AD-A118 296 NAVAL RESEARCH LAB WASHINGTON DC F/G 9/2
A COMPARISON OF ERRORS IN DIFFERENT SOFTWARE-DEVELOPMENT ENVIRO--ETC(U)
JUL 82 D M WEISS
UNCLASSIFIED NRL-8598 NL

END
DATE FILMED
09-82
DTIC

AD A118295

NRL Report 8598

A Comparison of Errors in Different Software-Development Environments

DAVID M. WEISS

Computer Science and Systems Branch
Information Technology Division

July 14, 1982

NAVAL RESEARCH LABORATORY
Washington, D.C.

DTIC FILE COPY

DTIC
ELECTE
AUG 17 1982
S D
A

Approved for public release; distribution unlimited.

82 08 17 018

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NRL Report 8598	2. GOVT ACCESSION NO. AD-A118 296	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A COMPARISON OF ERRORS IN DIFFERENT SOFTWARE-DEVELOPMENT ENVIRONMENTS		5. TYPE OF REPORT & PERIOD COVERED Interim report on a continuing NRL problem 6. PERFORMING ORG. REPORT NUMBER 7590-097:DW:rmr
7. AUTHOR(s) David M. Weiss		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Research Laboratory Washington, DC 20375		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N; RR0140941; NRL Problem 75-0199-0-2
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Naval Research Laboratory Washington, DC 20375		12. REPORT DATE July 14, 1982 13. NUMBER OF PAGES 18
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software measurement
Software error data
Software error analysis
Software development methodology evaluation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Error detection and error correction are now considered to be the major cost factors in software development. Much current and recent research has been devoted to finding ways to prevent software errors. The purpose of this paper is to compare error data obtained from two different software-development environments using different software-development methodologies The data are used to characterize the similarities and differences in the environments and may be used to evaluate the success with which different methodologies meet the claims made for them. Data were obtained by the use of a goal-directed data-collection process, which is described briefly. A key feature of the process is that data are collected and validated (continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (Continued)

concurrently with software development. Validation often involves interviewing the programmers supplying the data. The results are data distributions across categorizations, such as effort to correct error, type of error, and locality of error. The distributions show that in both environments the principal error source was in the design and implementation of single routines. Requirements misunderstandings, specifications misunderstandings, and interface misunderstandings were all minor sources of errors. Few errors were the result of changes, few errors required more than one attempt at correction, and few error corrections resulted in other errors. Most errors were correctable in a day or less.

CONTENTS

INTRODUCTION 1
RESEARCH METHODOLOGY 1
PROJECTS STUDIED 2
ARCHITECTURE RESEARCH FACILITY 2
SOFTWARE ENGINEERING LABORATORY 4
RESULTS 4
CONCLUSIONS 9
ACKNOWLEDGMENTS 14
REFERENCES 14

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
A

A COMPARISON OF ERRORS IN DIFFERENT SOFTWARE-DEVELOPMENT ENVIRONMENTS

INTRODUCTION

According to the mythology of computer science, the first computer program ever written contained an error. Error detection and error correction are now considered to be the major cost factors in software development [1-3]. Much current and recent research has been devoted to finding ways to prevent software errors. One result is that techniques claimed to be effective for preventing errors are in abundance. Unfortunately, there have been few attempts to verify empirically that proposed techniques work well in production environments. The purpose of this report is to compare error data obtained from two different software-development environments using different software-development methodologies.

To obtain data that were complete, accurate, and meaningful, a goal-directed data-collection methodology was used. The approach was to monitor changes made to software concurrently with its development. The results reported here were obtained by applying the methodology to three projects at NASA's Goddard Space Flight Center (GSFC) and one project at the Naval Research Laboratory (NRL). Although all changes were monitored for most projects, we are concerned here only with results obtained from the error data and only with data that may be used to compare the two environments. Readers interested in a more detailed description of the research methodology or in analyses using other data from the same sources can find these in Refs. 4 through 6.

RESEARCH METHODOLOGY

The methodology is goal oriented. It starts with a set of goals to be satisfied, generates a set of questions to be answered, and proceeds step by step through the design and implementation of a data collection and validation mechanism. Analysis of the data yields answers to the questions, and it may also yield a new set of questions. The procedure relies heavily on an interactive data-validation process; those supplying the data are interviewed for validation purposes concurrently with the software-development process. The methodology has six basic steps, as described in the following paragraphs.

Establish the Goals of the Data Collection

Goals are usually related to claims made for the software-development methodology being used. As an example, a goal of a particular methodology might be to develop software that is easy to change. The corresponding data-collection goal is to evaluate the success of the developers in meeting this goal, i.e., to evaluate the ease with which the software can be changed.

Develop a List of Questions of Interest

Once the data collection goals are established, they are used to develop a list of questions to be answered. In general, each goal will result in the generation of several different questions of interest. For example, if the goal is to evaluate the ease with which software can be changed, we may identify questions of interest, such as: Is it clear where a change has to be made? Are changes confined to single modules? What was the average effort involved in making a change?

Manuscript submitted March 9, 1982.

Establish Data Categories

Each question of interest generally induces a categorization scheme on the data to be collected. If one question is how many errors result from requirements changes, then one will want to categorize errors according to whether or not they are the result of a change in requirements.

Design and Test Data Collection Forms

To provide a permanent copy of the data and to reinforce the programmers' memories, a data collection form is used. Form design was one of the trickiest parts of the studies conducted, and it will not be discussed here.

