Annals of Software Engineering 1(1995)1–22

Software complexity and software maintenance: A survey of empirical research

Chris F. Kemerer†

Sloan School of Management, Massachusetts Institute of Technology, 50 Memorial Drive, E53–315, Cambridge, MA 02142-1347, USA

A number of empirical studies have pointed to a link between software complexity and software maintenance performance. The primary purpose of this paper is to document "what is known" about this relationship, and to suggest some possible future avenues of research. In particular, a survey of the empirical literature in this area shows two broad areas of study: complexity metrics and comprehension. Much of the complexity metrics research has focused on modularity and structure metrics. The articles surveyed are summarized as to major differences and similarities in a set of detailed tables. The text is used to highlight major findings and differences, and a concluding remarks section provides a series of recommendations for future research.

ACM CR categories and subject descriptors: D.2.7 [Software Engineering]: Distribution and Maintenance; D.2.8 [Software Engineering]: Metrics; D.2.9 [Software Engineering]: Management; F.2.3 [Analysis of Algorithms and Problem Complexity]: Tradeoffs among Complexity Measures; K.6.0 [Management of Computing and Information Systems]: General - Economics; K.6.1 [Management of Computing and Information Systems]: Project and People Management; K.6.3 [Management of Computing and Information Systems]: Software Management.

Keywords: Maintenance, complexity, metrics, modularity, comprehension.

1. INTRODUCTION

1.1 Why Empirical Studies of Software Maintenance?

While much is written about new tools and methods for developing new software, a significant percentage of professional software engineers' time is spent maintaining existing software. Software maintenance represents a large and growing expense for organizations. In addition, due to the shortage of experienced software engineers, the preponderance of maintenance work represents an opportunity cost of

†E-mail: ckemerer@mit.edu

those resources that would otherwise be devoted towards developing new systems. Therefore, software maintenance represents an activity of considerable economic importance and is a candidate for academic research.

As an aid to researchers interested in maintenance or maintenance-related issues, this paper surveys recent empirical studies of software maintenance. The focus on empirical studies was deliberately chosen due to the relative newness of the field. Unlike more mature disciplines, this area does not yet have a large body of well-accepted theory upon which to build. Therefore, the primary early gains have been made in careful observation of maintenance activities through empirical studies. The intent of the survey is to collect, classify, and analyze the existing body of work, with special attention paid to identifying those issues where further research would appear to be most beneficial. Of special interest is research exploring the relationships among software complexity and various important aspects of software maintenance.

1.2 The Neglect of Software Maintenance Research

Schneidewind, in his guest editor's introduction to a special issue on software maintenance in the March 1987 issue of *IEEE Transactions on Software Engineering*, (*IEEE-TSE*) noted that there was not a single article on maintenance in *IEEE-TSE* over a past period of a little more than a year. Following up on this observation, a recent comprehensive review of three leading archival journals (*Communications of the ACM* (*CACM*), *IEEE Transactions on Software Engineering* (*IEEE-TSE*), *Journal of Systems and Software* (*JSS*)) and two refereed conference proceedings (*IEEE International Conference on Software Engineering* and the *IEEE Conference on Software Maintenance*) covering 3,018 published papers over ten years found a total of only sixty-one articles on empirical studies of software maintenance, approximately 2% of the total.[1)] This exact percentage is, of course, a function of the publication outlets chosen, and the general difficulties in performing successful empirical research. However, given that it is typically estimated that commercial software development organizations spend half or more of their resources in software maintenance, the percentage of research effort devoted to software maintenance, as reflected by its publication in scholarly outlets, seems far below what its practical importance would seem to warrant. This neglect of software maintenance as a research area should

[1)]These statistics are documented in an unpublished working paper by Kemerer and Ream [1992]. (Much of the text of the current article is based on portions of that working paper.) The particular journals chosen were meant to represent a mainstream view of the field, since software maintenance is such a widely shared problem. Exclusion of some other relatively more specialized journals, such as the *International Journal of Man-Machine Studies*, may bias the percentage of articles downward, while the inclusion of the specialist *Conference on Software Maintenance* may have biased the percentage upward. However, the key result that the percentage of research effort is significantly out of proportion to the economic significance of this activity is unlikely to be reversed by other samples of publication outlets.

concern practitioners, since little effort is being devoted to discovering new knowledge about an activity of considerable economic importance.

For a paper to be included in the set of sixty-one articles it had to present and analyze empirical data relating to software maintenance. This research adopts the ANSI/IEEE standard 729 definition of maintenance: "Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment" [Schneidewind 1987]. Empirical research on software maintenance has much in common with research on new software development, since both involve the creation of working code through the efforts of human developers equipped with appropriate experience, tools, and techniques. However, as Swanson and Beath point out, software maintenance involves a fundamental difference from development of new systems in that the software maintainer must interact with an existing system [Swanson and Beath 1989].

Some of the research included herein overlaps the areas of both maintenance and development. One example is that there is evidence to suggest that development decisions, such as the use of structured programming techniques, are expected to have a noticeable effect on later maintenance efforts. Another example is that it has been noted that the cost of correcting program errors typically increases significantly the later they are discovered, suggesting that extra effort in the development phase will reduce maintenance costs [Shen *et al.* 1985]. Complexity metrics are another area of study that applies to both development and maintenance. To account for these sorts of overlaps, a study or experiment did not have to specifically address maintenance issues in order to qualify for inclusion, but was required to provide insight that could be readily applied to maintenance. It is suggested that this review may therefore be broadly useful to researchers in new software development who may also benefit from familiarity with this work. In particular, a greater familiarity with the expected maintenance consequences of certain development practices may be of particular value to software development managers.

1.3 Organization of the Paper

The primary presentation approach in this paper is to briefly summarize the contributions of every article in table form, and then expand on these comments in the text for a subset of those articles that merit additional discussion. An initial analysis of the papers suggested two areas, software complexity measurement and software comprehension. However, the general category of software complexity measurement had a sufficient number of articles to merit splitting it into two categories. Therefore, the organization of the paper will follow the following three section outline:

1. Modularity and structure metrics.
2. Other complexity metrics.
3. Comprehension.

Within the Comprehension section there are two subsections: differences across individual maintainers, and the impact of various comprehension aids. In each of the sections there is a table that summarizes the details of each of the studies. The format of the tables includes the following data:

- Author, year.
- Publication in which the article appeared.
- Methodology (field studies, experiments, and surveys, lab studies and experiments).
- Data source.
- Dependent variable(s).
- Statistical test(s) employed, if any.
- Brief summary of key results.

