Applications: Operations and Management

Lloyd Fosdick* Editor

Some Factors Affecting Program Repair Maintenance: An Empirical Study

IRIS VESSEY and RON WEBER *University of Queensland, Australia*

ABSTRACT: ***An empirical study of 447 operational commercial and clerical Cobol programs in one Australian organization and two U.S. organizations was carried out to determine whether program complexity, programming style, programmer quality, and the number of times a program was released affected program repair maintenance. In the Australian organization only program complexity and programming style were statistically significant. In the two U.S. organizations only the number of times a program was released was statistically significant. For all organizations repair maintenance constituted a minor problem: over 90 percent of the programs studied had undergone less than three repair maintenance activities during their lifetime.***

1. INTRODUCTION

The focus of recent research has been structured programming [13]. Previously the concerns were modular programming methodologies, use of decision tables, test data generators, automatic flowcharters, etc. [16]. To date the research on methods to improve program quality and lower program development, implementation, and maintenance costs has been primarily theoretical.[1]

Most of the developed theories have been normative, that is, they stated what *should* be done to improve the quality of programs and the programming process. Unfortunately these theories have rarely been subjected to empirical testing, and so their value remains unknown.[2] They provide the zealots with opportunities to market a rash of seminars and courses and to flood the literature with papers advocating the new technologies. When the theories are subjected to testing, what little evidence has been obtained sometimes suggests that the claimed benefits, in fact, may not exist [15, 20].

This paper describes three empirical studies of factors purported to affect the extent of repair maintenance carried out on programs. By repair maintenance we mean maintenance needed to correct logic errors discovered in a program after it has been released into production. These logic errors arise because program specifications are implemented incorrectly when the program is first written, or as the consequence of maintenance carried out incorrectly after the initial production release. We distinguish repair maintenance from adaptive maintenance and pro-

[1] We use the term "theory" here very loosely.

[2] For example, we know of only a few controlled experiments where attempts have been made to investigate rigorously the claims made for structured programming, e.g., [20]. Usually any empirical evidence provided to support structured programming reports the success of some project such as the New York Times Project. However, the projects cited often have a number of different variables changed, for example, the project management method, the way the programming team was organized, and the types of programmers employed on the project. The effects of these factors confound, so there is no way of sorting out individual effects. (See also [19].)

* Editor of record. Howard Morgan is the former editor of the department, of which Alan Merten is the current editor.

Authors' Present Address: Iris Vessey and Ron Weber, University of Queensland, Department of Commerce, St. Lucia, Queensland, Australia 4067.

0001-0782/83/0200-0128 75¢

ductivity maintenance [11, 12]. Adaptive maintenance permits a program to evolve to better meet user needs. Productivity (perfective) maintenance seeks to improve the efficiency with which a program consumes resources.

The paper proceeds as follows. First, we articulate the hypotheses tested in the studies and briefly discuss the theoretical, empirical, and popular bases that exist in support of these hypotheses. Second, we discuss the data collected and the results obtained in an Australian study. Third, we discuss the data collected and the results obtained in two U.S. studies. Fourth, we examine the implications of the results. Finally, we present our conclusions and identify several directions for further research.

2. HYPOTHESES

The amount of resources expended on maintenance is a significant proportion of the total life cycle costs of a system. Estimates of the amount vary, but 40–75 percent[3] is a common range. (See [12] for a brief survey of this research.) Although empirical studies on maintenance costs are not widespread, if the above estimates of costs are accurate, seeking to reduce maintenance costs is a laudable objective [4].

Strategies for reducing maintenance costs can be formulated only with an understanding of what factors affect maintenance costs. The following sections briefly discuss some factors believed to affect the extent of program repair maintenance carried out. We chose to investigate these factors for three reasons. First, there is some support either in the literature, prior research, or among practitioners for these factors being important determinants of repair maintenance activities. Second, the factors are global in nature. We prefer to focus on a few major variables rather than on a multitude of low level variables, where there is more uncertainty about the existence and direction of effects. Third, the factors chosen are uncorrelated. The existence, direction, and magnitude of effects were studied using the general linear statistical model [17]. We attempted to avoid the problems that arise with the model when independent variables in the model are correlated. To make our beliefs explicit, the relationships between the factors chosen and the extent of repair maintenance are stated as formal hypotheses to be investigated empirically.

