Association for Information Systems
AIS Electronic Library (AISeL)

PACIS 1997 Proceedings

Pacific Asia Conference on Information Systems (PACIS)

December 1997

Maintaing Software Maintainability: Controlling the Rate of Software Deterioration

Taizan Chan
National University of Singapore

Teck Ho
Anderson Graduate School of Management

Maintaining Software Maintainability: Controlling the Rate of Software Deterioration

Taizan Chan
Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 0511
Tel: 65-772-2862, Fax: 65-779-4580, email: chantz@iscs.nus.sg

Teck Hua Ho
Anderson Graduate School of Management, UCLA
110 Westwood Plaza, Suite B516, Box 951481
Los Angeles, CA 90095-1481
Tel: 310-825-6691, Fax: 310-206-3337, email: teho@agsm.ucla.edu

Keywords: Software maintenance, software deterioration, maintenance productivity

Executive Summary
Most software applications have a life expectancy of more than 10 years during which they must be constantly modified and enhanced. Thus it is crucial to achieve high maintenance productivity not only immediately after an application is installed but also over its entire operational life. However, it is often suggested that the productivity of maintaining a software tends to *decrease* with the age of the software. While it is recognized that this is due to the deterioration in the structure and documentation quality of the program, it is not clear what factors drive the *extent* of deterioration (in each maintenance job).

This paper examines the impact of several controllable factors including (i) the level of effort expended by a programmer on the maintenance task, (ii) the programmer's level of knowledge and skills, (iii) system characteristics, and (iv) task complexity on the rate of software deterioration. We found that a programmer with a higher level of programming skills and knowledge tend to produce more maintainable codes. We also found that a functionally more complex application leads to a greater rate of software deterioration and an ill-structured and poorly documented application tends to *prime* a patched and ad-hoc enhancement. The rate of software deterioration, however, decreases with the level of effort expended on the current maintenance job, suggesting that there may be a fundamental tradeoff between current and future maintenance productivities.

Several implications can be drawn from our results. Firstly, a higher price premium should be given to a programmer with a high level of programming knowledge since he can not only accomplish a maintenance task in less time but also with a lower extent of deterioration. Secondly, a system manager should ensure that all programmers contribute to maintaining a "quality chain" as it may be difficult to restore the structure and documentation quality once it starts to deteriorate. Thirdly, a large and complex software should be downsized into smaller, less complex ones since a larger software tends to deteriorate faster. Fourthly, related and small requests should be batched and serviced as a single request to reduce the frequency of modification to the software which may be a more significant driver of software deterioration than the complexity of the combined request. Finally, since effort that could otherwise be expended on producing codes must be expended on maintaining the structuredness and documentation quality of a program, it is important that an IS manager give sufficient recognition to code quality to motivate programmers to maintain software maintainability.

1. Introduction

Despite the rapid drop in the per unit cost of computing power, Information Systems (IS) expenditures continue to grow (Banker, Datar, and Kemerer, 1991). It is well-recognized that a major contributor to the rising IS expenditures is the escalating cost of software maintenance.[1] On average, 50%-80% of

[1] Another major factor for increasing IS expenditure is the increase in demand for computing in general (Boehm, 1987; Gurbaxani and Mendelson, 1990).

an organizational IS budget is allocated to software maintenance (Swanson and Beath, 1989; Nosek and Palvia, 1990). This is a substantial amount given that the average annual IS budget of a US organization is approximately US$2.5 million.[2] Furthermore, as new software systems continue to be developed at a faster rate than old software systems are discarded, the trend of high software maintenance cost is likely to continue in the near future (Boehm, 1983; Swanson and Beath, 1989).

Two factors contribute to the huge and rising software maintenance costs. First, software maintenance is a knowledge-intensive production process. Every software maintenance project involves (1) the understanding of the existing program and of the maintenance request initiated by the users; (2) the translation of the request into executable program code that fits into the existing software; and (3) the testing of the validity of the modifications. Each step of the process requires a considerable amount of programming, domain and application knowledge (Pennington and Grabowski, 1990). Consequently, the process is difficult to automate and expensive programming labor remains the key factor input to this knowledge-intensive production process (Gurbaxani and Mendelson, 1987).

Second, a software system often has a long operational life. Swanson and Beath (1989) found that the average life expectancy of a software is more than 10 years. During its life expectancy, an application must be constantly modified and enhanced to meet the changing needs of the organization. These changes are usually not well-integrated into the overall software design and not documented to reflect the current state of the program. The resulting program may contain ad-hoc "patches" such as "GOTO" statements and incomplete or out-dated documentation. Consequently, the program becomes harder to understand, change, and test (Lehman and Belady, 1985). This further exacerbates the level of programming effort and hence the cost required to maintain the software.

Some recent research has addressed the first factor by identifying the drivers of *maintenance productivity*. For example, Banker, Datar, and Kemerer (1991) determined, among other things, that the complexity of the maintenance task is a significant driver of maintenance productivity.[3] Banker et al. (1993), Gibson and Senn (1989), and Kafura and Reddy (1987) found that the structure of a software is a significant determinant of the effort required to accomplish a maintenance task on the software. Ho and Chan (1995) also found that a programmer's levels of domain and programming knowledge play significant roles in predicting his maintenance productivity. This stream of research, however, focuses on the control of *current* maintenance productivity. It does not address the second issue of deteriorating software structure and documentation quality, which affects *future* maintenance productivity.

