Management of Computing

Gordon B. Davis Editor

Determinants of Program Repair Maintenance Requirements

LEE L. GREMILLION

ABSTRACT: Considerable resources are devoted to the maintenance of programs including that required to correct errors not discovered until after the programs are delivered to the user. A number of factors are believed to affect the occurrence of these errors, e.g., the complexity of the programs, the intensity with which programs are used, and the programming style. Several hundred programs making up a manufacturing support system are analyzed to study the relationships between the number of delivered errors and measures of the programs' size and complexity (particularly as measured by software science metrics), frequency of use, and age. Not surprisingly, program size is found to be the best predictor of repair maintenance requirements. Repair maintenance is more highly correlated with the number of lines of source code in the program than it is to software science metrics, which is surprising in light of previously reported results. Actual error rate is found to be much higher than that which would be predicted from program characteristics.

1. INTRODUCTION

Whenever a piece of software is released for production, management information systems (MIS) executives make a commitment to devote resources in the future to the maintenance of that software. Some of this maintenance is unavoidable and its occurrence unpredictable since it is due to changes in user requirements or the computing environment. Unless the software is trivially simple, it will also undergo maintenance to correct errors present but undetected at the time of release. Predicting the number of such errors, and therefore, the extent of the requirement for corrective or repair maintenance, would provide management with valuable planning information.

A number of theories exist relating program characteristics to the expected occurrence of errors in the programs. This study examines those theories and hypothesized relationships between program characteristics and repair maintenance rates using data on 346 programs in a system used by a large electronics manufacturing firm. Results show that the occurrence of errors are in fact strongly related to measures of size and complexity of the programs and less strongly to the intensity with which the programs are used. Surprisingly, the actual number of errors reported against the program is larger than that predicted by software science measures. For these programs, size (numbers of lines of code) is found to be the best predictor of the number of errors remaining in a program after prerelease testing.

2. REPAIR MAINTENANCE AND FACTORS BELIEVED TO AFFECT IT

2.1 The Repair Maintenance Issue

Maintenance refers to changes made to operational programs in order to keep the programs operational and responsive to user needs. Maintenance activities can be broken down into several categories (e.g., [25, 27]) one

of which is corrective or repair maintenance. This essentially refers to fixing errors or "bugs," discovered after the program has been made operational. Other types of maintenance involve adapting programs to meet changing user needs or a changing computing environment.

Maintenance activities can account for a significant fraction of the cost and effort expended on a program during its life cycle. Lientz et al. [16] surveyed a number of studies which estimated that fraction to range between 40 and 75 percent. Less information is available specifying which part of that is repair maintenance. Lientz and Swanson [15] found it to be about 20 percent in one study of 487 data processing (DP) organizations, whereas Vessey and Weber [27] found it to be "a minor problem" without making a specific resource-expenditure estimate. Popular wisdom, as reflected in MIS textbooks [4] and MIS management publications [6] holds that the cost is significant enough to be a real management concern.

2.2 Program Complexity

Most research on factors affecting program repair maintenance has focused on the relationship between the number of bugs and some measure or measures of the "complexity" of the program. This complexity has been defined in a number of ways, most often utilizing the software science metrics developed by Halstead [14]. The basic idea is that the more complex the program or module, the more likely it is that the programmer made logic errors *and* failed to detect the errors before the module was released.

2.2.1 ***Program Size.*** A number of empirical studies have been reported using different complexity measures and with varying results. A common approach involves some measure of program size as an indicator of complexity. Lientz and Swanson [15] found that larger systems (as measured by numbers of source language statements) seemed to require more maintenance effort including debugging, as perceived by their respondents. Bell and Sullivan [2] examined a number of published algorithms and found a strong relationship between an algorithm's length and the occurrence of errors. Thayer [26], in a study of 249 modules, reported that the larger modules did experience a greater number of bugs, but he did not report correlation coefficients. Bowen [3] examined the correlations between errors and program length for 75 modules in three projects for the Department of Defense, and found correlation coefficients ranging from 0.51 to 0.91. (He found similar correlations when using McCabe's measure of cyclomatic complexity [18] as a predictor variable.) Vessey and Weber used categorical complexity measures, "simple," "moderately complex," and "complex," derived from the number of procedure-division statements and subjective evaluations as a predictor variable for 447 commercial programs in three organizations. They found only a weak relationship between this variable and the rate of repair maintenance for one organization's programs.

