SOFTWARE COMPLEXITY AND SOFTWARE MAINTENANCE COSTS

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

April 1990

CISR WP No. 208
Sloan WP No. 3155-90

Center for Information Systems Research
Massachusetts Institute of Technology
Sloan School of Management
77 Massachusetts Avenue
Cambridge, Massachusetts, 02139

SOFTWARE COMPLEXITY AND SOFTWARE MAINTENANCE COSTS

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

April 1990

CISR WP No. 208
Sloan WP No. 3155-90

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

Software Complexity and Software Maintenance Costs

Abstract

In an empirical analysis of software maintenance projects at a large IBM/COBOL transaction processing environment the impacts of software complexity upon project costs were estimated. Program size, modularity, and the use of branching were all found to significantly affect software maintenance costs. It was estimated that the maintenance projects dealing with more complex software were at a cost disadvantage of approximately 35% with respect to the site's average projects, and the disadvantage with respect to more favored projects is approximately twice that large. These costs amount to several millions of dollars a year at this site alone. A generalizable model is provided to allow researchers and managers to estimate these costs in other environments.

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: Software Productivity, Software Maintenance, Software Complexity.

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.

I. Introduction

With software costs now exceeding $200 billion annually and with most of that being spent on maintenance of all types, rather than new development, the economic incentives to develop software that requires less repair maintenance and is more easily adapted to changing requirements are quite strong [Boehm, 1979, 1987; Gallant, 1986]. In this paper, we test and measure the degree to which the maintainability of a system is influenced by the complexity of the existing code. In particular, we investigate the impact of software complexity upon the productivity of software maintainers.

The empirical evidence linking software complexity to software maintenance costs has been criticized as being relatively weak [Kearney, *et al.*, 1986]. Much of it is based on experiments involving small programs [e.g., Curtis *et al.*, 1979], or is based upon analysis of programs written by students [e.g., Kafura and Reddy, 1987]. Such evidence can be valuable, but several researchers have noted that caution must be used in applying these results to the actual commercial application systems which account for most software maintenance expenditures [Conte, *et al.*,1986, p. 114; Gibson and Senn, 1989]. And, the limited field research that has been done has generated either no or conflicting results; for example, in the case of degree of program modularity [Vessey and Weber, 1983, Basili and Perricone 1984, Card *et al.*, 1985], and in the case of program structure (see Vessey and Weber's 1984 review article.). Finally, none of the previous work develops estimates of the actual cost of complexity, estimates which could be used by software maintenance managers to make best use of their resources. Research supporting the statistical significance of a factor is a necessary first step in this process, but practitioners must also have an understanding of the magnitudes of these effects if they are to be able to make informed decisions regarding their control.

This study analyzes the effects of software complexity upon the costs of COBOL maintenance projects within a large commercial bank. Freedman notes that 60% of all business expenditures on computing are for maintenance of COBOL programs, and that there are over 50 billion lines of COBOL in existence worldwide, the maintenance of which, therefore, represents an information systems activity of considerable economic importance [1986]. Using a previously developed model of software maintenance productivity [Banker, Datar and

Kemerer, 1990[1]] we estimate the *marginal* impact of software complexity upon the costs of software maintenance projects in a data processing environment. The analysis confirms that software maintenance costs are significantly affected by software complexity, as measured by three metrics: a measure of module size, a measure of modularity, and a measure of control structure complexity. The results further suggest that the magnitudes of these costs are such that software maintenance managers should monitor the complexity of the software under their control, and take active steps to reduce that complexity.

This research makes contributions in two distinct areas. The first is in developing a model with which to resolve some current academic debate regarding the nature of the impact of software complexity, and the shape of the functional form relating complexity to the productivity of software maintainers. The second is in providing practicing software maintenance managers with a predictive model with which to evaluate the future effects of software design decisions. This model could also be used to assist in the cost-benefit assessment of a class of computer-aided software engineering (CASE) tools known as restructurers.

The remainder of this paper is organized as follows. Section II outlines the research questions, and summarizes previous field research in this area. Section III describes our research approach and methodology, and section IV presents our model and results. Implications for practitioners are presented in section V, and concluding remarks and suggestions for future research are provided in the final section.

II. Research Questions

Complexity and maintenance

The *complexity* of a software system is said to increase as "the number of control constructs grows and as the size in the number of modules grows" [Conte, *et al.*, 1986, p. 109.]. The formal characterization of the maintenance impacts of software complexity is sometimes ascribed to Belady and Lehman, who, in their Evolution Dynamics theory, propose that software systems, like their analogues in other contexts, face increasing entropy over time [Belady and Lehman, 1976]. As more changes are made to a system in the form of maintenance requests, the initial design integrity deteriorates, and the system's complexity increases. In addition, several longitudinal studies have noted increases in the size of software

[1]Hereafter referenced as "BDK, 1990".

systems that are in active use [Lawrence, 1982, Chong, 1987]. These two factors have been suggested to contribute to the increasing difficulty of software maintenance over time. Given the growing economic importance of maintenance, researchers have attempted to empirically validate these theories. In general, however, researchers have not been able to empirically test the impact of complexity upon maintenance effort while controlling for other factors known to affect costs. Therefore, our overall research question (to be developed into specific testable hypotheses below) will be:

Research question 1: Controlling for other factors known to affect software maintenance project costs, what is the impact of software complexity upon the productivity of software maintenance projects?

Size and Modularity

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" [Conte, *et al.*, 1986, p. 197]. Freedman and Weinberg have estimated that 75-80% of existing software was produced prior to significant use of structured programming [Schneidewind, 1987], 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 productivity with data from actual systems, and the results of this research are summarized in Table 1.

