

# Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study

Rajiv D. Banker • Gordon B. Davis • Sandra A. Slaughter

School of Management, University of Texas at Dallas, Richardson, Texas 75083

Carlson School of Management, University of Minnesota, Minneapolis, Minnesota 55455

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

---

Software maintenance claims a large proportion of organizational resources. It is thought that many maintenance problems derive from inadequate software design and development practices. Poor design choices can result in complex software that is costly to support and difficult to change. However, it is difficult to assess the actual maintenance performance effects of software development practices because their impact is realized over the software life cycle. To estimate the impact of development activities in a more practical time frame, this research develops a two-stage model in which *software complexity* is a key intermediate variable that links design and development decisions to their downstream effects on software maintenance. The research analyzes data collected from a national mass merchandising retailer on 29 software enhancement projects and 23 software applications in a large IBM COBOL environment. Results indicate that the use of a code generator in development is associated with increased software complexity and software enhancement project effort. The use of packaged software is associated with decreased software complexity and software enhancement effort. These results suggest an important link between software development practices and maintenance performance.

(*Software Maintenance; Software Complexity; Software Productivity; Software Quality; Software Economics; Software Metrics; Management of Computing and Information Systems*)

---

## 1. Introduction

Although Edward Yourdon claims that "maintaining a computer program is one of life's dreariest jobs . . . for most American programmers, it is a fate worse than death" (in Zvegintzov 1988, p. 12), software maintenance is an important problem for organizations. Software maintenance is the modification of a software system after delivery to correct faults, improve performance, or adapt to a changed environment (ANSI / IEEE 1983). Over the last several decades, software maintenance has been claiming a large proportion of organizational resources, with expenditures often as high as 50–80 percent of the information systems (IS) budget (Nosek and Palvia 1990). On a life-cycle basis, more

than three-fourths of the investment in software occurs *after* the system has been implemented (Arthur 1988). Thus, there is considerable motivation to improve software maintenance performance.

Software maintenance is an activity that is difficult to perform and manage effectively (Swanson and Beath 1989). It is thought that many problems in software maintenance result from inadequate practices in software development (Schneidewind 1987). Software systems are the legacy of the tools, techniques, and people involved in their creation. Because systems have a lifetime often measured in decades (Zvegintzov 1984), long-term costs of supporting poor quality software can be substantial. Thus, actions to achieve improvements

must begin when the software is initially designed or when existing software is re-engineered. However, because the major benefits and penalties of development tools and techniques for software maintenance are realized over the software life cycle, it is difficult to assess their actual maintenance performance effects. Therefore, a critical barrier to improvement in software maintenance is the lack of knowledge concerning the implications of development practices for maintenance performance. This provides the motivation for the central issue examined by this study: the effect of software development practices on software maintenance performance.

This study posits that the impact of development practices on software maintenance can be assessed within a practical time frame by considering their effects on software complexity (Banker et al. 1989). Generally, software complexity refers to the characteristics of the data structures and procedures within the software that make it difficult to understand and change (Curtis et al. 1979). Software complexity is emerging as a critical software design quality factor (Card 1992, Card and Glass 1990). Furthermore, there is growing empirical evidence that software complexity is a major factor influencing maintenance effort (Kemerer 1995, Banker et al. 1993, Gibson and Senn 1989). However, there is little knowledge of the link between development practices and software complexity, and the subsequent implications for maintenance. This limits the extent to which managers can take proactive steps to improve software quality and reduce long-term maintenance costs.

The study develops an integrative model, using software complexity as an intermediate variable that links development decisions to their downstream effects on software maintenance performance. The model is estimated using data collected from a national mass merchandising retailer on 29 software enhancement projects and 23 software applications in a large IBM COBOL environment. Although businesses spend considerable amounts on software, empirical studies of software in commercial environments are rare, due to difficulties in obtaining primary data from organizations (Hale and Haworth 1988). By examining both software development and software maintenance in a commercial environment, this research provides unique insights into

key performance and design issues for the support of business software.

## 2. Theoretical Framework

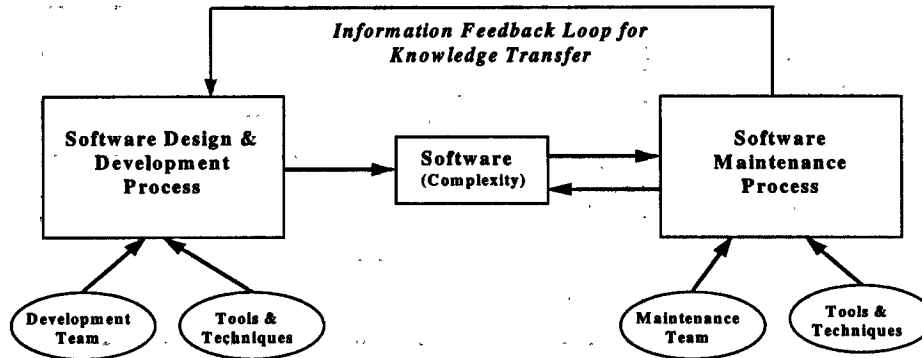
The theoretical framework for this study is developed by integrating several different perspectives. A production economics perspective enables specification of an integrative model for software development and maintenance. A psychological perspective forms the basis for conceptualizing software maintenance performance and motivates the choice of software complexity as a key variable influencing performance.

### 2.1. Economic View of Software Development and Maintenance

Building upon the conceptual foundations of Kriebel and Raviv (1980), Stabell (1982), and Banker et al. (1991), this study views software development and maintenance as economic production processes. In these processes, input factors including labor (the programming team) and capital (tools and techniques) are transformed into outputs such as new or modified software (Figure 1). A production economics perspective reveals two issues germane to this study. The first issue is the unidirectional flow of information between the development and maintenance processes. Although many software maintenance problems result from poor design choices, the traditional life-cycle model of software development discourages communication of feedback from maintenance into development (Schneidewind 1987). This lack of knowledge transfer has the result of locking organizations into suboptimal processes (Senge 1992) in which they support low quality software that cannot easily be modified.

Another issue highlighted by a production economics perspective is the difference in inputs to the software development and maintenance processes. While both development and maintenance employ people, tools, and techniques to produce software, a unique input to the maintenance process is the *software* that is created by the development process. This implies that the quality of the existing software plays an important role in software maintenance performance.

Figure 1 Software Development and Maintenance Production Processes



## 2.2. Psychological View of Software Maintenance and Software Complexity

While an economic perspective conceptualizes software maintenance as a production activity and enables insights into managerial concerns, a psychological perspective provides insights at the level of the individual maintenance task. Software maintenance can be conceptualized as a cognitive, human information-processing task in which input informational cues are interpreted and manipulated to create task outcomes (Ramanujan and Cooper 1994, Davis et al. 1993). According to this view, performance on cognitive tasks depends on effective processing of informational cues in the task environment (Simon 1981).

A key factor believed to influence cognitive task performance is complexity (Locke and Latham 1990, Campbell 1988, Wood et al. 1987, Simon 1981). Complexity is a phenomenon with multiple dimensions (Boulding 1970) in the form of component, coordinative, and dynamic complexity (Wood 1986). *Component complexity* refers to the number of distinct information cues that must be processed in the performance of a task, while *coordinative complexity* describes the form, strength, and interdependencies of the relationships between the information cues. *Dynamic complexity* arises from changes in the relationships between information cues over time, particularly during task performance: Because complexity obscures the perception and understanding of information cues, it is believed to significantly degrade task performance.

