to determine the ways in which they can best be used.

JOSEPH K. KEARNEY, ROBERT L. SEDLMEYER, WILLIAM B. THOMPSON, MICHAEL A. GRAY, and MICHAEL A. ADLER

In recent years much attention has been directed toward reducing software cost. To this end, researchers have attempted to find relationships between the characteristics of programs and the difficulty of performing programming tasks. The objective has been to develop measures of *software complexity* that can be used for cost projection, manpower allocation, and program and programmer evaluation.

Despite the growing body of literature devoted to their development, analysis, and testing, software complexity measures have yet to gain wide acceptance. Early claims for the validity of the metrics have not been supported, and considerable criticism has been directed at the methodology of the experiments that support the measures. Nonetheless, new complexity measures continue to appear, and new support for old measures is earnestly sought.

Complexity measures offer great potential for containing the galloping cost of software development and maintenance. Successful software-complexity-measure development must be motivated by a theory of programming behavior. An integrated approach to metric development, testing, and use is essential as development should anticipate the demands of the measure's usage. Testing should be performed with specific applications in mind, and where possible the test environment should simulate the actual use.

Complexity measures have been developed without any particular use in mind. Lacking a theory of programming behavior, researchers have sought meaningful applications through experimentation. The results of these explorations are difficult to interpret and provide only weak support for the use of complexity measures. Until more comprehensive evidence is available, software complexity measures should be used very cautiously.

© 1986 ACM 0001-0782/86/1100-1044 75¢

WHAT IS COMPLEXITY?

The first problem encountered when attempting to understand program complexity is to define what it means for a program to be complex. Basili defines complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task [3]. If the interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation. If the interacting system is a programmer, then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing, or modifying the software. The term *software complexity* is often applied to the interaction between a program and a programmer working on some programming task.

Usually these measures are based on program code disregarding comments and stylistic attributes such as indentation and naming conventions. Measures typically depend on program size, control structure, or the nature of module interfaces. The most widely known measures are those devised by Halstead and his colleagues that are collectively known as *software science* [11]. The Halstead measures are functions of the number of operators and

operands in the program. The major components of software science are

η_1 the number of unique operators,
η_2 the number of unique operands,
N_1 the total number of operators,
N_2 the total number of operands.

Halstead defined the volume, V, of a program to be

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

and program difficulty, D, to be

$$D = \frac{\eta_1 \times N_2}{2\eta_2}.$$

Halstead derived a number of other measures. The most extensively studied of these is an estimate of the effort, E, required to implement a program:

$$E = D \times V$$

The *cyclomatic number*, developed by McCabe [14] has also received a great deal of attention. McCabe considers the program as a directed graph in which the edges are lines of control flow and the nodes are straight line segments of code. The cyclomatic number represents the number of linearly independent execution paths through the program. For a well-structured module, the cyclomatic number is simply one plus the number of branching statements in the module. The cyclomatic number is strongly related to the Halstead metrics [5, 12]. Both are computed by counting lexical entities, and it has been shown that the measures are highly correlated. These token-based methods are strongly related to the most simplistic complexity measure, a count of the number of lines in the program.

The Halstead and McCabe measures treat a program or procedure as a single body of code. Henry and Kafura present a measure that is sensitive to the structural decomposition of the program into procedures and functions [13]. Their measure depends on procedure size and the flow of information into procedures (the fan-in) and out of procedures (the fan-out). Henry and Kafura define the complexity of a procedure as

$$\text{length} \times (\text{fan-in} \times \text{fan-out})^2.$$

Many other complexity measures have been developed. Those presented here are only representative examples of the work in the field.

METRIC DEVELOPMENT: THE NEED FOR A THEORY OF PROGRAMMING

Software complexity measures attempt to relate the contribution of the program to the difficulty of performing programming tasks. One of the reasons that the development of software complexity measures is so difficult is that programming behaviors are poorly understood. A behavior must be understood before what makes it difficult can be determined. To clearly state *what* is to be measured, we need a theory of programming that includes a model of the program, the programmer, the programming environment, and the programming task.

