

# The Moderating Effects of Structure on Volatility and Complexity in Software Enhancement

Rajiv D. Banker • Sandra A. Slaughter

*School of Management, The University of Texas at Dallas, Richardson, Texas 75083-0688*

*Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213*

*rbanker@utdallas.edu • sandras@andrew.cmu.edu*

---

The cost of enhancing software applications to accommodate new and evolving user requirements is significant. Many enhancement cost-reduction initiatives have focused on increasing software structure in applications. However, while software structure can decrease enhancement effort by localizing data processing, increased effort is also required to comprehend structure. Thus, it is not clear whether high levels of software structure are economically efficient in all situations. In this study, we develop a model of the relationship between software structure and software enhancement costs and errors. We introduce the notion of software structure as a moderator of the relationship between software volatility, total data complexity, and software enhancement outcomes. We posit that it is efficient to more highly structure the more volatile applications, because increased familiarity with the application structure through frequent enhancement enables localization of maintenance effort. For more complex applications, software structure is more beneficial than for less complex applications because it facilitates the comprehension process where it is most needed. Given the downstream enhancement benefits of structure for more volatile and complex applications, we expect that the optimal level of structure is higher for these applications. We empirically evaluate our model using data collected on the business applications of a major mass merchandiser and a large commercial bank. We find that structure moderates the relationship between complexity, volatility, and enhancement outcomes, such that higher levels of structure are more advantageous for the more complex and more volatile applications in terms of reduced enhancement costs and errors. We also find that more structure is designed in for volatile applications and for applications with higher levels of complexity. Finally, we identify application type as a significant factor in predicting which applications are more volatile and more complex at our research sites. That is, applications with induction-based algorithms such as those that support planning, forecasting, and management decision-making activities are more complex and more volatile than applications with rule-based algorithms that support operational and transaction-processing activities. Our results indicate that high investment in software quality practices such as structured design is not economically efficient in all situations. Our findings also suggest the importance of organizational mechanisms in promoting efficient design choices that lead to reduced enhancement costs and errors.

*(Software Enhancement; Software Structure; Software Volatility; Software Complexity; Structured Design; Management of Information Systems)*

---

## Introduction

Most of the costs associated with the design, development, implementation, and operation of computer-based applications occur in software maintenance. Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, and to enhance the product by adapting it to a modified environment (*IEEE Standard for Software Maintenance* 1993). The Year 2000 problem, in particular, has highlighted both the dynamic nature of software and the significant cost incurred to adapt, enhance, and upgrade it (Jones 1997a). As the costs of software maintenance soar, improved techniques for controlling and reducing maintenance effort become imperative.

One approach to improving the maintainability of software is to control its complexity at the design phase. Often this is done by increasing the organization or structure of the software application.<sup>1</sup> A key objective of techniques such as structured design and object-oriented design, for example, is to facilitate change by organizing and localizing information processing in an application (Pfleeger 1998). However, structure does not always appear to yield its promised benefits in software maintenance (Kemerer 1995). Too much structure can introduce additional complexity that complicates the comprehension process in software maintenance tasks (Banker et al. 1991). This raises the question of under what conditions structure facilitates software maintenance and reduces maintenance costs and errors.

Although software maintenance (software enhancement in particular) represents an important investment for organizations, it is relatively underresearched (Kemerer 1995). There has been even less research link-

ing software design practices to enhancement outcomes, despite the belief that maintainability must be designed into applications (Banker et al. 1998). In particular, structured design is a widely adopted technique in software development, and one that is thought to facilitate enhancement, but there is little understanding of its downstream impact in maintenance. The research that has been done has yielded equivocal results, with some researchers observing that too much structure may be as undesirable as too little structure (Kemerer 1995, Conte et al. 1986). Yet, the conditions under which software structure is beneficial in software enhancement are not known.

This study represents an effort to initiate research of the link between software design decisions and software enhancement outcomes by examining precisely when structure is advantageous in terms of reduced enhancement costs and errors. We introduce the notion of software structure as a *moderator* of the relationship between software volatility (the frequency of enhancement per unit of functionality), total data complexity (the number of data elements per unit of functionality), and software enhancement outcomes. For more volatile applications, the increased familiarity of the maintainers with the application structure due to frequent change enables localization of maintenance effort and facilitates enhancement. For more complex applications, software structure is more beneficial than for less complex applications because it facilitates the comprehension process in software enhancement where it is most needed. Given the downstream enhancement benefits of structure for more volatile and more complex applications, we posit that the optimal levels of structure are higher for these applications.

We empirically evaluate our software enhancement model using data collected on the software applications of a major mass merchandiser and a large commercial bank. Our results confirm that the same level of structure is not economically efficient in all situations: Higher levels of structure are more advantageous for the more volatile and more complex applications in terms of reduced enhancement costs and errors. We find analytically and empirically that the optimal level of structure increases with volatility and complexity. Finally, we identify application type as an

<sup>1</sup>In this study, an *application* is a collection of related software modules that together accomplish major functions. Examples of business applications include accounts receivable, payroll disbursement, and order management. A *module* refers to a group of executable instructions with a single point of entry and a single point of exit that is part of an application and that accomplishes a particular function within that application. A module could be a main program, a subroutine, or a paragraph in a COBOL program. Examples of modules in business applications include accounts receivable data entry, accounts receivable report, date validation routine, and edit validation routine.

indicator that can be used to predict in the design stage which applications are likely to be more volatile and more complex after they are implemented. Our results reveal that applications with induction-based algorithms such as those that support planning, forecasting, and management decision-making activities are more complex and more volatile than applications with rule-based algorithms that support operational and transaction-processing activities at our research sites.

Our paper is organized as follows. We begin by developing a conceptual model that links software volatility, complexity, structure, and enhancement performance. We then extend our conceptual model by deriving how the optimal level of software structure varies with complexity and volatility. We present the empirical evaluation of our software enhancement model, discuss the organizational processes and incentives that motivate the design choices for the applications, and conclude with suggestions for further research of performance in software tasks.

## **Complexity, Volatility, Structure, and Software Enhancement Outcomes**

*Software enhancement* includes adding, changing, and deleting software functionality to adapt to new and evolving business requirements (Kemerer and Slaughter 1997). The proportion of effort devoted to enhancement has frequently been assessed as the largest in the software life cycle (Dekleva and Zvegintzov 1991, Nosek and Palvia 1990). As such, it is clear that initiatives targeted at reducing enhancement costs would provide more cost reduction per unit of effort than those targeted at reducing other software costs.

One basic factor that drives enhancement costs and errors is *software volatility* (Butcher 1997, Hager 1991, Bækgaard 1990). In the context of our study, software volatility refers to the frequency or number of enhancements per unit of application functionality over a specified time frame. Software volatility is frequently an implicit or latent factor in studies of software enhancement and is not often the focus of attention. The earliest discussion of software volatility (although they did not call it this) can be found in Belady and Lehman's laws

of software evolution dynamics (1976). Their study determined that software changing at an increasing rate over time (in terms of fraction of modules modified per release) deteriorated monotonically; this formed the basis for their Law of Increasing Entropy. Other researchers concerned with software volatility have focused on methods to design software so that it is more flexible and can be more easily changed (e.g., Hager 1991, Bækgaard 1990, Martin and McClure 1983, Parnas 1979). However, few researchers have explicitly examined the link between software volatility and enhancement outcomes.

Although the effect of software volatility on enhancement performance is not well understood, enhancement costs and errors are likely to increase with software volatility. Applications that are frequently enhanced will have higher enhancement costs than applications that are never or rarely enhanced, *ceteris paribus*. Higher volatility also increases the likelihood of introducing errors in software enhancement. This is because every time an application is modified, there is an opportunity to introduce an entirely new error or series of errors (Littlewood and Strigini 1992). Thus, we expect that:

*HYPOTHESIS 1. Higher software volatility is associated with higher software enhancement costs and errors.*

Another important factor associated with enhancement performance is software complexity. Software complexity has been strongly linked with enhancement effort (Banker et al. 1998, Banker et al. 1993) and errors (Khoshgoftaar and Munson 1990, Shen et al. 1985, Gremillion 1984, Basili and Perricone 1984). Thus, the focus of many software enhancement improvement initiatives has been on mitigating software complexity early in the life cycle, in the design phase.

Software complexity ultimately derives from the amount of data that must be processed by an application (Card and Glass 1990). The *total data complexity* of an application can be defined in terms of the number of data elements per unit of application functionality. This is congruent with Halstead's concept (1977) of the representation of software as a set of input and output data (operands) that must be processed (operators). Total data complexity is a key indicator of an application's inherent design complexity because it reflects

the complexity of the coded software (Al-Janabi and Aspinwall 1993, Card and Glass 1990). For example, Warnier asserts that the decision or procedural complexity of an application depends on the quantity of the data in the application, where each data element translates into about one decision (Warnier 1976). McCabe has also related data elements to his measure of decision or cyclomatic complexity (McCabe 1996). Empirically, data elements have been shown to have a strong positive correlation with application size in terms of lines of code (Card and Glass 1990, Basili et al. 1983), and size is a major software complexity factor (Mata-Toledo and Gustafson 1992, Li and Cheung 1987).

High amounts of software complexity interfere with the process of comprehending the application in enhancement and make it difficult for maintainers to efficiently and correctly modify the software (Bergantz and Hassell 1991, Gibson and Senn 1989). This implies that the higher the amount of complexity in design, the greater the downstream penalties in terms of software enhancement effort and errors. Therefore, consistent with the prior research of software complexity, comprehension, and maintenance, we expect a positive relationship between total data complexity and enhancement costs and errors. This leads to our second hypothesis:

*HYPOTHESIS 2. Higher levels of total data complexity are associated with higher software enhancement costs and errors.*

Software engineers can implement data elements and processing in one large function, or they can decompose it into smaller functions using the principles of decomposition in structured design (Pfleeger 1998, Page-Jones 1980, Yourdon and Constantine 1979, Parnas 1972). The main principle of structured design is that an application should be designed from the top down in hierarchical fashion and refined to greater levels of detail. The designer first considers the primary function of the application, then breaks this function into subfunctions and decomposes each subfunction until the lowest level of detail has been reached. The lowest level functions describe the detailed data processing that will occur.