This psychological view of complexity is particularly appropriate for studying software maintenance. According to Brooks, software entities are more compli-

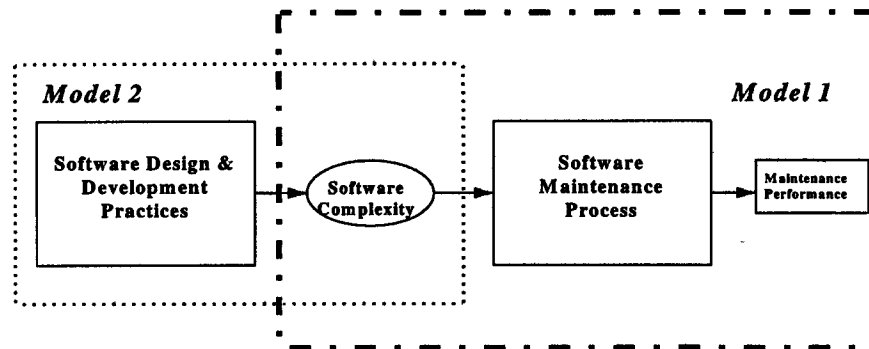
cated than any other human construct (1986, p. 11). Studies of software complexity suggest that software has multiple dimensions (Zuse 1990, Munson and Khoshgoftaar 1989, Weyuker 1988, Li and Cheung 1987). Furthermore, software maintainers are often required to modify software that is ill-documented, lacks comprehensible structure, and hides data representations (Guimaraes 1983). Thus, software maintenance becomes a largely cognitive task in which programmers perceive and manipulate relationships between informational cues presented by the existing software (Pennington 1987, Shneiderman 1980).

Empirical studies suggest that a significant portion of the software maintainer's time is required to understand the functionality of the software to be changed (e.g., Littman et al. 1987). A study of professional maintenance programmers by Fjeldstad and Hamlen (1983), for example, found that the programmers studied the original software code 3.5 times as long as they studied the supporting documentation, and equally as long as they spent implementing the enhancement. This suggests that software comprehension plays a major role in software maintenance performance. Another implication is that software complexity degrades maintenance performance because it interferes with understanding the functionality of the software. Thus, this study argues that software complexity is a key maintenance performance factor because it influences the critical activity of program comprehension.

## 2.3. Research Models and Hypotheses

The theoretical framework for this study is presented in Figure 2. The framework integrates two models in which

Figure 2 Theoretical Framework for Software Maintenance Performance



software complexity links software development practices to maintenance performance. The model for maintenance performance examines the effects of software complexity on maintenance project effort, controlling for factors such as project team experience, size of functionality modified, and other application and project factors. In this study, we focus on maintenance projects that adapt or enhance software functionality for existing systems because software enhancements represent the largest category of IS maintenance expenditures (Nosek and Palvia 1990). The model for application software complexity connects maintenance outcomes to software practices by assessing the impact of development tools and techniques on the complexity of the software application. A detailed explanation of each model follows.

### 2.3.1. Software Maintenance Performance Model.

The conceptual model for software maintenance performance is illustrated in Figure 3. *Software maintenance performance* is operationalized as software enhancement project effort. *Software enhancement* includes modifications to extend, change, and delete the functionality of existing software. *Project effort* refers to the time required by the maintenance project team to accomplish the software modifications for a particular end user request. In this model, software enhancement effort is specified as a function of software component complexity, software coordinative complexity, software dynamic complexity, project team experience, software functionality modified, application size, and application quality.

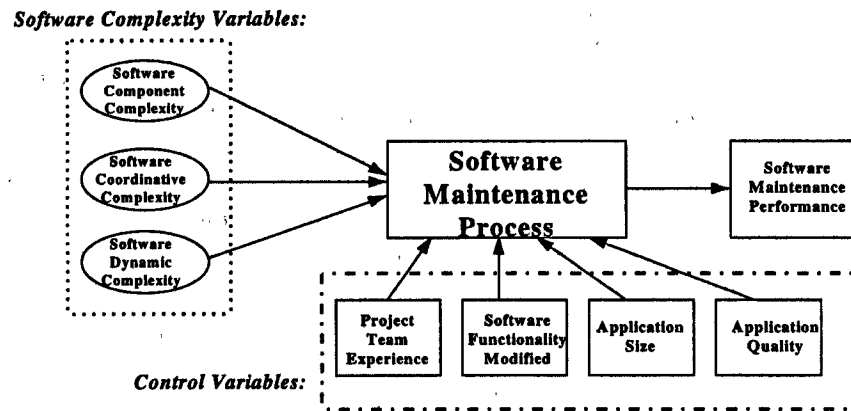
Dimensions of software complexity map well onto the dimensions of complexity identified by Wood (1986). As shown in Figure 3, software complexity is

operationalized in terms of its component, coordinative, and dynamic dimensions. *Software component complexity* is higher in software with "denser" code, where *data density* refers to the number of data elements embedded in each procedural line of code. A high level of data density increases the number of information cues the maintenance team must perceive and process in order to comprehend the software and to validate the modifications, and should increase effort required for the project. Prior empirical research generally supports the notion that programs with higher density are associated with increased maintenance effort (Shen et al. 1985, Gremillion 1984). Thus,

**HYPOTHESIS 1 (COMPONENT COMPLEXITY).** *Software enhancement project effort is positively related to data density for the software modified.*

*Software coordinative complexity* is high where there is excessive *decision density* within the software under modification. Frequent decision branching between modules obscures the relationship between program inputs and outputs and increases cognitive load on the maintenance team because they must search among dispersed pieces of code to determine logical flow. This is supported by several empirical studies that have found that maintenance effort increases as branching complexity increases (Boehm-Davis et al. 1992, Gill and Kemerer 1991). Thus, complex decision branching should have a detrimental effect on project effort because more time is required to follow the flow of logic within the code. This suggests the following hypothesis:

Figure 3 Model 1: Software Maintenance Performance



**HYPOTHESIS 2 (COORDINATIVE COMPLEXITY).** *Software enhancement project effort is positively related to decision density for the software modified.*

*Software dynamic complexity* is higher when there is increased instability of the input-output relationship for a task. An aspect of dynamic software complexity that is hypothesized to have a detrimental effect on maintenance performance is *decision volatility*. Decision volatility refers to the use of statements that enable modifications to control flow at execution time such as dynamically invoked subroutines. Decision volatility should increase maintenance effort because programmers must envision the multiple, potential control flow paths that could be followed and the procedures that could be invoked when the program executes. Effort required for testing and validation of modifications to dynamically complex software may be higher because programmers cannot easily anticipate the conditions that arise when the software executes, and may not be able to identify and validate all potential logic paths. This leads to the following hypothesis:

**HYPOTHESIS 3 (DYNAMIC COMPLEXITY).** *Software enhancement project effort is positively related to decision volatility for the software modified.*

There are several important aspects that are controlled in the software maintenance performance model. One aspect pertains to the characteristics of the project team. Prior empirical research (Vessey 1989, Oman et al. 1989) suggests that *project team experience* has a sig-

nificant influence on maintenance performance. Wood (1986) argues that experience plays an important role in task performance because it influences an individual's ability to process complex informational cues. Therefore, the study controls for project team experience and expects it to be inversely related to project effort. Also controlled is the amount of *software functionality modified*. Functionality is not usually under the control of the maintenance team, but rather is an inherent aspect of the user modification request. Larger changes to software functionality require additional activities in understanding, validation, and implementation. Thus, the amount of modified functionality should be positively related to project effort.

Other project factors that are controlled include *application size and quality*. Enhancements to larger applications require more effort due to difficulties in locating the code to be changed and in validating changes (Swanson and Beath 1989). The quality of the application software also influences maintenance performance. Applications of poor quality that have had large amounts of corrective modification are more difficult to change. A high amount of repair suggests that an application was poorly written or inadequately tested. In addition, successive changes tend to complicate the logic flows within the software (Guimaraes 1983).