2.2.2 ***Software Science Program Complexity Measures.*** A complexity measure which has had some empirical support is Halstead's E—the measure of mental effort required to create a program. E is derived from two other measures of a program—*difficulty* and *volume.* A program's difficulty is a function of the number of operators used in the program and the number of times variables are manipulated within the program. As pointed out by Christensen et al. [7], it appears to be a measure of both the "ease of writing" and "ease of reading" of the program. Volume is a function of the total usage of operators and operands and the number of unique operators and operands appearing in the program. It is a measure of the number of bits required to specify the program. E is the product of these two measures: As the size (volume) and/or difficulty of an algorithm increases, so should the effort required to code it into a program. (For a more complete explanation of these and other software science metrics, see [14].)

Some very impressive results have been obtained using these measures. Funami and Halstead [12] calculated the value of E for nine modules reported by Akiyama [1] and found a 0.98 correlation between the E measurement and the reported number of errors. Fitzsimmons and Love [11] calculated E measures for 140 programs in three large General Electric software development projects, and found correlations ranging from 0.75 to 0.81 between E and the number of documented errors for the programs.

Fitzsimmons and Love [11] pointed out that a likely problem in comparing their results with those of Halstead and his colleagues was a difference in the way the dependent variable was defined. For them, "delivered bugs" meant those discovered *after* the initial round of testing. The arguments made by Halstead [14] for the relationship between E and the number of bugs refers to all bugs initially coded into the program and the correlations reported by Funami and Halstead [12] were derived on that basis. Managers interested in future repair maintenance rates will be concerned with the bugs remaining in a program after all pre-release testing and debugging has been done. One issue that is addressed in this study is the strength of the correlation between E and only those bugs which remain after formal debugging is complete.

Ottenstein et al. [21] further argued that the number of bugs coded into a program was a function of two factors: the number of mental discriminations required to code the program (E) and the average amount of work (i.e., number of mental discriminations) a programmer can do without making an error. They claim that this function can be approximated by

$$B = E^{2/3}/3000$$

where B is the predicted number of bugs in a module. They found that the predictive power of this model was supported by Akiyama's data [1] and by Bell and Sullivan's suggested maximum module size [2].

2.3 Intensity of Program Use

A second factor which might affect the occurrence of repair maintenance is the intensity with which a program is used. Musa [20] and Littlewood [17], for example, suggested that the more "stress" a program undergoes, that is, the more it is executed, the shorter the expected time to failure, the sooner a bug will be discovered and have to be fixed. Gilb [13], likewise, pointed out that one should measure program reliability not in terms of absolute number of bugs but in terms of the number of transactions with failures as a fraction of the total number of transactions. The idea here is that the more a program is exercised, the more likely it is that the logic path with the hidden bug will be taken, sooner rather than later. Thus, all other things being equal, a program which is run more frequently would be expected to have a higher incidence of repair maintenance than one which is run less frequently. Vessey and Weber [27] cite this logic in using repair maintenance rate (the number of repairs carried out divided by the number of production runs) for a program as their dependent variable.

2.4 Program Age

A related but slightly different issue is that of program age. Vessey and Weber [27] reflected the common belief that the rate of discovery of bugs declines as the program grows older—fewer and fewer untried logic paths remain. On the other hand, it is only with the passage of time that some of these logic paths will be tried, when certain unusual circumstances arise. For example, there is the (possibly apocryphal) story related by Moore [19] about the early days of the SABRE system which crashed when a reservation was attempted in which the names totaled 244 characters ending with an "*n*." This bug was not discovered until one day when an agent attempted to book a flight for the Boston Bruins hockey team. Because only unusual circumstances will activate some logic paths, one would expect to find a correlation between program age and the incidence of repair maintenance, beyond that attributable to frequency of production runs. The older a program is, the more likely it is that those rarely encountered bugs have, in fact, been encountered.

