1989

SOFrWARE COMPLEXITY AND MAINTAINABILITY

Rajiv D. Banker
University of Minnesota

Srikant M. Datar
Carnegie Mellon University

SOFTWARE COMPLEXITY AND MAINTAINABILITY

Rajiv D. Banker
Carlson School of Management
University of Minnesota

Srikant M. Datar
Dani Zweig
Graduate School of Industrial Administration
Carnegie Mellon University

ABSTRACT

This paper examines the relationships between software complexity and software maintainability in commercial software environments. Models are proposed for estimating the economic impacts of software complexity and for identifying the factors which affect a system's complexity. Empirical work currently under way has shown these models to be implementable.

1. INTRODUCTION

With over $200 billion dollars being spent every year on software (Boehm 1987), considerable attention has been devoted to controlling software costs. Much of this attention has been focused on tools and techniques designed to make software development as rapid and inexpensive as possible. To an increasing extent, however, the focus of this attention is shifting from the development phase of the software life cycle to the maintenance phase: For every dollar spent on development, two or three dollars are routinely spent on subsequent maintenance and enhancement (Gallant 1986) -- and software which is inexpensive to develop but difficult to maintain is no bargain at all.

The major benefits, or penalties, of new software development tools and techniques will be realized over the lifetime of the software, a lifetime that is often measured in decades. This means that it can take five, ten, or more years to assess their actual impacts upon life-cycle software costs. As it is rarely practical to delay a decision so long, the adoption of new CASE (computer assisted software engineering) tools, new programming methodologies and other products or methods must often be an act of faith.

This paper presents a framework for evaluating such technologies within a practical time frame. A two-stage analysis, using software complexity as an intermediate variable mediating between current decisions such as the adoption of new software tools and the downstream economic impacts of these decisions is presented. (Software complexity refers to the extent to which a system is difficult to comprehend, modify and test, not to the complexity of the task which the system is meant to perform; two systems equivalent in functionality can differ greatly in their software complexity.)

Software complexity is widely regarded as an important determinant of software maintenance costs (Boehm 1981). Increased software complexity means that maintenance and enhancement projects will take longer, will cost more, and will result in more errors. What is more, the software complexity of a given system is one of the main long-term legacies of whatever tools and techniques were employed in its initial development. If, for example, the use of new CASE tools leads to the development of poorly structured software, the effects of that poor structure will be felt when the time comes to modify the system.

The framework developed in this paper requires two sets of models: one to estimate the economic impacts of software complexity upon long-term costs and one to assess the current factors which affect and determine the degree of software complexity. An effort to implement this framework is currently underway at a commercial software organization. At the time of this writing, this effort has progressed far enough to verify the implementability of all the models presented in this paper.

2. CONCEPTUAL FRAMEWORK

Along with the growing awareness of the need to control life-cycle software costs, rather than just software development costs, there is a growing list of CASE tools and other methodologies which claim to help do just that (Olle, Sol and Verrijn-Stuart 1982). Managers who do not wish to take these claims on faith have the option of waiting a few years for new tools and products to acquire a performance history.

We propose a framework to enable researchers (and managers) to assess such products and techniques more quickly by introducing software complexity as a factor linking software development tools and techniques and software maintenance costs. This allows us to break the

problem down into two manageable and estimable ones: determining whether given techniques reduce software complexity and estimating the degree to which software of low complexity is less expensive to maintain. Each of these questions can be answered by analyzing data collected over a relatively short period of time.

The conceptual framework and the various issues pertaining to software complexity are depicted in Figure 1. The right hand side of the figure focuses upon the relationship between software complexity and software costs. High levels of software complexity make software more difficult, and hence more costly, to maintain. Software complexity also makes such maintenance more error-prone. The left hand side of the figure models the impact of various software engineering tools and techniques on reducing and controlling software complexity and, by extension, maintenance costs. In order to model either set of relationships correctly, it is necessary to control for confounding factors such as the nature of the tasks which the software was designed to carry out, the skill and experience of the programmers, and the nature of the organizational environment.