Table 1: Previous Field research on Modularity

Year	Researchers	Data	Dependent Variable	Conclusions[2]
1983	Vessey & Weber	COBOL	Number of repairs	Unidirectional +,0
1984	Basili & Perricone	Fortran	Errors/KSLOC	Unidirectional -
1984	Bowen	Algol, CMS, etc	McCabe, Halstead metrics	Suggests 2-way relationship
1984	Boydston	Assembler, PLS	Effort	Suggests 2-way relationship
1985	Shen, *et al.*	Pascal, PLS etc	Problem reports	Suggests 2-way relationship
1985	Card, *et al.*	Fortran	Effort	Unidirectional -,0
1987	An, *et al.*	C	Change data	Unidirectional, -?
1989	Lind & Vairavan	Pascal, Fortran	Normalized change data	Suggests 2-way relationship

[2]For unidirectional tests, "+" indicates that greater modularity (more, smaller modules) improved performance, "-" indicates that less modularity (fewer, larger modules) improved performance, and "0" indicates mixed or no results. A 2-way relationship is one in which both positive and negative deviations from optimal module size reduce performance.

Perhaps the first widely disseminated field research in this area was by Vessey and Weber, in their study of repair maintenance in Australian and US data processing organizations [1983]. Their work relied on subjective assessments of the degree of modularity in a large number of COBOL systems. In one dataset they found that more modular code was associated with fewer repairs, in the other dataset no effect was found. Basili and Perricone, in an analysis of a large Fortran system, found more errors per thousand source lines of code (KSLOC) in smaller modules, which they hypothetically attributed to a) greater numbers of interface errors, b) possible greater care taken in coding larger modules, or c) simply the continued presence of undiscovered errors in larger modules [1984]. Shen, *et al.* disagreed with Basili and Perricone's analysis, noting that the higher error rate observed with smaller modules could be simply a function of an empirically observed phenomenon that modules contain a number of errors independent of size, in addition to a size-related error rate. Therefore, according to this model, smaller modules will show a higher rate of errors due to this size-independent error component being divided by a smaller number of lines of code. Shen, *et al.* conclude that *"...it may be beneficial to promote programming practices related to modularization that discourage the development of either extremely large or extremely small modules."* [1985, p. 323].

Bowen, in an analysis of secondary data, compared the number of SLOC / module with a set of assumed maximum values of two well-known complexity metrics, McCabe's V(G) and Halstead's N [1984]. He concluded that the optimal values of SLOC / module differed across languages, but that all were much less than the DoD's proposed standard of 200 SLOC / module. In his suggestions for future research, he notes that *"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 intermodule complexity, complicate software integration testing, and increase computer resource overhead."* [1984, p. 331] Boydston, in his analysis of programmer effort, noted 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."* [1984, p. 159]. Card, Page, and McGarry tested the impact of module size and strength (singleness of purpose) on programming effort [1985]. In their basic analysis, they found that effort decreased as the size of the module increased. However, they also noted that effort decreased as strength increased, but that increases in strength were associated with decreases in module size. Their conclusion was that nothing definitive could be stated about the impact of module size.

An, Gustafson, and Melton, in analyzing change data from two releases of UNIX, found that the average size of unchanged modules (417 lines of C) was larger than that of changed modules (279 lines of C) [1987]. Unfortunately, the authors do not provide any analysis to determine if this difference is statistically significant. Most recently, Lind and Vairavan analyzed the change rate (number of changes per 100 lines of code) versus a lines of code-based categorical variable [1989]. They found that minimum values of change density occurred in the middle of their ranges, suggesting that modules that were both too large and too small increased the amount of change density. They further suggest that, for the Pascal and Fortran programming languages, the optimum value might be between 100 and 150 SLOC.

The results of these previous studies can be summarized as follows. Researchers looking for unidirectional results (i.e., that either smaller modules or larger modules were better) have found no or contradictory results. Other researchers have suggested that a U-shaped function exists, that is, both modules that are too small and modules that are too large are problematic. In the case of many small modules, the number of intermodule interfaces is increased, and interfaces have been shown to be among the most problematic components of programs [Basili and Perricone, 1984]. In the case of a few very large modules, these modules are less likely to be devoted to a single purpose and may be assumed to be more complex, both of these factors having been linked with larger numbers of errors and therefore higher maintenance costs [Card, *et al.* 1985, Vessey and Weber, 1983].

However, the researchers who have suggested the U-shaped curve hypothesis either provide no or very limited (e.g., categorical) data linking size and cost . They also, in general, do not provide a methodology for determining the optimum program size.[3] Finally, previous research on software complexity metrics has suggested that, for large systems, modularity is most appropriately measured at multiple levels of program organization [Zweig, 1989]. This is because, as will be explained in greater detail in Section III below, the effects of breaking an application into modules of an appropriate size are distinct from those of breaking those modules into their component subprograms or procedures [Zweig, 1989]. Therefore, the research question to be addressed is:

Research question 2: Do software maintenance costs depend significantly upon degree of modularity, measured at multiple levels, with costs rising for applications that are either under or over modularized?

---

[3]Boydston does extrapolate from his dataset to suggest a specific square root relationship between number of new lines code and number of modules for his Assembler and PLS language data [1984].

