Software Errors and Software Maintenance Management

RAJIV D. BANKER
Carlson School of Management, University of Minnesota, Minneapolis, MN 55455, USA

SRIKANT M. DATAR
School of Management, Stanford University, Stanford, CA 94305, USA

CHRIS F. KEMERER
Katz Graduate School of Business, University of Pittsburgh, Pittsburgh, PA 15260, USA

DANI ZWEIG
Pittsburgh, PA 15218, USA

Abstract. A management model for explaining software errors is developed and estimated. The model is used to analyze two years of error log data at a commercial site. The focus is on identifying managerially controllable factors which affect software reliability. At the research site, application systems which (1) underwent frequent modification; (2) were maintained by programmers with low levels of application experience; (3) had high reliability requirements, and (4) had high levels of static complexity all showed particularly high error rates, other things being equal. It is suggested that that managers can make quantified judgements about the degree to which they wish to reduce error rates by implementing a number of procedures, including enforcing release control, assigning more experienced maintenance programmers, and establishing and enforcing complexity metric standards.

1. Introduction

The study of software reliability has generally focused upon the prevention and elimination of errors in newly developed software. The goal of such research is the production of software which is as close as possible to being error-free. However, even error-free software does not remain error-free; for every dollar spent on development, at least two or three dollars will eventually be spent on subsequent maintenance[1] – and every modification to existing software can result in the introduction of new errors [17]. Since software maintenance is an ongoing process required to keep software useful, poorly managed maintenance can result in a steady stream of errors throughout the life of the software.

[1] The term "maintenance" refers to the correction of errors, the implementation of modifications needed to allow an existing system to perform new tasks, and to perform old ones under new conditions [25].

The direct costs of error correction are estimated to exceed $10 Billion a year worldwide [7,25]. Potentially even more serious is the disruption associated with software errors. In the best of cases, software failure will inconvenience the operators or users who suffer it and the programmers who must correct it. In the worst cases it can threaten the credibility and viability of an organization.

What can be done to reduce the incidence of software errors and their associated costs? This article develops an explanatory model of error rates in software under maintenance, with a particular focus on determining the degree to which error rates can be explained by managerially controllable factors.

The results of an empirical implementation of this model are presented. Thirty five commercial application systems were tracked over a two year period at a commercial bank. In addition to enabling the prediction of error rates, the model provides guidance to those wishing to reduce them, in terms of identifying relevant factors and their impact. The analysis of these data suggests that managers at the research site can significantly reduce error rates, and allows for appropriate cost-benefit calculations to be done.

2. Previous research and conceptual model

2.1. Error rate prediction models

Most of the research on software reliability deals with the subject from the developer's point of view. It is assumed that software errors are introduced at the time the software is written, and then discovered either in testing or later in the life of the system [34]. There is a body of research which predicts the rate at which the errors will be found over time. This research typically models the discovery of errors as a Poisson process [15,27]. Extensions of this model can account for the introduction of new errors in the process of eliminating old ones [24,31]. Such an approach may enable researchers to fit a curve to observed error data, but it has little explanatory power. Its primary purpose is not to explain or prevent errors, but to aid testers in estimating the number of errors still undiscovered, and to suggest optimal testing strategies [29,30].

2.2. Factors affecting error rates

Conceptually, variations in error rates are expected to be a function of either the software system itself, or factors in the maintenance environment. There is a maintenance process, which takes the previous version of the system as its main input, but is also affected by other factors, such as the skill of the maintainers who are doing the work, and the application's required reliability. This conceptual model is pictured in figure 1, and follows from earlier work done by the researchers at the same data site [22,36].

This research has resulted in published papers dealing with the productivity of software maintainers [2,3] the measurement of software complexity [6], and the relationship between complexity and software maintenance productivity [4]. In this paper the focus of the research is on software reliability, or, more strictly speaking, software errors (the
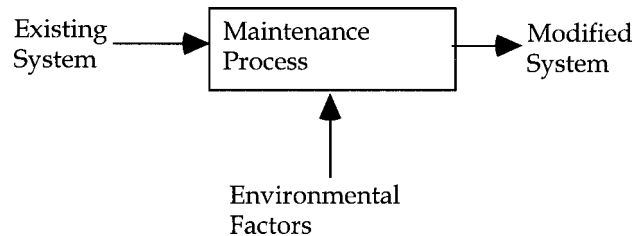
Figure 1. Conceptual model of maintenance process.

absence of reliability), and the appropriate related research is also on software reliability and software maintenance.

