COLLECTING AND CATEGORIZING SOFTWARE ERROR DATA IN AN INDUSTRIAL ENVIRONMENT

BY

THOMAS J. OSTRAND*

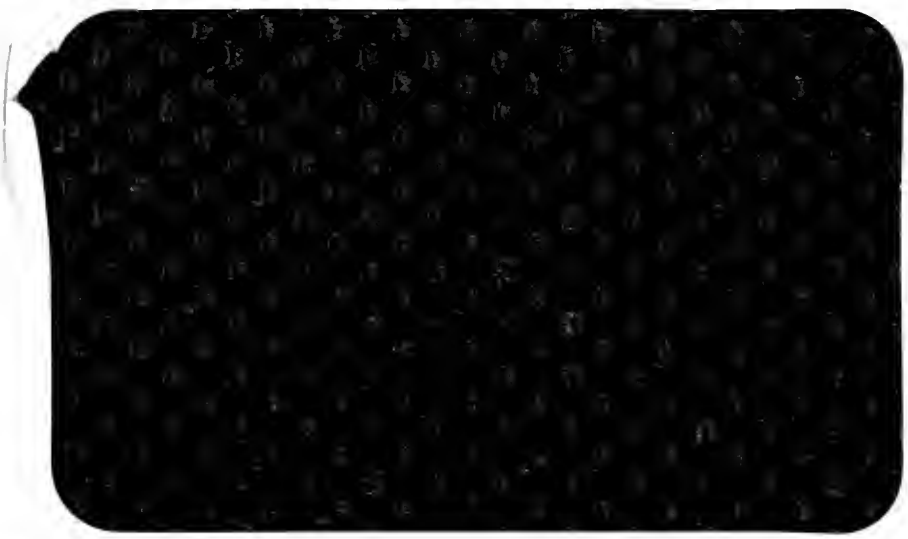
AND

ELAINE J. WEYUKER**

TECHNICAL REPORT #47

AUGUST 1982

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012



COLLECTING AND CATEGORIZING SOFTWARE ERROR DATA IN AN INDUSTRIAL ENVIRONMENT

BY

THOMAS J. OSTRAND*

AND

ELAINE J. WEYUKER**

TECHNICAL REPORT #47

AUGUST 1982

* Systems and Software Research
Sperry Univac
Blue Bell, PA 19424

** Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, New York 10012

This research was supported in part by the National Science Foundation under Grant MCS-82-001167.

ABSTRACT

A study has been made of the software errors committed during development of an interactive special-purpose editor system. This product has been followed during nine months of coding, unit testing, function testing, and system testing. Detected errors and their fixes have been described by testers and debuggers. To help analyze the relationship of error characteristics to the various aspects of the software development process, a new error categorization scheme has been developed. Within this scheme, 174 errors were classified. For each error, we asked the programmers to select the most likely cause of the error, report the stage of the software development cycle in which the error was created and first noticed, and the circumstances of its detection and isolation, including time required, techniques tried, and successful techniques.

The results collected in this study are compared to results from earlier studies, and similarities and differences are noted.

1. Introduction

The assessment of many software development and validation methods suffers from a lack of factual and statistical information characterizing their effectiveness, particularly in a real production environment.

For the past two years, we have been involved in the design and execution of a study of software errors at Sperry Univac. We have gathered information describing types of errors, the methods used to detect their presence and isolate problems, and the difficulty of detection, isolation, and correction. We also attempted to determine when an error enters the product being developed, how much time elapses until it is detected, and perhaps most difficult, to determine the real underlying cause of the error's existence.

The study's specific goals include finding answers to the following questions:

-- how do the types, frequencies, and difficulty of detection and isolation of errors vary over the steps of development and use of software?

-- can the trends in software error statistics reported in earlier studies be verified?

-- can a usable and practical method of error categorization be developed?

In addition, error information collected now will later be used as baseline data for evaluating the effect on the number and types of errors of different software development methods and tools.

In this paper we report the results obtained from collecting error data for about nine months. One medium sized project has been followed from approximately midway through coding to the completion of system testing. The product consists of about 10000 lines of high level source code and 70000 bytes of object code. Program design and coding for the project were done by three programmers over ten months, after the initial specification had been completed. The implementation represents approximately 18 person-months of effort. An additional five months were used for function and system testing by independent test groups, and for the consequent changes to the product. The product has now passed both function and system testing and has just been released to customers.