Decomposition reduces the amount of data processing for a particular function but increases the number of connections to pass data and control parameters between functions in an application. Structure emerges from the interdependencies or calls between the functions in an application. The lowest level of structure in an application consists of a function with no calls. Higher levels of structure consist of multiple calls between multiple functions. The literature on software design (Pfleeger 1998, Whitten and Bentley 1997, Page-Jones 1980) promotes decomposition, as smaller functions are believed to be easier to code, understand, and maintain.

Empirical studies of the relationship between software structure and software maintenance performance have suggested that structure is beneficial in maintenance. For example, an experiment by Rombach (1987) using graduate student programmers found that use of more structured code led to reduced maintenance effort. Similar results were found in a series of experiments by Gibson and Senn (1989) using 36 experienced programmers from industry. In these experiments, an ill-structured software application was improved in two stages of restructuring, and the improved structure was linked to reductions in the programmers' maintenance effort and errors. A field study of 311 COBOL applications (Benander et al. 1990) found that less structured applications took significantly longer to debug and were more likely to have errors than more structured applications. A more recent study by Kada et al. (1993) revealed that maintenance effort was higher for an informally developed application than for an application developed using structured design. Consistent with the literature on software design and the empirical results from studies of software structure and maintenance performance, we therefore expect that:

*HYPOTHESIS 3. Higher levels of structure are associated with reduced software enhancement costs and errors.*

The benefits of structure in software enhancement are magnified for the more volatile and more complex applications. For the more volatile applications, structure mitigates the costs and errors associated with each change. This is because every modification affords

maintainers an opportunity to learn about the application and comprehend its structure. As maintainers become more familiar with the application structure, they can focus their efforts on only the particular parts of the application that need to be modified. This localization of maintenance effort is a primary advantage of functional decomposition in structured design and leads to improved maintenance performance (Snyder 1994).

For more complex applications, software structure provides increased benefits in terms of facilitating the comprehension process. Without structure, it may be very difficult for maintainers to comprehend applications with high levels of total data complexity, particularly as these applications are likely to also have high levels of procedural complexity (given the relationship between data elements and decisions). Structure decomposes the complexity so that maintainers can focus on comprehending smaller, more manageable functions, and can more effectively enhance the application.

Thus, we posit that structure *moderates* the relationship between software volatility, total data complexity, and software enhancement outcomes, such that:

*HYPOTHESIS 4a. For the more structured applications, higher levels of volatility are associated with lower software enhancement costs and errors.*

*HYPOTHESIS 4b. For the more structured applications, higher levels of total data complexity are associated with lower software enhancement costs and errors.*

## **Optimal Levels of Software Structure**

Although structure is beneficial in software maintenance, there is some indication that too much structure may not be optimal (Kemerer 1995, Banker et al. 1993, Banker et al. 1991, Conte et al. 1986, Bowen 1984). While a fundamental principle of structured design is that an application should be decomposed into smaller and more manageable functions (Page-Jones 1980), as decomposition progresses, it becomes increasingly difficult to minimize the interdependency (coupling) between functions and to sustain the cohesion (task specificity) of a particular function. Thus, a very highly

structured application may be difficult to comprehend due to the complex interfaces between functions. Increased effort is required to understand highly structured applications because maintainers need to trace long chains of interdependencies in the interfaces when modifying the software (Cant et al. 1995). Interface complexity is also strongly associated with high error rates (Takahashi and Nakamura 1997).

Furthermore, in software development, structured design is sometimes considered to be a slow, inflexible, and time-consuming process (Laudon and Laudon 1997, p. 356). It can take effort to structure an application even with the aid of sophisticated, automated tools (Guinan et al. 1997). In practice, most software development projects have finite resources as well as aggressive timelines. In this context, the effort required to implement structure in development will compete with the effort required to implement the required functionality in a timely and cost-effective manner. This implies that it is important to focus efforts on highly structuring the applications where the benefit is greatest. The question then arises as to which applications will benefit the most from implementing higher levels of structure over the entire software life cycle.

To address this question, we begin by observing that some applications are more complex, or will be enhanced more often, than others during their lifetimes. The unequal distribution of complexity and volatility within and between applications has been observed in several empirical studies of software maintenance (Kemerer and Slaughter 1997, Butcher 1997, Swanson and Beath 1989). For example, the existence of a Pareto effect in maintenance was found in an empirical study by Kemerer and Slaughter (1997) where 20% of the software applications received 80% of the modifications over a period of 20 years.

Because software volatility and complexity are not uniformly distributed across applications, and given the downstream benefits of structure for the highly volatile and complex applications, it is more efficient at the margin to build in higher levels of structure in development for these applications. This is evident from our examination of the optimization problem to determine the levels of structure that minimize software life-cycle costs, for given levels of size, volatility,

and total data complexity (see Appendix A). In this problem, the level of structure is the key decision variable. The optimal level of structure is chosen to equate the marginal cost of building in structure in the development phase to the marginal benefit of structure in reducing downstream enhancement costs. In development, marginal costs with respect to structure increase with the level of structure implemented. This reflects the increasing difficulty for designers to control the complexity of the structure (i.e., minimize coupling and maximize cohesion of the functions) as the application is decomposed to finer levels of detail (Bitman 1997, Card and Glass 1990, Page-Jones 1980). In enhancement, while increased structure is beneficial at first, the marginal benefit from greater structure ultimately decreases as more structure is put in place (Kemerer 1995).

As we have discussed, one principal benefit of structure in enhancement is that it mitigates the marginal cost impact of volatility. For the more volatile applications, an increased understanding of the structure is acquired through frequent modification. This improved understanding facilitates the comprehension of the application by helping maintainers to focus only on the part of the application that must be changed, thereby reducing the marginal cost of enhancement. On the other hand, for less volatile applications, maintainers are unfamiliar with the structure as they have interacted less with the application. Thus, they must invest effort in understanding the application structure as well as in making the modification. This implies that high levels of structure are not optimal for infrequently enhanced applications. To illustrate, Chapin and Lau (1996) examined in some detail a highly structured COBOL application. The application was not particularly complex in terms of its computations and functions and was rarely enhanced, but had an unusually high cost to maintain per incident.<sup>2</sup> The infrequency of

<sup>2</sup>The 600 modules in the application were smaller in size than the average for other applications, and maintainers complained that the modules specified very small functions to be performed, giving a "fragmented" impression to the application (Chapin and Lau 1996, p. 103). The actual maximum depth of the application's structure was found to be 41 levels. This led to an effective size more than 13,000 times larger than the application's stated size of 600 modules. In other words, maintainers had to understand the application as if it had more than 8,000,000 modules, rather than 600!

enhancement for this application provided few opportunities for maintainers to learn about its structure, and the elaborate structure contributed to the difficulty and expense of maintaining the application. This implies that the optimal level of structure increases with greater volatility of an application.

Another benefit of structure in enhancement that we have identified is the reduction of the marginal cost impact of total data complexity. Higher levels of structure can decrease the effort to comprehend a very complex application by decomposing the data processing into smaller pieces so that the functionality can be more easily understood. However, for applications with low total data complexity, high levels of structure are less beneficial as these applications can be more easily comprehended without it. In fact, high levels of structure can introduce complicated interfaces into these less difficult applications, thereby increasing the enhancement effort. This implies that the optimal level of structure increases with higher levels of total data complexity in an application.

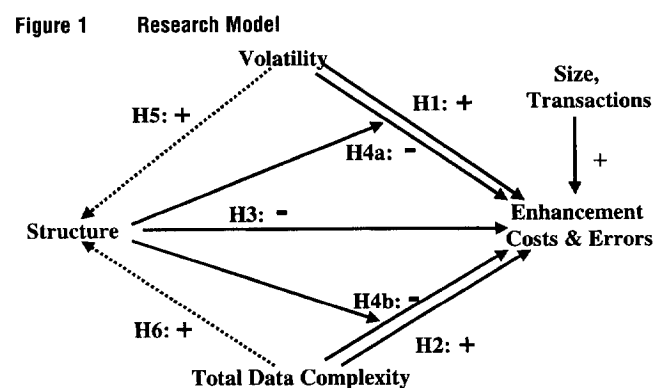
Our analysis of the life-cycle cost minimization problem (Appendix A) confirms that the optimal levels of structure increase with volatility as well as with total data complexity.

Therefore, we expect to find empirically that:

**HYPOTHESIS 5.** *Higher levels of structure are associated with higher levels of volatility.*

**HYPOTHESIS 6.** *Higher levels of structure are associated with higher levels of total data complexity.*

Our research model and hypotheses are depicted in Figure 1.



## Methodology

### Research Setting

To investigate our hypotheses concerning software enhancement outcomes, we conducted a field study in which we examined the software applications in two organizations: a national mass merchandiser (Merchandiser) and a large commercial bank (Bank).

**Merchandiser.** The Merchandiser is a national retailer. In the 1990s the company has been expanding its geographic presence and product lines by acquiring smaller retailers. The Merchandiser has a large portfolio of applications written in COBOL<sup>3</sup> and running on IBM mainframe computers. Most of this software was created in the 1970s and 1980s and is being enhanced in the 1990s to accommodate the growth and expansion of the organization. The Information Systems (IS) organization for the company supports all centralized computer processing activities. In 1983, the IS organization was divided into separate development and maintenance units. Development works exclusively on building new applications, and Maintenance implements enhancements as well as minor repairs to existing applications. In terms of resource allocation, half of the IS budget is spent on new development projects, and the other half on enhancement projects, corrective maintenance, and user support.

Enhancement projects are initiated by the end users in the company. Every year, each end user department is allocated a budget by the chief financial officer for enhancements, and the departments specify the projects they want the IS organization to complete according to the limits of their budgets. Every project must be cost justified in terms of providing business value.<sup>4</sup>

<sup>3</sup>COBOL is still heavily used in industry. It is estimated that tens of billions of lines of COBOL code are used daily in mission-critical applications worldwide, and that there are a million COBOL developers creating five billion lines of new COBOL code each year (*Software Economics Letter* 1998). Major corporations are estimated to spend between \$50 and \$100 million each year to extend their COBOL applications beyond the year 2000.