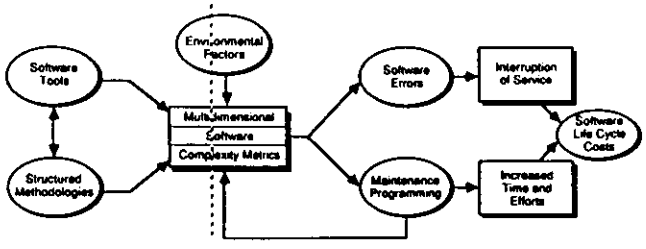

Figure 1. Framework for Evaluating Economic Impacts of Software Complexity

Our model raises many research questions that need to be addressed. We turn now to these issues.

A basic issue pertains to the measurement of software complexity. In searching for appropriate measures, we are not hampered by a shortage of candidates. On the contrary, we must select among over a hundred candidate metrics. There are several factors contributing to software complexity, including the size of a program and the complexity of its control structure, but the presence of over a hundred metrics does not imply the existence of over a hundred such dimensions of complexity. There is considerable duplication among the metrics (Munson and Khoshgoftaar 1989) and we must select a subset of metrics which measures the dimensions of greatest interest.

Having identified methods of measuring various dimensions of software complexity, we must also establish that they do in fact have a large enough effect on software costs to justify the expenditure of thousands or millions of dollars by managers wishing to control that complexity. Both direct and indirect effects must be considered. The direct effects are the added maintenance programming costs attributable to high levels of complexity. The indirect effects are the higher error rates associated with high levels of software complexity (Bowen 1978). If a maintenance project results in the propagation of new software errors, those errors will result in additional disruption and must be repaired at an additional programming cost.

Finally, if the study of software complexity is to be of practical value, we must understand the factors which cause some systems to be more complex than others so that we can take steps to control that complexity. The use of appropriate software development tools may affect a system's complexity, but we cannot assess the impact of such tools without controlling for environmental variables such as the volatility of the environment within which the system is operating, which also influences software complexity.

The models presented in the following sections are described within the context of a field site at which they are currently being implemented. This enables us to ground our discussion in a real-world environment and to be certain that we are not developing models which call for data which is impractical to collect.

We have begun to implement the models described in this paper at the information systems division of a major bank. This site was identified as being a highly typical commercial software environment. Its software consists of over twenty million lines of COBOL code, mostly transaction systems, running in an IBM mainframe environment. Much of this software dates back to the mid-1970s and earlier. Most of the programming expenditures at this site, over $25 million per year, go to maintaining these application systems.

At the time of this writing, this project has proceeded far enough to confirm the implementability of the proposed models: It has proven practical to collect all the data which these models require. Table 1 presents some descriptive statistics for the systems we have analyzed to date, including base error rates, system age, annual programming expenditures, and system size.

Table 1. A Profile of Application Systems being Analyzed

Variable	Mean	SD	Min	Max
Errors per year	348	561	14	2995
Age	11.3	5.6	3	21
Programming costs ($000/year)	693	661	83	3532
Appn size: Programs	217	266	18	1500
000 lines	215	185	54	702

The software at this site is divided into over a hundred **application systems**, each responsible for a different task. Examples of application systems are a payroll system, an inventory control system, and an accounts receivable system. Each system, in turn, is composed of a large number (typically dozens or hundreds) of **programs**. The accounts receivable system, for example, might have one program for aging the receivables and another program which reads the aged-receivables file and prints a report listing all receivables over three months old. Alternatively, depending upon the decisions of the programmers, there might be a single program combining those two functions.

The next four sections describe the specific models actually needed to implement this framework. Section three examines the concept of software complexity in greater detail, in order to propose appropriate measures for this variable. In section four, a model is developed for assessing the impact of software complexity upon software error rates. In section five, a model is developed for estimating the other major impact of software complexity: its effect upon long-term programming costs. These models allow us to assess the long-term economic impacts of software complexity. In section six, a model is developed to *explain* this complexity, and to assess the degree to which it can be *controlled*.

3. MEASUREMENT OF COMPLEXITY

If we wish to use software complexity as a central variable in our analysis, we must first determine an appropriate way to measure it. An examination of the literature on the measurement of software complexity provides us with an embarrassment of riches: We have over a hundred candidate metrics to choose from.