Programming behaviors are very complex and can be influenced by the experience and ability of the programmer, the specific task, the underlying problem domain, and the programming environment, as well as by the properties of the program. Although the experimental results are incomplete, there is ample evidence to suggest that all of these factors can affect the difficulty of performing programming tasks. Most complexity measures have been designed without regard to the programmer, task, problem domain, or environment and ignore the stylistic characteristics of the program. This approach implicitly assumes that the properties of the program determine the ease of performing programming tasks independent of these other factors. This assumption is unsupported. The available results indicate that the contribution of the program, that is, program complexity, is highly dependent on nonprogram factors.

Programmers are required to perform a variety of programming tasks including design, coding, debugging, testing, modification, and documentation. It should not be expected that all tasks are equally difficult. Further, it is unlikely that the factors that facilitate or hinder coding will influence debugging or maintenance activities in similar ways. A study performed by Dunsmore and Gannon, for example, found that the use of global variables for data communication among modules decreased error occurrence during program construction [8]. However, formal parameters proved less error prone for modification tasks. Before it can be determined which communication strategy leads to less complex programs, the task under consideration must be specified.

Experience surely influences the difficulty of the task. Experiential factors include general knowledge of programming languages, techniques, and algorithms, and specific knowledge of one or more application areas. Programming skills developed through practice, such as interpreting system diagnostics, are also important. The factors that contribute to the difficulty of performing programming tasks may be very different for novices and experts. Soloway and Ehrlich present evidence that experienced programmers develop schemata for stereotypical action sequences in programs [17]. For example, the pro-

grammer may have schemata for a RUNNING TOTAL LOOP or for an ITEM SEARCH LOOP. These schemata provide the expert programmer a framework for understanding the program and set up expectations for the detailed structure and function of program fragments. Soloway and Ehrlich also suggest that expert programmers follow a set of rules for programming discourse—conventions such as choosing variable names to reflect their function. Although schema and discourse rules assist the expert in understanding typical programs, violations of discourse rules and abnormal program structures can be a source of confusion. A pair of experiments performed by Soloway and Ehrlich suggests that novices are less disturbed by unusual structures and usages than experts. They argue that novices have not developed a repertoire of programming rules and hence have fewer expectations about the program. Thus, novices are less confused by unconventional programming practices.

The large majority of software complexity measures have been developed with little regard for the programmer, the programming task, or the programming environment. It is unlikely that the advances in software complexity measurements will be made until programmers and the programming process are better understood. This must be achieved by studying programmers and the means they use to construct, understand, and modify programs.

METRIC DEVELOPMENT: ANTICIPATING THE USE

Advocates of software complexity metrics have suggested that these tools can be used to predict program length, program development time, number of bugs, the difficulty of understanding a program, and the future cost of program maintenance. Measures, thus far, have been designed without any particular use in mind. A developer attempts to quantify what he or she considers to be an important characteristic of a program and then studies programs with the hope of discovering a relationship between the measure and some independent measure of program quality.

Software metrics must be designed to meet the needs of the end user. The properties of a metric critically determine the ways in which it can be used. This dependency must be kept in mind by those who are developing metrics and those who use metrics.

For example, let us say that we are interested in program debugging. Suppose that we have studied programmers and formed a model of the debugging process. Our model of the task proposes that effective debugging must be preceded by an understanding of the program. So we would like to develop a measure of program comprehensibility. Our study of the understanding process suggests that the difficulty of understanding depends, in part, on certain structural properties of the program. Now, we have specified what we want to measure, but why do we want to measure it? Just exactly how do we intend to use this index? If our measure is to be used as a tool to judge the quality of the programs, we must operate under a very different set of constraints than if the measure is to be used to allocate manpower for maintenance. When a measure is used to *evaluate* a programmer's work, it had better be the case that reductions in the measure lead to improvements in the programs; otherwise the use of the measure can be counterproductive. Whether or not the users plan to use the measure to direct the programming process, one can be sure that, if contracts or salaries are at stake, programmers will soon be writing programs that will minimize the measure.

Some important applications of complexity metrics, such as manpower allocation, do not evaluate the quality of the program. For these applications, software metrics are used only as *descriptive* tools. The demands of the measure are much less stringent here. For the example, it is adequate to know that a measure is strongly associated with the difficulty of understanding. The metric may be useful whether or not techniques that cause variations in measure improve the program.

