











HD28  
.M414  
No 3429

**EMPIRICAL RESEARCH ON  
SOFTWARE MAINTENANCE:  
1981-1990**

**Chris F. Kemerer  
A. Knute Ream II**

**May 1992**

**CISR WP No. 237  
Sloan WP No. 3429**

**Center for Information Systems Research**

Massachusetts Institute of Technology  
Sloan School of Management  
77 Massachusetts Avenue  
Cambridge, Massachusetts, 02139



**EMPIRICAL RESEARCH ON  
SOFTWARE MAINTENANCE:  
1981-1990**

**Chris F. Kemerer  
A. Knute Ream II**

**May 1992**

**CISR WP No. 237  
Sloan WP No. 3429**

©1992 C.F. Kemerer, A.K. Ream II

**Center for Information Systems Research  
Sloan School of Management  
Massachusetts Institute of Technology**

M.I.T. LIBRARIES  
JUL 16 1992  
RECEIVED



Abstract

Despite its economic importance, the activity of software maintenance is relatively under-studied by researchers. This comprehensive survey documents that only two percent of all articles appearing in three leading journals and two refereed conferences over the past decade were devoted to empirical studies of software maintenance. The primary purpose of this paper is to document "what is known" from this research, and to suggest future avenues of research. The sixty-one articles surveyed are conveniently summarized as to major differences and similarities in a set of detailed tables. The text is used to highlight major findings and differences. Although the emphasis of the paper is on the subject matter, a section discussing appropriate research methodologies is included as a guide to researchers new to this area.

Table of Contents

**1. INTRODUCTION**

- 1.1 Why Empirical Studies of Software Maintenance?
- 1.2 Scope of the Review
- 1.3 Organization of the paper

**2. SOFTWARE COMPLEXITY MEASUREMENT RESEARCH**

- 2.1 Introduction
- 2.2 Modularity and Structure
  - 2.2.1 *Module Size*
  - 2.2.2 *Coupling*
- 2.3 Complexity Metrics
  - 2.3.1 *Relationships among Metrics and Maintenance*
  - 2.3.2 *Dimensions of Software Complexity*

**3. COMPREHENSION RESEARCH**

- 3.1 Individual Differences
- 3.2 Aids to Comprehension

**4. GENERAL MANAGEMENT ISSUES RESEARCH**

- 4.1 Causes of Maintenance Activity
- 4.2 Repair versus Replace

**5. METHODOLOGICAL ISSUES IN EMPIRICAL RESEARCH IN SOFTWARE MAINTENANCE**

- 5.1 Methodological Choice
- 5.2 Methodological Rigor

**6. CONCLUDING REMARKS**

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

General Terms: Management, Measurement, Performance.

Additional Key Words and Phrases: 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<sup>1</sup> of professional software engineers' time is spent maintaining existing software. Software maintenance represents a large and growing expense for organizations<sup>2</sup>. In addition, due to the shortage of experienced software engineers, the preponderance of maintenance work represents an opportunity cost of 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 the past decade's 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.

## 1.2 Scope of the Review

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. And, a preliminary exploration of two years of archival journals revealed a general dearth of empirical work in software maintenance. Therefore, the scope of this review was set to comprehensively examine the leading archival journals and refereed conference proceedings over the past decade. The choice of publication

---

<sup>1</sup>Various studies have noted that maintenance is estimated to comprise from 50-80% of software development activities. Some of these are summarized in [Kemerer, 1987].

<sup>2</sup>For example, it has been estimated that 60 percent of all business expenditures on computing are for maintenance of software written in COBOL alone [Freedman, 1986].

outlets included three journals and two conference proceedings. These five outlets published 3,018 papers in the time span of the survey. The three journals were:

- *Communications of the ACM (CACM)*
- *IEEE Transactions on Software Engineering (IEEE-TSE)*
- *Journal of Systems and Software (JSS)*

*Communications of the ACM* is the journal with the largest circulation of the Association of Computing Machinery, a leading professional society. Due to its wide circulation and monthly format, it provides a large number of highly visible pages within which to publish refereed articles. It has also scored highly on subjective rankings of "journal quality", which contributes to its attractiveness as a publication outlet for scholars<sup>3</sup>. *IEEE Transactions on Software Engineering* is also a well-regarded monthly journal which is focused on software engineering topics. The *Journal of Systems and Software* is another frequent source of software engineering-related articles. It currently has nine issues per year, although there are plans to expand to a monthly format.

The refereed conferences chosen were:

- *IEEE International Conference on Software Engineering*
- *IEEE Conference on Software Maintenance*

The *IEEE International Conference on Software Engineering* is a well-regarded refereed conference proceedings, and is focused on software engineering topics. The *IEEE Conference on Software Maintenance* was an obvious choice given the topic.

There are arguably other journals that could be included on such a list.<sup>4</sup> However, given that this set alone generated over 3,000 possible articles to review implied that this sample would be likely to result in finding most of the important papers in this area. In addition, while the statistics and tables included below are limited to those papers that appeared in one of those five sources, a few widely-cited papers that have appeared elsewhere that are relevant to this review have also been included in the discussion.

---

<sup>3</sup>For example, in an unpublished survey by two computer information systems researchers at New York University of the top journals ranked by computer information systems faculty, *CACM* ranked third, and *IEEE-TSE* ranked fifth. *JSS* was not included in the study and therefore was unranked [Ramesh and Stohr, 1989].

<sup>4</sup>There is a relatively new journal from Wiley called the *Journal of Software Maintenance*. It was not included in this review due its relative absence during the period surveyed, but would be a logical choice for a review spanning the next decade.



The criteria for inclusion in this set were that the paper 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, 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 1989b].

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.

The identification of articles suitable for inclusion was done manually through the review of titles and abstracts of individual articles in each publication and then a reading of the full article for those which initially appeared to be appropriate. Eighty-three articles were originally identified as candidates, and of these, sixty-one were ultimately found to meet the criteria [Ream 1991]. This approach to selecting articles, of course, leaves open the possibility that some may have been inadvertently omitted.<sup>5</sup> To reduce the probability of this type of error, a check of the selected titles was made against an existing bibliography of empirical software maintenance research that was published in the 1988 IEEE Conference on Software Maintenance [Hale and Haworth 1988]. All of the articles cited there are

---

<sup>5</sup>Some difficult conscious omissions were made as well. For example a 1982 CACM article by Elshoff and Marcotty addresses many items of interest to maintenance. However, it does not present and analyze a new set of empirical data, but rather relies on a set of constructed examples.

included in the this survey, as well as approximately forty additional articles that were not included on that list. Thus, although inadvertent omissions may remain, this compilation is believed to be representative of empirical software maintenance over the last decade.

