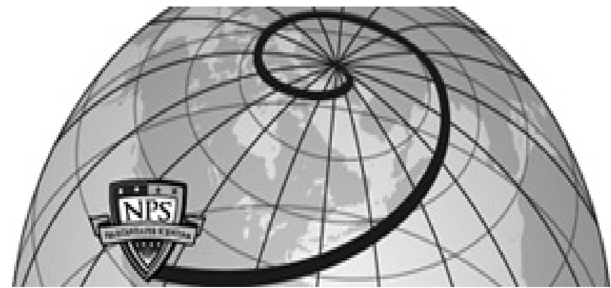
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers — Faculty and Researchers' Publications

1993-11

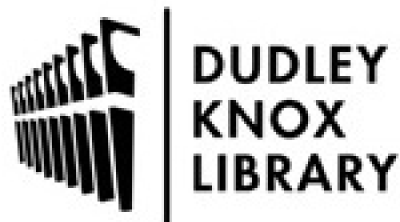
Software Complexity and Maintenance Costs

Banker, R.D.; Datar, S.M.; Kemerer, C.F.; Zweig, D.

Banker, Rajiv D., et al. "Software complexity and maintenance costs." Communications of the ACM 36.11 (1993): 81-95.
http://hdl.handle.net/10945/62193

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig

SOFTWARE COMPLEXITY AND MAINTENANCE COSTS

While the link between the difficulty in understanding computer software and the cost of maintaining it is appealing, prior empirical evidence linking software complexity to software maintenance costs is relatively weak [21]. Many of the attempts to link software complexity to maintainability are based on experiments involving small pieces of code, or are based on analysis of software written by students. Such evidence is valuable, but several researchers have noted that such results must be applied cautiously to the large-scale commercial application systems that account for most software maintenance expenditures [13, 17]. Furthermore, the limited large-scale research that has been undertaken has generated either conflicting results or none at all, as, for example, on the effects of software modularity and software structure [6, 12]. Additionally, none of the previous work develops estimates of the actual cost of complexity, estimates that could be used by software maintenance managers to make the best use of their resources. While research supporting the statistical significance of a factor is, of course, a necessary first step in this process, practitioners must also have an understanding of the practical magnitudes of the effects of complexity if they are to be able to make informed decisions.

This study analyzes the effects of software complexity on the costs of Cobol maintenance projects within a large commercial bank. It has been estimated that 60 percent of all business expenditures on computing are for maintenance of software written in Cobol [16]. Since over 50 billion lines of Cobol are estimated to exist worldwide, this also suggests that their maintenance represents an information systems (IS) activity of considerable economic importance. Using a previously developed economic model of software maintenance as a vehicle [2], this research estimates the impact of software complexity on the costs of software maintenance projects in a traditional IS environment. The model employs a multidimensional approach to measuring software complexity, and it controls for additional project factors under managerial control that are believed to affect maintenance project costs.

The analysis confirms that software maintenance costs are significantly affected by software complexity, measured in three dimensions: module size, procedure size, and branching complexity. The findings presented here also help to resolve the current debate over the functional form of the relationship between software complexity and the cost of software maintenance. The analysis further provides actual dollar estimates of the magnitude of this impact at a typical commercial site. The estimated costs are high enough to justify strong efforts on the part of software managers to monitor and control complexity. This analysis could also be used to assess the costs and benefits of a class of computer-aided software engineering (CASE) tools known as restructurers.

Previous Research and Conceptual Model

Software maintenance and complexity. 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 [28]. Research on the costs of software maintenance has much in common with research on the costs of new software development, since both involve the creation of working code through the efforts of human developers equipped with appropriate experience, tools, and techniques. Software maintenance, however, is fundamentally different from new systems development in that the soft-

ware maintenance must interact with an existing system. The goal of the current research is to identify the factors affecting the assimilation process and thereby increase (decrease) the amount of effort required to perform the maintenance task. In particular, the current research focuses on measuring the impact of the existing source code aspects believed to affect the amount of effort required.

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*" [4, p. 232]. Curtis et al. similarly define this concept as psychological complexity: "*Psychological complexity refers to characteristics of software which make it difficult to understand and work with*" [15, p. 96]. Both of these authors note that the lack of use of structured programming techniques is believed to increase the cognitive load on a software maintainer. In the current research this will simply be referred to as software complexity, with the focus being on correctable software complexity (i.e., complexity that results from specific syntactical choices made by the developer).

Factors that increase maintainer effort will increase project cost, since maintenance costs are most directly a function of the professional labor component of maintenance projects. Therefore, this research is designed to measure the impact of aspects of software complexity of the existing system that affect the cost of maintenance by increasing or decreasing the amount of maintainer effort to comprehend the software, while controlling for project factors that may also affect performance. Given the growing economic importance of maintenance, several researchers have attempted to validate hypotheses relating to complexity. However, researchers have not been able to empirically test the impact of complexity on maintenance effort while controlling for additional factors known to affect costs, such as project size and maintainer skill [19, 31]. The main research objective in this article is investigating the relationship between existing software complexity and maintenance costs. However, in order to properly understand this relationship, the effects of project factors will be controlled for. Figure 1 presents a simplified view of the conceptual model that will be tested in this research.

Modularization. Researchers have employed many measures in attempts to operationalize the concept of software complexity. The consensus is that there is no single best metric of software complexity [5, 13, 27]. However, two main concepts have emerged—modularity and branching.

Schneidewind estimates that 75- to 80% of existing software was produced prior to significant use of structured programming [28]. A key component of structured programming approaches is *modularity,* defined by Conte et al. as "the programming technique of constructing software as several discrete parts" [13, p. 197]. 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 (see Table 1).

In terms of positive impacts of greater modularity, perhaps the first widely disseminated field research in this area was by Vessey and Weber [30]. They studied repair maintenance in Australian and U.S. 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 more modular code was associated with fewer repairs; in the other data set no effect was found. In a later study, Korson and Vaishnavi [24] 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.

Card et al. [12] reached the opposite conclusion. They tested the impact of module size and strength (cohesion) on programming effort, measured as programmer hours per executable statement. They found that effort decreased as the size of the module increased. However, effort decreased as strength increased, but increases in strength were associated with decreases in module size. They concluded that nothing definitive could be stated about the impact of module size. A study by An et al. [1] analyzed changed data from two releases of Unix. They found 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.