Although several approaches, such as software restructuring and replacement, have been proposed for revitalizing the maintainability of a deteriorated software and hence future maintenance productivity (Jones, 1989; Gode, Barua, and Mukhopadhyay, 1990; Chan, Chung, and Ho, 1994; 1996), these approaches have their drawbacks. For example, restructuring would improve the structure of a software but increase its size (in SLOC).[4] Furthermore, program documentation and comments could not be brought up-to-date with software restructuring alone. While rewriting and replacing an aged software with a fresh one could overcome these problems, it would require a huge initial investment in the development of the new software. The investment may easily overwhelm the potential savings in maintenance effort derivable from the fresh software structure and documentation (Chan, Chung, and Ho, 1996). In particular, Chan, Chung, and Ho (1996) have shown that software replacement would not be economical if the rate of deterioration of the new software would be high anyway.

Thus, it may be more viable for an IS manager to control for future maintenance productivity by controlling the rate of software deterioration directly. Banker, Datar, and Kemerer (1991) found, contrary to conventional wisdom, that the use of structured methodology during maintenance increases the effort required in the maintenance task. This puzzling finding leads one to ask what is

---

2 This figure is derived based on the average budget figure of US$1.8 million obtained in the *Datamation* survey in 1989 (Hodges, 1989) and a 5% per annum rate of increase (based on the IS budget surveys by *Datamation* from 1990-1994).

3 They also found that good hardware response time can reduce the maintenance effort required while the use of structured methodology in maintenance increases the effort required to accomplish the maintenance task (Banker, Datar, and Kemerer, 1991).

4 Empirical research has shown that software size is a significant predictor of the magnitude of maintenance effort required (see, for example, Curtis, et al. (1979) and Kafura and Reddy (1987).

the payoff from using structured methodology. Banker, Datar, and Kemerer (1991) conjecture that the real benefit of structured methodology is that it slows down the rate of software deterioration, which helps to reduce future maintenance effort. If this conjecture is true, then there may be a tradeoff between the current and future maintenance productivities. This paper attempts to determine empirically if additional effort is indeed needed to maintain the structure and documentation quality of a program. That is, it attempts to evaluate what is the impact of the level of effort expended by a programmer on the rate of software deterioration. In addition, it evaluates the impact of three groups of factors that have been found to influence (current) maintenance productivity: (1) the level of knowledge and skills of a programmer, (2) the complexity of the maintenance task, and (3) the characteristics of the existing system. Specifically, this paper attempts to answer the following two research questions:

1. Is there a tradeoff between the current and future maintenance productivities?

2. What factors impact the rate of system deterioration?

An experiment is conducted to address these research questions as it is extremely difficult to obtain good data from the field for this study. In particular, we manipulate (1) the characteristics of the system (both the functional complexity and program structure and documentation quality), and (2) task complexity, and measure (3) the level of a programmer's level of programming and domain knowledge and analyze their impact on software deterioration.

The rest of the paper is organized as follows: Section 2 describes the conceptual model underlying our research hypotheses. Section 3 describes the experiment design and the variables used in data analysis. Section 4 presents and discusses the analysis results based on multiple linear regression. Section 5 concludes the paper by drawing managerial implications for software maintenance management.

2. Conceptual Model

Building upon the theoretical foundations of Banker, Datar, and Kemerer (1991), Kriebel and Raviv (1980), and Stabell (1982), this paper views a software maintenance project (i.e. the task of servicing a software maintenance request) as an economic production process, as illustrated in Figure 1.