**2.3.2. Software Complexity Model.** To link maintenance performance outcomes to software development practices, the study examines the intermediate effect of software complexity. Software component

complexity, software coordinative complexity, and software dynamic complexity are all specified as functions of software development practices (Figure 4). Two software practices are analyzed in this model: use of *code generators* and *packaged software*. Both code generators and packaged software are commonly used in software practice. Use of code generators is believed to contribute to improved software quality and reduced maintenance costs (Olle et al. 1983). Code generators require entry of a specified set of parameters and create similar kinds of software routines based upon these parameters (Davis and Olson 1985). Because code generators automate creation of voluminous amounts of standard procedural code to process a limited number of input parameters (Everest and Alanis 1992, Stamps 1987), use of these tools should reduce data density. This suggests that,

**HYPOTHESIS 4 (CODE GENERATORS).** *The use of code generators is inversely related to data density for the application software.*

Use of these tools should also introduce a consistent structure in the software and reduce decision density. Thus,

**HYPOTHESIS 5 (CODE GENERATORS).** *The use of code generators is inversely related to decision density for the application software.*

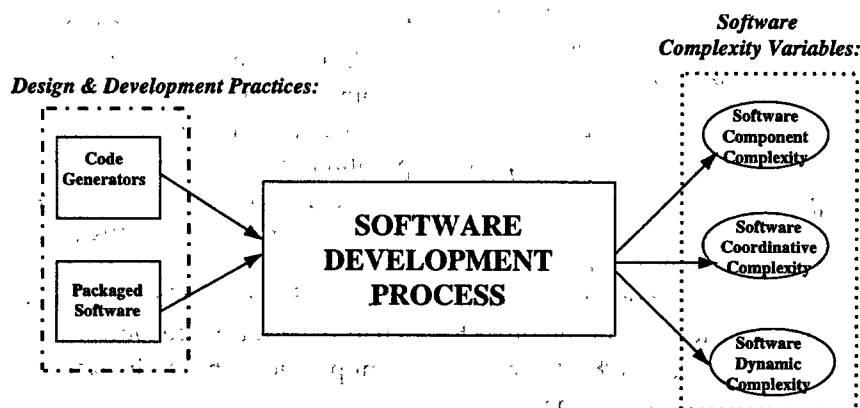
However, although code generators enable productivity gains in software development because of the automatic generation of code, such gains are offset in software maintenance by increased decision volatility. Pro-

grams generated by code generators include standard routines and custom software routines that execute conditionally depending on inputs entered by the end user of the system. This conditional invocation of custom code is very similar to the use of the "ALTER" or "GO TO DEPENDING ON" statements that enable dynamic modifications of program control flow at execution time. These routines are necessary to particularize the software to user requirements not met by the code generator's standard routines. The conditional execution of custom code based upon online screen input makes it difficult for maintenance programmers to discern the logical flow of information within the program because the programmers must envision dynamic characteristics of program inputs from an online environment, infer which routines will be called and executed, and deduce the subsequent effects on outputs. Thus, extensive use of custom routines contributes to increased decision volatility in the software created by the code generator because the routines complicate the standard control flow in the programs. This implies,

**HYPOTHESIS 6 (CODE GENERATORS).** *The use of code generators is positively related to decision volatility for the application software.*

Packaged software is created by vendors to satisfy the requirements of a large number of customers. Such software includes a larger volume of data and procedures that apply more generally to a number of organizations (Davis 1988, Martin and McClure 1983). Thus, this kind of software should have higher data and decision density than software developed in-house because it is not

Figure 4 Model 2: Software Complexity



restricted to the particular requirements of an organization (Lynch 1984). Thus,

**HYPOTHESIS 7 (PACKAGED SOFTWARE).** *The use of packaged software is positively related to data density for the application software.*

**HYPOTHESIS 8 (PACKAGED SOFTWARE).** *The use of packaged software is positively related to decision density for the application software.*

However, decision volatility in the form of invoked subroutines would be lower in packaged software. To facilitate incorporation of new releases from the vendor, organizations operate purchased software in isolation from their other applications, with the exception of necessary interfaces. As part of the vendor agreement, the organization may be prohibited from major customization of the purchased software code. Thus, packaged software tends to be less integrated into the application portfolio, with less invocation of organizational subroutines. This implies that,

**HYPOTHESIS 9 (PACKAGED SOFTWARE).** *The use of packaged software is inversely related to decision volatility for the application software.*

### 3. Research Design and Methodology

#### 3.1. Research Setting

The research site is the IS department of a national mass merchandising retailer (hereafter referred to as "Retailer"), located at company headquarters and supporting all centralized computer processing activities. In 1983, the IS department was divided into separate development and maintenance teams, with Development working exclusively on new systems, and Maintenance devoted to support and enhancement of existing systems. At the time of the study, Maintenance was responsible for the support of 23 major applications. An application at the Retailer includes a number of programs or modules (on average, about 200 programs) that together accomplish a major function such as accounts receivable, payroll, or order management. Maintenance supports these applications by correcting errors, doing preventative maintenance and conversions to accommodate changes in the technical environment,

answering user questions, and accomplishing user-requested enhancements to existing functionality.

The Retailer has a large investment in transaction-processing software written in COBOL and running on large IBM mainframe computers. The study of software maintenance in this kind of environment is important because a large portion of business expenditures for computing is still devoted to maintenance of COBOL software (Croxtton 1994). Approximately 65 percent of the Retailer's application portfolio at the time of the study was written in COBOL. Most of the application software was written in the 1970s and 1980s, with the oldest system written in 1974.

The Retailer employs two primary development practices: use of software packages and use of a code generator. Four of the 23 applications are software packages. The Retailer uses software packages for applications that are considered to be "generic," with relatively little unique, company-specific requirements. The packages are financial and accounting applications including payroll, general ledger, fixed asset accounting, and financial accounting. Agreements with the vendors for these packages do not prevent the Retailer from modifying the package software code. The Retailer has made minor modifications to the packages such as customizing the printing routines to accommodate the company name and other unique identifiers.

The Retailer uses a code generator for in-house development. The code generator is a mainframe software productivity tool widely available in the 1980s and 1990s that is intended to aid in the development of COBOL applications. It provides support for the design of online screens, for automatic generation of standard source code for input and output processing, and for interactive testing of the generated source code. The tool allows incorporation of custom source code within the generated code to accomplish more complex or unique processing that is not supported automatically, to modify the default program flow, and to override default field attributes. There was considerable interest in studying the impacts of the code generator and packages on Maintenance, because the Retailer relied heavily on these development practices:

All of our development work now is done using code generators or its packages. The trend is to scope down by buying it off the shelf or increase the number of people and tools to get

it done quicker. (Interview Transcript, Development Manager, February 8, 1993)

### 3.2. Data Collection Methods

Both quantitative and qualitative data were collected during the study. Multiple methods of data collection and multiple data sources were employed, including interviews, observation of meetings, project documentation, software archives, data archives, computer listings, internal company reports and memos, media information, and questionnaires. The triangulation made possible by multiple data collection methods and sources provides stronger substantiation of constructs and hypotheses, strengthening convergence of results (Eisenhardt 1989).

To improve the internal validity of the analysis, several criteria were used to select projects and applications for inclusion in the study: size, recency, change in functionality, and similarity of programming language. Larger projects (i.e., those requiring more than one work day to complete) were selected because factors affecting productivity on short, one-person projects can be overwhelmed by individual skill differences (Banker et al. 1991, DeMarco 1982, Curtis 1981). Project recency is important because personnel turnover and lack of documentation retention make accurate data collection impossible for older projects. The third criterion considered was whether the project modifies software functionality; this improves homogeneity of the projects analyzed, since projects such as package installations or conversions to new operating systems are excluded from the sample and are not compared with projects that modify functionality. The final criterion considered was whether the software was written in a similar programming language. All program code analyzed was written in COBOL. Therefore, the results are not confounded by the effects of multiple programming languages.