2.1 Effects of Program Complexity

We expect repair maintenance to be a function of system complexity. There is both theoretical and empirical support for this belief. The theoretical support comes from general systems theory—more complex systems experience greater entropy [2, 6]. Complexity is a function of the number of interfaces in a system. Simon [21] argues that systems that minimize the number of interfaces between their subsystems tend to survive longer. As open systems, programs must import energy from their environment to arrest entropy. This negative entropy takes the form of maintenance [5].

Thayer et al. [22] also obtained empirical support for a relationship between program complexity and maintenance, but their results varied considerably across the programs they studied. To further test the generality of their findings, we sought support for the following hypothesis:[4]

H1: More complex programs experience more repair maintenance.

2.2 Effects of Modular and Structured Programming

The proponents of modular programming claim it permits easier and more complete debugging.[5] Programs written in a modular manner should experience less repair maintenance because they have been tested more thoroughly before being released into production [3]. Modular programs are supposedly easier to maintain; thus there should be fewer logic errors introduced into a program whenever it is modified for any reason.

The modular programming discipline has been extended and formalized in the structured programming approach to writing programs. A major objective of structured programming research is to develop formal proofs of program correctness [7, 10]. If the underlying theory is correct, structured programs should experience less repair maintenance than modular programs, and modular programs should experience less repair maintenance than unstructured (convoluted) programs.

In our investigation we sought evidence to support this postulated relationship by testing the following hypotheses:

H2(a): Structured programs will experience less repair maintenance than modular programs.

H2(b): Modular programs will experience less repair maintenance than unstructured programs.

2.3 Effects of Programmer Quality

In our conversations with practitioners we have often heard it said that the quality of the programmer who initially wrote a program is a significant factor affecting the program's subsequent repair maintenance history. Presumably, higher quality programmers write programs with fewer logic errors, test their programs more thoroughly, and write programs that are easier to maintain. Weinberg [23] emphasizes the importance of the individual characteristics of programmers as basic determinants of program quality. After his study of errors found in IBM's DOS/VS Release 29 operating system, Endres [8] concluded the quality of the individual programmer was an important factor affecting the amount of maintenance needed by a module within the operating system. We tested the following hypothesis:

H3: Higher quality programmers will produce programs requiring less repair maintenance.

2.4 Effects of Frequency of Maintenance

A program may undergo maintenance for three reasons: (1) to repair a logic error, (2) to modify the program so that it better meets user needs, and (3) to improve its operational efficiency. The maintenance process itself is a cause of further repair maintenance being needed. After a study of modifications to large scientific batch programs, Boehm [1] reported that even with small modifications (involving changes to less than 10 statements), the chances of a successful first run after modification were,

[3] Care should be taken to show the extent of maintenance costs whenever percentage figures are given. For example, in a mature installation the amount of new development work to be done may be very small; thus maintenance costs would be a high percentage of the total operational costs of the installation. However, it does not follow that the installation is experiencing a severe maintenance problem.

[4] Note, the hypotheses are stated in a "natural" form rather than the traditional null form.

[5] Again, the theoretical basis for this claim is the notion that modular programs are simpler because they have fewer, better-defined interfaces between their subsystems.

at best, about 50 percent. If 50 or more statements were changed, the chances of a successful first run dropped below 20 percent. Consequently, in our study we tested the following hypothesis:

H4: The extent of repair maintenance increases as a program is modified more frequently (maintenance of any type).

2.5 Effects of Program Age

As a program gets older, we expect less repair maintenance. Presumably, when a program is run more times, a greater number of logic paths are exercised, and any logic errors existing in the program are discovered and corrected. Thus, we hypothesize that the number of production runs of a program between repair maintenance activities will increase exponentially as the program gets older. We tested our belief using the following hypothesis:

H5: The number of production runs between repair maintenance R_n and repair maintenance R_{n+1} will be greater than the number of production runs between repair maintenance R_{n-1} and R_n.