2.5 Programming Style

There is popular support for the notion that certain programming practices, specifically modular programming and structured programming, tend to reduce the number of delivered bugs in a program [5]. (Sheil [22], however, shows that support for this notion in the research literature is very weak indeed.) The basic idea is that the factoring, in the case of modular programming or formal structuring, in the case of structured programming, of the program makes it easier to understand and to debug more completely before release. These effects may be interpreted as resulting from a reduction in program complexity due to the use of the techniques. Christensen et al. [7] pointed out that the use of structured programming techniques should be captured in the Halstead difficulty measure which, in turn, affects the effort measure *E*. Breaking a program into modules reduces the number of unique operators used, which therefore reduces both difficulty and volume, reducing *E*. Thus, a larger, that is, nonmodularized or unstructured implementation of a given algorithm would be expected to require more mental discriminations (higher *E* measurement) than the same algorithm implemented using these programming techniques.

2.6 Programmer Competence

A final factor which might be expected to affect the number of delivered bugs is the competence of the programmer who wrote the program. Published findings on this question are sparse and conflicting. Endres [10] found evidence that programmer quality was an important determinant in the number of bugs in a release of an IBM operating system. Vessey and Weber, on the other hand, were unable to find any relationship between programmer quality and repair maintenance in their study [27]. Unfortunately, the data used in this study was captured for other purposes, and does not include information on the competence or experience of the programmers involved. This issue, therefore, will not be addressed, except to recognize it as a possible confounding variable.

2.7 Hypotheses

Several hypotheses can be identified which this study will test. These are:

- *H1*: The more complex a program is, in terms of size (lines of code or volume), difficulty or Halstead *E*, the greater the number of errors the program will contain when released.
- *H2*: The more intensively a program is used, the more errors will be discovered in it.
- *H3*: The older a program is, in terms of time since release, the more errors will have been discovered in it.

Each of these can be stated in the form of a null hypothesis; that is, there is no statistically significant relationship between number of delivered bugs in a program and any of the other factors mentioned. An additional prediction to be tested, although it is not in hypothesis form, is that the actual number of delivered bugs discovered in the programs will be related to but less than the number derived from the formula by Ottenstein et al. [21].

3. RESEARCH METHODOLOGY

To test these hypotheses, an analysis was performed on the 346 programs making up a manufacturing support system (manufacturing database maintenance and requirements planning) used by a large electronics equipment manufacturer. This system was developed and maintained by a central programming group which developed and maintained other systems as well. It is currently installed in 28 locations worldwide (all in-

TABLE I. Complexity Measurements and Repair Requests for Programs Included in the Analysis*

Measurement	Mean	SD	Minimum	Maximum
Halstead Volume (Kbits)	45.5	52.7	0.4	290.1
Halstead Difficulty	70.2	67.4	3.0	1,129
Halstead *E* (000)	4,769	8,521	1.6	81,965
Lines of Code (excluding comments)	1,173	1,168	51	6,572

* $n = 346$

house). All software maintenance done, for whatever reason, is performed by the central programming group. For repair maintenance, the user who discovers an error submits a formal request to this group for correction of the error.

The programs themselves are written in PL/I and vary in length from 51 to 6,572 source statements. All were written in a highly structured style according to the organization's programming standards. Halstead metrics were computed for each of these programs by means of a program which took as input the PL/I source code. It counted the number of unique operators and operands and the number of total occurrences of operators and operands along with the total lines of code for each program. Further, Halstead metrics were computed from these counts according to the procedures described in [14]. Table I summarizes this information.

