EVALUATING SOFTWARE DEVELOPMENT CHARACTERISTICS: A Comparison Of Software Errors In Different Environments

David M. Weiss
Naval Research Laboratory

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 [Boe72, Boe73, Wol74]. Much current and recent research is devoted to finding ways to prevent sotware errors. One result is that techniques claimed to be effective for preventing errors are in abundance. Unfortunately, there have been few empirical attempts to verify that proposed techniques work well in production environments. Indeed, there have been few attempts even to collect data that could yield insight into the issues involved. The purpose of this paper is to compare error data obtained from two different software development environments.

To obtain data that was 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/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 other analyses using other data from the same sources are referred to [Bas81, Wei79, Wei81].

Research Methodology

The methodology is goal oriented. It starts with 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 of interest, and 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.

1. Establish the goals of the data collection.
   Many (but not all) of our goals are 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. evaluate the ease with which the software can be changed.

2. Develop a list of questions of interest
Once the goals of the study are established, they are used to develop a list of questions to be answered by the study. 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 a single modules?", "What was the average effort involved in making a change?"

3. Establish data categories
Once the questions of interest have been established, categorization schemes for the changes and errors to be examined may be constructed. Each question generally induces a categorization scheme. If one question is, "How many errors result from requirements changes?", one will want to classify errors according to whether or not they are the result of a change in requirements.

4. 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. Forms design was one of the trickiest parts of the studies conducted, and will not be discussed here.

5. 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.

6. Analyze the data
Data are analyzed by calculating 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.

The 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 the Naval Research Laboratory (NRL), and the other was a NASA software production environment at Goddard Space Flight Center. Table 1 is an overview of the data collected for each project. For the ARF project, only error data were collected. Table 2 gives the values of parameters often used to characterize software development projects.

The 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 language [Bel71]. A complete description of the ARF simulator is available elsewhere [Elo79]. 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 was developed by a team of nine people, not all full time. Development took about ten months and 192 people-weeks, exclusive of consulting and secretarial support, to develop. The delivered system contained about 20,000 lines of FORTRAN code.

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 software engineering technology [Elo79]. The key parts of the technology used are the following.

* Rather than developing the whole system at one time, the ARF was to be done using the family approach to software development [Par76]. The system was to be built in three main stages. Each stage would produce a member of the ARF "family" of programs, providing different facilities.
* The information-hiding principle [Par72a] was to be applied to conceal design decisions that were expected to change during the lifetime of the ARF.
* Informal design specifications, followed by standardized interface specifications, followed by 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) 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. A method for detecting errors involving improper access to table entries, known as the binding mechanism,
    b. A consistent execution-time error reporting scheme for table interface functions, and
    c. A mechanism for inserting, and turning on and off, debugging code through the use of a compile-time preprocessor.

The Software Engineering Laboratory

The Software Engineering Laboratory (SEL) is a NASA sponsored project to investigate the software development process, based at Goddard Space Flight Center (GSFC). A number of different software development projects are being studied as part of the SEL investigations [Bai81, Bas77]. 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 size 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 re-used from one program to the next. The programs range in size from about 20,000 to about 120,000 lines of source code. 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 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 period of time that the SEL has been in existence, a structured FORTRAN preprocessor has come into general use.

In distinction to the ARF developers, NASA is not concerned with experimenting with new software engineering techniques. It is concerned with introducing improved techniques into its software development process.

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.

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 1 Overview of Data Collected

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

Table 2 Summary of Project Information

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.

Table 3 Measures of Erroneous Change

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 (recall that data on modifications is not available for the ARF project). Figure 3 shows sources of modifications, i.e. reasons for modifying the software, and figure 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 show a significant number of errors resulting from incorrect or misunderstood requirements.

For all projects, the major source of errors was the design and implementation of single components. (For these projects, a single component is nearly always a FORTRAN subroutine or block data.) Relatively few errors were the result of misunderstandings of requirements, specifications, programming language or compiler, or software or hardware environment. Aspects of the design involving more than one component was 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.

A further categorization of design and implementations errors, including both single and multi-component design errors is shown in figure 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 figure 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 such components required no more than one correction. For all projects, 80% or more of such components were corrected at most three times.

Locality of errors with respect to project subsystem (project module for the ARF), is shown in figure 9. The distributions here show the reverse pattern of those in figure 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 methdologies, people with different backgrounds, and different computer systems. Despite these differences there are a number of similarities between the two, as listed in the following.

1. There is a common pattern to the sources of error distributions. The principle error source is in the design and implementation of single routines. Requirements, specifications and interface misunderstandings are all minor sources of errors.
2. Few errors are the result of changes, few errors require more than one attempt at correction, and few error corrections result in other errors.
3. 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 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.

Acknowledgements

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, Drs. Victor Basili, David Parnas, John Shore, and Gerald Page, Honey Elovitz, Alan Parker, Jean Grondalski, Sam DePriest, Joanne, Shana, and Joshua Weiss, and Kathryn Kragh.