### 3.3. Construct Measurement, Reliability and Validity

The theoretical models for maintenance performance and software complexity are operationalized as follows. *Software enhancement project effort* is measured for a cross-sectional set of completed projects by counting the total number of labor hours logged to the projects. The study focuses on labor hours as the variable of interest,

since personnel time is the most expensive and scarce resource in software maintenance (Grammas and Klein 1985). A potential threat to validity of the effort measure is inaccurate project records. The Retailer's internal clients were charged by the hours spent on their projects. This motivates personnel to maintain accurate project time records. In addition, the project data did not reveal any systematic bias or unusual observations that could not be explained by logical causes when various subsets of the data were examined.

*Software development practices* include the use of a code generator and packaged software. The use of a code generator is measured as a ratio of the number of modules developed using such tools divided by the total number of modules in the application. Packaged software is represented with a binary variable to indicate whether the software was purchased or developed in-house. To increase the validity of the development practices measures, two sources of information were used as a cross-check: the software code itself and a questionnaire completed by application managers.

*Software complexity* is assessed in terms of its component, coordinative, and dynamic dimensions. *Data density* is measured by the number of data elements referenced in the software, using Halstead's N2 software science metric (Halstead 1977). This metric derives from a count of operands (data variables) in a program. *Decision density* is measured using McCabe's cyclomatic complexity (McCabe 1976). McCabe's metric counts the number of decision paths in the software. *Decision volatility* refers to the number of decision paths that are dynamically altered at software execution time. It is estimated by counting the number of dynamically executed subroutines invoked by the "CALL" command. To control for program size, the complexity variables are normalized by dividing each by the total lines of code in the programs modified by the project. For the application model, the complexity variables are normalized by dividing each by the total lines of code in the application.

Confirmatory factor analysis utilizing varimax rotation (Johnson and Wichern 1992) and the Kaiser-Meyer-Olkin index (1974) were employed to verify that the complexity constructs loaded on orthogonal factors. While there is significant pairwise correlation between the data density and decision density measures, the

software complexity variables cannot be adequately represented by a reduced number of factors. This supports the discriminant validity of the software complexity measures. Informal support for convergent validity of the empirical complexity constructs was obtained from interviews with maintenance personnel.

*Software functionality modified* is assessed by the number of function points added, changed, or deleted by the project. *Application size* is measured by counting the number of application function points. Function points measure the number of unique inputs, outputs, logical files, external interface files, and external queries included in a software application (IFPUG 1993). The counts are weighted for difficulty depending upon factors such as whether the application runs in a distributed environment, or whether the application is held to above average performance standards (Albrecht and Gaffney 1983). Function points is a widely used measure of software size (Perry 1986) and is considered a reliable measure of delivered software superior to the alternative measure of software lines of code (Kemerer 1993). For this study, function point counts were obtained from the Retailer's Quality Assurance team, which is independent of Development and Maintenance, but reports directly to the Chief Information Officer.

*Application quality* is measured by counting the number of repair hours for the application. Repair hours were obtained from company time-tracking systems and cross-checked against a parallel, paper-based system that reported repair, user support, and preventive maintenance hours for the applications.

*Project team experience* was subjectively assessed by maintenance managers in response to a project data questionnaire. Experience was measured on a five-point scale corresponding to the percentage of programmers on the project team who had worked with the application for three or more years. The scale was derived from an existing instrument used in prior empirical work (Banker et al. 1987, 1991).

## 4. Data Analysis and Results

### 4.1. Interviews

Interviews with IS personnel provide support for central ideas motivating this study: software development

and maintenance are linked, design decisions have long-term maintenance effects that are difficult to reverse, and there is little knowledge of the effects of design choices:

When Maintenance gets systems from Development, there are often many outstanding problems. Maintenance has to still work on things to get the system to work. If there are problems, we call Development, but Development's "stuck" and "has no time to work on it," so they can't often offer much assistance. Really complex programs often come from Development. On-line screens, for example, often do two or three functions in one screen . . . and Maintenance has to rewrite the screen into two programs. Certain people in Development do that constantly. The manager wanted to get it done and approved it. (Interview Transcript, Maintenance Programmer, February 1, 1993)

When we turn over systems to Maintenance, we don't get to see whether we made a good design choice. Usually there's two or three different ways to do things. But most people in Development haven't had Maintenance experience so they don't know which is better for Maintenance. (Interview Transcript, February 15, 1993, Development Programmer)

Although many development and maintenance programmers occupied adjacent cubicles, there appeared to be little communication between the groups. In fact, there appeared to be some friction, especially when new systems were transferred from Development to Maintenance:

There should be an evaluation of those who created the system in the first place. Downtime and abends (system failures) are important. But they're not given to Development so they can learn what was done. There's no interest by Development once the system's been turned over. There's a book of turnover standards, but the last sentence is like . . . "this can be waived at the discretion of the manager accepting the system." And they pressure the manager to accept the system . . . it's political. (Interview Transcript, Maintenance Programmer, March 1, 1993)

In one situation, an entire new application was rewritten over a period of three years by a maintenance team dedicated to redeveloping the code. A maintenance programmer described the interdependencies between Development and Maintenance:

You know, data processing's more mechanical now, like an assembly-line. Everything's so interrelated and there's ripple effects everywhere. We're machinists, not artists . . . we need to have a good definition of where we want to go with (software) quality so Development and Maintenance can work

together. (Interview Transcript, Maintenance Programmer, February 1, 1993)

Interview comments suggest the importance of understanding in more detail the link between development choices and maintenance effects. In the following sections, the analysis of quantitative data on software enhancement projects, software applications, and design practices provides insights into the precise nature of the relationship between development practices and maintenance performance at the research site.

#### 4.2. Model 1: Software Maintenance Performance

The effect of software complexity on enhancement project effort is analyzed using project-level data to estimate a multiple regression model that relates software enhancement project effort, software complexity, software functionality, and other project characteristics:

Project Hours

$$\begin{aligned}
 &= \beta_0 + \beta_1(\text{Data Density}) \\
 &+ \beta_2(\text{Decision Density}) + \beta_3(\text{Decision Volatility}) \\
 &+ \beta_4(\text{Project Function Points}) \\
 &+ \beta_5(\text{Team Experience}) \\
 &+ \beta_6(\text{Application Repair Hours}) \\
 &+ \beta_7(\text{Application Function Points}) + \epsilon
 \end{aligned}$$

After discussions with IS staff concerning accurate project record retention, only projects completed within a three-year time frame between January 1991 and December 1993 were considered for inclusion in the study. Of the 40 large projects completed during that time frame, 11 were eliminated because they were either non-

enhancement projects or non-COBOL projects. Twenty-nine software enhancement projects were thus selected for analysis. Table 1 presents a statistical profile of the projects. The average software enhancement project required 453 hours to complete, and added, changed, or deleted 143 function points. A correlation matrix for the variables appears in Table 2.

A variety of specification checks were performed for the estimated model to ensure that standard assumptions were satisfied. The Shapiro-Wilk test for normality of residuals in small samples (Shapiro and Wilk 1965) rejected the normality assumption at the 5 percent significance level. A logarithmic transformation of the dependent variable was indicated by the Box-Cox procedure (Box and Cox 1964) to correct for skewness. Specification checks were conducted after transforming to a semi-log model. The Shapiro-Wilk test indicated that the normality assumption was not violated. Both Glejser's (1969) and White's (1980) tests suggested that the homoscedasticity assumption was not violated for the revised model. Belsley-Kuh-Welsch (1980) collinearity diagnostics indicated a highest condition number for the model of 19.68, within the recommended limit (Greene 1993). Variance inflation factors for the independent variables were all below 10, also suggesting that multicollinearity is not a problem (Neter et al. 1990).