The Halstead measurements were taken at one point in time, and, therefore, do not reflect changes in a program's complexity due to adaptive or perfective maintenance. The few programs which were so substantially rewritten as to significantly change their complexity measures were from the analysis. Changes in complexity over time for the remaining programs should have been small, and should not have biased the results.

User records were examined to determine the frequency of use of each program. Table II shows the categorizations which were made and the distribution of programs by category. Although it would have been more desirable to use a continuous measure of frequency of use (e.g., number of times the program was run per month), this was not possible. The different system users kept their usage records in different formats and at differing levels of detail, so that only categorizations as shown in Table II could be made accurately.

TABLE II. Distribution of Programs by Frequency of Use

Level	Description	N	%
0	Used rarely (monthly or less)	113	32.7
1	Used at least monthly but not daily	99	28.6
2	Usually used once per day	84	24.3
3	Used several times per day	28	8.1
4	Used many times per day	22	6.4
	Total	346	100.0

TABLE III. Distribution of Programs by Age (Number of Years since Initial Release)

Age	N	%
Less than one year	7	2.0
At least one year but less than two	4	1.2
At least two years but less than three	9	2.6
At least three years but less than four	1	0.3
At least four years but less than five	325	93.9
Total	346	100

Age of the programs was obtained from records showing the date of initial release. Table III shows the distribution of programs by age. One shortcoming in this data becomes apparent from an examination of Table III—the lack of variation in program age. This reflects the fact that most of the programs were initially released as a group when the system as a whole was released. This severely limits the extent to which the effects of program age on repair maintenance (*H3*) can be tested.

The dependent variable is the total number of repair requests made for each program over its life. (One repair request represents one bug to be fixed.) This was obtained from records maintained by the programming group. For the 346 programs included in the study, the number of requests per program ranged from zero to a high of 268 with an average of 16.8 per program, and a standard deviation of 31.7.

4. ANALYSIS

4.1 Correlations Among Variables

The first step in the analysis of the data is to look at the paired (zero-order) correlations among variables as shown in Table IV. Several points should be noted in this data. The correlation between volume and difficulty is significant, but not perfect, suggesting that these measure related, but different, aspects of the program. The number of lines of code is more highly correlated with volume than with difficulty, lending support to the notion that a longer program is not necessarily a more difficult one. Age is not very highly correlated with anything, including the number of repairs. The apparently significant negative correlation between age and difficulty is probably spurious due to the limited variability in age. Frequency of use is significantly correlated with the number of repairs, as anticipated, but is also significantly correlated with volume, difficulty, *E*, and lines of code. These latter correlations are unexpected, and may indicate some coincidental patterns in the programs, for example, that the biggest and most difficult programs happen to be run fairly frequently.

The data lends support to *H1*, that the number of delivered errors increases as program complexity in-

TABLE IV. Paired (Zero-Order) Correlations among Variables**

	Vol	Diff	E	Loc	Age	Freq
DIFF	0.44*					
E	0.85*	0.79*				
LOC	0.97*	0.47*	0.82*			
AGE	0.05	−0.18*	−0.07	0.04		
FREQ	0.19*	0.20*	0.21*	0.22*	0.05	
REPAIRS	0.72*	0.25*	0.57*	0.74*	0.08	0.27*

* $p < 0.001$ ** $n = 346$

creases. The number of repairs is significantly correlated with E, although at a lower level than would be expected from previously reported findings. There is also a significant, but somewhat weak correlation between difficulty and number of repairs. Most striking, however, is the correlation between the measures of size of the program—volume and lines of code—and the number of repairs. It would appear that for these programs, lines of code would be the best measure of complexity to use for predicting repair requests.

