Edgar H. Sibley
Panel Editor

An analysis of the distributions and relationships derived from the change data collected during development of a medium-scale software project produces some surprising insights into the factors influencing software development. Among these are the tradeoffs between modifying an existing module as opposed to creating a new one, and the relationship between module size and error proneness.

SOFTWARE ERRORS AND COMPLEXITY: AN EMPIRICAL INVESTIGATION

VICTOR R. BASILI and BARRY T. PERRICONE

1. INTRODUCTION

The identification of the various factors that have an effect on software development is of prime concern to software engineers. The specific focus of this paper is to analyze the relationships between the frequency and distribution of errors during software development, the maintenance of the developed software, and a variety of environmental factors. These factors include the complexity of the software, the developer's experience with the application, and the reuse of existing design and code. Such relationships can provide an insight into the characteristics of computer software and the effects that an environment can have on the software product. Such relationships can also improve the *reliability* and *quality* with respect to computer software. In an effort to acquire knowledge of these basic relationships, change data for a medium-scale software project were analyzed. (Change data include any documentation that reports an alteration made to the software for a particular reason.)

The overall objectives of this paper are threefold: first, to report the results of the analyses; second, to review the results in the context of those reported by other researchers [2, 3, 5, 6]; third, to draw some conclusions based on the first two objectives. The analyses presented in this paper encompass various types of distributions based on the collected change data. The most important are the error distributions observed within the software project.

1.1 Description of the Environment

The software analyzed in this paper is from a large set of projects being studied in the Software Engineering Laboratory (SEL). This particular project is a general-purpose program for satellite planning studies. These planning studies include mission maneuver planning, mission lifetime, mission launch, and mission control. The overall size of the software project was approximately 90,000 lines of code. The majority of the software project was coded in Fortran for execution on an IBM 360.

Although the system outlined here uses many algorithms similar to those of the original SEL projects, it still represents a new application for the development group.

The requirements for the system kept growing and changing, much more so than for the typical ground-support software. Owing to the commonality of algorithms from existing systems, the developers reused the design and code for many algorithms needed in the new system. Hence a large number of reused (modified) modules became part of the new system.

An approximation of the software's life cycle is displayed in Figure 1. This figure only illustrates the ap-

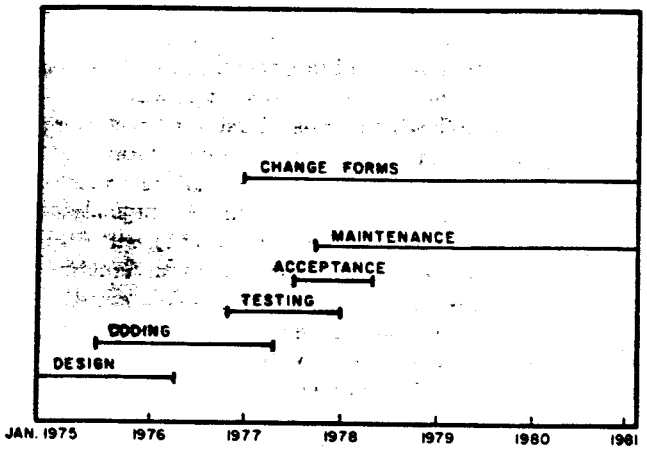

FIGURE 1. Life Cycle of Analyzed Software

proximate duration in time of the various phases of the software's life cycle. The information relating the amount of manpower involved with each of the phases was not specific enough to yield meaningful results, so it was not included.

1.2 Terms

This section defines the terms used in this paper. Please note that many of these terms often denote different concepts in the general literature.

Module: A module is defined as a named subfunction, subroutine, or the main program of the software system. Only segments that contained executable code written in Fortran were used for the analyses. Change data from the segments that constituted the data blocks, assembly segments, common segments, or utility routines were not included. However, a general overview of the data available on these segments is presented in Section 4.

There are two types of modules referred to in this paper. The first type is denoted as *modified*. These are modules that were developed for previous software projects and then modified to meet the requirements of the new project. The second type is referred to as *new*. These are modules that were developed specifically for the software project under analysis.

The entire software project contained 517 code segments, comprised of 36 assembly segments, 370 Fortran segments, and 111 segments that were either common modules, block data, or utility routines. Three hundred seventy out of 517 code segments (72 percent of the total modules) met the adopted module definition and constituted the majority of the software project. Of the modules found to contain errors, 49 percent were categorized as modified and 51 percent as new modules.