The tables are additionally designed to assist readers interested in narrower topics, e.g., "COBOL programming" or "laboratory experiments involving students". The remainder of this paper is organized as follows. The next section, "Software Complexity Measurement", presents work whose primary contribution lies in the relationship between complexity measurement and software maintenance results. Section 3, "Comprehension", focuses on research whose primary interest is in how maintainers' comprehension of existing software can be improved. A final section provides some concluding remarks.

2. SOFTWARE COMPLEXITY MEASUREMENT

2.1 Introduction

Research in this area is generally focused on the relationship between a complexity measure and maintenance effort, or among complexity measures. Basili defines software complexity as "... a measure of the resources expended by another system while interacting with a piece of software. If the interacting system is people, the measures are concerned with human efforts to comprehend, to maintain, to change, to test, etc., that software". [1980, p. 232]. Curtis *et al.* similarly define the same concept (which they refer to as psychological complexity) as: "Psychological complexity refers to characteristics of software which make it difficult to understand and work with" [1979, p. 96]. Both of these authors note that the cognitive load on a software maintainer is believed to be higher when structured programming techniques are not used.

Schneidewind estimates that 75–80 percent of existing software was produced prior to significant use of structured programming [1987]. A key component of structured programming approaches is *modularity*, defined by Conte *et al.* [1986, p. 197] as "the programming technique of constructing software as several discrete parts". Structured programming proponents argue that modularization is an improved programming style, and therefore, the absence of modularity is likely to be a significant

practical problem. A number of researchers have attempted to empirically validate the impact of modularity on either software quality or cost with data from actual systems, achieving somewhat mixed results. (See table 1.)

There is a significant amount of other work in the software complexity metrics area, for example, volume measures such as those of Halstead's Software Science. (See table 2.) Work in this area often overlaps the work in modularity and structure, with many articles reporting results for both. Given the relatively large amount of work in measurement, an attempt has been made to place an individual article into either table 1 or table 2, but not both. Researchers who are broadly interested in the issue of software complexity measurement and its relation to productivity should carefully examine both tables.

Dependent variables in this research are generally either quality related – number of errors or defects found (sometimes number of changes is used as a surrogate), or productivity related – effort required to make a change, time required to debug, et cetera. This emphasis on performance evaluation is a pervasive theme in this literature.

2.2 Modularity and Structure

2.2.1 Module Size

While there are potentially a number of important attributes associated with modules that merit study (e.g., cohesiveness), one fundamental issue has dominated a significant amount of the research in this area, which is the impact of the size of the module. An important widely disseminated early piece of research on the impact of modularity and structure was by Vessey and Weber [1983]. They studied repair maintenance in Australian and US data processing organizations and used subjective assessments of the degree of modularity in a large number of COBOL systems. In one data set they found that code with greater modularity (on average, more, smaller modules) was associated with fewer repairs; in the other data set no effect was found. These equivocal results were unexpected by the authors, and in their conclusion they note "*Our results stand as a challenge to some conventional wisdom and the proponents of structured programming (who include us). We readily acknowledge that our research is exploratory and there are problems with the statistical model. Nevertheless, the results are anomalous*". [1983, p. 134].

A number of researchers took up this challenge. Since Vessey and Weber focused on repair maintenance, many follow-on studies have examined the "number of errors" as their dependent variable. Basili and Perricone [1984] and Shen *et al.* [1985] in separate studies found that larger modules tended to have significantly fewer errors. Similarly, Compton and Withrow, in a recent examination of 263 Ada packages, found that smaller packages had a disproportionately high share of the errors. A study by An *et al.* [1987] analyzed change data from two releases of UNIX. They found that the average size of unchanged modules (417 lines of C) was larger

Table 1

Modularity and structure.

Author	Publication	Methodology	Data	Dependent Var's	Statistical tests	Reported results
Troy and Zweben [1984]	Journal of Systems and Software	field study	21 metrics from 73 designs	number of source code modifications	stepwise regression	Among the hypotheses tested (effects of coupling, cohesion, complexity, modularity and size) impact of coupling was most strongly supported.
Woodfield *et al.* [1981]	Journal of Systems and Software	lab study	33 programs	effort	Pearson correlation Spearman rank correlation, Mean Squared Error	McCabe's $V(G)$, Halstead's E, and SLOC all produced a significant correlation with actual effort, but the author's measure based on logical modules had the best correlation while keeping relative errors small.
Vessey and Weber [1983]	Communications of the ACM	field study	447 COBOL programs	number of repair incidents	ANOVA, ANCOVA	Analyses of 3 data-sets, one Australian and two US had mixed results. Conventional hypotheses about impact of complexity and structure received either weak or no support.
Bowen [1983]	IEEE Conference on Software Maintenance	field study	17 DoD projects, Hughes Air Defense	% errors, error rate per module	none	Suggests that patching is a poor maintenance technique, and that if a module requires substantial changes, it should be completely rewritten.
Basili and Perricone [1984]	Communications of the ACM	field study	33 months of data collected from development of a 90 KLOC FORTRAN system	number of errors	ANOVA	Most errors are the result of misinterpretation of functional specs, maintenance is more expensive for adapted modules taken from other systems, and larger modules appear significantly less error prone (per LOC).
Lohse and Zweben [1984]	Journal of Systems and Software	lab experiment	50 student programmers, 2 versions of a 500 line program	time to modify	ANOVA	The type of modification performed influenced modifiability, but there were no consistent effects due to the type of intermodule coupling.
Shen *et al.* [1985]	IEEE Transactions on Software Engineering	field study	3 program products (totaling 5 releases and 1428 modules)	number of errors	ANOVA	Smaller modules have a higher rate of errors per LOC than larger modules. Metrics related to amount of data and the structural complexity (number of loops, conditional statements, and Boolean operators) proved to be the most useful in identifying error prone modules at the earliest stages of testing.
Yau and Collofello [1985]	IEEE Transactions on Software Engineering	lab experiment	4 PASCAL programs (4KLOC each)	design stability	none	A new metric was introduced to predict maintenance ripple effects. Some correlation was noted, but the sample size was too small for metric validation.