An alternative hypothesis is that modules that are either too large or too small are unlikely to be optimal. If the modules are too large they are unlikely to be devoted to single purpose. 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. In contrast to the unidirectional studies cited previously, a few researchers have suggested the possibility of bidirectional effects. For example, Conte et al. note that : "*The degree of modularization affects the quality of a design. Over-modularization is as undesirable as undermodularization*" [13, p. 109]. In an analysis of secondary data, Bowen compared the number of source lines of code (SLOC)/module with a set of previously proposed maximum desirable values of two well-known metrics, McCabe's V(G) and Halstead's N [10]. He concluded that the optimal values of SLOC/module differed across languages, but that all were much lower than the Department of Defense's (DoD's) proposed standard of 200 SLOC/module. In his suggestions for future research, he notes the following:

> *More research is necessary to derive and validate upper and lower bounds for module size. Module size lower bounds, or some equivalent metric such as coupling, have been neglected; however they are just as significant as upper bounds. With just a module size upper bound, there is no way to dissuade the implementation of excessively small modules, which in turn*

introduce inter-module complexity, complicate software integration testing, and increase computer resource overhead. [10, p. 331]

Boydston undertook a study of completed systems programming projects at IBM whose main purpose was to gain greater accuracy in cost estimation [11]. One additional analysis he performed (p. 155) was to attempt to estimate the optimum SLOC/module ratio for new code, based on the hypothesis that *"Complexity of programming increases as the lines of code per module and the number of modules to interface increase."* In other words, extremes of either a very small number of large modules or a very large number of small modules would both be unlikely to be optimal. His regression analysis developed multiple, nonlinear functions of work-months as a function of the number of new modules, with SLOC held constant. He concludes that *". . . as a project gets larger, the additional complexity of larger modules has to be balanced by the increasing complexity of information transfer between modules"* (p. 159). However, his model does not control for any noncode factors.

While not examining maintenance cost directly, Lind and Vairavan obtained empirical evidence supporting the hypothesis of a nonextreme optimum value for module size (i.e., that the best-sized modules were those that were neither too big nor too small) [25]. They analyzed the relationship between the change rate (number of changes per 100 lines of code, a surrogate for cost) vs. a discrete (categorical) lines-of-code-based variable. Their five discrete SLOC categories were 0–50, 50–100, 100–150, 150–200, and 200+. They found that minimum change rates occurred in the 100 to 150 range, a result they describe (p. 652) as indicating the *" . . . program change density declines with increasing metric values up to a certain minimum value . . . beyond this minimum value, the program change density actually increases with an increase in the value of the metrics."*

The results of these previous studies can be summarized as follows. Researchers testing for unidirectional results (i.e., that either smaller modules or larger modules were better) have found either contradictory results or none at all. Other researchers have suggested that a U-shaped function exists, that is, modules that are either too small or too large are problematical. In the case of many small modules, more intermodule interfaces are required. In the case of a few large modules, these modules are less likely to be devoted to a single purpose.[1] However, researchers who suggest the U-shaped curve hypothesis provide either limited data or none at all linking size and cost. In general they also do not provide a model for determining the optimum module size.[2]

The most recent research includes an earlier study at the current research site where 35 application systems were analyzed to develop a basis for selecting among dozens of candidate software metrics that the research literature has suggested [32]. Figure 2 shows the relationship among the three software levels identified in this research. An *application system* has *M modules.* In turn, each module *m* has N_m *procedures.* Table 2 provides the definitions for these levels.

Previous research investigating a large number of proposed software complexity metrics has found them to be variations on a small number of orthogonal dimensions [27]. An analysis of software complexity metrics at this research site identified three major groups: procedure-level modularity, module-level modularity, and branching [32]. Despite their apparent similarities, previous research has suggested that the two kinds of modularity represent independent aspects of software complexity [14]. A commercial static code analyzer was used to compute these metrics. Given the high levels of correlation within (but not across) complexity metric groups, a representative metric from each group was selected, based in part on the ease with which it could be understood by software maintenance management and its ease of collection. This approach has been recommended by previous research [27].

[1]Interfaces are relevant because they have been shown to be among the most problematical components of programs [6]. Modules not devoted to a single purpose have been shown to result in a larger number of errors and therefore higher amounts of repair maintenance, which can be interpreted as increased cost [12, 30].

[2]Boydston [11] does extrapolate from his data set to suggest a specific square root relationship between number of new lines of code and number of modules for his Assembler and PLS language data.

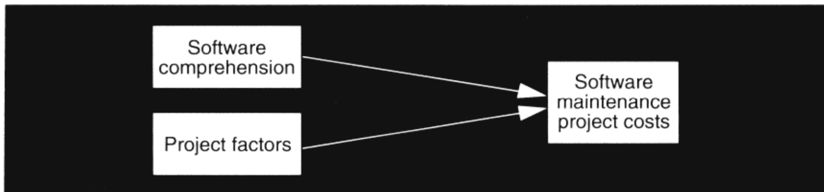
The first metric is the average size in executable statements of a module's procedures (PROCSIZE). There is an almost universal tendency to associate large procedure size with poor procedure-level modularity. However, intuitively, neither extreme is likely to be effective. If modules are broken into too many small procedures, complexity could rise, and in this case increasing the average procedure size would be expected to be beneficial.

Module length, in executable statements (MODLSIZE) was selected as the metric of module-level modularity [5].[3] The effect of this complexity metric is expected to depend on the application systems being analyzed. As discussed in the survey of previous research, it is generally believed that large modules will be more difficult to understand and modify than small ones, and maintenance costs will be expected to increase with average module size. As with procedures, however, a system can be composed of too many small modules. If modules are too small, a maintenance project will spread out over many modules with the attendant interface problems. Therefore, complexity could decrease as module size increases. Thus two specific research hypotheses concerning modularity are proposed:

Hypothesis 1. *Controlling for other factors known to affect software maintenance costs, the costs will depend significantly on average procedure size as measured by PROCSIZE, with costs rising for applications whose average procedure size is either very large or very small.*

Hypothesis 2. *Controlling for other factors known to affect software maintenance costs, the costs will depend significantly on average module size as measured by MODLSIZE, with costs rising for applications whose average module size is ei-*

[3]This metric was found to be uncorrelated with PROCSIZE (Pearson correlation coefficient = .10).



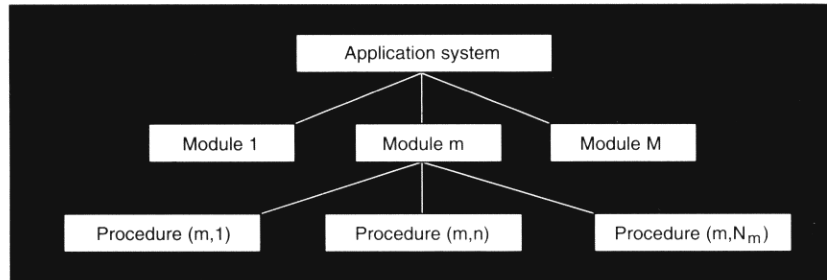


Figure 1. Software maintenance project costs conceptual model

Figure 2. Software level hierarchy

ther very large or very small.

Branching. Previous work has suggested that control constructs (branching) are expected to have a significant impact on comprehension [15]. Structured programming is a design approach that limits programming constructs to three basic means of branching through a piece of software. Because it is difficult to comply with these structures using the GOTO syntax found in older programming languages, this approach is sometimes colloquially referred to as GOTO-less programming. A review of work in this area before 1984 was conducted by Vessey and Weber [31]. While few negative results have been found, they note the absence of significant results is as frequent as a finding of positive results. They attribute this outcome, in part, to the researchers' not having adequately controlled for other factors. They also note the difficulty of achieving such control, particularly in nonlaboratory real-world settings.