3. THE AUSTRALIAN STUDY

The first test of our hypotheses was carried out using data collected on all Cobol production programs in a medium-sized Australian installation. The installation was mature; it had been started in May 1966. A variety of scientific and business applications were processed, and there were batch, online, and data communications systems. During the period of our data collection the installation was carrying out exploratory work with a database management system. The installation had 34 staff members: 4 management, 8 system analysts, 6 system analyst/programmers, and 16 programmers.

For control purposes we chose initially to focus only on a single installation and only on production,[6] business, or clerical application programs written in Cobol. The objective was to reduce the possibility of confoundings in our experimental design. For example, the repair maintenance profiles of programs in two installations may differ because of different organizational and management philosophies. Similarly, the type of programming language used may affect repair maintenance.

The installation we investigated had a reputation for being well managed and innovative. It established and enforced standards early in its history. Moreover, it experimented with and used a variety of management, system analysis, and programming aids: PERT, decision table preprocessors, flowcharters, test data generators, logic path monitors,[7] librarians, etc.

3.1 Data

The source of the data was a maintenance sheet included in all program folders. It was compulsory for programmers to complete this sheet whenever maintenance was carried out. The sheet documented, among other things, the nature of the maintenance carried out and the date and time that the new version of the program was released. In total we obtained data on 200 programs[8]—the entire number of operational commercial and clerical application programs in the installation.

3.1.1 Repair Maintenance. The major dependent variable in the study was repair maintenance. By examining the program maintenance sheets, we identified those instances of maintenance that involved repairs and the dates that the repaired program was rereleased for production.

For the purposes of the study the number of repairs was inadequate as the dependent measure. In light of our hypotheses, we believed the number of times a program was run in production affected the likely number of repairs carried out on the program. Logic errors are discovered during production running; thus, even though two programs may have been released for the same elapsed time, we would expect the program that is run more often to have a greater chance of having its logic errors discovered. Consequently, we defined the dependent variable to be the repair maintenance rate: the number of repairs carried out divided by the number of production runs of the program. The operator's instructions and the operations schedule showed how often a program was run. Ultimately the repair maintenance rate must be a major variable of interest to management: how many production runs of a program can be expected before a repair will need to be carried out? Elapsed time between repairs is not very meaningful when comparing a program run annually with a program run four times a day.

3.1.2 Program Complexity. Recently there have been several attempts to formalize the notion of program complexity [9, 14]. However, because the time required to collect data was substantial and we considered our research to be exploratory, we exercised our judgments in assigning each program a complexity rating.[9]

Programs were classified as simple, moderately complex, or complex. We considered the following parameters when making our judgments. First, the number of source statements in a program was estimated. We tended to classify programs having less than 300 source statements as simple, those having 300–600 source statements as moderately complex, and those having over 600 source statements as complex.[10] Second, we examined the relative size of the data division and the procedure division in a program. Programs having a large data division and a small procedure division tended to be rated downward in complexity. Similarly, to the extent that many of the source statements were comments, a 400 source statement program, for example, might have been classified as simple. Third, we examined the number of logic paths through the program. Programs having more logic paths were rated higher in complexity. In this respect update programs tended to rate higher in complexity than validation (edit) programs, and validation programs tended to rate higher than report programs. Fourth, we considered other factors that we believe affect a program's complexity, for example, the number of files it handles, the number of fields in its records, its core size, whether or not it is an online or a batch program, and the complexity of its file structures.

[6] To the extent that the repair maintenance profiles of programs that have been retired are different from production programs, our results will be biased. It may be, for example, that programs are retired because of a poor repair maintenance history.

[7] By a logic path monitor we mean a program that flags the various logic paths in the subject program and indicates which of these paths have not been traversed by test data.

[8] We did not purposely select 200 programs; it so happened there were exactly 200 operational Cobol commercial and clerical programs in the installation.

[9] Each of us has over 10 years experience in data processing and together 6½ years working in practice. Thus we believe our judgments to be at least reasonably sound.

