D7-61

N87-24902

Technical Report TR-1195 August 1982
NSG-5123

# SOFTWARE ERRORS AND COMPLEXITY: AN EMPIRICAL INVESTIGATION*

Victor R. Basili and Barry T. Perricone

A1 91796

*Research supported in part by National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland.. Computer time supported in part through the facilities of the Computer Science Center of the University of Maryland.

# SOFTWARE ERRORS AND COMPLEXITY:

# AN EMPIRICAL INVESTIGATION

Victor R. Basili and Barry T. Perricone

Department of Computer Science

University of Maryland

College Park, Md.

1982

ABSTRACT

The distributions and relationships derived from the change data collected during the development of a medium scale satellite software project shows that meaningful results can be obtained which allow an insight into software traits and the environment in which it is developed. Modified and new modules were shown to behave similarly. An abstract classification scheme for errors which allows a better understanding of the overall traits of a software project is also shown. Finally, various size and complexity metrics are examined with respect to errors detected within the software yielding some interesting results.

## 1.0 INTRODUCTION

The discovery and validation of fundamental relationships between the development of computer software, the environment in which the software is developed, and the frequency and distribution of errors associated with the software are topics of primary concern to investigators in the field of software engineering. Knowledge of such relationships can be used to provide an insight into the characteristics of computer software and the effects that a programming environment can have on the software product. In addition, it can provide a means to improve the understanding of the terms reliability and quality with respect to computer software. In an effort to acquire a knowledge of these basic relationships, change data for a medium scale software project was analyzed (e.g., change data is any documentation which reports an alteration made to the software for a particular reason).

In general, 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; and third, to draw some conclusions based on the aforementioned. The analyses presented in this paper encompass various types of distributions based on the collected change data. The most important of which are the error distributions observed within the software project.

In order for the reader to view the results reported in this paper properly, it is important that the terms used throughout this paper and the environment in which the data was collected are clearly defined. This is pertinent since many of the terms used within this paper have appeared in the general literature often to denote different concepts. Understanding the environment will allow the partitioning of the results into two classes: those which are dependent on and those which are independent of a particular programming environment.

## 1.1 DESCRIPTION OF THE ENVIRONMENT

The software analyzed within this paper is one of a large set of projects being analyzed in the Software Engineering Laboratory (SEL). The particular project analyzed in this paper is a general purpose program for satellite planning studies. These studies include among others: mission maneuver planning; mission lifetime; mission launch; and mission control. The overall size of the software project was approximately 90,000 source lines of code. The majority of the software project was coded in FORTRAN. The system was developed and executes on an IBM 360.

The developers of the analyzed software had extensive experience with ground support software for satellites. The analyzed system represents a new application for the development group, although it shares many similar algorithms with the system studied here.

It is also true that the requirements for the system analyzed kept growing and changing, much more so than for the typical ground support software normally built. Due to the commonality of algorithms from existing systems, the developers re-used the design and code for many algorithms needed in the new system. Hence a large number of re-used (modified)
modules became part of the new system analyzed here.

An approximation of the analyzed software's life cycle is displayed in Figure 1 . This figure only illustrates the approximate 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 shown was not specific enough to yield meaningful results, so it was not included.