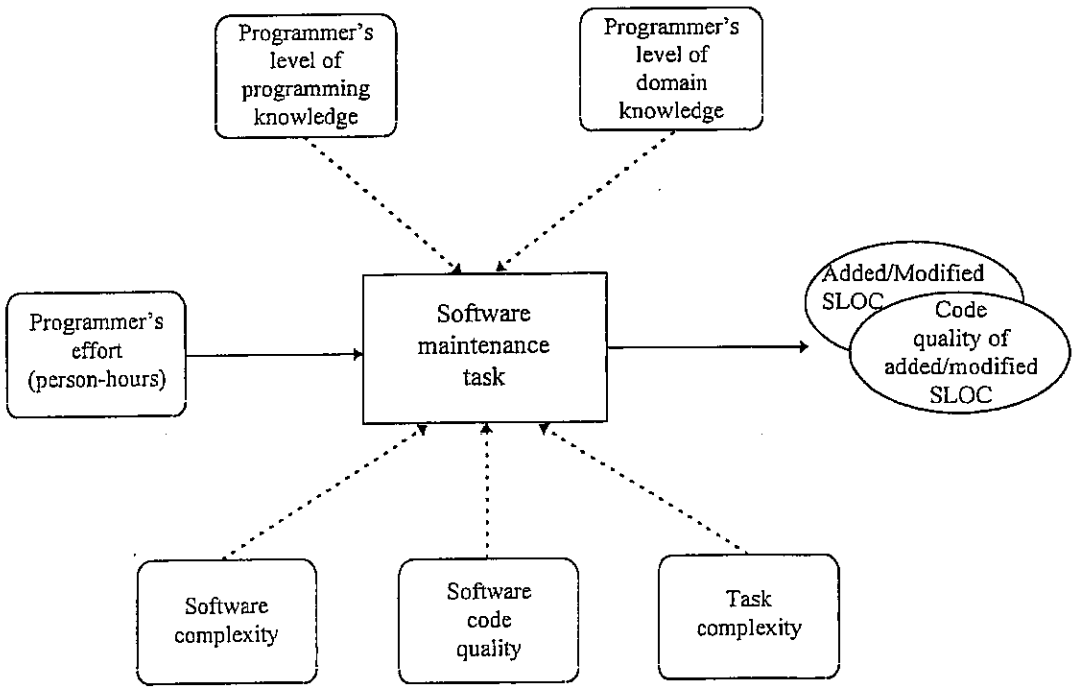

Figure 1. Software maintenance as a production process

2.1 The software maintenance production process

The software maintenance task generally consists of three stages (Pennington and Grabowski, 1990):

1. *Problem and program understanding.* Similar to new software development, a maintainer need to comprehend the objective and the problem entails in the maintenance request before proceeding to provide a solution to the problem. Unlike new software creation, however, the maintainer also need to understand the existing program to decide how a change could fit in the program. This involves understanding not only the business procedures and algorithms embodied in the program but also the structure, flow control, and operational requirements of the program to be modified (Berns, 1984; Elshoff and Marcotty, 1982).

2. *Program coding.* After determining how a change could be incorporated into the existing program, the maintainer must effect the modification. He may either change, delete, or add codes to the existing modules in order to correct or modify a processing logic or define new modules and integrate them with the existing modules.

3. *Program testing.* Finally, the maintainer must revalidate the correctness of the change (Yau and Collofello, 1985). Selective retest must be performed to ensure that not only the modified or new logic is correct but the program as a whole still functions correctly.

Each of these stages involve complex cognitive activities that rely on both the knowledge of the programming language in which the software is written and the knowledge of the functional domain of the software (Boehm-Davis, 1988; Pennington and Grabowski, 1990). Consequently, like other knowledge-intensive processes, the critical input to the software maintenance process is the effort expended by the programmer who possesses a certain level of knowledge and skills.[5] We measure the level of programmer's effort by the number of person-hours he expended on the maintenance task.

The output of this process is the modified and/or added codes that enable the application to meet the new user requirements. Similar to the standard production theory, the level of this output is found to be an increasing function of the amount of input resources expended and to be affected by several factors (Banker, Datar, and Kemerer, 1991). However, unlike the standard production theory, which often defines the level of output by its quantity (which in this case is the number of SLOC added and/or modified), output here can also vary in *quality*.

Existing literature on software maintenance productivity has focused solely on improving the first dimension (i.e. quantity) of output. That is, the emphasis is on raising the *current* maintenance productivity. Even when quality is considered (for example, in Banker, Datar, and Kemerer (1991)), it is only incorporated as an independent variable to control for its potential impact on maintenance productivity.[6] In contrast, this paper focuses directly on the output *quality* of a maintenance task as the dependent variable. It is measured by the extent of deterioration in the structure and documentation of the program[7] which has direct impact on *future* maintenance productivity.

2.2 Programmer's knowledge and skills, software characteristics, and task complexity

The existing stream of literature on software maintenance productivity has identified several factors that affect the output quantity from a software maintenance production process (with respect to a given level of effort input). The impact of these factors on the output *quality*, however, have not been systematically studied. This paper evaluates the impact of these factors on the resultant code quality (i.e. the extent of deterioration). Specifically, we consider three groups of factors: (i) the programmers' level of skill and knowledge, (ii) the characteristics of the software under maintenance, and (iii) the complexity of the software maintenance task.

---

5 Other input resources include hardware (CPU time, disk storage space, etc.) and software (program editors, debuggers, etc.). In this paper, we focus only on the programmer effort, which constitutes the largest proportion of the resources utilized in software maintenance (Lientz and Swanson, 1980; Swanson and Beath, 1989).

6 Furthermore, this stream of literature typically considers quality from the users' perspective (Banker, Datar, and Kemerer, 1991; Case, 1985; Kriebel, 1979). A typical measure of such quality is the number of errors per thousand lines of source codes. Here, we are concerned with the quality of the system from the maintainers' perspective.

7 A description of how this variable is measured is provided in the Appendix.

Programmer's skills and knowledge
There is an extensive body of literature on the impact of a programmer's knowledge and experience on his software development performance (see for example, Boehm, 1981; DeMarco, 1982; Jones, 1986). Recently, Ho and Chan (1995) also found that the level of a programmer's level of programming and domain knowledge are significant predictors of maintenance productivity. In this paper, the knowledge and skills of a programmer we are studying includes both the knowledge about the programming language in which the application has been developed and the functional domain for which application has been developed. Programming knowledge includes knowledge of the syntax and semantics of a programming language as well as knowledge of some common computational algorithms and methods (Jones, 1986). Domain knowledge includes general knowledge of the rules, methods, and policies of the problem domain (Jones, 1986). We measure the level of a subject's programming knowledge and skills by the examination score the subject obtained in a programming course and his domain knowledge by the score he obtained in a course involving the subject domain employed in this study.