2. The Collection of Data

Previous error studies have given us valuable insights into the way to organize data collection, and the type of information to collect [3, 5, 8, 10, 13, 14, 16, 17, 18, 20, 21]. We have drawn especially from Basili [4], Basili et al [6], Thayer et al [18], and Weiss [20, 21] in developing our collection goals and method. Our two-page Change Report form, shown in Figure 1, is modelled after those of Weiss [21]. The form is used during the entire development cycle to describe changes made to the product. During and after function testing, error symptoms are not entered on the Change Report, since a standard Univac form, the Software User Report (SUR), is used for this purpose. In those cases, all remaining questions are answered on the Change Report form and a copy of the SUR is attached to the form.

C H A N G E R E P O R T

Project: _____ Date problem found or SUR date, if applicable: _____

Unit(s) requiring change: _____ | _____ | _____ | _____

Programming language: _____ | _____ | _____ | _____

	PROBLEM DETECTION Methods used	PROBLEM DETECTION Problem noticed by	PROBLEM ISOLATION (debugging) Methods tried	PROBLEM ISOLATION (debugging) Successful method
Requirements Review	[ ]	[ ]	[ ]	[ ]
Design Review	[ ]	[ ]	[ ]	[ ]
Desk Checking by Programmer	[ ]	[ ]	[ ]	[ ]
Desk Checking by Other Person	[ ]	[ ]	[ ]	[ ]
Code Review	[ ]	[ ]	[ ]	[ ]
Test Runs by Programmer	[ ]	[ ]	[ ]	[ ]
Test Runs by Other Person	[ ]	[ ]	[ ]	[ ]
Execution under Trace Control	[ ]	[ ]	[ ]	[ ]
Examining Dump	[ ]	[ ]	[ ]	[ ]
Special Debugging Code	[ ]	[ ]	[ ]	[ ]
Internal SUR (# _____)	[ ]	[ ]	[ ]	[ ]
Customer SUR (# _____)	[ ]	[ ]	[ ]	[ ]
Other ( )	[ ]	[ ]	[ ]	[ ]
Other ( )	[ ]	[ ]	[ ]	[ ]
Other ( )	[ ]	[ ]	[ ]	[ ]

Description of problem symptoms (not needed if this problem is reported on a SUR):

Description of actual problem:

Date _____ and description of fix (Give as much detail as possible):

Was the original code or documentation that you just changed
- [ ] written entirely by you
- [ ] written partly by you
- [ ] not written by you at all

RETEST (for SUR writer only) Date:_____
- [ ] Passed
- [ ] Failed
  - [ ] with same symptoms
  - [ ] with new symptoms give new SUR # _____

Figure 1. Change Report Form

TIME FOR DEBUGGING AND FIXING

	under 1 hour	1 hour to 4 hours	4 hours to 1 day	over 1 day
Time Used to Isolate Problem	[ ]	[ ]	[ ]	[ ]
Time Used to Make Correction	[ ]	[ ]	[ ]	[ ]

CHANGES WERE MADE TO

Code

[] 5 or fewer instructions in 1 procedure [] more than 5 instructions in 1 proc

[] 2 or more procs in 1 module [] procs in more than 1 module

Documentation

[] Documentation in code [] High-level specification (in CPSD [] or PD [])

[] Program specification (in SDS) [] Interface specification (in SDS)

[] User documentation

STEPS OF SW DEVELOPMENT	WHEN WAS PROBLEM FIRST NOTICED	DURING WHICH STEP WAS PROBLEM CREATED
Requirements (SOR)	[ ]	[ ]
Feasibility	[ ]	[ ]
Definition (PD or CPSD)	[ ]	[ ]
Program Design (SDS)	[ ]	[ ]
Coding	[ ]	[ ]
Unit Testing	[ ]	[ ]
Integration	[ ]	[ ]
Function Testing	[ ]	[ ]
System Test & Qual	[ ]	[ ]
First Release	[ ]	[ ]
Unknown	[ ]	[ ]

WHY DO YOU THINK THE PROBLEM OCCURRED?