Collect and Validate Data

Data are collected by requiring those people who are making software changes to complete a change report form, for each change made, as soon as the change is completed. Validation consists of checking the forms for correctness, consistency, and completeness and interviewing those filling out the forms in cases where such checks reveal problems. Both collection and validation are concurrent with software development.

Analyze the Data

Data are analyzed by calculation of the parameters and distributions needed to answer the questions of interest.

To apply the methodology to the collection of change data, the following definitions were used:

- A *change* is an alteration to baselined design, code, or documentation.
- An *error* is a discrepancy between a specification and its implementation.
- A *modification* is a change made for any reason other than to correct an error.

PROJECTS STUDIED

The studies reported here contain complete results from four different projects. Two different environments and several different methodologies were used. One environment was a research group at NRL, and the other was a NASA software-production environment at GSFC. Table 1 is an overview of the data collected for each project. For the Architecture Research Facility project, only error data were collected. Table 2 gives the values of parameters often used to characterize software-development projects. It shows several different measures of size and effort for each project. Size is shown as total number of lines of code, number of developed lines of code (code that was either new or modified), and number of components, where a component is either a FORTRAN subroutine or block data.

ARCHITECTURE RESEARCH FACILITY

The purpose of the Architecture Research Facility (ARF) project, developed at NRL, was to develop a facility for simulating different computer architectures. The simulation is based on a description of the target architecture written in the Instruction Set Processor (ISP) language [7]. A complete description of the ARF simulator is available elsewhere [8]. Briefly, to simulate a machine the ARF uses a set of tables that describe the machine being simulated and its state, a module to perform instruction simulation, and a module to handle the interface to the user. The machine description contained in the tables is produced by an ISP compiler (an existing compiler was used). The ARF developers had no previous experience with such simulators.

Table 1 — Overview of Data Collected

Project	Number of Changes	Number of Modifications	Number of Errors
SEL1	281	101	180
SEL2	229	110	119
SEL3	760	453	307
ARF			143

Table 2 — Summary of Project Information

Project	Effort (months)	Number of Developers	Lines of Code (k)	Dev. Lines of Code (k)	Number of Components
SEL1	79.0	5	50.9	46.5	502
SEL2	39.6	4	75.4	31.1	490
SEL3	98.7	7	85.4	78.6	639
ARF	44.3	9	21.8	21.8	253

The primary goal of the ARF designers was to produce a working simulator that would permit the simulation of small target-machine programs. The designers also viewed the ARF development as an experiment in the application of recently-developed software engineering technology [8]. The key parts of the technology used were the following:

- Rather than developing the whole system at one time, the ARF project used the family approach to software development [9]. The designers planned to build the system in three main stages. Each stage was to produce a member of the ARF "family" of programs, providing different facilities.
- The information-hiding principle [10] was applied to conceal design decisions that were expected to change during the lifetime of the ARF.
- Informal design specifications, standardized interface specifications, and high-level language coding specifications were written for each major module of the ARF before any code was written. Each specification was reviewed before its successor was produced.
- FORTRAN code was written from the coding specifications, compiled, and then reviewed by someone other than the coder prior to debugging. The coder debugged the code and delivered it for testing. A tester, usually someone other than the coder or designer, was selected to test the debugged code.
- At the possible expense of some run-time performance, several debugging aids were designed into the system to make development easier. These included a method for detecting and reporting errors involving improper access to table entries and a mechanism for inserting, and turning on and off, debugging code through the use of a compile-time preprocessor.

SOFTWARE ENGINEERING LABORATORY

The Software Engineering Laboratory (SEL) is a NASA-sponsored project to investigate the software-development process and is based at GSFC. A number of different software-development projects are being studied as part of the SEL investigations [11,12]. Studies of changes made to the software as it is being developed constitute one part of those investigations.

Typical projects studied by the SEL are medium-sized FORTRAN programs that compute the position (known as attitude) of unmanned spacecraft, based on data obtained from sensors on board the

spacecraft. Attitude solutions are displayed to the user of the program interactively on CRT terminals. Because the basic functions of these attitude-determination programs tend to change slowly with time, large amounts of design, and sometimes code, are often reused from one program to the next. They include subsystems to perform such functions as reading and decoding spacecraft telemetry data, filtering sensor data, computing attitude solutions based on the sensor data, and providing an (interactive) interface to the user.

Development is done by contract in a production environment, and it is often separated into two distinct stages. The first stage is a high-level design stage. The system to be developed is organized into subsystems and then further subdivided. For the purposes of the SEL, each named entity in the system is called a component. The result of the first stage is a tree chart showing the functional structure of the subsystem, in some cases down to the subroutine level; a system functional specification describing, in English, the functional structure of the system; and decisions as to what software may be reused from other systems.

The second stage consists of completing the development of the system. Different components are assigned to teams of programmers, who write, debug, test, and integrate the software. Before delivery, the software must pass a formal acceptance test. On some projects, programmers produce no intermediate specifications between the functional specifications produced as part of the first stage and the code. Some projects produce pseudo-code specifications for individual subroutines before coding them in FORTRAN. During the time that the SEL has been in existence, a structured FORTRAN preprocessor has come into general use.

In contradistinction to the ARF developers, NASA is not concerned with experimenting with new software-engineering techniques. It is concerned with improving its software development process by introducing techniques that have been shown to be better than those currently used. Nonetheless, the principal design goal of the major SEL projects is to produce a working system in time for a spacecraft launch. Results from SEL studies of three different NASA projects, denoted SEL1, SEL2, and SEL3, are included here.