Rombach [1987]	IEEE Transactions on Software Engineering (also: 1985 IEEE Conf. on SW Maintenance)	lab experiment	6 grad student programmers, 12 SW systems (1.5–15.2 KLOC) in LADY and PASCAL	maintainability (effort)	Mann–Whitney U; Spearman correlation	More structured language reduced maintenance effort. Complexity metrics were good predictors of maintainability.
An *et al.* [1987]	3rd IEEE Conference on Software Maintenance	field study	123 modules of code, UNIX system 3 and 5	statement changes, nesting level, etc.	multiple regression, t-test	Patterns do exist – larger modules are less likely to be changed, modules with relatively high nesting levels are likely to have lower nesting levels after modification.
Selby and Basili [1988]	IEEE Conference on Software Maintenance	field study	135KLOC production system	total errors, errors/KLOC, correction effort, correction effort/KLOC	ANOVA	High coupling/strength ratios (degree of coupling with other clusters divided by a measure of cohesion within the cluster) corresponded to high error rates.
Yau and Chang [1988]	1988 IEEE Conference on Software Maintenance	lab experiment	5 experienced programmers, 5 programs ranging from 6.5–13KSLOC of PASCAL	programmer effort	Pearson correlation	For moderate to difficult modifications, (30–60 statements), their data interdependency metric was good at predicting effort. Most of the errors and changes encountered were caused by use of global variables and subsequent side effects.
Gibson and Senn [1989]	Communications of the ACM	lab experiment	36 programmers 3 differently structured versions of a 2KLOC program	time to modify, percent of errors and effort	ANOVA	Structure decreased overall maintenance time and reduced ripple errors.
Compton and Withrow [1990]	Journal of Systems and Software	field study	a software product containing 263 packages comprising 64KLOC of ADA	number of errors	Kruskal-Wallis, Pearson correlation	Smaller packages (~12 lines of executable code) exhibited a disproportionately high error density. Packages with system integration and test defects are six times more likely to exhibit postdelivery defects than those without.
Banker *et al.* [1993]	Communications of the ACM	field study	65 commercial software projects	effort, productivity	multiple regression	Module size, procedure size, and the use of complex branching were all found to significantly affect software maintenance costs.

Table 2

Complexity metrics.

Author	Publication	Methodology	Data	Dependent Var's	Statistical tests	Reported results
Henry and Kafura [1981]	IEEE Transactions on Software Engineering	field study	UNIX operating system	number of changes	rank correlation	Proposed complexity measure ("Fan-in, fan-out") highly correlated with number of changes made, suggesting that this metric may predict error rates.
Sunohara *et al.* [1981]	IEEE International Conference on Software Engineering	field study	45 FORTRAN modules of a 200-module 25.5KLOC program	correlation between metrics and error count/programming time	Pearson correlation	McCabe's $V(G)$, Halstead's E, Weighted Statement Count, and Process $V(G)$ correlated highly with management data, step count did less well but was still well correlated.
Gremillion [1984]	Communications of the ACM	field study	346 PL/I programs	number of repair requests	Pearson correlation linear regression	Lines of code was determined to be the most accurate predictor of repair request volume. Repair request increased as a function of size and frequency of use – related to complexity and age.
Gustafson *et al.* [1985]	2nd IEEE Conference on Software Maintenance	field study	49 modules of code, UNIX system 3 and 5	statement changes	none	It was noted that "if" statements are unusually likely to change (11.4%) compared to "while" (8.5%), "for" (7.9%), and "switch" (6.8%). It was also noted highly nested statements are unlikely to change.
Jensen and Vairavan [1985]	IEEE Transactions on Software Engineering	field experiment	202 PASCAL software modules	correletion of complexity metric values	Pearson correlation	The normalized discrepancy between Halstead's length estimator and actual program length was significantly higher than that reported in earlier studies. Correlation between McCabe's and either Halstead's N, V, and E or actual program length was also lower than the correlations indicated in previous studies.
Kafura and Reddy [1987]	IEEE Transactions on Software Engineering	field study	3 versions of one system	subjective evaluation	none	Extreme metric scores identified troublesome components and agreed with the manager's subjective evaluations.
Li and Cheung [1987]	IEEE Transactions on Software Engineering	lab study	255 short student-written FORTRAN programs	complexity	Pearson correlation	Metrics within each group (volume or control structure complexity) were highly correlated, but correlation between metric groups was much lower. This suggests that a hybrid metric using metrics from both groups may yield a more useful measure of complexity.
Card and Agresti [1988]	Journal of Systems and Software	field study	8 software projects, 37–106 KLOC each	number of errors, effort	Wilcoxon rank sum	Changes in measured complexity accounted for 60% of the variation in error rate.

Wake and Henry [1988]	IEEE Conference on Software Maintenance	field experiment	commercial software product 15KLOC of C	number of lines of code changed	Peason correlations, multiple regression	A combination of metrics from different classes (code, structure, hybrid) was found to be a far more effective predictor of maintenance than any single metric.
Lind and Vairavan [1989]	IEEE Transactions on Software Engineering	field study	400KLOC medical imaging application, mostly PASCAL and FORTRAN	number of changes to the code (errors)	Pearson correlation coefficient	The conceptually simple measures of complexity such as the number of lines of code correlated with the development effort just as well or better than the more sophisticated standard complexity metrics.
Coupal and Robillard [1990]	Journal of Systems and Software	field study	metrics obtained in 19 previous studies (14,348 rountines in various languages)	relative complexity	principle components factor analysis without rotation	63% of the variability in the measurement of classical metrics is represented by only one factor in the projects analyzed volume.
Gorla *et al.* [1990]	IEEE Transactions on Software Engineering	lab experiment	311 student COBOL programs	time to debug	linear and quadratic regression	Certain COBOL style metrics (characteristics) have significant correlations with debug times, to some degree independent of program complexity and size.
Munson and Khoshgoftaar [1990]	Journal of Systems and Software	lab study	27 FORTRAN programs	debugging effort	factor analysis with varimax rotation, linear and quadratic	The relative complexity metric is a "reasonable" statistical tool for identifying programs that will require more effort to debug.
Oman and Cook [1990]	Journal of Systems and Software	field study	3 student software projects, each in PASCAL and >3KLOC	complexity of PDL modules	Pearson correlation coefficient	Detailed PDLs had a high degree of correlation with corresponding source code. PDL metrics follow many of the same statistical patterns exhibited in code metrics.
Porter and Selby [1990]	Journal of Systems and Software	field study	16 FORTRAN systems 3k–100KLOC each (NASA)	accuracy in identifying high risk components	none	In the environment used in the testing, classification trees had an average accuracy of 79.3% for fault-prone and effort-prone components.
Benander *et al.* [1990]	Journal of Systems and Software	field study	311 student COBOL programs	correctness, debugging time, structure and style measures	linear and quadratic regression	Programs containing GOTOs were found more likely to have errors, took longer to debug, and had worse structure than GOTO-less programs.
Gill and Kemerer [1991]	IEEE Transactions on Software Engineering	field study	834 modules of PASCAL and FORTRAN	maintenance productivity	Pearson correlation coefficient; linear regression	Normalizing the McCabe Cyclomatic Complexity by LOC ("complexity density") was a useful explanator of maintanence productivity. Myers and Hansen variants of McCabe produced essentially similar results.

than that of changed modules (279 lines of C). Unfortunately, they did not provide any analysis to determine if this difference was statistically significant. However, other studies that have appeared elsewhere have suggested that some degree of modularity is necessary. Korson and Vaishnavi [1986] conducted four experiments comparing the time required to modify two alternative versions of a piece of software, one modular and one monolithic. In three of the four cases the modular version was significantly easier to modify.