Reliability of existing systems has been examined in several research studies that have identified static characteristics of the system that contribute to higher error rates. Most of these characteriscs can be described as software size and complexity. Table 1, adapted from [23], summarizes some of the most relevant research in this area.[2]

Several researchers have suggested that size, measured in thousands of lines of code (KLINES), is a good predictor of errors.[3] Gremillion in a field study of PL/1 programs, found this to be a useful explanator for the number of repair requests, which is suggestive of a measure of the number of errors [19]. Similarly, Lind and Vairavan, in a field study of a medical imaging application, found that KLINES was correlated with the development effort associated with repair changes to existing code [26].

However, beyond this simple measure of size, a number of researchers have argued for relationships between more sophisticated measures of complexity and ultimately greater levels of effort to correct software defects. Henry and Kafura and Card and Agresti were among the earliest to note this relationship [12,20]. Beyond these results researchers have focused on examining specific relationships within the area of software structure complexity. Basili and Perricone and Shen both found evidence between errors and the size of modules [8,33]. Benander *et al.* in a study of student COBOL programs, found that programs containing GOTOs were more likely to have errors [10]. Most recently, Banker *et al.* found a strong relationship between not only module size and effort, but also with procedure size and the use of the GOTO construct [4].

There are factors other than static systems complexity that are also suggested in the prior research to affect error rates. Some of this research is summarized in table 2.

Chong Hok Yuen analyzed nineteen months of error reports for a large (five thousand modules) operating system [13]. High error levels were found to be associated with

---

[2] There is an extensive literature outside of the narrower focus here of software maintenance. Interested readers are referred to the books of [16,28,35].

[3] Both KLINES and Function Points are widely used measures of software size. Function Points are the preferred measure under certain circumstances, e.g., for estimation purposes early in the life cycle when Function Points are easier to estimate than KLINES, or for *ex post* analyses of multiple systems implemented in different programming languages. Since neither of these circumstances held in this analysis, use of KLINES is believed to be appropriate.

Table 1
Software maintenance research, existing system.

Author	Publication	Dependent variables	Reported results
Gremillion (1984)	Communications of the ACM	Number of repair requests	Lines of code was determined to be the most accurate predictor of repair request volume. Repair request increased as a function of size and frequency of use-related to complexity and age.
Lind and Vairavan (1989)	IEEE Transactions on Software Engineering	Number of changes to the code (errors)	Lines of code correlated with the development effort as well as their more sophisticated standard complexity metrics.
Henry and Kafura (1981)	IEEE Transactions on Software Engineering	Number of changes	Complexity measure highly correlated with number of changes made, suggesting that complexity metrics may predict error rates.
Card and Agresti (1988)	Journal of Systems and Software	Number of errors, effort	Changes in measured complexity accounted for 60% of the variation in error rate.
Basili and Perricone (1984)	Communications of the ACM	Number of errors	Larger modules appear significantly less error prone (per LOC).
Shen et al. (1985)	IEEE Transactions on Software Engineering	Number of errors	Smaller modules have a higher rate of errors per LOC than larger modules. Metrics related to amount of data and the structural complexity (number of loops, conditional statements, and Boolean operators) proved to be the most useful in identifying error prone modules at the earliest stages of testing.
Benander et al. (1990)	Journal of Systems and Software	Correctness, debugging time, structure and style measures	Programs containing GOTOs were found more likely to have errors, took longer to debug, and had worse structure than GOTO-less programs.
Banker et al. (1993)	Communications of the ACM	Effort, productivity	Module size, procedure size, and the use of complex branching were all found to significantly affect software maintenance costs.

high levels of system activity. In addition, newly-modified portions of the system tended to experience high error rates. This is presumably due, at least in part, to the infusion of errors during the maintenance process. Both these observations are consistent with stochastic models of error detection and suggest that systems undergoing a significant amount or frequency of change tend to have higher error rates.

Table 2
Software maintenance research, environmental factors.

Author	Publication	Dependent variables	Reported results
Albrecht (1983)	IEEE Transactions on Software Engineering	Effort	Systems with high reliability requirements were expected to require additional effort.
Boehm (1984)	IEEE Transactions on Software Engineering	Effort	Systems with high reliability requirements were expected to require additional effort.
Chong Hok Yuen (1985)	Proc. of the Conf. on Software Maintenance	Error reports	High error levels were found to be associated with high levels of system activity. In addition, newly-modified portions of the system tended to experience high error rates.
Schaefer (1985)	Proc. of the Conf. on Software Maintenance	LOC added, changed, deleted	Error rates not uniformly distributed but rather tend to 'clump' in error-prone modules. Notes that software components produced by inexperienced staff are likely to be more error prone.
Curtis et al. (1989)	Journal of Systems and Software	Number of errors, effort	Natural language was less effective than constrained language or ideograms in aiding programmer comprehension. One third to one half of the variation in overall performance was attributed to individual differences among participants.
Banker et al. (1993)	Communications of the ACM	Effort	Maintainer application experience was found to significantly affect software maintenance costs.