More recently, Gibson and Senn have investigated the impact of software structure using a laboratory experiment [17]. They found that more structured versions of the same piece of software required less time to maintain on average. They also found that maintainers' subjective assessments of the complexity of the existing systems were not very accurate, a result they attribute to the maintainers' inability to separate task complexity from existing systems complexity. They recommend using objective measures of systems complexity to remedy this defect. However, the expected results from their experiments did not hold in all cases. In addition, as noted by the authors, laboratory experimentation is not a substitute for field research: *"Further research is needed to determine whether the relationships observed in this tightly controlled experiment exist in live settings"* (p. 357). In particular, laboratory experimentation is unlikely to provide estimates of the actual cost impacts of ill-structured programs in commercial settings.

In a recent pilot study of seven maintenance projects on Fortran and Pascal-based real-time systems, Gill and Kemerer found that maintainer productivity decreased as existing systems complexity increased, as measured by complexity density, a size-adjusted measure of branching complexity [18]. However, their model does not control for any noncode factors. The authors also note the need to validate these results on a larger sample of commercial systems. Therefore, the question of a negative impact of excessively complex branching on maintenance costs has only limited empirical support, and there is a need for further research.

In the current research the initial candidate metric chosen for branching was the proportion of the executable statements that were GOTO statements (GOTOSTMT). This branching metric is normalized for module size, so that it would not be confounded with MODLSIZE. This metric is also a measure of module divisibility, since the degree to which a module can be divided into small and simple procedures depends directly on the incidence of branching within the module. Highly divisible modules (modules with low values of GOTOSTMT) should be less costly to maintain, since a maintainer can deal with manageable portions of the module in relative isolation.

While the density of GOTO statements (GOTOSTMT), like other candidate control metrics examined, is a measure of divisibility,[4] it does not distinguish between more and less serious structure violations. A branch to the end of the current paragraph, for example, is unlikely to make that paragraph much more difficult to comprehend, while a branch to a different section of the module may. However, none of the existing structure metrics examined clearly differentiate between the two cases. In addition, the modules analyzed have a large incidence of GOTO statements (approximately 7 per 100 executable statements). If only a relatively small proportion of these seriously affect maintainability then the GOTOSTMT metric may be too noisy a measure of branching complexity. At this research site over half of the GOTOs in these modules (19 GOTOs out of 31 in the average module) are used to skip to the beginning or end of the current paragraph. Such branches would not be expected to contribute noticeably to the difficulty of understanding a module (in most high-level languages other than Cobol they would probably not be implemented by GOTO statements). Therefore, a metric such as GOTOSTMT, which does not distinguish between these and the approximately 40% less benign branch commands, will be unlikely to be managerially useful.

To avoid this problem, a modified metric was computed, GOTOFAR, which is the density of the GOTO

[4]Each GOTO command makes a module more difficult to understand by forcing a programmer to consider multiple portions of the module simultaneously.

Table 1. Previous field research on modularity

Year	Researchers	Language	Dependent variable	Conclusions [a]
1983	Vessey and Weber	Cobol	# of Repairs	Unidirectional ⇑
1984	Bowen	Algol, CMS, and others	McCabe, Halstead metrics	Suggests two-way relationship
1984	Boydston	Assembler, PLS	Effort	Suggests two-way relationship
1985	Card, et al.	Fortran	Effort	Unidirectional ⇓
1986	Korson and Vaishnavi	Pascal	Effort	Unidirectional ⇑
1987	An, et al.	C	Change data	Unidirectional ⇓
1989	Lind and Vairavan	Pascal, Fortran	Normalized change data	Suggests two-way relationship

Table 2. Source code definitions

Level	Definition
Application system	A set of modules assigned a common name by the research site, typically performing a coherent set of tasks in support of a given department and maintained by a single team. References to this term refer only to the source code, not to the JCL. 'Application' or 'system,' if used separately, mean the same thing. [b]
Module	A named, separately compilable file containing Cobol source code. A module will typically, though not necessarily, perform a single logical task or set of tasks. INCLUDE modules and COPY files were the only modules not included, since they contain only Cobol source code but not the headers that allow it to be run on its own.
Procedure	The range of a PERFORM statement. For example, if paragraphs are labeled sequentially, the statement PERFORM D THRU G invokes the procedure consisting of paragraphs D, E, F, G, and the paragraphs invoked by these paragraphs. [c]
Paragraph [d]	The smallest addressable unit within a piece of Cobol software. A sequence of Cobol-executable statements preceded by an address/identification label.

[a] For unidirectional test, "⇑" indicates that greater modularity (more, smaller modules) improved performance and "⇓" indicates that less modularity (fewer, larger modules) improved performance. Several of the analyses in these unidirectional studies also found no significant results in either direction. A two-way relationship is one in which both positive and negative deviations from optimal module size reduce performance.

[b] Application systems can be described as being composed of 'programs,' but the current research has analyzed the data at a finer level of detail, the module, and the program construct has not been used in the current research.

[c] The possibility exists, in Cobol, that procedures will overlap. (e.g., PERFORM D THRU G and PERFORM E THRU J will have at least E, F, and G in common.) This research followed previous work by Spratt and McQuilken in defining the union of overlapping procedures to be a single procedure to prevent double counting [29]. Such overlaps were relatively rare at this site, however, with the result that this research design decision results in no practical difference. Spratt and McQuilken use the term "components" instead of procedures, but the latter term will be used throughout this article.

[d] This construct is not used directly in this research, but is defined here as it is used in the definition of procedure.

statements that extend outside the boundaries of the paragraph and that can be expected to seriously impair the maintainability of the software.[5] Since the automated static code analyzer was not able to compute this metric, it was computed manually. Due to the large amount of data collection effort and analysis this computation required, the metric was manually computed by a single analyst for a random sample of approximately 50 modules per application system. This random sample consisted of approximately 1,500 modules in total, or approximately 30% of all modules in the total data set.[6]

Therefore, the third research hypothesis is:

Hypothesis 3. Controlling for other factors known to affect software maintenance costs, software maintenance costs will depend significantly on the density of branching as measured by GOTOFAR, with costs rising with increases in the incidence of branching.

Table 3 summarizes the software complexity variables used in the model. Figure 3 presents the full conceptual model.

Project factors.[7] The research model has two main components,

[5] This is believed to be similar in concept to Gibson and Senn's [17] elimination of "long jumps in code (GOTOs)."

[6] A later sensitivity analysis regression using GOTOSTMT instead of GOTOFAR lends credence to the belief that the excluded branch commands represent a noise factor. The estimated effect of GOTOSTMT had the same relative magnitude as that of GOTOFAR, but the standard error of the coefficient was 4 times as large.

one consisting of factors related to existing source code complexity, and one of controllable project factors that are believed to affect maintenance costs. While the current research focuses on assessing the effect of software complexity on maintenance costs, it is necessary to control for project factors (such as task size and the skill of the developers) known to affect these costs [19, 31]. The most significant of these is the size of the maintenance task. Excluding task size or other relevant factors would result in a misspecification of the model and incorrect inferences about the impact of software complexity on costs.[8] To control for this factor and for other project factors known to affect costs, the research began with a previously developed economic model of software maintenance. The initial data collection procedures and model development are described in detail in [2] and [22]. They will only be summarized here.