One of the first findings from this review is the relative scarcity of empirical work in software maintenance. A total of sixty-one articles out of 3,018 were found to meet the criteria set out, approximately 2% of the total (See Table 1). This scarcity of research confirms the earlier but less systematic observation of Schneidewind. Even allowing for inadvertent omissions, the percentage of effort devoted to this type of work in 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 concern practitioners, since little effort is being devoted to discovering new knowledge about an activity of considerable economic importance.

A related concern may be that there is no clear trend for more work in this area. Figure 1 shows both the raw frequency counts by year plus a cumulative average. The raw counts may be somewhat misleading, given the irregular publication cycles, particularly of the IEEE Conference on Software Maintenance. However, there is no strong trend in the average, which suggests that Schneidewind's call for more work in this area has not been acted upon.

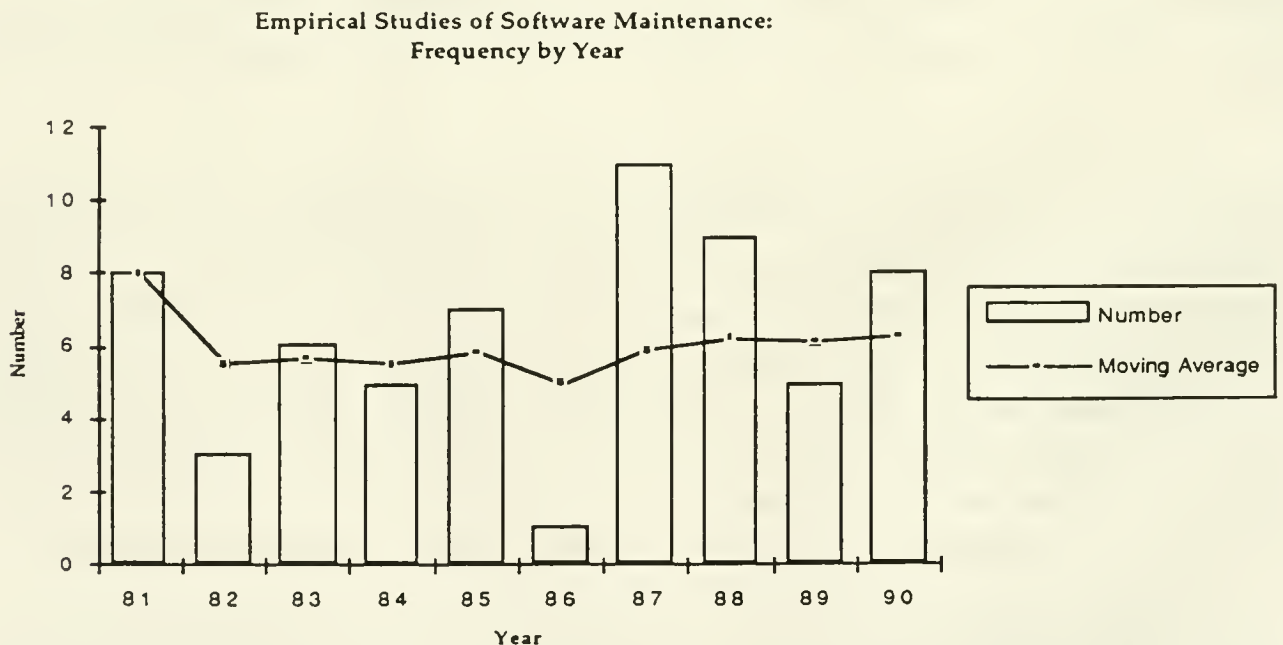


Figure 1: Frequency of articles by year

	1981	1982	1983	1984	1985	1986	1987	1988	1989	1990	TOTALS (vs possible)
Communications of the ACM	1	2	4	2	0	0	0	0	1	1	11 (1037)
IEEE Conference on Software Maintenance	0	0	2	0	3	0	5	6	0	0	16 (206)
IEEE International Conference on Software Engineering	3	0	0	0	0	0	0	1	0	0	4 (392)
IEEE Transactions on Software Engineering	1	0	0	2	4	0	4	1	2	1	15 (1074)
Journal of Systems and Software	2	0	0	1	0	1	2	1	2	6	15 (309)
TOTALS: (by year)	7	2	6	5	7	1	11	9	5	8	61 (3018)

**Table 1: Frequency of empirical software maintenance articles by year and by publication outlet**

### 1.3 Organization of the paper

Despite the small percentage of articles discovered, 61 articles form an amount of material sufficiently large that some structure needs to be imposed in order to properly convey the contributions made by each study. The approach adopted here was 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 merited additional discussion. The papers are organized under three broad areas, with one area subdivided into two more focused parts. These areas, with the number of articles in parentheses, are:

- Software Complexity Measurement
  - Modularity and Structure Metrics (15)
  - Other Complexity metrics (16)
- Comprehension (15)
- General Maintenance Management (15)

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
- 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. All other topics are summarized in section 4, "General Maintenance Management" and section 5 provides a summary and



discussion of some meta-issues highlighted by this review of the previous decade's worth of software maintenance research. A final section provides some concluding remarks.

## 2. SOFTWARE COMPLEXITY MEASUREMENT RESEARCH

### 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 2.)

There is a significant amount of other work in software complexity metrics area, for example, volume measures such as those of Halstead's Software Science. (See Table 3.) Work in this area often overlaps the work in modularity and structure, with many articles reporting results for both. Given the large amount of work in measurement, an attempt has been made to place an individual article into either Table 2 or Table 3, 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.

Table 2: Modularity and Structure

Author	Publication	Methodology	Data	Dependent Var's	Statistical Tests	Reported Results
Troy and Zweben (1981)	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 structured 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

Table 2: Modularity and Structure

Author	Publication	Methodology	Data	Dependent Var's	Statistical Tests	Reported Results
Rombach (1987)	IEEE Transactions on Software Engineering (also: 1985 IEEE Conf. on SW Maint)	lab experiment	6 grad student programmers, 12 SW systems (1.5-15.2KLOC) 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.