<sup>4</sup>It may be argued that the applications that are easier to maintain are modified more frequently. However, at our research sites, each enhancement project had to be justified in terms of its business value. Some enhancements were mandatory due to regulatory changes and had to be implemented, regardless of the condition of the application.

**Commercial Bank.** The Bank is a nationally and state-chartered financial institution. With the recent changes in interstate banking laws, the Bank has been aggressively pursuing a strategy of geographic and product diversification in the 1990s. To obtain economies of scale from the acquisitions, the Bank has been enhancing its existing applications to handle the data processing from the acquired institutions rather than managing multiple IS organizations. More than 50% of the Bank's IS budget is spent on enhancements to existing applications, 3% to 5% is spent on new development projects, and the remainder is allocated to feasibility studies, corrective maintenance, and technical conversions.

The IS organization for the Bank is centralized, and provides information services to the Commercial Banking, Mortgage, Credit Card and Trust/Brokerage groups. In contrast to the Merchandiser, the Bank's IS organization is structured along customer lines. Thus, an application team provides both development and support services to a set of internal clients. Like the Merchandiser, the majority of the Bank's software application portfolio consists of COBOL applications written in the 1970s and 1980s, and enhanced in the 1990s.

Enhancement projects at the Bank are initiated by the end users in the company. Every year, each end user department is allocated a budget by the chief financial officer for enhancements, and the departments specify the projects they want the IS organization to complete within the limits of their budgets. As with the Merchandiser, each enhancement project must demonstrate business value to be cost justified. In addition, a large number of enhancement projects at the Bank involve mandatory modifications to adhere to new federal and state banking regulations.

### Data

Detailed size, complexity, volatility, cost, and error data were collected for the Merchandiser's and Bank's software applications. Application size and enhancement size measured in terms of function points (Albrecht and Gaffney 1983) were obtained from counts by the quality assurance group and the information services consulting group in the organizations. The function point counts were archived in software metrics databases by application and enhancement project.

The primary source of the complexity metrics in both organizations was a software code-archiving system. We used commercial code analysis tools to obtain software complexity metrics for each of the organizations' applications. Enhancement frequency, hours, and costs for each application were collected from software metrics archives maintained by the Merchandiser's quality assurance group and the Bank's information services consulting group. Error data were available only for the Merchandiser's applications, and were extracted from the databases established by the Merchandiser's operations group, which had implemented an automated process for logging and archiving software errors.

### Measures

We define the constructs in our models and their measurement as follows. Application *enhancement costs* ( $ECOST_A$ ) are the total dollars that were incurred to modify the application during a specified time frame, where enhancement refers to additions, modifications, and deletions of functionality. In both companies, these costs are dominated by salaries and overhead and are based upon the number of hours logged by the application enhancement team members; major equipment purchases are not charged to a particular application. We measure enhancement costs over a three-year period from 1991 to 1994 in both organizations. We focus on that time period because the early 1990s was a time when major enhancement activity occurred in both organizations. In addition, multiple periods help to reduce noise and random variation in the cost data. For the same period, we assess task size in terms of the size of enhancements made to the applications. The total number of application *function points changed* ( $FPENH_A$ ) measures the size of modifications to the applications during the three years. This variable allows us to control for the impact of task size on enhancement effort.

We measure *software errors* ( $SERR_A$ ) over the same three years, 1991 to 1994. Software errors refer to the total number of failures of the application in a particular time period that were caused by software (not hardware) problems. Such problems required correction by the application support teams before the application could become operational. As a control measure for application usage, we measure the total

number of application *transactions* ( $TRANS_A$ ) over a similar time period.

The *total data complexity* ( $TOTDC_A$ ) of an application is expressed as the number of data elements input to and output from the application relative to its functionality:  $TOTDC_A = N2_A/FP_A$ , where  $N2_A$  = Halstead's "N2" metric (count of operands<sup>5</sup> in the application); and  $FP_A$  = number of application function points. Application *structure* ( $STRUC_A$ ) is defined in terms of the number of connections or calls per function in an application:  $STRUC_A = CALLS_A / FP_A$ , where  $CALLS_A$  = number of calls in an application; and  $FP_A$  = number of application function points.

We assess application *volatility* ( $VOLAT_A$ ) in terms of the total number of enhancements implemented over the three years ( $VOLAT_A = \#ENH_A/FP_A$ ). This measures the *frequency* with which the application was modified per unit of functionality in that time frame, where  $\#ENH_A$  = number of enhancements made to the application, and  $FP_A$  = number of application function points. We also operationalize and evaluate an alternative measure of volatility: the percent of application functionality that was modified in that time frame ( $VOLAT_A = FPENH_A/FP_A$ ). This measure assesses the *extent* of application volatility in a manner similar to Belady and Lehman (1976), where  $FPENH_A$  = number of application function points modified during the three-year period, and  $FP_A$  = number of application function points.

### Descriptive Statistics

Descriptive statistics for the variables in our models are presented in Table 1, and a correlation matrix is shown in Table 2.

### Analysis

To examine our hypotheses concerning software volatility, complexity, structure, and enhancement outcomes, we use moderated regression analysis (Cohen and Cohen 1983). That is, we first analyze enhancement cost and errors base models with main effects for

<sup>5</sup>Halstead's N2 metric sums all occurrences of operands in an application. That is why we use the label "total." While private variables as well as input and output data variables are included in N2, the vast majority of N2 includes input and output variables. Labeling "N2" as data complexity is not inconsistent with the literature in software engineering (e.g., Al-Janabi and Aspinwall 1993).

**Table 1 Descriptive Statistics**

Variable	MERCHANTISER (n = 23)			BANK (n = 41)		
	Mean	Median	Std Dev	Mean	Median	Std Dev
Total Data Complexity (N2/FP)	29.80	29.34	15.16	25.58	5.05	115.98
Structure (Calls/FP)	1.70	1.12	1.74	1.26	.10	7.05
Volatility (#Enhancements/FP)	1.81	1.76	1.59	0.97	0.08	5.09
Volatility (FP Enhanced/FP)	0.12	0.07	0.15	0.33	0.02	1.21
Function Points Enhanced	359.13	173.00	500.03	577.41	28.00	1389.41
Total Enhancement Costs	\$167,527.15	\$102,877.50	\$180,057.91	\$245,798.43	\$157,125.41	\$268,678.85
Software Errors	455.17	233.00	580.59	n/a	n/a	n/a
Transaction Usage	22,776.35	11,445.00	32,204.66	n/a	n/a	n/a
Application Function Points	2,368.26	1,847.00	1,693.16	5015.46	1,663.00	6982.45
Application Lines of Code	401,433.60	255,999.00	430,461.87	289,369.80	141,036.00	410,805.20

volatility, complexity, and structure, and then analyze moderated models including interactive effects for structure and complexity and for structure and volatility. We note that volatility, complexity, enhancement size, and application transactions are exogenous variables, and structure is an endogenous variable in our models:

$$\begin{aligned}
 (\text{ECOST}_A) = & \beta_{01} + \beta_{11}(\text{FPENH}_A) + \beta_{21}(\text{VOLAT}_A) \\
 & + \beta_{31}(\text{TOTDC}_A) + \beta_{41}(\text{STRUC}_A) + \beta_{51}(\text{STRUC}_A \\
 & * \text{VOLAT}_A) + \beta_{61}(\text{STRUC}_A * \text{TOTDC}_A) + \epsilon_{01}.
 \end{aligned}$$

$$\begin{aligned}
 (\text{SERR}_A) = & \beta_{02} + \beta_{12}(\text{TRANS}_A) + \beta_{22}(\text{VOLAT}_A) \\
 & + \beta_{32}(\text{TOTDC}_A) + \beta_{42}(\text{STRUC}_A) + \beta_{52}(\text{STRUC}_A \\
 & * \text{VOLAT}_A) + \beta_{62}(\text{STRUC}_A * \text{TOTDC}_A) + \epsilon_{02}.
 \end{aligned}$$

$$\begin{aligned}
 (\text{STRUC}_A) = & \beta_{03} + \beta_{13}(\text{VOLAT}_A) \\
 & + \beta_{23}(\text{TOTDC}_A) + \epsilon_{03}.
 \end{aligned}$$

We estimated the coefficients of the individual equations using ordinary least squares (OLS). We found that the error terms were not significantly correlated across equations. Thus, a generalized least squares (GLS) procedure that allows for correlation of disturbances across equations was not required for the estimation of this system (Greene 1997). As our equations form a recursive system with uncorrelated errors, OLS

estimates of the coefficients are consistent and efficient (Greene 1997, Kennedy 1992).

We verified that the residuals for our models satisfy the distributional assumptions (Greene 1997). The assumption of normality is not rejected for any of the models at the 5% significance level using the Shapiro-Wilk test (Shapiro and Wilk 1965). Collinearity as checked by the condition index and variance inflation factors was above acceptable levels in the moderated models. Thus, as recommended by Neter et al. (1990), we centered our independent variables by expressing each as a deviation from its mean value, and reran our analysis. This substantially reduced the collinearity in the models without significantly impacting the values of the coefficients. Outlier analysis (Belsley et al. 1980) indicated that there were no influential outliers in our enhancement costs and errors models. No evidence of heteroscedasticity in the models was detected by Glejser's test (1969) and White's test (1980).