Basic maintenance cost model. This model adopts the standard cost model formulation developed in the software engineering literature [3, 9]:

Effort = f(Size, Other Cost Drivers)

Table 4 summarizes the measures of the maintenance function used based on the model developed in [2]. The unit of analysis for this model is the project as defined by the research site. Each maintenance project has its own task requirements and its own budget. Table 5 shows the project factors that are included in the model for each project. The output of the software maintenance process is the modified system, and therefore measures of the size of the additions and changes need to be included in the model. Measures of size in a maintenance context are the size of the portions of the system that were added or changed by the maintenance project. While SLOC added or changed is the most widely used measure of size, function points (FPs) added or changed are gaining in acceptance [3]. FPs have an additional advantage of including a measure of *task* complexity.[9]

The SKILL variable is important, as previous research has found large differences in ability between top-rated developers and poorer ones [9]. All maintainers in the organization at the research site are rated on a numerical scale, and the measure used in the model is the percentage of hours that were charged to the project by staff who were highly rated. The SKILL variable is often neglected in research due to the practical difficulties involved in collecting these data. These practical difficulties include the fact that formal personnel ratings may not always be available, and, even if collected by the organization, may not be made available to researchers for confidentiality reasons. For the current work strict control over these data were guaranteed to the research site by the researchers.

A personnel-related variable distinct from ability is LOWEXPER [9, 20]. Even a good developer is at a disadvantage when faced with an unfamiliar system, as time must be expended in comprehending the software and becoming familiar with it.

METHOD, the use of a structured analysis and design methodology, is meant to increase developer performance. However, previous research has shown that such methods add costs in the short term at this site [2]. QUALITY may also be important, as it has been suggested that doing a careful job of error-free programming will cost more than a rushed job, although benefits will be realized in the long term. Conversely, some researchers believe that careful and systematic programming may not take longer, with some even arguing that it should be less expensive. The measure used here was one of operational quality, the degree to which the system operates smoothly after the maintenance project's changes are placed into production. The measure was generated from data on abnormal ends and user problem reports collected on an ongoing basis by the research site. Data from the two-month period following implementation were compared with data from the previous 12 months' trend. Statistically significant deviations from the previous mean resulted in above or below average operational quality ratings [22]. The RESPONSE variable is included as there has been some evidence that fast-turnaround environments enhance developer performance, an effect that is likely to be seen in maintenance work as well.

Based on the software economics literature the effects of these factors are believed to be proportional, rather than absolute [3, 9]. Thus they are weighted by project size, either FP added or changed or SLOC added or changed, depending on whether they are thought to be associated more strongly with the analysis/design phase or with the coding/testing phase of the project [2]. Skill and application experience are weighted by FPs, as it was believed their impact would be felt most strongly during analysis/design, where the greatest amount of leverage from capability and experience would be obtained. Use of the structured analysis/design methodology is also clearly associated with the analysis and design phase, measured here by FPs. Operational quality was weighted by SLOC, as the types of errors represented by the operational quality measure used reflect poor coding technique and/or insufficient testing. Response time was also weighted by SLOC, as it seems more relevant to coding/testing activities than to analysis/design work, since the latter is not dependent on access to machine cycles. Finally, all complexity measures are weighted by SLOC, since the impact of existing code complexity would be felt most strongly during coding/testing rather than analysis/design. As noted ear-

[7]This section draws heavily on work presented in [2].

[8]It should be noted that this research's inclusion of factors other than complexity militates *against* finding any statistical effect resulting from complexity, in contrast to previous research that examines the effect of complexity without controlling for other factors. While the model presented does not possess undesirable multicollinearity, no empirical model of this type has factors that are completely orthogonal. Therefore, inclusion of the other factors partially reduces any effect found for the complexity factors, making this a conservative test of the complexity hypotheses.

[9]This should not be confused with the application software complexity that is the focus of this research. Task complexity in FPs includes such factors as whether the project will be held to above average reliability standards, or whether the operational system will run in a distributed environment.

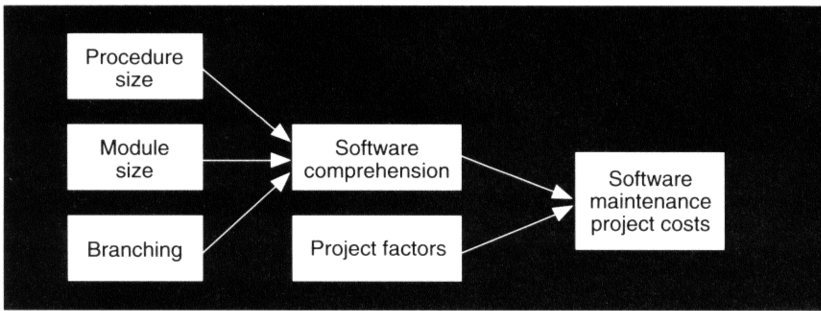


Figure 3. Software maintenance project cost model

Table 3. Software complexity variables

Name	Variable	Measurement	References
PROCSIZE	Average size of a module's procedures	A count of the number of noncomment SLOC in a module divided by the number of procedures	[32]
MODLSIZE	Average size of an application's modules	A count of the number of noncomment SLOC in the application divided by the number of modules.	[5, 32]
GOTOFAR	Density of the nonbenign GOTO statements	A normalized count of the GOTO statements which extend outside the boundaries of the paragraph	[17, 32]

Table 4. Cost drivers

Activity	Measured by	Mediated by
Analysis/ design	Function points (FPs) added or changed by the project	Maintainer skill Maintainer application experience Structured analysis/design methodology use
Coding/testing	Source lines of code (SLOC) added or changed by the project	Operational quality Hardware response time Application source code complexity (3 measures)

lier, any collinearity that may exist between the weighted complexity metrics and other independent variables that have been weighted by SLOC will cause the model to underestimate the significance of the complexity metric variable. Therefore, the following analysis is a conservative test.

Statistical Model and Results

The previous section described the selection of the variables in the model, including both the existing source code complexity variables and the project factors. In this section, following a brief description of the research site, the statistical model and its results are presented, followed by tests of the research hypotheses.

The research site. Data were collected at a major regional bank with a large investment in computer software. The bank's systems contain over 18 million lines of code. Almost all are written in the Cobol programming language, and are running on large IBM mainframe computers. The software is organized into large application systems (e.g., demand deposits), which have an average size of 226,000 SLOC.[10] Some of the bank's major application systems were written in the mid-1970s and are generally acknowledged to be more poorly designed and harder to maintain than recently written software.

[10]Mean = 226 KSLOC, standard deviation = 185 KSLOC, min = 54 KSLOC, max = 702 KSLOC.

Given that Cobol and IBM are the most widely used software and hardware in commercial information systems (IS), this software environment appears to be a typical commercial data processing environment. Thus, the research results should apply to other commercial environments, especially those with financial services transaction processing systems. The projects analyzed were homogeneous in that they all modified Cobol systems, and therefore the results are not confounded by the effects of multiple programming languages.