References

[Bai81] J. Bailey and V. Basili, "A Meta-Model For Software Development Resource Expenditures," Proc. Fifth Int. Conf. Software Eng., pp. 107-116, 1981

[Bas77] V. Basili, M. Zelkowitz, F. McGarry, et al., The Software Engineering Laboratory, University of Maryland Technical Report TR-535, May 1977

[Bas81] V. Basili and D. Weiss, "Evaluation of a Software Requirements Document By Analysis of Change Data," Proc. Fifth Int. Conf. Software Eng., pp. 314-323, 1981

[Bel71] C. Bell and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, New York, 1971

[Elo79] H. Elovitz, "An Experiment In Software Engineering: The Architecture Research Facility As A Case Study, Proc. Fourth Int. Conf. Software Eng., pp. 145-152, 1979

[Par72a] D. L. Parnas, "A Technique For Software Module Specification With Examples," Comm. ACM, vol. 15 no. 5, May, 1972, pp. 330-336

[Par76] D. L. Parnas, "On the Design and Development of Program Families," IEEE Trans. Software Eng., vol. SE-2 no. 1, pp. 1-9, 1976

[Wei79] D. Weiss, "Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility," J. Systems and Software, vol. 1, pp. 57-70, 1979

[Wei81] D. Weiss, "Evaluating Software Development By Analysis Of Change Data," Ph.D. Thesis, University of Maryland, 1981

THE VIEWGRAPH MATERIALS
for the
D. WEISS PRESENTATION FOLLOW

PURPOSE OF RESEARCH

* FIND A WAY OF EVALUATING SOFTWARE DEVELOPMENT METHODOLOGIES

* LEARN ABOUT THE SOFTWARE DEVELOPMENT PROCESS

* LEARN ABOUT MEASURING THE SOFTWARE DEVELOPMENT PROCESS

APPROACH

* STUDY CHANGES USING GOAL-DIRECTED DATA COLLECTION

RESEARCH METHODOLOGY DEVELOPED

* ESTABLISH GOALS

  EXAMPLE: EVALUATE THE DIFFICULTY OF CHANGING SOFTWARE

* DEFINE QUESTIONS OF INTEREST

  EXAMPLES: 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?

* DESIGN DATA COLLECTION FORM

* COLLECT AND VALIDATE DATA CONCURRENTLY WITH DEVELOPMENT

* ANALYZE DATA

TYPES OF CHANGES

* DEF: A CHANGE IS AN ALTERATION TO (BASELINED) DESIGN, CODE, OR DOCUMENTATION.

* DEF: AN ERROR IS A DISCREPANCY BETWEEN A SPECIFICATION AND ITS IMPLEMENTATION.

* DEF: A MODIFICATION IS A CHANGE MADE FOR ANY REASON OTHER THAN TO CORRECT AN ERROR.

* CHANGES = MODIFICATIONS + ERROR CORRECTIONS

SUBCATEGORIES OF CHANGES

* MODIFICATIONS

  IMPLEMENTATION OF REQUIREMENTS CHANGE

  OPTIMIZATIONS

  IMPROVEMENTS OF USER SERVICES

  IMPROVEMENT OF CLARITY, MAINTAINABILITY, OR DOCUMENTATION

  ADAPTATION TO ENVIRONMENT CHANGE

* ERROR CORRECTIONS

  CLERICAL ERRORS

  NON-CLERICAL ERRORS

    REQUIREMENTS INCORRECT OR MISINTERPRETED

    SPECIFICATIONS INCORRECT OR MISINTERPRETED

    DESIGN ERROR INVOLVING SEVERAL COMPONENTS

    ERROR IN DESIGN/IMPLEMENTATION OF A SINGLE COMPONENT

    ERROR IN USE OF PROGRAMMING LANG OR COMPILER

    MISUNDERSTANDING OF ENVIRONMENT

Project	Number of Changes	Number of Modifications	Number of Errors
SEL1	281	101	180
SEL2	229	110	119
SEL3	760	453	307
ARF			143
A-7	88	9	79

Table 5.4a Overview of Data Collected

Project	Effort	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
A-7					

Table 5.4b Summary of Project Information

Project	Changes Per K Lines Of Developed Code	Errors Per K Lines Of Developed Code	Error To Mod Ratio (NonClericals Only)
SEL1	6.0	3.9	1.3
SEL2	7.4	3.8	.92
SEL3	9.7	3.9	.54
ARF		6.6	

Table 5.5 Change and Error Densities

Project	Erroneous Change Rate (Ratio Of Changes Resulting In Errors To All Changes)	Errors Resulting From Change (As Percentage Of NonClericals)	Repeated Error Ratio (Average Number Of Corrections Per Error)
SEL1	.025	5	1.02
SEL2	.061	14	1.08*
SEL3	.041	12	1.05
ARF		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.