Software complexity is assumed to be a multidimensional construct (Wake and Henry 1988). The complexity of a program depends upon its magnitude, the complexity of its control structure, and the complexity of its data flows (Basili and Hutchens 1983). Other researchers add other factors to this list, such as the degree of modularity (Bowen 1978). Mason and Khoshgoftaar (1989) conclude that four or five such complexity factors suffice to describe the multi-dimensional complexity of a program.

On the other hand, correlations among the various kinds of metrics have generally proven to be extremely high (Li and Cheung 1987). For example, the most commonly used measure of program magnitude -- Halstead's Effort metric (Halstead 1977), a function of the number of operators and operands in the program -- and the most commonly used measure of control complexity -- McCabe's Cyclomatic Complexity (McCabe 1976), a function of the number of independent control paths through a program -- typically have correlations on the order of 90 percent. Our analysis will be greatly facilitated if we can select metrics which, while measuring the various dimensions of software complexity which we have identified, are not badly confounded with each other.

An examination of major complexity metrics reveals that most of them confound the complexity of a program with its size. One program will be that much more difficult to work with than a second if, for example, it is twice the size, has twice as many control paths leading through it, or contains twice as many logical decisions. Unfortunately, these various ways in which a program may increase in complexity tend to move in unison, making it difficult to identify the multiple dimensions of complexity.

This gives rise to practical problems. First, the effects of some of the dimensions of software complexity may be swamped by the effect of program size, making them more difficult to detect and estimate. Second, the use of several highly correlated metrics in a single analysis is econometrically undesirable. These problems may be mitigated by computing metrics which are normalized for size.

We can compute software complexity metrics at levels of analysis other than the aggregate program level. Table 2 shows examples of three major kinds of complexity metrics at four different levels of analysis. This classification can yield four independent metrics: the average size of the programs which constitute that system, the average size of the subprograms within each program, and two measures of control-structure complexity which have been normalized for size (the density of decision points and the density of branch points). (A fifth possible metric suggested by this table, the total size of the system, will be dealt with separately.)

Table 2. Complexity Metrics at Different Levels

	Statement	Subprogram	Program	System
Size		Lines	Effort Statements	Lines Modules
Decision Structure	Density of Decisions	Cyclomatic Complexity	Number of Binary Decisions	
Control Structure	Density of Branching	Essential Complexity	Number of Branches	

In commercial environments, program size is largely a programming decision. A commercial application system will typically consist of anywhere from hundreds of thousands to millions of lines of code, divided into a large number of programs, and a maintenance or enhancement project will typically affect many of these programs at the same time. The difficulty of such a project will depend to an extent on the size of these programs. If a system is divided into too many small programs, even a minor modification may force maintenance programmers to work

with a large number of programs, adding a considerable overhead cost for each program that must be comprehended and modified. If it is divided into too few programs, each program will be large and difficult to comprehend.

It is clear that **program size** is an important complexity factor to measure. We will also want to measure other complexity factors, but if we measure them at the program level, as is most commonly done in the literature, we will find ourselves with measures which are highly correlated with program size. We must look to other levels of analysis to overcome this.

A single program will be easier to work with if it, in turn, is broken down into reasonably sized subprograms. Many of the same metrics which may be used to compute program size may be used to compute subprogram size, the simplest of these being direct measures such as the average number of statements per subprogram.

Metrics computed at the subprogram level will be orthogonal to program level metrics. Rather than measuring program size, they are measuring **program modularity**, the degree to which a program is subdivided into components of manageable length. We may compute other metrics at the subprogram level too, but just as the metrics computed at the program level are correlated with program size, other metrics computed at the subprogram level will be correlated with subprogram size.

Many of the metrics proposed by researchers are based on the complexity of a program's decision structure. The more alternative paths there are through a program (as a result of there being many decision points within a program), the more contingencies a programmer must consider and, therefore, the more difficult the programming task will be. In addition, the larger number of contingencies to be considered often translates into more contingencies than it is practical to test, so errors are more likely to go undetected.

Proposed measures of decision complexity tend to be based upon a graph theoretic analysis of a program's control structure (McCabe 1976). Such measures are meaningful at the program or subprogram level, but metrics computed at those levels will be badly confounded with program or subprogram size. However, the values of these metrics depend primarily upon the number of decision points within a program. This suggests that we can compute a size-independent measure of decision complexity by measuring the **density of decision making** within a program.