Therefore, a newer, alternative hypothesis is that modules that are either too large (undermodularization) or too small (overmodularization) are unlikely to be optimal. For example, Conte *et al.* [1986, p. 109] note that: "*The degree of modularization affects the quality of a design. Overmodularization is as undesirable as undermodularization*". It is a common general belief that large modules are more difficult to understand and modify than small ones, and maintenance costs will be expected to increase with average module size. If the modules are too large they are unlikely to be devoted to single purpose. However, research has clearly shown that a system can be composed of too many small modules. If the modules are too small, then much of the complexity will reside in the interfaces between modules and therefore they will again be difficult to comprehend. Interfaces are relevant because they have been shown to be among the most problematical components of programs [Basili and Perricone 1984]. Therefore, complexity could decrease as module size increases. Some recent work has suggested that a U-shaped function is likely, with an optimal module size that lies between the extremes noted by earlier research [Banker *et al.* 1993].

2.2.2 *Coupling*

Another important issue within this set of literature is the effect of module coupling on performance[2)]. A 1981 study by Troy and Zweben explored a number of hypotheses dealing with structured programming concepts, including the notion of coupling. Some of the intuition behind structured programming is that minimally related tasks should be kept independent by locating their functions in separate modules. Independence of modules is maximized to the degree that coupling among modules is minimized [Lohse and Zweben 1984]. Of all the hypotheses tested by Troy and Zweben, they found the strongest support for the notion that the number of source code modifications (a surrogate for errors) was positively correlated with a high degree of coupling, i.e., highly cohesive but loosely coupled modules were less likely to require modification.

Continuing in this stream of research, Selby and Basili studied a large production system for which actual error data were available [1988]. They used as their independent

[2)]Intuitively, coupling refers to the degree of interdependence between parts of a design, while cohesion refers to the internal consistency within parts of the design. All other things being equal, good software design practice calls for minimizing coupling and maximizing cohesiveness.

variable the ratio of coupling to cohesion, (cohesion defined intuitively as the amount of interaction among elements within a module), where a low value of such a ratio was believed to reflect good structured programming practice. They found strong support for the notion that high values of their ratio were associated with higher error rates and higher efforts to correct errors.

Lohse and Zweben note that there are multiple dimensions to improving module coupling, including the size and type of the information passed to the module. They performed a lab experiment using student programmers to determine whether passing information using either global variables or parameter lists had an effect on the time required to modify a program. They note that the literature offers conflicting advice on this question[3] and therefore it was a topic meriting experimental study. Unfortunately, their experiment yielded no conclusive results. A later study by Yau and Chang, however, found that use of global variables was correlated with more errors and changes [Yau and Chang 1988].

In general, not enough is known about the proper ways to minimize coupling. This is clearly a topic that merits further research, particularly in newer implementations, such as object-oriented environments, where the equivalent of coupling needs to be considered in the design of objects, methods and classes.

2.3 Complexity Metrics

Within the empirical research on software maintenance surveyed, the largest part was devoted to software metrics, particularly those relating to aspects of software complexity as defined above. With only a few exceptions, the emphasis in this review is on those studies of metrics that examined the relationship between the metrics and maintenance-related dependent variables, such as error rates, time to locate and correct defects (debugging), and number of subsequent changes.

2.3.1 Relationships among Metrics and Maintenance

Sunohara *et al.* simultaneously collected data on several of the main complexity metrics, including McCabe's $V(G)$ and Halstead's E, as well as source lines of code (SLOC)[4] for a medium-sized FORTRAN system and calculated the inter-metric correlations [McCabe 1976; Halstead 1977; Sunohara *et al.* 1981]. For example, they found a Pearson correlation coefficient value for the pairwise correlation of non-comment SLOC and Halstead's E of .812 ($p < .001$). The implication of these strong correlations among these metrics, is that a metric such as SLOC may be preferable, since it

[3] Structured design argues that use of global variables will result in higher coupling, while complexity metrics such as Halstead's E would indicate less coupling stemming from use of global variables [Lohse and Zweben 1984].

[4] These are referred to as "steps" in their paper, since this is the standard nomenclature in Japan. (See, for example, [Cusumano and Kemerer 1990].)

provides similar information but with greater ease of collection and of managerial interpretation. Similar results were obtained by Gremillion, who collected multiple metrics for 346 PL/1 programs [Gremillion 1984]. Interestingly, his correlation between SLOC and E was .82 ($p < .001$), nearly identical to the Sunohara *et al.* study. Gremillion's main finding was that the number of program defects was significantly related to the complexity metrics, and in particular that the best single predictor metric was SLOC. Essentially the same results were found by Lind and Vairavan in a study of a number of releases of a large medical imaging system [Lind and Vairavan 1989]. They found a high correlation between the more complex metrics and SLOC, and found that SLOC was the best single predictor of number of "system performance reports" and development effort. However, most recently, work by Gill and Kemerer suggested that normalizing McCabe Cyclomatic Complexity by SLOC ("complexity density") did produce a useful explanator of software maintenance productivity [Gill and Kemerer 1991].

There are two main conclusions that can be drawn from this set of research. The first is that complexity metrics can be useful predictors of the maintenance behavior of systems, and that greater use of measurement in systems development, testing, and maintenance is recommended. The second conclusion is that a number of the more complex metrics may be essentially measuring the size of the program or other component under investigation, and therefore may provide little additional information, unless normalized or otherwise calibrated.

2.3.2 Dimensions of Software Complexity

Stemming in part from the results summarized above, some research has focused on attempting to identify unique dimensions of software complexity, i.e., which metrics can be seen as relatively independent and thus may represent different dimensions. Li and Cheung, in a study of 255 student FORTRAN programs, collected data on 31 separate metrics [Li and Cheung 1987]. They found that the metrics could be roughly divided into two groups, "volume metrics" (i.e., size) and "control metrics". Their recommendation was to use a metric from both groups, or to use a hybrid metric that could capture elements of both. A similar conclusion was reached by Wake and Henry, who investigated the relationship between software metrics and the number of LOC changed in a set of 193 modules of C code [Wake and Henry 1988]. They suggest that a model with a combination of metric types predicts better than any single metric. Most recently, Munson and Khoshgoftaar used factor analysis to isolate two dimensions of complexity which they label "volume" and "modularity" [Munson and Khoshgoftaar 1990]. They found their generated metric to be good predictor of debugging time for a set of 27 FORTRAN programs.