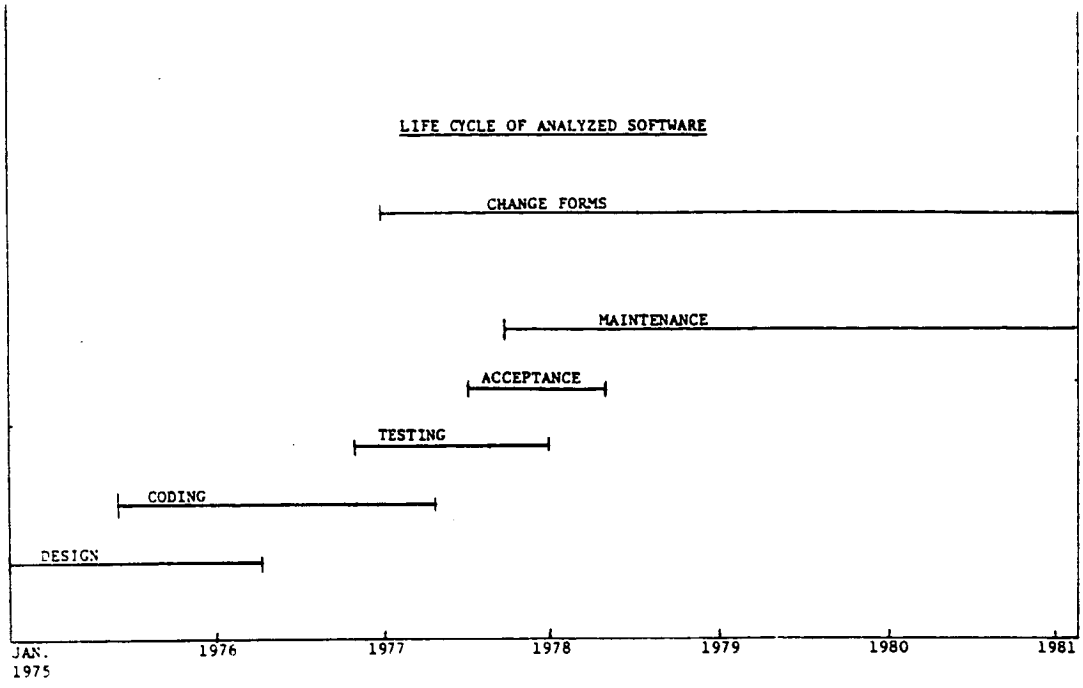



Figure 1

## 1.2 TERMS

This section presents the definitions and associated contexts for the terms used within this paper. A discussion of the concepts involved with these terms is also given when appropriate.

Module: A module is defined as a named subfunction, subroutine, or the main program of the software system. This definition is used since only segments written in FORTRAN which contained executable code were used for the analyses. Change data from the segments which constituted the data blocks, assembly segments, common segments, or utility routines were not included. However, a general overview of the data available on these types of segments is presented in Section 4.0 for completeness.

There are two types of modules referred to within this paper. The first type is denoted as modified. These are

modules which 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 which were developed specifically for the software project under analyses.

The entire software project contained a total of 517 code segments. This quantity is comprised of 36 assembly segments, 370 FORTRAN segments, and 111 segments that were either common modules, block data, or utility routines. The number of code segments which met the adopted module definition was 370 out of 517 which is 72% of the total modules and constitutes the majority of the software project. Of the modules found to contain errors 49% were categorized as modified and 51% 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, etc. The number of modules which were contained in each module size is given in Table 1 for all modules and for modules which contained errors (i.e., a subset of all modules) with respect to source and executable lines of code.

| Number of Lines | Number modules: All Modules Source | All Modules Exececutable | Modules with Errors Source | Modules with Errors 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 |

Table 1

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

Errors were quantified from two view points in this paper, depending upon the goals of the analysis of the error data. 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 "*" was incorrectly used in place of a "+" then all occurrences of the "*" will be considered an error. This is the situation 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 analyzed was 215 contained within a total of 96 modules, implying 26% of the modules analyzed contained errors.

The second type of quantification was used to measure the effect of an error across modules, textual errors associated with the same conceptual problem were combined to yield one conceptual error. Thus in the example above, all incorrectly used *'s replaced by +'s in the same formula were combined and the total number of modules effected by that error are listed. This is done only for the errors reported in Figure 2. There are a total of 155 conceptual errors. All other studies in this paper are based upoon the

first type of quantification described.

Statistical Terms and Methods: All linear regressions of the data presented within this paper employed as a criterion of goodness the least squares principle (i.e., "choose as the 'best fitting' line that one which minimizes the sum of squares of the deviations of the observed values of y from those predicted" [1]).

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

All of the equations and explanations for these statistics can be found in [1]. 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.

Now that the software's environment and the key terms used within the paper have been defined and outlined, a discussion of the basic quantification of the data collected, the relationships and distributions derived from this quantification, and the resulting conclusions are presented.

## 2.0 BASIC DATA

The change data analyzed was collected over a period of 33 months, August 1977 through 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 is not complete since changes are still being made to the software analyzed. However, it is felt that enough data was viewed in order to make the conclusions drawn from the data significant.