System characteristics
Unlike new system development, a maintainer must work within the constraint of an existing program. Therefore, factors related to the characteristics of the existing program under maintenance must be considered. In this study, we consider two program characteristics: (1) functional complexity and (2) the code quality of the program. Program functional complexity refers to the number of functions entailed in the program. It is often measured by the number of function points (Albretch and Gaffney, 1983).[8] The code quality of the program refers to the structuredness and the quality of the comments embedded in the program. In this study, program functional complexity is varied at two levels (low and high) and code quality is varied at three levels (low, medium, and high).

Task complexity
The complexity of a task has been found to be a significant determinant of maintenance effort (Banker, Datar, and Kemerer, 1991). It is therefore included in this study. We introduce task complexity through the differences in the complexity of the business rules associated with each task. One task involves a sequence of well-specified steps while the other involves more complex comparison and decision structures.

2.3 Research hypotheses
Based on the production process model discussed above, we make the following hypotheses.

Impact of maintenance effort
As pointed out earlier, the level of output, in terms of the quantity of SLOC, is found to increase with the level of programmer's effort input (Banker, Datar, and Kemerer, 1991; Banker, et al., 1993). Similar to the number of SLOC, we hypothesize that good program structure and documentation does not come free. With the same amount of SLOC added/modified (and other factors being equal), *additional* effort must be spent in structuring codes so that they are well-modularized and in deciding what and how to write the comments that are clear and useful. Consequently, we hypothesize that, controlling for the quantity of output defined by the number of SLOC, a higher level of effort is required to achieve a lower rate of deterioration. This leads us to our first hypothesis:

H1. Controlling for the number of SLOC added/modified, the extent of system deterioration will decrease with the level of effort expended on the maintenance request.

Impact of programmer's knowledge and skills
Good programming practices such as structured programming and programs documenting are usually integral parts of the programming skills and knowledge that students acquired in programming courses.[9] We therefore conjecture that students with a higher level of programming skill and

[8] Function point is a measure of the functional complexity for an MIS-type system. It involves first the calculation of a raw measure, which is a weighted function of the number of inputs and outputs, the number of files, and the number of interfaces a software system has. The raw measure is then adjusted by a set of 14 technical characteristics of the software system to yield the final function point measure. For further discussions of function point measurement, see Dreger (1989); Jones (1991); and Kemerer (1987).

[9] In all graded assignments as well as the final examination in the programming course taught at the University where the study is conducted, the extent of documentation was given a significant weight in the grading scheme.

knowledge (as measured by their course grades and explained in the following section) will tend to pay more attention to the structuredness and the documentation of the program. That is, we make the following hypothesis:

H2. *The extent of system deterioration decreases with the level of programming knowledge and skills of the maintainer.*

Ho and Chan (1995) found that domain knowledge has significant impact only on the effort required in the understanding phase but not on the coding phase of the software maintenance process. That is, the level of domain knowledge does not appear to significantly influence the coding behavior of a programmer. Since program codes (and hence their structuredness) and comments are usually produced in the coding phase, we conjecture that the level of domain knowledge will have no significant impact on the extent of software deterioration. This leads us to the next hypothesis:

H3. *The extent of system deterioration is not significantly affected by the level of domain knowledge of the maintainer.*

Impact of system characteristics

A functionally more complex application requires a greater level of effort to understand and modify (Banker et al., 1993). There is thus a strong tendency for the programmer to "zero-in" only on the affected portions of the program and modify them without considering the overall program structure (Littman, et al., 1986). Consequently, these modifications may not be well integrated with other parts of the program. Thus a more complex software is likely to aggravate system deterioration. We therefore make the following hypothesis:

H4. *The extent of system deterioration increases with the complexity of the existing program.*

The structuredness and documentation quality of the software can also affect the rate of software deterioration. Existing psychology literature on the problem of "framing" (see, for example, Tversky and Kahneman, 1982) suggests that an equivalent problem framed (i.e. described or presented) differently can lead to different behaviors. This is because the way a problem is presented can affect the frame of reference of an individual with regard to that problem, which in turn determine the individual's subsequent behavior with respect to that problem (Clancey, 1991). We hypothesize that programs with (significantly) different levels of code quality would induce different levels of reference (of code quality) and would elicit codes with corresponding levels of quality from a programmer. That is, a highly structured and well-documented program should tend to prime an enhancement with high code quality (i.e. a lower rate of deterioration) while a less well-structured and documented program would prime an enhancement with lower code quality (i.e. a greater extent of deterioration). We therefore make the following hypothesis:

H5. *The extent of system deterioration decreases with the current structure and documentation quality of the program.*

Impact of task complexity

Like the complexity of the existing program, the complexity of the task may focus a programmer's attention on the understanding step of the maintenance process. Consequently, he is likely to be less attentive to the structure and documentation quality. In addition, a complex task is likely to involve more changes to the software which makes it harder for a programmer to integrate all the changes into the existing program. Thus, the rate of software deterioration is hypothesized to increase with the task complexity:

H6. *The extent of system deterioration increases with the difficulty of the maintenance request.*

3. Experiment Design and Variables

We conducted a controlled experiment to investigate the impact of a programmer's effort, his level of knowledge and skills, the characteristics of the existing software, and the task difficulty on the rate of system deterioration. The methodology adopted is similar to that in Tenny (1985), and Gibson and Senn (1989). A 2x3 factorial design was used to control the program complexity (low or high) and program structure and documentation (low, medium, or high).