Number of Source and Executable Lines: The number of source lines within a module refers to the number of lines of executable code and comment lines contained within it. The number of executable lines within a module refers to the number of executable statements; comment lines are not included.

Some of the relationships presented in this paper are based on a grouping of modules by module size in increments of 50 lines. This means that a module containing 50 lines of code or less was placed in the module size of 50, modules between 51 and 100 lines of code into the module size of 100, and so on. The number of modules contained in each module size category is given in Table I for all modules and for modules that contained errors (i.e., a subset of all modules) with respect to source and executable lines of code.

Error: An error is something detected within the executable code that caused the module in which it occurred to perform incorrectly (i.e., contrary to its expected function).

Errors were quantified from two viewpoints, depending upon the goals of the error analysis. The first quantification was based on a textual rather than a conceptual viewpoint. This type of error quantification is best illustrated by an example. If a "*" is incorrectly used in place of a "+," then all occurrences of the "*" will be considered an error, even if the "*"s appear on the same line of code or within multiple modules. The total number of errors detected in the 370 software modules was 215 contained within a total of 96 modules. This implies that 26 percent of the modules analyzed contained errors.

The second type of quantification measured the effect of an error across modules. Textual errors associated with the same conceptual problem were combined to yield one conceptual error. If a procedure was called with the same incorrect parameter list in multiple modules, this would constitute multiple textual errors but only one conceptual error. This is done only for the errors reported in Table II. There are a total of 155 conceptual errors. All other studies in this paper are based upon the first type of error quantification.

Statistical Terms and Methods: All linear regressions of the data presented in this paper employ the least

TABLE I. Module Size Categories

Number of Lines	All Modules		Modules with Errors	
	Source	Executable	Source	Executable
0–50	53	258	3	49
51–100	107	70	16	25
101–150	80	26	20	13
151–200	56	13	19	7
201–250	34	1	12	1
251–300	14	1	9	0
301–350	7	1	4	1
351–400	9	0	7	0
>400	10	0	6	0
Total	370	370	96	96

squares principle as a criterion of goodness. (That is, "choose as the 'best-fitting' line the one that minimizes the sum of squares of the deviations of the observed values of y from those predicted." [7])

Pearson's product moment coefficient of correlation was used as an index of the strength of the linear relationship, regardless of the respective scales of measurement for y and x. This index is denoted by the symbol r. The measure for the amount of variability in y accounted for by linear regression on x is denoted as r2.

All of the equations and explanations for these statistics can be found in [7]. It should be noted that other types of curve fits were conducted on the data. The results of these fits will be mentioned later in the paper.

2. BASIC DATA

The change data were collected over a period of 33 months (August 1977–May 1980). These dates correspond in time to the software phases of coding, testing, acceptance, and maintenance (Figure 1). The data collected for the analyses are not complete since changes were still being made to the analyzed software. However, enough data were viewed in order to make the conclusions drawn from the data significant.

The change data were entered on detailed report sheets, which were completed by the programmer responsible for implementing the change. A sample of the change report form is given in the Appendix. In general, the form required that several short questions be answered by the programmer implementing the change. These queries documented the cause of a change in addition to other characteristics and effects attributed to the change. The majority of this information was found useful in the analyses. The key information used from the form was:

- The data of the change or error discovery.
- The description of the change or error.
- The number of components changed.
- The type of change or error.
- The effort needed to correct the error.

It should be mentioned that the particular change report form shown in the Appendix is the most current form but was not uniformly used over the entire period of this study. In actuality there were three different versions of the change report form; each form required slightly different information. Therefore, for the data that were not present on one form but that could be inferred, the inferred value was used. An example of such an inference is that of determining the *error type*. Since the error description was given on all of the forms, the error type could be inferred with a reasonable degree of reliability. Data not incorporated into a particular data set used for an analysis were data for which inference was deemed unreliable. Therefore, the reader should be alert to the cardinality of the data set used as a basis for some of the relationships presented in this paper. A total of 231 change report forms were examined for the purpose of this paper.

The quality of the change and error data was checked in the following manner. First, the supervisor of the project looked over the change report forms and verified them (denoted by his or her signature and the date). Second, when the data were reduced for analysis, they were closely examined for contradictions. It should be noted that interviews with the individuals who filled out the change forms were not conducted. This was the major difference between this work and other error studies performed by the SEL, where interviews were held with the programmers to help clarify questionable data. [2]

The review of the change data yielded an interesting result. The errors due to previous correction attempts were shown to be three times as common after the form review process was performed, that is, before the review process they accounted for 2 percent of the errors and after the review process they accounted for 6 percent of the errors. These recording errors are probably attributed to the fact that the corrector of an error did not know the error was due to a previous fix because the fix occurred several months earlier or was made by a different programmer.