The change data was 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 allowed a means to document 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 in the study 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, and the effort needed to correct the error.

It should be mentioned that the particular change report form shown in the Appendix was 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, not all of which required the same set of questions to be answered. Therefore , for the data that was not present on one type of form but could be inferred, the inferred value was used. An example of such an inference would be 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 was that data for which this 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. There was a total of 231 change report forms examined for the purpose of this paper.

The consistency and partial validity of the forms 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 was being reduced for analysis it was 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 Software Engineering Laboratory, where interviews were held with the programmers to help clarify questionable data (8).

The review of the change data as described above yielded an interesting result. The errors due to previous miscorrections showed to be three times as common after the form review process was performed, i.e. before the review process they accounted for 2% of the errors and after the review process they accounted for 6% of the errors. These recording errors are probably attributable to the fact that the corrector of an error did not know the cause was due to a previous fix because the fix occurred several months earlier or was made by a different programmer, etc.

## 3.0 RELATIONSHIPS DERIVED FROM DATA

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

## 3.1 CHANGE DISTRIBUTION BY TYPE

Types of changes to the software can be categorized as error corrections or modifications (specification changes, planned enhancements, clarity and optimization improvements). For this project, error corrections accounted for 62% of the changes and modifications 38%. In studies of other SEL projects, errors corrections ranged from 40% to 64% of the changes.

## 3.2 ERROR DISTRIBUTION BY MODULES

Figure 2 shows the effects of an error in terms of the number of modules that had to be changed. (Note that these errors here are counted as conceptual errors.) It was found that 89% 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.

---

NUMBER OF MODULES AFFECTED BY AN ERROR (data set: 211 textual errors)
174 conceptual errrors)

| #ERRORS | #MODULES AFFECTED |
|---|---|
| 155 (89%) | 1 |
| 9 | 2 |
| 3 | 3 |
| 6 | 4 |
| 1 | 5 |

---

Figure 2

---

Figure 3 shows the number of errors found per module. The type of module is shown in addition to the overall total number of modules found to contain errors.

NUMBER OF ERRORS PER MODULE (data set: 215 errors)

| #MODULES | NEW | MODIFIED | #ERRORS/MODULE |
|---|---|---|---|
| 36 | 17 | 19 | 1 |
| 26 | 13 | 13 | 2 |
| 16 | 10 | 6 | 3 |
| 13 | 7 | 6 | 4 |
| 4 | 1** | 3* | 5 |
| 1 | 1** | | 7 |

Figure 3

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 among the two types of modules.

The effort associated with correcting an error is specified on the form as being (1) 1 hour or less, (2) 1 hour to 1 day, (3) 1 day to 3 days, (4) more than 3 days. These categories were chosen because it was 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, i.e. assuming an 8 hour day, an error correction in category (1) was assumed to take .5 hours, an error correction in category (2) was assumed to take 4.5 hours, category (3) 16 hours, and category (4) 32 hours.

The types of errors found in the three most error prone modified modules (* in Figure 3) and the effort needed to correct them is shown in Table 2. 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 was between one and three shows that the total number of errors that occurred per module was on the average very small.

The twelve errors contained in the two most error prone new modules (** in Figure 3) are shown in Table 3 along with the effort needed to correct them.

| | NUMBER OF ERRORS (15 total) | AVERAGE EFFORT[ TO CORRECT |
|---|---|---|
| misunderstood or incorrect specifications | 8 | 24 hours |
| incorrect design or implementation of a module component | 5 | 16 hours |
| clerical error | 2 | 4.5 hours |

EFFORT TO CORRECT ERRORS IN THREE MOST ERROR PRONE MODIFIED MODULES

Table 2

| | NUMBER OF ERRORS (12 total) | AVERAGE EFFORT TO CORRECT |
|---|---|---|
| misunderstood or incorrect requirements | 8 | 32 hours |
| incorrect design or implementation of a module | 3 | 0.5 hours |
| clerical error | 1 | 0.5 hours |