[] Requirements or [] specification changed for [] this unit or [] other unit.
[] Requirements were unclear, incomplete, ambiguous, etc.
[] Specification was unclear, incomplete, ambiguous, etc.
[] Programming language doesn't do what manual claims it does.
[] Programmer misunderstood programming language.
[] Programmer misunderstood development system.
[] Clerical error (Terminal, keypunching, unclear handwriting, etc.)
[] Communication failure between people (e.g., misunderstanding of subroutine interface).
[] Programmer error
[] A previous fix led to this problem
[] Other (specify):

Please give any other information that may help to describe what the error is, how it was found, or explain its cause.

Figure 1. Change Report Form

The form does not ask the programmer to categorize program errors. Rather, in its only part which requires substantial writing, the programmer describes the error symptoms, the actual problem discovered, and the correction made. From these descriptions we have developed a new error classification scheme. This attribute categorization scheme, described in the next section, is more accurate in defining errors, more flexible, and easier to use than previous classifications. The written error descriptions are also valuable for data validation interviews with the programmers, since they enable accurate recall of an error's circumstances. The validation interviews are used to confirm that programmers are interpreting the check-off categories on the form consistently, and to eliminate ambiguous or unclear descriptions of errors.

The Change Report form includes several questions which try to pinpoint the reason that the error entered the software. Although the ultimate objective of software error analysis is to understand the underlying causes of errors, of the studies referenced above, only Weiss [20] has asked similar questions of his study participants. Other researchers have attempted to deduce error causes from knowledge of the programming environment and types of errors detected. For example, although the goals of his study are to enumerate error causes and find ways to prevent their occurrence, Endres [8] did not collect any direct information about specific causes. He defines six broad categories of error cause, and then presents some reasonable guesses as to causes of various error types, possible preventive measures, and detection methods.

The categorization scheme and data on frequency of various error types and their underlying causes can be used to develop and evaluate new strategies for program testing. Goodenough [11] has pointed out the advantages of developing tests with the purpose of detecting specific errors. Several recently proposed testing techniques are based on anticipation of frequently occurring software errors. These techniques include error-based testing [15], mutation testing [1,7], error-sensitive test cases [9], and functional testing [12]. Since these methods are intended to be applied during both unit and function testing, we have begun collection of error data as early as possible in the software development process. Approximately one-third of the errors reported for the project were discovered during unit testing. This contrasts with most previous software error studies, which have begun collecting data after the software has passed unit testing and been integrated into a system complete enough for functional testing to begin.

Before starting error data reporting, the programmers working on the project completed a form describing their general experience, their experience with the project's implementation language and system, and their knowledge of various software development techniques and languages. The project manager completed a form recording the project's time schedule, the resources available, methods to be used during the various development steps, and tools available to the project personnel. He was asked to update this project profile whenever any significant changes, such as a revised schedule, took place. This background information will be used as a basis for comparing

error statistics of this project with results collected from other projects.

3. Error Categorization

A major goal of this research is to devise a usable and practical scheme for categorizing software errors. Such a scheme is essential for understanding and controlling the factors which affect software quality. Although several categorization schemes have been used in previous studies [8,17,18], there are serious problems with all of them. These problems are discussed in [19]; they include ambiguous, overlapping, and incomplete categories, too many categories, and confusion of error symptoms, error causes, and actual errors.

The categorization scheme we have developed attempts to make assigning a category as simple and mechanical as possible, thereby reducing the possibility of misclassification. The usual approach to classification is to place a given error at an appropriate node in a tree of categories. The TRW system [18] is an example of such a scheme. The primary characteristic of a tree scheme is the requirement to place each error in a unique category that simultaneously represents all its features. The many characteristics of a software error and the large number of categories in tree schemes make them difficult to use and subject to the problems mentioned above.

A slightly different approach to classification is the five-dimensional scheme proposed by Amory and Clapp [2]. Errors are classified in the dimensions WHERE, WHAT, HOW, WHEN, and WHY.

Within each of these dimensions is a hierarchy or tree of characteristics. Classification of a specific error consists of assigning one or more nodes from each tree to the error, and then attaching more detailed information describing the error to each assigned node. The scheme's five dimensions provide a broad description of the error; at the same time the tree structure allows very detailed information to be recorded.

Amory and Clapp intended "to define a framework which was open-ended and extendible and could be adapted to the particular needs and orientations of users wishing to classify errors." To our knowledge, no one has reported using this classification scheme in a study.

Our error categorization scheme describes only the programming aspects of errors. The scheme records information roughly equivalent to the dimensions HOW and WHAT of Amory and Clapp's method. (However, our Change Report form collects sufficient information to identify other aspects of an error, including the three dimensions WHERE, WHEN, and WHY.)