4.2 Regression Model

Regression analysis lets us look at the combined effects of these variables. To do this, a series of regression analyses were run, in which the dependent variable was number of repairs. All the possible combinations of difficulty, volume, lines of code, E, program age, and frequency of use were used as independent variables. Table V shows the "best" regression model that could be built from the data predicting repair requests as a function of the independent variables mentioned above. This model is best in the sense that substituting any other measure of program complexity for number of lines of code reduced the overall R^2. Also, if any other complexity measure was added to the equation along with number of lines of code, the model's explanatory power was not significantly enhanced. Program age was not a significant variable in any formulation. The variability in the two measures used accounted for 56 percent of the variability in the number of repair requests. Although both independent variables were significant at the 0.001 level, most of the explanatory power of the model lies in the lines of code measure. As was shown in the correlation matrix, there was statistical support for *H2*, that the number of errors discovered increases with increasing intensity of program use, but operationally, this relationship was weak.

Beta in Table VI was the coefficient in the regression equation for each variable. It was the amount the dependent variable changes for each unit change in the independent variable. Since frequency of use was an ordinal variable, its actual coefficient value was less meaningful than is the coefficient for lines of code, which was a ratio-level variable [23].

TABLE V. Results of Least-Squares Regression (Dependent Variable = Number of Repair Requests)

Variable	Beta	t Statistic
Constant term	−9.85	−5.2*
Number of lines of code	0.0194	19.5*
Frequency of use	3.12	3.2*

$R^2 = 0.56$ $F = 218.5$ (Significance of $F < 0.001$)
* Significance of $t < 0.001$

TABLE VI. Results of Least-Squares Regression (Dependent Variable = Number of Repair Requests per 1,000 Lines of Code)

Variable	Beta	t Statistic
Constant term	4.46	3.9*
Number of lines of code	0.0017	2.84**
Frequency of use	3.87	6.5*

$R^2 = 0.15$ $F = 30.7$ (Significance of $F < 0.001$)
* Significance of $t < 0.001$ ** Significance of $t < 0.01$

4.3 Program-Error Characteristics

The number of bugs in these programs, as reflected by the number of repair requests, was significantly greater than that predicted by Ottenstein et al. [21]. Actual repair requests showed a total of 5,822 bugs (an average of 16.8 per program) while the predicted number was 2,613 (an average of 7.6 per program). These results were particularly surprising in light of our expectation that the actual number of discovered bugs should be, if anything, *less* than the predicted number due to some having been caught in pre-release testing. Instead, it appears that the larger the program (number of lines of code), the greater the error in the prediction. The correlation coefficient between number of lines of code and the difference between actual and predicted number of bugs was 0.58, which was significant at the 0.001 level.

Four possible explanations for this discrepancy come to mind. The first is that the formula proposed by Ottenstein et al. is incorrect for programs such as the ones studied here. Size of the programs may be a factor—there are some very large programs in this set, and it is for these large programs that the discrepancy is greatest. It may be that programs become more error prone as they increase in complexity at a rate even faster than that predicted.

A second possibility is that we are seeing the effects of uncontrolled intervening variables. Ottenstein et al. point out that the relationship they postulate may be confounded by factors such as programmer experience, method of programming, and amount of machine time available for testing. Since programmer characteristics and test time are not measured in this study, their effect is impossible to determine.

TABLE VII. Frequency of Repair Requests

Number of Repair Requests	Frequency	%
0	59	17.1
1- 5	127	36.7
6-10	32	9.2
11-20	42	12.1
21-30	30	8.7
31-40	15	4.3
41-50	9	2.6
51-100	26	7.5
101-200	3	0.9
201-300	3	0.9
Totals	346	100.0

Third, it may be that some of the repairs were to fix bugs introduced by previous maintenance efforts. In particular, one would expect some bugs to be introduced by adaptive maintenance which added code to a program. Since extensively rewritten programs were excluded from the data set, however, it seems unlikely that this would account for such a large discrepancy between predicted and actual bugs.

Finally, there is the possibility that some of the "repair requests" were actually requests for adaptive maintenance (i.e., changes to the program's function, not fixes). As Swanson [24] points out, MIS organizations tend to resist requests for repair maintenance less than they do requests for adaptive maintenance, leading users to try to disguise their requests for changes as requests to fix bugs. While there is no evidence that this has occurred with the programs included in this study, it is a possibility that must be recognized. Clearly, this is an area which merits further study.