EFFORT TO CORRECT ERRORS IN THE TWO MOST ERROR PRONE NEW MODULES

Table 3

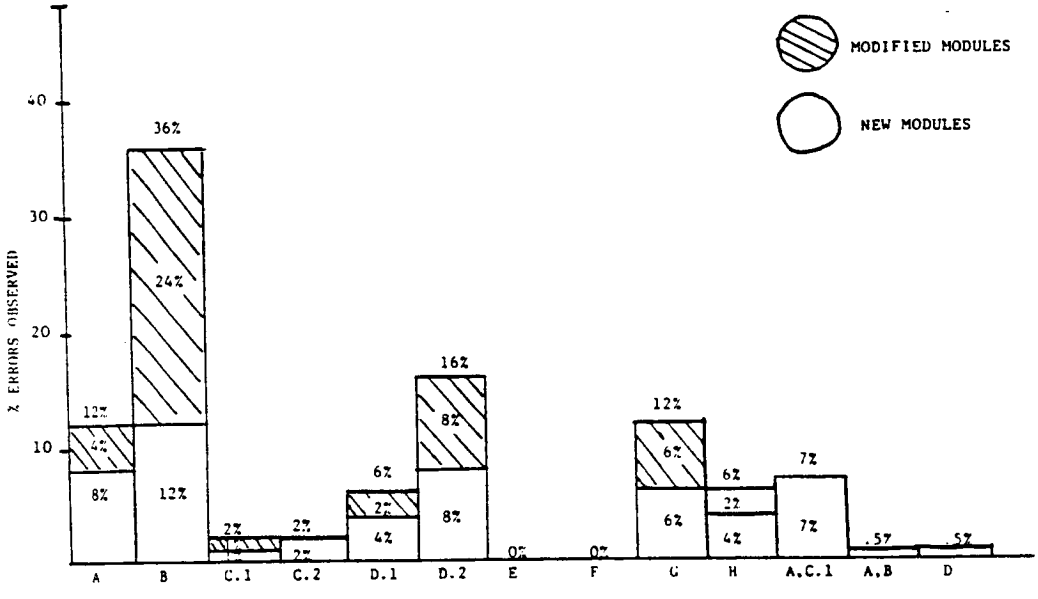
## 3.3 ERROR DISTRIBUTION BY TYPE

In Figure 4 the distribution of errors are shown by type. It can be seen that 48% of the errors were attributed to incorrect or misinterpreted functional specifications or requirements.

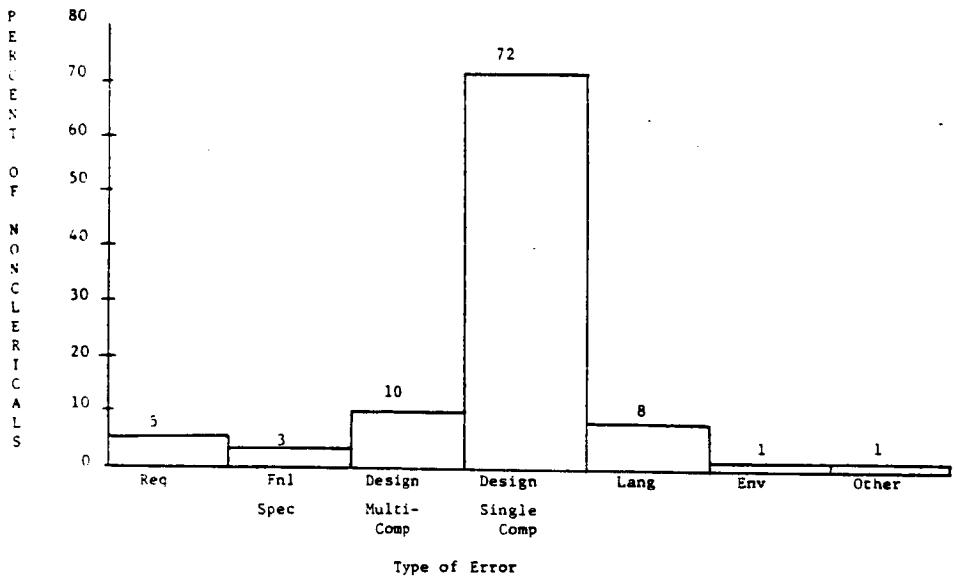
The classification for error used throughout the Software Engineering Laboratory is given below. The person identifying the error indicates the class for each error.