Structure

An excellent review of the empirical research on structured programming is provided by Vessey and Weber (1984). Therefore, this section will only briefly summarize the arguments presented there. Structured programming is a design approach that limits programming constructs to three basic control structures. Because these structures are often difficult to adhere to using the GOTO syntax found in older programming languages, this approach is sometimes colloquially referred to as "GOTO-less programming". Vessey and Weber note that, while few negative results have been found, absence of significant results is as frequent as a finding of positive results, a development that they attribute, in part, to the fact that researchers have not adequately controlled for other factors. They note the difficulty of achieving such control, particularly in non-laboratory, real world settings. Therefore, the question of a positive impact of structure on maintenance costs is still unanswered, and requires further empirical support. This suggests the following research question:

Research question 3: Do software maintenance costs depend significantly upon the degree of control structure complexity, with costs rising with increases in complexity?

In the following section we describe our approach to answering these research questions.

III. Research Approach

In attempting to answer the research questions posed above, we needed to test the impact of complexity on real-world systems, and to attempt to control for other factors that may have an impact on labor productivity, since labor costs are the single largest cost component in commercial software maintenance. For this purpose we began with the data and model developed in our previous research in software maintenance productivity. The data collection procedures and model development are described in detail in [Kemerer, 1987] and [BDK, 1990], and will only be summarized here.

The Research Site

Data were collected at a major regional bank with a large investment in computer software. The bank's systems contain over 10,000 programs, totalling over 20 million lines of code. Almost all of them are COBOL programs running on large IBM mainframe computers. The programs are organized into application systems (e.g. Demand Deposits) of typically 100 - 300 programs each. Some of the bank's major application systems were written in the mid-1970's, and are generally acknowledged to be more poorly designed and harder to maintain than more recently written software.

The software environment in which we are conducting our research is a quite typical commercial data processing environment. The empirically based results of the research should, therefore be highly generalizable to other commercial environments. The projects analyzed were homogeneous in that they all affected COBOL systems, so our results are not confounded by the effects of multiple programming languages.

We analyzed 65 software maintenance projects from 17 major application systems (see Table 2 in Section IV). These projects were carried out between 1985 and 1987. An average project took about about a thousand hours (at an accounting cost of $40 per hour) and changed or created about five thousand source lines of code.

Modeling maintenance productivity

Our major goal in this study is to evaluate the impact of software complexity on maintenance labor productivity. In order to do so, however, we must control for the effects of other factors, such as task magnitude and the skill of the developers, that also affect the developer hours required on a project. Excluding task size (or other relevant factors) would result in a mis-specification of the model and incorrect inferences about the impact of software complexity on costs. For example, a large maintenance project dealing with an application system of low complexity may require more hours than another project meant to make a small modification to a system of higher complexity. A failure to control for the different task sizes could lead us to the unjustified conclusion that higher software complexity will result in lower costs.

Figure 1 presents a measurement model of the maintenance function developed in [BDK, 1990]. Software maintenance is viewed as a production process whose inputs are labor and computing resources and whose output is the modified system. Since labor hours are considerably more expensive than computer resources, and there are limited substitution possibilities between the two, we focus upon labor hours as the major expense incurred in software maintenance. The productivity of this process depends upon a number of environmental variables, including the skill and experience of the developers, and the software tools available to the developers [Boehm 1987].

Activity	Output Measure	Input Measure
Analysis/Design	Function Points	Total Labor Hours
Coding/Testing	Source Lines of Code	

Figure 1: Measurement Model

We write labor hours required as a function of the task requirement and environmental factors:

> Labor Hours = f(Task magnitude and complexity, Developer skill, Developer application experience, Working environment, Product quality, Software tools, Software complexity)

The unit of analysis for this model is the project as defined by the bank. Each maintenance project has its own task requirements and its own budget. For each such project, the following factors are considered:[4].

> *Task magnitude: The output of the software maintenance process is the modified system. The amount of change will naturally have a major effect on the amount of work required
>
> *Task complexity: Other things being equal, some maintenance tasks are simply more difficult and demanding than others. This may be because they demand a more sophisticated level of programming. It may be because the task specification includes more stringent reliability requirements. In either case, such a task may require more and better maintenance resources.
>
> *Developer skill: Previous research has found large differences in productivity between top rated developers and poorer ones.
>
> *Developer application experience: 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.
>
> *Working environment: There is evidence that fast-turnaround maintenance environments enhance developer productivity.
>
> *Product quality: It has been suggested that doing a careful job of error-free programming will naturally cost more than a rushed job would cost, although its benefits will be realized in the long term. And there are those who believe that careful and systematic programming does not take any longer; some even argue that it should be less expensive.

[4]The interested reader is referred to [BDK, 1990] for references to each of these factors.

*Software tools: Many commercially available products have been designed to increase developer productivity. To the extent that they do so, they will have noticeable beneficial effects upon maintenance costs.

*Software complexity: We are primarily concerned here with the impact of this factor upon maintenance costs. Any practical cost estimation model, however, must consider and control for the effects of other factors such as those discussed above.

This model of software maintenance costs has already been tested [BDK, 1990] at the research site without the software complexity variables. The model was explicitly designed to allow the introduction of new factors, and by introducing software complexity we can confirm its robustness. Of more immediate interest is that we can test the marginal impact of software complexity upon maintenance costs. And we can compute the actual magnitude of the cost impact of complexity, so as to determine the extent to which the effect is of managerial interest.

Definitions

The following definitions will be used throughout the rest of this paper:

**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. All the modules counted in the analysis were of this type. Modules containing COBOL source code but not the headers which allow it to be run on its own (e.g., INCLUDE modules and COPY files) were not included.