This research provides additional support to the notion of using software complexity metrics to predict maintenance activity. It further refines earlier metric work in noting that a small number of underlying dimensions of complexity are

represented in the literature by a relatively large number of proposed metrics. For practitioners the result is that they should consider adopting a small set of metrics to aid their management of the maintenance process. For researchers the conclusion is that proposals for new metrics must demonstrate both orthogonality to existing metrics and superior performance in terms of predicting dependent variables of interest.

3. COMPREHENSION RESEARCH

A critical factor that differentiates software maintenance from new software development is the software engineer's need to interact with existing software and documentation. Quite often the maintainer will not have participated in the development of the system, or at least may be asked to maintain unfamiliar portions of the system. Therefore it is not surprising that a significant amount of software maintenance research has focused on the issue of comprehension of the existing system. The research described in the previous section on complexity metrics may also be seen as applying to comprehension. This is because a program that is considered to be more error-prone because it, say, contains more complex logic paths, must be founded on the notion that such a program is harder for the maintainer to comprehend and therefore harder to correctly maintain.

However, such arguments about the impact of complexity on comprehension are only indirect in that, even when increased complexity is shown to be correlated to a decrease in a performance variable, it is only a presumption that such effects are caused through difficulties in comprehending the more complex artifacts. This section focuses on studies that more directly address the issue of comprehension, through use of dependent variables that operationalize comprehension or other types of emphasis. This issue has been identified as critical to the subject of maintenance for some time. It was reported as early as the late 1970's that more than fifty percent of all software maintenance effort was devoted to comprehension [Fjelstad and Hamlen 1983]. Dean and McCune, in a survey of Air Force maintainers reported that the top three problems in software maintenance were all comprehension related: (1) a high rate of personnel turnover requiring that unfamiliar maintainers work on the systems, (2) difficulty in understanding the software, particularly in the absence of good documentation, and (3) difficulty in determining all of the relevant places to make changes due to an inadequate understanding of how the program works [Dean and McCune 1983]. Of the work covered in this review, two research problems dominate: the variation in individual maintainer's ability and the efficacy of various aids to maintenance comprehension.

3.1 Individual Differences

One consistent empirical observation has been that certain individuals, often those with greater experience, are simply better at maintenance tasks under nearly

Table 3

Comprehension.

Author	Publication	Methodology	Data	Dependent Var's	Statistical tests	Reported results
Woodfield *et al.* [1981]	IEEE International Conference on Software Engineering	lab experiment	48 experienced programmers, 8 versions of the same program	comprehension (correctly answering questions about unfamiliar code in a limited time)	ANOVA	Functional modularization and super modularization were worst in comprehensibility, no modularization did surprisingly much better, and abstract data type was the most effective. Commenting alone had more of an effect than variations in modularization, boosting scores in every case.
Shneiderman [1982]	Communications of the ACM	lab experiment	57, 32 undergraduates, 223 LOC PASCAL program	comprehension test score	ANOVA, t-test	In two experiments, groups given data structure diagrams scored higher on comprehension tests than did groups without aids or with control flow documentation.
Weiser [1982]	Communications of the ACM (also IEEE ICSE 1981)	lab experiment	21 experienced programmers, using Algol-W	immediate recall through identification	ANOVA, Wilcoxon matched pairs, signed ranks, Spearman rank correlation	Three programs were used in the testing. One showed strong evidence of "slicing" by debuggers, the other two were much weaker in supporting the slicing hypothesis.
Dean and McCune [1983]	IEEE Conference on Software Maintenance	field survey	3 Air Force sites	problem areas in the maintenance process	none	The bulk of maintenance problems are in the area of comprehension: lack of understanding of the programming environment, documentation, and relevant places to enact the changes.
Miara *et al.* [1983]	Communications of the ACM	lab experiment	79 undergraduates	% of 10 comprehension questions answered correctly	ANOVA	Expert and novice groups were givem programs with varying levels of indentation (0, 2, 4, or 6 spaces). Groups with 2 or 4 spaces performed significantly better. Experts outperformed novices, and were less concerned with the indentation levels.
Ramsey *et al.* [1983]	Communications of the ACM	lab experiment	20 graduate student programmers, 3 tasks each	subjective preference, comprehension, implementation quality/errors/time	unknown (only p-values reported)	Program Design Languages (PDLs) outperformed flowcharts in terms of quality of the designs and in subjective preferences. Flowcharts were not superior on any tested outcome.
Harrison and Cook [1986]	Journal of Systems and Software	lab experiment	148 undergraduates	% of 6 comprehension questions answered correctly	t-test	Deeply nested conditionals (IF-THEN-ELSE-IF ...) were no more difficult to comprehend than "skinny" nested decision trees.

Littman *et al.* [1987]	Journal of Systems and Software	lab experiment	10 professional programmers, 250-line FORTRAN program	successful enhancement	none	A strong relationship was established between using a systematic approach in understanding a program and modifying it successfully.
Letovsky [1987]	Journal of Systems and Software	lab study	6 programmers, 250-line FORTRAN program	cognitive process	none	A model was developed to analyze the "thinking aloud protocol" of software engineers as they form understanding of code and perform modifications.
Tenny [1988]	IEEE Transactions on Software Engineering	lab experiment	148 students, 6 versions of a PL/I program	programmer comprehension	ANOVA	The commented versions of the code always did better than the uncommented versions, with the largest difference evident in the absence of procedure formatting. The positive effect of procedure formatting was noted only in the absence of comments.
Baecker [1988]	IEEE International Conference on Software Engineering	lab experiment	44 student programmers, two versions of a C program	number of questions answered correctly about the code	ANOVA	Graphically enhancing the source code text increased readability 25% over standard printout.
Curtis *et al.* [1989]	Journal of Systems and Software	lab experiment	3 FORTRAN programs (50 LOC each)	number of errors, effort	ANOVA	Natural language was less effective than constrained language or ideograms in aiding programmer comprehension. One third to one half of the variation in overall performance was attributed to individual differences among participants.
Oman *et al.* [1989]	Journal of Systems and Software	lab experiment	193 undergraduates	% correct fixes and time taken	MANOVA	Groups given error messages that contained line numbers outperformed groups without. More experienced students did better than those with less experience, and more experienced students were less affected by better documentation.
Lehman [1989]	IEEE Transactions on Software Engineering	lab experiment	52 evening students	error rates and time required	chi-square, Multiple Classification Analysis	Experiment compared textual documention (Yourdon data dictionaries) to graphical documentation (Jackson data structure diagrams) as aids in understanding a COBOL program. Subjects with the graphical documentation took less time and had slightly better performance.