There is an important distinction to be made here between program properties that cause the program to be complex and program characteristics that are associated with complexity. Consider, for example, the number of program changes made during program development. It has been shown that the number of changes correlates well ($r \approx 0.8$) with errors [8]. It would be difficult to argue that the changes themselves cause the errors. More likely, the changes are indicative of the trouble that the programmer had constructing the program. Although program changes are associated with the occurrence of errors, we would not want to uniformly advocate that programmers should minimize the number of changes made during development. (Although it might be found that some practices, such as extending the design phase, lead to both fewer changes and better programs.) And, though using the number of program changes to predict future difficulties or direct debugging efforts might be considered, program changes should not be used for salary review.

THE PROPERTIES OF MEASURES

Several properties of measures determine the way in which the measure can be used.

Robustness

If software complexity measures are to be used to evaluate programs, then it is important to consider the measure's responsiveness to program modifications. Not only should the measure be shown to reliably predict the complexity of the software, but programming techniques that minimize the measure should be examined to assure that reductions in the measure consistently produce improvements in the program. In particular, it should not be possible to reduce the measure through incidental modifications of the program. Also, programming techniques that modify the program in a desirable way with respect to one property must not produce an undesirable change in another property as a side effect. A robust measure of software complexity is sensitive to the underlying complexity of the program and cannot be misdirected by incidental modifications to the program.

Several authors have examined the relationship between complexity measures and commonly accepted axioms for good programming [1, 2, 9, 10]. Their strategy has been to study how complexity measures are affected by following maxims of good programming style. Halstead's *E*, the cyclomatic number, and the number of lines have been examined for their responsiveness to modularization, the use of temporary variables, initialization procedures, and such. The results of these analyses do not provide strong support for these measures. For some classes of programs, some measures are reduced by some good programming practices.

Even if it could be shown that some favored programming technique consistently leads to reductions in a measure, this is not strong evidence for the measure's robustness. One problem with this approach is that there is only weak evidence that these rules actually lead to better programs [15]. More importantly, this approach is limited because there may also be techniques that reduce the measure and worsen the quality of the program. Likewise, just because a measure is not responsive to some improvements does not mean that the measure is not valid and robust—it is unreasonable to expect a metric to be responsive to all possible improvements. What must be demonstrated is that reductions in the measure, however they are achieved, will lead to an improvement in the program.

Normativeness

The interpretation of complexity measurements is facilitated if the metric provides a norm against which measurements can be compared. Without such a standard, it is meaningless to apply the metric to programs in isolation. To judge whether or not a program is overly complex, a norm that identifies some acceptable level of complexity must be specified.

Specificity

Software complexity analysis may provide an assessment tool that can be used during program development and testing. Designers and programmers could use the measure to find deficiencies in program construction. A complexity measure might also be used as a guide to testing and maintenance efforts. The degree to which a measure is able to perform these functions will depend on how well it specifies what is contributing to the complexity of a program.

Henry and Kafura, for example, argue that complexity usually arises from a small number of disproportionately complex procedures [13]. Their measure of information flow assigns a value of complexity to each procedure. The major contributors to complexity can be identified by comparing complexity measurements from the set of procedures. Often large software systems are layered into levels of abstraction. Henry and Kafura argue that such systems should distribute complexity evenly across levels. They suggest that, if the total complexity within a level is large compared to surrounding levels, then there is evidence that a level of abstraction is missing from the system design. As a demonstration of the usefulness of their measure, Henry and Kafura applied their measure to a portion of the UNIX® operating system. They discovered that their measure produced inordinately high values for one level of the system, due largely to the density of interconnections between procedures at this and adjacent levels. They plausibly argue that the system could be simplified by the introduction of another layer of abstraction.

Prescriptiveness

If software complexity measures are to prove useful in the containment of program complexity, then they must not only index the level of a program's complexity, but also should suggest methods to reduce the amount of complexity. A measure could prescribe techniques to avoid excess complexity as well as direct modification of overly complex programs already written.

Henry and Kafura's work illustrates how a measure can influence program design. Their measure heavily penalizes single procedures with a large number of interconnections. They show how this can encourage modularization and simple patterns

UNIX is a trademark of AT&T Bell Laboratories.

of module communication. Note, however, that this is not the only way to minimize their measure. The smallest value can be achieved by writing a program as a single module with no procedures and, hence, no flow of information between procedures.