Table 3: 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 that highly nested statements are unlikely to change.
Jensen and Vairavan (1985)	IEEE Transactions on Software Engineering	field experiment	202 PASCAL software modules	correlation 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 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.

Table 3: Complexity Metrics

Author	Publication	Methodology	Data	Dependent Var's	Statistical Tests	Reported Results
Wake and Henry (1988)	IEEE Conference on Software Maintenance	field experiment	commercial software product 15KLOC of C	number of lines of code changed	Pearson correlation, 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 routines in various languages)	relative complexity	principal components 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, Benander, and Benander (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 regression	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.

## 2.2 Modularity and Structure

### 2.2.1 Module Size

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 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.* 1992].

### 2.2.2 Coupling

Another important issue within this set of literature is the effect of module coupling on performance. 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 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<sup>6</sup> 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 of that 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)<sup>7</sup> 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 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

---

<sup>6</sup>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, p. 303].

<sup>7</sup>These are referred to as "steps" in their paper, as this is the standard nomenclature in Japan. (See, for example, [Cusumano and Kemerer 1990].)



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. Clearly, at least one aspect of complexity is represented by the simple size metric SLOC.

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. This may obviate their use if it is believed to be difficult to collect or implement use of these metrics within an organization.

### *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

The single critical factor that differentiates software maintenance from new software development is the software engineer's need to interact with existing software and documentation. Therefore it is not surprising that a significant amount of software maintenance research has focused on the issue of comprehension. 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 affects 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. Fjelstad and Hamlen reported back in the late 1970's their belief 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]. (See Table 4.) 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 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 for 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.* videotaped 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 this approach may be infeasible [Robson *et al.* 1991].



**Table 4: 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/CSE 1981)	lab experiment	21 experience 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 given 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.
Ransney et al. (1983)	Communications of the ACM	lab experiment	20 grad 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.

Table 4: Comprehension

Author	Publication	Methodology	Data	Dependent Var's	Statistical Tests	Reported Results
Litman 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 documentation (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.

Letovsky videotaped 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 statistically 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. GENERAL MANAGEMENT ISSUES RESEARCH

While the research in the preceding two sections tends to be centered on narrowly defined research questions, the work that is grouped together here centers on research questions that are higher level and more general in nature. (See Table 5.) The unit of analysis in these studies is more typically at the project or system level, as opposed to the the work in the previous sections which was much more focused at the program or module level. Therefore, while all maintenance research tends to have implications for management, the work reviewed in this section generates conclusions that typically require higher level management intervention if the recommendations are to be successfully implemented.

The higher level unit of analysis is also reflected in the fact that a much higher percentage of the studies reviewed in this section do not report statistical test results, but tend to rely more on descriptive data. This difference is, of course, related in that a larger unit of analysis generally results in a smaller sample size which may be less amenable to statistical analysis.

Two main streams of research are present in this work. The first focuses on the causes of maintenance work, and seeks to prevent or reduce the need for maintenance. The second focuses on the 'repair vs. replace' question, seeking to determine whether it is more cost effective to maintain an existing piece of software or to simply write a new program to replace it.

### 4.1 Causes of Maintenance Activity

A survey of DPMA members by Lientz and Swanson laid the groundwork for much later work in software maintenance [Lientz and Swanson 1980; Lientz and Swanson 1981]. They present a typology of maintenance consisting of *corrective* (repairs), *adaptive* (change accommodation) and *perfective* (enhancements), which has since gone on to become the standard terminology in this area<sup>8</sup>. The approximate distribution of maintenance work was that more than half was perfective, approximately one-quarter was adaptive, and the remainder was corrective. Their survey respondents reported that user problems, specifically lack of user knowledge, was believed to be a critical source of maintenance activity.

---

<sup>8</sup>This typology was first presented by Swanson in "The Dimensions of Maintenance" in *Proceedings of the Second International Conference on Software Engineering*. 1976, pp. 492-497.

Table 5: General Management Related Issues

Author	Publication	Methodology	Data	Dependent Var's	Statistical Tests	Reported Results
Lientz and Swanson (1981)	Communications of the ACM	field study	487 DP managers	effort	Pearson correlation, ANOVA, factor analysis with rotation	Determined that maintenance consumed about half of IS department's staff time. Six problem areas were identified. Lack of user knowledge was a leading source of reported problems.
Guimaraes (1983)	Communications of the ACM	field study	43 companies surveyed, 5 companies studied in depth	maintenance expenditures	ANOVA, Pearson correlation	Maintenance expenditures tend to increase with program age. Redeveloped programs (functional replacement) tend to require less maintenance expenditures than the originals. Poor documentation significantly increases maintenance expenditures. Application programs written in assembly language require higher maintenance expenditures than their higher level counterparts.
Wiener-Ehrlich et al. (1984)	IEEE Transactions on Software Engineering	field study	four projects at Bankers Trust	man months of effort	chi square test	Based on effort requirements for the development phase, the Rayleigh curve seems to work to predict effort requirements for corrective maintenance, but significantly underestimates effort when adaptive and perfective maintenance are included.
Lanegan and Grasso (1984)	IEEE Transactions on Software Engineering	field experiment	5000 COBOL programs	amount of reusable code	none	Significant portions of the code were proven to be reusable, and subsequent evaluation of the actual effects of reuse led to estimates of an increase in productivity of up to 50%. Reuse also leads to simplification, which should be of major benefit to maintenance efforts by aiding programmer comprehension.
Chong Hok Yuen (1985)	IEEE Conference on Software Maintenance	field study	mainframe operating system (3 MLOC) monitored over 19 months	bug reports / bug responses	auto-correlation function, chi-square, contingency coefficient	A nineteen month longitudinal study suggested that significant difference existed in maintainers' performance levels.
Weiss and Basili (1985)	IEEE Transactions on Software Engineering	field study	3 projects ranging from 51 to 85 KLOC	distribution of causes of changes and errors	none	Provides detailed data regarding causes of modifications and errors. Unplanned design errors were most common.
Blum (1987)	IEEE Conference on Software Maintenance	field study	5 systems consisting of 8K MUMPS programs	number of generated programs	none	Detailed longitudinal data of 6 years of system updates to a set of 5 medical systems. Systems proved relatively stable and were able to be maintained by a relatively small staff believed due, in part, to the application generator used.