all conditions than those without such skills. In a study whose main focus was on the optimum amount of program indention, Miara *et al.* found that expert subjects (those with three or more years of programming in school and/or more than two years of professional programming) outperformed novices under all conditions [Miara *et al.* 1983]. Curtis *et al.* report that in a series of experiments involving professional programmers, the number of years of experience was not a significant predictor of comprehension, debugging, or modification time, but that number of languages known was [Curtis *et al.* 1989]. They suggest that this means that breadth of experience may be a more reliable guide to ability than length of programming experience. Most recently, in a study of undergraduate programmers, Oman *et al.* found that seniors outperformed juniors who outperformed sophomores in all categories [Oman *et al.* 1989].

All of this research gives an important message to researchers that the ability and experience levels of subjects in experiments must be carefully controlled if meaningful results are to be obtained. However, ultimately knowing that more experienced maintainers perform at a higher level is only interesting if managers understand why this is so. For example, do some individuals' problem solving styles naturally lend themselves to being good maintainers, such that they perform well, are rewarded appropriately, and stay to gain additional experience in maintenance? Or, does performing a lot of maintenance work provide experiential learning such that all or most software engineers could eventually become good maintainers? If this were better understood then managers could take action to (1) make more informed choices about assigning individual maintainers to tasks, and (2) improve conditions under which maintainers gain such experience faster, so that less-skilled maintainers can emulate the better performers.

Two studies in this review have had as their focus an attempt to construct theories of comprehension from detailed investigations of observing software engineers performing maintenance. Littman *et al.* video-taped ten professional programmers as they went about doing a constructed maintenance problem [Littman *et al.* 1987]. They identified two generic strategies which they called "systematic" and "as-needed". As the names imply, maintainers employing a systematic strategy attempted to construct a mental model of how the program worked, and then used that mental model in the performance of their maintenance task. Others only examined the program code when necessary to check specific hypotheses. The systematic maintainers were the only ones who successfully completed the maintenance tasks. Recently, Robson *et al.* have noted that this finding may be an artifact of the small program used in the experiment, and that on large programs the systematic approach may be infeasible, leaving no preferred strategy [Robson *et al.* 1991].

Letovsky video-taped and analyzed verbal protocols of six professional programmers [Letovsky 1987]. These verbal protocols revealed micro-level processes that maintainers performed as well as knowledge types that maintainers sought out as they went about their task. The author suggests that such data will be useful both to researchers in developing cognitive theories of maintenance and to practitioners in identifying what types of aids might be most useful in supporting maintenance.

3.2 Aids to Comprehension

Within this area a significant portion of the research has been addressed to the relative utility of various aids to comprehension, most particularly graphical versus text-based aids. Shneiderman *et al.* in a lab experiment testing comprehension found that groups using data structure diagrams outperformed those without such aids or with control flow documentation [Shneiderman 1982]. Lehman conducted an experiment and found that the graphical data structure diagrams-equipped group took less time and had fewer errors on the same task as a group equipped with textual Yourdon style data dictionaries [Lehman 1989]. An experiment by Baecker even showed that graphically enhanced text was a significantly superior aid to plain text in a test of comprehension [Baecker 1988].

However, in a study by Ramsey *et al.*, they found that groups equipped with program design language documentation (PDLs) performed better than flowchart groups [Ramsey *et al.* 1983]. This study was later criticized by the previously cited study by Curtis *et al.* for having results that may have been confounded by inadequate controls in the experimental design with respect to the experience level of the programmers [1989, pp. 170-171]. In particular, it may have been the case that the flowcharts were used by a group that was, on average, of less ability than the PDL group. In their own experiments Curtis *et al.* found the choice of whether a constrained language or ideograms (symbols) was superior to be somewhat task-dependent. However, natural language was never found to be a superior format in any of their four experiments.

The Curtis *et al.* experiments, besides being the most recent of the studies reviewed here, also offer a clear model for how such experiments on comprehension should be performed. They also provide a detailed review of previous research on comprehension, and this paper is recommended reading for researchers beginning work in this area. It concludes with the suggestion that "*Little additional research is needed that compares flowcharts to a program design language on module-level tasks. Rather, attention needs to be focused on the context of the documentation, such as different ways of representing data structures or state transitions*". [1989, p. 202].

4. CONCLUDING REMARKS

The first broad conclusion from this review and analysis of empirical research in software maintenance is that the area has been understudied relative to its practical import. It confirms Schneidewind's observation that the software engineering field needs to reassess its priorities with regard to research topic selection and devote more attention to maintenance.

In terms of specific research areas covered, this review noted three broad areas of coverage: (1) software modularity and structure, (2) general software complexity metrics, and (3) software comprehension. This section focuses on discussing suggestions

for future research, and these recommendations are summarized in table 4, which appears towards the end of this section.

A great deal of work has been directed at determining the benefits of modularity, with the most recent work suggesting that there is an optimum level in each environment that can be discovered through the use of statistical models. Further work to confirm this finding and to determine the range of values and determinants of the differences would be useful, and could eventually lead to the development of local standards for proper practice. There has been less work on the issue of inter-module coupling, but all of the results argue for greater emphasis on reducing coupling when possible. There is some limited evidence that the "ripple effects" caused by the propagation of errors through coupling are more expensive to correct than primary errors, but further work on this topic seems necessary.

Considerable effort has gone into correlating complexity metric scores with increased effort, errors, changes or all three, and it seems clear that strong relationships do exist. What also seems clear is that many complexity metrics measure the same dimension, e.g., program size. Therefore, in the absence of some other compelling argument, the publication criteria for new metrics must be that they be shown to be sufficiently orthogonal to existing measures. That is, complexity metrics need to be shown to be adding value beyond representing size. It has also been suggested that systems grow in complexity as they age, but why this may be true is not well documented. There is a need for more longitudinal studies that can reflect a system's status at various points in its life. Most useful would be studies that track all phases of the life cycle (including analysis and design) so that investigations could be done to determine the effects on subsequent maintenance requirements caused by using different techniques and emphases during the earlier phases.