Sixty-five software maintenance projects from 17 major application systems were analyzed. These projects were carried out between 1985 and 1987. An average project took about 1,000 hours (at an accounting

Table 5. Maintenance model project factor variables [2]

Name	Variable	Measurement	References
HOURS	Effort	Number of hours charged to the project. This information was obtained from the project billing files, which were collected contemporaneously with the project.	[3] [9]
FP	Task magnitude and task complexity	The number of function points added or changed by the maintenance project.	[3]
SLOC	Task magnitude	The number of source lines of code added or changed by the maintenance project.	[9]
SKILL	Maintainer skill	The percentage of developer hours billed by the most highly skilled (by formal management evaluation) class of developers. This variable is distinct from the following one, which depends on the developer's experience with a specific application system.	[9]
LOWEXPER	Maintainer application experience	The extensive use (over 90% of hours billed to the project) of developers lacking experience with the application being modified. (A binary variable.)	[9, 20]
METHOD	Structured analysis/ design method use	The use of a structured design methodology (a binary variable). This is expected to have an adverse effect on single-project performance, although it is meant to reduce costs to the organization in the long run.	[2]
QUALITY	Operational quality	A measure (on a three-point scale of low/average/high quality) of the degree to which the completion of the project was followed by a change in the number of operational errors. This measure was based on information obtained from the site's error logs.	[2, 22]
RESPONSE	Hardware response time	The availability of a fast-turnaround programming environment. (A binary variable.)	[9, 20]

cost of $40 per hour) and added or changed approximately 5,000 SLOC.

Statistical Model. The statistical model is described by:

$$\begin{aligned} \text{HOURS} = {} & \beta_0 + \beta_1\text{*FP} + \\ & \beta_2\text{*SLOC} + \beta_3\text{*FP*FP} + \\ & \beta_4\text{*SLOC*SLOC} + \\ & \beta_5\text{*FP*SLOC} + \\ & \beta_6\text{*FP*SKILL} + \\ & \beta_7\text{*FP*LOWEXPER} + \\ & \beta_8\text{*FP*METHOD} + \\ & \beta_9\text{*SLOC*QUALITY} + \\ & \beta_{10}\text{*SLOC*RESPONSE} + \\ & \beta_{11}\text{*SLOC*PROCSIZE} + \\ & \beta_{12}\text{*SLOC*PROCSIZE}^2 + \\ & \beta_{13}\text{*SLOC*MODLSIZE} + \\ & \beta_{14}\text{*SLOC*MODLSIZE}^2 + \\ & \beta_{15}\text{*SLOC*GOTOFAR} + \epsilon \end{aligned}$$

This model, without the five complexity terms (the terms associated with parameters β_{11} through β_{15}), has been previously validated at the research site. The relationships between maintenance costs and procedure size and between maintenance costs and module size are expected to be U-shaped, rather than monotone, with costs being lowest for some optimal size and higher for larger or smaller sizes. The squared terms PROCSIZE^2 and MODLSIZE^2 are included to model this effect.

In this model project costs (measured in developer HOURS) are primarily a function of project size, measured in FPs and in SLOC. To model the known nonlinearity of development costs with respect to project size, not only FP and SLOC are included, but also their second-order terms. This approach is expected to result in a high degree of multicollinearity among the size variables (the terms associated with parameters β_1 through β_5) which will make the interpretation of their coefficients difficult [3]. The multicollinearity among the size variables, however, is not of concern for examining the current research hypotheses relating to the impact of complexity, since the complexity variables are not collinear with the size variables. Table 6 presents the summary statistics for this data set. The values given for the complexity metrics are application system averages. The model was estimated using ordinary least squares (OLS) regression, since the OLS assumptions were satisfied in the context of the estimation. The statistical results from two-tailed tests are presented in Table 7 with the complexity metric variables in bold type. The summary statistical results are as follows:

$F_{15,49} = 28.63$ $(p < .0001)$, $R^2 = 89.76\%$, Adjusted $R^2 = 86.62\%$.

Although not all project factor variables are significant for this sample, none of the project factor variables are eliminated in order to achieve a more parsimonious fit. The interest

in the current research is in assessing the *marginal* impact of adding the complexity metrics to an earlier version of the model (see [2]). The Belsley-Kuh-Welsch multicollinearity diagnostics (see [8]) indicated that the complexity metrics are not significantly confounded with the other regression variables. Thus, their coefficients may be interpreted with relative confidence. Also the residuals and the absolute residuals were uncorrelated with size. The latter result supports the homoskedasticity assumption in regression analysis. The former supports the decision to model the complexity effects in the regression as proportional ones rather than use the unweighted metrics alone. If the complexity effects were not proportional to project magnitude, use of the weighted metrics would cause the model to overestimate the costs of large projects, resulting in residuals negatively correlated with size.

Tests of the research hypotheses. Hypothesis 1 was that maintenance costs would be significantly affected by procedure size. This general hypothesis is confirmed by an F-test on the joint effect of the two PROCSIZE terms:

$P(H_0: \beta_{11} = \beta_{12} = 0) < 0.0001$ as $F_{2,49} = 14.20$.

A U-shaped relationship between PROCSIZE and software maintenance costs was hypothesized, and the data confirm this relationship, given that the two coefficients are significantly different from zero and that the linear term is negative and the squared term is positive. The minimum of the U-shaped curve may be computed by dividing the negated coefficient of the linear term by twice that of the quadratic term.[11] At this site the minimum-cost procedure size was computed to be $(0.0106/(2*0.00012)) = 44$ executable statements PROCSIZE (See Table 7). This value is very close to the mean (43) and to the median (40) for this organization. However, individual applications vary in average PROCSIZE from 13 to 87 executable statements.

As is often the case in this type of estimation there was a high degree of multicollinearity between the linear term and the quadratic term. This means that the estimates of the two individual coefficients (and hence the minimum point) are to be taken with caution. To test the robustness of this calculation the analysis was repeated using a model that replaced the linear and quadratic PROCSIZE terms with two linear variables, representing positive and negative deviations from a conjectured optimum respectively.[12] This model was repeatedly estimated using a different conjectured optimum value each time. The results consistently showed cost increases resulting from deviations in either direction from the minimum point. This sensitivity analysis supports the results shown in Table 7 suggesting a bidirectional (U-shaped) relationship.

Hypothesis 2, that costs increase for both large or small values of MODLSIZE, was not supported, as the conditions described in the discussion for PROCSIZE were not met. Since the coefficients for both the linear and quadratic MODLSIZE variables are in the same direction, they are likely picking up each other's effects, and therefore the individual t-test values are low. However, a hypothesis that maintenance costs are not significantly affected by module size can be rejected:

$P(H_0: \beta_{13} = \beta_{14} = 0) = 0.0076$ as $F_{2,49} = 5.39$

which supports the notion that MODLSIZE, as suggested by previous research, is a variable worthy of managerial attention. A similar insight is obtained from a simplified version of the model that excludes the $MODLSIZE^2$ term. There the coefficient for the SLOC *MODLSIZE term = −.00012, t = −3.32 (p = .0017). This result can be interpreted in the traditional way, that is, the effect at this site tended to be linear over the observed range of module sizes (controlling for project factors) with costs decreasing as module size increases.[13]

It should be noted, however, that while these data do not support a U-shaped relationship, they are not necessarily inconsistent with such a hypothesis. The observed linear relationship is consistent with the data falling on the downward sloping arm of this U, with the possibility that costs would again begin to rise had sufficiently large modules been available. Therefore, if there is a U-shaped relationship, the turning point appears to be outside the range of data collected at this site. Further empirical work at other research sites will be required for this alternative interpretation to be verified.