Property Definition

The properties identified above are not rigorously defined, and it is sometimes difficult to tell whether or not a metric possesses one or another of these properties. Although the preceding list of properties may be flawed, it is essential that the designers and users of software complexity measures recognize that the properties of measures constrain their usefulness and applicability.

TESTING

Once a measure has been developed, it must be tested to be sure it actually measures what it purports to measure.

Experimental Design

Researchers attempting to validate measures of software complexity face a methodological morass. An enormous number of parameters may influence the outcome of an experiment. Subject selection, programming language, programming task, and the algorithms implemented can all profoundly affect both the nature of the results and the extent to which experimental observations will generalize to a larger set of programming environments. The problem is compounded by the uncertainty of how these parameters interact to determine programmer behavior. Worse yet, there are not even good methods to quantify parameters such as programmer ability and problem difficulty.

Once the programming environment is specified, the experimenter must devise a method to manipulate the independent variable—typically some program property. If research is conducted as a natural experiment (observing actual programs produced in a real work setting), then the problem is to find programs that differ only in the variables of interest. The difficulty of obtaining uncontrived programs that vary only in one or two dimensions should not be underestimated.

Next, a measurement technique for the dependent variable—programmer performance in some task or program quality—must be selected. A large number of human performance variables have been suggested including time to implement or modify a program, number of debugging runs, number of bugs detected, and level of understanding as measured by ability to recall, reconstruct, and answer questions about a program's function. Program quality has been assessed by run-time efficiency and number of errors. Detailed analysis of the problems associated with subject selection criteria, manipulation of program properties, and choice of performance measures can be found in [15].

The uncertainty of how complexity measures relate to one another and to performance measures has led to a style of experimentation that is exploratory. Faced with an enormous number of variables, many researchers have chosen to examine multiple combinations of program properties, environmental parameters, and performance measures. Such experiments tend to become probes for viable hypotheses rather than tests of specific predictions. Dunsmore and Gannon, for example, examined the importance of five program properties in construction and maintenance tasks [8]. They studied construction in two programming languages and separated subjects in two groups based on the experimenter's judgments about the quality of their performance. Their experiment includes a total of 20 different statistical tests. When large numbers of experimental conditions are examined, the likelihood of finding accidental relationships is high. The unfortunate consequence of this practice is a substantial inflation of the probability of making a type I error—inferring the existence of a nonexistent relationship. Therefore, the experimentwide confidence in these results is sharply reduced.

Consider a series of three experiments, performed by a research group at General Electric, testing the Halstead and McCabe measures against a simple count of program lines [6, 7, 16]. Across the three experiments, a total of 170 statistical tests are reported (correlations between pairs of complexity measures and between complexity measures and performance measures). More recently, Basili and Hutchens examined a family of related complexity measures [4]. They report 120 correlation coefficients and mention another set of tests that were performed but not given because the correlations were not significant. The likelihood of obtaining a statistically significant result due to sampling error or measurement error for any one or any small set of tests is quite high when the number of tests is as large as in these experiments.

Exploration as an Experimental Paradigm

Disregarding the statistical significance of the findings, how good a strategy is this approach? There are several reasons why these explorations are unlikely to be productive. It is possible to devise an enormous number of measures based on intuitions and intro-

spection about programming. The likelihood that any one of these will reveal insights into a behavior as complex and intricate as programming is small. If an index of comprehension is wanted, the processes by which programmers come to understand programs should be studied first. If an index of errors is wanted, the causes of errors must be determined. Robust, prescriptive measures of complexity will probably not be found by groping for relationships between the surface features of programs and program or programmer behavior.

Even when a large correlation is discovered, the finding often has little practical value. Without an understanding of the underlying processes that lead to the association, it is difficult to know how to use this result to advantage. The existence of a linear relationship between a measure of program size and the number of bugs in the program does not mean that program size is a good predictor of errors. Nor does it suggest that program size should be minimized to minimize the number of errors. It is also true that the weight of basketball players is highly correlated with scoring ability. This does not mean that we should choose players by weight or force-feed players to raise their scoring averages.

It would be surprising if the number of bugs in a program were not related to the size of the program. If it turned out that a specific linear relationship held over a population of programs, implementing a variety of problems, written by many programmers, then this measure could be considered as the measure to predict bugs.