Statistical results from one-tailed tests of the hypotheses are presented in Table 3. The results support the hypotheses that enhancement effort is positively related to decision density and decision volatility. As expected, enhancement effort is positively related to project size, application repair hours, and application size, and is inversely related to team experience. The joint effect of

Table 1 Statistical Profile of Software Enhancement Projects

Variable	Mean	Std. Dev.	Min	Max	Q1	Q2	Q3
Project Hours	453.0	541.6	48.0	2,450.6	92.0	265.0	573.5
Project Function Points	143	210	8	1,052	13	86	217
Application Function Points	3,105	1,475	1,319	5,355	1,847	2,640	4,876
Application Experience	3.0345	1.5232	1.0000	5.0000	2.0000	3.0000	5.0000
Repair Hours	989.1	698.8	35.75	2,828.5	433.0	851.0	1,329.0
Data Density	0.7624	0.2064	0.1228	1.1994	0.6559	0.7502	0.8734
Decision Density	0.0536	0.0242	0.0100	0.1190	0.0399	0.0472	0.0706
Decision Volatility	0.0109	0.0059	0.0000	0.0212	0.0061	0.0125	0.0149

**Table 2** Correlation Matrix for Software Enhancement Projects

	Project Hours	Project Function Points	Application Function Points	Application Experience	Repair Hours	Data Density	Decision Density	Decision Volatility
Project Hours	1.0000							
Project Function Points	0.7550*	1.0000						
Application Function Points	0.0882	0.1057	1.0000					
Application Experience	-0.0969	0.1870	-0.4778*	1.0000				
Repair Hours	0.3011	0.0056	-0.0699	-0.2765	1.0000			
Data Density	-0.0222	-0.0527	-0.0246	0.0935	0.4046*	1.0000		
Decision Density	0.1377	0.0684	-0.0503	0.2412	0.2338	0.7188*	1.0000	
Decision Volatility	0.0763	0.1671	-0.1500	0.3183	-0.1795	-0.1518	-0.0997	1.0000

\* Indicates that correlation is statistically significant at the 5% level.

the three software complexity variables on enhancement effort is also significant.

The coefficient for data density is significantly negative, contrary to Hypothesis 1. We examined several alternative explanations for this finding. One explanation

is the high correlation between the data and decision density variables. To check this explanation, the model was estimated without the decision density variable. Neither the sign nor significance of the data density variable changed, suggesting that collinearity with the

**Table 3** Regression Results for Software Enhancement Model (*t* statistics in Parentheses)

Variables	Coefficient (predicted sign)	Basic Model ( <i>n</i> = 29)	Standardized Model ( <i>n</i> = 29)	Without Outliers ( <i>n</i> = 27)	Rank Regression ( <i>n</i> = 29)
Intercept	$\beta_0$	5.1992* (6.984)	0.0000	5.1730* (8.518)	2.5945 (0.690)
Data Density	$\beta_1$ (+)	-2.7406* (-2.964)	-0.5161*	-2.0156* (-2.619)	-0.2017** (-1.410)
Decision Density	$\beta_2$ (+)	24.0894* (3.176)	0.5321*	29.6446* (3.989)	0.5454* (3.856)
Decision Volatility	$\beta_3$ (+)	49.1781* (2.179)	0.2636*	37.7802* (2.069)	0.0850 (0.909)
Project Function Points	$\beta_4$ (+)	0.0032* (5.156)	0.6188*	0.0054* (5.647)	0.7353* (7.361)
Application Experience	$\beta_5$ (-)	-0.1499** (-1.327)	-0.2083**	-0.2199* (-2.232)	-0.3626* (-2.841)
Repair Hours	$\beta_6$ (+)	0.0005* (2.101)	0.2889*	0.00002 (0.079)	0.0053 (0.056)
Application Function Points	$\beta_7$ (+)	0.0001 (0.480)	0.0659	-0.00002 (-0.201)	0.0202 (0.192)
$R^2$		0.7395	0.7395	0.8257	0.8535
Adjusted $R^2$		0.6526	0.6526	0.7615	0.8047
<i>F</i> test (Model)	$\beta_i = 0$ for all <i>i</i>	8.515*	8.515*	12.861*	17.4822*
<i>F</i> test (Complexity)	$\beta_1 = \beta_2 = \beta_3 = 0$	5.248*	5.248*	6.354*	6.4082*

\* Indicates significance at 5% level.

\*\* Indicates significance at 10% level.

decision density variable is not causing a shift in sign. Another alternative is that data density is correlated with an omitted variable. Data density was then regressed on several additional variables (including application lines of code, project team skill, project team IS experience, and total lines of code modified in the project). The regression was not significant, indicating that data density was not related to any measured variable omitted from the model. A third alternative is that there is a causality problem in the model; however, the time precedence in the data should reduce this possibility.

A potential explanation for the result that data density is associated with less project effort is that programs high in data density contain lower functional complexity. Programs high in data density are in applications representing lower levels of information processing (Davis and Olson 1985, p. 7). The applications with the highest data density include the accounting and basic financial transaction-processing systems such as accounts receivable, general ledger, accounts payable, and payroll, as well as interface systems. Such applications process volumes of data in a relatively simple manner and are less functionally sophisticated than systems like merchandise forecasting and planning, and may thus be more easily modified. An interesting direction for further research would be to examine the relationship between the levels of information processing in the applications and the effort required for maintaining them.

To assess the robustness of the estimated parameters, a number of sensitivity analyses were conducted. Belsley-Kuh-Welsch (1980) procedures led to the omission of two influential observations from the regression. The sign and significance of the re-estimated software complexity coefficients after omitting the influential observations correspond to those in the original model (Table 3). An additional sensitivity check involved a rank regression to assess the robustness of results to functional forms (Iman and Conover 1979). The sign and significance of the complexity coefficients in the rank regression correspond to those in the original model with the exception of the decision volatility variable whose coefficient was not significant (Table 3).

### 4.3. Model 2: Software Complexity

The effect of software development practices on software complexity is assessed using application-level data to estimate models relating software complexity attributes with measures of development practices:

Data Density

$$= \gamma_{01} + \gamma_{11}(\text{Code Gen}) + \gamma_{21}(\text{Package}) + \epsilon_1,$$

Decision Density

$$= \gamma_{02} + \gamma_{12}(\text{Code Gen}) + \gamma_{22}(\text{Package}) + \epsilon_2,$$

Decision Volatility

$$= \gamma_{03} + \gamma_{13}(\text{Code Gen}) + \gamma_{23}(\text{Package}) + \epsilon_3.$$

Because the equations have identical explanatory variables, ordinary least squares and generalized least squares estimates are identical.

Table 4 presents a statistical profile of the 23 COBOL applications included in the study. The average application included 401,434 lines of code and 2,368 function points. The total size of the portfolio analyzed was 9,232,973 lines of code and 54,470 function points. This represents about 65 percent of the company's total application portfolio. A correlation matrix for key variables appears in Table 5.

Normality and homoscedasticity assumptions were satisfied for all three equations. The Box-Cox procedure indicated that no transformations were necessary. Pearson partial correlations, condition numbers for the models (highest of 3.76), and variance inflation factors (all less than 10) suggested that multicollinearity is not unduly influencing the estimates. Statistical results (Tables 6a through 6c) support the hypotheses that packaged software is associated with increased data and decision density and with decreased decision volatility. Also supported are the hypotheses that use of the code generator is associated with reduced data and decision density and with increased decision volatility.