Hypothesis 3 was that maintenance costs would be significantly affected by the density of branch instructions within the modules. This hypothesis is confirmed.

$P(H_0: \beta_{15} = 0) = 0.0021$ as $t_{49} = 3.25$.

Software maintenance costs are seen to increase linearly with an increase in the number of long GOTO statements, as defined earlier.

Implications for Software Maintenance Management

Through the preceding analysis the effect of software complexity on software maintenance costs has been estimated. While it is a firmly established article of conventional wisdom that poor programming style and practices increase programming costs, little empirical evidence has been available to support this notion. Consequently, efforts and investments meant to improve programming practices have relied largely on faith. The current research has extended an existing model of software maintenance and used it as a vehicle to confirm the significance of the impact of software complexity on project costs and to estimate its magnitude.

This model provides managers with estimates of the benefits of improved programming practices that can be used to justify investments designed to improve those practices. Given these data and estimates relat-

[11]This can easily be seen by differentiating with respect to x the quadratic equation $y = ax + bx^2$, and setting $dy/dx = 0$ which yields $x = -a/2b$.

[12]This can be seen as measuring the relationship as a 'V' rather than a 'U.'

[13]With this simplified model it is noteworthy that while concern over modularity typically focuses on large modules, at this site the systems that cost more to maintain tended to have modules that were too small.

Table 6. Maintenance project summary statistics (65 projects)

Variable	Mean	Stand. Deviation	Min	Max
HOURS	937	718	130	3342
FP	118	126	8	616
SLOC	5416	7230	50	31060
SKILL	65	34	0	100
LOWEXPER	.66	.48	0	1
METHOD	.32	.47	0	1
QUALITY	2.06	.58	1	3
RESPONSE	.65	.48	0	1
MODLSIZE	681	164	382	1104
PROCSIZE	43	18	13	87
GOTOFAR	0.024	0.016	0.0	0.07

Table 7. Regression results

Variable	β	*Coefficient*	*Standardized* β*s*	*t*	*p*
Intercept	0	333	0	4.96	.0001
Project size					
FP	1	3.152	.554	1.98	.0533
SLOC	2	0.342	3.448	5.01	.0001
FP*FP	3	.009	.774	2.80	.0072
SLOC*SLOC	4	−2.8E-6	−.743	−0.92	.3614
FP*SLOC	5	−.0001	−.439	−1.29	.2026
Project Environment					
FP*SKILL	6	−.049	−.64	−3.48	.0011
FP*LOWEXPER	7	.122	.02	0.18	.8578
FP*METHOD	8	1.764	.228	3.03	.0039
SLOC*QUALITY	9	.027	.575	2.74	.0085
SLOC*RESPONSE	10	−.019	−.196	−1.17	.2486
Software complexity					
SLOC*PROCSIZE	**11**	**−.0106**	**−5.404**	**−4.85**	**.0001**
SLOC*PROCSIZE^2	**12**	**.00012**	**3.708**	**5.30**	**.0001**
SLOC*MODLSIZE	**13**	**−.00011**	**−.774**	**−1.36**	**.1815**
SLOC*MODLSIZE^2	**14**	**−4.4E-10**	**−.077**	**−0.09**	**.9279**
SLOC*GOTOFAR	**15**	**1.317**	**.401**	**3.25**	**.0021**

ing software complexity and costs, the form of the model allows inference about the productivity of software maintainers. Productivity is typically defined as the ratio of output to input. Since the model controls for task size (output) variables on the RHS (right-hand size), any LHS (left-hand size) increases in required inputs that are associated with increases in complexity can be interpreted as decreases in productivity. Therefore, the model results may be interpreted to mean that *increased existing software complexity significantly decreases the productivity of software maintainers.* This result accords with strongly held intuition. The current research also provides actual estimates of the magnitude and significance of this effect, results that have generally not been available, particularly for commercial applications involving actual maintenance activities and controlling for project factors believed to affect productivity.

This model enables managers to estimate the *benefits* of improving software development and maintenance practices, and to justify investments designed to improve those practices. In the following illustrative computations, the impact of a one-standard deviation change in the value of each of the complexity variables is computed for a project of 5416 SLOC (the site mean) with average complexity values. The effects of PROCSIZE on HOURS is estimated in the regression model as follows:

$$0.00012 * PROCSIZE^2 * SLOC - 0.0106 * PROCSIZE * SLOC$$

Solving this equation once for the mean value of PROCSIZE (43) and once for a one-standard deviation increase in PROCSIZE (to 61), and then subtracting the first result from the second results in a difference of 183.28 hours, or an increase of 20% of the average project cost of 937 hours. A similar calculation for a decrease of one standard deviation in PROCSIZE (to 25) is 25%.[14] The calculations for MODLSIZE and GOTOFAR are similar, and the results are shown in Table 8.

Another way to use the results of the model for managerial planning is to estimate the aggregate cost impact to the organization of software complexity. To do this a manager might postulate the following question: What would be the estimated cost savings if the more complex systems were improved, not to some *optimal* level, but merely to the current *average* level of all systems?

The measurement of the individual systems and the model can be used to develop such an estimate. The first step is to note that the current actual projects have an average cost of 937 hours. The second step is to modify the data set in the following manner: test each of the three complexity variables for each of the 65 projects to determine whether it is of higher complexity than average. If it is, replace that value with the average complexity value. If not, leave it unchanged. Once this transformation of the data is complete, the model is used to estimate the cost of hypothetical projects based on the transformed data, which gives a predicted cost of 704 hours for an average project, a 25% savings over the actual situation.

In order to determine the estimated dollar value to the organization of this reduction in complexity, a "back-of-the-envelope" calculation of the estimated aggregate possible savings can be done. Two assumptions are necessary for this calculation to be valid, (1) that the projects studied represent a typical mix (believed to be the case), and (2) that maintenance projects represent 70% of the budget (also true for this site). The result is that improving the site's more poorly written systems, not to optimality, but merely to the level of the site's average complexity, could result in an aggregate savings of more than 17% (.25 * .7) of the applications software budget, which at this site translates into a savings of several million dollars in the year following such an improvement.

These quantified impacts of complexity can help software maintenance managers make informed decisions regarding preferred managerial practice. For example, one type of decision that could be aided by such information is the purchase of CASE tools for code restructuring. The benefits of these tools have generally had to be taken on faith. The current analysis, however, indicates that the magnitude of the economic impact of software complexity is sufficiently great that many organizations may be able to justify the purchase and implementation of CASE tools for code restructuring on the basis of these estimated benefits.

More generally, a common belief in the long-term importance of good programming practice has not been powerful enough to stand in the way of expedience when "quick-and-dirty" programming has been perceived to be needed immediately. An awareness of the magnitude of the cost of existing software complexity can combat this tendency. The cost of software complexity at this research site is the legacy of the practices of previous years.

Taken together, these ideas show

[14]Note that these results are not symmetric as the site mean is not identical to the optimum value.