RESULTS

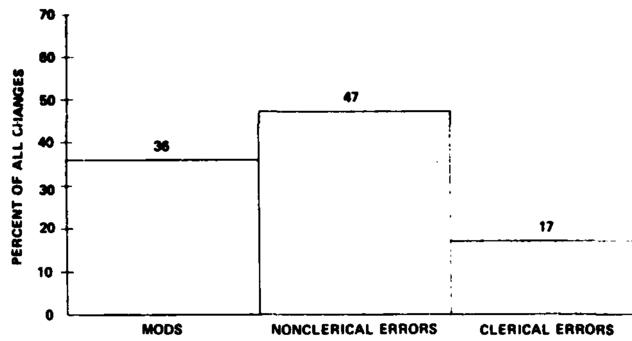
The results presented here are derived from analyses of several different data parameters and distributions. Table 3 shows error density, errors resulting from change, and repeated error ratio for each project. These parameters indicate that for all projects most changes were made correctly on the first attempt.

Figures 1 and 2 are an overview of the change distributions for the SEL projects (data on modifications are not available for the ARF project). Figure 3 shows sources of modifications, i.e., reasons for modifying the software; and Fig. 4 shows sources of nonclerical errors. Although there were a significant number of requirements changes for two of the SEL projects, none of the projects shows a significant number of errors resulting from incorrect or misunderstood requirements.

Table 3 — Measures of Erroneous Change

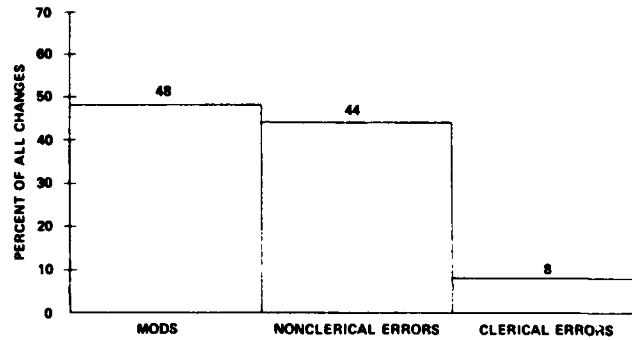
Project	Errors Per K Lines Of Developed Code	Errors Resulting From Change (as percentage of nonclericals)	Repeated Error Ratio (average number of corrections per error)
SEL1	3.9	5	1.02
SEL2	3.8	14	1.08*
SEL3	3.9	12	1.05
ARF	6.6	13	1.007

*Upper bound. Exact number of repeated errors for SEL2 is unknown. By conservative means, the ratio could be estimated as 1.04.