The estimates of the coefficients for our models are presented in Tables 3 through 5. We evaluated the robustness of our results in several ways. We estimated rank regressions for the models (Iman and Conover 1979). The results from the rank regressions are generally consistent with the results from the estimation of the original models. Further, we checked the robustness of our results using our alternative measure of volatility (FPENH<sub>A</sub>/FP<sub>A</sub>). We find that the sign and significance of the variables are generally consistent between our original formulation and the models with

**Table 2** Correlation Matrix

	Total Data Complexity	Structure	Volatility (#Enh/FP)	Volatility (FP Enh/FP)	Function Points Enhanced	Transaction Usage
Total Data Complexity						
Merchandiser	1.000	0.809*	0.203	0.256	0.155	0.114
Bank	1.000	0.874*	0.292*	0.021	-0.223	n/a
Structure						
Merchandiser	0.867*	1.000	0.391*	0.517*	0.445*	0.263
Bank	0.899*	1.000	0.384*	0.162	-0.070	n/a
Volatility (#Enh/FP)						
Merchandiser	0.329	0.435*	1.000	0.723*	0.579*	0.388*
Bank	0.178	0.221	1.000	0.653*	0.442*	n/a
Volatility (FP Enh/FP)						
Merchandiser	0.361*	0.480*	0.490*	1.000	0.886*	0.499*
Bank	0.147	0.198	0.897*	1.000	0.893*	n/a
Function Points Enhanced						
Merchandiser	0.222	0.454*	-0.697*	0.748*	1.000	0.606*
Bank	-0.134	-0.006	0.138	0.307*	1.000	n/a
Transaction Usage						
Merchandiser	0.219	0.289	0.174	0.326	0.276	1.000
Bank	n/a	n/a	n/a	n/a	n/a	n/a

Notes: \* indicates two-tailed significance at 10% level. Spearman correlations in upper right of matrix; Pearson correlations in lower left of matrix.

the alternative measure of volatility, supporting the robustness of the results.<sup>6</sup>

#### Tests of Hypotheses 1, 2, and 3 (Software Enhancement Main Effects)

To evaluate our hypotheses about the main effects of volatility and complexity, we differentiate our software enhancement moderated models with respect to the particular effect under consideration and examine the change in enhancement costs and errors with a unit change in that effect, holding structure constant at its mean value (Neter et al. 1990, Greene 1997).<sup>7</sup> Hypothesis 1, about the positive effect of volatility, is supported in the software errors model; however, this hypothesis is not supported in the enhancement cost models in either organization. Hypothesis 2, concerning the effect of total data complexity on software en-

<sup>6</sup>Interested readers may obtain the results from this analysis by contacting the second author of this paper.

<sup>7</sup>For example, to assess the main effect of volatility on enhancement costs, we evaluate whether the following expression is significantly different from zero:  $\partial \text{ECOST}_A / \partial \text{VOLAT}_A = \beta_{21} + \beta_{51}(\text{STRUC}_A)$ , inserting the mean value of  $(\text{STRUC}_A)$ .

hancement costs and errors, is supported. While the coefficient for application structure is negative (as we have posited in our Hypothesis 3), it is not always statistically significant in the enhancement cost-and-errors models.

Thus, we find mixed empirical support for our hypotheses concerning the main effects of volatility, complexity, and structure on enhancements costs and errors.

#### Tests of Hypothesis 4 (Software Enhancement Moderated Effects)

Moderated models are supported in both organizations. That is, the change in  $R^2$  between the main effects and moderated models is significant for enhancement costs (Bank: F Change = 4.428,  $p = 0.020$ ; Merchandiser: F Change = 4.085,  $p = 0.037$ ) and errors (Merchandiser: F Change = 6.158,  $p = 0.010$ ). As hypothesized, there is a negative and significant interactive effect of structure and volatility in the enhancement cost-and-errors models in both organizations. The coefficient of the interaction between structure and total data complexity, while negative, is statistically significant only in the software errors model.

**Table 3 Results: Software Enhancement Base Model**

$$\begin{aligned}
 (\text{ECOST}_A) &= \beta_{01} + \beta_{11}(\text{FPENH}_A) + \beta_{21}(\text{VOLAT}_A) + \beta_{31}(\text{TOTDC}_A) \\
 &\quad + \beta_{41}(\text{STRUC}_A) + \epsilon_{01} \\
 (\text{SERR}_A) &= \beta_{02} + \beta_{12}(\text{TRANS}_A) + \beta_{22}(\text{VOLAT}_A) \\
 &\quad + \beta_{32}(\text{TOTDC}_A) + \beta_{42}(\text{STRUC}_A) + \epsilon_{02}
 \end{aligned}$$

Variable	ENHANCEMENT COSTS		SOFTWARE ERRORS
	MERCHANDISER (n = 23)	BANK (n = 41)	MERCHANDISER (n = 23)
	Estimate	Estimate	Estimate
	t-value	t-value	t-value
	(p-value)	(p-value)	(p-value)
Intercept	<b>7.508</b>	<b>9.869</b>	<b>-0.814</b>
	4.508	15.367	-1.526
	(0.001)	(0.001)	(0.066)
FPs Enhanced or Transactions	<b>0.507</b>	<b>0.904</b>	<b>0.642</b>
	1.980	3.834	12.348
	(0.027)	(0.001)	(0.001)
Volatility	<b>0.356</b>	<b>-0.861</b>	<b>0.199</b>
	0.801	-0.484	1.635
	(0.213)	(0.315)	(0.054)
Total Data Complexity	<b>2.209</b>	<b>2.208</b>	<b>0.628</b>
	1.806	2.896	1.614
	(0.038)	(0.003)	(0.056)
Structure	<b>-0.343</b>	<b>-1.714</b>	<b>-0.183</b>
	-0.451	-2.306	-0.769
	(0.327)	(0.012)	(0.223)
R <sup>2</sup>	0.636	0.368	0.906
Adjusted R <sup>2</sup>	0.555	0.297	0.885
F Model	7.052	6.237	46.814
(p-value)	(0.001)	(0.001)	(0.001)

Note: \*p-values are one-tailed.

Thus, we find empirical support for Hypotheses 4a and 4b, which posit that structure mitigates the cost impact of volatility in terms of reducing enhancement costs and errors, and mitigates the cost impact of total data complexity in terms of reducing enhancement errors.

#### Tests of Hypotheses 5 and 6 (Levels of Structure)

We find empirical support for Hypothesis 5, as the coefficient for volatility in the software structure model

is positive and significant in both organizations. Hypothesis 6 is also supported, as the coefficient for total data complexity in the software structure model is positive and significant in both organizations. Thus, we find empirically that higher levels of structure are associated with increased total data complexity and volatility.

### Predicting Complexity and Volatility

If, as we have demonstrated, structure mitigates the costs of volatility and complexity in software enhancement, from the perspective of effective software management, it would be highly desirable to be able to predict in the design phase which applications could be expected to be more complex or more volatile after they have been implemented. This would enable the designers to focus their efforts on structuring the applications where the cost benefit is greatest. Unfortunately, there has been relatively little research to systematically predict ex ante in design where complexity and volatility are likely to occur in software enhancement.

In the software engineering literature, there is a notion that applications of a similar type or algorithmic domain experience similar levels of complexity and productivity. Many software cost estimation models include application type as a significant cost adjustment factor (also called the "fourth factor" in Putnam and Myers (1997); the "mode" in Boehm's COCOMO model (1981); the "project type" in Jones' SPQR model (1991); the "influential factor" in Albrecht and Gaffney's Function Point model (Kemerer 1993); and "algorithm type" in Jones' feature point model (1991)).

Application type is often specified in these models in terms of a continuum ranging from nonprocedural, to batch processing, to on-line processing, to decision support, to real-time and embedded, to diagnostic and predictive applications. Implicit in this taxonomy of application types is the idea that applications with rule-based algorithms are less complex than are applications with induction-based algorithms (Jones 1991). That is, rule-based applications have processes determined by reasonably well-defined rules that can be precisely articulated and whose complexity is more finite and bounded. An example is an accounts receivable application that performs arithmetic operations

**BANKER AND SLAUGHTER**  
*The Moderating Effects of Structure*

**Table 4 Results: Software Enhancement Moderated Model**

$$\begin{aligned}
 (\text{ECOST}_A) &= \beta_{01} + \beta_{11}(\text{FPENH}_A) + \beta_{21}(\text{VOLAT}_A) + \beta_{31}(\text{TOTDC}_A) \\
 &+ \beta_{41}(\text{STRUC}_A) + \beta_{51}(\text{STRUC}_A * \text{VOLAT}_A) + \beta_{61}(\text{STRUC}_A * \text{TOTDC}_A) + \epsilon_{01} \\
 (\text{SERR}_A) &= \beta_{02} + \beta_{12}(\text{TRANS}_A) + \beta_{22}(\text{VOLAT}_A) + \beta_{32}(\text{TOTDC}_A) \\
 &+ \beta_{42}(\text{STRUC}_A) + \beta_{52}(\text{STRUC}_A * \text{VOLAT}_A) + \beta_{62}(\text{STRUC}_A * \text{TOTDC}_A) + \epsilon_{02}
 \end{aligned}$$

Variable	ENHANCEMENT COSTS		SOFTWARE ERRORS
	MERCHANTISER (n = 23)	BANK (n = 41)	MERCHANTISER (n = 23)
	Estimate	Estimate	Estimate
Variable	t-value	t-value	t-value
	(p-value)	(p-value)	(p-value)
Intercept	<b>8.398</b>	<b>10.392</b>	<b>-0.141</b>
	5.089	15.975	-0.291
	(0.001)	(0.001)	(0.386)
FPs Enhanced or Transactions	<b>0.589</b>	<b>0.950</b>	<b>0.638</b>
	2.534	4.319	14.261
	(0.007)	(0.001)	(0.001)
Volatility	<b>0.416</b>	<b>-1.114</b>	<b>0.229</b>
	1.049	-0.687	2.312
	(0.150)	(0.247)	(0.012)
Total Data Complexity	<b>2.210</b>	<b>1.354</b>	<b>0.422</b>
	1.967	1.915	1.286
	(0.028)	(0.030)	(0.102)
Structure	<b>-0.930</b>	<b>-0.625</b>	<b>-0.308</b>
	-1.377	-0.879	-1.579
	(0.088)	(0.191)	(0.061)
Structure * Volatility	<b>-0.542</b>	<b>-0.177</b>	<b>-0.137</b>
	-2.223	-3.021	-1.716
	(0.016)	(0.002)	(0.046)
Structure*	<b>-0.284</b>	<b>-0.103</b>	<b>-0.226</b>
Total Data Complexity	-0.857	-0.808	-2.151
	(0.198)	(0.211)	(0.018)
R <sup>2</sup>	0.759	0.498	0.947
Adjusted R <sup>2</sup>	0.668	0.410	0.927
F Model	8.647	5.960	51.521
(p-value)	(0.001)	(0.001)	(0.001)

Note: \* p-values are one-tailed.