A: Requirements incorrect or misinterpreted
B: Functional specification incorrect or misinterpreted
C: Design error invloving 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

The distribution of these errors by source is plotted in Figure 4 with the appropriate subdistribution of new and modified errors displayed. This distribution shows the majority of errors were the result of the functional specification being incorrect or misinterpreted . Within this category, the majority of the errors (24%) involved modified modules This is most likely due to the fact that the modules reused 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.




SOURCES OF ERRORS
Figure 4



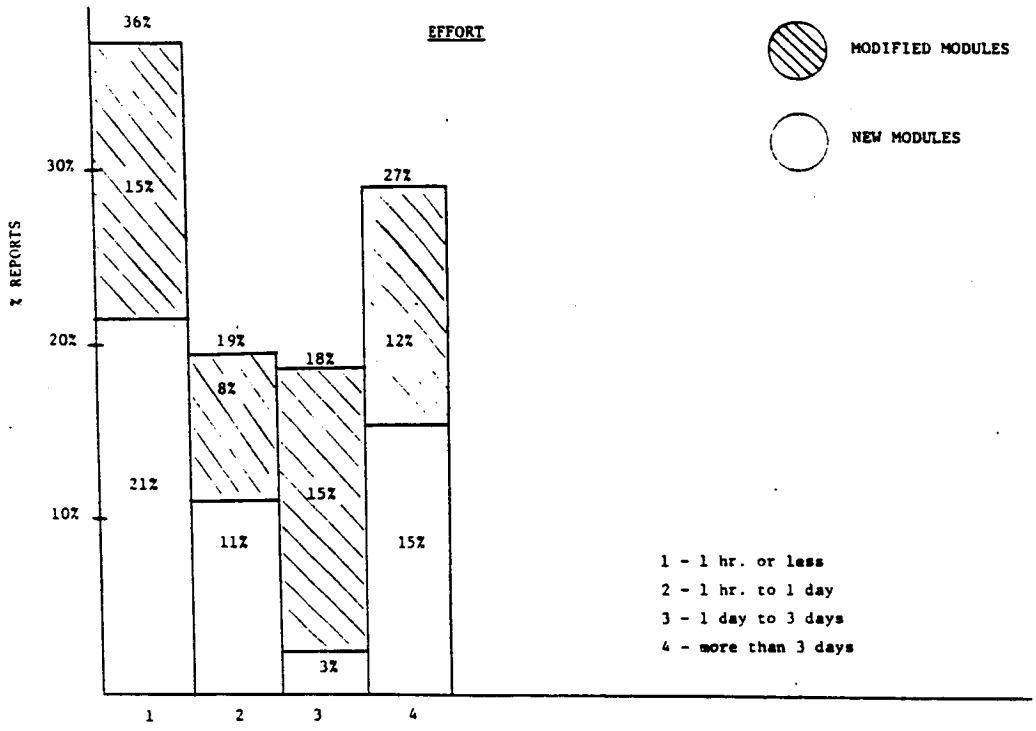

SOURCES OF ERROR ON OTHER PROJECTS
Figure 5

The distribution in Figure 4 should be compared with the distribution of another system developed by the same organization shown in Figure 5. Figure 5 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 this paper however, 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, the developers have had a reasonable amount of experience with the application. Any re-used design or code comes from similar systems, 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 the work of Endres'[1]. Endres found that 46% 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.

Of the errors due to the misunderstanding of a module's specifications or requirements (48%), 20% involved new modules while 28% 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 6) supports this viewpoint: 50% 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% of the total effort with 27% 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 occurring in modified modules.




EFFORT GRAPH
Figure 6

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 is due to an inadequate form or interpretation of the specifications. This seems especially true when the software contains modified modules.