Since even the programming aspects of an error are multi-dimensional, we do not assign an error to a single category, as in a tree scheme. Instead, we attempt to identify the error's characteristics in several distinct areas, including major category, type, presence, and use of data. There are only a few possible values for each area, making it easy to choose the correct one. Taken together, all the area values for a given error should provide even more detailed information about the error than a unique node in a category tree.

We call this approach an attribute categorization scheme; the various areas whose values describe an error are similar to the attributes of a relation in a relational database. A significant advantage to this method is the relative ease of recording additional information about each error by adding new attributes to the scheme. The characteristics already recorded for an error do not become invalid when the new information is added.

The attributes described below were defined in order to classify the errors collected from the Univac project. They obviously do not include all potential error characteristics; as we monitor additional projects, and collect error reports from the operational use and maintenance phases, the scheme will be extended to include other characteristics.

Major Category

An error's major category identifies what type of code was changed to fix the error, and corresponds roughly to the first level branching of an error category tree. Errors from the Univac project have been placed in the following major categories.

Data Definition - Code which defines constants, storage areas, control codes, etc.

Data Handling - Code which modifies or initializes the values of variables.

Test - Code which evaluates a condition and branches according to the result.

Test plus Process - Code which evaluates a condition and performs a

specific computation if the test is satisfied.

Documentation - Written description of the product.

System - This category describes any error in the program's environment, including operating system, compiler, hardware, etc.

Not an Error - This category is provided for problem reports which are resolved without changing any part of the system or product.

Type

The type of an error modifies the major category. For a major category involving data, the type can be one of the following:

Address - Information which locates values in memory. Typical examples are an array index, list pointer, offset into a defined storage area, and the name of a table of continuation codes.

Control - Information to determine flow of control, such as entries in a branch table or codes identifying different output display formats.

Data - Primary information which is processed, read, or written.

For a major category involving a test, the types are:

Loop - If the test controls a loop.

Branch - If the test controls a multiple-way branch such as a simple IF or a CASE statement.

Presence

The possible values are

Omitted - Something was left out.

Superfluous - Something was present that should not have been.

Incorrect - Something present had to be corrected.

Use

This attribute has been used only rarely in the errors reported on to date. It describes the operation being performed in a data handling error. The possible values are

Initialize - Variable x is initialized with value k if k is independent of the present value of x and subsequent values of x will depend on k.

Set - Variable x is set to value k if k is independent of the present value of x and subsequent values of x will be independent of k.

Update - Variable x is updated with value k if k depends on the present value of x.

To illustrate the categorization scheme's use, we give two (slightly edited) examples of problem descriptions and show how the errors were classified. Problem 1 was classified as a single data handling error. Problem 2 was classified as two errors, an omitted test plus processing and an omitted data definition.

Problem 1:

Description of error symptoms:

Lines which needed to be created at the end of Section A were

not created.

Description of actual problem:

A counter indicating the number of lines to be created was cleared at the wrong point in the module.

Description of fix:

Counter should have been initialized only at first time Section A was entered. Initialization instruction was moved to an existing processing block that was invoked only first time Section A was entered.

Problem 2:

Description of error symptoms:

When output screen C was redisplayed to add more names to the destination-table, source lines were created with the "indexed-by" phrase overlaying the "destination-table" phrase.

Description of actual problem:

Code to eliminate the reserved word "DESTINATION-TABLE" on second use was omitted. A partial line was created but not written; then next line was created without clearing the partial line. Result was jumbled output.

Description of fix:

New CASE statement added to handle situation. New constants added for screen C to allow the new CASE processing.

Figure 2 shows the three error categorizations that resulted from these two problems.

	Major Category	Type	Presence	Use
Problem 1	Data Handling	Control	Incorrect	Initialization
Problem 2	Test plus Processing	Branch	Omit	
Problem 2	Data Definition	Control	Omit	

Figure 2. Error Categorization

4. Data Collection Results

The Univac product which was followed for this error study had a two-year history from its initial statement of requirements to its certification by the system testing organization. The product is a special-purpose editor used for interactive writing and maintenance of high-level source programs. The initial specification (also the first version of the user manual) was completed by month 5. The system's overall structure was determined and a program design in pseudo-code completed by month 10. Coding began in month 10 and by month 16 enough had been completed for an initial version of the system to run.