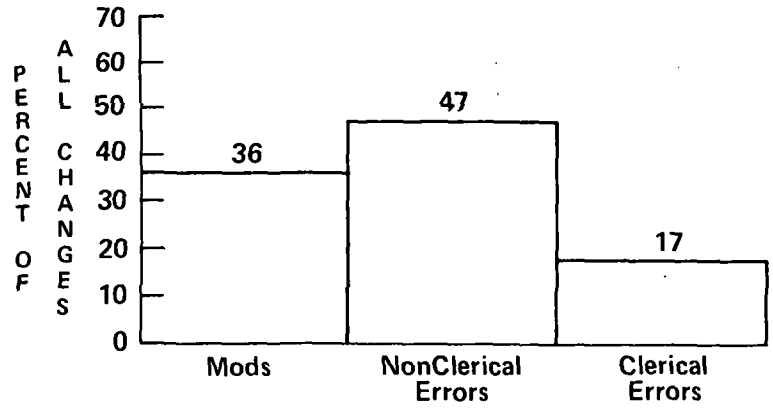
Table 5.6 Measures of Erroneous Change

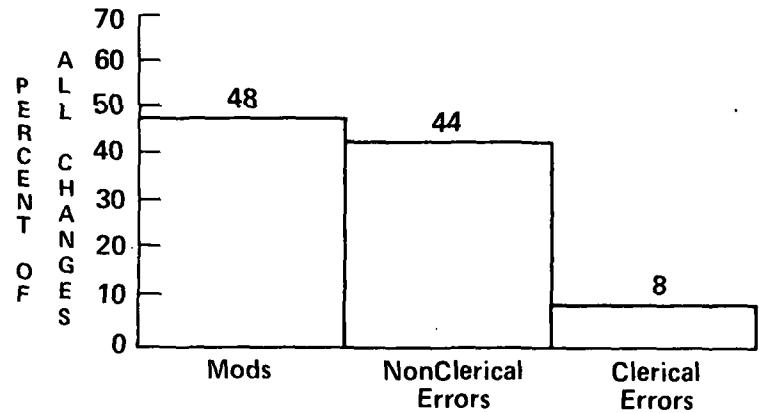
Project	Number Of People	Errors Per Person
SEL2	4	25
SEL1	5	26
SEL3	7	44
ARF	9	10

Table 5.7 Errors Per Person By Number Of People

Project	Effort (People-Months)	Errors Per Person-Month	Changes Per Person-Month
SEL2	39.6	2.4	5.8
ARF	44.3	2.1	
SEL1	79.0	1.7	3.6
SEL3	98.7	3.1	7.7

Table 5.8 Errors Per Effort By Effort





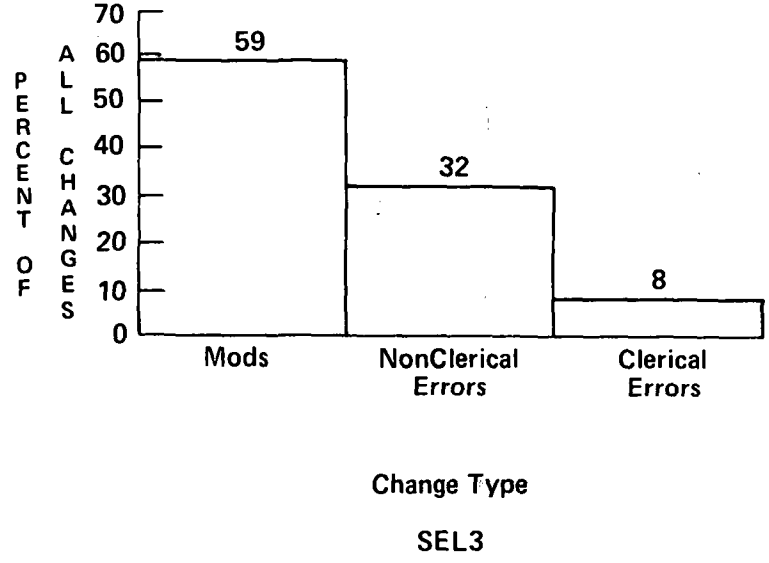


Figure 5.1 Changes.