Sensitivity analyses included re-estimating the decision density equation after deleting two influential observations identified using the Belsley-Kuh-Welsch (1980) criteria (there were no influential outliers in the data density and decision volatility models). The sign and significance of the development practices

**Table 4** Statistical Profile of Software Applications

Variable	Mean	Std Dev	Min	Max	Q1	Q2	Q3
Total Lines of Code	401,434	430,462	3,636	1,528,670	138,069	255,999	512,037
Proportion of Code Generator	0.2276	0.2521	0.0000	0.7351	0.0000	0.1021	0.4000
Application Function Points	2,368	1,693	273	5,482	814	1,847	3,704
Package	0.174	0.388	0.00	1.000	0.000	0.000	0.000
Data Density	0.7255	0.1191	0.5555	1.0050	0.6186	0.7122	0.8024
Decision Density	0.0515	0.0118	0.0357	0.0850	0.0441	0.0489	0.0579
Decision Volatility	0.0083	0.0049	0.0008	0.0160	0.0042	0.0078	0.0130

**Table 5** Correlation Matrix for Software Applications

	Total Lines of Code	Application Function Points	Proportion of Code Generator	Package	Data Density	Decision Density	Decision Volatility
Total Lines of Code	1.0000						
Application Function Points	0.6132*	1.0000					
Proportion of Code Generator	0.7069*	0.2541	1.0000				
Package	-0.1599	0.1660	-0.4025	1.0000			
Data Density	-0.4179*	-0.2698	-0.6086*	0.5207*	1.0000		
Decision Density	-0.2991	-0.1848	-0.6103*	0.6192*	0.8133*	1.0000	
Decision Volatility	0.5412*	0.2466	0.8216*	-0.3864	-0.3783	-0.4733*	1.0000

\* Indicates that correlation is statistically significant at the 5% level.

variables in the revised decision density regression correspond to those in the original model. Estimation of rank regression models also confirmed that the sign and significance of the development practices variables in the original models are robust with the exception of the package variable, which is no longer significant at the 5 percent level in the decision density model.

## 5. Discussion

To examine the implications of software development practices for software maintenance performance, this study developed an integrative model with software complexity as an intermediate variable linking design decisions to their downstream maintenance impacts. This model was empirically tested using data collected from a national mass merchandising retailer. The detailed results provide a number of interesting insights and suggest several conclusions.

The study examined the maintenance impact of two key development practices at the research site: use of a code generator and packaged software. Software enhancement effort was significantly related to software complexity and was positively associated with the size of project functionality, application size, and application repair hours, and inversely associated with project team experience.

The use of a code generator was significantly related to reduced data and decision density and increased decision volatility. The overall impact of the code generator on software enhancement effort is assessed using the estimated coefficients for the code generator variable, the individual software complexity variables, and the enhancement effort variable (Table 7). Results indicate that, *ceteris paribus*, an increase of 10 percent in the proportion of code generator use is associated with a 3.8 percent *increase* in software enhancement project hours. This was a surprising result to software managers at the Retailer, but not to the maintenance

**BANKER, DAVIS, AND SLAUGHTER**  
*Software Development Practices, Software Complexity, and Maintenance Performance*

**Table 6a** Regression Results for Data Density Model (*t* Statistics in Parentheses)

Variables	Coefficient (Predicted Sign)	Basic Model ( <i>n</i> = 23)	Standardized Model ( <i>n</i> = 23)	Without Outliers ( <i>n</i> = 23)	Rank Regression ( <i>n</i> = 23)
Intercept	$\gamma_{01}$	0.7882* (21.028)	0.0000	n/a	14.9095* (3.346)
Proportion of Code Generator	$\gamma_{11}$ (-)	-0.1228* (-2.887)	-0.5023* (-2.887)	n/a	-0.06175* (-3.641)
Package	$\gamma_{21}$ (+)	0.1002* (1.874)	0.3261* (1.874)	n/a	0.3751** (1.484)
$R^2$	-	0.4855	0.4855	n/a	0.5605
Adjusted $R^2$	-	0.4341	0.4341	n/a	0.5166
F test (Model)	$\gamma_{11} = \gamma_{21} = 0$	9.437*	9.437*	n/a	12.753*

**Table 6b** Regression Results for Decision Density Model (*t* Statistics in Parentheses)

Variables	Coefficient (predicted sign)	Basic Model ( <i>n</i> = 23)	Standardized Model ( <i>n</i> = 23)	Without Outliers ( <i>n</i> = 21)	Rank Regression ( <i>n</i> = 23)
Intercept	$\gamma_{02}$	0.0573* (17.518)	0.0000	0.0579* (25.005)	18.143* (4.419)
Proportion of Code Generator	$\gamma_{12}$ (-)	-0.0124* (-3.326)	-0.5083* (-3.326)	-0.0131* (-4.938)	-0.7315* (-4.68)
Package	$\gamma_{22}$ (+)	0.0129* (2.763)	0.4223* (2.763)	0.0158* (3.868)	0.2196 (.943)
$R^2$	-	0.6030	0.6030	0.7201	0.6269
Adjusted $R^2$	-	0.5633	0.5633	0.6890	0.5896
F test (Model)	$\gamma_{12} = \gamma_{22} = 0$	15.191*	15.191*	23.158*	16.801*

**Table 6c** Regression Results for Decision Volatility Model (*t* Statistics in Parentheses)

Variables	Coefficient (predicted sign)	Basic Model ( <i>n</i> = 23)	Standardized Model ( <i>n</i> = 23)	Without Outliers ( <i>n</i> = 23)	Rank Regression ( <i>n</i> = 23)
Intercept	$\gamma_{03}$	0.00493* (3.001)	0.0000	n/a	3.0180 (0.735)
Proportion of Code Generator	$\gamma_{13}$ (+)	0.00573* (3.075)	0.5669* (3.075)	n/a	0.7944* (5.084)
Package	$\gamma_{23}$ (-)	-0.00212 (-.905)	-0.1668 (-.905)	n/a	-0.0459 (-0.197)
$R^2$	-	0.4225	0.4225	n/a	0.6269
Adjusted $R^2$	-	0.3647	0.3647	n/a	0.5896
F test (Model)	$\gamma_{13} = \gamma_{23} = 0$	7.315*	7.315*	n/a	16.800*

\* Indicates significance at 5% level.

\*\* Indicates significance at 10% level.

**Table 7** Calculation of Effects of Development Practices

<i>i</i>	Complexity	In Project Hours ( $\beta_i$ )	CodeGen ( $\gamma_{1i}$ )	Package ( $\gamma_{2i}$ )
Estimated Parameter Coefficients from Tables 3 and 6				
1	Data Density	-2.7406	-0.1228	0.1002
2	Decision Density	24.0894	-0.0124	0.0129
3	Decision Volatility	49.1781	0.0057	-0.0021
<i>i</i>	Partial Effects of the Following:		CodeGen ( $\beta_i * \gamma_{1i}$ )	Package ( $\beta_i * \gamma_{2i}$ )
Impact on Enhancement Project Effort				
1	Data Density		0.33654	-0.27460
2	Decision Density		-0.29870	0.31075
3	Decision Volatility		0.28179	-0.10327
	Total [ $=\sum \beta_i \gamma_{ki}$ ]:		0.31963	-0.06712
	% Change [ $= (\exp^{\sum \beta_i \gamma_{ki}} - 1) * 100$ ]:		37.66183%	-6.49170%

programmers. Although the code generator creates standardized routines that reduce data and decision density, conditionally invoked routines that alter the standard flow of program logic were embedded in the code to customize it to user requirements, increasing decision volatility and leading to lower software maintenance performance. Maintenance programmers indicated that it was difficult to work with the code generator and to make modifications because of the mix of generated and custom code. In many cases, the code no longer matched the original application design in the tool because the programmers had changed the generated code but not the design:

Tools like (the code generator) solve old problems but create new ones. They just shift the problems to Maintenance. The problem with (the code generator) is that most changes are to the user (customized) logic. Then it's hard to figure out the logic paths in that stuff because it calls in routines from all over. And then the compiles take longer, and the testing and debugging are more complicated. At first (the code generator) only had a screen painter, so you had to edit the program in Panvalet, and there was no validation except when you compiled it! Also, navigating through the tool is difficult. There are layers and layers of menus, and you just want to make a change to one or two lines of code. (Interview Transcript, Maintenance Programmer, March 1, 1993)

The use of purchased software was significantly associated with increased data and decision density, and

inversely related to decision volatility. Overall, an increase in the use of packaged software is associated with a 6.5 percent *decrease* in software enhancement project hours (Table 7). This result was surprising at first to both the software managers and the maintenance programmers. As one programmer stated,

Most of our packages have really junk code . . . you know, written in the most complicated way. I don't think the vendors care. It's so much better if they're written simply. (Interview Transcript, Maintenance Programmer, February 22, 1993)

Further investigation revealed that the accounting and financial packages used by the Retailer are data and decision intensive, reflected in higher data and decision complexity. However, the packages are low in decision volatility because they are less integrated into the application portfolio and have fewer invoked subroutines. In addition, modifications to these packages tend to be relatively simple, involving customization of printing routines and internal table values to accommodate the Retailer's unique codes, and thus require less maintenance effort.

## 6. Concluding Remarks

This study provides insight into how performance in software maintenance can be improved. From a continuous quality improvement perspective, the key idea is

that software development practices have long-term consequences that are difficult and costly to reverse, and therefore, to improve the software process, innovations must focus on improving the efficacy of design and development procedures. The software industry has been plagued by the "silver bullet syndrome"—the belief that new tools and techniques will solve software problems, without investing the time and effort needed to gain an understanding of the cause and effect relationships relating to the problems (Fenton et al. 1994). Without this understanding, each new innovation, such as structured design, CASE tools, and object-oriented programming, is adopted as the latest "silver bullet" and subsequently rejected as expectations are lowered, and original problems persist, perhaps accompanied by new problems. To that end, a critical element of this study has involved collecting evidence on the maintenance performance effects of development practices to foster learning about the software process (Garvin 1993, Grady 1993).

In addition, software process innovations are primarily directed toward improvement of development productivity, when the majority of software life cycle costs are post-implementation (Swanson and Beath 1989). Results from this study suggest that development productivity tools do not have similar benefits for maintenance, and that software maintenance concerns should play a larger role in the adoption of design and development practices (Banker and Kemerer 1992). The IS function is under increasing pressure to demonstrate productive performance and to contribute to the firm's business objectives. Such pressures work against a long-term perspective, discouraging the investments necessary to measure, evaluate, and learn, and to make continuous software process improvements. These sentiments are eloquently summarized by a development programmer at the research site in response to a question about software quality:

Yes, I think it's important, but you need to prove it. Look at Japan—they're innovating. They're taking the process and improving it. The auto industry is a good example. They taught us how important it was. You can attach a dollar figure to this, although the actual payback is kind of fuzzy . . . for example, if a 10 percent design fix upfront will save 90 percent rework later. But people want immediate results. In IS, the standards keep going up. It's a frustrated department at every level. You've got management. They always feel like they're the bad

guy. Q.A. (Quality Assurance) has no upper level management support. Programmers think management is out of touch—they just do the best they can. You've got programmers making decisions on quality versus time. Everyone knows we need to improve quality. It's not finger pointing—it should be forward-direction pointing. [Interview Transcript, Development Programmer, February 8, 1993]<sup>1</sup>

<sup>1</sup> The authors thank the departmental editor, the associate editor, and three anonymous reviewers for their valuable comments and suggestions. We gratefully acknowledge research support from the Quality Leadership Center at the University of Minnesota. The cooperation and assistance of managers and staff at our data site was invaluable. Helpful comments were provided by participants in research seminars at the University of British Columbia, Carnegie Mellon University, Georgia State University, University of Minnesota, University of Pittsburgh, and Stanford University.

## References

- Albrecht, A. J. and J. Gaffney, "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Trans. on Software Engineering*, SE-9, 6 (1983), 639-648.
- ANSI/IEEE Standard 729, *An American National Standard IEEE Standard Glossary of Software Engineering Terminology*, 1983.
- Arthur, L. J., *Software Evolution*, John Wiley & Sons, Inc., New York, 1988.
- Banker, R. D., S. Datar, and C. F. Kemerer, "Factors Affecting Software Maintenance Productivity: An Explanatory Study," *Proc. Eighth International Conf. on Information Systems*, ACM, New York, 1987, 160-175.
- , —, and —, "A Model to Evaluate Variables Impacting the Productivity of Software Maintenance Projects," *Management Sci.*, 37, 1 (1991), 1-18.
- , —, —, and D. Zweig, "Software Complexity and Software Maintenance Costs," *Comm. ACM*, November, 36, 11 (1993), 81-94.
- , —, and D. Zweig, "Software Complexity and Maintainability," *Proc. Tenth International Conf. on Information Systems*, ACM, New York, 1989, 247-255.
- and C. F. Kemerer, "Performance Evaluation Metrics for Information Systems Development: A Principal-Agent Model," *Information Systems Res.*, 3, 4 (1992), 379-400.
- Belsley, D. A., E. Kuh, and R. E. Welsch, *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*, Wiley, New York, 1980.
- Boehm-Davis, D. A., R. W. Holt, and A. C. Schulz, "The Role of Program Structure in Software Maintenance," *International J. Man-Machine Studies*, 36, 1 (1992), 21-63.
- Boulding, K. E., *Economics as a Science*, McGraw-Hill, New York, 1970.
- Box, G. E. and D. R. Cox, "An Analysis of Transformations," *J. Royal Statistical Society B*, 26 (1964), 211-243.
- Brooks, F. P., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, April (1986), 10-19.