Another source of errors is the use of maintainers with low levels of experience with the application being maintained. Curtis, in a laboratory experiment, found that as much as one half of the variation in overall performance could be attributed to differences between individual programmers [14]. Schaefer and Dunn also cite inexperienced programmers and volatility of specifications as major contributors to error-proneness [32]. Most recently, Banker *et al.* have shown that lack of application experience was a major source of variation in maintenance effort, and therefore might also contribute to higher error levels [4].

A final factor that has been proposed is based on the intuitive idea that systems with high performance requirements require more effort to develop and maintain [1,11]. It could therefore be expected that, failing such additional effort, these systems may develop greater numbers of defects.

3.   A model of software reliability under maintenance

Given the previous research, it seems appropriate to test the following factors for their relationship with error rates:

- Static software system factors:

  - ● size;
  - ● complexity.

- Dynamic software system factors:

  - ● operational frequency;
  - ● volatility.

- Environmental factors:

  - ● maintainer experience;
  - ● performance requirements.

Error rate is modeled as a stochastic variable whose mean varies from application system to application system, specifically as a multiplicative function of several explanatory variables pertaining to those systems. A multiplicative form is used because the impact of each variable is believed to be higher for higher levels of other explanatory variables rather than independent of other variables as in a linear model. In particular, the error rate is modeled as a random draw from a lognormal distribution with mean Lambda (λ), where

$$\lambda(\text{mean error rate}) = f(\text{static software factors},$$
$$\text{dynamic system factors, environmental factors}),$$

where $f(\cdot)$ is a multiplicative function.

The model developed here draws upon prior research for its explanatory variables: Error rates are expected to be higher for complex and volatile systems, systems with high levels of activity, and for systems being maintained by inexperienced programmers. The number of errors detected is expected to vary in a given application system from period to period, however, because of the inherently random nature of the way in which ongoing use continues to surface hitherto-unsuspected errors.

The measures used in the model developed here are described below and implementation details are summarized at the end in table 3.

3.1. Existing system factors

3.1.1. Static software system factors
System size (KLINES – thousands of source lines of code) was employed as a measure of the magnitude of the task performed by the system. This is reasonable for traditional transaction-based systems that this research examines, as the tasks they perform may
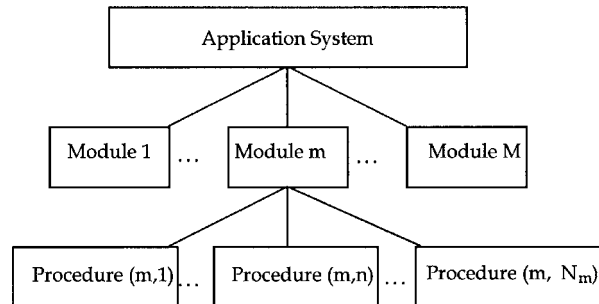
Figure 2. Software level hierarchy [4].

vary greatly in scope, but tend to be of uniformly low algorithmic complexity [5]. Also, the larger the system, the more potential there is for errors. This is one of the less controllable factors affecting error rates – a system's performance requirements tend to be given for maintenance programmers – but this effect must be controlled for in the analysis.

A given program may be difficult to comprehend and modify because of the way it is written, as well as because of what it does. New errors are most likely to propagate in systems written in a style that makes them hard for maintenance programmers to read or modify. On the basis of an earlier study [4] three measures of software complexity were selected: Average module length, in executable statements (MOD), average procedure length, in executable statements (PROC), and the proportion of executable statements which are long (outside the boundary of a paragraph) branching instructions (GOTO) [4].[4]

Figure 2 shows how modules and procedures are defined in the context of this research.

3.1.2. Dynamic software system factors

Systems which undergo frequent modification are expected to have higher error rates, because each modification represents an opportunity for new errors to be introduced. It may also be that when systems are undergoing frequent changes, there is less opportunity and less interest in testing those changes thoroughly. This effect corresponds to that of frequent specification changes in software development. The binary variable VOLATILE identifies systems which are subject to an ongoing stream of modifications. A related factor, TURNOVER is a binary variable changing over time to identify reporting periods in which a formal software maintenance project for the system has just been completed and implemented. A short-term increase in error reports is expected in such periods. Note that in the absence of good release controls at the research site, most minor modifications did not result in a formal turnover, so this variable typically captures the completion of medium to large projects.

Unlike the other factors in the model developed here, system activity does not directly influence the number of errors which exist in a system. Rather, it influences the

---

[4] Interested readers are referred to appendix A for details on computation of GOTO density.

rate at which the errors that escaped formal testing are discovered through use. The more frequently the software is used, and the greater the variety of inputs to the software, the more errors such usage may be expected to expose, all else being equal. RUNS is the number of times per period that the system is used.