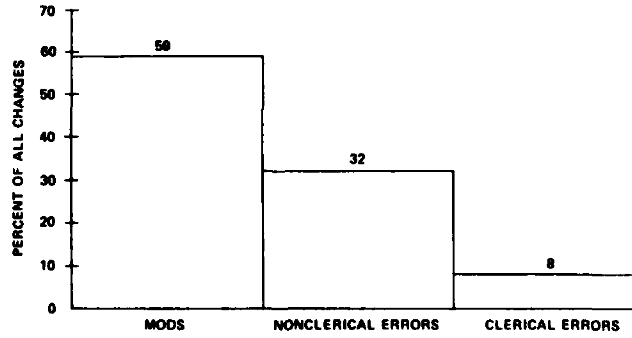
CHANGE TYPE

(a) SEL1



CHANGE TYPE

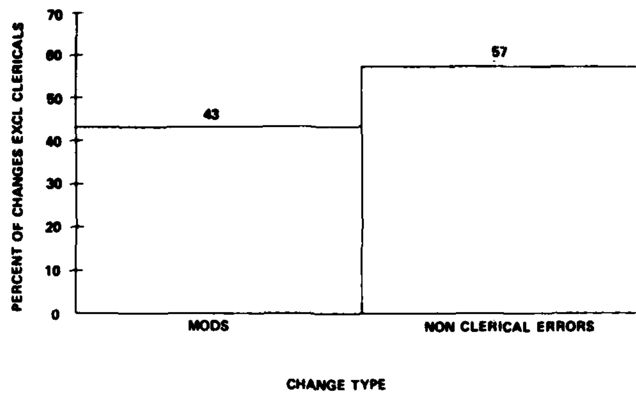
(b) SEL2



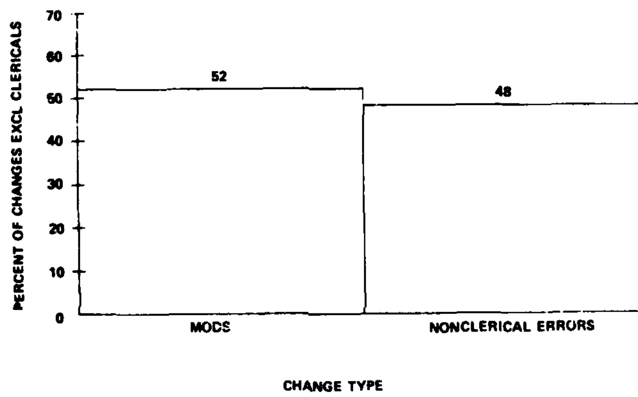
CHANGE TYPE

(c) SEL3

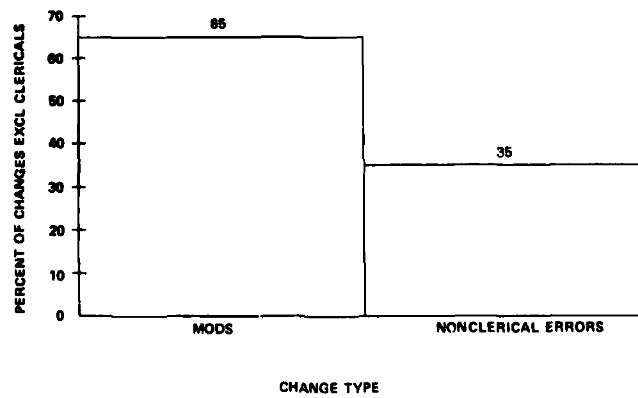
Fig. 1 — Changes



(a) SEL1

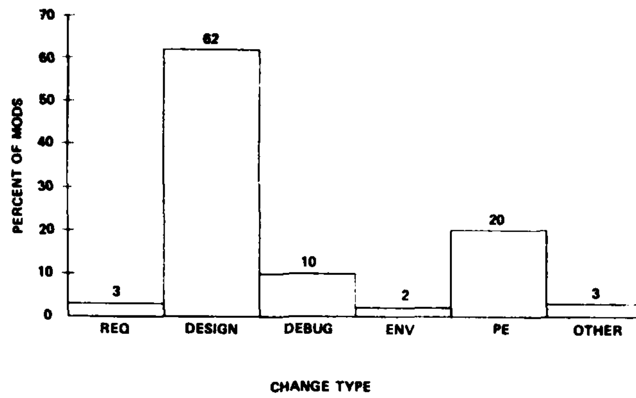


(b) SEL2

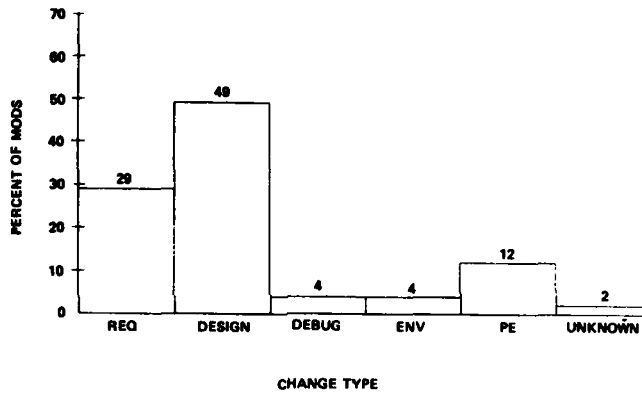


(c) SEL3

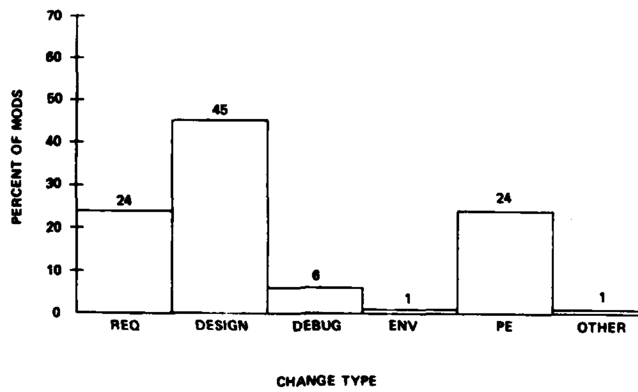
Fig. 2 — Changes with clerical errors excluded