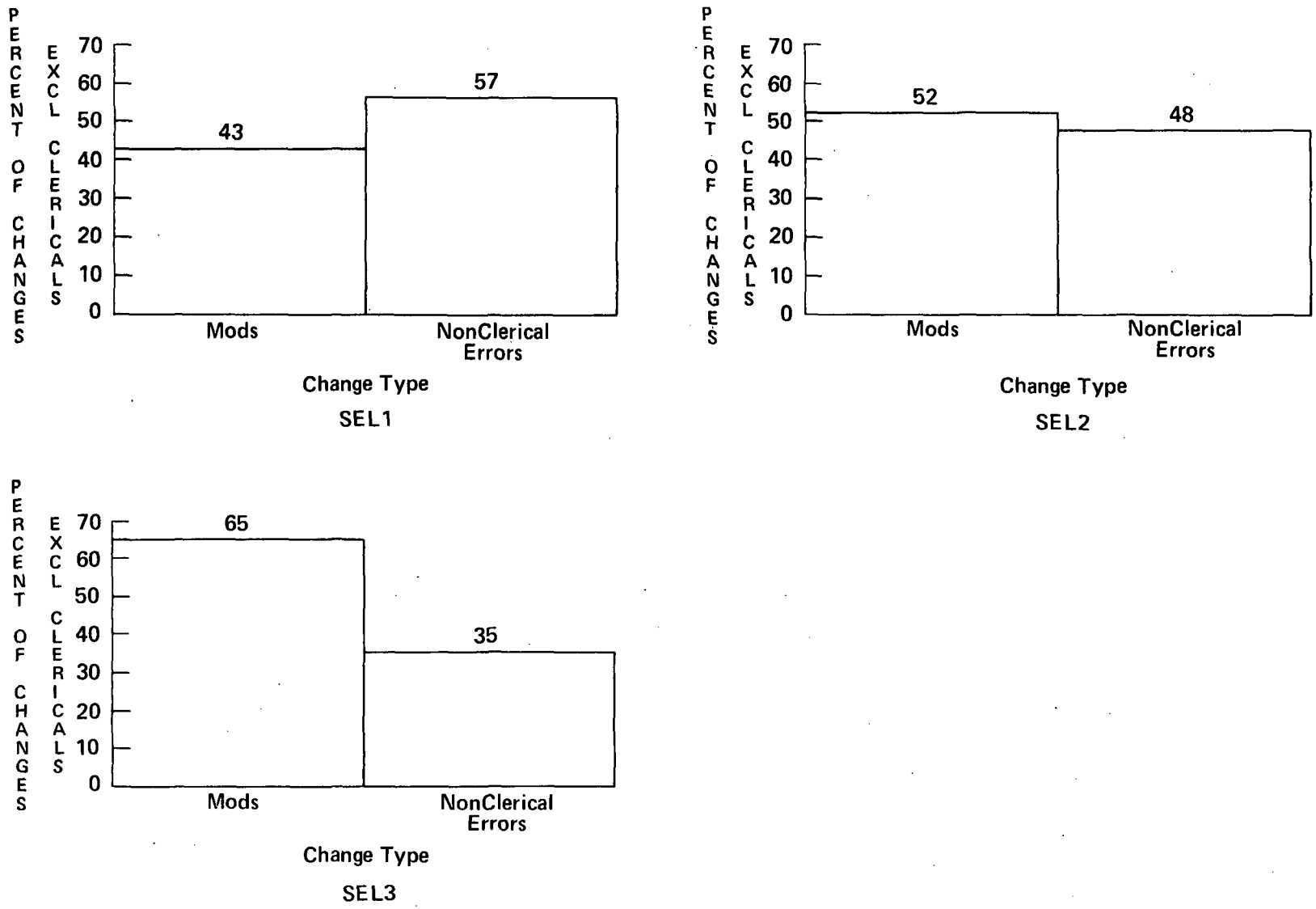
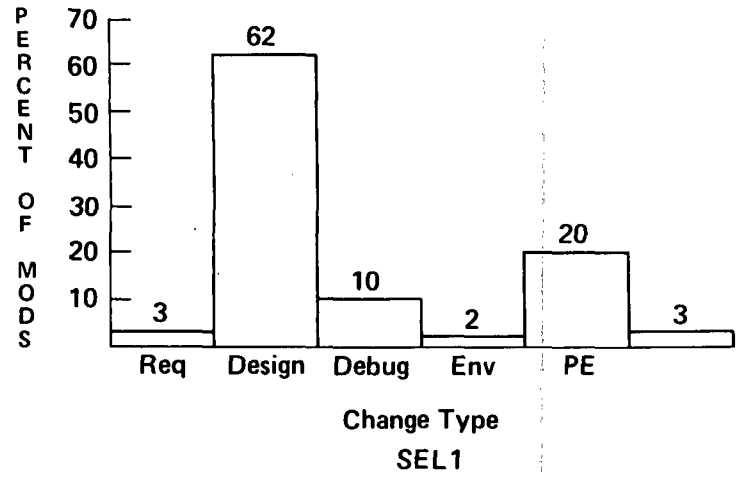
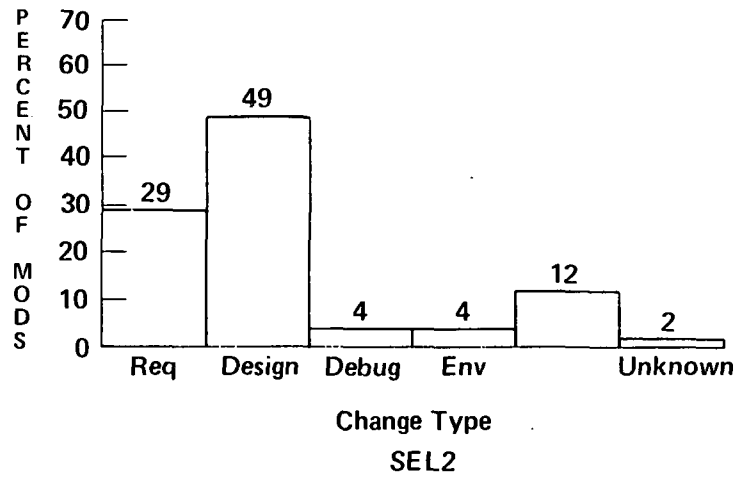
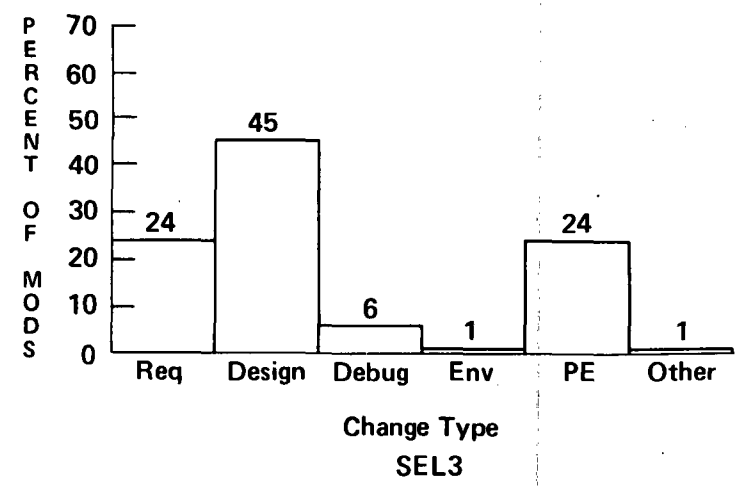


Figure 5.2 Changes (Clerical Errors Excluded).



