SOFTWARE ENGINEERING LABORATORY SERIES SEL-85-001

A COMPARISON OF SOFTWARE VERIFICATION TECHNIQUES

(NASA-~~TM-88585)~~ A COMPARISON OF SOFTWARE VERIFICATION TECHNIQUES (NASA) 116 p HC A06/MF A01 CSCL 09B

N86-19965

Unclas
G3/61 05487

APRIL 1985

NASA
National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

SOFTWARE ENGINEERING LABORATORY SERIES SEL-85-001

A COMPARISON OF SOFTWARE VERIFICATION TECHNIQUES

APRIL 1985

NASA
National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organization members:

NASA/GSFC (Systems Development and Analysis Branch)
The University of Maryland (Computer Sciences Department)
Computer Sciences Corporation (Flight Systems Operation)

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document. A version of this document was also issued as Computer Sciences Corporation document CSC/TM-85/6017.

Contributors to this document include

David Card	(Computer Sciences Corporation)
Richard Selby	(University of Maryland)
Frank McGarry	(Goddard Space Flight Center)
Gerald Page	(Computer Sciences Corporation)
Victor Basili	(University of Maryland)
William Agresti	(Computer Sciences Corporation)

Single copies of this document can be obtained by writing to

Frank E. McGarry
Code 552
NASA/GSFC
Greenbelt, Maryland 20771

ABSTRACT

This document describes a controlled experiment performed by the Software Engineering Laboratory (SEL) to compare the effectiveness of code reading, functional testing, and structural testing as software verification techniques. It is one of a series of three experiments organized by R. W. Selby as part of his doctoral dissertation. The experiment results indicate that code reading provides the greatest error detection capability at the lowest cost, whereas structural testing is the least effective technique. This document explains the experiment plan, describes the experiment results, and discusses related results from other studies. It also considers the application of these results to the development of software in the flight dynamics environment. Appendixes summarize the experiment data and list the test programs. A separate Data Supplement contains original materials collected from the participants.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION 1 - INTRODUCTION

Probably less work, both theoretical and practical, has been invested in the development of effective tools, practices, and techniques for testing than for any other phase of the software life cycle. Many of the innovations suggested have been organizational rather than methodological, for example, independent verification and validation (Reference 1), clean-room development (Reference 2), and separate test teams (Reference 3). Decisions about the organization of testing activities are premature, however, when the relative effectiveness of different testing methodologies has not yet been established.

The principal methodological approaches to software testing and verification are code reading (Reference 4), functional testing (Reference 5), and structural testing (Reference 5). Most experts recommend a mix of these techniques (e.g., Reference 6); the relative merits of these very different approaches are not, however, well understood.

This document describes an experiment (Reference 7) performed by the Software Engineering Laboratory (SEL) to compare the effectiveness of these three techniques. It is one of a series of experiments organized by R. W. Selby as part of his doctoral dissertation (Reference 8) at the University of Maryland. The results of the experiment were examined to determine how the use of code reading, functional testing, and structural testing can best be organized in the flight dynamics environment (where they are already applied to differing degrees).

1.1 SOFTWARE VERIFICATION EXPERIMENT

The SEL planned a controlled experiment to compare the effectiveness of code reading, functional testing, and structural testing. During a 1-month period, over 40 professional programmers participated in the screening and other phases of the experiment. Subjects completed a background survey and a pretest prior to the actual experiment. The experiment consisted of three software testing sessions. This document presents an analysis and discussion of the experiment, describes the test programs, and summarizes the experiment data. The Data Supplement contains the original materials collected from the participants.

1.2 SOFTWARE ENGINEERING LABORATORY

The SEL conducted the experiment described in this document. The SEL is a research project sponsored by Goddard Space Flight Center (GSFC) and supported by Computer Sciences Corporation and the University of Maryland (Reference 9). The overall objective of the SEL is to understand the software development process at GSFC and to identify ways in which it can be modified to improve the quality and reduce the cost of the product.

The SEL collects and analyzes data from software development projects that support flight dynamics activities at GSFC. Measures collected include staffing, computer utilization, error reports, and product size/complexity statistics, as well as the technologies applied. More than 40 projects have been monitored by the SEL during the past 8 years. SEL principals also participated in the management of these projects. The data collected from these projects have been assembled in the SEL computer data base. Reference 9 describes the SEL and its activities in more detail.

Three types of experiments have been performed by the SEL: screening, semicontrolled, and controlled. Screening experiments provide detailed information about how software is currently developed in the environment under study. No attempt is made to impose new or different methodologies on these tasks. In semicontrolled experiments, on the other hand, specific methodologies are designated for application, reinforced by training and management direction. Controlled experiments can be implemented in either of two ways. Two (or more) carefully matched individuals (or teams) may be assigned the same task but required to use different methodologies. Alternatively, the teams (or individuals) may not be matched but instead be assigned consecutive tasks, applying different methodologies to each task.

Screening and semicontrolled experiments employ production flight dynamics projects; however, the high cost of full-scale development makes controlled experiments with production projects impractical. The testing experiment described in this document represents the next best alternative, a controlled experiment using production programmers.

1.3 FLIGHT DYNAMICS ENVIRONMENT

The general class of spacecraft flight dynamics software studied by the SEL includes ground-based applications to support attitude determination, attitude control, maneuver planning, orbit adjustment, and mission analysis. System sizes range from 33 to 159 thousand lines of source code, and development schedules range from 13 to 21 months. The fixed spacecraft launch date requires close adherence to schedule. Table 1-1 summarizes the characteristics of flight dynamics software.

Most flight dynamics projects are developed on a group of IBM mainframe computers using FORTRAN and assembly languages. Smaller projects are developed on a VAX super minicomputer. Some support software is developed on a PDP minicomputer. Remote terminals provide easy access to all computers. Data are collected from flight dynamics projects via questionnaires, computer accounting, automated tools, and management reviews. Reference 9 describes the flight dynamics environment in more detail.

Table 1-1. Characteristics of Flight Dynamics Environment

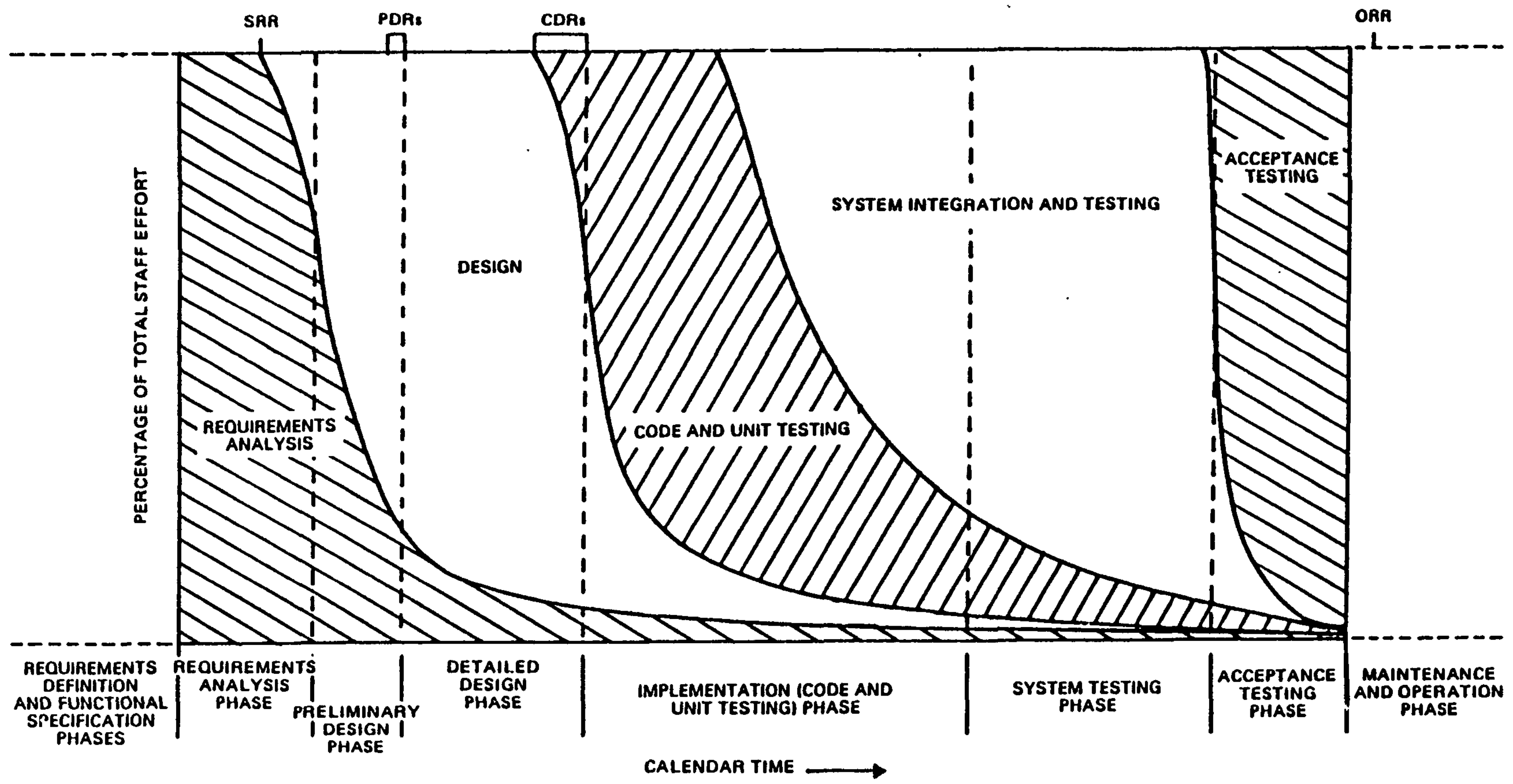
Process Characteristics	Av	High	Low
Duration (months)	16	21	13
Effort (staff years)	8	24	2
Size (1000 source lines of code)			
Developed	57	142	22
Delivered	62	159	33
Staff (full-time equivalent)			
Average	5	11	2
Peak	10	24	4
Individuals	14	29	7
Application experience (years)			
Managers	6	7	5
Technical staff	4	5	3
Overall experience (years)			
Managers	10	14	8
Technical staff	9	11	7