and data selection and manipulation activities. In contrast, induction-based applications are less bounded and less precise. These applications examine large numbers of specific instances and attempt to derive the rules that govern the class memberships of the in-

**Table 5 Results: Software Structure Model**

$$(\text{STRUC}_A) = \beta_{03} + \beta_{13}(\text{VOLAT}_A) + \beta_{23}(\text{TOTDC}_A) + \epsilon_{03}$$

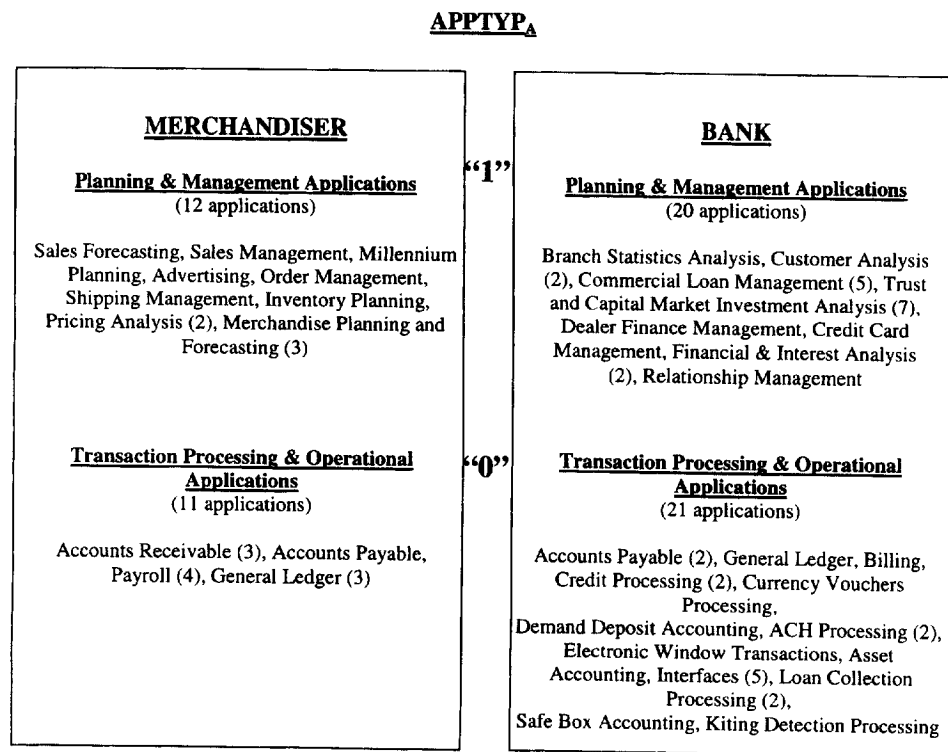
Variable	MERCHANTISER (n = 23)	BANK (n = 41)
	Estimate	Estimate
Variable	t-value	t-value
	(p-value)	(p-value)
Intercept	<b>-1.829</b>	<b>-0.000</b>
	-5.911	-0.003
	(0.001)	(0.499)
Volatility	<b>0.165</b>	<b>0.058</b>
	1.451	2.098
	(0.076)	(0.020)
Total Data Complexity	<b>1.406</b>	<b>0.798</b>
	7.012	8.994
	(0.001)	(0.001)
R <sup>2</sup>	0.777	0.702
Adjusted R <sup>2</sup>	0.754	0.687
F Model	32.483	44.254
(p-value)	(0.001)	(0.001)

Note: \* p-values are one-tailed.

stances. An example is a strategic planning application that does environmental scanning, forecasting, rule derivation, and pattern recognition.

In a similar way, the literature in management information systems classifies applications into different types based upon the nature of the management activities they support (Laudon and Laudon 1997, Davis and Olson 1985, Gorry and Scott-Morton 1971, Head 1967). At the base of the pyramid are the basic transaction-processing and operational applications, such as payroll disbursement, general ledger, and order processing. These applications are clerical in nature, and process transactions in a relatively straightforward manner. At the peak of the pyramid are planning and management applications, such as sales forecasting, pricing analysis, and inventory planning, which process data in more complicated ways needed to support the decision-making, planning, and control activities of knowledge workers and managers. In this taxonomy, the transaction-processing and operational applications describe structured, well-defined procedures and decisions, while the planning and

**Figure 2 Two Types of Applications**



management applications represent more ad hoc, unstructured processes and decisions (Davis and Olson 1985).

The planning and management applications are likely to be more complex and more volatile in terms of frequency of enhancement than the operational and transaction-processing applications. Transaction-processing and operational applications rely on pre-established internal procedures, data, and decision algorithms that tend to be more deductive, specifiable, and narrower in scope than those embedded in planning and management applications (Davis and Olson 1985). Decision rules in the planning and management applications are less programmable (Gorry and Scott Morton 1971), with more inductive algorithms, and often depend upon data and factors external to the organization that are likely to be highly dynamic. These decision-support applications also tend to be more interactive than batch in nature, with a significant amount of the software devoted to user interfaces that also are likely to be very dynamic. Therefore, planning

and management applications are subject to greater pressure for change and are likely to be enhanced more frequently than the other types of applications.

To determine whether this is the case at our research sites, we construct a binary variable ( $APPTYP_A$ ) to distinguish between two major application types in the organizations. A “0” designates the Merchandiser’s and the Bank’s basic transaction-processing and operational applications, and a “1” designates the Merchandiser’s and the Bank’s planning and management applications. To classify each application, we interviewed application managers, referenced guidelines for categorizing management activities (Davis and Olson 1985, pp. 41–44), and reviewed company documentation that described the primary purpose of each application.

Figure 2 shows the Merchandiser’s and Bank’s applications classified into their respective application types. A descriptive profile by application type is presented in Table 6. As Table 6 suggests, there are no significant differences between average enhancement

costs, software errors, and size for each application type in the organizations, with the exception of enhancement costs which are significantly higher for the Bank's planning and management applications. The age for an application is calculated by averaging the installation date for every module in the application. Age has been argued to approximate for "entropy" in software applications (Gode et al. 1990, Jones 1989, Vessey and Weber 1983). As indicated in Table 6, we find no statistical difference in the average age between application types. This suggests that age differences in the applications in our sample do not explain differing amounts of structure due to the effects of entropy.

We compare the mean and median volatility for the different application types. As Table 7 indicates, the planning and management applications are enhanced more often on average and per function point than the transaction-processing and operational applications. This is true for the alternative measures of volatility. These results provide support for our assertion

that planning and management applications are relatively more volatile in these organizations.

Comparing mean and median levels of total data complexity and structure for the applications (Table 7) also shows that structure and complexity are significantly higher on average for the planning and management applications. It is striking to note that for the Merchandiser, while total data complexity for planning and management applications is *twice* as high as total data complexity for transaction-processing and operational applications, levels of structure for planning and management applications are *ten* times as high. For the Bank, total data complexity for planning and management applications is *six* times as high as for transaction-processing and operational applications, but levels of structure for planning and management applications are *fourteen* times as high.

## Discussion

The results provide support for our assertions that, given the greater complexity and expected volatility

**Table 6** Descriptive Statistics by Application Type

Variable	MERCHANTISER			BANK		
	Mean for Transaction Processing and Operational Applications (n = 11)	Mean for Planning and Management Applications (n = 12)	t-test (one-sided p-values)	Mean for Transaction Processing and Operational Applications (n = 21)	Mean for Planning and Management Applications (n = 20)	t-test (one-sided p-values)
Enhancement Costs	\$146,071.00	\$187,195.13	0.298	\$201,848.48	\$331,708.67	0.069
Software Errors	542.36	375.00	0.251	n/a	n/a	n/a
Transaction Usage	26,156.55	19,677.67	0.320	n/a	n/a	n/a
Application Function Points	2,174.73	2,545.67	0.306	5,250.52	4,768.65	0.414
Average Year of Installation	1986.55	1987.33	0.215	1986.38	1988.00	0.129

Variable	MERCHANTISER			BANK		
	Median for Transaction Processing and Operational Applications (n = 11)	Median for Planning and Management Applications (n = 12)	Mann Whitney test (one-sided p-values)	Median for Transaction Processing and Operational Applications (n = 21)	Median for Planning and Management Applications (n = 20)	Mann Whitney test (one-sided p-values)
Enhancement Costs	\$95,395.50	\$145,329.30	0.162	\$117,023.30	\$203,341.10	0.003
Software Errors	233.00	256.50	0.451	n/a	n/a	n/a
Transaction Usage	9,205.00	11,962.00	0.322	n/a	n/a	n/a
Application Function Points	1,622.00	2,023.50	0.249	2,498.00	1,533.00	0.274
Average Year of Installation	1987.00	1988.00	0.250	1988.00	1988.00	0.341

**BANKER AND SLAUGHTER**  
*The Moderating Effects of Structure*

**Table 7 Comparison of Mean and Median Volatility, Complexity, and Structure by Application Type**

Variable	MERCHANDISER			BANK		
	Mean for Transaction Processing and Operational Applications (n = 11)	Mean for Planning and Management Applications (n = 12)	t-test (one-sided p-values)	Mean for Transaction Processing and Operational Applications (n = 21)	Mean for Planning and Management Applications (n = 20)	t-test (one-sided p-values)
Function Points Changed	207.45	497.58	0.085	304.00	863.85	0.101
Ratio of Function Points Changed/ Application Function Points	0.07	0.17	0.054	0.06	0.62	0.073
Total Number of Enhancements	3,739.73	6,024.33	0.073	245.69	673.35	0.032
Ratio of Total Number of Enhancements/ Application Function Points	1.72	2.37	0.069	0.07	1.91	0.063
Total Data Complexity	19.51	39.22	0.003	7.46	44.61	0.156
Structure	0.29	2.99	0.001	0.09	1.24	0.098

Variable	MERCHANDISER			BANK		
	Median for Transaction Processing and Operational Applications (n = 11)	Median for Planning and Management Applications (n = 12)	Mann Whitney test (one-sided p-values)	Median for Transaction Processing and Operational Applications (n = 21)	Median for Planning and Management Applications (n = 20)	Mann Whitney test (one-sided p-values)
Function Points Changed	34.00	270.50	0.095	21.00	31.50	0.130
Ratio of Function Points Changed/ Application Function Points	0.01	0.14	0.052	0.01	0.04	0.056
Total Number of Enhancements	3,000.80	6,150.40	0.048	89.47	250.29	0.016
Ratio of Total Number of Enhancements/ Application Function Points	2.16	2.44	0.075	0.05	0.12	0.002
Total Data Complexity	10.92	39.42	0.002	5.05	5.17	0.428
Structure	0.26	2.68	0.001	0.05	0.16	0.001

for the planning and management applications, higher levels of structure have been implemented in these applications and provide downstream advantages in terms of reduced enhancement costs and errors. But what promoted such efficient design choices in the organizations?