[10] There appears to be a reasonable correlation between program length and the McCabe and Halstead metrics [5].

In some cases the program specifications prepared by the system analyst contained a complexity rating for the program, and we could compare our ratings with this rating. Also, we asked the project managers responsible for the programs to judge the complexity of the programs according to their own criteria. Initially, we were nervous about the project managers' ratings since we felt their ratings may be *ex post* ratings, that is, ratings affected by the repair history of the program rather than ratings based on the characteristics of the program when it was first released into production. In general, however, rating consensus was high, and to the extent possible we reconciled our differences.

3.1.3 Design and Coding Discipline Used. At first we attempted to classify programs in the installation as being unstructured, modular, or structured; however, only a few programs had been written strictly according to a top-down design and structured programming discipline. Thus we classified programs as being unstructured or modular; those structured programs existing were classified as modular. Consequently, the Australian data allows us to test only hypothesis 2(b): modular programs will experience less repair maintenance than unstructured programs.

Again, we exercised our judgment on whether a program was unstructured or modular. We attempted to identify whether the major functions in a program had been organized into logical units and coded as sections or subroutines within the program.[11] We tried to the extent possible to identify those programs that superficially appeared modular because of well-documented code but were, in fact, unstructured.

As with complexity we checked our judgments on whether a program was unstructured or modular with the judgments of the project manager responsible for the application programs. Again, consensus was high and we attempted to reconcile differences.

3.1.4 Programmer Quality. Two managers within the installation rated the quality of the programmers who prepared the programs in the study as either average or good. One manager was in charge of all programmers within the installation and had final responsibility for the quality of all programs. The other manager had responsibility for the final production release of all systems and programs within the installation. Both managers had been with the installation a long time: 10 years and 13 years, respectively. Both had substantial knowledge of the programs, the programmers, and the programmers' work within the installation.

We asked the managers to rate the programmers on their ability to produce high quality programs. The notion of "quality" was left undefined. The managers were provided with a list of the programmers who had written the programs in the study and the dates at which they had written the programs. They were asked to make an independent judgment on programmer quality first, and then to compare their ratings and to reconcile any differences. They were also asked to consider possible changes in their rating of the quality of a programmer as the programmer gained more experience. Since the managers knew when a programmer wrote a program, they could assess the quality of a programmer at a particular date.

[11] For example, we looked for a mainline section that contained primarily PERFORM statements.

TABLE I. Relative Frequency (Percent) of Number of Repair Maintenance Activities for One Australian Organization and Two U.S. Organizations

Number of repairs	Australian organization	U.S. organizations	
		A	B
0	55.5	63.0	72.0
1	18.5	22.0	13.0
2	16.0	7.1	9.0
3	1.0	6.3	4.0
4	4.5	...	1.0
5	...	...	...
6	1.0	0.8	1.0
7	1.5	...	...
8	0.5	...	...
9	1.0	...	...
10	...	...	...
11	...	...	...
12	...	0.8	...
21	0.5	...	...
	100.0	100.0	100.0

3.1.5 Number of Production Releases. The number of production releases of a program was needed in order to determine whether more frequently modified programs experienced a higher repair maintenance rate. The program maintenance sheets showed the date of each production release of a program. In a few cases a program had been completely rewritten at some date. Whenever this occurred we counted this date as the initial production release of the program.[12]

3.1.6 Number of Production Runs Between Repairs. To obtain the number of production runs between repairs, we calculated the elapsed time between successive repairs and determined the number of production runs that would have occurred during this elapsed time. There are some inaccuracies in this calculation. We do not know how long a program was under repair; thus our estimates overstate the number of production runs between repairs. Because different programs were under repair for different times and the frequency of production runs varied, our estimates have varying accuracy. Similarly, programs were not in production during times of adaptive and productivity maintenance. We were unable to determine the extent to which estimates should have been adjusted to take these periods into account.