We can look at the distribution of frequency of repair requests as shown in Table VII. While a number of programs (59) have had no repair maintenance at all, most have had some repairs, and almost half have had more than five repair requests. For the 346 programs as a group, there were almost 6,000 repairs requested over a 4½-year period. These findings are in sharp contrast to those reported by Vessey and Weber [27]. One possible explanation for this discrepancy would be that Vessey and Weber studied systems with much shorter programs. This is supported by the fact that they classify programs with more than 600 source statements as "complex," while the average length of programs included in this study was 1,168 lines of code.

5. CONCLUSIONS

In drawing conclusions from this data, one must remember that it represents only one particular situation, utilizing one programming language and style, and it would be improper to generalize beyond this situation. Still, it is instructive to look at the results obtained from particular cases in light of predictions which have been made concerning factors affecting the occurrence of delivered bugs. Only through the examination of empirical evidence can the theories be tested and refined.

The most surprising result was the large number of bugs discovered in the programs relative to the number of bugs which would be predicted by the formula developed by Ottenstein et al. Unfortunately, measures of possible intervening variables which may account for this difference—programmer characteristics and the extent of adaptive maintenance performed on the programs—were not available in the data used. Given the strong correlations reported by Ottenstein et al. in support of their model, and the potential usefulness of that model, the need for further investigation is clearly indicated.

The data analyzed above agree with the widely held view that the number of delivered bugs in a program is strongly related to the complexity of that program. Surprisingly, however, the best measure of this complexity appears to be simple count of the lines of code in the program, rather than such measures as Halstead's E, which have been used successfully in previously reported research. The correlation between E and number of repair requests, while significant, is lower than that reported in previous studies. The number of mental discriminations a programmer makes in creating a program (measured as E) may be an important determinant of the number of bugs initially coded into the program. It appears, however, that the size of the program (as measured by lines of code or volume) is more important in determining the number of those errors which are found during debugging, and, therefore, the number of remaining bugs which are delivered to the user. These findings appear to vindicate those who advocate limiting program module size in order to help reduce delivered bugs (see [2, 21]).

Frequency of use of the programs studied does not seem to be as important a predictor of repair maintenance as the literature suggests. While the correlation between frequency of use and number of repair requests is statistically significant, the amount of variation explained is small, only about 6 percent. Perhaps the fact that most of the programs were over four years old influenced this. After four years, one might hypothesize, all the programs would have been run so many times that any effect of frequency of use would be lost. This badly skewed distribution of program age also made it possible to test the hypothesis that the occurrence of repairs increases with program age.

The need is clearly indicated for further research to refine our understanding of what factors relate to delivered bugs and, therefore, to repair maintenance. Almost one half of the variability in occurrence of repair maintenance among the programs included in this study remains unexplained—despite the fact that program size, complexity, intensity of use, and age were included as predictor variables, and programming style was common across all modules. As mentioned above, other factors not included as predictor variables or controlled for in the experimental design may be at work. This is a disadvantage of using existing data captured for purposes other than the one under study—important constructs may simply not have been measured.

This problem is best addressed by studies in which constructs and the way in which they will be measured are determined before or during the time the programs are actually written. In addition, where certain factors do not vary within a particular study (e.g., programming style did not differ in this study), multiple comparative studies are required to see the effect of that factor. The justification for such studies is clear—the more we know about the factors affecting the occurrence of bugs in delivered programs, the better we should be able to predict, and ultimately, control those bugs.