3.1.3. Environmental factors

If the programmers maintaining a system have not been maintaining it long, their relative inexperience will increase the chances of unnoticed errors slipping by them. They will also be less likely to know the problem areas which require the most careful testing. Since the effort of figuring out the existing code typically consumes more time than its actual modification, application-specific experience is particularly important in maintenance. INEXPER identifies application systems maintained primarily by programmers with relatively little (under two years) experience with that system. This threshold effect was suggested by managers at the site [3]. Finally, PERFORM identifies application systems with particularly stringent performance requirements as identified by the site's managers responsible for maintaining them. These are systems in which maintaining performance efficiencies was an explicit and major goal maintenance goal. For example, in a banking environment an ATM or other customer-facing system would be expected to have high performance requirements in terms of response time and availability, while many other traditional batch processing or other periodic reporting systems would have less strenuous requirements.

3.1.4. Control variables

While the above factors are believed to capture the main managerially-controllable factors affecting software reliability in this environment, there may be other application system-specific factors that have not been accounted for. This study estimates a "fixed effects" mode with a panel of data for 35 application systems for 24 months [18]. The omission of some application system-specific factors is likely to result in residual (or error) terms that are correlated over time for the same system. The fixed effect mode controls for this possibility by including a set of dummy variables corresponding to the specific application software maintenance groups that the system belonged to. These groups, labeled GROUP1 through GROUP15, correspond to application areas within the commercial bank such as Trust Accounting or Demand Deposits (see section 4 for more details about the organization and its systems). While not a primary focus of the research, given that they do not lend themselves to the same type of managerial action as the primary explanatory variables, these dummy variables serve to control for possibly omitted variables in the analysis. The model variables with their definitions are summarized in table 3.

4.   The research site

In order to study the characteristics of typical commercial software, transaction processing application systems at a major commercial bank were analyzed [4]. The bank's

Table 3
Variable definitions.

Dependent variable	
ERRORS	Basis for the dependent variable. Equals the number of software errors reported by the site's error logs for that application in that month.
Static software system factors	
KLINES	Size of application system, in thousands of non-comment source lines of code.
MOD	Average number of executable statements per module in the system.
PROC	Average number of executable statements per procedure in the system.
GOTO	Percentage of executable statements which are branch instructions in the system. This figure excludes branches to the beginning or end of the same COBOL paragraph.
Dynamic software system factors	
VOLATILE	System is subject to frequent changes, including minor ones: Value is 1 if system undergoes modification every two months or more often, 0 otherwise.
TURNOVER	Value is 1 if a maintenance project has been completed for that system in that month, 0 otherwise.
RUNS	Number of times the system was run in that month, taken from site's operation logs.[a]
Environmental factors	
INEXPER	Value is 1 if over 80% of this system's programmers have less than 3 years of application experience, 0 otherwise.
PERFORM	Value is 1 if managers identified the system as requiring particular maintenance effort to meet performance requirements, 0 else.
Control variables	
GROUP1–GROUP15	Each system belongs to one of fifteen application-oriented groups. Value is 1 if system belongs to that group, 0 otherwise.

[a] Note that this measure refers to batch runs, and does not distinguish between high and low transaction volumes.

systems contain over 10,000 modules, totaling over 20 million lines of code. Almost all of them are COBOL programs running on large IBM mainframe computers. The programs are organized into application systems (e.g., Accounts Payable, General Ledger, Payroll) of (typically) 100–300 modules each. Some of the bank's major application systems were written in the mid-1970's or earlier, and are generally acknowledged to be more poorly designed and harder to maintain than more recently written software.

The bank's application programmers are organized into divisions (each MIS division is responsible for the software of a parallel operating division) and further divided into groups: A group is typically responsible for the ongoing maintenance of two or three application systems. There is a long-term relationship between each group and the users of the software which it maintains. Partly for this reason, formal controls on the maintenance process are relatively weak at this site, and minor modifications are typically made

Table 4
Application systems profile (35 systems).

Variable	Mean	Standard deviation	Minimum	Maximum
Errors per month	8.3	9.7	1	101
Maintenance costs/year	$693K	$661K	$83K	$3532K
Application size-modules	217	266	18	1500
Application size-lines	215KSLOC	185KSLOC	54KSLOC	702KSLOC

without the benefit of a change control process. There is little usable design or other documentation for most of the application systems. The researchers spent several months on-site collecting the data. Thirty five application systems from fifteen groups, accounting for a total of over $20 Million per year in maintenance costs, were monitored over a two year period yielding 840 system-month observations. For each system, monthly activity (batch runs) and error logs were obtained. Descriptive information about each system was obtained in a series of interviews with programmers and managers. A commercial static analyzer, INSPECTOR, was used to compute system size and software complexity metrics. Table 4 gives a brief profile of the application systems.