At an individual level, software engineers may not make efficient design choices. Prior experimental research on decision making and incentives in psychology (Paich and Sterman 1993, Hogarth et al. 1991, Sterman 1989, Kahneman and Tversky 1982) indicates that cognitive constraints limit the ability of decision makers to consider complicated long-term implications of their actions. For software engineering decisions, a lack

of precise measures of maintainability as well as inappropriate incentives can also contribute to the difficulty of optimizing maintenance productivity (Banker and Slaughter 1997, Banker and Kemerer 1992). This suggests that software engineers would individually do a relatively poor job of making design decisions that have long-range consequences for maintenance.

However, at a departmental or organizational level, there may be greater knowledge of the effects of design choices on maintenance costs than at the individual level. The organization's perspective is more encompassing, including both development and maintenance, and the organization can draw upon an extensive collection of past experiences to learn more

efficient practices (Harkness et al. 1996). Organizations can promote efficient design choices by implementing a number of different motivational mechanisms including incentives, standards, tools and techniques, job assignment, organizational structure, training, and other personnel policies (Nadler and Lawler 1977). Reward schemes such as those that reward code reuse can provide incentives for programmers to act efficiently. Programming standards can be established for coding, such as the optimal size and the maximum defect rate. Organizations can choose tools and methodologies that support such standards. To ensure that developers are sensitive to maintainability concerns, organizations can rotate technical and managerial personnel between development and maintenance departments (if the functions are separate) or between development and maintenance projects if the departments are combined (Swanson and Beath 1989).

To gain deeper insight into the software design decisions for these applications, we examined the motivational mechanisms in place at each organization. We also interviewed managers and programmers who had long tenure in the companies to gain insight into the development practices employed to create the different applications.

#### **Motivational Mechanisms at the Merchandiser**

**Incentives.** Characteristics of the dynamic retailing market and the focus on cost control to fund acquisitions contribute to incentives that reward both development and maintenance productivity at the Merchandiser. A key productivity measure by which development and maintenance are evaluated is the "delivery rate" or the number of project function points delivered per hour. An additional productivity measure for maintenance is the "support rate" or the number of application function points supported per person. With the acquisition of other retailers, the average support rate is expected to increase as functionality is consolidated, and errors and redundancies are removed.

**Tools, Techniques, Standards, and Training.** To support goals for high productivity in maintenance, the Merchandiser has instituted tools, techniques, standards, and training that encourage maintainability of

applications. "Turnover" standards have been established, which new applications must meet before they can be transferred from development to maintenance. Over time, these standards have become particularly stringent for planning and management applications that are likely to be frequently enhanced, and are intended to prevent delays in implementation as well as extensive rework in maintenance. As a result, software designers are more conscious of maintainability concerns for the planning and management applications.

A significant example concerns the use of CASE (computer-aided software engineering) tools. The Merchandiser uses a major CASE tool to generate code for many applications. The CASE tool creates standard COBOL routines and also allows for the incorporation of "custom" logic in the generated code to adapt its standard routines to specific end-user requirements. Development and maintenance programmers have been trained to implement custom logic in the generated code in a structured manner by using subroutines invoked with "CALL" statements. This practice reflects the belief that it is easier to maintain the custom logic if it is structured and less integrated into the generated code. In addition, the subroutines may be referenced by multiple functions, so there is potential for code reuse.

We investigated whether there were variations in application type related to the use of tools, techniques, and standards. We find that structure is significantly associated with use of the CASE tool (pairwise correlation of 0.871, significant at  $p < 0.001$ ). Furthermore, the CASE tool is used significantly more for the planning and management applications than for the transaction-processing and operational applications. On average, the CASE tool is used for 40.72% of the programs in the planning and management applications and for 3.17% of the programs in the transaction-processing and operational applications. The average use is significantly different ( $F = 28.845$ ,  $p < 0.001$ ).

Our interviews with development and maintenance managers suggest that the primary motives for use of the CASE tool in the planning and management applications is to assist in the requirements determination and modeling process, and to improve development and maintenance productivity by enabling reuse

of code and design models. Because the planning and management applications have higher strategic importance to the company and are perceived to be more volatile and more complex, managers expect that higher levels of structure and the code reuse enabled by the tool will provide significant cost savings in maintenance. For example, when questioned about the motivation for using CASE tools for the planning and management applications, the maintenance director replied:

CASE is the key to improving quality without degrading our productivity. We should see less user support and fewer repairs, because the development process results in better determination of requirements. Also, our code is more structured and we can re-use system components, so our delivery and support rates are higher.

(Interview transcript, maintenance director, January 11, 1993)

**Job Assignment and Organization Structure.** The separation of the development and maintenance functions could cause problems in the transfer of applications from one group to another. However, job assignment is used to compensate for these potential difficulties. Both technical and managerial personnel rotate assignments between groups. This rotation of personnel increases sensitivity to maintenance concerns. As one programmer commented:

In making the switch to Development, I'm more conscious of software quality. I don't want to create problems. In Support I could identify the problem and work on it without affecting anyone.

(Interview transcript, February 1, 1993,  
development programmer)

One result of this job rotation strategy is the institution of and adherence to standards for the use of structured techniques when implementing custom logic in the code generated by the CASE tool.

### **Motivational Mechanisms at the Bank**

**Incentives.** In contrast to the Merchandiser, the Bank employs a significant number of contract programmers and consultants. Often the IS department competes with consultants and contractors for project work with the Bank's business units. As a result, there is a focus on productivity and cost control within the IS department, to the extent that project dates and effort estimates are set aggressively. Most

software projects at the Bank involve enhancements to existing applications, particularly to the more externally-oriented planning and management applications that have been heavily modified in the 1990s to accommodate the acquired banks' information processing. Because of the focus on enhancement productivity, a central concern in these projects is the ability of the project team to understand the software code that is to be modified:

I think the most critical factor for project success is whether the team understands the software—how it works and how it's supposed to work.

(Interview transcript, July 13, 1994, project leader)

**Tools, Techniques, and Standards.** Structured methodologies are viewed as essential in facilitating understanding of the software code, particularly in applications that are complex and frequently modified. There is an emphasis on the use of structured methodologies, and an internal consulting group has been formed to provide advice and support to the application teams for a variety of structured methodologies. Both programmers and managers stress the importance of structured methodologies. For example:

This methodology works very successfully. The data model is solid. Structured analysis has a great effect on design and coding. There's a tight correlation between models and systems. . . . I'm a fan of the methodology. Maintenance should be easy. Coding errors should be minimal. We are well-positioned for the next steps, for changes to the existing system.

(Interview transcript, July 13, 1994, application programmer)

The comments of a manager reflect the sensitivity to maintenance concerns for the more volatile and complex planning and management applications:

Our legacy systems are mainframe COBOL. A major issue for us historically has been how to increase productivity. Many of these mainframe systems, like the ones in my area, are really complex. We change them all the time, and when you make changes to them, they add to the complexity. . . . The key thing about the code is understanding it. I mean, is it modularized? Was a structured methodology used? Or is it just a bunch of spaghetti code?

(Interview transcript, July 18, 1994, application manager,  
trust/capital market investment applications)

We investigated whether structured methodologies are selectively used for the planning and management

applications by examining data on application development practices. When the Bank's information services consulting group began collecting application function point data, the group asked the application managers to evaluate the use of a number of development practices for each application. Managers assigned each application a three-point ranking to indicate the degree of use of structured programming techniques in the development of the application (1 = none, 2 = some, and 3 = extensive). We confirmed these rankings using a software code analysis tool.<sup>8</sup> We found that the mean ranking for use of structured techniques for the planning and management applications (2.25) is significantly higher than the mean ranking (1.81) for the transaction-processing and operational applications ( $F = 3.102$ ,  $p = 0.043$ ), supporting selective use of structured methodologies for planning and management applications.

**Job Assignment and Organization Structure.** At the Bank, the combined development and maintenance roles of managers and programmers as well as the focus on enhancement productivity encourage sensitivity to maintainability concerns and the use of structured methodologies, particularly for the more volatile customer-oriented applications.

Overall, we find that motivational mechanisms at the Merchandiser and at the Bank promote efficient design choices by encouraging the use of structure for the more volatile and complex planning and management applications.

## Concluding Remarks

### Implications and Directions for Further Research

We have posited that structure mitigates the cost of the volatility and complexity of data processing in software enhancement. Our results indicate that there are

<sup>8</sup>The tool assessed the degree to which the programs in the software applications conformed to the tenets of structured methodologies. Specifically, the tool tested for the use of PERFORM-based control logic, modular organization, and routines with one entry and exit point. Based upon how well these criteria were met, the analyzer classified each application as highly structured, structured, or unstructured. We compared these classifications to the ranking of the managers to ensure that applications were similarly ranked. Differences were resolved by using the tool's rating as it is a more objective measure.

optimal levels of structure in software applications that increase with volatility and complexity, and suggest that it is beneficial to implement higher levels of structure for more volatile or more complex applications. At our research sites, we find that it is more advantageous to structure planning and management applications that are more complex and that are likely to be enhanced frequently. Structure facilitates change by localizing information processing. However, for the relatively stable and less complex transaction-processing and operational applications, there is less downstream benefit to structure.

One rather controversial implication of our study is that high levels of investment in software quality practices are not economically efficient in all situations. Advocates of structured design, software process improvement, and other quality practices have argued strongly for the cost benefits of investment in software quality practices (e.g., Jones 1997b). Recent empirical studies have supported the significance of better processes and increased investment in the initial stages of design for improving the quality of software products as well as the productivity of development and error correction (Krishnan 1996).