3. RELATIONSHIPS DERIVED FROM DATA

This section presents and discusses the relationships derived from the change data.

3.1 Change Distribution by Type

Changes to the software can be categorized as error

TABLE II. Number of Modules Affected by an Error (data set: 211 textual errors; 174 conceptual errors)

Number of Errors	Number of Modules Affected
155 (89%)	1
9	2
3	3
6	4
1	5

TABLE III. Number of Errors per Module (data set: 215 errors)

Number of Modules	New	Modified	Number of Errors per Module
36	17	19	1
26	13	13	2
16	10	6	3
13	7	6	4
4	1**	3*	5
1	1**		7

TABLE IV. Effort to Correct Errors in the Three Most Error-Prone Modified Modules

	Number of Errors (15 total)	Average Effort to Correct (hrs)
Misunderstood or incorrect specifications	8	24.0
Incorrect design or implementation of a module component	5	16.0
Clerical error	2	4.5

TABLE V. Effort to Correct Errors in the Two Most Error-Prone New Modules

	Number of Errors (12 total)	Average Effort to Correct (hrs)
Misunderstood or incorrect requirements	8	32
Incorrect design or implementation of a module component	3	0.5
Clerical error	1	0.5

corrections or modifications (specification changes, planned enhancements, and clarity and optimization improvements). For this project, error corrections accounted for 62 percent of the changes and modifications accounted for 38 percent. In studies of other SEL projects, error corrections accounted for 40–64 percent of the changes.

3.2 Error Distribution by Modules

Table II shows the number of modules that had to be changed because of an error. (Note that these errors are counted as conceptual errors.) It was found that 89 percent of the errors could be corrected by changing only one module. This is a good argument for the modularity of the software. It also shows that there is not a large amount of interdependence among the modules with respect to an error.

Table III shows the number of errors found per module. The type of module is shown in addition to the total number of modules found to contain errors.

The largest number of errors found were 7 (located in a single new module) and 5 (located in 3 different modified modules and 1 new module). The remainder of the errors were distributed almost equally between the two types of modules.

The effort associated with correcting an error is specified on the form as (1) 1 hour or less, (2) 1 hour to 1 day, (3) 1 day to 3 days, or (4) more than 3 days. These categories were chosen because it is too difficult to collect effort data to a finer granularity. To estimate the effort for any particular error correction, an average time was used for each category; that is, assuming an 8-hour day, an error correction in category (1) was assumed to take 0.5 hour, in category (2) 4.5 hours, in category (3) 16 hours, and in category (4) 32 hours.

The types of errors found in the three most error-prone modified modules (* in Table III) and the effort needed to correct them is shown in Table IV. If any type contained error corrections from more than one error correction category, the associated effort for them was averaged. The fact that the majority of the errors detected in a module is between one and three shows that the total number of errors that occurred per module is, on the average, very small.

The twelve errors contained in the two most error-prone new modules (** in Table III) are shown in Table V along with the effort needed to correct them.

3.3 Error Distribution by Type

Figure 2 shows the distribution of errors by type. It can be seen that 48 percent of the errors was attributed to incorrect or misinterpreted functional specifications or requirements.

The error classification used throughout the Software Engineering Laboratory is given below. The person identifying the error indicates the class for each error.

A: Requirements incorrect or misinterpreted.
B: Functional specification incorrect or misinterpreted.
C: Design error involving several components.
1. Mistaken assumption about value or structure of data.
2. Mistake in control logic or computation of an expression.

D: Error in design or implementation of single component.
1. Mistaken assumption about value or structure of data.
2. Mistake in control logic or computation of an expression.

E: Misunderstanding of external environment.
F: Error in the use of programming language/compiler.
G: Clerical error.
H: Error due to previous miscorrection of an error.