Two Pascal programs of different functional complexity were used for maintenance.[10] Both programs were developed for inventory control and forecasting. The high complexity program could process four types of transactions and involved a more elaborate (transaction) error checking routine. The lower complexity program was similar to the high complexity one but processed only two types of transaction records and had a simpler error checking routine. The domain of these applications was taught as part of the course in which the subjects were enrolled. Three code quality versions of each program were created. The original program, which was carefully modularized and commented, served as the high code quality version. The medium quality program was obtained by removing all the functional comments in the high quality program. The low quality program was obtained from the medium code quality program by replacing some of the procedure-call statements with the codes that define the body of the procedures. This is done only for all the procedures that were called only once since they were less likely to be proceduralized by programmers in practice.

Each subject (320 all together) was randomly assigned to one of the six software versions and one of the two maintenance tasks, 1 and 2. TASK 1 involved enhancing the capability of the program to forecast the demand for inventory more accurately while task 2 involved the enhancement of the inventory reorder function of the program that will minimize cost.[11] The new forecasting algorithm required involved straight-forward step by step computations while the new reorder function involved a more complex decision structure that require nested branching. The subjects were told that they had 10 days to complete the task and were asked to record the total time they spent on the maintenance task.[12]

We manipulate the following software and task characteristics:

1. Program complexity (COMPLEXITY) - The functional complexity of the program being maintained is measured by the number of function points. The more complex program has a function points count of 10 while the less complex one has a function points count of 8.

2. Program structure and documentation quality (CODEQUAL) - The structure and degree of documentation of the program being maintained takes the value of either 0, 1, or 2. 0 represents low quality, i.e. un-modularized and un-commented program, 1 represents modularized but un-commented program, and 2 represents well-modularized and well-documented program.

3. TASK complexity (TASK) - The difficulty of the maintenance task takes the value of either 1 or 2. Task 1 represents the task of modifying the forecasting algorithm, and task 2 represents the task of modifying the reordering algorithm which is more complex.

We collected the following data about each subject's level of knowledge and skills:

1. Programming knowledge and skills (PROGKNOW) - The level of a subject's programming knowledge is measured by his examination score in the Pascal programming course taken in the semester preceding this study.

2. Domain knowledge (DOMAINKNOW) - The level of a subject's domain knowledge is measured by the examination score obtained in the course in which the maintenance assignment was given.

---

10 Pascal was chosen because it is the main programming language used in many courses and projects undertaken by the subjects. Thus, our measure of their programming knowledge would reflect closely the level of programming knowledge of a fresh graduate that a firm is likely to hire.

11 The six versions of the programs and the descriptions of the tasks are available from the authors upon requests.

12 Subjects were told that the length of time has no bearing on their grades. The reported times would be used to better design the course assignments next year.

For each maintenance task, the following data are collected:

1. Extent of system deterioration (DETERIORATE) - The degree of deterioration in the structure and documentation of the program after modification is measured on a 10-point scale. 0 implies new codes that are carefully modularized and documented while 10 implies new codes that are neither modularized nor documented. One expert rated all programs. A random sample of 30 modified programs was drawn and rated independently by two other experts who are unaware of the design and purpose of the experiment. The Cronbach's alpha for the inter-rater reliability is 0.862, suggesting that the deterioration measure is stable and consistent.[13] Please refer to the Appendix for details on how this measure is obtained.

2. Total effort (EFFORT) - The self-reported total effort spent in performing the maintenance task is measured in (person-) hours.

3. Source lines of codes added/modified (SLOC) - Since both maintenance tasks involve enhancements to the existing program, the output consists of largely added (rather than modified) codes. Consequently, the quantity of codes output can be obtained by subtracting the total number of SLOC in the modified program from the number of SLOC in the original program assigned to each student.

4. Results and Discussion

Of the total 320 subjects involved, 31 subjects did not submit the assignments and 19 did not provide all the necessary data in the correct form (for example, some subjects report the elapsed time rather than the effort expended). Consequently, only 270 data points were usable. Table 1 presents the descriptive statistics of the continuous variables.

N = 270				
Variables	Mean	S.D.	Min.	Max.
DETERIORATE	6.34	2.06	0.0	10.0
EFFORT	20.97	10.55	3.75	63.0
PROGKNOW	50.89	10.07	29.0	84.0
DOMAINKNOW	45.76	14.0	0.0	82.0
SLOC	216.72	71.56	83.0	564.0

Table 1. Summary statistics of the continuous variables

Table 2 shows the correlations among the continuous variables. Further analysis of collinearity is performed by regressing each of the continuous independent variable against the other independent variables. The maximum R^2 is 0.3073 (when PROGKNOW is regressed against other independent variables), suggesting the absence of significant collinearity among the independent variables.