Table 5: General Management Related Issues

Author	Publication	Methodology	Data	Dependent Var's	Statistical Tests	Reported Results
Bendifallah and Scacchi (1987)	IEEE Transactions on Software Engineering	field study	evolution of software systems in two university settings	relationship between workplace arrangements and maintenance work	none	Maintenance activities involve non-programming tasks in addition to the programming tasks. Thus, further emphasis on tools cannot alone solve the problems of software maintenance.
Card et al. (1987)	IEEE Conference on Software Maintenance	field study	8 releases of a large software project	effort, error rate, "LOC measures"	Pearson correlation	It costs as much to delete a line of code as it does to write a new one. A count of new units and problem fixes accurately predicts maintenance cost. Problem fixes require 10 times the effort of developing a new unit.
Rombach and Basili (1987)	IEEE Conference on Software Maintenance	field study	2 systems at a major computing company	28 different data items	none	A preliminary report, but suggests that many characteristics of the maintenance process can be made visible by analyzing even a small set of available data.
Wu (1987)	IEEE Conference on Software Maintenance	field study	information systems in a "top ranked" international bank	maintenance person hours, maintenance costs	Pearson correlation	Quality assurance audits during the development phase do reduce maintenance effort (and thus cost), especially when conducted during the "definition" phase of development.
Lin and Gustafson (1988)	IEEE Conference on Software Maintenance	field experiment	53 versions (total) of 13 COBOL programs	changes, by statement type	none	Six distinct types of maintenance activities were identified. (corrective, adaptive, retrenchment, retrieving, pretty printing, and documentation)
Lindberg et al. (1988)	IEEE Conference on Software Maintenance	lab experiment	6 programmers	effort required to implement a change	none	No difference was indicated between incremental and top-down delivery.
Chong Hok Yuen (1988)	IEEE Conference on Software Maintenance	field study	3 years of bug notifications related to a large operating system, spanning 5 releases	number of notices (per week, per release)	Runs test, Turning points test, Phase length test	In large or complex systems, maintenance data does not appear to exhibit patterns. This paper shows that there are definite patterns common to similar system components, but they are masked when the data is aggregated for global study.
Swanson and Beath (1990)	Communications of the ACM	field study	12 organizations	distribution of staffing roles	none	Life cycle based organization (development vs. maintenance) forms seem to have more strengths than departmentalization by work type (systems analysis vs. programming).

Lin and Gustafson further investigated the distribution of work by examining before and after versions of two COBOL systems [Lin and Gustafson 1988]. The combined percentage of perfective and corrective maintenance activity was greater than seventy percent in one case and greater than ninety percent in the other. Adaptive was only approximately ten percent, and a number of new categories (e.g., adding and deleting comments) all represented small percentages of the work.

Weiss and Basili did a detailed investigation of the change data from three systems at the Software Engineering Laboratory [Weiss and Basili 1985]. They found that approximately forty percent of changes were to correct errors. Their data did not support some conventional wisdom in software engineering; for example, interfaces did not appear to be particularly problematic, and most corrections were small changes in only one location.

Additional work in this area would be useful in better understanding how maintainers actually spend their time. In particular, it may be time to develop a finer-grained taxonomy that further develops the three types of activities first proposed by Swanson. Beyond this documentation of effort distribution, analysis linking patterns in the distribution of maintenance work could suggest improvements in the initial development process that would reduce later expenditures on maintenance. For example, lower than average amounts of corrective maintenance and/or easier (less expensive) adaptive and perfective maintenance might be associated with systems developed with certain modern development practices. Systems with higher levels of software re-use may be associated with lower levels of corrective maintenance.

#### 4.2 Repair versus Replace

One relatively unsettled question is how the distribution of work may change over time as systems age. Guimaraes observed that successive program changes tend to complicate the logical flows of the program and to render program documentation obsolete, thus increasing maintenance expenditures [Guimaraes 1983]. Lientz and Swanson [Lientz and Swanson 1981] agree that maintenance costs increase with program age, but offer results that suggest that the increases may be avoidable through managerial action: *"Though system size and age are seen to be strongly associated with the problems of maintenance, this association was shown in subsequent analysis to be explainable in terms of other, intervening variables, viz. magnitude and allocation of maintenance effort and the relative development experience of maintainers of the system."*

If the effects of age on software were better understood, then this could offer insight into the question of when to replace rather than repair (maintain) a given software component. Most of the data collected so far suggest that modification is more expensive than is commonly believed, and that the development cost savings of using modified modules may pale in comparison to the later costs of maintaining the resulting system [Bowen 1983] [Basili and Perricone 1984].<sup>9</sup>

Bowen analyzed error data from a large (6000 module) Hughes air defense project and determined that a composition of a balanced mixture of new and lifted (modified from existing code) software (e.g. 35/65 to 75/25) is nearly four times as error-prone as a composition of extremely unbalanced mixtures of new/lifted software (e.g. 15/85 or 90/10). This implies that if one is planning to utilize pieces of an existing system, one should either use it sparingly in a new system, or use it nearly completely intact. If large scale modifications are planned, it seems much more efficient to design from scratch to avoid the prohibitive maintenance costs of problem fixes associated with reuse. Supporting this view is the study by Card *et al.* where problem fixes required ten times the effort of developing new code [Card *et al.* 1987].<sup>10</sup>

While they did not specify optimal blends of new and modified modules in the construction of new systems, Basili and Perricone concluded that adapted modules taken from other systems were more expensive to maintain [Basili and Perricone 1984]. One factor that may have contributed to this result is that they also determined that most of the errors in the systems they analyzed were due to incorrect or misinterpreted functional specifications, and the single largest error types were those involving interfaces. Modules borrowed from other systems are likely to be less comprehensible to programmers than the newly designed code, and thus would be especially prone to these types of errors.