Function testing by an independent quality assurance group began in month 17, and system testing began in month 22. The editor was certified to have met system testing requirements in the middle of month 23.

We began monitoring the project for changes at the beginning of month 15, shortly after the programmers had started unit testing their modules. A small amount of program design was still going on at this time, accounting for two errors discovered during the program design phase. Every Change Report for the editor was completed by the original programming team. During unit testing, the programmers detected errors and filled out the

entire form themselves. During function and system testing, errors were detected by independent testers who entered a date and identification number on the Change Report, and described the error symptoms on a Software User Report. The two Reports were then sent to the programmers who analyzed the problem, made necessary changes, and completed the Change Report.

A total of 174 Change Report forms were completed; 160 of these represented errors. The others included 7 compiler or system bugs, 3 cases in which an independent tester misunderstood the specification, 3 changes in the specification, and 1 case in which the programmer deliberately did not fulfill the specification since he felt it was too difficult to do. This case ultimately led to a change in the specification.

In 15 cases, a given response was categorized as containing two distinct types of error -- frequently a data definition error and a test. In one case a response led to three categories: two different types of omitted data definition information as well as omitted test plus processing code. For questions involving error categorizations, therefore, the database contains 177 entries. For most other queries, such as those involving when the error was made and why the error was made, there are 160 entries, since that is the number of distinct errors made.

Table 1 gives a capsule definition of the development steps monitored, and shows the total number of errors detected during each phase.

Step	Description	Performed by	Number of Errors	Percent
Program Design	Creation of system's modular structure. High-level description (pseudo-code) of system's algorithms.	Developers	2	1%
Coding	Translation of program design from pseudo-code to implementation language.	Developers	8	5%
Unit testing	Execution of separate modules. Test cases designed by developers attempt to perform module's functions.	Developers test their own and each other's code.	46	29%
Function testing	Execution of entire product from user's point of view. Attempts to execute all possible functions of product.	Separate testing group, using a predefined test plan developed from user manual.	101	63%
System testing	Execution of product in environment of the operating system. Tests behavior of product-product interfaces.	Separate system test group.	3	2%

TABLE 1. Software Development Steps and Errors Found

Table 2 shows the distribution of the 162 non-clerical errors by major category:

Major category	Total number	Percent
data definition	54	33
data handling	38	23
test plus processing	32	20
test (alone)	31	19
systems	5	3
documentation	1	1
unknown	1	1

TABLE 2. Major Categories of Errors

For data handling and data definition errors, the type attribute describes the data that was involved in the error. The three classes, address, control, and data, are defined in Section 3. Table 3 shows the results of this finer categorization.

	DATA HANDLING		DATA DEFINITION	
Type of data error	Total number	Percent	Total number	Percent
address	14	37	5	9
control	10	26	24	44
data	14	37	25	46

TABLE 3. Types of Data Errors

For data handling errors, we also attempted to categorize the error by use. In each case we tried to determine whether the error involved an initialization, a setting, or an updating of a variable. Our use of these terms is also defined in Section 3. Of the 38 data handling errors, we were able to make a use determination in 32 cases. The results are shown in Table 4.

Use of variable	Total number	Percent
initialize	8	25
set	12	38
update	12	38

TABLE 4. Uses in Data Handling Errors

It is interesting to compare the distribution of types of non-clerical errors detected during unit testing and function testing. The results are shown in Table 5.

Major category	UNIT TESTING Total number	Percent	FUNCTION TESTING Total number	Percent
data definition	7	16	47	44
data handling	22	51	11	10
test plus proc	1	2	25	24
test (alone)	10	23	20	19
systems	3	7	2	2
unknown	0	0	1	1

TABLE 5. Categories of Errors Found during Unit and Function Testing

Thus the original programmers were quite successful in detecting data handling errors, while most data definition and test problems were not detected until function testing.

In an attempt to understand this distribution, we examined programmer's explanations for why errors were made within each major category. Unclear, incomplete or ambiguous specifications were cited as the cause of 41% of test plus processing errors, 19% of the errors involving test statements alone, and 19% of data definition errors. In contrast, less than 3% of all data handling errors were attributed to a problem with the specifications. It is not surprising that errors caused by a misunderstanding of the specifications by the programmer (for whatever reason) are less likely to be detected by the person who originally wrote the code than by someone else. With this perspective, the difference in pattern of detection time by major category makes sense.