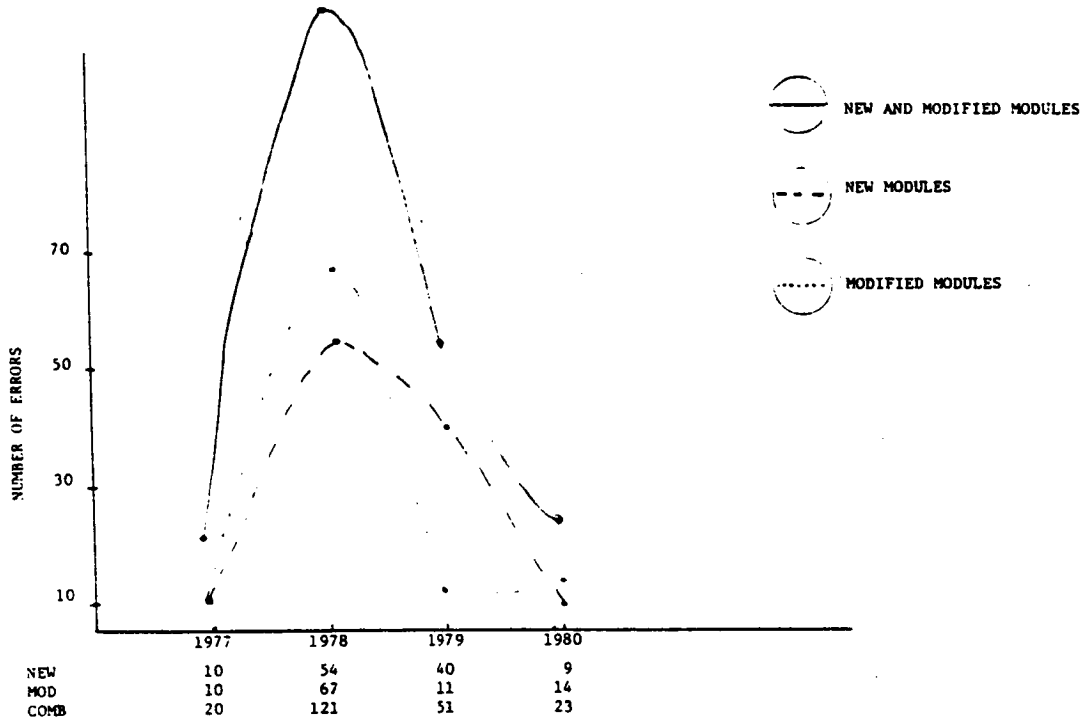
In general, these observations tend to indicate that there are disadvantages in modifying a large number of already existing modules to meet new specifications. The alternative of developing a new module might be better in some cases if there does not exist good specifications for the existing modules.

## 3.4 OVERALL NUMBER OF ERRORS OBSERVED

Figure 7 displays the number of errors observed in both new and modified modules. The curve representing total

modules (new and modified) is basically bell-shaped. One interpretation is that up to some point errors are detected at a relatively steady rate. At this point at least half of the total "detected-undetected" errors have been observed and the rate of discovery thereafter decreases. It may also imply the maintainers are not adding too many new errors as the system evolves.

It can be seen, however, that errors occurring in modified modules are detected earlier and at a slightly higher rate than those of 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 as was mentioned earlier in the paper. 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.)




NUMBER OF ERRORS OCCURRING IN MODULES
Figure 7

## 3.5 ABSTRACT ERROR TYPES

An abstract classification of errors was adopted by the authors which 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 present in all modules independent of size. These error classes are only roughly defined so examples of these abstract error types are presented below. It should be noted that even though the authors were consistant with the categorization for this project, another error

analyst may have interpreted the categories differently.

Failure to initialize or re-initialize a data structure properly upon a module's entry/exit would be considered an initialization error. Errors which caused an "incorrect-path" in a module to be taken were considered control errors. Such a control error might be a conditional statement causing control to be passed to an incorrect path. Interface errors were those which 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 would be an interface error. An error in the declaration of the COMMON segment was considered an interface error and not an initialization error since the COMMON segment was used by the module but was not part of its' local environment. Data error would be those errors which are a result of the incorrect use of a data structure. Examples of data errors would be 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 were those which caused a computation to erroneously evaluate a variable's value. These errors could be equations which were incorrect not by virtue of the incorrect use of a data structure within the statement but rather 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 were further partitioned into errors of commission and omission. Errors of commission were 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 were those errors which were 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 would be an example of an interface omission error. In both of the above examples some aspect needed for the correct execution of a module was forgotten.