Our results do not contradict these findings, and we do not advocate ad hoc development of software. Rather, we have suggested that high investment in certain quality practices may not pay off for all software applications. We have focused on a particular quality practice (structured design) and on software enhancement activities, which are different in many respects from software development. Software enhancement requires comprehension of the existing software, and structure influences the efficacy of comprehension. In this context, we have shown that structure plays a critical role in influencing the cost impact of volatility and complexity. Our results imply that it is not optimal to highly structure all applications; selective use of structure for the more volatile or more complex applications offers the greatest cost benefits in software enhancement.

Our empirical results suggest that the benefits of structure are greatest in mitigating the enhancement cost effects of volatility; for complexity, we found that structure led to reduced errors, but had no statistically

significant effect on enhancement costs in either organization. This finding could be specific to our research sites, but may generalize to other organizations. For complex applications, structure may not be as effective in reducing effort (and costs) because while structure mitigates data complexity, it introduces interface complexity. However, increased structure may lead to fewer errors because it may be easier to debug smaller functions. For volatile applications, the increasing familiarity with the structure magnifies the benefits of structure without adding to the costs of structure. On the other hand, with increased volatility is an increased probability of introducing errors, and structure may not completely offset this probability. Research to further explore these findings would be instructive.

It would also be useful to develop and empirically evaluate a more detailed model of the development and total life-cycle cost trade-offs to implement structure and other quality practices. Such a model might include other application characteristics besides application type to predict the more volatile and more complex applications that benefit the most from software structure and other quality techniques. At our research sites, application type is a good indicator of complexity and expected volatility. At other organizations, other variables may be associated with complexity and volatility or finer level distinctions may be necessary. For example, the use of a particular development tool (such as a CASE tool) may create applications that have different levels of complexity than applications where CASE has not been deployed, because the CASE tool may enforce a particular development discipline. It could also be that applications that are more unique or specific in supporting an organization's core competence are more likely to be frequently changed, as the organization may invest more in adapting applications that are central to its strategic mission. An interesting direction for further research would be to identify and model the areas within software applications that are expected to be more complex or more volatile. Such a model could be used to more precisely identify the software components where the return to quality practices is greatest.

At an individual level, software engineers may have neither the knowledge nor the motivation to make efficient design choices. Therefore, another implication

of our study is that it is important for organizations to institute a number of motivational mechanisms to encourage the use of good practices where they are most beneficial. At our research sites, we found that managers used a variety of motivational mechanisms including incentives, standards, tools, job assignment, and training to encourage the use of structure, particularly for the planning and management applications where it has the largest benefit for enhancement performance. Our study implies that the impact of organizational incentives in promoting or discouraging the use of optimal software practices is significant. Another important direction for future research of performance in software tasks is to more formally examine the link between incentives, use of software practices, and performance.

Finally, we have focused on COBOL environments. We believe our study and findings could readily be extended into other development environments such as object orientation. The fundamental design principles of decomposing complexity and balancing cohesion and coupling still apply (Bitman 1997, Chidamber and Kemerer 1994, Honiden et al. 1993, Booch 1991).<sup>9</sup> As Booch describes, object class design (much like structured design) is an iterative process that involves creating new classes based on existing ones (subclassing), splitting existing classes into smaller ones (factoring), and combining several existing classes into one class (composition). As the designer splits large classes into smaller ones, more references or calls are needed to pass information between them, and unless encapsulation techniques are used, coupling will increase (Bitman 1997). Analogous to our results, researchers who have studied software maintenance in object-oriented environments observe that as the number of small classes increases, maintenance effort and errors are higher because of increases in the number of relationships that a maintainer must understand and the difficulties in tracing interdependencies between

<sup>9</sup>In an object-oriented design, coupling occurs when a method or object uses methods or instance variables of another object; classes are coupled when methods declared in one class use methods or instance variables of the other class. An object class is cohesive if different methods perform different operations on the same set of instance variables (Chidamber and Kemerer 1994).

classes (Hsia et al. 1995, Wilde et al. 1993).<sup>10</sup> An interesting direction for future research would be to empirically evaluate the software life-cycle cost benefits and trade-offs involved in balancing complexity, coupling, and cohesion in object-oriented environments.

### Implications for Practice

Our study has a number of implications for practitioners, particularly for software managers and organizations interested in software process improvement. First, our results highlight the importance of implementing design choices that facilitate enhancement in organizations where software applications are long-lived and require significant effort to enhance and maintain. Given the time and resource constraints in software development, we suggest that there is a benefit to focusing quality efforts on those applications that will offer the greatest return over the software life cycle. Second, our results demonstrate the economic cost benefit of considering volatility and complexity in making software design choices. In particular, we find that structured design is especially cost effective for the applications that are more volatile or more complex. At our research sites, application type is a good predictor of the more volatile and complex applications. At other organizations, different factors may be useful in identifying the applications that will benefit from more structure. Finally, our results suggest the significance of motivational mechanisms in promoting effective software design choices.

**Acknowledgments.** The authors thank the editor, the associate editor, and five anonymous reviewers for their valuable comments and suggestions. We gratefully acknowledge research support from the Graduate School of Industrial Administration at Carnegie Mellon University. The cooperation and assistance of managers and staff at our data sites were invaluable. Helpful comments were provided by participants in research seminars at the University of Rochester, Carnegie Mellon University, and the Workshop for Information Systems and Economics (WISE).

<sup>10</sup>Similarly, in the validation of the metrics they propose for object-oriented design, Chidamber and Kemerer (1994) note that experienced object-oriented designers reported more perceived complexity and greater difficulty in memory management and run time detection of errors when there are a large number of small classes to deal with (p. 490).

## Appendix A Comparative Statics for the Optimal Levels of Software Structure

The optimal level of structure is chosen to minimize the total life-cycle costs as depicted in the following:

Minimize  $DEV(STR;TOT) + \rho ENH(STR;TOT;VOL)$

**STR**

where

**DEV(•)** is Development Cost

**ENH(•)** is Enhancement Cost

$\rho$  is the present value factor to make downstream Enhancement Costs comparable to Development Costs

**STR** is Structure

**TOT** is Total Data Complexity

**VOL** is volatility

and other variables are suppressed for the purposes of analysis.

We employ the following notation:

$F_i = \partial F / \partial i$  and  $F_{ij} = \partial^2 F / \partial i \partial j$ , where  $F = DEV, ENH$  and  $ij = STR, TOT, VOL$ .

We assume an interior solution to the optimization problem, satisfying the following first and second order conditions:

(I) First Order Condition:  $DEV_{STR} + \rho ENH_{STR} = 0$ .

(II) Second Order Condition:  $DEV_{STR,STR} + \rho ENH_{STR,STR} > 0$ .

The two conditions together imply that the weighted sum of development and enhancement costs decreases initially as the level of structure increases, but begins to increase once the optimal level of structure is exceeded.

We wish to examine how the optimal level of structure varies with the levels of complexity and volatility.

Differentiating (I) with respect to **VOL**, we obtain:  $DEV_{STR,VOL} + DEV_{STR,STR} STR^*_{VOL} + \rho [ENH_{STR,VOL} + ENH_{STR,STR} STR^*_{VOL}] = 0$ . Therefore,

(III)  $STR^*_{VOL} = -(DEV_{STR,VOL} + \rho ENH_{STR,VOL}) / (DEV_{STR,STR} + \rho ENH_{STR,STR})$ .

Differentiating (I) with respect to **TOT**, we obtain:  $DEV_{STR,TOT} + DEV_{STR,STR} STR^*_{TOT} + \rho [ENH_{STR,TOT} + ENH_{STR,STR} STR^*_{TOT}] = 0$ . Therefore,

(IV)  $STR^*_{TOT} = -(DEV_{STR,TOT} + \rho ENH_{STR,TOT}) / (DEV_{STR,STR} + \rho ENH_{STR,STR})$ .

We have argued that structure reduces the marginal cost impact of both volatility and complexity, that is:

(V)  $\partial (DEV_{VOL} + \rho ENH_{VOL}) / \partial STR < 0$  and

(VI)  $\partial (DEV_{TOT} + \rho ENH_{TOT}) / \partial STR < 0$ .

Then, from (III) and (IV), and using (II), (V), and (VI), we have:

$STR^*_{VOL} > 0$  and  $STR^*_{TOT} > 0$ .

## References

- Albrecht, A., J. Gaffney. 1983. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Trans. Software Eng.* SE-9(6) 639-648.
- Al-Janabi, A., E. Aspinwall. 1993. An evaluation of software design using the Demeter tool. *Software Eng. J.* 8(6) 319-324.
- Bækgaard, L. 1990. Designing adaptable software: Parameterization of volatile properties. Conference on Software Maintenance, San Diego, CA.