NOTES: Type of Software: Scientific, ground-based, interactive graphic.

Languages: 85 percent FORTRAN, 15 percent assembler macros.

Computers: IBM, PDP, VAX.

Figure 1-1 shows the distribution of effort by software life cycle activity in the flight dynamics environment. As indicated in the figure, testing consumes a substantial portion (up to 40 percent) of development resources. Any method of increasing the efficiency and effectiveness of testing activities would be welcomed. The purpose of the experiment described in this report was to compare three relevant software test/verification techniques and determine how they can best be integrated into flight dynamics software development practice.



NOTE: FOR EXAMPLE, AT THE END OF THE IMPLEMENTATION PHASE (4TH DASHED LINE), APPROXIMATELY 79% OF THE STAFF ARE INVOLVED IN SYSTEM INTEGRATION AND TESTING; APPROXIMATELY 2% ARE ADDRESSING REQUIREMENTS CHANGES OR PROBLEMS; APPROXIMATELY 2% ARE DESIGNING MODIFICATIONS; AND APPROXIMATELY 17% ARE CODING AND UNIT TESTING CHANGES.

9108-(51)-83

Figure 1-1. Activities by Percentage of Total Development Staff Effort

SECTION 2 - EXPERIMENTAL APPROACH

This section describes the three software verification techniques evaluated in the experiment as well as the experimental procedure followed and the statistical design employed in the analysis of the experiment data. It also includes some observations on the way the experiment was conducted.

2.1 VERIFICATION TECHNIQUES

Many different variations of the code reading, functional testing, and structural testing approaches are possible. For purposes of this experiment, one specific variation of each was selected. The three techniques differ with respect to the access they provide to information about the program tested (see Table 2-1). Consequently, the techniques differ with respect to who can apply them effectively. For example, users and analysts can perform functional testing because it does not require studying the source code, but they would probably not be very successful with code reading or structural testing. The following subsections define the specific versions of these techniques that the experimenters encouraged the subjects to use.

Table 2-1. Characteristics of Verification Techniques

Characteristic	Code Reading	Functional Testing	Structural Testing
View program specification	Yes	Yes	No[a]
View source code	Yes	No[b]	Yes
Execute program	No	Yes	Yes

[a]Specification was supplied after all tests were executed.

[b]Source code was supplied after all tests were executed.

2.1.1 CODE READING

Code reading is a systematic procedure for reading and understanding the operation of a program. Developers read code to determine if a program is correct with respect to a given function. Techniques of code reading include checklists, simulated execution, and step-wise abstraction (Reference 4). In practice, developers employ some aspects of all three techniques. For this experiment, subjects were trained in and encouraged to use the method of step-wise abstraction.

The method of step-wise abstraction is based on the concepts of proper subprograms, prime programs, and structured programs. Large structured programs are made up of smaller ones (subprograms). Those subprograms that cannot be further decomposed are prime programs. Identifying the prime programs in a software segment and then combining them to form higher levels of abstraction allows the actual function of the software to be defined and any errors or inconsistencies to be recognized (Reference 4). During the experiment, subjects attempted to recognize prime programs in the source code and abstract their functions.

2.1.2 FUNCTIONAL TESTING

Functional testing is a strategy for designing and selecting test cases. It treats the software like a "black box." Input is supplied, and output is observed. Comparison of the software specification with the observed input/output relationship indicates whether an error is present in the software. Functional testing does not require the tester to have any knowledge of the design or operation of the software.

For functional testing to be complete, all possible input must be tested. Clearly, this is not practical for a program of any significant size. One approach is to select at random from among the possible test cases (inputs). However, the experimenters selected the alternative strategy of equivalence partitioning for study. Equivalence partitioning also reduces the amount of input that must be tested to have reasonable confidence in the system. This is done by dividing each input condition (usually a statement in the specification) into two or more groups (Reference 5). Both valid and invalid equivalence classes must be tested. After testing was completed, subjects were provided with the source code to isolate and correct the errors found.

2.1.3 STRUCTURAL TESTING

Structural testing is another strategy for designing and selecting test cases. As opposed to functional testing, it treats the software like a "white box." Tests are specified based on an examination of the software structure (rather than the specification). Structural testing compares the detailed system design to its software implementation. Ideally, structural tests should be based on the program design language (PDL) and developed at the same time as the PDL. Structural tests alone do not provide a mechanism for verifying the software against the specification.

Coverage (the degree to which the software is exercised) serves as the basic criteria for completion of structural testing (Reference 5). Three levels of coverage are recognized: Statement coverage requires that every statement in the source code be executed at least once. Condition (or branch) coverage requires that each outcome of every decision be executed at least once. Path coverage requires that every possible sequence of decision outcomes be executed, which can lead to an impractically large number of tests for a program of any significant size or complexity. For example, backward transfers can produce an infinite number of paths.

Statement coverage is the weakest form of coverage in the sense that it can generally be satisfied with the fewest test cases. It is, however, the form of structural testing evaluated in this experiment. Subjects were directed to stop testing after 100-percent statement coverage had been achieved. After testing was completed, subjects were provided with the specification to compare against test results.

2.2 EXPERIMENT PROCEDURE

The experiment proceeded in three phases: experiment preparation, experiment execution, and data analysis. The following subsections describe these phases and present some observations on the way the experiment was conducted.

2.2.1 PREPARATION

During the month prior to the start of the experiment sessions, several preparatory steps were taken:

- Subject selection
- Preexperiment survey
- Training session
- Experiment pretest
- Environment setup

The experimenters screened about 50 professional programmers to select an appropriate sample of 32 for the planned statistical design (Section 2.2.3). Subjects were selected to represent three different levels of expertise defined by professional experience and educational background. Subjects were also about equally divided between IBM and VAX users.

All experiment candidates completed an extensive questionnaire describing their education and relevant experience. Subject selection was based on questionnaire responses as well as information supplied by managers. The Data Supplement contains the original questionnaire materials.

Subjects participated in a 3-hour tutorial explaining the three software verification techniques. Most subjects had previous experience with functional testing and informal code reading. Most, however, lacked previous formal training in any of the three techniques.

After the training session, subjects completed an experiment pretest. The subjects solved problems and answered questions defining their understanding of the three techniques and their attitude toward the experiment. The Data Supplement contains the original pretest materials.

Each subject used either an IBM 4341 or VAX 11/780 computer to perform both functional and structural testing. Before the experiment sessions began, separate versions of the three test programs were developed for each computer. Most of the differences between the two versions are minor. Appendix A provides complete source listings for the VAX version. Table 2-2 summarizes the test programs.

Table 2-2. Test Programs

Test Program	Software Type	Executable Statements	Subroutines	Faults
1	Text formatter	55	3	9
2	List manager	48	9	7
3	File maintainer	144	7	12

The experimenters established temporary IDs and special accounts on each computer. During testing, subjects invoked each program via a driver that also recorded the number of tests performed and the degree of structural coverage attained. Operating system accounting information provided the connect time and CPU utilization for each subject.

2.2.2 EXECUTION

The actual experiment was conducted in three sessions over a 2-week period. Although tight, the schedule of experimental activity was strictly adhered to. During each experiment session, all subjects tested the same program. However, all three verification techniques were applied to each program as prescribed in the statistical design (Section 2.2.3). Each subject used only one technique in any given experiment session. Tests were executed on the same computers used by the subjects in their usual work.