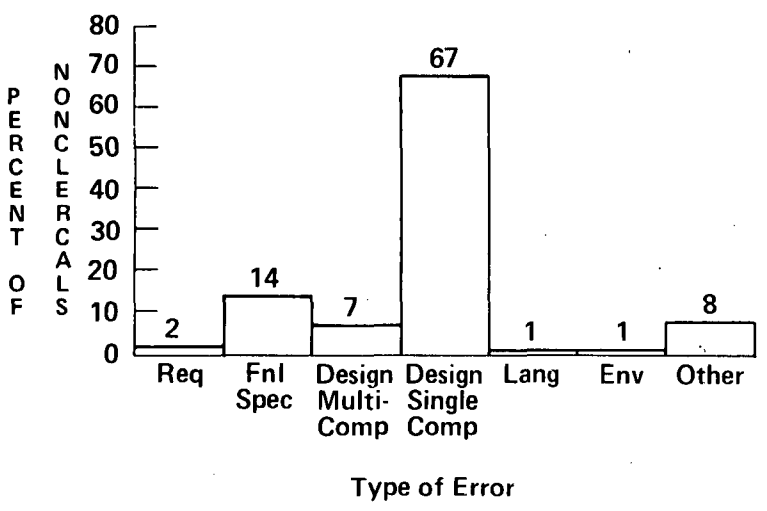


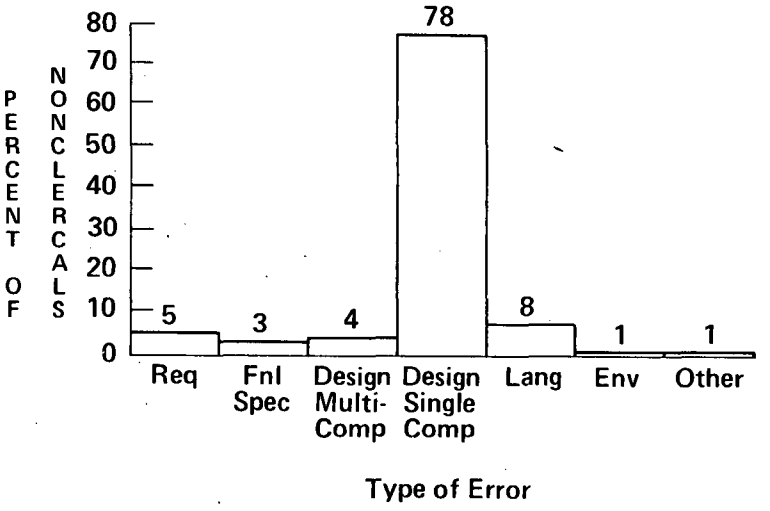


Key to Figure 5.3

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

Figure 5.3. Sources of Modifications.



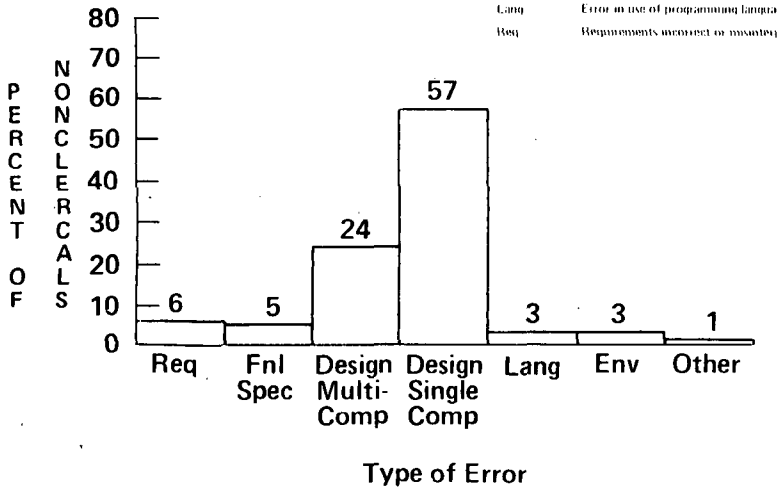


KeyTo Figure 5.5

Design	Design error involving several compn

Key To Figure 5.5

Design Multi Comp	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/compiler
Req	Requirements incorrect or misinterpreted



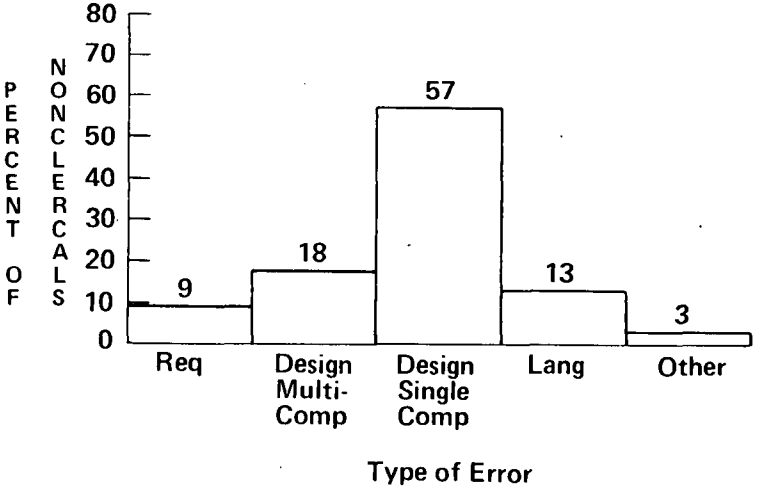


Figure 5.5. Sources of Nonclerical Errors.