Another kind of control structure complexity which may be measured is the structuredness of a program, as measured by the degree to which is may be logically decomposed (McCabe 1982). A highly decomposible program is easier to analyze and maintain, and it is easier to assure oneself that a modification to such a program will not introduce new errors (Lyle and Gallagher 1988). A more poorly structured program will impose a heavy cognitive load upon a programmer, which may in turn incur a heavy penalty in added programming time. The decomposibility of a program depends primarily upon the degree of branching within its control structure. Thus we may compute a second size-independent complexity metric by measuring the **density of branching** within a program.

In summary, we propose the use of four metrics: **program size**, measured at the *program* level by the average number of statements per program; **modularity**, measured at the *subprogram* level by the average number of statements per subprogram; **decision structure complexity**, measured at the *statement* level by the proportion of IF statements within a program; and **program decomposibility**, measured at the *statement* level by the proportion of GOTO statements within a program. Table 3, presents statistics describing the software complexity of the application systems analyzed at the research site. An examination of these variables confirms that, as intended, they are reasonably unconfounded, their correlations varying from a ten percent correlation between program size and subprogram size to a 38 percent correlation between branching density and decision density.

Table 3. Summary Statistics for Four Software Complexity Metrics

Complexity Metrics	Mean	SD	Min	Max
Program Size (Stmts)	564	190	302	1119
Subprogram Size (Stmts)	50	31	13	136
Branch Density (per stmt)	.078	.03	.016	.150
Decision Density (per stmt)	.150	.03	.076	.196

Our metrics measure four important dimensions of complexity, computing four distinct measures at three different levels of analysis. A fifth metric, the total size of the application system, will be useful in some contexts, but not in others: System size depends primarily upon the functionality of the system and is a policy decision, rather than a programming decision, so there is some doubt as to whether it should be considered a measure of software complexity. In the next two sections, we will present models for analyzing the downstream impacts of these metrics.

4. COMPLEXITY IMPACTS ON ERROR RATES

The more difficult a system is to comprehend, the more likely it is that programming errors will go unnoticed. The more difficult it is to test, the less likely it is that those errors will be caught before the software goes into operation. Such errors represent both current disruption costs, as they manifest, and future programming costs, as they are corrected.

That a relationship exists between software errors and software complexity is well documented. In order to actually estimate that relationship, we must model the error process in a way which allows us to control for the confounding effects of other factors which may affect error rates.

The site at which this analysis is being implemented has an error tracking system, which we are using to monitor the number of errors for each application system in each month over a period of time (a two year period, in this case).

The number of errors for a given application system will vary from month to month as ongoing maintenance corrects old errors and introduces new ones and as different data inputs flush out previously unsuspected errors. We model the error rates as a stochastic process, whose mean varies from application system to application system, as a multiplicative function of each system's software complexity, and several other factors. In particular, we model the error rate as a Poisson process which may be closely approximated by an Exponential error distribution with mean Lambda, where

Mean Error Rate (Lambda) = f(Software Volatility, Developer, Application Experience, Primary User, Software complexity)

- **Software Volatility**: This is a measure of the frequency with which changes are made to the application software. Systems which undergo frequent modification have higher error rates, because each modification represents an opportunity for new errors to be generated. It may also be the case that when systems are undergoing frequent changes, there is less opportunity and less interest in testing those changes thoroughly.

- **Developer**: Systems which were purchased, or developed by contract programmers, will be less familiar to their maintainers than systems which were developed internally. These systems are less likely to follow house standards or conventions. The lessened familiarity can make it easier for errors to go undetected.

- **Application Experience**: If the programmers maintaining a system haven't been maintaining it long, their relative inexperience will increase the chances of unnoticed errors slipping by them. They will also be less likely to know the problem areas which require the most careful testing.

- **Primary User**: Not surprisingly, greater efforts will be made to avoid and to detect errors if the consequences of an error are severe. In particular, errors that will be tolerated on systems whose main users are other departments within an organization may not be tolerated on systems which directly affect customers or clients. The latter systems will see more careful programming and more thorough testing.