(a) SEL1

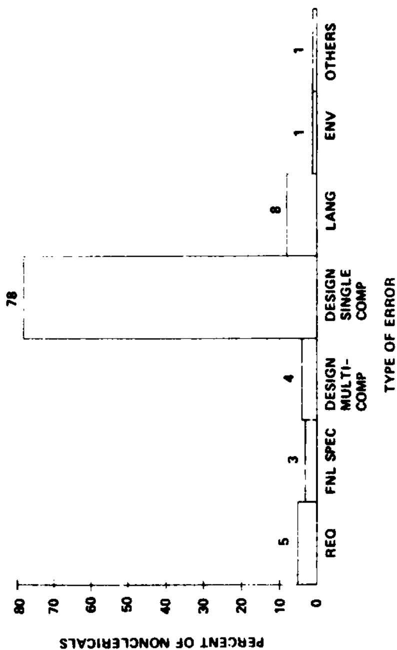


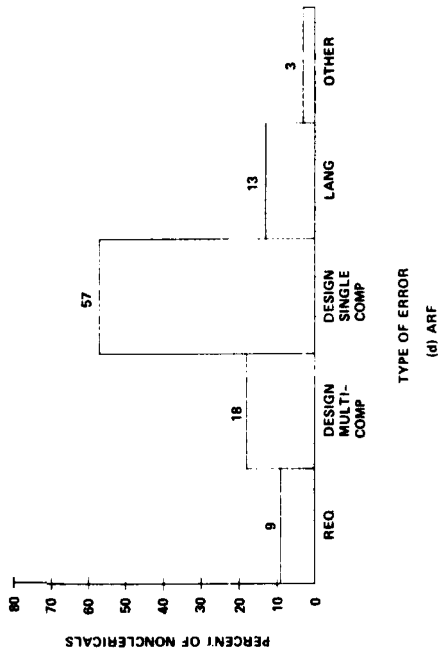
(b) SEL2

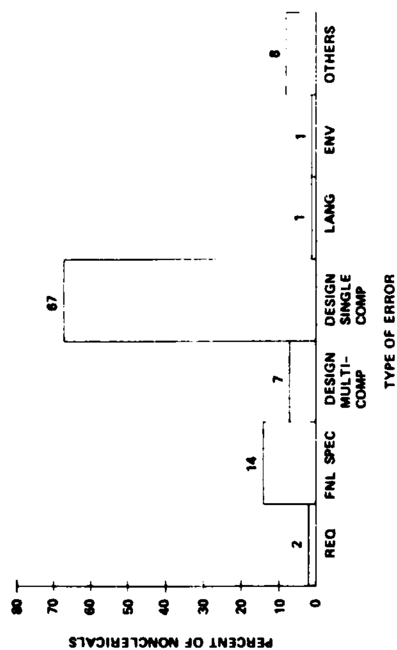


(c) SEL3

Fig. 3 — Sources of modifications: Design = modifications caused by changes in design; Debug = modifications to insert or delete debug code; Env = modifications caused by changes in the hardware or software environment; PE = planned enhancements; Req = modifications caused by changes in requirements or functional specifications; Unknown = causes of these modifications are not known







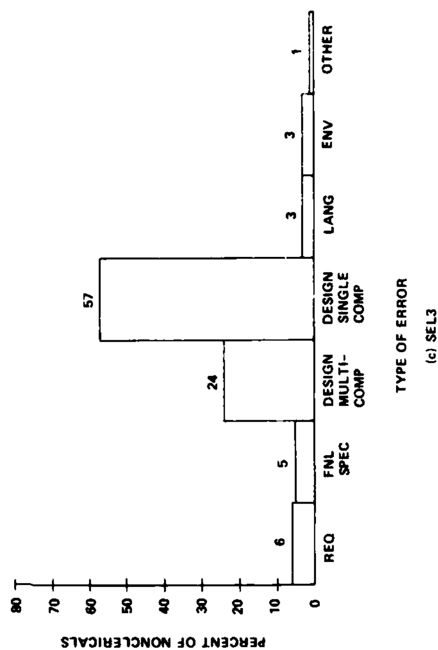


Fig. 4 — Sources of nonclerical errors: Design Multicomp = design error involving several components; Design Single Comp = error in the design or implementation of a single component; Env = misunderstanding of external environment, except language; Fnl Spec = functional specifications incorrect or misinterpreted; Lang = error in use of programming language or compiler; Req = requirements incorrect or misinterpreted

For all projects, the major source of errors was the design and implementation of single components (nearly always a FORTRAN subroutine or block data). Relatively few errors were the result of misunderstandings of requirements, specifications, programming language, or compiler or of software or hardware environment. Aspects of the design involving more than one component were also not a major source of errors. Figure 5 shows a continuation of the same pattern. For most projects, interfaces were not a significant source of errors.

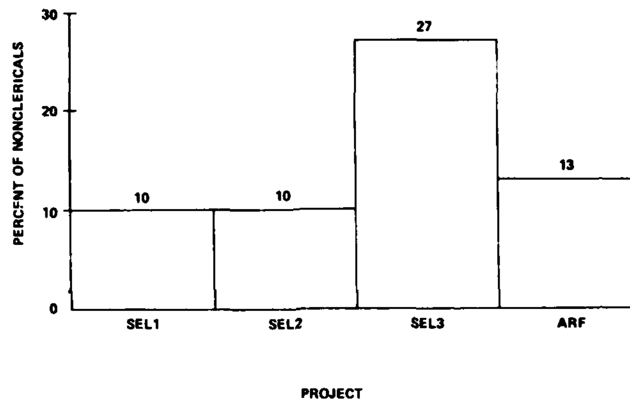


Fig. 5 — Interface errors

A further categorization of design and implementation errors, including both single-component and multicomponent design errors, is shown in Fig. 6. The pattern for the SEL and ARF projects is quite different here; relatively few ARF errors involved the use (including definition, representation, and access) of data. For the SEL projects, data errors were a significant fraction of design and implementation errors.

A direct measure of ease of error correction is shown in Fig. 7. For all projects, the overwhelming majority of errors took less than a day of effort to correct. Indeed, most error corrections took an hour or less of effort.

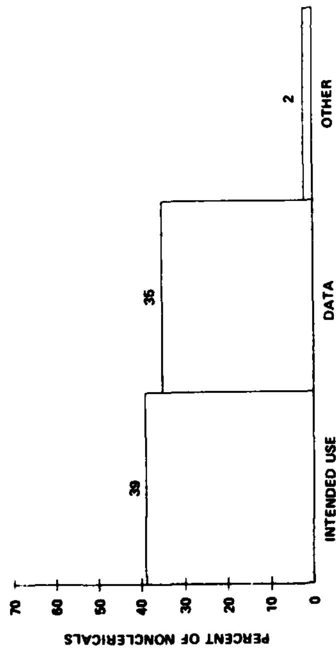
Figure 8 is a measure of locality of errors with respect to project components. Only components that required at least one error correction (one fix) are represented. The majority of these required no more than one correction. For all projects, 80% or more of such components were corrected at most three times.