Of the 162 responses for which it was meaningful, and for which we were able to make the determination, we found that 55% involved omitted code, 44% involved incorrect but not omitted code, and 1% involved superfluous code. A total of 39% of all

responses which were nonclerical errors were categorized as involving a test (either a test statement alone (19%) or a test statement plus processing (20%)). Of these, 81% involved omitted code and the remaining 19% incorrect code. In no instance did a programmer describe an error involving a superfluous test. Of the errors involving a test statement alone, 65% involved omitted code and 35% incorrect code. For the test plus processing category, the figures were 97% omitted versus 3% incorrect.

Of the non-clerical data definition errors, 31% were omissions while 69% involved incorrect data definitions. Notice that while the majority of all errors involving tests were instances of omitted code, less than one-third of the data definition errors were omissions. For data handling errors, 45% involved omitted code, 50% incorrect code, and 5% superfluous code.

The large number of errors of omission, especially those involving tests, is an important signal about program testing techniques. Many popular methods, including branch testing, rely solely on analysis of the existing program code to generate tests. The above figures show the need for developing program tests which are based on the problem's requirements and specifications as well.

Tables 6 and 7 report on the effort required to isolate errors (i.e., to determine the problem), and the effort to fix errors (i.e., to actually write and install the correction.)

Time to isolate error	Total number	Percent
less than 1 hour	127	79
1 to 4 hours	14	9
4 hours to 1 day	4	3
more than 1 day	12	8
no response	3	2

TABLE 6. Error Isolation Time

Time to fix error	Total number	Percent
less than 1 hour	114	71
1 to 4 hours	30	19
4 hours to 1 day	9	6
more than 1 day	3	2
no response	4	3

TABLE 7. Error Fixing Time

Thus most errors were isolated and corrected in less than an hour. Do isolation and correction times vary with the stage of the software development cycle at which the error was first noted? We were surprised to find that during unit testing 61% of all detected errors were isolated in less than 1 hour, while during function testing 85% of all detected errors were isolated in less than 1 hour. A possible explanation for this difference is that the error information supplied to the programmer by the function tester who detects the error provides a fresh or different perspective about the problem. Although the formal error reporting mechanism (the SUR) contains only a brief description of the error's symptoms, in practice the programmer usually receives additional written and oral diagnostic information from the tester.

Time to isolate error	UNIT TESTING Total number	Percent	FUNCTION TESTING Total number	Percent
less than 1 hour	28	61	86	85
1 to 4 hours	10	22	4	4
4 hours to 1 day	3	7	1	1
more than 1 day	4	9	8	8
no response	1	2	2	2

TABLE 8. Error Isolation Time during Unit and Function Testing

Error fixing, however, does not become easier during function testing. Table 9 shows that practically all errors detected during unit testing were fixed in under one hour, while more than one-third of those detected during function testing required over one hour.

Time to fix error	UNIT TESTING Total number	Percent	FUNCTION TESTING Total number	Percent
less than 1 hour	42	91	65	64
1 to 4 hours	3	7	24	24
4 hours to 1 day	0	0	7	7
more than 1 day	0	0	3	3
no response	1	2	2	2

TABLE 9. Error Fixing Time during Unit and Function Testing

Why	Total number	Percent
programmer error	108	68
language misunderstood	6	4
previous fix	3	2
communication failure	2	1
spec unclear, etc	21	13
clerical	15	9
other	3	2
unknown	2	1

TABLE 10. Error Causes

Table 10 shows the distribution of responses to the question "Why do you think the error was made?" Of the 160 errors, 68% were categorized by the programmer as programmer errors, 4% as a programmer misunderstanding of the language, 2% as due to a previous fix, and 1% as a human communcation error involving the programmer. Thus a total of 75% of the errors were categorized by the programmer as a failing of a programmer (as opposed to clerical errors or a failing of the specification writer). Such a high percentage of self-attributed errors was unexpected. Weiss [21] reported that up to 19% of the errors originally described as "clerical" by programmers were reclassified after validation. For about two-thirds (118) of our reported errors we also have information on whether the person correcting the error was the original writer of the code. Although only 10 errors out of 118 were not corrected by the original writer, in none of these 10 cases was the problem attributed to programmer error. The programmers were thus harsher in assigning blame to themselves than to their colleagues. We believe that these results reflect an especially well-motivated and mature group of programmers.