5.   Analysis

The rate of appearance of software errors in a maintenance environment is modeled as a random draw from a lognormal distribution.[5] In this model each application system will have its own distribution and mean error rate which is modeled as a multiplicative function of a number of explanatory factors. The parameter value of the mean error rate varies from application to application, based on the values of the structural variables which determine it. Note that, unlike the case in the development reliability models, mean error rates are not expected to decrease (or increase) over time, except as a result of changes to the structural variables. Since the maximum likelihood estimates for the lognormally distributed model with the mean expressed as a multiplicative function of the explanatory variables are yielded by minimizing the sum of squares for a logarithmically transformed dependent variable, the following fixed effects regression model was estimated:

$$\ln \text{ERRORS} = \beta 0 + \beta 1 * \ln \text{KLINES} + \beta 2 * \ln \text{MOD} + \beta 3 * \ln \text{PROC} + \beta 4 * \ln \text{GOTO}$$
$$+ \beta 5 * \text{VOLATILE} + \beta 6 * \text{TURNOVER} + \beta 7 * \text{INEXPER}$$
$$+ \beta 8 * \ln \text{RUNS} + \beta 9 * \text{PERFORM} + \sum (\chi_i * \text{GROUP}_i) + \varepsilon,$$

where i varies from 2 to 15.

[5] The lognormal distribution and the exponential distribution are widely used in the software reliability literature, both being consistent with the intuition that error rates should be distributed with an early peak and a single long tail. While the lognormal distribution was used for reasons of analytical tractability, it was verified (see below) that the observed data was indeed consistent with this distribution.

Table 5
Estimation results for fixed effects model.[a]

Explanatory variable	Variable (category)		
	Parameter estimate	T for H0: parameter $= 0$	Prob $> |T|$
ln KLINES (static)	0.12	1.92	0.055
ln MOD (static)	1.23	8.86	0.000
ln PROC (static)	−0.41	−4.29	0.000
ln GOTO (static)	0.28	4.75	0.000
VOLATILE (dynamic)	0.56	7.57	0.000
TURNOVER (dynamic)	0.13	1.72	0.086
ln RUNS (dynamic)	0.08	1.60	0.109
INEXPER (environment)	1.01	11.28	0.000
PERFORM (environment)	0.58	5.94	0.000

[a] F value 234.39 (0.0000) ($n = 840$), R-squared 0.8733.

There are 840 observations: 24 months of panel data for 35 application systems. Note that, for each application system, ERRORS, TURNOVER, and RUNS vary from month to month. The 35 systems are divided into their fifteen groups, and groups dummies (for any observation, one has value 1 and the rest have value 0). Table 5 shows the result of the estimation of the fixed effects model for the explanatory variables.

Of the static systems variables, all of the complexity-related variables (ln MOD, ln PROC, and ln GOTO) were statistically significant at usual levels, was the significance level for the size variable (ln KLINES) was 5.5%. Of the dynamic systems variables, ongoing volatility (VOLATILE) was statistically significant at the 0.01% levels, while the significance level for the rejection of the system turnover variable (TURNOVER) was 8.6%. Of the environmental variables, maintainer inexperience with the system (INEXPER), and the presence of system performance requirements (PERFORM) were both statistically significant at the 0.01% level. The number of times the systems was operated (RUNS) level of significance was 10.9%.

Diagnostic tests

Standard diagnostic tests were conducted on the estimated model. Analysis of the residuals revealed no significant heteroskedasticity, supporting the model of error rates (in mature systems) being constant over time. The White test for normality showed $W = 0.9875$ (where 1.0 is perfectly normal) consistent with the maintained hypothesis of normality of the residuals.

The Belsley–Kuh–Welsch test indicated little multicollinearity among most of the variables [9]. There were two exceptions. First, the section dummies showed very high multicollinearity. This has no important implications for the testing of the hypotheses of interest to this research, as the section dummies are only control variables and no inferences are drawn from their estimated coefficients. Second, there was high collinearity between PROC and GOTO, which is expected to have the effect of inflating the estimated standard errors and, in turn, lowering the t-statistics for both PROC and GOTO.

Table 6
Results of impact calculations.

Explanatory variable	Variable one standard deviation impact
KLINES	7.7%
MOD	40.4%
PROC	−17.5%
GOTO	23.5%
VOLATILE	75.6%
TURNOVER	12.0%
RUNS	8.1%
INEXPER	173.5%
PERFORM	79.0%

6.   Discussion

The statistical results from the previous section provide insights into how the software maintenance process can be improved by reducing the number of software errors. The implications of each set of factors from the estimated model are discussed below. Each of the coefficients can be interpreted as the effect of that variable holding all others constant. If the levels of two variables are chosen independent of one another, then their effects are interpreted as being additive.