- Banker, R., S. Datar, C. Kemerer. 1991. A model to evaluate variables impacting productivity on software maintenance projects. *Management Sci.* 37(1) 1-88.
- , ———, D. Zweig. 1993. Software complexity and software maintenance costs. *Comm. ACM* 36(11) 81-94.
- , G. Davis, S. Slaughter. 1998. Software development practices, software complexity, and software maintenance performance: A field study. *Management Sci.* 44(4) 433-450.
- , C. Kemerer. 1992. Performance evaluation metrics for information systems development: A principal-agent model. *Inform. Systems Res.* 3(4) 379-400.
- , S. Slaughter. 1997. A field study of scale economies in software maintenance. *Management Sci.* 43(12) 1709-1725.
- Basili, V., B. Perricone. 1984. Software errors and complexity: An empirical investigation. *Comm. ACM* 27(1) 42-52.
- , R. Selby, T. Phillips. 1983. Metric analysis and data validation across Fortran projects. *IEEE Trans. Software Eng.* 9(7) 652-663.
- Belady, L., M. Lehman. 1976. A model of large program development. *IBM Systems J.* 15(3) 225-252.
- Belsley, D., E. Kuh, R. Welsch. 1980. *Regression Diagnostics*. John Wiley and Sons, New York.
- Benander, B., N. Gorla, A. Benander. 1990. An empirical study of the use of the goto statement. *J. Systems and Software* 11(3) 217-223.
- Bergantz, D., J. Hassell. 1991. Information relationships in prolog programs: How do programmers comprehend functionality. *Internat. J. Man-Machine Stud.* 35(3) 313-328.
- Bitman, W. 1997. Balancing software composition and inheritance to improve reusability, cost, and error rate. *Johns Hopkins APL Techn. Digest* 18(4) 485-500.
- Boehm, B. 1981. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ.
- Booch, G. 1991. *Object Oriented Design with Applications*. Benjamin Cummings, Redwood City, CA.
- Bowen, J. 1984. Module size: A standard or heuristic? *J. Systems and Software* 4 327-332.
- Butcher, G. 1997. Addressing software volatility in the system life cycle. Ph.D. dissertation, Colorado Technical University, UMI#9815557.
- Cant, S., D. Jeffery, B. Henderson-Sellers. 1995. A conceptual model of cognitive complexity of elements of the programming process. *Inform. Software Tech.* 37(7) 351-362.
- Card, D., R. Glass. 1990. *Measuring Software Design Quality*. Prentice Hall, Englewood Cliffs, NJ.
- Chapin, N., T. Lau. 1996. Effective size: An example of use from legacy systems. *J. Software Maintenance: Res. Practice* 8(2) 101-116.
- Chidamber, S., C. Kemerer. 1994. A metrics suite for object-oriented design. *IEEE Trans. on Software Eng.* 20(6) 476-493.
- Cohen, J., P. Cohen. 1983. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*, 2nd ed. Erlbaum, Hillsdale, NJ.
- Conte, S., H. Dunsmore, V. Shen. 1986. *Software Engineering Metrics and Models*. Benjamin-Cummings, Reading, MA.
- Davis, G., M. Olson. 1985. *Management Information Systems: Conceptual Foundations, Structure and Development*, 2nd ed. McGraw-Hill, New York.
- Dekleva, S., N. Zvegintzov. 1991. Real maintenance statistics. *Software Maintenance News* 9(2) 6-9.
- Gibson, V., J. Senn. 1989. System structure and software maintenance performance. *Comm. ACM* 32(3) 347-358.
- Glesjer, H. 1969. A new test for heteroscedasticity. *J. Amer. Statist. Assoc.* 64 316-323.
- Gode, D., A. Barua, T. Mukhopadhyay. 1990. On the economics of the software replacement problem. *Proc. 11th Internat. Conf. Inform. Systems*, Copenhagen, Denmark.
- Gorry, G., M. Scott-Morton. 1971. A framework for management information systems. *Sloan Management Rev.* 13(1) 55-70.
- Greene, W. 1997. *Econometric Analysis*, 3rd ed. Macmillan Publishing Company, New York.
- Gremillion, L. 1984. Determinants of program repair maintenance requirements. *Comm. ACM* 27(8) 826-832.
- Guinan, P., J. Coopridge, S. Sawyer. 1997. The effective use of automated application development tools. *IBM Systems J.* 36(1) 124-139.
- Hager, J. 1991. Software cost reduction methods in practice: A post-mortem analysis. *J. Systems and Software* 14(2) 67-79.
- Halstead, M. 1977. *Elements of Software Science*. Elsevier, North-Holland, NY.
- Harkness, W., W. Kettinger, A. Segars. 1996. Sustaining process improvement and innovation in the information services function: Lessons learned at the Bose corporation. *MIS Quart.* 20(3) 349-368.
- Head, R. 1967. Management information systems: A critical appraisal. *Datamation* 13(5) 23.
- Hogarth, R., B. Gibbs, C. McKenzie, M. Marquis. 1991. Learning from feedback: Exactingness and incentives. *J. Experiment. Psych.: Learning, Memory, and Cognition* 17 734-752.
- Honiden, S., N. Kotaka, Y. Kishimoto. 1993. Formalizing specification modeling in OOA. *IEEE Software* 10(1) 54-66.
- Hsia, P., A. Gupta, C. King, J. Peng, S. Liu. 1995. A study on the effect of architecture on maintainability of object-oriented systems. *Proc. Internat. Conf. Software Maintenance*, Opio, France, 4-11.
- IEEE Standard for Software Maintenance. 1993. IEEE, New York.
- Iman, R., W. Conover. 1979. The use of the rank transform in regression. *Technometrics* 21(4) 499-509.
- Jones, C. 1997a. Year 2000: What's the real cost? *Datamation* 43(3) 88-90, 92-93.
- . 1997b. *Software Quality: Analysis and Guidelines for Success*. International Thompson Computer Press, London, U.K.
- . 1991. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, New York.
- . 1989. Software enhancement modeling. *Software Maintenance: Res. Practice* 1 91-100.
- Kada, S., D. Woods, R. Cole. 1993. Design methods and code structure: A comparative case study. *Software Quality J.* 2(3) 163-176.
- Kahneman, D., A. Tversky. 1982. *Judgment Under Uncertainty: Heuristics and Biases*. Cambridge University Press, Cambridge, UK.

**BANKER AND SLAUGHTER**  
*The Moderating Effects of Structure*

---

- Kemerer, C. 1993. Reliability of function points measurement. *Comm. ACM* **36** 85–97.
- . 1995. Software complexity and software maintenance: A survey of empirical research. *Ann. Software Engrg.* **1** 1–22.
- , S. Slaughter. 1997. Determinants of software maintenance profiles: An empirical investigation. *Software Maintenance: Res. Practice* **9** 235–251.
- Kennedy, P. 1992. *A Guide to Econometrics*, 3rd ed. MIT Press, Cambridge, MA.
- Khoshgoftaar, T., J. Munson. 1990. Predicting software development errors using software complexity metrics. *IEEE J. Selected Areas in Comm.* **8**(2) 253–261.
- Krishnan, M. 1996. Cost and quality considerations in software product management, Ph.D. dissertation, GSIA, Carnegie Mellon University, Pittsburgh, PA.
- Laudon, K., J. Laudon. 1997. *Essentials of Management Information Systems: Organization and Technology*, 2nd ed. Prentice-Hall, Upper Saddle River, NJ.
- Li, H., W. Cheung. 1987. An empirical study of software metrics. *IEEE Trans. Software Eng.* **SE-13**(6) 697–708.
- Littlewood, B., L. Strigini. 1992. The risks of software. *Sci. Amer.* **267**(5) 62.
- Martin, J., C. McClure. 1983. *Software Maintenance: The Problem and Its Solutions*. Prentice-Hall, Englewood Cliffs, NJ.
- Mata-Toledo, R., D. Gustafson. 1992. A factor analysis of software complexity measures. *J. Systems and Software* **17** 267–273.
- McCabe, T. 1996. Cyclomatic complexity and the year 2000. *IEEE Software* **13**(3) 115–117.
- Nadler, D., E. Lawler, III. 1977. Motivation: A diagnostic approach. J. Hackman, E. Lawler, L. Porter, eds. *Perspectives on Behavior in Organizations*. McGraw-Hill, New York.
- Neter, J., W. Wasserman, M. Kutner. 1990. *Applied Linear Statistical Models*, 3rd ed. Irwin, Homewood, IL.
- Nosek, J., P. Palvia. 1990. Software maintenance management: Changes in the last decade. *J. Software Maintenance* **2**(3) 157–174.
- Page-Jones, M. 1980. *The Practical Guide to Structured Systems Design*. Yourdon Press, New York.
- Paich, M., J. Serman. 1993. Boom, bust, and failures to learn in experimental markets. *Management Sci.* **39**(12) 1439–1458.
- Parnas, D. 1972. On the criteria to be used in decomposing systems into modules. *Comm. ACM* **15**(12) 1053–1058.
- . 1979. Designing for ease of extension and contraction. *IEEE Trans. Software Eng.* **SE-5**(2) 129–137.
- Pfleeger, S. 1998. *Software Engineering: Theory and Practice*. Prentice-Hall, Upper Saddle River, NJ.
- Putnam, L., W. Myers. 1997. *Industrial Strength Software: Effective Management Using Measurement*. IEEE Computer Society Press, Los Alamos, CA.
- Rombach, H., 1987. A controlled experiment on the impact of software structure on maintainability. *IEEE Trans Software Eng.* **SE-13**(3) 344–354.
- Shapiro, S., M. Wilk. 1965. An analysis of variance test for normality. *Biometrika* **52** 591–612.
- Shen, V., T. Yu, S. Thebaut, L. Paulsen. 1985. Identifying error-prone software—An empirical study. *IEEE Trans. Software Eng.* **SE-11**(4) 317–323.
- Snyder, R. 1994. The role of local program information in software maintenance productivity. Ph.D. dissertation, Carlson School of Management, University of Minnesota, Minneapolis, MN.
- . 1998. The cost benefits of COBOL. *7*(8) 4–6.
- Sterman, J. 1989. Modeling managerial behavior: Misperceptions of feedback in a dynamic decision making environment. *Management Sci.* **35**(3) 321–339.
- Swanson, E., C. Beath. 1989. *Maintaining Information Systems in Organizations*. John Wiley and Sons, New York.
- Takahashi, R., Y. Nakamura. 1997. A software maintainability evaluation model based on re-use ratio and interface complexities between modified parts and unmodified parts. *Trans. Inst. Electr., Inform. Comm. Eng.* **J80D-I**(5) 441–449.
- Vessey, I., R. Weber. 1983. Some factors affecting program repair maintenance: An empirical study. *Comm. ACM* **26**(2) 128–134.
- Warnier, J. 1976. *Logical Construction of Programs*, 3rd ed. B. Flanagan, trans. Van Nostrand Reinhold, New York.
- White, H. 1980. A heteroscedasticity-consistent covariance matrix estimator and a direct test for heteroscedasticity. *Econometrica* **48** 817–838.
- Whitten, J., L. Bentley. 1997. *Systems Analysis and Design Methods*, 4th ed. Irwin/McGraw-Hill, Boston, MA.
- Wilde, N., P. Matthews, R. Huit. 1993. Maintaining object-oriented software. *IEEE Software* **10**(1) 75–80.
- Yourdon, E., L. Constantine. 1979. *Structured Design*. Prentice-Hall, Englewood Cliffs, NJ.

Cynthia Beath, Associate Editor. This paper was received on November 11, 1997, and was with the authors 7 months for 1 revision.