Each experiment session was scheduled to be completed in a single day. The experimenters felt that the experimental tasks could easily be completed within 8 hours. An experimenter distributed the test materials in the morning, then collected the results in the afternoon. However, a few subjects who did not complete testing on the prescribed day due to interruptions submitted their results on the following day. Each subject returned a list of errors, corrected source listing, and estimates of effort expended and faults found. An experimenter was available throughout the testing period to answer the subjects' questions.

Code readers received a specifications statement and source listing at the start of the experiment session. They were not allowed to execute the test program. All errors detected were found by inspecting the code and performing the step-wise abstraction process described in Section 2.1.1.

Functional testers received a specifications statement and access to the program driver at the start of the experiment session. They defined tests as described in Section 2.1.2. After executing the tests, access to the program driver was removed, but a source listing was provided. Subjects separately identified the errors detected during testing and those found during source correction.

Structural testers received a source listing and access to the program driver at the start of the experiment session. They defined and executed tests until achieving near 100-percent statement coverage (Section 2.1.3). After executing the tests, access to the program driver was removed, but a specifications statement was provided. Subjects identified errors by comparing the test output with the specifications.

2.2.3 ANALYSIS

The data analysis employed a fractional factorial design (Reference 10). The experimenters divided the subjects into three groups on the basis of overall experience. Within each group, the sequence of code reading, functional testing, and structural testing was varied to cover all possible combinations about equally. Table 2-3 shows how the subjects were divided into expertise groups and then assigned combinations of verification techniques and programs.

Because each subject used each technique once and tested each program once, rather than experiencing every possible combination of technique and program, the design is a fractional rather than full factorial model. The three programs were always presented in the same order, thereby limiting the opportunity for subjects who had completed testing a program to discuss it with others who had not yet tested it. This statistical design enabled the effects of programmer experience and program tested to be eliminated from the evaluation of the three verification techniques. It assumes, however, that there is no interaction between subject and technique or program.

The dependent variables studied were number of faults found, effort to detect faults, and effort to correct faults. Section 3 presents the results of this analysis. Although the analysis based on the fractional factorial design was

Table 2-3. Fractional Factorial Design (Reference 7)

SUBJECT CLASSIFICATION		CODE READING			FUNCTIONAL TESTING			STRUCTURAL TESTING		
		P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_2	P_3
	S_1			X		X		X		
ADVANCED	S_2		X		X					X
	⋮				•	•	•			
					•	•	•			
	S_8	X					X		X	
	S_9		X		X					X
INTERMEDIATE	S_{10}			X		X		X		
	⋮				•	•	•			
					•	•	•			
	S_{19}	X					X		X	
	S_{20}		X		X					X
JUNIOR	S_{21}	X					X		X	
	⋮				•	•	•			
					•	•	•			
	S_{32}			X		X		X		

9846-(70*)-85

NOTES: BLOCKING ACCORDING TO EXPERIENCE LEVEL AND PROGRAM TESTED.

EACH SUBJECT USES EACH TECHNIQUE AND TESTS EACH PROGRAM.

S_X = SUBJECT X.

P_X = PROGRAM X.

completed within 2 months of the last experiment session, the data collected during the experiment will enable additional analyses to be performed later.

2.2.4 OBSERVATIONS

After the completion of the experiment sessions, the subjects indicated their reactions to the experiment process and verification techniques evaluated. Relevant comments included the following:

- A majority of subjects believed functional testing to have been the most effective software verification strategy.
- A majority of subjects did not follow the step-wise abstraction method of code reading exactly or exclusively.
- Several subjects indicated that satisfaction of the 100-percent statement coverage criteria caused them to stop structural testing, even though they felt that more testing was needed.
- Several subjects disagreed with the experimenter's definitions of errors. These conflicts arose from differing interpretations of the severity of problems as well as specification ambiguity.
- Some subjects were not able to concentrate fully on the experiment due to conflicting responsibilities (not unlike conditions obtaining during normal software development).
- An office relocation that occurred on the same day as the last experiment session interrupted the activities of almost all the subjects.

Nevertheless, none of the subjects indicated that the overall test results seemed biased or that the experiment procedure was seriously flawed.

SECTION 3 - EXPERIMENT RESULTS

Thirty-two subjects participated in the three experiment sessions. Appendix B documents their performance with each verification technique. The data were analyzed with a fractional factorial design (described in Section 2.2.3). The principal areas for comparison among the techniques were as follows:

- Effectiveness of fault detection in terms of the number of faults detected
- Cost of fault detection/correction in terms of the effort (in hours) per fault to detect and correct the faults identified
- Types of errors to which each technique was sensitive
- Role of subject experience in technique effectiveness

The following subsections describe the results obtained from the experiment in each of these areas.

3.1 EFFECTIVENESS OF FAULT DETECTION

Table 3-1 lists the average number of errors detected by subjects using each technique for each program. Figure 3-1 summarizes the faults detected by the techniques across programs. The figure shows that code reading detected the most faults overall. Code reading detected significantly more faults than functional testing, and functional testing detected significantly more faults than structural testing.

Each program contained a different number of faults. Nevertheless, analyzing the data in terms of the percentage of total faults detected yields the same result (Figure 3-2). Code reading is clearly superior to either online testing technique with respect to fault-detection effectiveness. However, the variation due to verification technique appears to be much less than that due to the specific program under examination.

Table 3-1. Fault Detection Effectiveness

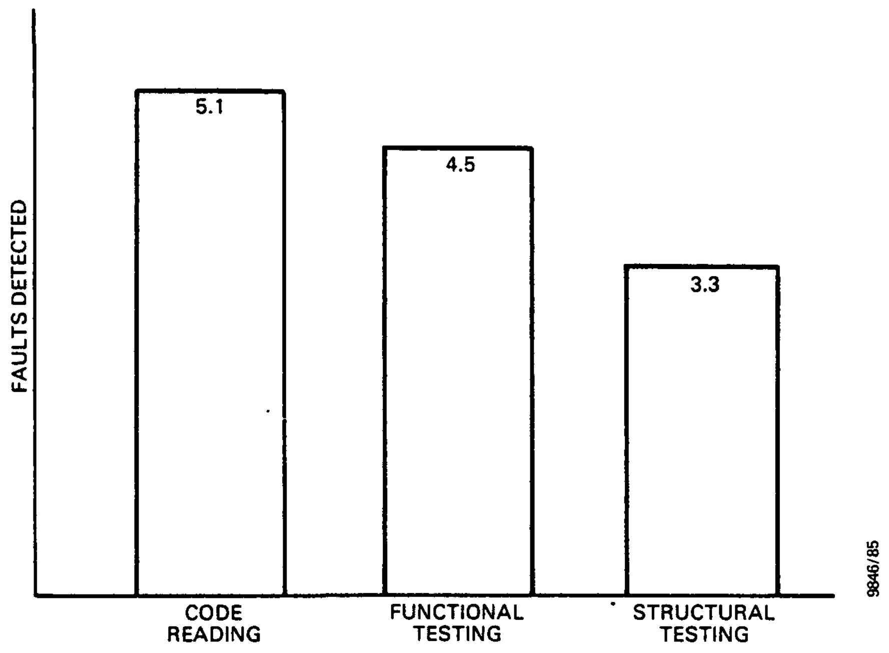
Verification Technique	Program 1	2	3	Overall
Code reading	5.5	6.6	3.2	5.1
Functional testing	4.6	4.6	4.2	4.5
Structural testing	2.5	4.4	2.8	3.3

NOTE: Values are average number of faults detected.

The difference in fault-detection effectiveness between functional and structural testing stands out because these techniques are similar in many ways. Both structural and functional testing achieved about the same level of statement coverage (97 percent), although it was not a criterion for functional testing. Furthermore, both structural and functional testing exposed about the same percentage of faults (62 percent), but not all faults exposed by testing were recognized as such by the subjects. Functional testers recognized 19 percent more of the observable faults than did structural testers.