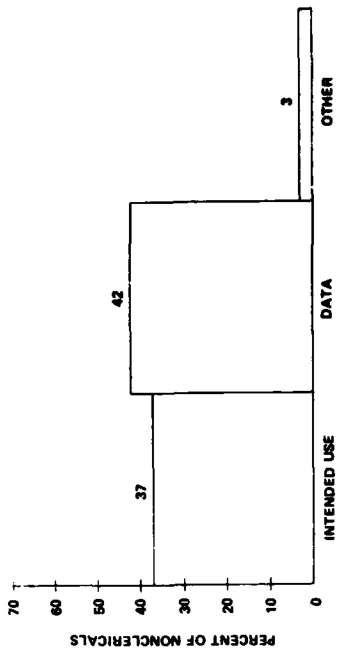
The locality of errors with respect to project subsystems (project module for the ARF) is shown in Fig. 9. The distributions here show the reverse pattern of those in Fig. 8; i.e., most corrections are clustered in a few subsystems (modules).

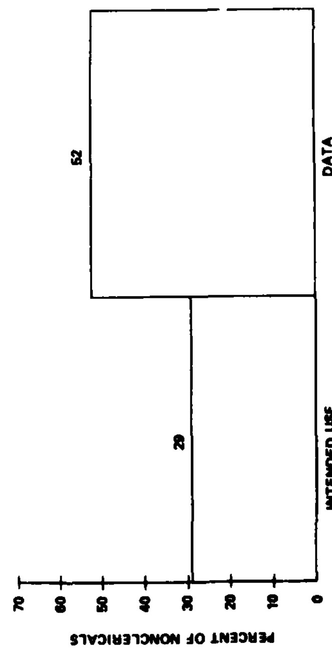
CONCLUSIONS

The ARF and SEL projects involved different applications and were developed in different environments, using different methodologies, people with different backgrounds, and different computer systems. Despite these differences, there are the following similarities between the two:

- There is a common pattern to the sources of error. The principle error source is in the design and implementation of single routines. Requirements, specifications, and interface misunderstandings are all minor sources of errors.







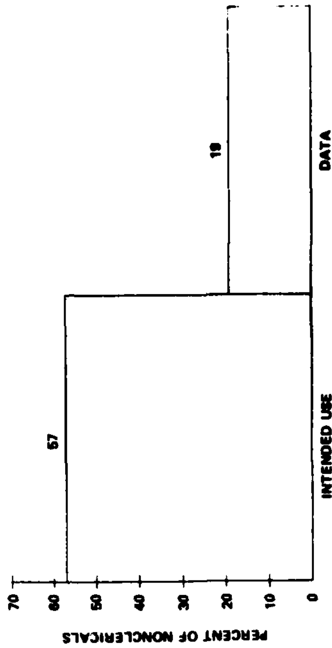
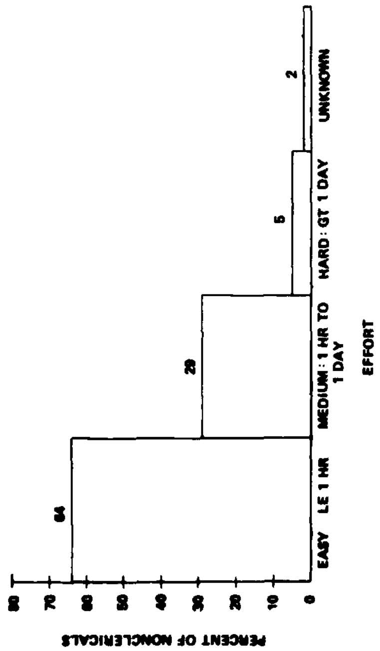
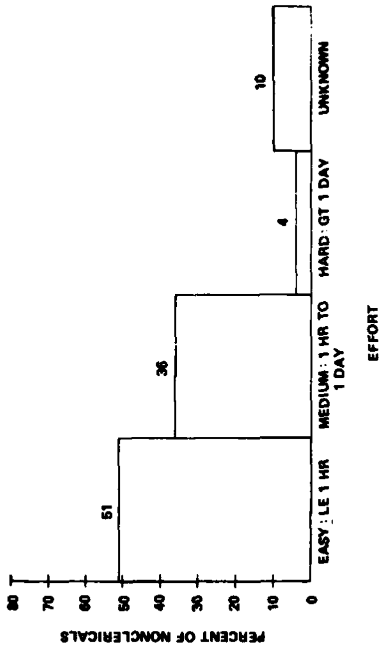


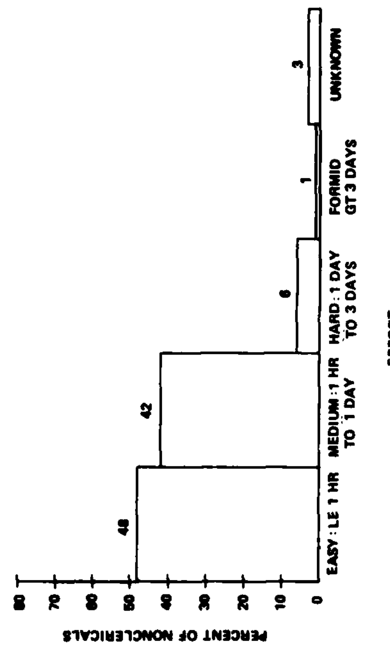
Fig. 6 — Sources of design and implementation errors: Data = error in the use of data; Intended Use = error in intended function, i.e., program behavior does not correspond to the intended use of the program.