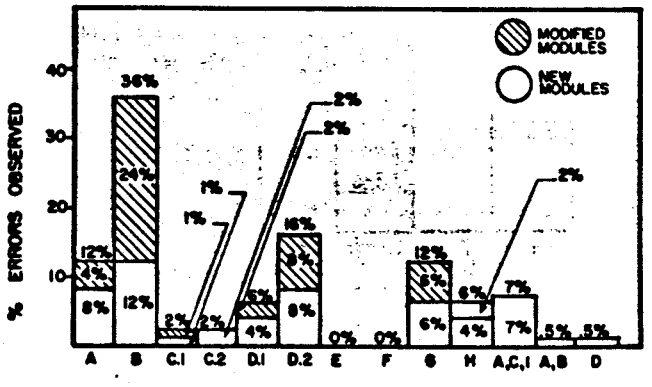

FIGURE 2. Sources of Errors

The distribution of these errors by source is plotted in Figure 2 with the appropriate subdistribution of new and modified errors displayed. This distribution shows that the majority of errors were the result of functional specification (incorrect or misinterpreted). Within this category, the majority of the errors (24 percent) involved modified modules. This is most likely due to the fact that the reused modules were taken from another system with a different application. Thus, even though the basic algorithms were the same, the specification was not well-enough defined or appropriately defined for the modules to be used under slightly different circumstances.

The distribution in Figure 2 should be compared to the distribution of another system developed by the same organization, shown in Figure 3(a) [3]. For a basis of comparison, the categories in Figure 2 are mapped into a classification scheme [Figure 3(b)] equivalent to

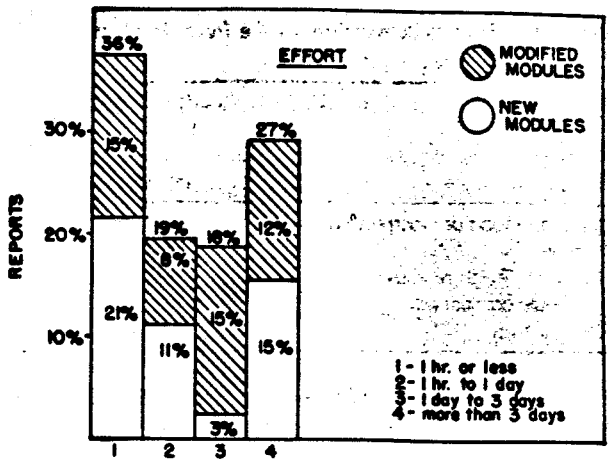

FIGURE 4. Effort Graph

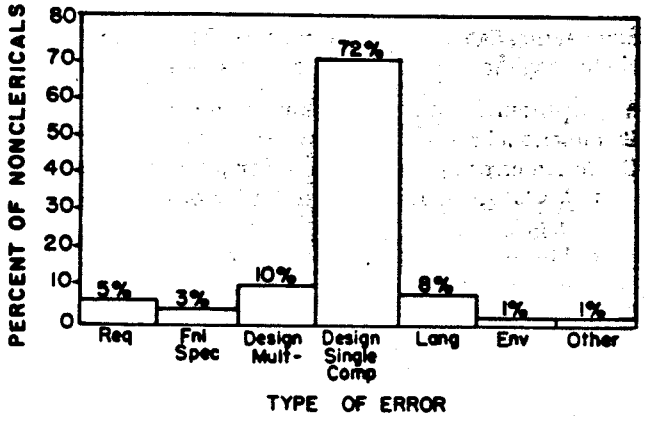

FIGURE 3(a). Sources of Errors on Other Nonclerical SEL Projects.

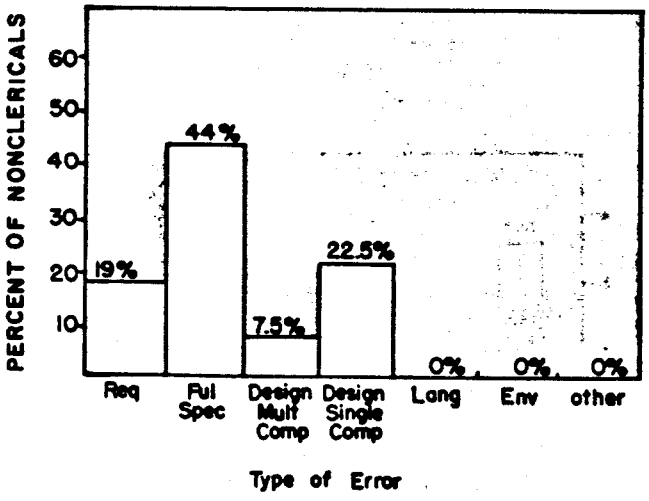

FIGURE 3(b). Sources of Nonclerical Errors on this Project

those for Figure 3(a) (eliminating the categories of G and H within Figure 2). Figure 3 represents a typical ground-support software system and was rather typical of the error distributions for these systems. It is different from the distribution for the system we are discussing in that the majority of the errors were involved in the design of a single component. The reason for the difference is that in ground-support systems, the design is well understood and the developers have had a reasonable amount of experience with the application. Any reused design or code comes from a similar system and the requirements tend to be more stable. An analysis of the two distributions makes the differences in the development environments clear in a quantitative way.