Table 8. Estimated cost impacts

Name	Mean, std. dev.	Impact of a 1 std. dev. variation . . . . . . in hours	. . . as a % of total
PROCSIZE	43, 18	183, 238	20%, 25%
MODLSIZE	681, 164	98	10%
GOTOFAR	.024, .016	114	12%

how, through the use of the model developed here, managers can make decisions *today* on systems design, systems development, and tool selection and purchase that depend on system values that will affect *future* maintenance. This model can be a valuable addition to the traditional exclusive emphasis on software development project schedules and budgets because it allows for the estimation of full life-cycle costs. Given the significant percentages of systems resources devoted to maintenance, improving managers' ability to forecast these costs will allow them to be properly weighted in current decision making.

As with any empirical study, some limitations of these research results must be observed. The results were found to exist in a site which, due to its size, software tools, hardware tools, and application type, is typical of a large number of commercial IS applications, particularly financial transaction processing systems. However, additional studies at other sites, especially maximally dissimilar sites with applications such as real-time command and control applications should be done before claims can be made about the overall generalizability of these results. Also, values of specific parameters, such as the optimal number of SLOC/module, are likely to differ with programming languages,[15] particularly nonthird-generation languages.

In summary, this research suggests that considerable economic benefits can be expected from adherence to appropriate programming practices. In particular, aspects of modern programming practice, such as the maintenance of moderate procedure size and the avoidance of long branching, seem to have great benefits. The informed use of tools or techniques that encourage such practices should have a positive net benefit.

Concluding Remarks

This study has investigated the links between software complexity and software maintenance costs. On the basis of an analysis of software maintenance projects in a commercial application environment, it was confirmed that software maintenance costs are significantly affected by the levels of existing software complexity. In this study, software maintenance costs were found to increase with increases in the complexity of a system's implementation, as measured by its average procedure size, average module size, and its branching complexity.

Historically, most models of software economics have focused on new development. Therefore, they have not used software complexity metrics. After controlling for project factors believed to affect maintenance costs, the analysis at this site suggests that high levels of software complexity account for approximately 25% maintenance costs or more than 17% of total life-cycle costs. Given the extremely high cost of maintenance in commercial applications, the neglect of software complexity is potentially a serious omission.

The results presented here are based on a detailed analysis of maintenance costs at a site judged to be typical of traditional transaction processing environments. These types of environments account for a considerable percentage of today's software maintenance costs. Based on this analysis, the aggregate cost of poor programming practice for industry as a whole is likely to be substantial.

Appendix A. Data Collection and Analysis in a Commercial Maintenance Environment

The main body of this research presents a generalizable model that has been estimated in a specific environment. Researchers and practitioners who are interested in validating this work in other environments will need to follow a series of three steps: data requirements analysis; metrics selection; data modeling and interpretation.

Data requirements analysis begins with determining the research goals that will drive the modeling process. This is, in many ways, the most critical of the three steps. In the current research the focus was on measuring the impact of complexity on maintenance productivity, and therefore the data requirements included cost (effort), project size, application complexity, and other nonsize project factors (See Figure 2 in the text.)[16] Each of these may require multiple metrics, as was done in the current research. In particular, the use of the two size variables and their second order terms in the current model, while requiring additional observations, does allow for a better fit of the data.

There are a number of sources for use in determining an appropriate set of complexity metrics from the universe of available metrics, for example, the work by Basili and Weiss [7]. Despite the large number of metrics proposed in the literature, most of them are highly correlated and a relatively small number will cover the major identified dimensions. In the current work the emphasis has been on metrics that reflect the adherence (or its lack) to structured programming precepts, and that are relatively easy for maintainers to understand and use.

Finally, it is very important not to lose sight of the need to collect data on the other project factors. These factors are likely to have a significant impact on the results, and failure to include them is likely to result in a misspecified model. The appropriate measures are likely to be highly site-specific, but the general areas are likely to include project size (in a maintenance environment, functionality added or changed), staffing, process or tool variables, and project quality. A data site that has already devoted some time to formal project cost estimation is likely to have a head start on what the important factors are likely to be.

Once the data requirements are determined, great care must be taken in data collection to ensure the data accurately represent the project experience. (Some of these issues are discussed in [23].) Generally speaking, *ex post* collection of metrics is often difficult, and the final sample size will typically be significantly

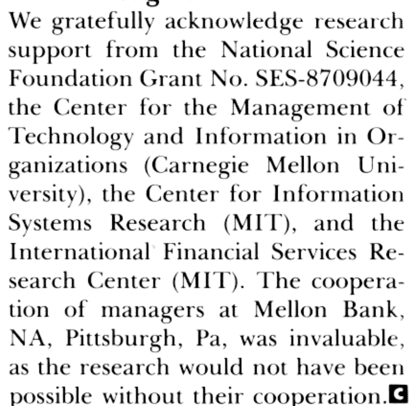
[15] Although it is interesting to note that the optimal value of statements/module found here for Cobol code, 44, is similar to the maximum size heuristic used at Toshiba in Japan for Fortran code, 50 (See [26], pp. 45–52.

[16] In other environments the center of the investigation might be on other areas, such as defect detection and prevention, which would generate a different list.

smaller than the initial universe of projects selected. This phenomenon should be taken into account when outlining the scope of the research effort, and, in particular, when specifying the model. A significant amount of effort will need to be devoted to initial data collection, both on the part of the research team and on team members from the project being studied, in order to locate and verify data on completed projects. Organizations that follow a disciplined methodology in their software development and maintenance will generally have more success at collecting these data than organizations that do not.

Given careful attention to the earlier steps, model estimation can proceed quite directly. The general form of the model used in the current research, a multiplicative formulation, is a general model that can be expected to be applicable in a wide variety of software engineering settings. Interpreting the model results can proceed as described in the main text. Feedback of these results to the participants at the data site is a critical step as a check to ensure that there are no extramodel factors influencing the results. Additionally, this feedback can help to ensure that the results of the model are actually used to modify and improve the process of software maintenance at the site.

Acknowledgment



We gratefully acknowledge research support from the National Science Foundation Grant No. SES-8709044, the Center for the Management of Technology and Information in Organizations (Carnegie Mellon University), the Center for Information Systems Research (MIT), and the International Financial Services Research Center (MIT). The cooperation of managers at Mellon Bank, NA, Pittsburgh, Pa, was invaluable, as the research would not have been possible without their cooperation.