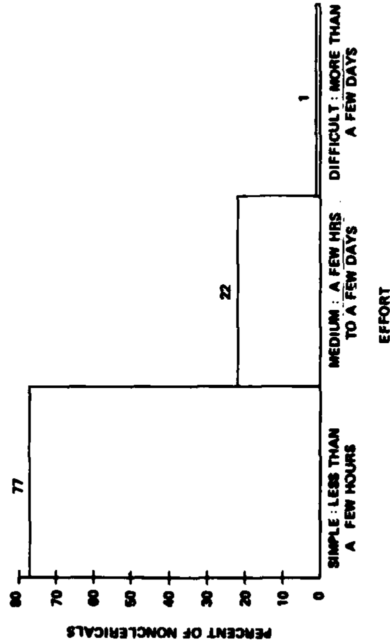
(a) SEL1 EFFORT TO DESIGN CHANGE



(b) SEL2 EFFORT TO DESIGN CHANGE



(c) SEL3 EFFORT TO MAKE CHANGE



(d) ARF EFFORT TO FIX

Fig. 7 — Effort to change nonclerical errors

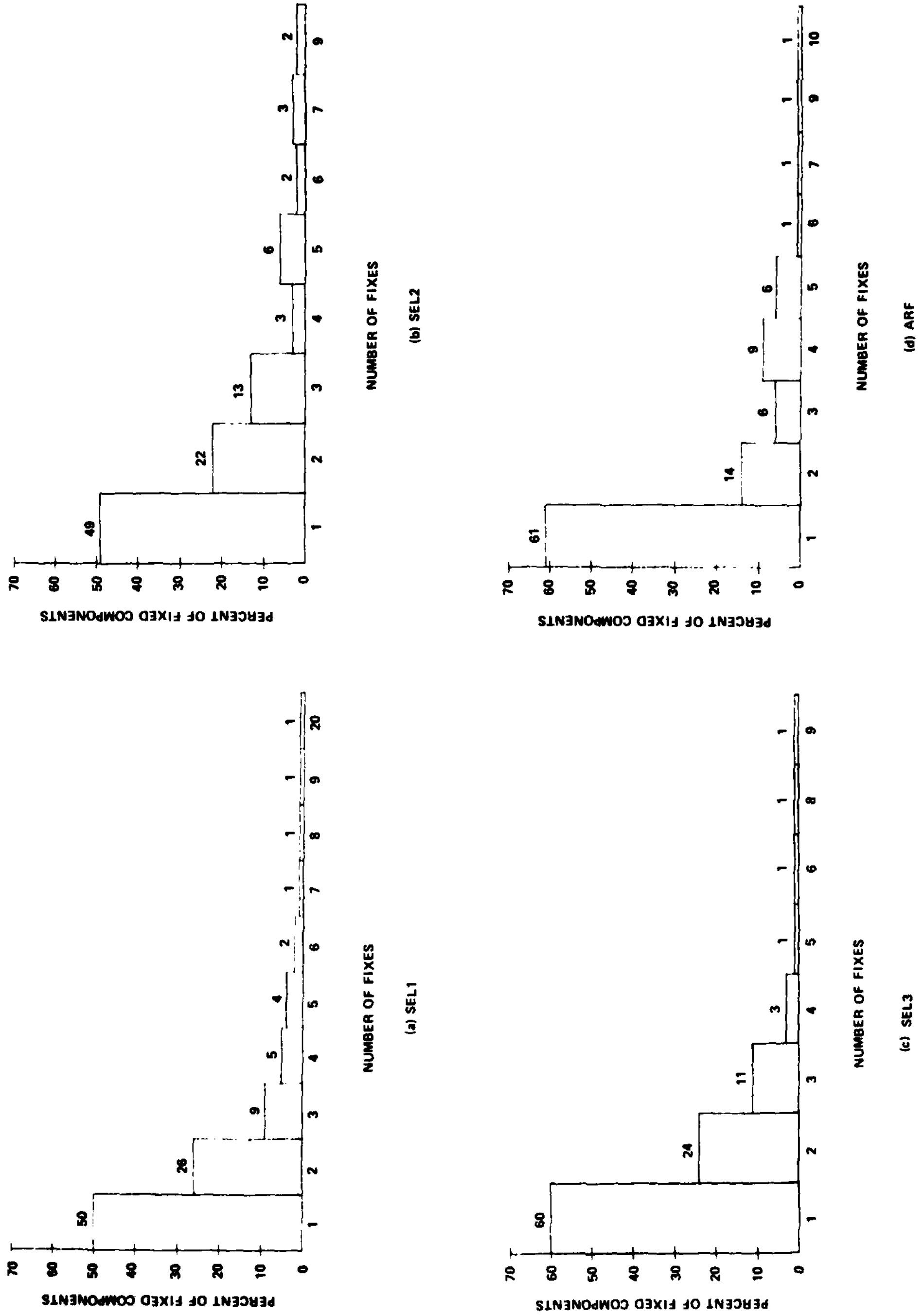
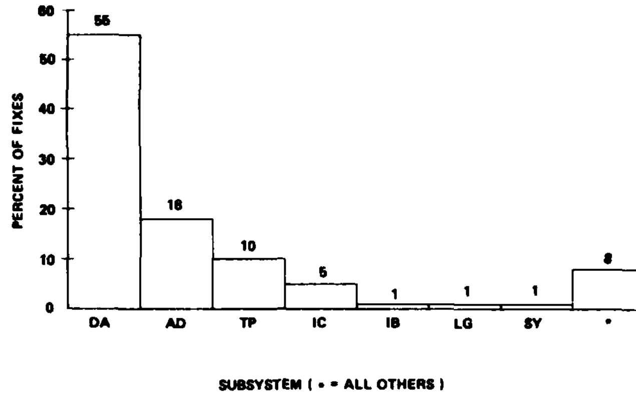
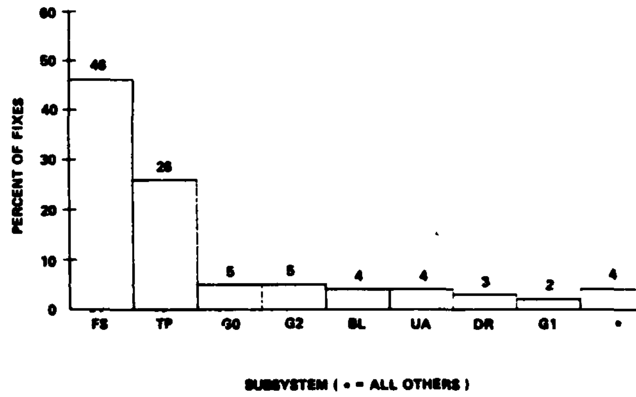