Variables	EFFORT	PROG-KNOW	DOMAIN-KNOW	SLOC
DETERIORATE	-0.1213 (0.0465)	-0.1387 (0.0227)	-0.1588 (0.0089)	-0.1915 (0.0016)
EFFORT	-	-0.3412 (0.0001)	-0.2353 (0.0001)	0.0888 (0.1456)
PROGKNOW	-	-	0.4712 (0.0001)	-0.1109 (0.0688)
DOMAINKNOW	-	-	-	0.0436 (0.4756)

Table 2. Pearson correlation coefficients among the continuous variables

[13] The three experts are instructors of programming courses in the computer science department. All of them have masters degree in computer science and have worked for more than 5 years in the department.

We also tested for the potential interaction between the variable EFFORT and the other independent variables.[14] The multiple-partial *F-test* yields a test statistics of 1.0317 ($F_{(7,254,0.95)} = 2.0457$), rejecting the hypothesis that there is any interaction among EFFORT and the other independent variables.

Consequently, no other variable is included in the analysis and the multiple linear regression model tested is as follow:

DETERIORATE = F(COMPLEXITY, CODEQUAL, TASK, EFFORT, PROGKNOW, DOMAINKNOW, SLOC).

Dependent variable: DETERIORATE						
Independent Var.	d.f.	S.S.	Estimate	t-value	F-value	Significance
COMPLEXITY	1	14.75	-	-	4.72	0.0307
CODEQUAL	2	185.03	-	-	29.54	0.0001
TASK	1	9.02	-	-	2.88	0.0908
EFFORT	1	22.08	-0.0299	-2.66	7.05	0.0084
PROGKNOW	1	35.46	-0.0433	-3.36	11.32	0.0009
DOMAINKNOW	1	5.47	-0.0117	-1.32	1.75	0.1872
SLOC	1	48.80	-0.0065	-3.95	15.58	0.0001
Model S. S.	323.15 (d.f. = 8)					
Error S. S.	817.32 (d.f. = 261)					
F-value	12.90					
Pr > F	0.0001					
R^2	0.2833					

Table 3. Results of the multiple regression analysis

The results of the multiple regression analysis are shown in Table 3. The following observations can be made from the multiple regression results:

1. Consistent with hypothesis H1, the extent of system deterioration decreases with the level of effort expended on the maintenance task. This result supports the conjecture that there is a tradeoff between current and future maintenance productivities. A higher level of effort is indeed required in the current maintenance task to lower the extent of deterioration to enhance future maintenance productivity.

2. Consistent with our hypotheses H2 and H3, the extent of system deterioration is significantly dependent on the level of programming knowledge and skills but not on the level of domain knowledge of a programmer. Indeed, the average extent of deterioration of a good programmer is lower than that of the average programmer: the average extent of software deterioration for students with grades A or B is only 5.82 while that of students with grades C or D is 6.47 (11.2% more). This result suggests that a programmer with a higher level of programming skills and knowledge can reduce not just the current maintenance effort, but also the future maintenance effort as he is more likely to maintain the structure and documentation quality of the program.

3. Consistent with hypothesis H4, the extent of system deterioration increases with the functional complexity of the existing program.[15] This result suggests that a larger or functionally more complex application tends to deteriorate faster. As Swanson and Beath (1989) noted, an older application increases in size because more functions have been added to it since installation. Our result therefore

14 These are EFFORT*COMPLEXITY, EFFORT*CODEQUAL, EFFORT*PROGKNOW, EFFORT*DOMAINKNOW, EFFORT*TASK, and EFFORT*SLOC.

15 The mean deterioration in the high complexity program group is 6.6090 while that in the low complexity group is 6.0876. A Ryan-Einot-Gabriel-Welsch Multiple F Test reveals that these means are significantly different at $p < 0.05$ level.

suggests that not only do the absolute effort (per unit of maintenance) increase but the *marginal* effort (per unit of maintenance) required to maintain an application also increases as a software ages.[16]

4. Consistent with hypothesis H5, the extent of system deterioration reduces with the code quality of the existing program.[17] This result suggests that "framing" effect (Tversky and Kahneman, 1982) is indeed present in software maintenance. In fact, the code quality of the existing program appears to be the most significant predictor of the extent of software deterioration (*F-value* = 29.54), suggesting a strong priming effect of the existing structure and documentation of the program.

5. Contrary to hypothesis H6, the extent of system deterioration is not affected by the difficulty of the task. This result suggests that over the software life cycle, the *frequency* of maintenance performed on the system may be a more important determinant of the extent of system deterioration that the collective complexities of the tasks.

5. Implications

This paper investigates the impact of a programmer's level of effort expended on a maintenance task, his level of knowledge and skills, system characteristics, and task complexity on the extent of deterioration of a software after a maintenance job is performed on the software. It was found that the level of programming knowledge and skills is a main driver of the resultant system deterioration. We also found that the existing code quality of a program has a strong priming effect on the resultant extent of system deterioration in the program, suggesting the need for a high level of structure and documentation quality right from the beginning of a software's life cycle. Finally, we found that the rate of system deterioration can be reduced by investing in current maintenance effort.