REFERENCES

1. Akiyama, F. An example of software system debugging. *Proceedings of the IFIPS Congress*, 1971, 353–359.
2. Bell, D.E., and Sullivan, J.E. Further investigations into the complexity of software. *MITRE Technical Report MTR 2874*, vol. II, Bedford, Maine, 1974.
3. Bowen, J.B. Are current approaches sufficient for measuring software quality? *Proc. Softw. Quality Assurance Workshop*, 3, 5, 148–155.
4. Burch, J.G., Strater, F.R., and Grudnitski, G. *Information Systems: Theory and Practice*. New York: John Wiley and Sons, Inc., 1983.
5. Canning, R.G. Modular COBOL programming. *EDP Anal. 10*, 7 (July 1972), 1–14.
6. Canning, R.G. That maintenance "iceberg." *EDP Anal. 10*, 10 (Oct. 1972), 1–14.
7. Christensen, K., Fitsos, G.P., and Smith, C.P. A perspective on software science. *IBM Syst. J. 20*, 4 (1981), 372–387.
8. 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 Trans. Softw. Eng.* SE-5, 2 (Mar. 1979), 96–104.
9. Elshoff, J.L. Measuring commercial PL/I programs using Halstead's criteria. *SIGPLAN Not.* (May 1976), 38–46.
10. Endres, A. An analysis of errors and their causes in systems programs. *IEEE Trans. Softw. Eng. SE-1*, 2 (June 1975), 140–149.
11. Fitzsimmons, A. and Love, T. A review and evaluation of software science. *Comput. Surv. 10*, 1 (Mar. 1978), 3–18.
12. Funami, Y., and Halstead, M.H. A software physics analysis of Akiyama's debugging data. *CSD-TR-144*, Purdue University, Lafayette, Ind., May 1975.
13. Gilb, T. *Software Metrics*. Winthrop Publishers, Cambridge, Mass., 1977.
14. Halstead, M.H. *Elements of Software Science*. Elsevier North-Holland, Inc., New York, 1977.
15. Lientz, B.P., and Swanson, E.B. *Software Maintenance Management*. Addison-Wesley Publ. Co., Inc., Reading, Mass., 1980.
16. Lientz, B.P., Swanson, E.B., and Tompkins, G.E. Characteristics of application software maintenance. *Commun. ACM, 21*, 6 (July 1978), 466–471.
17. Littlewood, B. How to measure software reliability and how not to. *Proc. Third International Conf. Softw. Eng.*, Apr. 1978, 37–55.
18. McCabe, T.J. A complexity measure. *IEEE Trans. Soft. Eng. SE-2*, 4 (Dec. 1976), 308–320.
19. Moore, T.E. *The Traveling Man*. Doubleday & Co., Inc., Garden City, N.Y., 1972.
20. Musa, J.D. The use of software reliability measures in project management. *Proceedings: COMPSAC '78*, 493–498.
21. Ottenstein, L.M., Schneider, V.B., and Halstead, M.H. Predicting the number of bugs expected in a program module. *CSD-TR-205*, Purdue University, Lafayette, Ind., Oct. 1976.
22. Sheil, B.A. The psychological study of programming. *ACM Comput. Surv. 13*, 1 (Mar. 1981), 101–120.
23. Stevens, S.S. On the theory of scales of measurement. *Science 103* (1946), 677–680.
24. Swanson, E.B. On the user-requisite variety of computer application software. *IEEE Trans. Reliab. R-28*, 3 (Aug. 1979), 221–226.
25. Swanson, E.B. The dimension of maintenance. *Proc. Second International Con. Softw. Eng.*, Oct. 1976, 492–497.
26. Thayer, T.A., et al., Software reliability study. *RADC-TR-76-2238*, Rome Air Development Center, Grifiss Air Force Base, N.Y., Aug. 1976.
27. Vessey, I., and Weber, R. Some factors affecting program repair maintenance. *Commun. ACM 26*, 2 (Feb. 1983), 128–134.

CR Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distributions and Maintenance—*corrections*; D.2.8 [**Software Engineering**]: Metrics—*software science*; K.6.m [**Management of Computing and Information Sciences**]: Miscellaneous

General Terms: Management

Additional Key Words and Phrases: program maintenance, repair maintenance, program complexity, software science

Received 6/83; revised 12/83; accepted 1/84

Author's Present Address: Lee L. Gremillion, School of Management, 704 Commonwealth Avenue, Boston University, Boston, MA 02215.