The percent of requirements and specification errors is consistent with Endres' work [7]. Endres found that 46 percent of the errors he viewed involved the misunderstanding of the functional specifications of a module. Our results are similar even though Endres' analysis was based on data derived from a different software project and programming environment. The software project used in Endres' analysis contained considerably more lines of code per module, was written in assembly code, and was within the problem area of operating systems. However, both of the software systems Endres analyzed did contain new and modified modules. In this study, of the errors due to the misunderstanding of a module's specifications or requirements (48 percent), 20 percent involved new modules while 28 percent involved modified modules.

Although the existence of modified modules can shrink the cost of coding, the amount of effort needed to correct errors in modified modules might outweigh the savings. The effort graph (Figure 4) supports this view: 50 percent of the total effort required for error correction occurred in modified modules; errors requiring one day to more than three days to correct accounted for 45 percent of the total effort with 27 per-

cent of this effort attributable to modified modules within these greater effort classes. Thus, errors occurring in new modules required less effort to correct than those in modified modules.

The similarity between Endres' results and those reported here tend to support the statement that, independent of the environment and possibly the module size, the majority of errors detected within software are due to an inadequate form or misinterpretation of the specifications. This seems especially true when the software contains modified modules.

3.4 Overall Number of Errors Observed

Figure 5 displays the number of errors observed in both new and modified modules. It can be seen that errors occurring in modified modules are detected earlier and at a slightly higher rate than those in new modules. One hypothesis for this is that the majority of the errors observed in modified modules are due to the misinterpretation of the functional specifications. Errors of this type would certainly be more obvious since they are more blatant than those of other types and, therefore, would be detected both earlier and more readily. (See next section.)

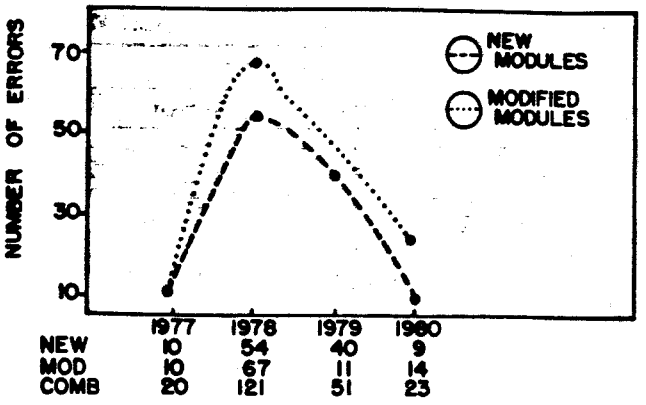

FIGURE 5. Number of Errors Occurring in Modules

3.5 Abstract Error Types

The authors adopted an abstract classification of errors that classified errors into one of five categories with respect to a module: (1) initialization, (2) control structure, (3) interface, (4) data, and (5) computation. This was done in order to see if there existed recurring classes of errors in all modules, independent of size. These error classes are only roughly defined. It should be noted that even though the authors were consistent with the categorization for this project, another error analyst may have interpreted the categories differently.

Failure to initialize or reinitialize a data structure properly upon a module's entry/exit is considered an *initialization error.* Errors that cause an "incorrect path" in a module to be taken are considered *control errors.* Such a control error might be a conditional statement causing control to be passed to an incorrect path. *Interface errors* are those that were associated with structures existing outside the module's local environment but which the module used. For example, the incorrect declaration of a COMMON segment or an incorrect subroutine call is an interface error. An error in the declaration of the COMMON segment is considered an interface error and not an initialization error since the COMMON segment has been used by the module but is not part of its local environment. *Data errors* are those errors that are a result of the incorrect use of a data structure. Examples of data errors are the use of incorrect subscripts for an array, the use of the wrong variable in an equation, or the inclusion of an incorrect declaration of a variable local to the module. *Computation errors* are those that cause a computation to erroneously evaluate a variable's value. These errors could be equations that are incorrect not by virtue of the incorrect use of a data structure within the statement but by miscalculations. An example of this error might be the statement $A = B + 1$ when the statement really needed was $A = B/C + 1$.