3.2 COST EFFECTIVENESS OF FAULT DETECTION/CORRECTION

Table 3-2 lists the average number of faults detected and corrected per hour of effort using each technique for each program. Figure 3-3 summarizes the fault detection/correction rate for each technique across programs. The figure shows that code reading detected the most faults per hour of effort. Code reading performed significantly better than functional or structural testing, which were not

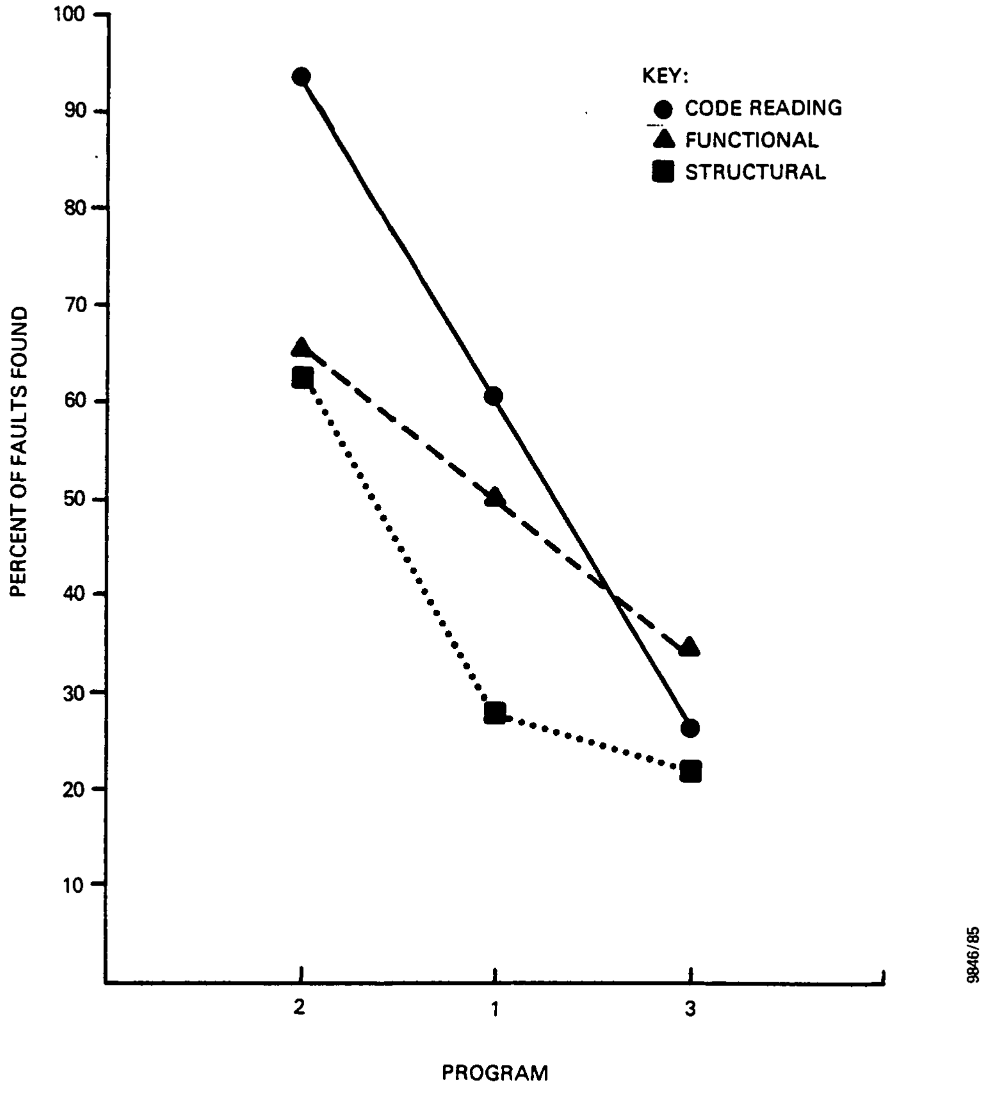


NOTES: CODE READING > OTHERS.

FUNCTIONAL >STRUCTURAL.

PROBABILITY OF DIFFERENCE BEING RANDOM IS LESS THAN 0.005.

Figure 3-1. Number of Faults Detected (Reference 7)



NOTE: PROGRAMS ORDERED ACCORDING TO SIZE.

Figure 3-2. Percentage of Faults Detected

significantly different from each other. No significant differences, however, exist among the techniques with respect to the total time spent looking for and correcting faults (see Figure 3-4).

Table 3-2. Fault Detection/Correction Rate

Verification Technique	Program 1	Program 2	Program 3	Overall
Code reading	1.1	3.6	0.9	1.9
Functional testing	1.1	1.2	0.7	1.0
Structural testing	1.1	1.7	0.5	1.1

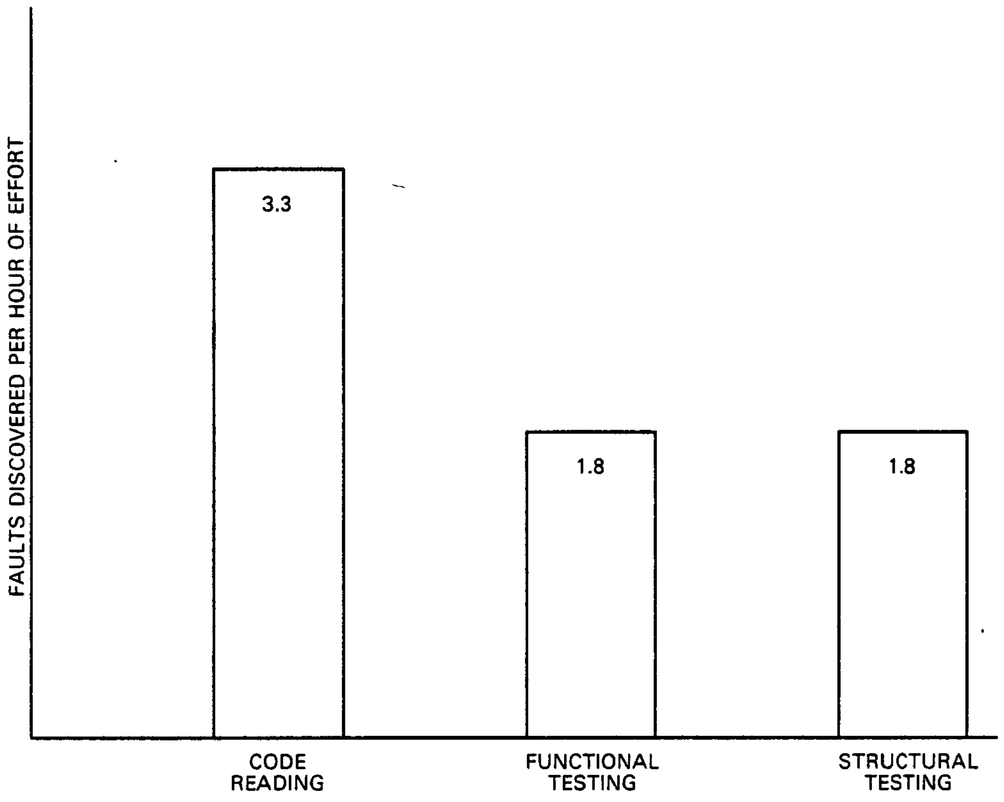
NOTE: Values are average number of faults detected and corrected per hour of effort.

The experimenters also monitored computer utilization during functional and structural testing. (Code reading requires no computer resources.) Functional testing expended significantly more CPU time than structural testing; however, no differences were detected in the number of test runs.

3.3 CHARACTERISTICS OF FAULTS

Table 3-3 identifies the classes of faults present in the test programs according to two criteria. The action associated with a fault identifies whether it is due to omitting some necessary code or incorrectly implementing code. The location of a fault defines where it occurs in the code.

Table 3-4 indicates that code reading and functional testing proved to be substantially more effective than structural testing in detecting faults of omission. Seven such faults were detected by less than 25 percent of the structural testers, whereas the same proportion of code readers and functional testers left only five and four omission faults,

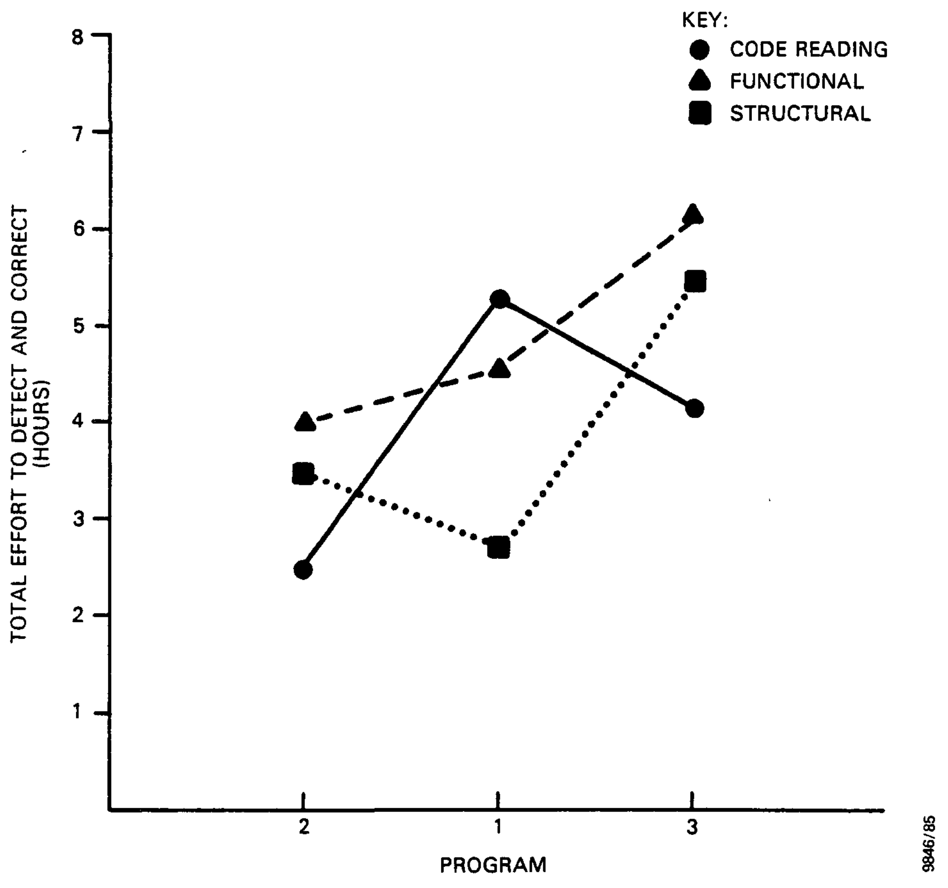


NOTES: CODE READING > OTHERS.