   The managerial impact was computed as follows. For binary variables the impact of $\ln x_i$ taking the value 1, rather than 0 in the logarithmically transformed model (other things remaining constant), is to increase the percentage of errors by a factor of $(e_1^{\beta} - 1)$. For example, the parameter value for VOLATILE, 0.56, corresponds to $(1.756 - 1)$, or a 75.6% increase in errors, *ceteris paribus*. Similarly, for the continuous variables the impact of increasing x_i by a factor of k (holding everything else constant) is to increase the expected number of errors by a factor of $(k_1^{\beta} - 1)$. A representative estimate of the impact can be computed by examining the expected impact of increasing a factor's value by one standard deviation from the site mean for the 35 applications. In order to compute this for the continuous variables k is set equal to ((mean + standard deviation)/mean). For example, the coefficient parameter value for KLINES is 0.12. Also, KLINES has a mean of 245 and a standard deviation of 207, so we set $k = (245 + 207)/245 = 1.85$. This implies a $(1.85^{0.12} - 1)$, or a 7.7% increase in the mean error rate for a one standard deviation increase in KLINES. Table 6 presents a summary of these impacts.

6.1. Static software system factors

MOD, PROC, GOTO.   Each of the software complexity metrics, module size, component size, and branching density, used in this analysis was statistically significant at the 0.01% level. Together, the three variables are significant ($F(3, 816) = 38.06$, $P = 0.0001$). It is believed, based on prior research, that both too many small procedures and too few large procedures are harmful [4]. At this site, the data indicate that

errors decrease with procedure size over the observed range, as reflected in the negative sign on the coefficient for PROC. The addition of the squared term, PROC^2, did not significantly alter the estimation results. It is believed that, if procedure size were increased beyond the observed range, mean error rates would have exhibited an increase with procedure size. More importantly, for any given project, the maintenance programmers have relatively little control over the software complexity of the application; it is a legacy of the original developers. However, management can elect to improve the quality of the software maintenance process by emphasizing software complexity standards during initial development, and by similarly controlling them throughout the new releases. For old systems undergoing only infrequent maintenance, a regime of program restructuring can be initiated to restore a higher level of "systems hygiene" to these older systems.

6.2. Dynamic software system factors

VOLATILE, TURNOVER. Errors are introduced in the course of maintenance: the more frequent the maintenance, the more numerous the errors [32]. Applications which undergo frequent (every few weeks) minor changes experience 76% higher error rates than those which are modified less often. It is believed that one possible explanation for this phenomenon is that some minor modifications may not be subject to the same level of management scrutiny and quality control as might be modifications that are perceived to be major. In addition, mean error rates increase about 12% in the specific month that newly modified software goes into production, as many of the bugs which escaped the testers are discovered, but this result is not significant at the 1% level. However, together these two variables are significant at the 1% level, which suggests that organizations could improve their error performance through greater attention to consolidating small changes to ensure fewer new releases of the software.

RUNS. One standard deviation beyond the mean in the frequency of usage (batch runs) is associated with an 8% increase in the mean error rate. System activity does not directly affect the number of errors in the system, but rather the rate at which they are uncovered. The level of system activity must be considered in any error-prediction model but, like task complexity, it is a factor which is not usually controllable by those managing the maintenance activity.

6.3. Environmental factors

INEXPER. Application systems whose programmers had relatively little application-specific experience averaged 174% more errors than systems maintained by more experienced programmers. This finding is of particular managerial interest, given the research site's policy of rotating its programmers among application systems in order to gain a breadth of experience. While no doubt management had an intuitive understanding of the value of maintainer experience, they felt that it was only through a quantitative model such as the one presented here, that such ideas could be explicitly traded off against the potential benefits of rotating staff. Based on these results the bank decided to reduce the

frequency of its rotation policy. While it can be expected that maintainer experience is likely to be an important variable at any site, it might be especially important at sites without high levels of quality systems documentation.

PERFORM.   Systems with stringent performance requirements had mean rates about 79% higher than for other systems. There are two factors at work here. Mean error rates are higher because these tend to be the more complex systems along a number of dimensions, not just performance, and therefore maintaining them tends to be a more difficult task. This is not something over which managers have much control. However, the insight about performance may be used proactively to budget for more extensive testing. Error rates are higher also because the steps programmers take to meet difficult performance requirements may entail deviations from programming standards: code tends to become more dense, more obscure, and more error-prone. Once this factor is recognized and quantified, managers can better anticipate it and choose to compensate for it through the programming budget, the testing budget, or the hardware budget.