In seeming contrast to the above, Lanergan and Grasso offer evidence of a large scale success in the reuse of software components [Lanergan and Grasso 1984]. They examined 5000 source COBOL programs at Raytheon, and identified redundant sections of code that were prime candidates for standardization. Subsequent evaluation of the actual effects of using these standardized functional modules led to estimates of an increase in productivity of up to 50%. The examples cited were simple, however, comprising routines

---

<sup>9</sup>Although not all data collected on this topic are in agreement. In a study of 65 COBOL maintenance projects it was found that the costs associated with modified lines of code were approximately equal to new lines [Banker, *et al.*, 1987]

<sup>10</sup>This is also related to some interesting theoretical work done by Gode, *et al.*, whose model results suggest, among other propositions, the somewhat surprising conclusion that the optimal time within which to replace larger systems is shorter than that for smaller systems [Gode, *et al.*, 1990].



to perform date conversions, part number validations, or data field edits. Reuse of relatively atomic functions such as these has proved effective, but the advantages may not carry over quite as well to modules with more complex functions and interfaces.

Further research into when the benefits of reusability are offset by the cost to modify seem warranted, as well as more longitudinal studies that document how systems evolve.

## 5. METHODOLOGICAL ISSUES IN EMPIRICAL RESEARCH IN SOFTWARE MAINTENANCE

One advantage of a review that examines so many years of research is that it permits some observations to be made about meta-issues. One issue that has already been raised is the sheer dearth of research in this area. A second such issue are methodological concerns in the research. Two main topics merit discussion here, the choice of methodologies and the care with which research is conducted.

### 5.1 Methodological Choice

Proponents of alternative research methodologies seem somewhat inclined to criticize other approaches rather than simply benefiting from assimilating those findings into their own work. A common division is between those who conduct field research (typically field studies rather than field experiments) and those who conduct experiments (typically laboratory experiments rather than field experiments). The experimentalists emphasize the need to find causes of behaviors and often complain about the lack of a theoretical base in some field studies. For example, Soloway and Ehrlich, at the end of an article describing their experiments, note *"More importantly, our approach is to provide explanations (emphasis in original) for why a program may be complex and thus hard to comprehend. Towards this end we have attempted to articulate the programming knowledge that programmers have and use. Thus, our intent is to move beyond correlations (emphasis in original) between programmer performance and surface complexity as measured by Halstead metrics, lines of code, etc, to a more principled, cognitive explanation."* [Soloway and Ehrlich 1984].

On the other side, field researchers often complain about the lack of external validity of most lab experiments which typically use student programmers and small programs. For example, Conte *et al.* note *"The results from controlled experiments which will be discussed later, are usually limited by economic constraints to small projects by individual programmers, and are usually performed only in universities. Such results are useful in providing insights to certain parameters of the programming process, but are not normally*

generalizable to team programming and large projects, which are common in industry." [Conte *et al.* 1986].

Both of these statements, while true, emphasize the shortcomings of alternative research methodologies without conveying the notions (1) that difficult research problems such as those being investigated in this research are likely to benefit from attack by dissimilar methods, and (2) that given the current shortage of research in this area, almost all published research is providing positive marginal contribution. It would seem appropriate for researchers to attempt to assimilate the findings from the other streams into their own work so that all groups would move ahead. Only a very small number of field experiments have been reported, and some of these have been criticized as not being done as well as they might have [General Services Administration 1987; Zvegintzov 1988]. As it stands now, a review of Tables 2, 3, and 4 reveals that problems and methodologies are tightly linked, e.g., complexity metrics work is almost entirely field study based and comprehension work is almost entirely laboratory experiments. While to a certain degree this bias is natural and appropriate, given the topics studied, over-reliance on a subset of research tools may hinder progress. What may be required is collaboration among maintenance researchers who reflect different traditions and who possess complementary research skill sets.

## 5.2 Methodological Rigor

Empirical work in software engineering in general (not just maintenance) has been sometimes criticized for lack of methodological rigor, e.g. [Kearney *et al.* 1986]. Work in this area suffers from a number of handicaps owing to the difficulty of the research problem -- the large number of potential factors to model, the absence of standard definitions for dependent and independent variables, and the lack of large and/or readily available data sets with which to analyze.

Unfortunately, these limitations are sometimes overlooked, or at least not acknowledged, by researchers. A recent summary of a set of thirteen general criticisms has been provided by MacDonell, where he notes deficiencies in such areas as experimental method and design, data collection, and statistical analysis and interpretation [MacDonell 1991].

One particular point of MacDonell's that is borne out by the data collected in this review and is highlighted in the tables is the (over)-reliance on Pearson correlation [MacDonell 1991, pp. 146-147]. One concern is the sometimes casual manner in which researchers move from interpreting what are often exploratory correlation results with

causation. Kearney *et al.* note "When large numbers of differing 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 non-existent relationship." (page 1048) This concern seems worthy of repeating, especially in light of a recent trend observed in the tables towards greater use of exploratory factor analysis in software engineering maintenance research.

A more general concern is the extensive use of parametric statistical methods, such as Pearson correlation, whose proper use includes an understanding of the method's distributional assumptions. Shepperd provides a very relevant example of where such assumptions are violated - the use of the number of errors as a dependent variable [Shepperd 1988]. Clearly, this can never be negative, and therefore at best this distribution is truncated normal, yet such concerns are rarely acknowledged by authors. Two exceptions from this review worthy of emulation by other researchers are acknowledgements by Curtis *et al.* and Woodfield, *et al.*:

*"In using ANOVA, we assume that the values of the dependent variable are normally distributed. Unfortunately, this is typically not the case with response-time measures. For most response-time measures, the variance is proportional to the mean, since many of the values are near zero and the distribution is positively skewed. For all the analysis reported in experiment 1, a logarithmic transformation was applied to the response time to attenuate the influence of extreme scores and produce a more normal distribution..."* [Curtis *et al.* 1989].

*"The most common correlation measure is the Pearson product-moment correlation coefficient, which requires that data be from interval scales with underlying normal distributions, with the sets of data being correlated having nearly equal variance...some models yield outlier estimates that do not meet the normal distribution assumption. Thus, we also use the Spearman rank correlation coefficient to determine how well estimates of programming times relate to actual programming times."* [Woodfield *et al.* 1981b]

Despite the ease in doing so, such acknowledgements are rare in this literature. In general, for much of the empirical research in software maintenance it would seem that greater use of non-parametric (distribution free) statistical tests would be appropriate.



## 6. 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 four broad areas of coverage: (1) software modularity and structure, (2) general software complexity metrics, (3) software comprehension, and (4) general management issues. This section focuses on discussing suggestions for future research, and these recommendations are summarized in Table 6 which appears at 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.<sup>11</sup> 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

---

<sup>11</sup>A pilot study in this regard is the work in cyclomatic complexity density [Gill and Kemerer, 1991].