PROBABILITY THAT THIS DIFFERENCE IS RANDOM IS 0.005.

FUNCTIONAL ≅ STRUCTURAL.

Figure 3-3. Cost-Effectiveness (Number of Faults Detected Per Hour of Effort) (Reference 7)



NOTE: PROGRAMS ORDERED ACCORDING TO SIZE.

Figure 3-4. Total Detection/Correction Effort

respectively, undetected. This result seems reasonable given that structural tests were based solely on the existing code.

Table 3-3. Fault Characterization (Reference 7)

Location	Action	
	Omission	Commission
Initialization	0	2
Computation	2	2
Control	2	4
Interface	2	11
Data	2	0
Cosmetic	0	1
TOTAL	8	20

NOTE: Values are numbers of faults.

Table 3-4 also shows structural testing to be less effective in detecting faults of commission. Only three such faults were detected by 75 percent or more of the structural testers, whereas the same proportion of code readers and functional testers detected nine and six commission faults, respectively.

The only fault location sufficiently represented in the sample to provide a basis for conclusions was interface fault. As shown in Table 3-5, two or three such faults were found by 75 percent or more of online testers, whereas seven interface faults were detected by 25 percent or more of code readers. Code reading thus appears to be the superior technique with respect to interface errors.

Table 3-4. Detection of Omission/Commission Faults (Reference 7)

Percent of Subjects Detecting This Fault	Code Reading	Functional Testing	Structural Testing
100	CCCO	CCO	
	CC	C	
	C	CC	C
	CCCO		CO
75		CO	C
	C		
	CCCC	CCCCO	CCCC
50		CC	C
	O		C
			CC
	C	CO	CCC
	C	CC	
	CC		
25		C	CC
	CO	C	CO
	O		O
		COO	OO
0	COOO	CCOO	CCCOOO

NOTES: C = fault of commission
O = fault of omission

Table 3-5. Detection of Interface Faults (Reference 7)

Percent of Subjects Detecting This Fault	Code Reading	Functional Testing	Structural Testing
100	III	I	
	II		
		II	
	II		I
75			I
			II
		I	
50			
			I
	I	II	II
		II	
	I		
25		I	I
	I		I
	I		
		II	I
0	II	II	III

NOTES: I = interface fault

3.4 EFFECT OF SUBJECT EXPERTISE

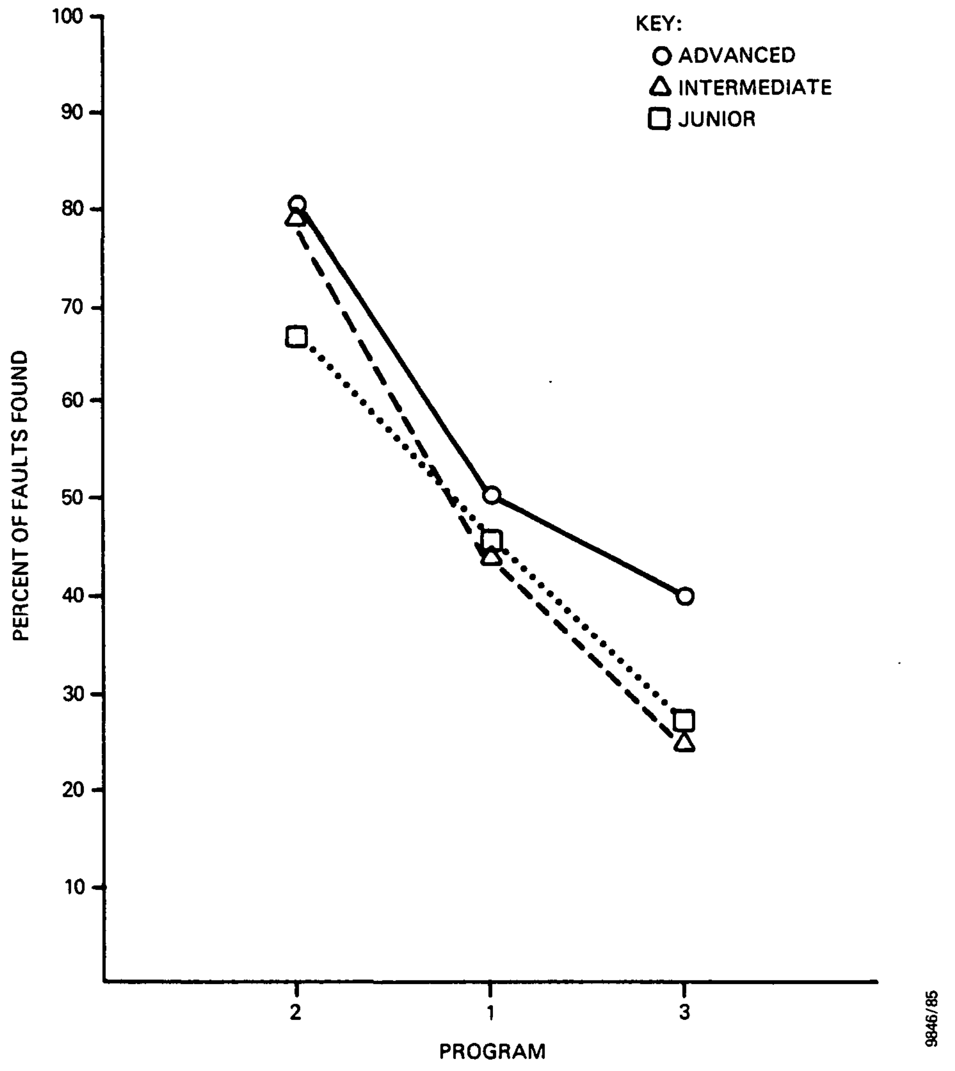
The experimenters assigned subjects to three expertise levels on the basis of their professional experience and educational background. Table 3-6 shows subject performance by expertise level. Advanced subjects detected significantly more faults than intermediate or junior subjects. However, the detection rate (faults detected per hour of effort) did not appear to be related to level of expertise.

Figures 3-5 and 3-6 show the interaction of expertise level with program tested and verification technique, respectively. The specific program tested appears to have a greater effect on the percent of faults found than expertise level does. Also, the advanced subjects performed substantially better than intermediate or junior subjects when code reading or structural testing. The percent of faults found when functional testing was similar for all expertise levels.

Table 3-6. Effect of Subject Expertise

Expertise Level	Number of Subjects	Faults Detected	Detection Rate[a]
Advanced	8	5.0	2.36
Intermediate	11	4.2	2.53
Junior	13	3.9	2.14

[a]Faults detected per hour of effort.



NOTE: PROGRAMS ORDERED ACCORDING TO SIZE.