The limitations of this analysis are typical of empirical studies such as these. The model is incomplete, as there is unexplained variation in error rates determined by factors not captured in the model, such as individual maintainer differences, team effectiveness, quality of documentation, and users' knowledge. More detailed information about maintainers' prior experience and behavior would likely lead to even stronger results, and researchers are encouraged to depart, as this stream of research [2–4] has done, from early research's neglect of personnel-related variables. Second, the research site is a classic "level-one" organization, characterized by weak formal controls on maintenance, and by a general lack of good system documentation [21]. These factors could be expected to enhance the impacts of application experience and volatility. While this type of environment is believed to represent a considerable portion of the commercial segment, the results cannot be extrapolated with any confidence to organizations that do not share these features.

Finally, while the model is able to show the relationships among the variables, the appropriate managerial inferences to draw from them are dependent on a knowledge of the target environment and of the range of managerial options available. Thus, the assertion that better change and release control procedures would alleviate the level of errors found at this site, is an inference drawn from the statistical analysis, rather than a direct result.

7.   Conclusion

In this paper an explanatory model of error rates which does not assume a constant decline in errors over time was developed. The explanatory variables used in this model extended the set of variables proposed by past research. In general, those variables proposed to affect development reliability in the former case were also found to be important factors in a maintenance environment. Additional factors in the model which were relatively controllable by maintenance management were programmer application expe-

rience and system volatility, with the impact of the latter being particularly pronounced. In particular, very frequent minor modifications to a system are associated with a general increase in error rates. More frequent changes mean more errors. This implies that a maintenance team can reduce its error rates by consolidating or batching modifications into relatively infrequent releases – a conclusion which is not at odds with much conventional wisdom. But there is usually much organizational pressure to ignore this factor and to implement modifications as soon as possible. Armed with the quantitative results from the model, managers can seek to rationally balance these two concerns.

Programmers' application experience was also important, with systems maintained by relatively inexperienced programmers averaging significantly higher error rates. Of course, the more experienced programmers typically command higher wages, and programmers who have maintained a system long enough may be particularly anxious to move on to new tasks. Managers must decide what they are willing to pay for the estimated reduction in error rates from higher application experience.

Error rates increase with software complexity, but this complexity reflects a decision made early in the system's life. The findings suggest that a deliberate attempt, at the time of development, to control software complexity, could significantly reduce error rates over the life of the software. The model presented here can be used to cost-justify such an attempt.

The analysis presented here suggests that the mean error rates at the research site error rates could be more than halved by appropriate changes in its maintenance policy. The model developed for this analysis can be used to confirm or modify these findings with respect to other organizations. While the specific parameter estimates will vary from organization to organization, the basic findings are expected to have wide application. Most important, it has been shown that data can be collected which can be used to identify managerially controllable factors which affect software reliability, and to estimate the degree to which modifying those factors will reduce error rates.

Appendix A: GOTO variable

Following [4], the metric chosen for branching in the current research is a refinement of the proportion of the executable statements that were GOTO statements. Highly divisible modules should be less costly to maintain, since a maintainer can deal with manageable portions of the module in relative isolation.

While the density of GOTO statements is a measure of divisibility, it does not distinguish between more and less serious structure violations. A branch to the end of the current paragraph, for example, is unlikely to make that paragraph much more difficult to comprehend, while a branch to a different section of the module may. In addition, the modules analyzed have a large incidence of GOTO statements (approximately seven per hundred executable statements). If only a relatively small proportion of these seriously affect maintainability, then such a metric may be too noisy a measure of branching complexity. At this research site over half of the GOTOs in these modules (19 GOTOs out of 31 in the average module) are used to skip to the beginning or end of the current

paragraph. Such branches would not be expected to contribute noticeably to the difficulty of understanding a module (in most high-level languages other than COBOL they would probably not be implemented by GOTO statements). Therefore, a simple GOTO metric which does not distinguish between these and the approximately 40% less benign branch commands, will be unlikely to be managerially useful.

To avoid this problem, a modified metric was manually computed, which is the density of the GOTO statements that extend outside the boundaries of the paragraph and that can be expected to seriously impair the maintainability of the software. This is similar in concept to Gibson and Senn's (1989) elimination of "long jumps in code (GO TOs)" and is consistent with previous research at this site [4].

References

[1] A.J. Albrecht and J. Gaffney, Software function, source lines of code, and development effort prediction: A software science validation, IEEE Transactions on Software Engineering SE-9(6) (1983) 639–648.

[2] R.D. Banker, S.M. Datar and C.F. Kemerer, Factors affecting software maintenance productivity: An exploratory study, in: *Proceedings of the 8th International Conference on Information Systems,* Pittsburgh, PA (1987) pp. 160–175.

[3] R.D. Banker, S.M. Datar and C.F. Kemerer, A model to evaluate variables impacting productivity on software maintenance projects, Management Science 37(1) (1991) 1–18.

[4] R.D. Banker, S.M. Datar, C.F. Kemerer and D. Zweig, Software complexity and software maintenance costs, Communications of the ACM 36(11) (1993) 81–94.