The danger of these experiments is that the results are easily misinterpreted. The potential for misinterpreting is illustrated by Basili and Hutchens' [4] recent study. Motivated by an analysis of several syntactic complexity measures, Basili and Hutchens proposed a measure, Syntactic Complexity (SynC), which is a hybrid of volume and control organization measures. They attempted to validate the measure by looking for a relationship between program changes made during development, which are correlated with errors ($r \approx 0.8$) and SynC. The study was based on 19 compilers written by student programmers. No significant correlations were found over grouped data, so the authors examined each individual's programs separately. They found that for a collection of program segments written by a single programmer SynC and program changes were well correlated ($r \approx 0.8$). Lines were fitted to each programmer's data. A graph of the results is presented in Figure 1. Basili and Hutchens noted the variation in the slope of the best-fit lines and proposed that "the slope of the line may be viewed as a measure of a programmer's ability to cope with complexity since it estimates the number of program changes he makes in developing a program for each unit of complexity." They conclude that "slope ... measures the relative skills of programmers at handling a given level of complexity" and suggest that the measure may eventually be used in managerial decisions.

This inference is unwarranted from the data. An equally plausible explanation is that those programmers with steep slopes worked harder, made more changes, and consequently found a better solution, with a lower value of SynC. Under this interpretation a high slope indicates a greater ability to deal with complexity. Neither interpretation is justified by the results. The confusion can be attributed to a misunderstanding of the meaning of the correlations. For individual programmers, SynC is only *correlated* with changes that are *correlated* with errors. Changes are not errors and SynC is not complexity. The slope of the line that relates program changes to errors (unmeasured in this experiment) may vary between programmers. On the basis of the evidence, the changes of one programmer cannot be equated to those of another.

The relationship between program changes and SynC is serving double duty. Let us assume, for the sake of the argument, that errors are in constant proportion to changes for all programmers. The correlation between changes and SynC is first used to establish SynC as a measure of complexity—more specifically as a predictor of errors. The slope argu-

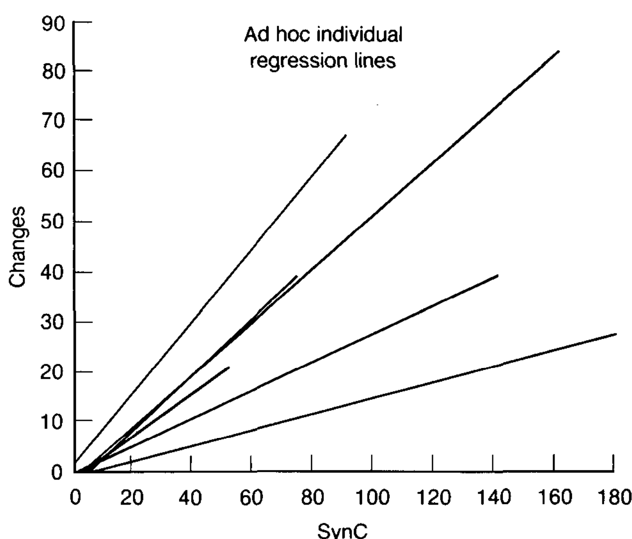

FIGURE 1. Each Line Is a Regression Line Relating SynC to Program Changes for Program Modules Written by a Single Programmer (reprinted with permission © 1983 IEEE)

ment then treats SynC as an independent measure of complexity. Two programs of equal length, generated by different programmers, are considered to be of equal complexity; if one programmer made more changes than another to implement his or her piece of software, then he or she is judged to be less competent. But remember, in this experiment students implemented the same problem. By the slope criteria, a programmer can implement the same problem with lower SynC and make fewer changes and cause fewer errors, but will be judged less competent than another programmer because the ratio of changes to SynC is greater.

It can be argued that, with limited understanding of programs and programming, exploratory work is necessary to provide direction for further study. Experiments that examine the relationship between program properties and programmer behavior, such as the Dunsmore and Gannon study, can serve to focus attention on particular aspects of the programming process. However, it should be realized that these experiments are not sufficient to justify the use of complexity metrics. Another phase of testing is needed that is directed toward a particular application. The properties of the metric must be examined to be sure that the metric is appropriate for the suggested use; specific, testable hypotheses must be generated. The experiment should mimic the anticipated application as closely as possible.

CONCLUSION

Software complexity measures have not realized their potential for the reduction and management of software cost. This failure derives from the lack of a unified approach to the development, testing, and use of these measures.