A significant amount of research activity has been devoted to the issue of maintainer comprehension of existing source code and documentation. Wide individual variations in performance have been noted by many researchers. One laboratory finding on this topic is that a systematic approach to performing maintenance tasks appears more effective than the technique of referencing the code only as needed for each step in performing the task. Further work is required both to validate this finding and to discover other habits of good maintainers so that these techniques can be further routinized and taught to new maintainers. A second finding in this area is that graphical aids seem to be, on the whole, as good as or better than text-based documentation. With the increasing availability of easy to use software for generating this documentation this would appear to be an inexpensive recommendation for managers to adopt.

In general, software maintenance is likely to gradually evolve into a better understood activity, but there are economic advantages to speeding this process. As software managers recognize the importance of the maintenance process, more resources can be allocated to improve it. This gradual realization of importance may help alleviate the possible stigma and morale problems associated with maintenance work, and is crucial to promoting further research.

Table 4

Summary recommendations for future empirical maintenance research.

Software modularity and structure

1. More work on determining optimal levels of modularity
2. More work on effects of coupling minimization techniques
3. More work on relationship between coupling and ripple errors

General software complexity metrics

1. Less work on new metrics that have high correlations with existing metrics
2. More experimentation with regard to impacts of complexity on performance

Software comprehension

1. More work on developing measures of maintainer ability and experience
2. More work on impact of experience on performance
3. More work on how documentation is used (or not) in the field
4. More work on examining the habits of high ability maintainers

Because so little theory currently exists it remains important that research be empirically driven in order to record the observations that will lead to greater theory development in this area. An obstacle faced by researchers is the difficulty in obtaining good data to analyze. Data collected from field studies are often not complete, and can be inaccurate depending on how well constraints are enforced ensuring consistent data reporting. In addition to inaccuracies, it may be the case that organizations are reluctant to release what they may view as proprietary data. This has been suggested as one of the causes for the emphasis in the research literature on maintenance tasks being done in an academic or military setting [Hale and Haworth 1988]. One solution to this problem may be the establishment of "software maintenance research databases" where data could be contributed by organizations under the agreement that a neutral party, such as a university-affiliated research center, would maintain the anonymity of the individual contributions.

In order to facilitate such industry cooperation and therefore an increase in the quantity of maintenance research, studies need to be conducted with an eye towards how the results can be eventually utilized by maintenance managers. The influence of approaches such as the Software Engineering Institute's Capability Maturity Model and the ISO 9000 standard may further encourage the measurement of software processes, including maintenance. As managers acquire the skills to use metrics effectively and begin to benefit from software maintenance research, they will be increasingly willing to encourage further studies.

Lastly, tools for metric collection have historically been constructed by the researchers as needed, and were not readily available. More recently, automated tools have come on the market and it is expected that as data collection becomes easier, more data will be available to analyze and more research will be conducted. As new automated metric gathering tools become increasingly commercially available, validation

research of applying metrics to different environments will become much easier and the quantity of research should increase. This validation research needs to be coordinated, correlating the measurement observations from a wide variety of metrics and environments. With these common definitions, better tools, and greater sharing of data significant progress can be expected in the next decade.

ACKNOWLEDGEMENTS

Research support from the Center for Information Systems Research (CISR) at MIT and the research assistance of A. Knute Ream II is gratefully acknowledged.

REFERENCES

An, K.H., D.A. Gustafson, and A.C. Melton (1987), "A Model for Software Maintenance", *3rd IEEE Conference on Software Maintenance*, pp. 57-62.

Baecker, R. (1988), "Enhancing Program Readability and Comprehensibility with Tools for Program Visualization", *10th International Conference on Software Engineering*, pp. 356-366.

Banker, R.D., S.M. Datar, C.F. Kemerer, and D. Zweig (1993), "Software Complexity and Software Maintenance Costs", *Communications of the ACM 36*, 11, 81-94.

Basili, V.R. and B. Perricone (1984), "Software Errors and Complexity: An Empirical Investigation", *Communications of the ACM 27*, 1, 42-52.

Benander, B.A., N. Gorla, and A.C. Benander (1990), "An Empirical Study of the Use of the GOTO Statement", *Journal of Systems and Software 11*, 3, 217-223.

Bowen, J.B. (1983), "Software Maintenance, An Error Prone Activity", *1st IEEE Conference on Software Maintenance*, pp. 102-105.

Card, D.N. and W.W. Agresti (1988), "Measuring Software Design Complexity", *Journal of Systems and Software 8*, 3, 185-197.

Compton, B. and C. Withrow (1990), "Prediction and Control of ADA Software Defects", *Journal of Systems and Software 12*, 3, 199-207.

Conte, S.D., H.E. Dunsmore, and V.Y. Shen (1986), *Software Engineering Metrics and Models*, Benjamin–Cummings, Reading, MA.

Coupal, D. and P.N. Robillard (1990), "Factor Analysis of Source Code Metrics", *Journal of Systems and Software 12*, 3, 263-269.

Curtis, B., S.B. Sheppard, P. Milliman, M.A. Borst, and T. Love (1979), "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics", *IEEE Transactions on Software Engineering SE-5*, 2, 96-104.

Curtis, B., S.B. Sheppard, J.B. Kruesi-Bailey, and D. Boehm-Davis (1989), "Experimental Evaluation of Software Documentation Formats", *Journal of Systems and Software 9*, 2, 167-207.

Cusumano, M. and C.F. Kemerer (1990), "A Quantitative Analysis of US and Japanese Practice and Performance in Software Development", *Management Science 36*, 11, 1384-1406.

Dean, J.S. and B.P. McCune (1983), "An Informal Study of Software Maintenance Problems", *1st IEEE Conference on Software Maintenance*, pp. 137-139.

Fjelstad, R.K. and W.T. Hamlen (1983), "Application Program Maintenance Study: Report to Our Respondents", In *Tutorial on Software Maintenance*, G. Parikh and N. Zvegintzov, Eds., IEEE Computer Society Press, Los Angeles, CA, pp. 11-27.

Gibson, V.R. and J.A. Senn (1989), "System Structure and Software Maintenance Performance", *Communications of the ACM 32*, 3, 347-358.

Gill, G.K. and C.F. Kemerer (1991), "Cyclomatic Complexity Density and Software Maintenance Productivity", *IEEE Transactions on Software Engineering 17*, 12, 1284-1288.

Gorla, N., A.C. Benander, and B.A. Benander (1990), "Debugging Effort Estimation Using Software Metrics", *IEEE Transactions on Software Engineering 16*, 2, 223-231.