[5] R.D. Banker, S.M. Datar and D. Zweig, Software complexity and maintainability, in: *International Conference on Information Systems* (Boston, MA, 1989) pp. 247–255.

[6] R.D. Banker, S.M. Datar and D. Zweig, Software complexity metrics: An empirical study, working paper, University of Minnesota (1991).

[7] K.A. Bannick, Breakdown of software expenditures in the Department of Defense, United States, and World, unpublished Masters thesis, Naval Postgraduate School (1991).

[8] V.R. Basili and B. Perricone, Software errors and complexity: An empirical investigation, Communications of the ACM 27(1) (1984) 42–52.

[9] D.A. Belsley, E. Kuh and R. E. Welsch, *Regression Diagnostics* (Wiley, New York, NY, 1980).

[10] B.A. Benander, N. Gorla and A.C. Benander, An empirical study of the use of the GOTO statement, Journal of Systems and Software 11(3) (1990) 217–223.

[11] B.W. Boehm, Software engineering economics, IEEE Transactions on Software Engineering SE-10(1) (1984) 10–21.

[12] D.N. Card and W.W. Agresti, Measuring software design complexity, Journal of Systems and Software 8(3) (1988) 185–197.

[13] C.K.S. Chong Hok Yuen, An empirical approach to the study of errors in large software under maintenance, in: *2nd IEEE Conference on Software Maintenance* (1985) pp. 96–105.

[14] B.S. Curtis, E. Sheppard, J.B. Kruesi-Bailey and D. Boehm-Davis, Experimental evaluation of software documentation formats, Journal of Systems and Software 9(2) (1989) 167–207.

[15] W.K. Ehrlich and T.J. Emerson, Modelling software failures and reliability growth during system testing, in: *Proceedings of the 9th International Conference on Software Engineering* (1987) pp. 72–82.

[16] N.E. Fenton, *Software Metrics, A Rigorous Approach* (Chapman & Hall, New York, 1991).

[17] J. Gallant, Survey finds maintenance problem still escalating, Computerworld 20 (January 1986).

[18] W.H. Greene, *Econometric Analysis,* 2nd edn. (Macmillan Publishing, New York, NY, 1993).

[19] L.L. Gremillion, Determinants of program repair maintenance requirements, Communications of the ACM 27(8) (1984) 826–832.

[20] S. Henry and D. Kafura, Software structure metrics based on information flow, IEEE Transactions on Software Engineering SE-7 (September 1981) 510–518.

[21] W.S. Humphrey, Characterizing the software process: A maturity framework, IEEE Software 5(3) (1988) 73–79.

[22] C.F. Kemerer, Measurement of software development productivity, unpublished Carnegie Mellon University Ph.D. thesis (1987).

[23] C.F. Kemerer, Empirical research on software complexity and software maintenance, Annals of Software Engineering 1(1) (August 1995).

[24] W. Kremer, Birth–death and bug counting, IEEE Transactions on Reliability 32(1) (April 1983) 37–47.

[25] B.P. Lientz, E.B. Swanson and G.E. Tompkins, Characteristics of application software maintenance, Communications of the ACM 21(6) (1978) 466–471.

[26] R. Lind and K. Vairavan, An experimental investigation of software metrics and their relationship to software development effort, IEEE Transactions on Software Engineering 15(5) (1989) 649–653.

[27] R. Moawad, Comparison of concurrent software reliability models, in: *Proceedings of the 7th International Conference on Software Engineering* (1984) pp. 222–229.

[28] K.H. Moller and D.J. Paulish, *Software Metrics* (Chapman & Hall, London, 1993).

[29] J. Musa, A. Iannino and K. Okumoto, *Software Reliability* (McGraw-Hill, New York, NY, 1987).

[30] J.D. Musa and W.W. Everett, Software-reliability engineering: Technology for the 1990s, IEEE Software (November 1990) 36–43.

[31] M. Ohba and X.M. Chou, Does imperfect debugging affect software reliability growth? in: *Proceedings of the 11th International Conference on Software Engineering* (1989) pp. 237–244.

[32] H. Schaefer, Metrics for optimal maintenance management, in: *Proceedings of the Conference on Software Maintenance* (1985) pp. 114–119.

[33] V.Y. Shen, T.-J. Yu, S.M. Thebaut and L.R. Paulsen, Identifying error-prone software – An empirical study, IEEE Transactions on Software Engineering SE-11(4) (1985) 317–323.

[34] M.L. Shooman, Software reliability: A historical perspective, IEEE Transactions on Reliability 33(1) (April 1984) 48–55.

[35] H. Zuse, *Software Complexity: Measures and Methods* (Walter de Grutyer, New York, 1991).

[36] D. Zweig, Software complexity and maintainability, unpublished Doctoral dissertation thesis, Carnegie Mellon University (1989).