A suggested approach for the creation of complexity measures: Before a measure can be developed, a clear specification of what is being measured and why it is to be measured must be formulated. This description should be supported by a theory of programming behavior. The developer must anticipate the potential uses of the measure, which should be tested in the intended arena of usage.

Complexity measures currently available provide only a crude index of software complexity. Advancements are likely to come slowly as programming behavior becomes better understood. Users of complexity measures must be aware of the limitations of these measures and approach their applications cautiously. Before a measure is incorporated into a programming environment, the user should be sure that the measure is appropriate for the task at hand. The measure must possess the properties demanded by the use. Finally, users should always view complexity measurements with a critical eye.

REFERENCES

1. Baker, A.L. The use of software science in evaluating modularity concepts. *IEEE Trans. Softw. Eng. SE-5*, 2 (Mar. 1979), 110–120.
2. Baker, A.L., and Zweben, S.H. A comparison of measures of control flow complexity. *IEEE Trans. Softw. Eng. SE-6*, 6 (Nov. 1980), 506–512.
3. Basili, V.R. Qualitative software complexity models: A summary. In *Tutorial on Models and Methods for Software Management and Engineering*. IEEE Computer Society Press, Los Alamitos, Calif., 1980.
4. Basili, V.R., and Hutchens, D.H. An empirical study of a syntactic complexity family. *IEEE Trans. Softw. Eng. SE-9*, 6 (Nov. 1983), 664–672.
5. Basili, V.R., Selby, R.W., Jr., and Phillips, T. Metric analysis and data validation across Fortran projects. *IEEE Trans. Softw. Eng. SE-9*, 6 (Nov. 1983), 652–663.
6. Curtis, B., Milliman, P., and Sheppard, S.B. Third time charm: Stronger prediction of programmer performance by software complexity metrics. In *Proceedings of the 4th IEEE Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, Calif., 1979, 356–360.
7. Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., and Love, T. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Trans. Softw. Eng. SE-5*, 1 (Jan. 1979), 45–50.
8. Dunsmore, H.E., and Gannon, J.D. Analysis of the effects of programming factors on programming effort. *J. Syst. Softw. 1*, 2 (Feb. 1980), 141–153.
9. Evangelist, W.M. Software complexity metric sensitivity to program structuring rules. *J. Syst. Softw. 3*, 3 (Sept. 1983), 231–243.
10. Gordon, R.D. Measuring improvements in program clarity. *IEEE Trans. Softw. Eng. SE-5*, 2 (Mar. 1979), 79–90.
11. Halstead, M.H. *Elements of Software Science.* Elsevier North-Holland, New York, 1977.
12. Henry, S., and Kafura, D. On the relationship among three software metrics. *Perform. Eval. Rev. 10*, 1 (Spring 1981), 81–88.
13. Henry, S., and Kafura, D. The evaluation of software systems' structure using quantitative software metrics. *Softw. Pract. Exper. 14*, 6 (June 1984), 561–573.
14. McCabe, T.H. A complexity measure. *IEEE Trans. Softw. Eng. SE-2*, 6 (Dec. 1976), 308–320.
15. Sheil, B.A. The psychological study of programming. *ACM Comput. Surv. 13*, 1 (Mar. 1981), 101–120.
16. Sheppard, S.B., Milliman, P., and Curtis, B. Experimental evaluation of on-line program construction. GE Tech. Rep. TR-79-388100-6, General Electric, Dec. 1979.
17. Soloway, E., and Ehrlich, K. Empirical studies of programming knowledge. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept. 1984), 595–608.

CR Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics—*complexity measures*; *software science*; D.m [**Miscellaneous**]: *software psychology*; K.6.3 [**Management of Computing and Information Systems**]: Software Management

General Terms: Human Factors, Measurement

Authors' Present Addresses: Joseph K. Kearney, Dept. of Computer Science, Cornell University, Ithaca, NY 14853; Robert L. Sedlmeyer, Dept. of Computer Technology, Purdue University, West Lafayette, IN 47907; William B. Thompson, Dept. of Computer Science, University of Minnesota, Minneapolis, MN 55455; Michael A. Gray, MCC, Austin, TX 78712; Michael A. Adler, Control Data Corporation, Bloomington, MN 55440.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.