These five abstract categories basically represent all activities present in any module. The five categories are further partitioned into errors of commission and omission. Errors of *commission* are those errors present as a result of an incorrect executable statement. For example, a commissioned computational error would be $A = B * C$ where the '*' should have been '+'. In other words, the operator was present but was incorrect. Errors of *omission* are those errors that are a result of forgetting to include some entity within a module. For example, a computational omission error might be $A = B$ when the statement should have read $A = B + C$. A parameter required for a subroutine call but not included in the actual call is an example of an interface omission error. In both of the above examples some aspect needed for the correct execution of a module has been forgotten.

The results of this abstract classification scheme are given in Table VI. Since there were approximately an equal amount of new (49) and modified (47) modules viewed in the analysis, the results do not need to be normalized. Some errors and thereby modules were counted more than once, since it was not possible to associate some errors with a single abstract error type based on the error description given on the change report form.

According to Table VI, interfaces appear to be the major problem, regardless of the module type. Control is more of a problem in new modules than in modified modules. This is probably because the algorithms in the old modules had more test and debug time. On the other hand, initialization and data are more of a problem in modified modules. These facts, coupled with the small number of errors of omission in the modified

TABLE VI. Abstract Classification of Errors

	Commission		Omission		Total	
	New	Modified	New	Modified	New	Modified
Initialization	2	9	5	9	7	18 – 25 (11%)
Control	12	2	16	6	28	8 – 36 (16%)
Interface	23	31	27	6	50	37 – 87 (39%)
Data	10	17	1	3	11	20 – 31 (14%)
Computation	16	21	3	3	19	24 – 43 (19%)
	28%	36%	23%	12%	115	107
	64%		35%			

modules, might imply that the basic algorithms for the modified modules were correct but needed some adjustment with respect to data values and initialization for the application of that algorithm to the new environment.

3.6 Module Size and Error Occurrence

Scatter plots for executable lines per module versus the number of errors found in the module were graphed. It was difficult to see any trend within these plots, so the number of errors/1000 executable lines within a module size was calculated (Table VII). The number of errors was normalized over 1000 executable lines of code in order to determine if the number of detected errors within a module was dependent on module size. All modules within the software were included, even those with no detected errors. If the number of errors/1000 executable lines was found to be constant over module size, this would show independence. An unexpected trend was observed: Table VII implies that there is a higher error rate in smaller sized modules. Since only the executable lines of code were considered, the larger modules were not COMMON data files. Also the larger modules will be shown to be more complex than smaller modules in the next section. Then how could this type of result occur?

The most plausible explanation seems to be that the large number of interface errors spread equally across all modules is causing a larger number of errors per 1000 executable statements for smaller modules. Some tentative explanations for this behavior are that: the majority of the modules examined were small (Table I), causing a biased result; larger modules were coded with more care than smaller modules because of their size; and errors in smaller modules were more apparent. There may still be numerous undetected errors present within the larger modules since all the "paths" within the larger modules may not have been fully exercised.

TABLE VII. Errors/1000 Executable Lines (Includes all modules)

Module Size	Errors/1000 Lines
50	16.0
100	12.6
150	12.4
200	7.6
>200	6.4

TABLE VIII. Average Cyclomatic Complexity for all Modules

Module Size	Average Cyclomatic Complexity
50	6.0
100	17.9
150	28.1
200	52.7
>200	60.0

3.7 Module Complexity

Cyclomatic complexity [8] (number of decisions + 1) was correlated with module size. This was done in order to determine whether or not larger modules were less dense or complex than smaller modules containing errors. Scatter plots for executable statements per module versus the cyclomatic complexity were graphed. Since it was difficult to see any trend in the plots, modules were grouped according to size. The complexity points were obtained by calculating an average complexity measure for each module size class. For example, all the modules that had 50 executable lines of code or less had an average complexity of 6.0. Table VIII gives the average cyclomatic complexity for all modules in each of the size categories. The complexity relationships for executable lines of code in a module are shown in Figure 6. As can be seen from Table VIII, the larger modules were more complex than smaller modules.

Table IX gives the number of errors/1000 executable statements and the average cyclomatic complexity only for those modules containing errors. When these data are compared with Table VIII, one can see that the

average complexity of the error-prone modules was no greater than the average complexity of the full set of modules.

4. DATA NOT EXPLICITLY INCLUDED IN ANALYSES

The 147 modules not included in this study (i.e., assembly segments, common segments, utility routines) contained six errors. These six errors were detected within three different segments. One error occurred in a modified assembly module because of a misunderstanding or incorrect statement of the functional specifications for the module. The effort needed to correct this error was minimal (1 hour or less).