**Paragraph:* The smallest addressable unit of a COBOL program. A sequence of COBOL statements preceded by an address/identification label. This construct is not precisely paralleled in other high level languages.

**Procedure:* The range of a PERFORM statement. For example, if paragraphs are labelled sequentially, the statement PERFORM D THRU G invokes the procedure consisting of paragraphs D, E, F, G and the paragraphs invoked by these paragraphs.

**Component*: The union of two or more overlapping procedures. (e.g., PERFORM D THRU G and PERFORM E THRU J will have at least E, F, and G in common.) Measurement of components prevents possible double counting. Such overlaps are relatively rare, however, with the result that components and procedures behave almost identically for all statistical purposes.

**Application System:* A set of modules assigned a common name by the bank, 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 and other material associated with it. 'Application' or 'system', if used separately, mean the same thing.

Software Complexity Metrics

A number of steps must be taken before it can be determined whether reductions in software maintenance costs can be achieved by monitoring and controlling software complexity. First,

we must identify appropriate metrics with which to measure software complexity. Having identified such measures, we can then attempt to establish that their effects are managerially important -- that they do in fact have a large enough effect upon software costs to justify possibly significant expenditures by those wishing to control them.

The first step was accomplished in an earlier study at the same research site [Banker, Datar and Zweig, 1989[5]]. We analyzed over five thousand application programs in order to develop a basis for selecting among dozens of candidate software metrics which the research literature has suggested. Although much research has been devoted to testing single software metrics in isolation, our analysis suggested that the metrics which we analyzed could be classified into three major groups, measuring three distinct dimensions of software complexity: measures of module size; measures of procedure complexity; and measures of the complexity of a module's control structure [BDZ, 1989]. That research also identified representative metrics from each group which could be expected to be orthogonal to each other [BDZ, 1989].

In this study we undertake the second step required to validate the practical use of software complexity metrics by assessing their effect upon maintenance costs. We used a commercial static code analyzer to compute metrics from each of the groups of metrics identified earlier. Three software complexity metrics, representing each of the previously identified dimensions, were used in this study. Choice of these three metrics was based upon the ease with which they could be understood by software maintenance management and the ease of their collection. Given the typical high levels of correlation among complexity metric groups [Zweig, 1989, Munson and Khoshgoftaar, 1989], this approach has been recommended by previous research [Shepperd, 1988][6]. Consistent with previous research, we used module length, in statements (STMTS) for the first metric, a measure of size[7]. The effect of this complexity metric will depend upon the application systems being analyzed. Module for module, larger modules will be more difficult to understand and modify than small ones, and maintenance costs will be expected to increase with module size. However, a system can be composed of too many small modules as easily as too few large ones. If modules are too small, a maintenance project

---

[5]Hereafter referenced as "[BDZ, 1989]".

[6]However, in order to test the sensitivity of our results to choices of alternative metrics, the model described below was re-estimated using other metrics. No significant changes in the results were found due to specific metric choice.

[7]For these data this metric is highly correlated (Pearson correlation coefficient > .92) with other size metrics, such as physical lines of code, and Halstead Length, Volume, and Effort. [Zweig, 1989]

will spread out over many modules with the attendant interface problems and therefore maintenance costs could actually decrease as module size increases.

For the second metric, to measure module complexity, we computed the average size of a module's procedures (STMTCOMP)[8]. The same argument concerning the effect of module size applies here. And if modules are broken into too many small procedures, then an increase in average component size will be associated with a decrease in maintenance costs. There is an almost universal tendency to associate large component size with poor modularity, but, intuitively, neither extreme is effective.

A third dimension of software complexity was the complexity of the module's control structure. The initial candidate metric chosen for this dimension was the proportion of the statements which were GOTO statements (GOTOSTMT) We selected a control structure metric which was normalized for module size, so that it would not be confounded with STMTS. This metric is also a measure of module decomposability, as the degree to which a module can be decomposed into small and simple components depends directly upon the incidence of branching within the module. Highly decomposable modules (modules with low values of GOTOSTMT) should be less costly to maintain, since a developer can deal with manageable portions of the module in relative isolation.

The density of GOTO statements (GOTOSTMT), like other candidate control metrics we examined, is a measure of decomposability -- each GOTO command makes a module more difficult to understand by forcing a programmer to consider multiple portions of the module simultaneously -- but 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 [Vessey, 1985]. Yet none of the structure metrics we examined differentiate between the two.

The modules we analyzed have a large incidence of GOTO statements (approximately seven per hundred statements) but if only a relatively small proportion of these are seriously affecting maintainability, then the GOTOSTMT metric may be too noisy a measure of control structure complexity. Empirically, over half of the GOTOs in these programs (19 GOTOs out of 31 in the average module) are used to skip to the beginning or end of the current paragraph. Such

---

[8]This metric was found to be uncorrelated with STMTS (coefficient = .10).

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) and a metric (such as GOTOSTMT) which does not distinguish between these and the less benign 40% of the branch commands will be understandably imperfect.

To avoid this problem, a modified metric was computed (GOTOFAR) which is the density of the non-trivial GOTO statements i.e., the 40% of the GOTO statements which extend outside the boundaries of the paragraph and which can be expected to seriously impair the maintainability of the software. (Since the automated static code analyzer was not able to compute this metric, it was computed manually. Due to the large amount of time this computation required, this metric was not computed for all the modules analyzed, but for a random sample of approximately fifty modules per application system (about 1500 modules in total, or approximately 30% of all modules))[9].