Figure 3-5. Interaction of Expertise Level and Program Tested

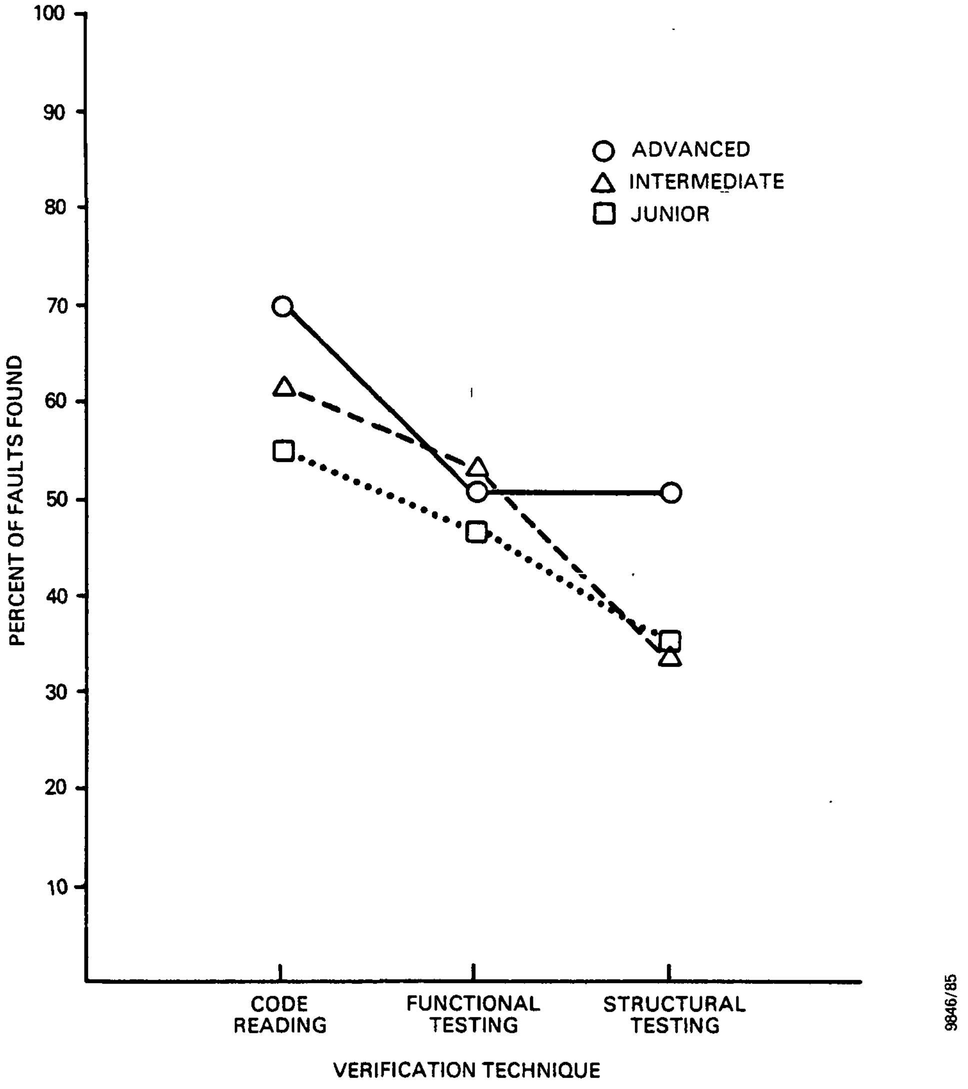


Figure 3-6. Interaction of Expertise Level and Verification Technique

SECTION 4 - RELATED RESULTS

This section reviews the results of the experiment described in Sections 2 and 3 in light of related research performed by the SEL and others. Specifically, an attempt is made to identify the implications of this experiment for developer training and development organization.

4.1 OTHER VERIFICATION TECHNIQUE EVALUATIONS

The results of this experiment imply that code reading detects the most faults at the lowest cost. Several relevant studies have been conducted at the University of Maryland. A prototype study of the same three verification techniques by Hwang (Reference 11) suggested that code reading does at least as well as the computer-based techniques. Two earlier experiments by Selby (Reference 12) using student subjects found that functional testing detected the most faults. In one of these experiments, code reading demonstrated the lowest fault detection cost; functional testing led in the other.

Other researchers, as well, have studied these techniques. Card et al. (Reference 13) showed that code reading increases the reliability of the delivered software product. It was not, however, compared with the other techniques. Myers (Reference 14) evaluated functional testing and three-person code reviews versus a control group. All three groups proved to be equally effective in detecting errors, but code reviews cost substantially more. Hetzel (Reference 15) compared functional testing, code reading, and a composite testing technique and found code reading to be inferior to the other techniques.

A significant difference between the experiment reported in this document and many other studies is the use of professional programmers as subjects. Viewed in this context, the effectiveness of code reading relative to the other techniques appears to increase with experience. Effective code reading may require not only training in the technique of step-wise abstraction but also the recognition of common programming paradigms. Experiments by Soloway and Ehrlich (Reference 16) showed that experienced programmers surpassed novices in detecting and correcting faults when certain common programming plans and rules of discourse were followed in the implementation of the code. The performance of experienced programmers approximated that of novices when the plans and rules were not followed.

Together, these results suggest that code reading is most effective when performed by individual experienced programmers. On the other hand, computer-based testing can be performed effectively by less-experienced programmers. Furthermore, functional testing works well for an independent test team (who do not have the developers' knowledge of the code).

4.2 INDEPENDENT VERIFICATION AND VALIDATION

Independent verification and validation (IV&V) is an approach to organizing software development in which an independent team is assigned to verify and validate each life-cycle product of the development team (Reference 17). Specifically, the IV&V team performs independent system testing in addition to that done by the development team. Many of the errors reported by an IV&V team duplicate those found by the development team.

The IV&V team does not usually read code. Given that online testing is less efficient than code reading, it is not surprising that a recent study showed that the principal effect of IV&V was to increase development costs in the flight dynamics environment (Reference 18), where code reading is already practiced. IV&V may provide some benefit in the earlier phases of requirements and design, but that is more difficult to demonstrate.

4.3 CLEAN-ROOM DEVELOPMENT

Dyer and Mills (Reference 2) proposed that software development can and should be done without the aid of computers. Instead of computer-based testing, the development team should rely on code reading, code inspections, and formal verification techniques to identify and correct faults. After development (of a build or a system) was complete, an independent testing team would certify the reliability of the developed software.

A recent experiment by Selby (Reference 8) showed that, relative to a control group, development teams using the clean-room approach delivered products that passed a higher percentage of test cases, more completely satisfied the system requirements, included more comments, and exhibited lower complexity. Another SEL study (Reference 13) indicated that a high rate of computer use by the development team is associated with low productivity.

The results of the testing experiment reported in this document demonstrate that one clean-room technique, code reading, surpasses the effectiveness of online testing techniques. This evidence, together with the results of the clean-room experiment (Reference 8), indicates that the clean-room approach to software development is well founded and viable. However, it does not prove its value in every circumstance, or specifically in the flight dynamics environment. Further study of this approach is needed.

SECTION 5 - CONCLUSIONS

As described in Section 2, the controlled experiment compared the performance of code reading, functional testing, and structural testing on three software verification tasks. These techniques were applied by professional programmers working in their usual production environment. The results of the controlled experiment (Section 3), together with the related results presented in Section 4, provide considerable guidance about the overall effectiveness of the software verification techniques studied and about how to best employ them in the flight dynamics environment.

5.1 EVALUATION OF VERIFICATION TECHNIQUES

Each of the verification techniques showed some merit in the controlled experiment. Code reading performed best overall. Functional testing was second. The relatively poor showing of structural testing may be due, in part, to the weak coverage criterion used (i.e., statement coverage).

Prior to the experiment, only 22 percent of respondents to the background survey believed that code reading was the most effective software verification technique. About equal proportions (38 percent and 40 percent, respectively) favored structural or functional testing. After participating in the experiment, the majority of subjects felt that functional testing had proved to be the most effective technique.

Nevertheless, subjects applying code reading detected the most faults and expended the least effort per fault to do so. Functional testing actually proved to be the second most effective technique in terms of the number of faults detected. Functional testers were more likely to recognize an error that was uncovered by testing than were structural testers.

Both code reading and functional testing detected faults of commission and omission better than did structural testing. Code reading showed itself to be the most effective technique with respect to interface errors. Furthermore, the advanced expertise group succeeded substantially better than intermediate or junior subjects at code reading effectively. The results of the experiment also indicated that the effectiveness of these verification techniques may depend on program size. This factor still needs to be investigated.

5.2 APPLICATION IN FLIGHT DYNAMICS ENVIRONMENT

The results of the controlled experiment and related studies suggest several modifications to the process of software development in the flight dynamics environment. These include the following:

- Formalize the code reading process.
- Provide specific instruction in code reading.
- Assign senior developers to code reading.
- Assign junior developers to functional testing, possibly as an independent test team.
- Do not apply structural testing. However, monitoring the coverage achieved may be useful for functional testing.

An earlier study of flight dynamics acceptance testing practices (Reference 19) supports many of these conclusions. That study was based on a review of the relevant literature and observation of then-current procedures. Its recommendations included the following: acceptance test each build, trace acceptance tests to requirements, measure test coverage, use equivalence partitioning (to reduce test cases), perform independent acceptance testing, and record error histories. Many of these recommendations have already been implemented.

Together, these studies demonstrate that the specific techniques applied, as well as the expertise level of the testers, affect the overall rate and cost of fault detection/correction. Selecting and combining the appropriate techniques, personnel, and organization can make a significant contribution to software quality.