Gremillion, L.L. (1984), "Determinants of Program Repair Maintenance Requirements", *Communications of the ACM 27*, 8, 826-832.

Gustafson, D.A., A.Melton, and C.S. Hsieh (1985), "An Analysis of Software Changes During Maintenance and Enhancement", *2nd IEEE Conference on Software Maintenance*, pp. 92-95.

Hale, D.P. and D.A. Haworth (1988), "Software Maintenance: A Profile of Past Empirical Research", *Proceedings of the 4th IEEE Conference on Software Maintenance*, pp. 236-240.

Halstead, M. (1977), *Elements of Software Science*, Elsevier/North-Holland, New York, NY.

Harrison, W. and C. Cook (1986), "Are Deeply Nested Conditionals Less Readable?", *Journal of Systems and Software 6*, 335-341.

Henry, S. and D. Kafura (1981), "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering SE-7*, 510-518.

Jensen, H.A. and K. Vairavan (1985), "An Experimental Study of Software Metrics for Real-Time Software", *IEEE Transactions on Software Engineering SE-13*, 2, 231-234.

Kafura, D. and G.R. Reddy (1987), "The Use of Software Complexity Metrics in Software Maintenance", *IEEE Transactions on Software Engineering SE-13*, 3, 335–343.

Kemerer, C.F. and A.K. Ream (1992), "Empirical Research on Software Maintenance: 1981-1990", Working Paper 237, MIT Center for Information Systems Research.

Lehman, J.A. (1989), "An Empirical Comparison of Textual and Graphical Data Structure Documentation for COBOL Programs", *IEEE Transactions on Software Engineering SE-11*, 2, 12-26.

Letovsky, S. (1987), "Cognitive Processes in Program Comprehension", *Journal of Systems and Software 7*, 4, 325-339.

Li, H.F. and W.K. Cheung (1987), "An Empirical Study of Software Metrics", *IEEE Transactions on Software Engineering SE-13*, 6, 697-708.

Lind, R. and K. Vairavan (1989), "An Experimental Investigation of Software Metrics and their Relationship to Software Development Effort", *IEEE Transactions on Software Engineering SE-15*, 5, 649-653.

Littman, D. C., J. Pinto, S. Letovsky, and E. Soloway (1987), "Mental Models and Software Maintenance", *Journal of Systems and Software 7*, 341-355.

Lohse, J.B. and S.H. Zweben (1984), "Experimental Evaluation of Software Design Principles: An Investigation Into the Effect of Module Coupling on System Modifiability", *Journal of Systems and Software 4*, 4, 301-308.

McCabe, T.J. (1976), "A Complexity Measure", *IEEE Transactions on Software Engineering SE-2*, 4, 308-320.

Miara, R.J., J.A. Musselman, J.A. Navarro, and B. Shneiderman (1983), "Program Indentation and Comprehensibility", *Communications of the ACM 26*, 11, 861-867.

Munson, J.C. and T.M. Khoshgoftaar (1990), "Applications of a Relative Complexity Metric for Software Project Management", *Journal of Systems and Software 12*, 3, 283-291.

Oman, P. and C. Cook (1990), "Design and Code Traceability Using a PDL Metrics Tool", *Journal of Systems and Software 12*, 3, 189-98.

Oman, P.W., C.R. Cook, and M. Nanja (1989), "Effects of Programming experience in Debugging Semantic Errors", *Journal of Systems and Software 9*, 192-207.

Porter, A. A. and R. Selby (1990), "Evaluating Techniques for Generating Metric-Based Classification Trees", *Journal of Systems and Software, 12*, 3, 209-218.

Ramsey, H.R., M.E. Atwood, and J.R. Van Doren (1983), "Flowcharts Versus Program Design Languages: An Experimental Comparison", *Communications of the ACM 26*, 6, 445-449.

Robson, D.J., K.H. Bennett, B.J. Cornelius, and M. Munro (1991), "Approaches to Program Comprehension", *Journal of Systems and Software 14*, 79-84.

Rombach, H.D. (1987), "A Controlled Experiment on the Impact of Software Structure on Maintainability", *IEEE Transactions on Software Engineering SE-13*, 3, 344-354.

Schneidewind, N.F. (1987), "The State of Software Maintenance", *IEEE Transactions on Software Engineering SE-13*, 3, 303-310.

Selby, R. and V. Basili (1988), "Error Localization During Software Maintenance: Generating Hierarchical System Descriptions from the Source Code Alone", *Proceedings of the 4th IEEE Conference on Software Maintenance*, pp. 192-197.

Shen, V.Y., T.-J.Yu, S.M. Thebaut, and L.R. Paulsen (1985), "Identifying Error-Prone Software – An Empirical Study", *IEEE Transactions on Software Engineering SE-11*, 4, 317-323.

Shneiderman, B. (1982), "Control Flow and Data Structure Documentation: Two Experiments", *Communications of the ACM 25*, 1, 55-63.

Sunohara, T., A. Takano, K. Vehara, and T. Ohkawa (1981), "Program complexity measure for software development management", *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, CA, pp. 100-106.

Swanson, E.B. and C.M. Beath (1989), "Reconstructing the Systems Development Organization", *MIS Quarterly 13*, 3, 293-308.

Tenny, T. (1988), "Readability: Procedures Versus Comments", *IEEE Transactions on Software Engineering SE-14*, 9, 1271-1279.

Troy, D.A. and S.H. Zweben (1981), "Measuring the Quality of Structured Designs", *Journal of Systems and Software 2*, 113-120.

Vessey, I. and R. Weber (1983), "Some Factors Affecting Program Repair Maintenance: An Empirical Study", *Communications of the ACM 26*, 2, 128-134.

Wake, S. and S. Henry (1988), "A Model Based on Software Quality Factors Which Predicts Maintenance", *Proceedings of the 4th IEEE Conference on Software Maintenance*, pp. 382-387.

Weiser, M. (1982), "Programmers Use Slices When Debugging", *Communications of the ACM 25*, 7, 446-452.

Woodfield, S.N., H.E. Dunsmore, and V.Y. Shen (1981), "The Effect of Modularization and Comments on Program Comprehension", *5th International Conference on Software Engineering*, pp. 215-223.

Yau, S.S. and P.S. Chang (1988), "A Metric of Modifiability for Software Maintenance", *Proceedings of the 4th IEEE Conference on Software Maintenance*, pp. 374-381.

Yau, S.S. and J.S. Collofello (1985), "Design Stability Measures for Software Maintenance", *IEEE Transactions on Software Engineering, SE-11*, 9, 849-856.