The other five errors occurred in two separate new data segments with the major cause of the errors also being related to their specifications. The effort needed to correct these errors was on the average from 1 hour to 1 day (1 day representing 8 hours).

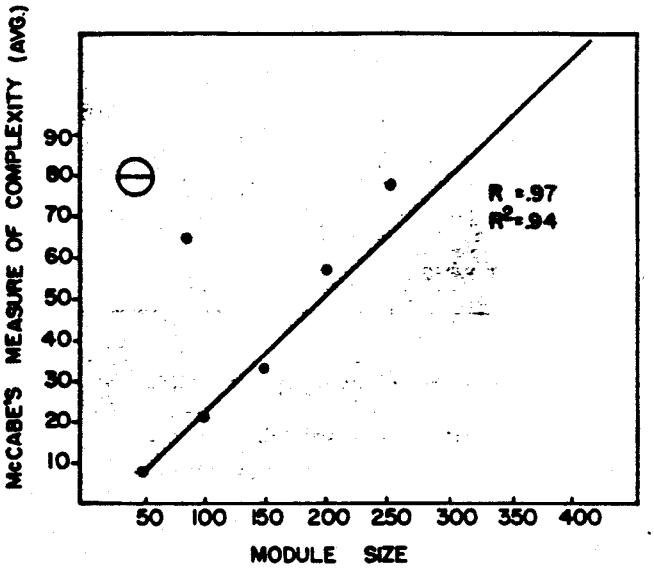

FIGURE 6. Complexity versus Module Size

5. CONCLUSIONS

The data contained in this paper help explain and characterize the software developed. It is clear from the data that this was a new application for the developers, with changing requirements.

Modified and new modules were shown to behave similarly except for the types of errors prevalent in each and the amount of effort required to correct an error. Both had a high percentage of interface errors. However, new modules had an equal number of errors of omission and commission and a higher percentage of control errors. Modified modules had a high percentage of errors of commission and a small percentage of errors of omission with a higher percentage of data and initialization errors. Another difference was that modified modules appeared to be more susceptible to errors due to the misunderstanding of the specifications. Misunderstanding of a module's specifications or requirements constituted the majority of detected errors. This duplicates Endres' earlier result, which implies that more work needs to be done on the form and content of the specifications and requirements in order for them to be used more effectively across applications.

There are some disadvantages to modifying an existing module for use instead of creating a new module. Modifying an existing module to meet a similar but different set of specifications reduces the development costs of that module. However, the disadvantage is that there are hidden costs. Errors contained in modified modules were found to require more effort to correct than those in new modules, although the two classes contained approximately the same number of errors. The majority of these errors were because of incorrect or misinterpreted specifications for a module. Therefore, there is a trade-off between minimizing development time and time spent to align a module to new specifications. However, if better specifications could be developed, it might reduce the more expensive errors contained within modified modules. In this case, the use of "old" modules could be more beneficial in terms of cost and effort since the hidden costs would have been reduced.

One surprising result was that module size did not account for error proneness. In fact, it was quite the contrary—the larger the module, the less error prone it was. This was true even though the larger modules were more complex. Additionally, the error-prone modules were no more complex across size grouping than the error-free modules. This result implies we are not yet ready to put artificial limits on module size and complexity.

In general, error analysis provides useful information. For this project, it shows that the developers were involved in a new application with changing requirements. It provides insight into the different ways of handling new and modified modules. It shows areas of potential problems with a new application. It ultimately allows us to identify the various factors that influence software development.

TABLE IX. Complexity and Error Rate for Errored Modules

Module Size	Average Cyclomatic Complexity	Errors/1000 Executable Lines
50	6.2	65.0
100	19.6	33.3
150	27.5	24.6
200	56.7	13.4
>200	77.5	9.7

APPENDIX—Change Report Form

PROJECT NAME _____ CURRENT DATE _____

SECTION A - IDENTIFICATION

REASON: Why was the change made? _____

DESCRIPTION: What change was made? _____

EFFECT: What components (or documents) are changed? (Include version) _____

EFFORT: What additional components (or documents) were examined in determining what change was needed? _____

	(Month	Day	Year)
Need for change determined on			
Change started on			

What was the effort in person time required to understand and implement the change?

_____1 hour or less, _____1 hour to 1 day, _____1 day to 3 days, _____more than 3 days

SECTION B - TYPE OF CHANGE (How is this change best characterized?)