Research Hypotheses

Based on the above research approach, we propose four specific research hypotheses based on the initial research questions that can be empirically tested:

Hypothesis 1: Controlling for other factors known to affect software maintenance costs, software maintenance productivity increases significantly with increases in software complexity, as measured by STMTS, STMTCOMP and GOTOFAR.

Hypothesis 2: Software maintenance costs will depend significantly upon average module size as measured by STMTS, with costs rising for applications whose average module size is either too large or too small.

Hypothesis 3: Software maintenance costs will depend significantly upon average procedure size as measured by STMTCOMP, with costs rising for applications whose average procedure size is either too large or too small.

Hypothesis 4: Software maintenance costs will depend significantly upon the density of branching as measured by GOTOFAR, with costs rising with increases in the incidence of branching.

---

[9] A sensitivity analysis regression using GOTOSTMT instead of GOTOFAR lends credence to our 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 four times as large.

IV. Model and Results

Factors Affecting Maintenance Costs

In assessing the effect of software complexity upon maintenance costs, it is necessary to control for other factors known to affect these costs. The most significant of these, of course, is the magnitude of the maintenance task. To control for this, and for other factors known to affect costs, we began with a previously developed model of software maintenance costs [BDK, 1990].

In testing the various complexity metrics, we shall be interested in their impact upon maintenance costs controlling for these other factors. To do so, we shall estimate the following model:

$$\begin{aligned}HOURS = {} & \beta_0 + \beta_1*FP + \beta_2*SLOC + \beta_3*FP*FP + \beta_4*SLOC*SLOC + \beta_5*FP*SLOC + \\ & \beta_6*FP*LOWEXPER + \beta_7*FP*SKILL + \beta_8*FP*METHODOLOGY + \\ & \beta_9*SLOC*QUALITY + \beta_{10}*SLOC*RESPONSE + \\ & \beta_{11}*SLOC*STMTS + \beta_{12}*SLOC*STMTCOMP + \beta_{13}*SLOC*GOTOFAR + \varepsilon\end{aligned}$$

This model, without the three complexity terms (the terms associated with parameters β_{11} through β_{13}), has been previously validated at the research site [BDK, 1990]. In this model, project costs (measured in developer HOURS) are primarily a function of project size, measured in function points (FP) and in source lines of code (SLOC). The number of hours was obtained from the site's billing files. The size measures were computed by the development staff after the projects were complete. In order to model the known nonlinearity of development costs with respect to project size, we include not only FP and SLOC, but also their second-order terms. We expect this to result in a high degree of multicollinearity among the size variables which will make the interpretation of their coefficients difficult [Banker and Kemerer, 1989]. Those coefficients, however, are of no concern to us for examining the current research hypotheses relating to the impact of complexity.

Other factors, shown to be significant in affecting project costs included:

*METHOD: The use of a structured design methodology. (A binary variable.) This is expected to have an adverse effect upon single-project productivity, although it is meant to reduce costs in the long run. [BDK, 1990]

*RESPONSE: The availability of a fast-turnaround programming environment. (A binary variable.) [BDK, 1990]

The values of these binary variables were obtained by interviewing developers and project managers.

> *SKILL: The percent of developer hours billed to the most highly skilled (by formal management evaluation) developers. This variable is quite distinct from the following one, which depended upon the developers' experience with a specific application system. [BDK, 1990]
>
> *LOWEXPER: The extensive use (over 90% of hours billed to the project) of developers lacking experience with the application being modified. (A binary variable.) [BDK, 1990]

The values of these variables depended upon the number of hours billed to each project. This information was obtained from the project billing files. [BDK, 1990]

> *QUALITY: A measure (on a three-point scale of low/medium/high quality) of the degree to which the completion of the project was followed by an increase in the number of operational errors. This measure was based upon information obtained from the site's error logs. [BDK, 1990]

These explanatory factors are weighted by a measure of project size, either by FP or by SLOC, depending on whether they are thought to be associated more strongly with the analysis phase or with the coding phase of the project. In a manner consistent with the software productivity literature [Boehm, 1981; Albrecht and Gaffney, 1983] we model the effects of these factors to be proportional, rather than absolute, so they are weighted by program size[10]. Table 2 presents the summary statistics for this dataset.

[10]It should be noted that any collinearity which may exist between the weighted metrics and other independent variables which have been weighted by SLOC will cause us to underestimate the significance of the metric; the analysis presented below, therefore, is a conservative test.

Table 2: Maintenance Project Summary Statistics*

VARIABLE	MEAN	Stand. Deviation	MIN	MAX
HOURS	937	718	130	3342
FP	118	126	8	616
SLOC	5416	7230	50	31060
LOWEXPER	.68	.48	0	1
SKILL	63	34	0	100
METHOD	.29	.46	0	1
QUALITY	2.08	.53	1	3
RESPONSE	.63	.49	0	1
STMTS*	681	164	382	1104
STMTCOMP*	43	18	13	87
GOTOFAR*	0.026	0.02	0.0	0.07

*The values given for the complexity metrics are averages over multiple programs.

Analysis and Statistical Results

One additional extension to the original model was made before the research hypotheses related to complexity could be tested. We expect the relationship between maintenance costs and procedure size to be U-shaped, rather than monotonic, with costs being lowest for some optimal size and higher for larger or smaller sizes. It is inappropriate, then, to use the STMTCOMP metric directly. Rather, developer productivity should decline as this metric moves away from its optimum value. To model this effect we will add another term, $STMTCOMP^2$, and compute the joint effect of STMTCOMP and $STMTCOMP^2$ upon costs. Our revised model, then, is:

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

Using Ordinary Least Squares regression, we estimated the new model, which includes our measures of module, component size, and branching density. The results of this regression are presented in Table 3.