The results of this abstract classification scheme as discussed above is given in Figure 8. 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.

| | commission | | omission | |
|---|---|---|---|---|
| | new | modified | new | modified |
| initialization | 2 | 9 | 5 | 9 |
| control | 12 | 2 | 16 | 6 |
| interface | 23 | 31 | 27 | 6 |
| data | 10 | 17 | 1 | 3 |
| computation | 16 | 21 | 3 | 3 |
| | 28% | 36% | 23% | 12% |
| | 64% | | 35% | |

| | total | | |
|---|---|---|---|
| | new | modified | |
| initialization | 7 | 18 | 25 (11%) |
| control | 28 | 8 | 36 (16%) |
| interface | 50 | 37 | 87 (39%) |
| data | 11 | 20 | 31 (14%) |
| computation | 19 | 24 | 43 (19%) |
| | 115 | 107 | |

ABSTRACT CLASSIFICATION OF ERRORS
Figure 8

According to Figure 8, 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 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 plotted. 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 4).

| Module Size | Errors/1000 lines |
|---|---|
| 50 | 16.0 |
| 100 | 12.6 |
| 150 | 12.4 |
| 200 | 7.6 |
| >200 | 6.4 |

ERRORS/1000 EXECUTABLE LINES (INCLUDES ALL MODULES)
Table 4

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 errors detected. If the number of errors/1000 exececutable lines was found to be constant over module size this would show independence. An unexpected trend was observed: Table 4 implies that there is a higher error rate within 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 plausable explanation seems to be that since there are a large number of interface errors, these are spread equally across all modules and so there are a larger number of errors/1000 executable statements for smaller modules. Some tentative explanations for this behavior are: the majority of the modules examined were small (Table 1) causing a biased result; larger modules were coded with more care than smaller modules because of their size; errors in smaller modules are more apparent and there may indeed still be numerous undetected errors present within the larger modules since all the "paths" within the larger modules may not yet have been fully exercised.

## 3.7 MODULE COMPLEXITY

Cyclomatic complexity [5] (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 statments per module versus the cyclomatic complexity were plotted and again, 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 which had 50 executable lines of code or less had an average complexity of 6.0. Table 5 gives the average cyclomatic complexity for all modules within each of the size categories. The complexity relationships for executable lines of code within a module is shown in Figure 9. As can be seen from the table the larger modules were more complex than smaller modules.

| Module size | Average Cyclomatic Complexity |
|---|---|
| 50 | 6.0 |
| 100 | 17.9 |
| 150 | 28.1 |
| 200 | 52.7 |
| >200 | 60.0 |