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 terms of higher level managerial issues two foci were noted, the causes of maintenance and the question of repair versus replacement. The data on the causes of maintenance are somewhat mixed, and do not always represent consistent or sufficiently detailed definitions. It will be extremely difficult to evaluate the impact of improved practices, in design or elsewhere, if accurate tracking of the scope and origin of maintenance requests cannot be done. More work needs to be done to track maintenance work in practice, in part to support the aforementioned need for more longitudinal data.

The repair/replace issue is often discussed, but is difficult to research. Some research suggests that repair is more expensive than new development, but research in the software reuse literature suggests that significant savings can be achieved through code reuse. Savings depend on the degree to which the reused code needs to be modified, but little is known about even how to measure this phenomenon.

In terms of methodological issues greater emphasis should be placed on using multiple, diverse research methods to address the large number of remaining research issues. Empirical researchers in software maintenance, particularly new ones, are reminded by a number of authors about using appropriate caution in borrowing techniques, particularly statistical tools, from other disciplines, without examining the assumptions necessary to appropriately apply them.

It is important to try and step back from the existing studies to attempt to determine what is missing or at least neglected. One common concern about documentation not addressed by laboratory studies is that in practice maintainers often do not use it at all,



regardless of format, perhaps because they do not trust that it has been kept consistent with the existing system. Researchers and vendors in new systems development need to address this issue by making automatic generation and update of documentation of feature of their new tools, lest the potential comprehension gains of proper formatting of such documentation be wasted.

An area of research that is conspicuous by its absence is work on the organizational aspects of software maintenance. Work on comprehension focuses narrowly on an individual's approach to a piece of code and work on complexity metrics tends to ignore the maintainer completely. In practice there is considerable influence from the organizational environment in terms of the presumed undesirability of maintenance work and the subsequent likely effects on morale and performance.<sup>12</sup> While several academic studies surveyed here mention this in passing, with the exception of recent work by Swanson and Beath, none address the organizational component [Swanson and Beath 1989a; Swanson and Beath 1990]. It seems likely that the organizational effects on performance are at least as great as those that have been studied in detail, such as work on documentation formats.

For example, is poor performance in maintenance a result of low morale of the maintainers? Is maintenance's low occupational status in the software engineering community a function of the common practice of assigning relatively inexperienced staff members to this role? And, in turn, how does the use of these junior staff members contribute to poor performance? Do the benefits of assigning software engineers with higher levels of experience to maintenance outweigh the possibly increased cost of turnover? These are difficult research questions to operationalize and test in the field, and what would be appropriate are collaborative research projects between organizationally-oriented researchers and more traditional software engineering researchers, where the respective interests and skills of each could lead to some very interesting and carefully researched results.

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.

---

<sup>12</sup>Schneidewind likens working in maintenance to "having bad breath" [Schneidewind 1987].

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. 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.

**Table 6: 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

*General management issues*

1. More work on a finer grained taxonomy of maintenance activities
2. More work on linking maintenance tasks to earlier lifecycle phase activities
3. More work on documenting modification costs and relationship with reuse
4. More work on organizational issues, including morale and turnover

## BIBLIOGRAPHY

- An, K. H., D. A. Gustafson and A. C. Melton, "A Model for Software Maintenance", *3rd IEEE Conference on Software Maintenance*, pp. 57-62, 1987.
- Baecker, R., "Enhancing Program Readability and Comprehensibility with Tools for Program Visualization", *10th International Conference on Software Engineering*, pp. 356-366, 1988.
- Banker, R. D., S. M. Datar and C. F. Kemerer, "Factors Affecting Software Maintenance Productivity: An Exploratory Study", *Proceedings of the 8th International Conference on Information Systems*, Pittsburgh, Pennsylvania, pp. 160-175, 1987.
- Banker, R. D., S. M. Datar, C. F. Kemerer and D. Zweig, "Software Complexity and Software Maintenance Costs", MIT Sloan School Working Paper #3155-90, (January 1992).
- Basili, V. R., "Quantitative software complexity models: A panel summary", pp. 232-233 in *Tutorial on Models and Methods for Software Management and Engineering*, Basili, V. R. (ed.), IEEE Computer Society Press, Los Alamitos, CA, (1980).
- Basili, V. R. and B. Perricone, "Software Errors and Complexity: An Empirical Investigation", *Communications of the ACM*, 27, (1): 42-52, (January 1984).
- Benander, B. A., N. Gorla and A. C. Benander, "An Empirical Study of the Use of the GOTO Statement", *Journal of Systems and Software*, 11, (3): 217-223, (1990).
- Bendifallah, S. and W. Scacchi, "Understanding Software Maintenance Work", *IEEE Transactions on Software Engineering*, SE-13, (3): 311-323, (March 1987).
- Blum, B. I., "Evolution With System Sculpture: Some Empirical Results", *3rd IEEE Conference on Software Maintenance*, pp. 45-56, 1987.
- Bowen, J. B., "Software Maintenance, An Error Prone Activity", *1st IEEE Conference on Software Maintenance*, pp. 102-105, 1983.
- Card, D. N. and W. W. Agresti, "Measuring Software Design Complexity", *Journal of Systems and Software*, 8, (3): 185-197, (June 1988).
- Card, D. N., D. V. Cotnoir and C. E. Goorevitch, "Managing Software Maintenance and Cost", *3rd IEEE Conference on Software Maintenance*, pp. 145-152, 1987.
- Chong Hok Yuen, C. K. S., "An Empirical Approach to the Study of Errors in Large Software Under Maintenance", *2nd IEEE Conference on Software Maintenance*, pp. 96-105, 1985.