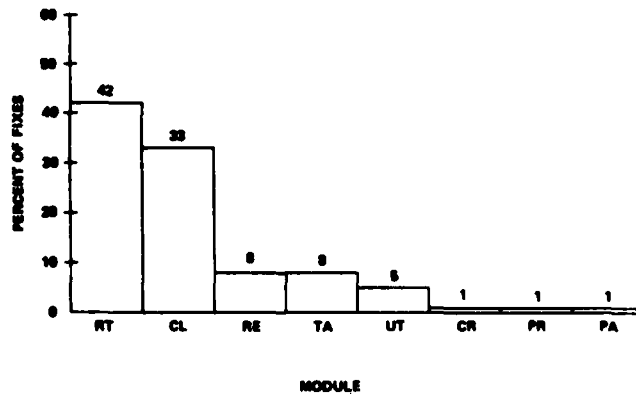
Fig. 8 — Frequency distribution of fixes



(a) SEL1 FIXES BY SUBSYSTEM



(b) SEL2 FIXES BY SUBSYSTEM



(c) ARF FIXES BY MODULE

Fig. 9 — Fixes by subsystem or module

- Few errors are the result of changes, few errors require more than one attempt at correction, and few error corrections result in other errors.

- Relatively few errors take more than a day to correct.

These similarities may be explained by different factors in the different environments. The SEL projects may be viewed as redevelopments. Much of the same design, and some of the same code, is reused from one project to the next. As a result of experience with the application, the changes most likely to occur from one project to the next have been identified by the designers. The systems are now designed so that these changes are easy to make. Confirmation of this explanation was provided by one of the primary system designers in discussions held after the data were analyzed. In the ARF environment, the explicit use of techniques to identify and design for potential changes is a likely contributing factor to the similarities in the distributions.

Common factors to both the SEL and ARF projects were the stability of the hardware and software supporting the development and the familiarity of the programmers with the language they were using.

The most striking difference between the ARF and SEL projects is in the proportion of intended-use errors to data errors. The ARF project has a considerably smaller proportion of data errors than the SEL projects. One reason for this may be the conscious attempt of the ARF developers to apply abstract data typing and strong typing in their design.

One might interpret the similarity in error distributions between the different environments as evidence that the conscious application of modern software-engineering technology may be the equivalent of considerable experience with a particular application.

ACKNOWLEDGMENTS

Support for a research project involving data collection in a production environment must come from many sources. These sources include project management, the programmers supplying the data, those maintaining the data base (in both paper and computerized form), those assisting in data analysis, and those providing technical review and guidance. A few of the people providing such support were Frank McGarry, Dr. Victor Basili, Dr. David Parnas, Dr. John Shore, Dr. Gerald Page, Honey Elovitz, Alan Parker, Jean Grondalski, Sam DePriest, Joanne, Shana, and Joshua Weiss, and Kathryn Kragh.

REFERENCES

1. B. Boehm et al., *Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980's (CCIP-85)*, Vol. 4, *Technology Trends: Software*, Space and Missile Systems Organization, Los Angeles, Feb. 1972.

2. B. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation* **19(5)**, 48-59 (May 1973).

3. R.W. Wolverton, "The Cost of Developing Large-Scale Software," *IEEE Trans. Comput.* **C-23(6)**, 615-636 (1974).

4. V.R. Basili and D.M. Weiss, "Evaluation of a Software Requirements Document by Analysis of Change Data," *Proceedings of the Fifth International Conference on Software Engineering*, 1981, pp. 314-323.

5. D.M. Weiss, "Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility," *J. Syst. Software* **1**, 57-70 (1979).

6. D.M. Weiss, "Evaluating Software Development by Analysis of Change Data," Ph.D. Thesis, University of Maryland, 1981.

7. C.G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971.

8. H. Elovitz, "An Experiment in Software Engineering: The Architecture Research Facility as a Case Study," *Proceedings of the Fourth International Conference on Software Engineering*, 1979, pp. 145-152.

9. D.L. Parnas, "On the Design and Development of Program Families," *IEEE Trans. Software Eng.* **SE-2**(1), 1-9 (1976).

10. D.L. Parnas, "A Technique for Software Module Specification with Examples," *Commun. ACM* **15**(5), 330-336 (May 1972).

11. J.W. Bailey and V.R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*, 1981, pp. 107-116.

12. V.R. Basili et al., Technical Report TR-535, The Software Engineering Laboratory, University of Maryland, May 1977.

9-8