AVERAGE CYCLOMATIC COMPLEXITY FOR ALL MODULES
Table 5

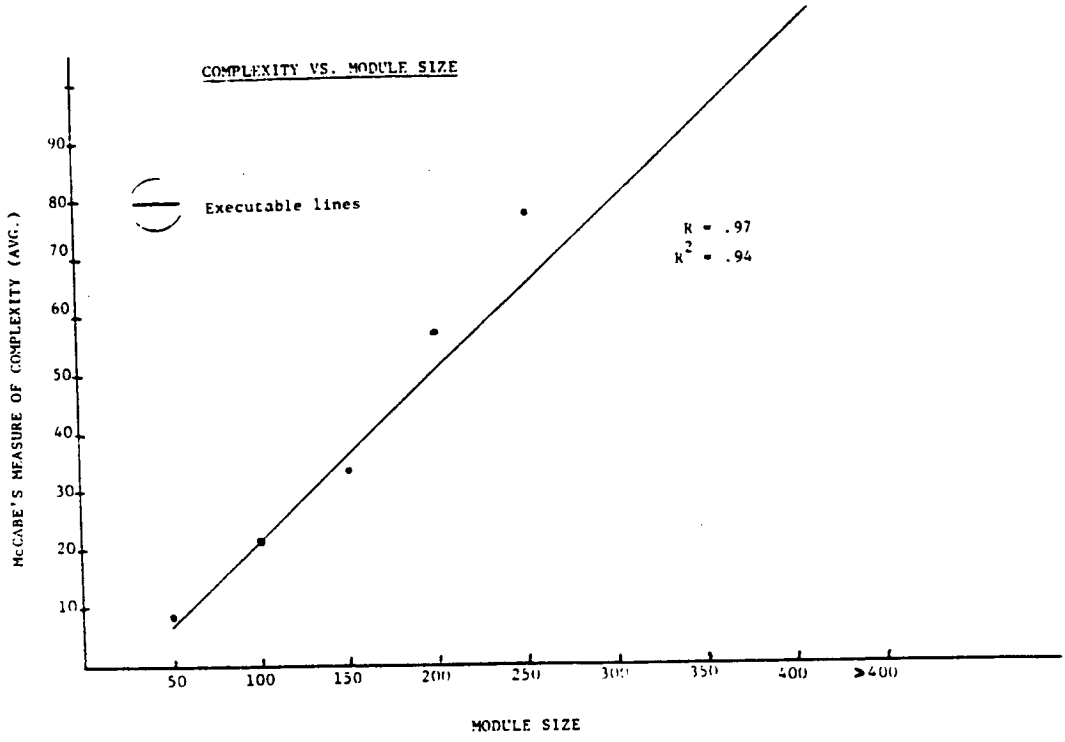



Figure 9

For only those modules containing errors, Table 6 gives the number of errors/1000 executable statements and the average cyclomatic complexity. When this data is compared with Table 5 , 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.

| 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 |

COMPLEXITY AND ERROR RATE FOR ERRORED MODULES
Table 6

## 4.0 DATA NOT EXPLICITLY INCLUDED IN ANALYSES

The 147 modules not included in this study (i.e., assembly segments, common segments, utility routines) contained a total of six errors. These six errors were detected within three different segments. One error occurred in a modified assembly module and was due to the 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).

## 5.0 CONCLUSIONS

The data contained in this paper helps explain and characterize the environment in which the software was developed. It is clear from the data that this was a new application domain in an application with changing requirements.

Modified and new modules were shown to behave similarly except in 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 errors detected. This duplicates an earlier result of Endres which implies that more work needs to be done on the form and content of the specifications and requirements in order to enable them to be used across applications more effectively.

There were shown to be 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 developmental costs of that module. However, the disadvantage to this is that there exists 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 was due to incorrect or misinterpreted specifications for a module. Therefore, there is a tradeoff 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 reuse 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.

In general, investigations of the type presented in this paper relating error and other change data to the software in which they have occurred is important and relevant. It is the only method by which our knowledge of these types of relationships will ever increase and evolve.

Acknowledgments

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

References

(1) Mendenhall,W. and Ramey,M., Statistics for Psychology, Duxbury Press, North Scituate, Mass., 1973, pp. 280-315.

(2) Endres,A.,"An Analysis of Errors and their Causes in System Programs", Proceedings of the International Conference on Software Engineering, April, 1975, pp. 327-336.

(3) Belady,L.A. and Lehman,M.M., "A Model of Large Program Development", IBM Systems Journal, Vol.15, 1976, pp.225-251.

(4) Weiss,D.M., "Evaluating Software Development by Error Analysis : The Data from the Architecture Research Facility", The Journal of Systems and Software, Vol.1, 1979, pp. 57-70.

(5) Schneidewind,N.F., "An Experiment in Software Error Data Collection and Analysis", IEEE Transactions on Software Engineering , Vol. SE-5, No.3, May 1979, pp.276-286.

(6) McCabe, T.J., "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, Dec. 1976, pp.308-320.

(7) Basili,V. and Freburger,K., "Programming Measurement and Estimation in the Software Engineering Laboratory", The Journal of Systems and Software, Vol.2, 1981, pp.47-57.

(8) Weiss, D.M.," Evaluating Software Development by Analysis of Change Data", University of Maryland Technical Report TR-1120, November 1981.

APPENDIX

APPENDIX
CHANGE REPORT FORM

NUMBER _____

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 _____

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:_____

Change Report Form