- Chong Hok Yuen, C. K. S., "On Analyzing Maintenance Process Data at the Global and Detailed Levels: A Case Study", *Proceedings of the 4th IEEE Conference on Software Maintenance*, pp. 248-255, 1988.
- Compton, B. and C. Withrow, "Prediction and Control of ADA Software Defects", *Journal of Systems and Software*, 12, (3): 199-207, (July 1990).
- Conte, S. D., H. E. Dunsmore and V. Y. Shen, *Software Engineering Metrics and Models*, Benjamin-Cummings, Reading, MA, (1986).
- Coupal, D. and P. N. Robillard, "Factor Analysis of Source Code Metrics", *Journal of Systems and Software*, 12, (3): 263-269, (July 1990).
- Curtis, B., S. B. Sheppard, P. Milliman, M. A. Borst and T. Love, "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics", *IEEE Transactions on Software Engineering*, SE-5, (2): 96-104, (1979).
- Curtis, B. S., E. Sheppard, J. B. Kruesi-Bailey and D. Boehm-Davis, "Experimental Evaluation of Software Documentation Formats", *Journal of Systems and Software*, 9, (2): 167-207, (February 1989).
- Cusumano, M. and C. F. Kemerer, "A Quantitative Analysis of US and Japanese Practice and Performance in Software Development", *Management Science*, 36, (11): 1384-1406, (November 1990).
- Dean, J. S. and B. P. McCune, "An Informal Study of Software Maintenance Problems", *1st IEEE Conference on Software Maintenance*, pp. 137-139, 1983.
- Fjelstad, R. K. and W. T. Hamlen, "Application Program Maintenance Study: Report to Our Respondents", pp. 11-27 in *Tutorial on Software Maintenance*, Parikh, G. and N. Zvegintzov (ed.), IEEE Computer Society Press, Los Angeles, CA, (1983).
- Freedman, D. H., "Programming without Tears", *High Technology*, 6, (4): 38-45, (April 1986).
- General Services Administration, U. S., "Parallel Test and Evaluation of a COBOL Restructuring Tool", Federal Software Management Support Center Office of Software Development and Information Technology Report (September 1987).
- Gibson, V. R. and J. A. Senn, "System Structure and Software Maintenance Performance", *Communications of the ACM*, 32, (3): 347-358, (March 1989).
- Gill, G. K. and C. F. Kemerer, "Cyclomatic Complexity Density and Software Maintenance Productivity", *IEEE Transactions on Software Engineering*, 17, (12): 1284-1288, (December 1991).

- Gode, D. K., A. Barua and T. Mukhopadhyay, "On the Economics of the Software Replacement Problem", *Proceedings of the 11th International Conference on Information Systems*, Copenhagen, Denmark, pp. 159-170, 1990.
- Gorla, N., A. C. Benander and B. A. Benander, "Debugging Effort Estimation Using Software Metrics", *IEEE Transactions on Software Engineering*, 16, ( 2): 223-231, (February 1990).
- Gremillion, L. L., "Determinants of Program Repair Maintenance Requirements", *Communications of the ACM*, 27, (8): 826-832, (August 1984).
- Guimaraes, T., "Managing Application Program Maintenance Expenditures", *Communications of the ACM*, 26, (10): 739-746, (October 1983).
- Gustafson, D. A., A. Melton and C. S. Hsieh, "An Analysis of Software Changes During Maintenance and Enhancement", *2nd IEEE Conference on Software Maintenance*, pp. 92-95, 1985.
- Hale, D. P. and D. A. Haworth, "Software Maintenance: A Profile of Past Empirical Research", *Proceedings of the 4th IEEE Conference on Software Maintenance*, pp. 236-240, 1988.
- Halstead, M., *Elements of Software Science*, Elsevier North-Holland, New York, NY, (1977).
- Harrison, W. and C. Cook, "Are Deeply Nested Conditionals Less Readable?", *Journal of Systems and Software*, 6, 335-341, (1986).
- Henry, S. and D. Kafura, "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering*, SE-7, 510-518, (September 1981).
- Jensen, H. A. and K. Vairavan, "An Experimental Study of Software Metrics for Real-Time Software", *IEEE Transactions on Software Engineering*, SE-13, (2): 231-234, (February 1985).
- Kafura, D. and G. R. Reddy, "The Use of Software Complexity Metrics in Software Maintenance", *IEEE Transactions on Software Engineering*, SE-13, ( 3): (March 1987).
- Kearney, J. K., R. L. Sedlmeyer, W. B. Thompson, M. A. Gray and M. A. Adler, "Software Complexity Measurement", *Communications of the ACM*, 29, (11): 1044-1050, (1986).
- Kemerer, C. F., *Measurement of Software Development Productivity*, unpublished Carnegie Mellon University Ph.D. thesis, (1987).
- Korson, T. D. and V. K. Vaishnavi, "An Empirical Study of the Effects of Modularity on Program Modifiability", pp. 168-186 in *Empirical Studies of Programmers*, Soloway, E. and S. Iyengar (ed.), Ablex Publishing Co., (1986).

- Lanergan, R. G. and C. A. Grasso, "Software Engineering With Reusable Designs and Code", *IEEE Transactions on Software Engineering*, SE-10, (5): 498-501, (September 1984).
- Lehman, J. A., "An Empirical Comparison of Textual and Graphical Data Structure Documentation for COBOL Programs", *IEEE Transactions on Software Engineering*, 11, (2): 12-26, (September 1989).
- Letovsky, S., "Cognitive Processes in Program Comprehension", *Journal of Systems and Software*, 7, (4): 325-339, (December 1987).
- Li, H. F. and W. K. Cheung, "An Empirical Study of Software Metrics", *IEEE Transactions on Software Engineering*, SE-13, (6): 697-708, (June 1987).
- Lientz, B. and E. B. Swanson, *Software Maintenance Management*, Addison-Wesley, Reading, MA, (1980).
- Lientz, B. P. and E. B. Swanson, "Problems in Application Software Maintenance", *Communications of the ACM*, 24, (11): 31-37, (November 1981).
- Lin, I.-H. and D. A. Gustafson, "Classifying Software Maintenance", *Proceedings of the 4th IEEE Conference on Software Maintenance*, pp. 241-247, 1988.
- Lind, R. and K. Vairavan, "An Experimental Investigation of Software Metrics and their Relationship to Software Development Effort", *IEEE Transactions on Software Engineering*, 15, (5): 649-653, (May 1989).
- Lindberg, M. V., H. Pei and R. Bond, "A Study of Requirements Change Effects on Incrementally Delivered Systems", *Proceedings of the 4th IEEE Conference on Software Maintenance*, pp. 204-211, 1988.
- Littman, D. C., J. Pinto, S. Letovsky and E. Soloway, "Mental Models and Software Maintenance", *Journal of Systems and Software*, 7, 341-355, (1987).
- Lohse, J. B. and S. H. Zweben, "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, (November 1984).
- MacDonell, S. G., "Rigor in Software Complexity Measurement Experimentation", *Journal of Systems and Software*, 16, 141-149, (1991).
- McCabe, T. J., "A Complexity Measure", *IEEE Transactions on Software Engineering*, SE-2, (4): 308-320, (1976).
- Miara, R. J., J. A. Musselman, J. A. Navarro and B. Shneiderman, "Program Indentation and Comprehensibility", *Communications of the ACM*, 26, (11): 861-867, (November 1983).