Based on our findings, we draw the following implications for controlling the maintenance cost of a program over its life cycle:

1. *Emphasize on the level of programming knowledge and skills when considering candidates for the maintenance job.* Our analysis shows that a programmer with a high level of programming knowledge should be given a price premium because not only can he accomplish a maintenance task in less time but also with a lower extent of deterioration. This helps to reduce both current and future maintenance efforts.

2. *Ensure that all programmers are equally responsible for upkeeping the structure and documentation quality of the program.* Our results show that a programmer tends to maintain the structure and documentation quality of a program if the structuredness is already high. This implies that a system manager should ensure that all programmers contribute to maintaining a "quality chain". It may be difficult to restore the structure and documentation quality once it starts to deteriorate. In addition, it is important to ensure that a newly developed software has a high initial code quality.

3. *Downsize (i.e. offload) large application software.* Our findings suggest that a system will deteriorate faster when it is large (i.e. functionally more complex). This finding, couples with the existing literature on the negative impact of software size on maintenance effort (Jones, 1991; Swanson and Beath, 1989), suggest that maintenance effort can perhaps be reduced by downsizing large application programs into smaller, less complex ones. Downsizing also helps to distribute the requests over a number of systems to reduce maintenance frequency and therefore, reduce the extent of deterioration in any one particular subsystem.

4. *Batch related requests.* Banker and Slaughter (1994) found that (up to certain project size) there is an economy of scale in software maintenance. They suggest that a higher level of maintenance productivity can be achieved when several small maintenance requests are batched and serviced as a single request rather than separately. Our findings suggest further that such batching has the

---

16 A qualification is need here. This statement does not take into consideration the saving in effort that application familiarity of maintainers can bring to bear. For a detailed analysis of the impact of program familiarity on software maintenance effort, please refer to Ho and Chan (1995).

17 The mean deterioration in the low, medium, and high code quality groups are 7.2416, 6.5330, and 5.2667 respectively. A Ryan-Einot-Gabriel-Welsch Multiple F Test reveals that these means are significantly different at $p < 0.05$ level.

additional benefit of lowering the rate of deterioration. This is because it reduces the frequency of modification to the software, which may be more significant in determining the total extent of deterioration than the complexity of the combined request.

5. *Give credit to effort spent in maintaining the code quality of the program.* As our results show, there is indeed a tradeoff between the current and future maintenance productivities. Maintainability does not come free and additional effort must be expended to maintain the structuredness and documentation quality of the program. It is therefore important for an IS manager to give sufficient recognition to code quality to motivate programmers to spend effort in maintaining software maintainability.

6. References

Albrecht A. J.; and Gaffney, J. E. "Software function, source lines of code, and development effort prediction: A software science validation." IEEE Transactions on Software Engineering, Volume 9, November 1983, pp. 639-648.

Banker, R. D.; Datar, S. M.; and Kemerer, C. F. "A model to evaluate variables impacting the productivity of software maintenance projects." Management Science, Volume 37, January 1991, pp. 1-18.

Banker, R. D.; Datar, S. M.; Kemerer, C. F.; and Zweig, D. "Software complexity and maintenance costs," Communications of the ACM, Volume 36, November 1993, pp. 81-94.

Banker, R. D.; Slaughter, S. A. "Project size and software maintenance productivity: Empirical evidence on economies of scale in software maintenance," In J.I. Degross, S. L. Huff, and M. C. Munro (eds.), Proceedings of the Fifteenth International Conference on Information Systems, Vancouver, Canada, December 1994, pp. 279-289.

Berns, G. M., "Assessing software maintainability," Communications of the ACM, Volume 27, January 1984, pp. 14-23.

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

Boehm, B. W. "The economics of software maintenance." in R. S. Arnold (ed.), Software Maintenance Workshop Record, Monterey, California, December 1983, pp. 14-37.

Boehm, B. W. "Improving software productivity." IEEE Computer, Volume 20, September 1987, pp. 43-57.

Boehm-Davis, D. "Software comprehension," in M. Helander (ed.), Handbook of Human-Computer Interaction, Amsterdam: Elsevier Science, 1988, pp. 107-121.

Case, A., "Computer-aided software engineering." Database, Volume 17, Fall 1985, pp. 35-43.

Chan, T.; Chung, S.; and Ho, T. "Timing of software replacement." In J.I. Degross, S. L. Huff, and M. C. Munro (eds.), Proceedings of the Fifteenth International Conference in Information Systems, Vancouver, December 1994, pp. 291-307.

Chan, T.; Chung, S.; and Ho, T. "An economic model to estimate timings of software rewriting and replacement." IEEE Transactions on Software Engineering. Volume 22, August 1996, pp. 580-598.

Clancey, W. J. "The frame of reference problem in the design of intelligent machines." In K. Vanlehn (ed.), Architectures for Intelligence, Hillsdale, New Jersey: Erlbaum, 1991, pp. 357-423.

Curtis, B.; Sheppard, S. B.; Milliman, P.; Borst, M. A.; and Love, T.; "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics." IEEE Transactions on Software Engineering, Volume 5, February 1979, pp. 96-104.

DeMarco, T. Controlling Software Projects. New York, New York: Yourden Press, 1982.

Dreger, J. B. Function Point Analysis. Englewood Cliffs, New Jersey: Prentice-Hall, 1989.