Table 3: Regression Results

VARIABLE	MEAN	COEFFICIENT	t
HOURS	937	n/a	n/a
Intercept	n/a	333	5.06
FP	118	3.21	2.09
SLOC	5416	0.34	6.57
FP*FP	29.6K	0.008	2.80
SLOC*SLOC	81M	-3.1E-6	-2.99
FP*SLOC	1.1M	-.0001	-1.34
FP*SKILL	8009	-.049	-3.50
FP*LOWEXPER	81	0.105	0.16
FP*METHOD	44	1.78	3.07
SLOC*QUALITY	11K	0.027	2.76
SLOC*RESPONSE	4341	-0.019	-1.17
SLOC*STMTS	3.7M	-1.2E-4	-3.33
SLOC*STMTCOMP	241K	-.011	-4.90
SLOC*STMTCOMP^2	12.0M	.00012	5.40
SLOC*GOTOFAR	152	1.307	3.22

$F_{14,50} = 31.07$. $R^2=.89$. Adjusted $R^2=.87$.

Testing for the marginal significance of the complexity metrics (β_{11}-β_{14}) we get:

$F_{4,50} = 12.01$ (p=.0001).

Testing for the joint significance of the two component size terms (β_{12}-β_{13}) we get:

$F_{2,50} = 12.51$ (p=.0001).

Although, for this sample, not all of the regression variables are highly significant, no attempt is being made to eliminate variables so as to achieve a more parsimonious fit. We are only interested in assessing the marginal impact of adding of complexity metrics.

The addition of the second-order complexity term, STMTCOMP^2, to the original model. allows us to compute the minimum points of the U-shaped curve through interpretation of the estimates of the coefficients of the model generated using OLS. (Since it is a quadratic relationship, the stationary point may be computed by dividing the coefficient of the linear term by twice that of the quadratic term.)

At this site, the minimum-cost component size was computed to be (0.011/(2*0.00012)) = 45 executable statements per component (See Table 3). This value is very close to the mean (43)

and to the median (40) for this organization. However, individual applications vary in average procedure size from 13 to 115 statements![11]

As an additional test of the robustness of these results, after determining the minimum value of 45, we developed a linear model incorporating two linear variables representing the deviations below and deviations above the optimum value. This model generated similar results (R^2=.90, adjusted R^2=.87, $F_{14,50}$ = 31.15).

Analogous to the U-shaped relationship between maintenance costs and procedure size, there is also reason to expect a similar U-shaped relationship between maintenance cost and module size. An additional model was tested, adding a second order term, $STMTS^2$. However, this relationship was not supported by the data at this site, as the second order term was found to be statistically insignificant. The resulting coefficients showed that all the application systems examined fell on the downward-sloping portion of the computed curve. In fact, a direct plotting of the data confirmed that the relationship was downward-sloping and approximately linear across the observed range of the data, so no second-order term was included for this complexity metric in the final model.

The Belsley, Kuh, Welsch test of multicollinearity [Belsley, *et al.*, 1980] did not show the complexity metrics to be significantly confounded with the other regression variables, so we may interpret their coefficients with relative confidence. We also detected no significant heteroskedasticity. This supports our decision to model the complexity effects in our regression as proportional ones, rather than use the unweighted metrics alone.[12]

Tests of the Research Hypotheses

This analysis confirms all four of our hypotheses:

Hypothesis 1 was the general hypothesis that, controlling for the other explanatory factors, software complexity has a significant impact upon software maintenance costs. This is confirmed. Recall

[11] As is often the case in this type of estimation [Banker and Kemerer, 1989] there was a high degree of multicollinearity between the linear term and the quadratic term, which required the computation of the minimum point to be taken with caution. An attempt to confirm the results of this computation by plotting the data directly also yielded a minimum point at 45. Sensitivity analysis, using different minimum points, showed the estimation to be insensitive to moderate variations in this value.

[12] If the complexity effects were not proportional to project magnitude, our use of the weighted metrics would cause our model to overestimate the costs of large projects, resulting in residuals negatively correlated with size.

$P(H_0: \beta_{11}=\beta_{12}=\beta_{13}=\beta_{14}=0)=0.0001$ as $F_{4,50}=12.02$.

Hypothesis 2 was that maintenance costs would be significantly affected by module size. This is confirmed.

$P(H_0: \beta_{11}=0)=0.001$ as $t_{50}=-3.33$.

We also tested for a U-shaped relationship between *module* size and software maintenance costs. The maintenance costs at this site the data tended to be linear over the observed range of module sizes, controlling for other factors. It should be noted that, while these data do not indicate a U-shaped relationship, they are not necessarily inconsistent with such a hypothesis. (The data can be seen as falling on the downward sloping arm of this U, with the possibility that had sufficiently large modules been available, that costs would again begin to rise.)

Hypothesis 3 was that maintenance costs would be significantly affected by procedure size. This hypothesis is confirmed by an F test on the joint effect of the two procedure-size terms. Recall

$P(H_0: \beta_{12}=\beta_{13}=0)=0.0001$ as $F_{2,50}=12.02$.

Again, we hypothesized a U-shaped relationship between *procedure* size and software maintenance costs. At this site the data are supportive of the U-shaped hypothesis, with actual application systems observed to fall on both arms of the U, and minimum costs observed for a procedure size of approximately 45 statements.

Hypothesis 4 was that maintenance costs would be significantly affected by the density of branch instructions within the modules. This is confirmed.