- ☐ Error correction
- ☐ Planned enhancement
- ☐ Implementation of requirements change
- ☐ Improvement of clarity, maintainability, or documentation
- ☐ Improvement of user services
- ☐ Insertion/deletion of debug code
- ☐ Optimization of time/space/accuracy
- ☐ Adaptation to environment change
- ☐ Other (Explain in E)

Was more than one component affected by the change? Yes _____ No _____

FOR ERROR CORRECTIONS ONLY

SECTION C - TYPE OF ERROR (How is this error best characterized?)

- ☐ Requirements incorrect or misinterpreted
- ☐ Functional specifications incorrect or misinterpreted
- ☐ Design error, involving several components
- ☐ Error in the design or implementation of a single component
- ☐ Misunderstanding of external environment, except language
- ☐ Error in use of programming language/compiler
- ☐ Clerical error
- ☐ Other (Explain in E)

FOR DESIGN OR IMPLEMENTATION ERRORS ONLY

If the error was in design or implementation:

The error was a mistaken assumption about the value or structure of data _____

The error was a mistake in control logic or computation of an expression _____

580-2 (6/78)

APPENDIX—Change Report Form

FOR ERROR CORRECTIONS ONLY

SECTION D - VALIDATION AND REPAIR

What activities were used to validate the program, detect the error, and find its cause?

	Activities Used for Program Validation	Activities Successful in Detecting Error Symptoms	Activities Tried to Find Cause	Activities Successful in Finding Cause
Pre-acceptance test runs				
Acceptance testing				
Post-acceptance use				
Inspection of output				
Code reading by programmer				
Code reading by other person				
Talks with other programmers				
Special debug code				
System error messages				
Project specific error messages				
Reading documentation				
Trace				
Dump				
Cross-reference/attribute list				
Proof technique				
Other (Explain in E)				

What was the time used to isolate the cause?

____one hour or less, ____one hour to one day, ____more than one day, ____never found

If never found, was a workaround used?____Yes____No (Explain in E)

Was this error related to a previous change?

____Yes (Change Report #/Date____) ____No ____Can't tell

When did the error enter the system?

____requirements ____functional specs ____design ____coding and test ____other ____can't tell

SECTION E - ADDITIONAL INFORMATION

Please give any information that may be helpful in categorizing the error or change, and understanding its cause and its ramifications.

Name:____ Authorized:____ Date:____

580-2 (6/78)

The results of this study are by no means conclusive. They pose more questions than they answer; they suggest that software development must be better understood. More data must be collected on different projects.

Acknowledgments. The authors would like to thank F. McGarry, NASA Goddard Space Flight Center, for his cooperation in supplying the information needed for this study and his helpful suggestions on earlier drafts of this paper.

REFERENCES

1. Basili, V., and Freburger, K. Programming measurement and estimation in the Software Engineering Laboratory. *The Journal of Systems and Software 2*, 1 (Mar. 1981), 47–57.
2. Basili, V., and Weiss, D. A methodology for collecting valid software engineering data. University of Maryland Tech. Rep. TR-1235, Dec. 1982.
3. Basili, V., and Weiss, D. Evaluating software development by analysis of changes: The data from the Software Engineering Laboratory. University of Maryland Tech. Rep. TR-1236, Dec. 1982.
4. Belady, L. A., and Lehman, M. M. A model of large program development. *IBM Systems Journal 15*, 3 (1976), 225–251.
5. Endres, A. An analysis of errors and their causes in system programs. In *Proceedings of the International Conference on Software Engineering*. (April 1975), pp. 327–336.
6. McCabe, T. J. A complexity measure. *IEEE Transactions on Software Engineering SE-2*, 4 (Dec. 1976), 308–320.
7. Mendenhall, W., and Ramey, M. *Statistics for Psychology*. Duxbury Press, North Scituate, Mass., 1973, pp. 280–315.
8. Schneidewind, N. F. An experiment in software error data collection and analysis. *IEEE Transactions on Software Engineering SE-5*, 3 (May 1979), 276–286.
9. Weiss, D. M. Evaluating software development by error analysis: The data from the architecture research facility. *The Journal of Systems and Software 1*, 1 (Mar. 1979), 57–70.

CR Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics
General Terms: Experimentation, Measurement, Reliability
Additional Key Words and Phrases: error analysis, complexity metrics

Received 10/82; revised 6/83; accepted 7/83

The research for this study was supported in part by the National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland.

Authors' Present Address:
V. R. Basili and B. T. Perricone, Dept. of Computer Science, University of Maryland, College Park, Maryland 20742