- Munson, J. C. and T. M. Khoshgoftaar, "Applications of a Relative Complexity Metric for Software Project Management", *Journal of Systems and Software*, 12, (3): 283-291, (July 1990).
- Oman, P. W., C. R. Cook and M. Nanja, "Effects of Programming experience in Debugging Semantic Errors", *Journal of Systems and Software*, 9, 192-207, (1989).
- Porter, A. A. and R. Selby, "Evaluating Techniques for Generating Metric-Based Classification Trees", *Journal of Systems and Software*, 12, (3): 209-218, (July 1990).
- Ramesh, B. and E. A. Stohr, "Survey of Journals and Proceedings in Computer Science and MIS", New York University working paper (May 18, 1989).
- Ramsey, H. R., M. E. Atwood and J. R. Van Doren, "Flowcharts Versus Program Design Languages: An Experimental Comparison", *Communications of the ACM*, 26, (6, June): 445-449, (1983).
- Ream, A. K., *A Survey and Review of Recent Empirical Research on Software Maintenance*, unpublished Massachusetts Institute of Technology B.S. thesis, (1991).
- Robson, D. J., K. H. Bennett, B. J. Cornelius and M. Munro, "Approaches to Program Comprehension", *Journal of Systems and Software*, 14, 79-84, (1991).
- Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability", *IEEE Transactions on Software Engineering*, SE-13, (3): 344-354, (March 1987).
- Rombach, H. D. and V. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study", *3rd IEEE Conference on Software Maintenance*, pp. 134-144, 1987.
- Schneidewind, N. F., "The State of Software Maintenance", *IEEE Transactions on Software Engineering*, SE-13, (3): 303-310, (March 1987).
- Selby, R. and V. Basili, "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, 1988.
- Shen, V. Y., T.-J. Yu, S. M. Thebaut and L. R. Paulsen, "Identifying Error-Prone Software - An Empirical Study", *IEEE Transactions on Software Engineering*, SE-11, (4): 317-323, (1985).
- Shepperd, M., "A critique of cyclomatic complexity as a software metric", *Software Engineering Journal*, 3, (2): 30-36, (March 1988).
- Shneiderman, B., "Control Flow and Data Structure Documentation: Two Experiments", *Communications of the ACM*, 25, (1): 55-63, (1982).



- Soloway, E. and K. Ehrlich, "Empirical Studies of Programming Knowledge", *IEEE Transactions on Software Engineering*, SE-10, (5): 595-609, (September 1984).
- Sunohara, T., A. Takano, K. Vehara and T. Ohkawa, "Program complexity measure for software development management", *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, CA, pp. 100-106, 1981.
- Swanson, E. B., "The Dimensions of Maintenance", *Proceedings of the Second International Conference on Software Engineering.*, pp. 492-497, 1976.
- Swanson, E. B. and C. M. Beath, *Maintaining Information Systems in Organizations*, John Wiley & Sons, New York, (1989a).
- Swanson, E. B. and C. M. Beath, "Reconstructing the Systems Development Organization", *MIS Quarterly*, 13, (3): 293-308, (September 1989b).
- Swanson, E. B. and C. M. Beath, "Departmentalization in Software Development and Maintenance", *Communications of the ACM*, 33, (6): 658-667, (June 1990).
- Tenny, T., Program, "Readability: Procedures Versus Comments", *IEEE Transactions on Software Engineering*, SE-14, (9): 1271-1279, (1988).
- Troy, D. A. and S. H. Zweben, "Measuring the Quality of Structured Designs", *Journal of Systems and Software*, 2, 113-120, (1981).
- Vessey, I. and R. Weber, "Some Factors Affecting Program Repair Maintenance: An Empirical Study", *Communications of the ACM*, 26, (2): 128-134, (1983).
- Wake, S. and S. Henry, "A Model Based on Software Quality Factors Which Predicts Maintenance", *Proceedings of the 4th IEEE Conference on Software Maintenance*, pp. 382-387, 1988.
- Weiser, M., "Programmers Use Slices When Debugging", *Communications of the ACM*, 25, (7, July): 446-452, (1982).
- Weiss, D. M. and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory", *IEEE Transactions on Software Engineering*, SE-11, (2): 157-168, (February 1985).
- Wiener-Ehrlich, W. K., J. R. Hamrick and V. F. Rupolo, "Modelling Software Behavior in Terms of a Formal Life Cycle Curve: Implications for Software Maintenance", *IEEE Transactions on Software Engineering*, SE-10, ( 4, July): 376-383, (1984).
- Woodfield, S. N., H. E. Dunsmore and V. Y. Shen, "The Effect of Modularization and Comments on Program Comprehension", *5th International Conference on Software Engineering*, pp. 215-223, 1981a.

- Woodfield, S. N., V. Y. Shen and H. E. Dunsmore, "A study of several metrics for programming effort", *Journal of Systems & Software*, 2, 97-103, (1981b).
- Wu, C. F., "Information System Development Audits and Software Maintenance", *3rd IEEE Conference on Software Maintenance*, pp. 190-197, 1987.
- Yau, S. S. and P. S. Chang, "A Metric of Modifiability for Software Maintenance", *Proceedings of the 4th IEEE Conference on Software Maintenance*, pp. 374-381, 1988.
- Yau, S. S. and J. S. Collofello, "Design Stability Measures for Software Maintenance", *IEEE Transactions on Software Engineering*, SE-11, (9): 849-856, (September 1985).
- Zvegintzov, N., "High Noon III: Continuing the quest for a true test of software maintenance tools", *Software Maintenance News*, 6, (1): 6-7, (January 1988).

2110 071







Date Due 1992

FEB. 13 1992  
APR. 17 1992

Lib-26-67

MIT LIBRARIES



3 9080 00719087 6