- **Software Complexity**: After controlling for the factors discussed above, we are in a position to examine the marginal impact of software complexity upon error rates. We might particularly expect to see higher error rates in systems which display high decision densities (with their correspondingly large number of decision paths to test for errors). New errors are most likely to propagate in such systems (Gibson and Senn 1989). More prosaically, we would expect larger application systems to have higher error rates, since there are more things to go wrong.

These explanatory variables (except for the previously computed software complexity metrics) were gathered by means of questionnaires administered to managers responsible for the maintenance of the application systems being analyzed. The proposed model was used to analyze the error rates of 34 application systems. Preliminary analysis confirms that all of these variables affected error rates, in the expected directions, at the five percent significance levels.

5. COMPLEXITY AND MAINTENANCE COSTS

Software complexity may have a direct impact upon maintenance costs as well as the costs incurred through the presence of software errors. The more difficult a highly complex application system is to maintain, the more billable hours it will accumulate in the course of the maintenance and enhancements which consume most of a system's life cycle costs.

Our major focus in this section is on evaluating the impact of software complexity on labor maintenance costs. In order to do so, however, we must control for the effects of other factors, such as task magnitude and the skill of the programmers, that also affect the programmer hours required on a project. For example, a large maintenance project dealing with an application system of low complexity may easily require more hours than another project meant to make a small modification to a system of higher complexity. Excluding task size and complexity (and other environmental variables) will result in a misspecification of the model and incorrect inferences about the impact of software complexity on costs. In the example just given, not controlling for task size will lead to the conclusion that higher software complexity will result in lower costs.

Figure 2 presents a model of the maintenance function which is based upon the Cocomo model of software cost estimation (Kemerer 1987). Software maintenance is viewed as a production process whose inputs are labor and computing resources and whose output is modified code. Since labor hours are considerably greater than computer resources and there are limited substitution possibilities

between the two, we focus on labor hours as the major expense incurred in software maintenance. The productivity of this process depends upon a number of environmental variables. Among the most important of these (Boehm 1987) are the skill and experience of the programmers, the skill and involvement of the users, and the software tools available to the programmers. Following Banker, Datar and Kemerer (1988), we write labor hours required as some function of the task requirement and environmental factors.

Labor Hours = f(Task magnitude and complexity, Programmer skill, Programmer experience, User skill and involvement, Software tools, Software complexity)

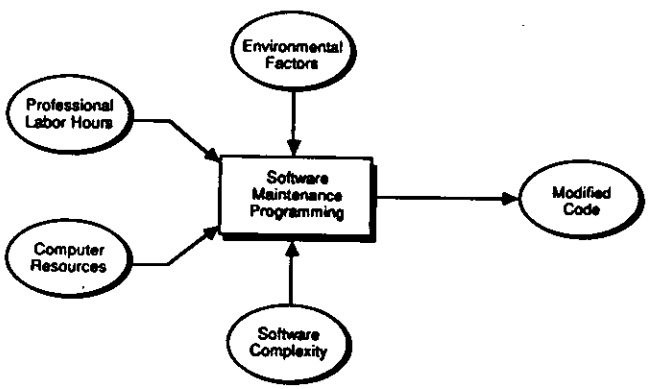

Figure 2. Model of Software Maintenance Programming Code

The unit of analysis for this model is the project. Each maintenance project has its own task requirements and its own budget. For each such project, seven types of data must be collected.

- **Task Magnitude**: The output of the software maintenance process is modified code. This will naturally have a major effect on the amount of work required.
- **Task complexity**: Other things being equal, some programming 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 programming resources.
- **Programmer skill**: Studies have found ten-to-one differences in productivity between top rated programmers and poorer ones.
- **Programmer experience**: Even a good programmer is at a disadvantage when faced with an unfamiliar system as time must be expended in comprehending the software and becoming familiar with it.
- **User skill and involvement**: Requests for modifications to a system typically emanate from the system's users. An unsophisticated user can add considerably to the cost of a project by generating confused or incoherent requirements, by frequently changing the requirements, and by failing to adequately communicate relevant information.
- **Software tools**: Many products are commercially available which have been designed to increase programmer 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.