These "failing of a programmer errors" are not distributed evenly over all the reported errors. The programmers attributed 94% of "incorrect" code errors, 100% of "superfluous" code errors, but only 70% of "omitted" code errors to programmer mistakes. The remaining 30% of "omitted" errors was attributed to unclear, incomplete, or ambiguous specifications. Since omitted code errors are among the most difficult to detect and correct, this distribution shows the importance of producing

high-quality specifications before program design and coding begin.

5. Comparison to Other Studies

In several areas, results collected from the Univac editor project are very similar to those reported in other studies. These similar results encourage us to believe that there is some uniformity in the data collection methods, and validity to the data from which we expect to learn how to improve our software design and implementation methods.

The most striking common result running through the error categorization studies is the predominance of errors involving condition testing and associated processing. Furthermore, within the set of errors involving tests, a substantial number are errors of omission, where either an entire test was omitted, or a case was left out of a Boolean expression being tested.

A high proportion of test errors appears in all the projects which made up the TRW study. Counting test errors in the TRW study is not completely straightforward, since different categorization schemes were used. It is possible, however, to extract subcategories which represent the same types of errors we have classified as test-related. For the TRW projects, test-related errors ranged from 11% to 36% of the total of all recorded errors. Test-related errors of omission ranged from 65% to 96% of the test errors, and 8% to 33% of all recorded errors.

Other studies did not provide the detailed category breakdown needed to identify test-related errors. However, certain categories are quite similar, and there is always a large

number of decision-related and omitted decision errors. In Glass's study [10], out of 200 errors, 60 were classified as occurrences of "omitted logic", and 11 (possibly overlapping with the omitted logic category) as "IF statement too simple". Presson [16] found that "logic errors" accounted for 31% to 63% of the errors in the four development projects he analyzed. The next most frequent category, over all the projects, was "database errors" with 15% for one project. In the study by Mendis and Gollis [14], "logic errors" were 30% of the total; the two next most frequent were "computational errors" at 19% and "documentation errors" at 13%. Neither Presson nor Mendis and Gollis characterized errors as "omitted" or "incorrect".

In the three latter studies the class of "logic errors" contains more than what we have called "test-related errors". The term "logic" is interpreted in the broad sense of "design and implementation of the algorithm to solve the problem".

A second area where our results are similar to those of previous studies is the effort required to isolate and correct software errors. Weiss [21] and Presson have collected this type of information and the results can be directly compared. For comparison with Presson's work we combined our categories of under 1 hour and 1 to 4 hours. The overall trend is that the large majority of errors are both isolated and fixed very easily, each step usually requiring under four hours.

In Presson's projects, 57% to 89% of the errors were isolated in under four hours, while 76% to 96% were fixed within that time. The corresponding figures for the Univac project are 88% and 90%, respectively. Weiss monitored three projects at the

Software Engineering Laboratory and another at the Naval Research Laboratory. He noticed similar figures for isolation and correction effort, and with one exception, correction was again generally easier than isolation.

In an effort to determine if errors discovered later in the development process are more difficult to isolate and correct, we calculated the percentages for errors detected before function testing began, from the beginning of function testing to the beginning of system testing, and after the beginning of system testing. Since some of these activities were going on in parallel, this produces a different breakdown of errors than the one in Tables 8 and 9, which is according to the type of testing which detected the error. There is, however, no apparent increase in difficulty. We intend to continue following the editor during use and maintenance and to watch for such a trend to develop.

6. Conclusions

The three specific goals mentioned in the Introduction have been partially achieved by this study.

First, we have studied the occurrence of error types and the difficulty of error correction in relation to distinct steps of the software development process. Some of the unexpected results are that unit testing was very effective in detecting data handling errors, but considerably less effective than function testing in detecting data definition errors and errors involving a test and associated processing. During unit testing, error isolation tended to be more difficult than error fixing, while

during function testing this trend reversed and error fixing was more difficult than error isolation.

Second, we found strong agreement with some results reported in earlier error studies. In particular, all studies which categorized errors found that errors involving test predicates and combinations of tests and processing were by far the most common type. Among these, errors of omission have always been a large majority, leading to doubts about the value of test generation techniques such as branch coverage that are based solely on the existing program code.

Third, the attribute categorization scheme was a useful and practical method for classifying the errors reported in this study. The scheme needs further development to make it applicable to broader types of errors, and we expect to add attributes as they are needed when we accumulate additional error descriptions.