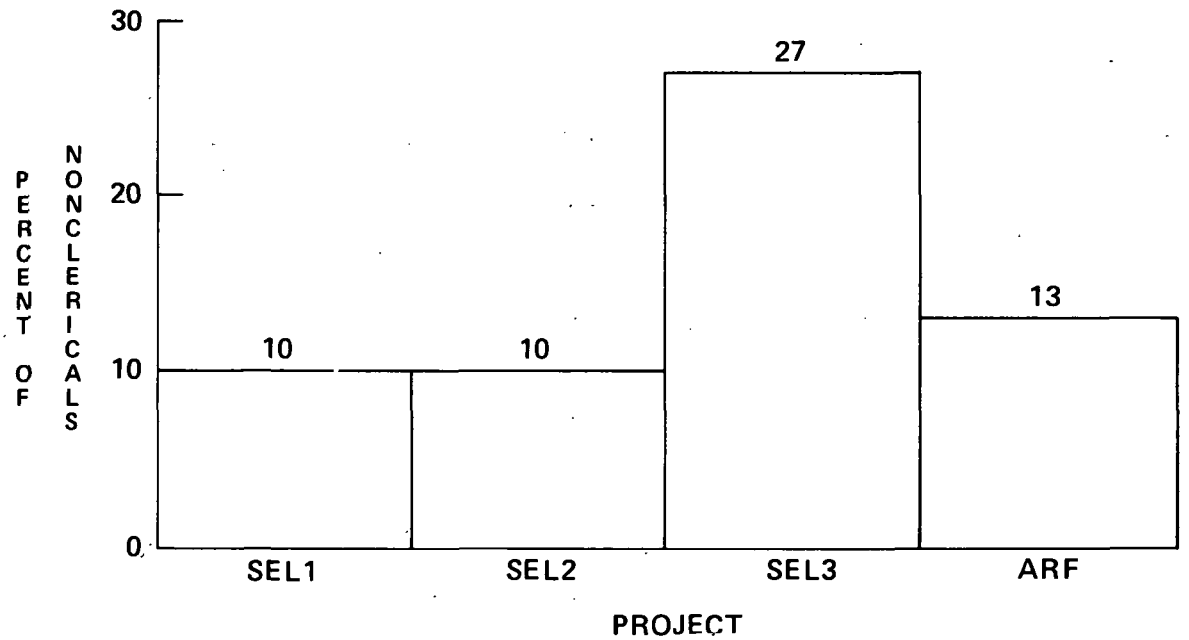
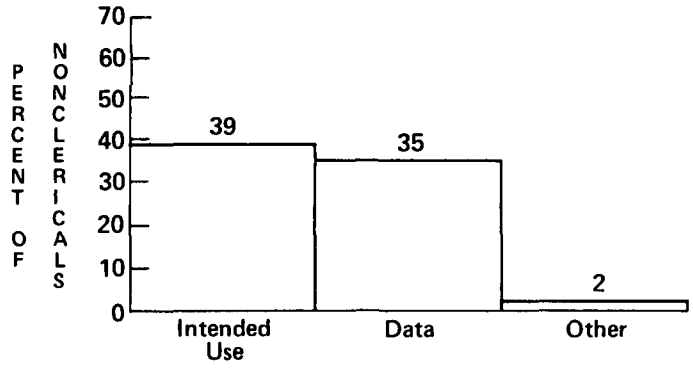
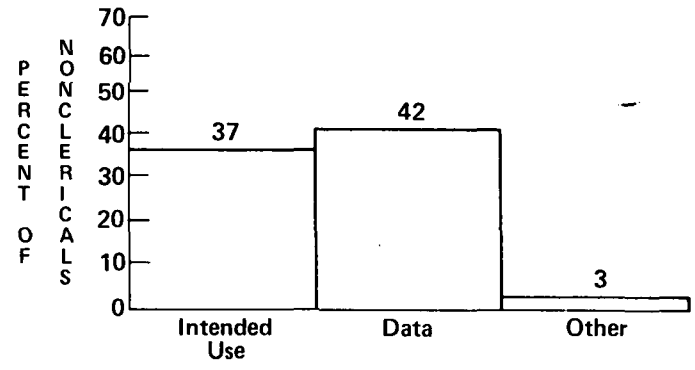


Figure 5.7 Interface Errors.