The collection of the necessary data is practical, but somewhat labor intensive. Task magnitude and complexity can be obtained by counting the number of source lines of code added or modified in the course of the project and by computing the number of function points added or modified. (Function points [Albrecht and Gaffney 1983] are a language-independent measure of software functionality.)

Programmer skill and experience are obtained from corporate personnel files (which include formal managerial assessments of each programmer). The other data may be obtained by interviewing the managers of the projects being analyzed.

Higher maintenance costs attributable to high levels of software complexity constitute an ongoing tax which a software system may levy upon its maintainers. Once highly complex software has been identified, and its long-term cost impact has been estimated, managers can choose from a number of options: If the cost impacts are relatively minor, they may opt to leave the system as it is, but to monitor future maintenance to the system to be sure that it results in modifications which are of lower complexity than the original, or at least no higher.

Alternatively, it may be practical to identify key components of the system which are particularly complex and modify just those components. (Much of the complexity of a system tends to reside within a small proportion of its code.) Finally, a system may be so complex that it would be cheaper to rewrite or replace it than it would be to tinker with it.

Most of the cost of maintaining a program is expended in comprehending it rather than in making the actual modifications (Lientz, Swanson and Tompkins 1978). We may expect all four of the complexity factors which we have chosen to affect maintenance costs by affecting the effort

level required by a programmer to understand the code that must be modified.

The proposed model was used to estimate the costs of 59 maintenance projects of average size (the mean project size being 910 hours, billed at forty dollars per hour). The results were strongly consistent with the earlier use of the model in both the significance levels of the explanatory variables and the direction of their effects. Controlling for the effects of task size, task complexity and environmental factors, program size and modularity were found to affect maintenance costs at the five percent level of significance.

6. FACTORS AFFECTING COMPLEXITY

In order to control software complexity, we must ascertain its causes. Three sources suggest themselves as major contenders. First, a system may be complex because it does complicated things. There is a common belief that the magnitude of a task should not affect software complexity, that any task can be programmed well or poorly (Cavano and McCall 1978), but this is a counsel of perfection.

A system may be complex because of poor initial design. This may be attributed to poor programming practices, either because the programmers developing a system are insensitive to considerations of maintainability or because they are following organizational standards which are insensitive to those considerations. This is where modern development methodologies promise the greatest impact. Since understanding of structured programming has become more general in the past decade, and software development tools more widely available, there may be a tendency for older systems to be more complex than newer ones. Much of the currently installed software base is over a decade old.

Finally, software may be "maintained to death" (Belady and Lehman 1979). Enhancements to a system often consist of poorly planned modifications to the system. Even relatively careful maintenance may result in the creation of new modules which the application system was never designed to contain. Thus, the more maintenance a system undergoes, the more complex it is likely to become. (There is a vicious circle in place here, since a complex system will probably accumulate programming errors which will necessitate more frequent maintenance.)

We expect older systems to exhibit higher levels of software complexity. The older the system is, the longer it has had to be modified and patched and otherwise to have its structure and integrity eroded. In addition, the older systems were written at a time when there was a lower awareness of the importance of software maintainability, so they were probably more poorly designed to start with.

Software Complexity = f(software tools, volatility, functionality, age, operating requirements, error rates)

- **Software Tools**: The complexity of a system is initially determined when the system is developed. Appropriate software tools and development methodologies can aid programmers to write maintainable software. Whether they actually do so is an empirical question that our analysis can answer. In order to correctly specify the model, a number of other factors must be considered.

- **Volatility**: Some systems will undergo more frequent maintenance because they are subject to frequent regulatory changes, used in a highly dynamic and competitive industry, or they must interface with a rapidly evolving system. We expect that more volatile systems will experience more frequent and more hurried maintenance, both of which result in a degradation of software complexity over time.

- **Functionality**: One may think of the size of the entire application system as a complexity metric, but it is one which is not under the control of the programmers, being largely determined by user requirements. Since more ambitious systems may require more ambitious code, the size of an application system may make it difficult for programmers to maintain good levels of software complexity.

- **Age**: Age is not a factor which is directly controllable by management, except in the extreme sense that management may choose to replace an old system. It is, however, a surrogate for a number of factors contributing to software complexity, such as the level of programming sophistication which existed at the time the software was first developed.