3.2 Data Analysis[13]

Table I shows one striking attribute of the programs studied in the Australian organization, namely, the small number of times that the programs underwent repair maintenance. Interestingly, 55.5 percent of the programs experienced no repair maintenance, and 90 percent of the programs experienced two or fewer repair maintenance activities. On the average, repair maintenance constituted 11.13 percent of the production releases of a program.[14]

[12] Basically we believe the program is "new" at this date. However, we recognize that the programmer who rewrites the program has the benefit of a prior version of the program. We do not know the extent to which this biases our results.

[13] In the interests of brevity and readability, the statistical hypothesis testing and estimation described in this paper is a considerably shortened version from that in our initial report. Table I shows a skewed dependent variable, which presented problems for our analysis, that we attempted in various ways to overcome. The interested reader can contact the authors for a copy of the report containing the full statistical analysis.

[14] In fact, this figure is an overstatement since in some cases adaptive and productivity maintenance were carried out at the same time as repair maintenance.

The average repair maintenance rate was 2.35 repairs per hundred production runs.

To test the first four hypotheses listed in Sec. 2, an analysis of covariance (ANCOVA) model was fitted to the data. The dependent variable was the repair maintenance rate. There were three factors: (1) program complexity measured at three levels, (2) programming style measured at two levels, and (3) programmer quality measured at two levels. The covariate was the number of production releases. Only two factors were significant at the 0.05 level: program complexity ($F = 7.16$, $df = 2/187$, $p < 0.001$) and programming style ($F = 4.85$, $df = 1/187$, $p < 0.03$). Thus we have support only for hypotheses 1 and 2(b). Overall, the factors and covariate accounted for 7.8 percent of the variance in the repair maintenance rate.

To determine the practical significance of the statistically significant factors, we undertook estimation of the differences between factor-level means using the Bonferroni method of multiple comparisons and a 0.90 family confidence coefficient [17]. In terms of the program complexity factor, moderately complex programs had between 0.27 and 5.73 more repairs per hundred production runs than simple programs; furthermore, complex programs had between −0.22 and 5.88 more repairs per hundred production runs than simple programs, and between −3.23 and 2.89 more repairs per hundred production runs than moderately complex programs. In terms of the programming style factor, use of a modular style instead of an unstructured style reduced the repair maintenance rate by between −1.12 and 3.7 repairs per hundred production runs.

To test hypothesis 5, we used a single factor analysis of variance (ANOVA) model. The dependent variable was the number of production runs between successive releases of a program where program maintenance had been carried out. The independent variable was the number of the time period between successive repair maintenance activities, that is, time period 1 was the time period between the initial production release and that after the first repair maintenance activity, time period 2 was the time period between the production release after the first repair maintenance activity and the production release after the second repair maintenance activity, etc. Since only one program had more than nine repair maintenance activities and the ANOVA model needs at least two observations per time period, we could test hypothesis 5 over 9 time periods.

The F test for equality of the factor-level means was significant only at the 0.4511 level. We conclude there are no differences between any of the factor-level means, and we must reject hypothesis 5.

4. THE U.S. STUDY

Because the results from the first study were contrary to expectations, we replicated the research to determine whether the results appeared to hold generally. Cobol programs in two U.S. organizations were studied. Both organizations were large and mature: organization A had over 240 analysts and programmers, and organization B had over 40 analysts and programmers.

4.1 Data

Data was collected on 127 programs in organization A and 100 programs in organization B. The programs did not constitute the whole set of production programs in both organizations. Unfortunately, the needed data had been recorded routinely by the organizations only in recent years. The studied programs were developed and implemented from about 1975 onward.

There were two differences between the data obtained in the Australian study and the data obtained in the U.S. study. First, the programs examined in the U.S. study, on the average, were less complex than those examined in the Australian study. Hence, it was not possible to use the same complexity ratings as were used in the Australian study. Instead of classifying programs with 1-300 source statements as simple, 301-600 as moderately complex, and over 600 as complex, the upper category limits were established at 150, 300, and 450 procedure division statements. Again, adjustments were made to this initial complexity rating based on the number of logic tests, number of files handled, etc. This difference between the two studies affects the comparability of results, but it enables the effects of program complexity to be examined using a finer measurement scale.