APPENDIX A - TEST PROGRAMS

This appendix reproduces the specifications, source listings, and error identifications for the test programs used in the experiment. The test programs represent three different types of software:

- Text formatter
- List manager
- File maintainer

Congruent versions of these programs were developed for both the VAX-11/780 and IBM 4341 computers. The VAX versions are supplied in this appendix. The Data Supplement contains both versions of the specifications and source listings.

1) VAX: Code Reading ID# ______

Specification

Given an input text of up to 80 characters consisting of words separated by blanks or new-line characters, the program formats it into a line-by-line form such that 1) each output line has a maximum of 30 characters, 2) a word in the input text is placed on a single output line, and 3) each output line is filled with as many words as possible.

The input text is a stream of characters, where the characters are categorized as either break or nonbreak characters. A break character is a blank, a new-line character ($), or an end-of-text character (/). New-line characters have no special significance; they are treated as blanks by the program. The characters $ and / should not appear in the output.

A word is defined as a nonempty sequence of nonbreak characters. A break is a sequence of one or more break characters and is reduced to a single blank character or start of a new line in the output.

When the program is invoked, the user types the input on a single line, followed by by a / (end-of-text) and a carriage return. The program then formats the text and types it on the terminal.

If the input text contains a word that is too long to fit on a single output line, an error message is typed and the program terminates. If the end-of-text character is missing, an error message is issued and the program awaits the input of a properly terminated line of text. (End of specification.)

1) **VAX: Reading Functional Structural** ID# ______

```
C NOTE THAT YOU DO NOT NEED TO VERIFY THE FUNCTION 'MATCH'.
C   IT IS DESCRIBED THE FIRST TIME IT IS USED, AND ITS SOURCE CODE
C   IS INCLUDED AT THE END FOR COMPLETENESS.
C
C NOTE THAT FORMAT STATEMENTS FOR WRITE STATEMENTS INCLUDE A LEADING
C   AND REQUIRED ' ' FOR CARRIAGE CONTROL

C VARIABLE USED IN FIRST, BUT NEEDS TO BE INITIALIZED
      INTEGER MOREIN

C STORAGE USED BY GCHAR
      INTEGER BCOUNT
      CHARACTER*1 GBUFER(80)
      CHARACTER*80 GBUF
C GBUFER AND GBUFSTR ARE EQUIVALENCED

C STORAGE USED BY PCHAR
      INTEGER I
      CHARACTER*1 OUTLIN(31)
C OUTLIN AND OUTLINST ARE EQUIVALENCED

      CHARACTER*1 GCHAR

C CONSTANT USED THROUGHOUT THE PROGRAM
      CHARACTER*1 EOTEXT, BLANK, LINEFD
      INTEGER MAXPOS

      COMMON /ALL/ MOREIN, BCOUNT, I, MAXPOS, OUTLIN,
   X    EOTEXT, BLANK, LINEFD, GBUFER, GBUF

      DATA EOTEXT, BLANK, LINEFD, MAXPOS / '/', ' ', '&', 31 /


      CALL FIRST
      END


      SUBROUTINE FIRST
      INTEGER K, FILL, BUFPOS
      CHARACTER*1 CW
      CHARACTER*1 BUFFER(31)

      INTEGER MOREIN, BCOUNT, I, MAXPOS
      CHARACTER*1 OUTLIN(31), GCHAR, EOTEXT, BLANK, LINEFD,
   X          GBUFER(80)
      CHARACTER*80 GBUF

      COMMON /ALL/ MOREIN, BCOUNT, I, MAXPOS, OUTLIN,
   X    EOTEXT, BLANK, LINEFD, GBUFER, GBUF

      BUFPOS = 0
      FILL = 0
      CW = ' '
```

1) VAX: Reading Functional Structural ID# ______

```

        MOREIN = 1

        I = 1
        K = 1
        DOWHILE (K .LE. MAXPOS)
                OUTLIN(K) = ' '
                K = K + 1
        ENDDO

        BCOUNT = 1
        K = 1
        DOWHILE (K .LE. 80)
                GBUFER(K) = 'Z'
                K = K + 1
        ENDDO

        DOWHILE (MOREIN)
                CW = GCHAR()
                IF ((CW .EQ. BLANK) .OR. (CW .EQ. LINEFD) .OR.
     X                          (CW .EQ. EOTEXT)) THEN
                        IF (CW .EQ. EOTEXT)  THEN
                                MOREIN = 0
                        ENDIF
                        IF ((FILL+1+BUFPOS) .LE. MAXPOS)  THEN
                                CALL PCHAR(BLANK)
                                FILL = FILL + 1
                            ELSE
                                CALL PCHAR(LINEFD)
                                FILL =0
                        ENDIF
                        K = 1
                        DOWHILE (K .LE. BUFPOS)
                                CALL PCHAR(BUFFER(K))
                                K = K + 1
                        ENDDO
                        FILL = FILL + BUFPOS
                        BUFPOS = 0
                    ELSE
                        IF (BUFPOS .EQ. MAXPOS)  THEN
                                WRITE(6,10)
10                              FORMAT(' ','***WORD TO LONG***')
                                MOREIN = 1
                            ELSE
                                BUFPOS = BUFPOS + 1
                                BUFFER(BUFPOS) = CW
                        ENDIF
                ENDIF
        ENDDO
        CALL PCHAR(LINEFD)
        END


```

1) **VAX: Reading Functional Structural** **ID# ______**

```
        CHARACTER*1 FUNCTION GCHAR()
        INTEGER MATCH
        CHARACTER*80 GBUFSTR

        INTEGER MOREIN, BCOUNT, I, MAXPOS
        CHARACTER*1 OUTLIN(31), EOTEXT, BLANK, LINEFD,
     X          GBUFER(80)
        CHARACTER*80 GBUF
        COMMON /ALL/ MOREIN, BCOUNT, I, MAXPOS, OUTLIN,
     X    EOTEXT, BLANK, LINEFD, GBUFER, GBUF

        EQUIVALENCE (GBUFSTR,GBUFER)

        IF (GBUFER(1) .EQ. 'Z')  THEN
                READ(5,20) GBUF
20              FORMAT(A80)
C
C MATCH(CARRAY,C,ARSIZE) RETURNS 1 IF CHARACTER C IS IN CHARACTER ARRAY
C   CARRAY, RETURNS 0 OTHERWISE. ARSIZE IS THE SIZE OF CARRAY.
C
                IF (MATCH(GBUF,EOTEXT) .EQ. 0)  THEN
                        WRITE(6,30)
30                      FORMAT(' ','***NO END OF TEXT MARK***')
                        GBUFER(2) = EOTEXT
                    ELSE
C                       GBUFER(1) = GBUF
                        GBUFSTR = GBUF
                ENDIF
        ENDIF
        GCHAR = GBUFER(BCOUNT)
        BCOUNT = BCOUNT + 1
        END


        SUBROUTINE PCHAR (C)
        CHARACTER*1 C
        CHARACTER*31 SOUT, OUTLINST
        INTEGER K

        INTEGER MOREIN, BCOUNT, I, MAXPOS
        CHARACTER*1 OUTLIN(31), GCHAR, EOTEXT, BLANK, LINEFD,
     X          GBUFER(80)
        CHARACTER*80 GBUF
        COMMON /ALL/ MOREIN, BCOUNT, I, MAXPOS, OUTLIN,
     X    EOTEXT, BLANK, LINEFD, GBUFER, GBUF

        EQUIVALENCE (OUTLINST,OUTLIN)

        IF (C .EQ. LINEFD)  THEN
                SOUT = OUTLINST
                WRITE(6,40) SOUT
40              FORMAT(' ',A31)
                K = 1
                DOWHILE (K .LE. MAXPOS)
                        OUTLIN(K) = ' '
                        K = K + 1
                ENDDO
                I = 1
            ELSE
                OUTLIN(I) = C
                I = I + 1
        ENDIF
        END
C
C NOTE: YOU DO NOT NEED TO VERIFY THE FOLLOWING FUNCTION.  ITS SOURCE
C   CODE IS INCLUDED JUST FOR COMPLETENESS.
C
C MATCH(CARRAY,C,ARSIZE) RETURNS 1 IF CHARACTER C IS IN CHARACTER ARRAY
C   CARRAY, RETURNS 0 OTHERWISE. ARSIZE IS THE SIZE OF CARRAY.
C
        INTEGER FUNCTION MATCH (STRIN, CH)
        CHARACTER*100 STRIN
        CHARACTER*1 CH

        INTEGER PTR
        CHARACTER*100 STR
        CHARACTER*1 SARRAY(100)
        EQUIVALENCE (STR,SARRAY)

        STR = STRIN
        DO 100 PTR = 1,100
                IF (SARRAY(PTR) .EQ. CH)  THEN
                        MATCH = 1
                        RETURN
                ENDIF
100     CONTINUE
        MATCH = 0
        RETURN
        END
```

PROGRAM 1 FAULTS

1. Blank is printed before the first word on the first line unless the first word is 30[a] characters long.

1.[b] Leading blank line printed when first word is 30[a] characters long.