$P(H_0: \beta_{14}=0)=0.002)$ as $t_{50}=3.22$.

V. Software Maintenance Management Results

Through the above analysis we have estimated the effect of software complexity upon developer productivity in a maintenance environment. While it is a firmly established article of conventional wisdom that poor programming style and practices increase programming costs, there has been little empirical evidence to support this notion. As a result, efforts and investments meant to improve programming practices have had to be undertaken largely on faith. We have extended an existing model of maintainer productivity and used it to confirm the significance of the impact of software complexity upon productivity. We used a model which allowed us to not only verify this significance, but also to estimate the magnitude of the effect. The existence of such a model provides managers with estimates of the benefits of

improved programming practices which can be used to cost-justify investments designed to improve those practices.

One of the benefits of the methodology employed in this study is that once we have established that these metrics have a significant effect upon costs, we can then estimate the actual magnitude of these costs by interpreting the regression coefficients. Based upon the regression estimates in Table 4, the effects of the metrics for projects of average size (about 5400 source lines of code) are approximately

- 0.6 hours reduction for every statement added to average module size.
- 15 hours added for every statement deviation from an optimum average component size of 45.
- 140 hours added for every 1% absolute increase in the proportion of statements which are non-benign GOTO statements.

A perhaps more informative way to interpret these results is to compute the percent change in average project costs associated with metric values which deviate unfavorably from the research site's mean values by one standard deviation. The advantage of this approach is that we know we are comparing the more complex software to complexity standards observed in practice at this site, rather than to a perhaps-arbitrary ideal. The penalties associated with these less favored complexity scores are:

- 10% of total costs for module size.
- 30% of total costs for procedure size[13].
- 15% of total costs for branching density.

Armed with these quantified impacts of complexity, software maintenance managers can make informed decisions regarding preferred managerial practice. For example, one type of decision that could by aided by such information is the purchase of CASE reengineering tools. A great many claims are made for such tools; improved programming practice is only one of them. The benefits of these tools have also generally had to be taken on faith. Our 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 on the basis of these estimated benefits.

---

[13]There is an asymmetry here. The estimates are a 25% penalty for applications whose procedures are 1 SD smaller than average and a 35% penalty for those whose procedures are 1 SD larger than average. We cannot statistically reject the hypothesis that these two values are actually equal.

More generally, a common belief in the long-term importance of good programming practice has generally 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 correctable complexity at this research site amounts to several million dollars per year, the legacy of the practices of previous years.

Taken together these ideas show how, through the predictive use of the model developed here, managers can make decisions today on systems design, systems development, and tool selection and purchase that depend upon system values that will affect future maintenance. This can be a valuable addition to the traditional emphasis on current on-time, on-budget systems development in that 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 for them to be properly weighted in current decision-making.

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

VI. Concluding Remarks

In this study we have investigated the links between software complexity and software maintenance productivity. On the basis of an analysis of software maintenance projects in a commercial application environment we confirmed that software maintenance costs rise significantly as software complexity increases. In this study software maintenance costs were found to increase with increases in the complexity of a program's components, as measured by the programs' average procedure size, average module size, and control structure complexity.

Historically, most models of software labor productivity have not explicitly used software metrics. Our analysis suggests that high levels of software complexity account for approximately 30% of maintenance costs at this site, or about 20% of total life-cycle costs. Therefore, the neglect of software complexity is potentially a serious omission.

The results presented here are based upon a highly detailed analysis of programming costs at a site we judge to be very typical of the traditional transaction processing environments which account for such a considerable percentage of today's software maintenance costs. Based upon our analysis, the aggregate cost of poor programming practice for industry is substantial.

Bibliography

Albrecht, A. and J. Gaffney
"Software Function, Source lines of Code, and Development Effort Prediction: A Software Science Validation"
IEEE Transactions on Software Engineering, v. SE-9, n.6, pp.639 - 648, November, 1983.

An, K. H., D. A. Gustafson, and A. C. Melton
"A Model for Software Maintenance"
Conference on Software Maintenance, 1987, pp. 57-62.

Banker, R. and Kemerer, C.,
"Scale Economies in New Software Development",
IEEE Transactions on Software Engineering, v. SE-15 n. 10, October, 1989, pp. 416-429.

Banker R., Datar S., and Kemerer, C.,
"A Model to Evaluate Factors Impacting the Productivity of Software Maintenance Projects",
accepted for publication in *Management Science*, 1990.

Banker R., Datar S., and Zweig D,
"An Empirical Analysis of Software Complexity Metrics",
Carnegie Mellon University working paper, 1989.

Basili, V. R. and B. Perricone,
"Software Errors and Complexity: An Empirical Investigation"
Communications of the ACM, v. 27, n. 1, January 1984, pp. 42-52.

Basili, V.R. and Selby, R.W.,
"Calculation and Use of an Environment's Characteristic Software Metric Set",
Proc. of the Eighth International Conference on Software Engineering, 1985, pp. 386-391.

Belady, L.A. and Lehman, M.M.
"The Characteristics of Large Systems",
in *Research Direction in Software Technology*, MIT Press, 1979.

Belady, L. A. and M M. Lehman
A model of large program development
IBM Systems Journal, v. 15, n. 1, pp. 225-252 (1976).

Belsley, D.A., Kuh, E. and Welsch, R.E.,
Regression Diagnostics, John Wiley and Sons, NY, 1980.

Boehm, B.,
"Improving Software Productivity",
Computer, September 1987, pp. 43-57.

Boehm, B.,
Software Engineering Economics, Englewood Cliffs, NJ, Prentice-Hall, 1981.