The second difference between the two studies relates to the distribution of programming styles used. Since data could be collected only on the more recently developed programs within each U.S. organization, the programming style used was primarily a modular or a structured style. Both organizations had enforced standards aimed at eliminating unstructured code. The smaller size of the programs relative to those in the Australian study reflects these standards. Thus the programming style factor has only two levels: modular and structured. Again, this affects the comparability of the Australian and U.S. results; however, a test of hypothesis 2(a) now could be performed.

4.2 Data Analysis

Table I shows the relative frequency of repair maintenance activities for the two U.S. organizations. Note the similarities between the Australian and U.S. data; most programs experienced only a small number of repair maintenance activities. In organization A, repair maintenance activities occurred, on the average, 0.823 times per hundred production runs and constituted 18.8 percent of production releases. The corresponding figures for organization B are 0.627 and 21.43 percent.

To test the first four hypotheses listed in Sec. 2, an ANCOVA model was again fitted to the data. Only the covariate was significant at the 0.05 level in both cases ($F = 13.86$, $df = 1/114$, $p < 0.001$ for organization A, and $F = 7.79$, $df = 1/87$, $p < 0.006$ for organization B). Thus we have support only for hypothesis 4. Overall, the factors and covariate accounted for 11.7 percent of the variance in the repair maintenance rate for organization A and 12.4 percent of the variance in the repair maintenance rate for organization B.

To evaluate the practical significance of the statistically significant covariate, we undertook statistical estimation of the slope of the regression line for the covariate [17]. For organization A, at the 95 percent confidence level each release of a program resulted in between 0.55 and 1.92 more repairs per hundred production runs. For organization B, at the 95 percent confidence level each release of a program resulted in between 0.22 and 1.26 more repairs per hundred production runs.

Hypothesis 5 was tested again using a one-way ANOVA model. The F test for equality of the factor level means was significant at the 0.777 level for organization A and the 0.001 level for organization B. Hence, the effect

of program age is significant for organization B only. However, the result gives only weak support to hypothesis 5. For the first four time periods between successive repair maintenance activities, the average number of production runs was 19.14, 18.33, 15.83, and 112.5. Thus the significant result is obtained because the mean for time period 4 differs considerably from the other three time periods.

5. DISCUSSION OF RESULTS

Our first conclusion from the results is that repair maintenance does not seem to constitute a very important activity in any of the three installations. Adaptive maintenance is far more important. (We noted only a few instances of productivity maintenance.) We do not know whether this conclusion holds generally, but it is apparent that certain organizations test their programs thoroughly before releasing them for production running.

In two of the three organizations studied, we found support for Boehm's [1] hypothesis that the likelihood of a successful first run after only a minor modification is small. For both U.S. organizations the number of program releases affected the repair maintenance rate. Nevertheless, the practical significance of the result might be questioned. The confidence interval for the covariate effects shows the increase in the repair maintenance rate with an extra program release to be between 0.55 and 1.92 and between 0.22 and 1.26 more repairs per hundred production runs, respectively, for organizations A and B.

The complexity factor was significant only for the Australian organization. What is surprising, however, is that we found little difference between the repair maintenance rates for moderately complex programs and complex programs. The factor is statistically significant because the repair maintenance rate for easy programs differs from the repair maintenance rate for moderately complex or complex programs. In fact, the estimate of the repair maintenance rate for moderately complex programs is slightly higher than the rate for complex programs.

We offer three possible explanations for this finding. First, our judgment on the level of complexity of a program may be inaccurate. As mentioned earlier, formal measures of program complexity are still evolving. Second, the repair maintenance rate may be a logarithmic function of complexity. At moderate levels of complexity the rate of increase of the function may be small. Third, programmers may exercise greater care when they design, code, and test complex programs. They may recognize the increased potential for logic errors and take precautionary measures as a result.