References

1. An, K.H., Gustafson, D.A. and Melton, A.C. A model for software maintenance. *Conference on Software Maintenance* (1987), pp. 57–62.
2. Banker, R.D., Datar, S.M. and Kemerer, C.F. A model to evaluate variables impacting productivity on software maintenance projects. *Manage. Sci. 37*, 1 (1991) 1–18.
3. Banker, R.D. and Kemerer, C.F. Scale economies in new software development. *IEEE Trans. Softw. Eng. SE-15*, 10 (1989), 416–429.
4. Basili, V.R. Quantitative software complexity models: A panel summary. In V.R. Basili, Ed., *Tutorial on Models and Methods for Software Management and Engineering*, IEEE Computer Society Press, Los Alamitos, Calif, 1980.
5. Basili, V.R. and Hutchens, D.H. An empirical study of a syntactic complexity family. *IEEE Trans. Softw. Eng. SE-9*, 6 (1983), 664–672.
6. Basili, V.R. and Perricone, B. Software errors and complexity: An empirical investigation. *Commun. ACM 27*, 1 (1984), 42–52.
7. Basili, V.R. and Weiss D.M. A methodology for collecting valid software engineering data. *IEEE Trans. Softw. Eng. SE-10*, 6, 728–738.
8. Belsley, D.A., Kuh, E. and Welsch, R.E. *Regression Diagnostics*. Wiley and Sons, New York, 1980.
9. Boehm, B., *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N.J., 1981.
10. Bowen, J.B. Module size: A standard or heuristic? *J. Syst. Softw. 4* (1984) 327–332.
11. Boydston, R.E., Programming cost estimate: Is it reasonable? In *Proceedings of the Seventh International Conference on Software Engineering* (1984), pp. 153–159.
12. Card, D.N., Page, G.T. and McGarry, F.E. Criteria for software modularization. In *Proceedings of the Eighth International Conference on Software Engineering* (1985), pp. 372–377.
13. Conte, S.D., Dunsmore, H.E. and Shen, V.Y. *Software Engineering Metrics and Models*. Benjamin-Cummings, Reading, Mass., 1986.
14. Curtis, B., Shepperd, S. and Milliman, P. Third time charm: Stronger prediction of programmer performance by software complexity metrics. In *Proceedings of the Fourth International Conference on Software Engineering* (1979), pp. 356–60.
15. Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A. and Love T. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Trans. Softw. Eng. SE-5*, 2 (1979), 96–104.
16. Freedman, D.H. Programming without tears. *High Tech. 6*, 4 (1986), 38–45.
17. Gibson, V.R. and Senn, J.A. System structure and software maintenance performance. *Commun. ACM 32*, 3 (1989), 347–358.
18. Gill, G.K. and Kemerer, C.F. Cyclomatic complexity density and software maintenance productivity. *IEEE Trans. Softw. Eng. 17*, 12 (1991), 1284–1288.
19. Gremillion, L.L. Determinants of program repair maintenance requirements. *Commun. ACM 27*, 8 (1984), 826–832.
20. Jeffery, D.R. and Lawrence, M.J. Managing programming productivity. *J. Syst. Softw. 5* (1985), 49–58.
21. Kearney, J. et al. Software complexity measurement. *Commun. ACM 29*, 11 (1986), 1044–1050.
22. Kemerer, C.F. Measurement of software development productivity. Ph.D. thesis, Carnegie Mellon University, 1987.
23. Kemerer, C.F. An agenda for research in the managerial evaluation of computer-aided software engineering tool impacts. In *Proceedings of the Twenty-Second Hawaii International Conference on System Sciences*, (Jan. 1989), pp. 219–228.
24. Korson, T.D. and Vaishnavi, V.K. An empirical study of the effects of modularity on program modifiability. In *Empirical Studies of Programmers*, E. Soloway, and S. Iyengar, Eds., Ablex, Norwood, N.J., 1986.
25. Lind, R. and Vairavan K. An experimental investigation of software metrics and their relationship to software development effort. *IEEE Trans. Softw. Eng. 15*, 5 (1989), 649–653.
26. Matsumura, K., Furuya, K., Yamashiro, A. and Obi, T. Trend toward reusable module component: Design and coding technique 50SM. In *Proceedings of the Eleventh Annual International Computer Software and Applications Conference (COMPSAC)*, (Tokyo, Japan, Oct. 7–9 1987), pp. 45–52.
27. Munson, J.C. and Koshgoftaar, T.M. The dimensionality of program complexity. In *Proceedings of the International Conference on Software Engineering* (1989), pp. 245–253.
28. Schneidewind, N.F. The state of software maintenance. *IEEE Trans. Softw. Eng. SE-13*, 3 (1987), 303–310.
29. Spratt, L. and McQuilken, B. Applying control-flow metrics to COBOL. In *Proceedings of the Conference on Software Maintenance* (1987), pp. 38–44.
30. Vessey, I. and Weber, R. Some factors affecting program repair maintenance: An empirical study. *Commun. ACM 26*, 2 (1983), 128–134.
31. Vessey, I. and Weber, R. Research on

structured programming: An empiricist's evaluation. *IEEE Trans. Softw. Eng. SE-10*, 4 (1984), 394–407.
32. Zweig, D. Software complexity and maintainability. Ph.D. thesis, Carnegie Mellon University, 1989.

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**]: Trade-offs 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: Software complexity, software economics, software maintenance, productivity

About the Authors:
RAJIV D. BANKER is the Arthur Andersen Chair in accounting and information systems at the Carlson School of Management, University of Minnesota. His research interests are in the management of IS development, and competitive value of information technology deployment. **Author's Present Address:** University of Minnesota, Carlson School of Management, 271 19th Avenue South, Minneapolis, Minnesota 55455

SRIKANT M. DATAR is a professor at the Stanford University Graduate School of Business. **Author's Present Address:** Stanford University, Room 276, Graduate School of Business, Littlefield Management Center, Stanford, California 94305

CHRIS F. KEMERER is the Douglas Drane Career Development associate professor of information technology and management at the MIT Sloan School of Management. His research interests are in the measurement and modeling of software development. **Author's Present Address:** MIT Sloan School of Management, E53-315, 50 Memorial Drive, Cambridge, MA 02139

DANI ZWEIG is an assistant professor at the Naval Postgraduate School, Monterey, California. His research interests include software metrics and software reuse. **Author's Present Address:** Naval Postgraduate School, Code AS/Zg, Administrative Science Dept., Ingersoll Hall #231, Monterey, CA 93943-5000; email: dani@netcom.com

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

© ACM 0002-0782/93/1100-081 $1.50

CONTINUED FROM PAGE 49

benchmark comparison. In *comp. benchmarks*, Nov. 12, 1990.
21. Thaik, R.W., Lek, N. and Kang, S.-M. A router using zero-one integer linear programming techniques for sea-of-gates and custom logic arrays. *IEEE TCAD-ICS 11*, 12 (Dec. 1992) 1495–1507.
22. Thinking Machines Corp., Cambridge, Mass. The connection machine CM-5 Tech. Sum., Jan. 1992.

CR Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems; D.3.4 [**Programming Languages**]: Processors; K.6.2 [**Management of Computing and Information Systems**]: Installation Management; K.6.3 [**Management of Computing and Information Systems**]: Software Management
General Terms: Economics, Performance
Additional Key Words and Phrases: Vector supercomputer

About the Author:
CLARK D. THOMBORSON is a professor of computer science at the University of Minnesota. His research is concerned with the design, implementation and evaluation of algorithms which are optimized for supercomputers, workstations, microprocessors, or semicustom VLSI. He is particularly interested in pseudorandom number generation and data structures for supercomputers, statistical inference, and computer arithmetic for VLSI. **Author's Present Address:** Computer Science Dept., University of Minnesota, Duluth, MN 55812; email: cthombor@ua.d.umn.edu

Cray X-MP, Y-MP, and Y-MP C90 are registered trademarks of Cray Research Inc. The term Cray in this article refers to any of these products.

This research was supported in part by the National Science Foundation, through its Design, Tools and Test Program under grant number MIP 9023238, by a visiting professorship of EECS at the Massachusetts of Technology, and by the Minnesota Supercomputer Institute.

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

© ACM 0002-0782/93/1100-041 $1.50