Acknowledgement: We are very grateful for the cooperation and assistance of Bob Hux, Bettye Scott, Jim Paul, Chris Rowan, Reggie King, and Joanne Heimbrook of Sperry Univac. Without their help the study could not have been done. Jim Paul was especially generous with his time and willingness to explain details of the editor system and to clarify error reports. We are also grateful to Max Goldstein of the Courant Institute for providing us access to the Ingres database system.

References

[1] Acree, A.T., R.A. DeMillo, T.J. Budd, R.J. Lipton, and F.G. Sayward, "Mutation Analysis", Technical Report GIT-ICS-79/08, Georgia Inst. Tech., Sept. 1979.

[2] Amory, W. and J.A. Clapp, "A Software Error Classification Methodology", MTR-2648, Vol. VII, Mitre Corp., Bedford, MA, 30 June 1973.

[3] Baker, W.F., "Software Data Collection and Analysis: A Real-Time System Project History", RADC-TR-77-192, Rome Air Development Center, Griffis AFB, NY, June 1977.

[4] Basili, V.R., "Data Collection Validation and Analysis", Draft Software Metrics Panel Final Report, ed. A.J. Perlis, F.G. Sayward, and M. Shaw, Washington, DC, 30 June 1980.

[5] Basili, V.R. and D.M. Weiss, "Analyzing Error Data in the Software Engineering Laboratory", Fourth Minnowbrook Workshop on Software Performance Evaluation, Blue Mtn. Lake, NY, August 1981.

[6] Basili, V.R., M.V. Zelkowitz, F.E. McGarry, R.W. Reiter, W.F. Truszkowski, and D.M. Weiss, "The Software Engineering Laboratory", Tech. Report TR-535, U. Maryland Computer Science Center, College Park, MD, May 1977.

[7] DeMillo, R.A., R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", Computer, 11(4), April 1978, 34-41.

[8] Endres, A., "An Analysis of Errors and Their Causes in System Programs", IEEE Trans. Softw. Eng., Vol SE-1, June 1975, 140-149.

[9] Foster, K.A., "Error-Sensitive Test Cases Analysis", IEEE Trans. Softw. Eng., Vol. SE-6, May 1980, 258-264.

[10] Glass, R.L., "Persistent Software Errors", IEEE Trans. Softw. Eng., Vol. SE-7, March 1981, 162-168.

[11] Goodenough, J.B., "The Ada Compiler Validation Capability", Proc. ACM SIGPLAN Symp. on the Ada Programming Language, Boston, Dec. 1980.

[12] Howden, W.E., "Completeness Criteria for Testing Elementary Program Functions", U. Victoria Dept. of Mathematics, DM-212-IR, May 1980.

[13] Litecky, C.R. and G.B. Davis, "A Study of Errors, Error-Proneness, and Error Diagnosis in Cobol", Comm. ACM, Vol. 19, January 1976, 33-37.

[14] Mercer, M.L. and M.L. Soffa, "Detecting and Predicting Errors in Software Programs", Proc. 2nd Ann. Computers in Aerospace Conf., Los Angeles, October 1979, 220-226.

[15] Osterweil, T. and E. Weyuker, "Error-Based Program Testing", Proc. 1979 Conf. Inf. Sciences and Systems, Baltimore, MD, March 1979.

[16] Presson, P.E., "A Study of Software Errors on Large Aerospace Projects", Proc. Nat. Conf. on Software Technology and Management, Alexandria, VA, October 1978.

[17] Schneidewind, N. and H. Hoffman, "An Experiment in Software Error Data Collection and Analysis", IEEE Trans. Softw. Eng., Vol SE-5, May 1979, 276-286.

[18] Thayer, T.A., M. Lipow, and E.C. Nelson, Software Reliability, TRW Series of Software Technology, Vol. 2, North-Holland, Amsterdam, 1978.

[19] Thibodeau, P., "The State-of-the-Art in Software Error Data Collection and Analysis - Final Report", General Research Corp., Huntsville, AL, Jan. 31, 1978.

[20] Weiss, D.M., "Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility", J. Systems and Software, Vol. 1, 1979, 57-70.

[21] Weiss, D.M., "Evaluating Software Development by Analysis of Change Data", Tech. Report TR-1120, U. Maryland Computer Science Center, College Park, MD, November 1980.

N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York N. Y. 10012