- Campbell, D. J., "Task Complexity: A Review and Analysis," *Acad. Management Rev.*, 13, 1 (1988), 40-52.
- Card, D. N., "Designing Software for Producibility," *J. Systems and Software*, 17 (1992), 219-225.
- and R. L. Glass, *Measuring Software Design Quality*, Prentice Hall, Englewood Cliffs, NJ, 1990.
- Croxton, M., "Maintenance Still on IT Resume," *Software Magazine*, 14, 6 (1994), 4-6.
- Curtis, B., "Substantiating Programmer Variability," *IEEE Proceedings*, 69, 7 (1981), 846.
- , S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," *IEEE Trans. on Software Engineering*, SE-5, 2 (1979), 96-104.
- Davis, G., "To Buy, Build, or Customize?" *Accounting Horizons*, March (1988), 101-103.
- , R. Collins, M. Eierman, and W. Nance, "Productivity from Information Technology Investment in Knowledge Work," in R. D. Banker, R. J. Kauffman, and M. A. Mahmood (Eds.), *Strategic Information Technology Management: Perspectives on Organizational Growth and Competitive Advantage*, Idea Group Publishing, Harrisburg, PA, 1993, 327-345.
- and M. Olson, *Management Information Systems: Conceptual Foundations, Structure, and Development*, McGraw-Hill, New York, 1985.
- DeMarco, T., *Controlling Software Projects*, Yourdon Press, New York, 1982.
- Eisenhardt, K. M., "Building Theories from Case Study Research," *Acad. Management Rev.*, 14, 4 (1989), 532-550.
- Everest, G. and M. Alanis, "Assessing User Experience with CASE Tools: An Exploratory Analysis," *Proc. 25th Hawaii International Conf. on Systems Science*, IEEE Computer Society Press, Los Alamitos, CA, January 1992.
- Fenton, N., S. L. Pfleeger, and R. L. Glass, "Science and Substance: A Challenge to Software Engineers," *IEEE Software*, 11, 4 (1994), 86-95.
- Fjeldstad, R. K. and W. T. Hamlen, "Application Program Maintenance Study—Report to Our Respondents," in G. Parikh and H. Zvegintzov (Eds.), *Tutorial on Software Maintenance*, IEEE Computer Society Press, Silver Spring, MD, 1983.
- Garvin, D. A., "Building a Learning Organization," *Harvard Business Rev.*, July-August (1993), 78-91.
- Gibson, V. R. and J. A. Senn, "System Structure and Software Maintenance Performance," *Comm. ACM*, 32, 3 (1989), 347-358.
- Gill, G. K. and C. F. Kemerer, "Cyclomatic Complexity Density and Software Maintenance Productivity," *IEEE Transactions on Software Engineering*, 17, 12 (1991), 1284-1288.
- Glesjer, H., "A New Test for Heteroscedasticity," *J. American Statistical Association* (1969), 316-323.
- Grady, R. B., "Practical Results from Measuring Software Quality," *Comm. ACM*, 36, 11 (1993), 62-68.
- Grammas, G. W. and J. R. Klein, "Software Productivity as a Strategic Variable," *Interfaces*, 15, 3 (1985), 116-126.
- Greene, W. H., *Econometric Analysis*, 2nd ed., Macmillan, New York, 1993.
- Gremillion, L. L., "Determinants of Program Repair Maintenance Requirements," *Communications of the ACM*, 27, 8 (1984), 826-832.
- Guimaraes, T., "Managing Application Program Maintenance Expenditures," *Comm. ACM*, 26, 10 (1983), 739-746.
- Hale, D. P. and D. A. Haworth, "Software Maintenance: A Profile of Past Empirical Research," *Proc. Fourth Conf. on Software Maintenance*, IEEE Computer Society Press, Washington, DC, 1988, 236-240.
- Halstead, M., *Elements of Software Science*, Elsevier North-Holland, New York, 1977.
- IFPUG, *Function Point Counting Practices Manual*, International Function Points User's Group, J. Sprouls (Ed.) AT&T, Piscataway, NJ, 1993.
- Iman, R. L. and W. J. Conover, "The Use of the Rank Transform in Regression," *Technometrics*, 21, 4 (1979), 499-509.
- Johnson, R. A. and D. W. Wichern, *Applied Multivariate Statistical Analysis*, 3rd ed., Prentice-Hall, Englewood Cliffs, NJ, 1992.
- Kaiser, H. F., "An Index of Factorial Simplicity," *Psychometrika*, 39 (1974), 31-36.
- Kemerer, C., "Reliability of Function Points Measurement," *Comm. ACM*, 36 (1993), 85-97.
- , "Software Complexity and Software Maintenance: A Survey of Empirical Research," *Ann. Software Engineering*, 1 (1995), 1-22.
- Kriebel, C. and A. Raviv, "An Economics Approach to Modeling the Productivity of Computer Systems," *Management Sci.*, 26, 3 (1980), 297-311.
- Li, H. F. and W. K. Cheung, "An Empirical Study of Software Metrics," *IEEE Trans. on Software Engineering*, SE-13, 6 (1987), 697-708.
- Littman, D. C., J. Pinto, S. Letovsky, and E. Soloway, "Mental Models and Software Maintenance," *J. Systems and Software*, 7 (1987), 341-355.
- Locke, E. A. and G. P. Latham, *A Theory of Goal-Setting and Task Performance*, Prentice-Hall, Englewood Cliffs, NJ, 1990.
- Lynch, R. K., "Implementing Packaged Application Software: Hidden Costs and New Challenges," *Systems, Objectives, Solutions*, 4 (1984), 227-234.
- McCabe, T. J., "A Complexity Measure," *IEEE Trans. on Software Engineering*, SE-2, 4 (1976), 308-320.
- Martin, J. and C. McClure, "Buying Software Off the Rack: Packages Can Be the Solution to the Software Shortage," *Harvard Business Rev.*, November-December (1983), 32-60.
- Munson, J. C. and T. M. Khoshgoftaar, "The Dimensionality of Program Complexity," *Proc. International Conf. on Software Engineering*, IEEE Computer Society Press, Washington, DC, 1989, 245-253.
- Neter, J., W. Wasserman, and M. H. Kutner, *Applied Linear Statistical Models*, 3rd ed., Irwin, Homewood, IL, 1990.
- Nosek, J. T. and P. Palvia, "Software Maintenance Management: Changes in the Last Decade," *J. Software Maintenance*, 2, 3 (1990), 157-174.
- Olle, T. W., H. G. Sol, and A. A. Verrijn-Stuart, Eds., *Information System Design Methodologies: A Comparative Review*, North-Holland, Amsterdam, 1983.

- Oman, P. W., C. R. Cook, and M. Nanja, "Effects of Programming Experience in Debugging Semantic Errors," *J. Systems and Software*, 9 (1989), 192-207.
- Pennington, N., "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs," *Cognitive Psychology*, 19 (1987), 295-341.
- Perry, W. E., "The Best Data Processing Measures," *System Development*, 6, 6 (1986), 4-6.
- Ramanujan, S. and R. B. Cooper, "A Human Information Processing Approach to Software Maintenance," *OMEGA*, 22 (1994), 185-203.
- Schneidewind, N. F., "The State of Software Maintenance," *IEEE Trans. on Software Engineering*, SE-13, 3 (1987), 303-310.
- Senge, P. M., *The Fifth Discipline: The Art and Practice of the Learning Organization*, Doubleday, New York, 1992.
- Shapiro, S. S. and M. B. Wilk, "An Analysis of Variance Test for Normality," *Biometrika*, 52 (1965), 591-612.
- Shen, V. Y., T. J. Yu, S. M. Thebaut, and L. R. Paulsen, "Identifying Error-Prone Software—An Empirical Study," *IEEE Trans. on Software Engineering*, SE-11, 4 (1985), 317-323.
- Shneiderman, B., *Software Psychology: Human Factors in Computer and Information Systems*, Winthrop Publishers, Inc., Cambridge, MA, 1980.
- Simon, H. A., *The Sciences of the Artificial*, 2nd ed., The MIT Press, Cambridge, MA, 1981.
- Stabell, C., "Office Productivity: A Microeconomic Framework for Empirical Research," *Office Technology and People*, 1, 1 (1982), 91-106.
- Stamps, D., "CASE: Cranking out Productivity," *Datamation*, July 1 (1987), 55-58.
- Swanson, E. B. and C. M. Beath, *Maintaining Information Systems in Organizations*, John Wiley and Sons, New York, 1989.
- Vessey, I., "Toward a Theory of Computer Program Bugs: An Empirical Test," *International J. Man-Machine Studies*, 30, 3 (1989).
- Weyuker, E. J., "Evaluating Software Complexity Measures," *IEEE Trans. on Software Engineering*, 14, 9 (1988), 1357-1365.
- White, H., "A Heteroscedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroscedasticity," *Econometrica*, 48 (1980), 817-838.
- Wood, R. E., "Task Complexity: Definition of a Construct," *Organizational Behavior and Human Decision Processes*, 31 (1986), 60-82.
- , A. J. Mento, and E. A. Locke, "Task Complexity as a Moderator of Goal Effects: A Meta-Analysis," *J. Applied Psychology*, 72, 3 (1987), 416-425.
- Zuse, H., *Software Complexity: Measures and Methods*, de Gruyter, Amsterdam, 1990.
- Zvegintzov, N., *Software Maintenance News*, 6, 8 (1988).
- , "Immortal Software," *Datamation*, June (1984), 170-180.

*Accepted by John C. Henderson; received April 10, 1996. This paper has been with the authors 4 months for 1 revision.*