Elshoff, J. L.; and Marcotty, M. "Improving computer program readability to aid modification." Communications of the ACM, Volume 32, March 1982, pp. 512-521.

Gibson, V. R.; and Senn, J. A. "System structure and software maintenance performance," Communications of the ACM, Volume 32, March 1989, pp. 347-358.

Gode, D. K.; Barua, A.; and Mukhopadhyay, T. "On the Economics of the Software Replacement Problem." In J. I. Degross, M. Alavi, and H. Oppelland (eds.), Proceedings of the Eleventh International Conference on Information Systems, Copenhagen, Denmark, December 1990, pp. 159-170.

Gurbaxani, V.; and Mendelson, H. "Software and Hardware in Data Processing Budgets." IEEE Transactions on Software Engineering}, Volume 13, September 1987, pp. 1010-1017.

Ho, T.; and Chan, T. "Estimating Service Time in Software Maintenance: Application-Specific and General-Purpose Human Capital," UCLA Working Paper, 1995.

Hodges, P. "IS budget growth slides." Datamation, Volume 35, April 1 1989, pp. 18-22.

Jones, C. Programming Productivity. New York, New York: McGraw-Hill, 1986.

Jones, C. "Software enhancement modelling." Journal of Software Maintenance, Volume 1, September 1989, pp. 91-100.

Jones, C. Applied Software Measurement. New York, New York: McGraw-Hill, 1991.

Kafura, D.; and Reddy, G. "The use of software complexity metrics in software maintenance." IEEE Transactions on Software Engineering, Volume 13, March 1987, pp. 335-343.

Kemerer, C., "An empirical validation of software cost estimation models." Communications of the ACM, Volume 30, May 1987, pp. 416-429.

Kriebal, C. "Evaluating the quality of information systems." In N. Szyperski and E. Grochla (eds.), Design and Implementation of Computer Based Information Systems, The Netherlands: Sijthoff & Noordhoff, 1979, pp. 29-43.
Kriebel, C.; and Raviv, A. "An economics approach to modeling the productivity of computer systems", Management Science, Volume 26, March 1980, pp. 297-311.
Lehman, M. M.; and Belady, L. A. Program Evolution: Processes of Software Change. London, England: Academic Press, 1985.
Lientz, B.; and Swanson, E. Software Maintenance Management. Reading, Massachusetts: Addison-Wesley, 1980.
Littman, D. C.; Pinto, J.; Letovsky, S.; and Soloway, E. "Mental Models and Software Maintenance." In G. M. Olson, S. Sheppard, and E. Soloway (eds.), Empirical Studies of Programmers. Norwood, New Jersey: Ablex Publishing, 1986.
Nosek, J. T.; Palvia, P. "Software maintenance management: Changes in the last decade." Journal of Software Maintenance, Volume 2, September 1990, pp. 157-174.
Pennington, N.; and Grabowski, B. "The tasks of programming." In J.-M. Hoc, T. R. G. Green, R. Samurcay, and D. J. Gilmore (eds.), Psychology of Programming. New York, New York: Academic Press, 1990, pp. 45-62.
Stabell, C. "Office productivity: A microeconomic framework for empirical research." Office Technology and People, Volume 1, 1982, pp. 91-106.
Swanson, E.; and Beath, C. M. Maintaining Information Systems in Organizations, New York, New York: John Wiley and Sons, 1989.
Tenny, T. "Procedures and comments vs. the Banker's algorithm." SIGSCE Bulletin, Volume 17, September 1985, pp. 44-53.
Tversky, A.; and Kahneman, D. "The Framing of Questions and the Consistency of Response." In R.M. Hogarth (ed.), New Directions for Methodology of Social and Behavioral Science, San Francisco, California: Jossey-Bass, 1982.
Yau, S.; and Collofello, J. "Design Stability Measures for Software Maintenance." IEEE Transactions on Software Engineering, Volume 11, September 1985, pp. 849-856.

APPENDIX

Measurement of extent of system deterioration
The extent of system deterioration is obtained in the following two steps:

1. Measure the structure and documentation quality by judging only the modified and/or added codes and comments in the program on a scale of 0 to 10. 0 implies codes that are not structured nor commented at all. 10 implies that the new codes are well-structured and documented with useful comments that can aid future maintenance task.

A program is judged for its structure based on these criteria (Banker, Datar, Kemerer, and Zweig, 1993):

(i) Each module in the program is a logical unit of operation that can be understood and modified relatively independent of other modules (i.e. low coupling between modules but high cohesion within a module).

(ii) It does not contain excessive number of GOTO statements.

Documentation quality of a program is judged according to these criteria:

(i) Readability. That is, it is written in a clear and concise manner and can be interpreted easily.

(ii) Usability. That is, the documentation is useful to a fellow maintainer. For example, a comment that simply copies the procedure declaration is redundant whereas a comment that describes which arguments serve as input and which serve as output is useful.

2. Take the difference between the most ideal system structure and documentation quality (10) and the value obtained in step 1.

As the judgment of structure and documentation quality could be subjective, we perform an inter-rater reliability test among 3 independent raters based on a random samples of 30 programs submitted by the students. Cronbach's alpha is 0.862, suggesting that the measure used is stable and consistent.