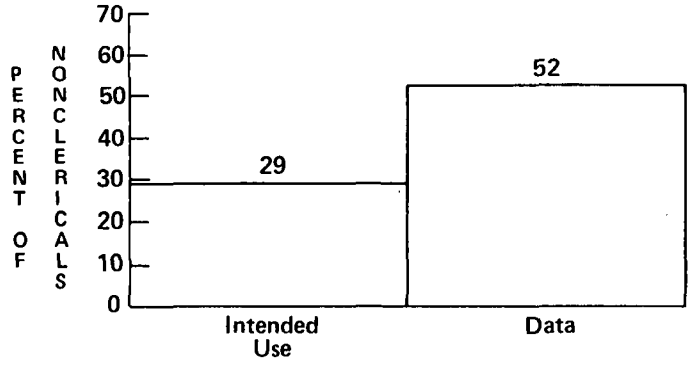
SEL1



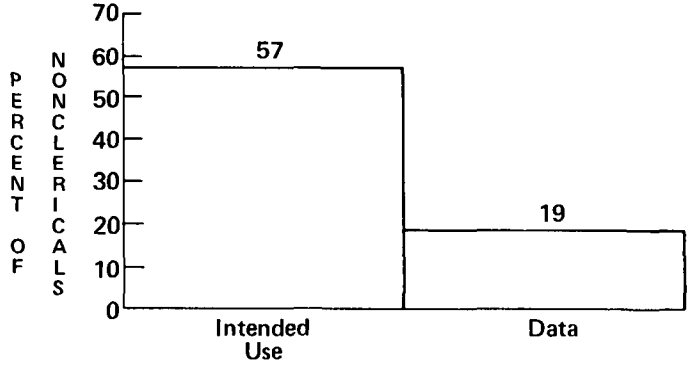
SEL2

Key to Figure 5.6

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

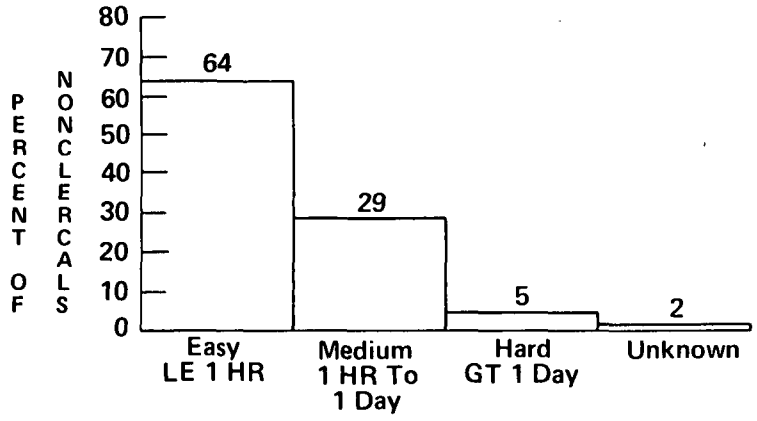


SEL3

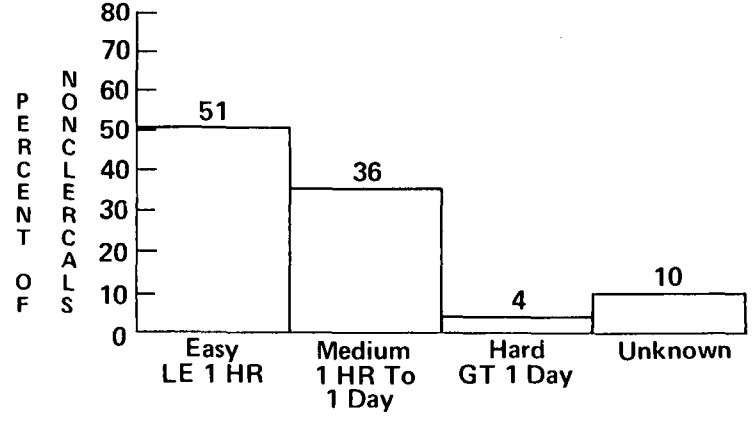


ARF

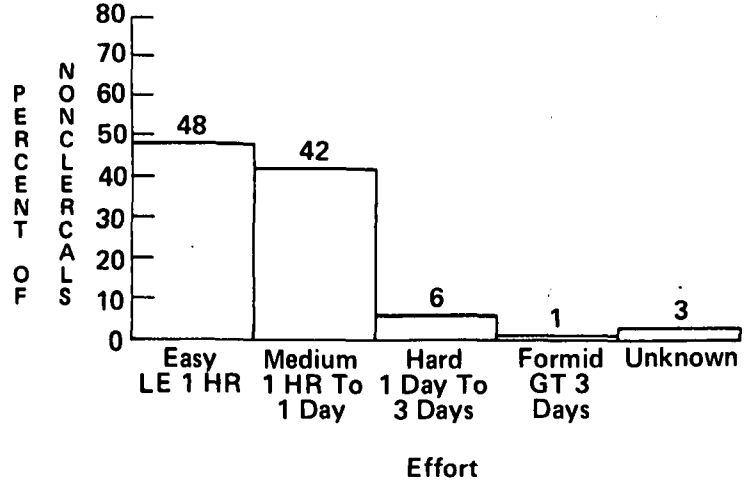
Figure 5.6 Sources of Design/Implementation Errors.