Why program complexity was not significant in the two U.S. organizations is unclear. For the three organizations studied, program complexity was significantly related to the number of releases; but the number of releases was not significant as a covariate in the Australian organization, while program complexity still was not significant in the two U.S. organizations when the covariate was excluded from the model. It seems as though program complexity and number of releases may be explaining different parts of the total variability of the repair maintenance rate. Recall that for the three organizations studied, the covariate and the factors accounted for less than 13 percent of the total variance in the dependent variable. Thus much of the variance in the repair maintenance rate still has to be "explained."

A possible reason for the conflicting results is the different ways in which program complexity was measured for the Australian and U.S. organizations. The programs in the Australian organization, in general, were more complex than those in the U.S. organizations. This also might explain the higher mean repair maintenance rate in the Australian organization: 2.35 repairs per hundred production runs versus 0.82 and 0.63 repairs per hundred production runs for the U.S. organizations. Still another explanation for the conflicting results might be that development programmers in the U.S. organizations tested their programs better than programmers in the Australian organization.

Only weak support exists for programming style having an effect on the repair maintenance rate. While this factor is significant in the Australian study, there is no evidence of an effect in the U.S. studies. On the basis of the Australian study we suspect that modular (and structured) programming has a greater impact on the repair maintenance rates of moderately complex and complex programs than simple programs. However, there was no evidence of an interaction effect to support this hypothesis. Further research might investigate this issue.

We are unable to explain the results relating to programmer quality. Perhaps the managers judged the quality of programmers incorrectly. In retrospect, however, we suspect that good programmers differ from average quality programmers on the basis of attributes other than repair maintenance, for example, the speed with which they design and implement programs, how easy their programs are to maintain, and the efficiency with which their programs run. Still the question begs: What are the attributes of a good programmer?

We found no support for the hypothesis that the number of production runs between repairs increases after each repair. One possible explanation for this result is that more adaptive maintenance must be carried out as a program gets older (to arrest entropy); consequently, though the initial logic errors are removed, new logic errors creep into the program as more adaptive maintenance is carried out.

6. CONCLUSIONS

Our study confirms the need for further empirical research in the programming area [18]. Although we found support for some of the hypotheses advanced about repair maintenance, other hypotheses still await statistically significant results. Moreover, the independent variables we examined do not account for a large percentage of the variation in the dependent variables.

It is not difficult to identify further research topics. Textbooks and articles are rife with prescription. For example, Yourdon [24] claims top-down testing reduces system testing, allows major bugs to be discovered earlier in testing, facilitates finding bugs, distributes testing more evenly through a project's life, etc. All of these propositions are testable hypotheses.

What is difficult is operationalizing the research. For example, one way of testing the claim that structured code is easier to maintain than unstructured code would be to run a controlled experiment. Two groups of programmers would code a set of programs: an experimental group would use the structured programming methodology and a control group would use an unstructured methodology. A series of modifications to the program then could be made and such variables as the time to accomplish the modifications and the accuracy of the modifica-

tions made could be measured. Unfortunately, carrying out this experiment would be difficult; it would be time consuming and costly; the experimenter would have to ensure homogenity of the quality of the programmers in the experimental and control groups; the control group should not have been exposed to structured programming in case it biased the way they wrote code; etc. Until these types of problems are overcome, it is unlikely there will be rapid developments in empirical work to support the theory of programming.

Our results stand as a challenge to some conventional wisdom and the proponents of structured programming (who include us). We readily acknowledge that our research is exploratory and there are problems with the statistical model. Nevertheless, the results are anomalous. Careful thought needs to be given to the nature of the functional relationship between different program quality measures and, say, programming style. Formal empirical work needs to be undertaken to validate the nature of the functional relationships hypothesized. Perhaps our failure to obtain statistically significant results reflects the need to develop more formal, *operational* measures of program complexity, programming style, and programmer quality. But in the case of repair maintenance rates we suspect this will do little good. Table I shows that the variability of repair maintenance rates in three organizations is almost negligible. If this result holds generally, more careful measurement of the independent variables will not account for variability in the dependent variable if there is no variability to be explained anyway! Instead, we believe that an essential prerequisite to obtaining the desired results is that, for example, the proponents of structured programming enunciate *precisely* what dependent variables will be affected by structured programming.