Boehm, B.
"Software Engineering -- As It Is",
Proc. of the Fourth International Conference on Software Engineering, 1979, pp. 11-22.

Bowen, J.B.,
"Module Size: A Standard or Heuristic?"
Journal of Systems and Software, v. 4, 1984, pp. 327-332.

Boydston, R. E.
"Programming Cost Estimate: Is It Reasonable?"
Proc. of the 7th International Conference on Software Engineering, 1984, pp. 153-159.

Card, D.N., Page, G.T. and McGarry, F.E.,
"Criteria for Software Modularization",
Proc. of the Eighth International Conference on Software Engineering, 1985, pp. 372-7.

Chong Hok Yuen, C.K.S
A Statistical Rationale for Evolution Dynamics Concepts
Proceedings of the Conference on Software Maintenance 1987, pp. 156-164.

Conte, Dunsmore, and Shen
Software Engineering Metrics and Models, Benjamin-Cummings, 1986.

Curtis, B., Shepperd, S. and Milliman, P.,
"Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics",
Proc. of the Fourth International Conference on Software Engineering, 1979, pp. 356-60.

Curtis, B., et. al.,
"Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics",
IEEE Transactions on Software Engineering, v. SE-5, n. 2, March, 1979. pp. 96-104.

Freedman, David H.
Programming without Tears
High Technology, April 1986, v. 6, n. 4, pp. 38-45.

Gallant, J.
"Survey Finds Maintenance Problem Still Escalating".
Computerworld , v.20, Jan 27, 1986.

Gibson, V.R. and Senn, J.A.,
"System Structure and Software Maintenance Performance",
Communications of the ACM, v. 32, n.3, March, 1989, pp. 347-358.

Gilb, T.,
Software Metrics, Cambridge, Winthrop 1977.

Gordon, R.D.,
"Measuring Improvement in Program Clarity",
IEEE Transactions on Software Engineering, v. SE-5 n. 2, 3/79, pp. 79-90.

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

Halstead, M.,
Elements of Software Science, Elsevier North-Holland, 1977.

Hazzah, A.
"Everything is in the Names for Coming "Capture" Tools"
Software Magazine, October 1989, pp. 40-50.

Kafura, Dennis and Reddy, Geereddy R.,
"The Use of Software Complexity Metrics in Software Maintenance",
IEEE Transactions on Software Engineering, v. SE-13 n. 3, March 1987

Kearney, Joseph K. et. al.,
"Software Complexity Measurement",
Communications of the ACM, v. 29, No. 11, November 1986, pp. 1044-1050.

Kemerer, Chris F.,
Measurement of Software Development Productivity",
Doctoral dissertation, Carnegie Mellon University, 1987, UMI Order #8905252.

Lawrence, M. J.
An Examination of Evolution Dynamics
6th International Conference on Software Engineering, (1982), pp. 188-196.

Lawler, R.W.,
"System Perspective on Software Quality",
Proceedings, 5th International Computer Software and Applications Conference, 1981, pp. 66-74.

Li, H.F. and Cheung, W.K.,
"An Empirical Study of Software Metrics",
IEEE Transactions on Software Engineering, Vol SE-13 n. 6, 6/87, pp. 697-708.

Lind, R. and K. Vairavan
"An Experimental Investigation of Software Metrics and their Relationship to Software Development Effort"
IEEE Transactions on Software Engineering, v, 15, n. 5, May 1989, pp. 649-653.

Lyle, J.R. and Gallagher, K.B.
"Using Program Decomposition to Guide Modification",
Proceedings of the Conference on Software Maintenance, 1988, pp. 265-269.

McCabe, T.J.,
"Structured Testing: A Software Testing Methodology using the Cyclomatic Complexity Metric",
National Bureau of Standards special publication 500-99, December 1982.

McCabe, T.J.
"A Complexity Measure",
IEEE Transactions on Software Engineering, v. SE-2 n. 4, 12/76, pp. 308-320.

Munson, John C. and Khoshgoftaar, Taghi, M.,
"The Dimensionality of Program Complexity",
Proceedings of the International Conference on Software Engineering, 1989, pp. 245-253.

Schneidewind, Norman F.
"The State of Software Maintenance",
IEEE Transactions on Software Engineering, v. SE-13, n. 3, March 1987, pp. 303-310.

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, v. SE-11, n. 4, April 1985, pp. 317-323.

Shepperd, M.
"A critique of cyclomatic complexity as a software metric",
Software Engineering Journal, v. 3, n. 2, March, 1988, pp. 30-36.

Spratt, L. and McQuilken, B.,
"Applying Control-Flow Metrics to COBOL",
Proceedings of the Conference on Software Maintenance, 1987, pp. 38-44.

Vessey, I.,
"On Program Development Effort and Productivity",
Information and Management , v. 10 , 1986, pp. 255-266.

Vessey, I., and Weber, R.,
"Research on Structured Programming: An Empiricist's Evaluation",
IEEE Transactions on Software Engineering, SE-10 n. 4, July 1984, pp. 394-407.

Vessey, I. and Weber, R.,
"Some Factors Affecting Program Repair Maintenance: An Empirical Study",
Communications of the ACM, v. 26, No. 2, February 1983, pp. 128-134.

Wei, Duan and Bau, Jinn-Jomp,
"Some Optimal Designs for Grouped Data in Reliability Demonstration Tests",
IEEE Transactions on Reliability, v. R-36 n. 5, December 1987, pp. 600-603.

Zweig, D.
Software Complexity and Maintainability
unpublished Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, (1989).