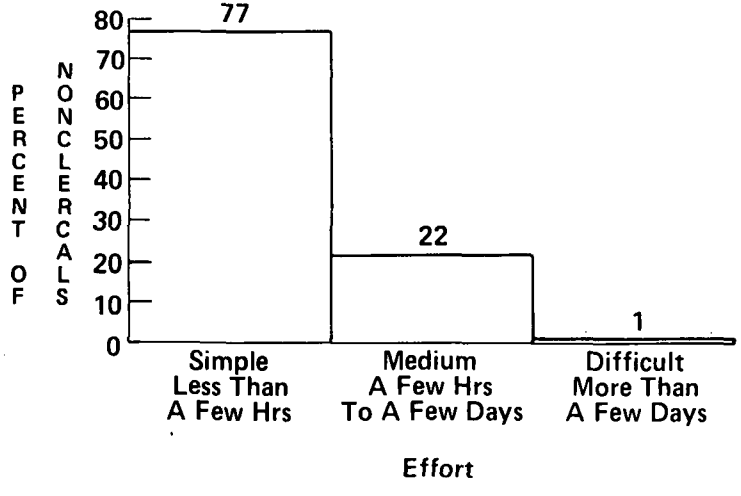
SEL1 Effort to Design Change



SEL2 Effort to Design Change

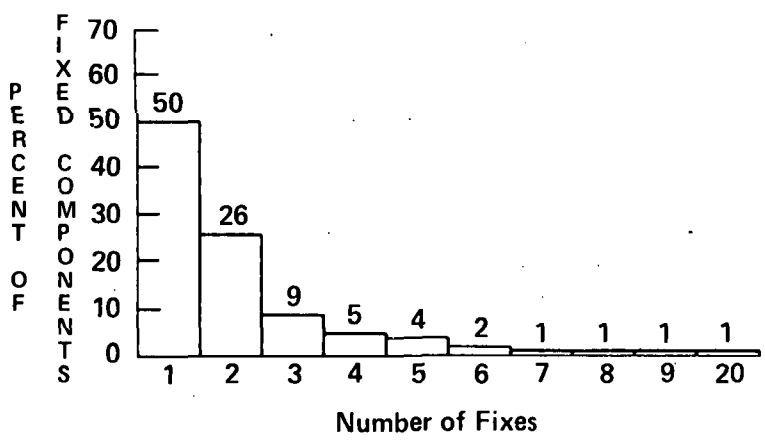


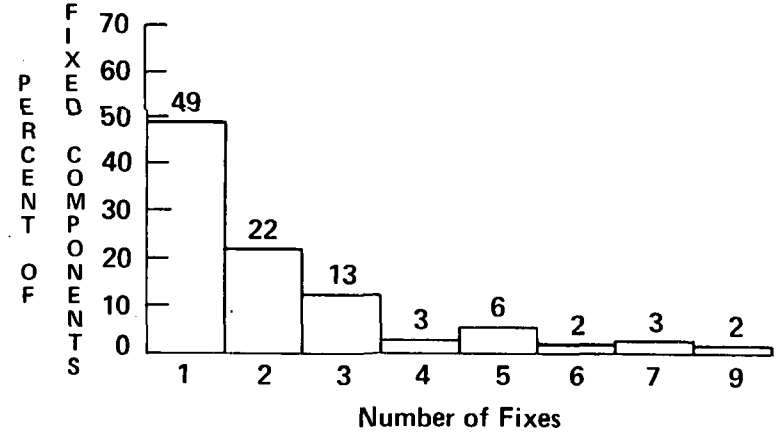
SEL3 Effort to Make Change

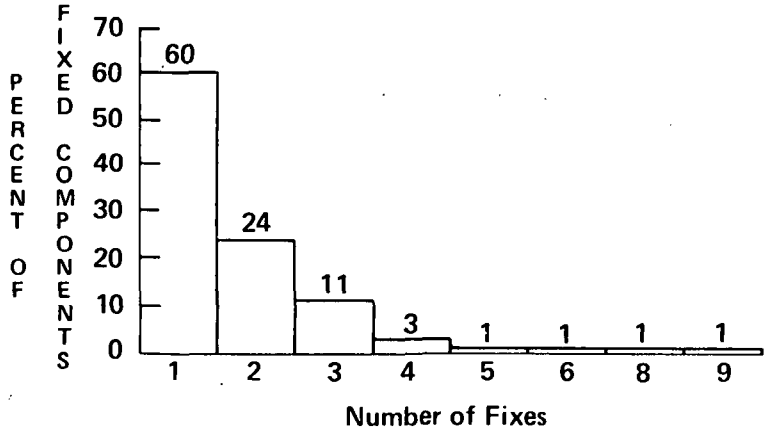


ARF Effort to Fix

Figure 5.10. Effort to Change Nonclerical Errors.







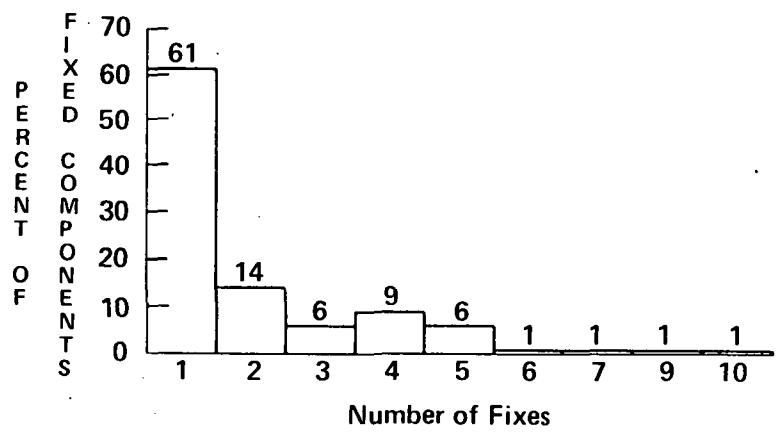


Figure 5.15. Frequency Distribution of Fixes.

CONCLUSIONS ABOUT SOFTWARE DEVELOPMENT COMMON TO NRL AND NASA/GSFC

* PRINCIPAL ERROR SOURCE IS DESIGN AND IMPLEMENTATION OF SINGLE ROUTINES

  REQUIREMENTS, SPECIFICATIONS, AND INTERFACE MISUNDERSTANDINGS ARE MINOR SOURCES OF ERRORS.

* 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.

DIFFERENCES BETWEEN ARF AND SEL SOFTWARE DEVELOPMENT

* THE PROPORTION OF ARF ERRORS INVOLVING DATA IS CONSIDERABLY SMALLER THAN THE CORRESPONDING PROPORTION FOR SEL ERRORS