- **Operating Requirements**: Another source of system complexity is the imposition of operating requirements such as rapid response time or cost efficiency. While it is not necessarily the case that technical constraints force programmers to write complex code, they do tend to cause programmers to pay less attention to such niceties.

- **Error Rates**: Just as we expect software complexity to affect error rates, error rates may, in their turn feed back upon the software complexity of an application system. Thus, we will probably not wish to estimate this model using Ordinary Least Squares. Rather, it may be more appropriate to estimate it as a system of simultaneous equations, with metrics and error rates both acting as endogenous variables.

Preliminary analysis suggests that the importance of the various factors determining software complexity varies from metric to metric. Age, for example, has the strongest significance in explaining the level of branching within a

system -- not surprising, given that the older systems were written before such branching came to be considered harmful.

7. CONCLUSIONS

Most software expenditures are devoted to maintaining existing software and anything we can do to make that software easier to maintain can mean enormous monetary savings. This is the attraction which commercially available software engineering tools and methodologies have for programming organizations. In this paper, we have presented a framework for assessing such tools.

This framework allows us to perform our assessment by analyzing the ability of the products being examined to affect software complexity and by contemporaneously estimating the long-term economic impact of high (or low) levels of software complexity.

We have examined the questions which must be answered before managers commit large expenditures of money and resources in this direction. We have presented a basis for selecting appropriate metrics to use in monitoring software complexity. We have presented a set of models for estimating the economic impacts of software complexity. Initial analysis shows these models to be practical to implement and tends to confirm our expectation that the impacts which we are investigating are present and are significant.

Finally, we have presented a model for testing the effect of the tools or techniques in question upon software complexity.

We are in the process of implementing these models at a commercial site. Data collection undertaken to date confirms the practical implementability of our framework. Analysis undertaken to date has yielded results supportive of the models presented and of our expectations: Higher levels of software complexity tend to result in higher error rates. Higher levels of software complexity tend to result in higher maintenance costs. Software complexity may be explained by a number of factors, including the age, functionality and volatility of the software. Controlling for these factors, we may assess the marginal impact of CASE (Computer Aided Software Engineering) tools and other products upon this complexity. These results will be presented in subsequent papers at the completion of the study.

8. REFERENCES

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

Banker R.; Datar S.; and Kemerer, C. "A Model to Evaluate Factors Impacting the Productivity of Software Maintenance Projects." MIT Sloan School, Working Paper Number 2093-88, 1988.

Basili, V. R., and Hutchens, D. H. "An Empirical Study of a Syntactic Complexity Family." *IEEE Transactions on Software Engineering*, Volume SE-9, Number 6, November 1983, pp. 664-672.

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

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

Boehm, B. *Software Engineering Economics*, Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

Bowen, J. B. "Are Current Approaches Sufficient for Measuring Software Quality?" *Proceedings of the ACM Software Quality Assurance Workshop*, November 1978, pp. 148-155.

Cavano, J. P., and McCall, J. A. "A Framework for the Measurement of Software Quality." *The Proceedings of the ACM Software Quality Assurance Workshop*, November 1978, pp. 133-139.

Gallant, J. "Survey Finds Maintenance Problem Still Escalating." *Computerworld*, Number 20, January 27, 1986.

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

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

Kemerer, C. F. "Measurement of Software Development Productivity." Doctoral Dissertation, Carnegie-Mellon University, 1987.

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

Lientz, B. P.; Swanson, E. B.; and Tompkins, G. E. "Characteristics of Application Software Maintenance." *Communications of the ACM*, Volume 21, Number 6, June 1978, pp. 466-471.

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. "A Complexity Measure." *IEEE Transactions on Software Engineering*, Volume SE-2, Number 4, December 1976, pp. 308-320.

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

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

Olle, T. W.; Sol, H. G.; and Verrijn-Stuart, A. A., Editors. *Information System Design Methodologies: A Comparative Review*, Amsterdam, North-Holland, 1983.

Wake, S., and Henry, S. "A Model Based on Software Quality Factors which Predict Maintainability." *Proceedings of the Conference on Software Maintenance*, 1988, pp. 382-387.