Acknowledgments. This paper has benefited from the comments of participants in workshops at the University of Minnesota, Purdue University, and Indiana University. We are especially indebted to Andrew Bailey and Izak Benbasat for detailed comments on earlier versions of the paper. The responsibility for the contents of this paper rests with the authors.

REFERENCES

1. Boehm, B.W. Software and its impact: A quantitative assessment. *Datamation 19*, 5 (May 1973), 48–59.
2. Buckley, W. *Sociology and Modern Systems Theory*. Prentice-Hall, Englewood Cliffs, N.J., 1967.
3. Canning, R.G. Modular COBOL programming. *EDP Analyzer 10*, 7 (July 1972), 1–14.
4. Canning, R.G. That maintenance "iceberg." *EDP Analyzer 10*, 10 (Oct. 1972), 1–14.
5. 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 SE-5*, 2 (March 1979), 96–104.
6. Davis, G.B. *Management Information Systems: Conceptual Foundations Structure, and Development*. McGraw-Hill, New York, 1974.
7. De Millo, R.A., Lipton, R.J., and Perlis, A.J. Social processes, and proofs of theorems and programs. *Comm. ACM 22*, 5 (May 1979), 271–280.
8. Endres, A. An analysis of errors and their causes in systems programs. *IEEE Transactions on Software Engineering SE1*, 2 (June 1975), 140–149.
9. Halstead, M.H. *Elements of Software Science*. Elsevier, New York, 1977.
10. Hantler, S.L., and King, J.C. An introduction to proving the correctness of programs. *Computing Surveys 8*, 3 (Sept. 1976), 331–353.
11. Lientz, B.P., and Swanson, E.B. Problems in application software maintenance. *Comm. ACM 24*, 11 (Nov. 1981), 763–769.
12. Lientz, B.P., Swanson, E.B., and Tompkins, G.E. Characteristics of application software maintenance. *Comm. ACM 21*, 6 (June 1978), 466–471.
13. Linger, R.C., Mills, H.D., and Witt, B.I. *Structured Programming: Theory and Practice*. Addison-Wesley, Reading, Mass., 1979.
14. McCabe, T.J. A complexity measure. *IEEE Transactions on Software Engineering SE2*, 4 (Dec. 1976), 308–320.
15. Myers, G.J. A controlled experiment in program testing and code walkthroughs/inspections. *Comm. ACM 21*, 9 (Sept. 1978), 760–768.
16. Naftaly, S.M., Cohen, M.C., and Johnson, B.G. *COBOL Support Packages: Programming and Productivity Aids*. Wiley, New York, 1972.
17. Neter, J., and Wasserman, W. *Applied Linear Statistical Models*. Irwin, Homewood, Ill., 1974.
18. Sheppard, S.B., Curtis, B., Milliman, P., and Love, T. Modern coding practices and programmer performance. *Computer 12*, 12 (Dec. 1979), 41–49.
19. Sheil, B.A. The psychological study of programming. *Computing Surveys 13*, 1 (March 1981), 101–120.
20. Shneiderman, B., Mayer, R., McKay, D., and Heiler, P. Experimental investigations of the utility of detailed flowcharts in programming. *Comm. ACM 20*, 6 (June 1977), 373–381.
21. Simon, H. A. *The Sciences of the Artificial*. MIT Press, Cambridge, Mass., 1969.
22. Thayer, T.A., Lipow, M., and Nelson, E.C. *Software Reliability*. North-Holland, Amsterdam, 1978.
23. Weinberg, G.M. *The Psychology of Computer Programming*. Van Nostrand Reinhold, New York, 1971.
24. Yourdon, E. *Techniques of Program Structure and Design*. Prentice-Hall, Englewood Cliffs, N.J., 1975.

CR Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution and Maintenance—*corrections*; D.2.2 [**Software Engineering**]: Tools and Techniques—*Structured Programming*
General Term: Experimentation
Additional Key Words and Phrases: program maintenance, repair maintenance, program complexity, modular programming, structured programming, programmer quality, programming management

Received 1/80; revised 4/82; accept 5/82