2. Assumes & (not $) is new-line character.

3. Assumes line size is 31 (not 30) characters.

4. First character of input line equal to 'z' results in line being ignored.

5. Does not condense successive break characters.

6. Spelling mistake in 'word to long' (WTL).

7. After WTL message, formatting does not terminate.

7.[b] After WTL condition, message printed once for every character of word in excess of 30[a].

8. After WTL condition, program prints output buffer.

9. If a line is entered without the end-of-text (EOT) character, a message is printed, but if the next line has EOT, this character is changed to 'z' in output.

9.[b] After two successive omissions of EOT, the program prints 'z' and terminates.

---

[a]Actually 31, due to Error 3.

[b]Alternate manifestation of this error.

2) VAX: Functional Testing

ID# ______

Specification

A list is defined to be an ordered collection of elements which may have elements annexed and deleted at either end, but not in the middle. The operations that need to be available are ADDFIRST, ADDLAST, DELETEFIRST, DELETELAST, FIRST, ISEMPTY, LISTLENGTH, REVERSE and NEWLIST. Each operation is described in detail below.

The lists are to contain up to a maximum of five (5) elements. If an element is added to the front of a "full" list (one containing five elements already), the element at the back of the list is to be discarded. Elements to be added to the back of a full list are discarded. Requests to delete elements from empty lists result in an empty list, and requests for the first element of an empty list results in zero (0) being returned. The detailed operational descriptions are as below.

ADDFIRST I
: Returns the list with the integer I as its first element followed by all the elements of the list. If the list is "full" to begin with, its last element is lost.

ADDLAST I
: Returns the list with all of the elements of its elements followed by the integer I. If the list is full to begin with, the list is returned (i.e., I is ignored).

DELETEFIRST
: Returns the list containing all but its first element. If the list is empty, then an empty list is returned.

DELETELAST
: Returns the list containing all but its last element. If the list is empty, then an empty list is returned.

FIRST
: Returns the first element in the list. If the list is empty, then it returns zero (0).

ISEMPTY
: Returns one (1) if the list is empty, zero (0) otherwise.

LISTLENGTH
: Returns the number of elements in the list. An empty list has zero (0) elements.

NEWLIST
: Returns an empty list.

REVERSE
: Returns the list containing its original elements in reverse order.

(End of specification.)

2) VAX: Reading Functional Structural ID# ______

```
C NOTE THAT YOU DO NOT NEED TO VERIFY THE FUNCTIONS DRIVER, GETARG,
C    CHAREQ, CODE, AND PRINT. THEIR SOURCE CODE IS DESCRIBED AND
C    INCLUDED AT THE END FOR COMPLETENESS.
C NOTE THAT FORMAT STATEMENTS FOR WRITE STATEMENTS INCLUDE A LEADING
C    AND REQUIRED ' ' FOR CARRIAGE CONTROL
C
        INTEGER POOL(7), LSTEND
        INTEGER LISTSZ
C
        COMMON /ALL/ LISTSZ
C
C
        LISTSZ = 5
        CALL DRIVER (POOL, LSTEND)
        STOP
        END
C
C
        FUNCTION ADFRST (POOL, LSTEND, I)
        INTEGER ADFRST
        INTEGER POOL(7), LSTEND, I
        INTEGER LISTSZ
        COMMON /ALL/ LISTSZ
C
        INTEGER A
C
        IF (LSTEND .GT. LISTSZ) THEN
                LSTEND = LISTSZ - 1
        ENDIF
        LSTEND = LSTEND + 1
        A = LSTEND
        DOWHILE (A .GE. 1)
                POOL(A+1) = POOL(A)
                A = A - 1
        ENDDO
C
        POOL(1) = I
        ADFRST = LSTEND
        RETURN
        END
C
C
        FUNCTION ADLAST (POOL, LSTEND, I)
        INTEGER ADLAST
        INTEGER POOL(7), LSTEND, I
        INTEGER LISTSZ
        COMMON /ALL/ LISTSZ
C
        IF (LSTEND .LE. LISTSZ) THEN
                LSTEND = LSTEND + 1
                POOL(LSTEND) = I
        ENDIF
        ADLAST = LSTEND
```

2) **VAX: Reading Functional Structural** **ID#** ______

```
        RETURN
        END
C
C
        FUNCTION DELFST (POOL, LSTEND)
        INTEGER DELFST
        INTEGER POOL(7), LSTEND
        INTEGER LISTSZ
        COMMON /ALL/ LISTSZ
C
        INTEGER A
        IF (LSTEND .GT. 1) THEN
                A = 1
                LSTEND = LSTEND - 1
                DOWHILE (A .LE. LSTEND)
                        POOL(A) = POOL(A+1)
                        A = A + 1
                ENDDO
        ENDIF
        DELFST = LSTEND
        RETURN
        END
C
C
        FUNCTION DELLST (LSTEND)
        INTEGER DELLST
        INTEGER LSTEND
C
        IF (LSTEND .GE. 1) THEN
                LSTEND = LSTEND - 1
        ENDIF
        DELLST = LSTEND
        RETURN
        END
C
C
        FUNCTION FIRST (POOL, LSTEND)
        INTEGER FIRST
        INTEGER POOL(7), LSTEND
C
        IF (LSTEND .LE. 1) THEN
                FIRST = 0
            ELSE
                FIRST = POOL(1)
        ENDIF
        RETURN
        END
C
C
        FUNCTION EMPTY (LSTEND)
        INTEGER EMPTY
        INTEGER LSTEND
C
```

```
        IF (LSTEND .LE. 1) THEN
                EMPTY = 1
            ELSE
                EMPTY = 0
        ENDIF
        RETURN
        END
C
C
        FUNCTION LSTLEN  (LSTEND)
        INTEGER LSTLEN
        INTEGER LSTEND
C
        LSTLEN = LSTEND - 1
        RETURN
        END
C
C
        FUNCTION NEWLST (LSTEND)
        INTEGER NEWLST
        INTEGER LSTEND
C
        NEWLST = 0
        RETURN
        END
C
C
        SUBROUTINE REVERS (POOL, LSTEND)
        INTEGER POOL(7), LSTEND
C
        INTEGER I, N
C
        N = LSTEND
        I = 1
        DOWHILE (I .LE. N)
                POOL(I) = POOL(N)
                N = N - 1
                I = I + 1
        ENDDO
        RETURN
        END
C
C
C NOTE: YOU DO NOT NEED TO VERIFY THE FOLLOWING PROCEDURES.  THEIR SOURCE
C   CODE IS INCLUDED JUST FOR COMPLETENESS.
C
C DRIVER ACCEPTS THE COMMANDS, CALLS THE APPROPRIATE ROUTINES, AND DISPLAYS
C   THE RESULTS.
C
        SUBROUTINE DRIVER (POOL, LSTEND)
        INTEGER POOL(7), LSTEND
C
        INTEGER ADFRST,ADLAST,DELFST,DELLST,FIRST,EMPTY,LSTLEN,NEWLST
        INTEGER ARG, GETARG
        LOGICAL*1 CODE
        LOGICAL*1 CMD(80)
        LOGICAL*1 CMD1(8),CMD2(7),CMD3(11),CMD4(10),CMD5(7),CMD6(5),
     X    CMD7(10),CMD8(7),CMD9(7)
        DATA CMD1 /'A','D','D','F','I','R','S','T'/
        DATA CMD2 /'A','D','D','L','A','S','T'/
        DATA CMD3 /'D','E','L','E','T','E','F','I','R','S','T'/
        DATA CMD4 /'D','E','L','E','T','E','L','A','S','T'/
        DATA CMD5 /'I','S','E','M','P','T','Y'/
        DATA CMD6 /'F','I','R','S','T'/
        DATA CMD7 /'L','I','S','T','L','E','N','G','T','H'/
        DATA CMD8 /'R','E','V','E','R','S','E'/
        DATA CMD9 /'N','E','W','L','I','S','T'/
C
C
        LSTEND = NEWLST(LSTEND)
        EXECNT = 1
        DOWHILE (1 .EQ. 1)
                READ(5,130,END=999) CMD
130        FORMAT(80A1)
                WRITE(6,132) EXECNT
132        FORMAT(' ','< INPUT ',I3,' : >')
                WRITE(6,134) CMD
134        FORMAT(' ','<',60A1,'>')
                WRITE(6,136)
136        FORMAT(' ','< OUTPUT : >')
                EXECNT = EXECNT + 1
                IF (CODE(CMD,80,CMD1,8)) THEN
                        ARG = GETARG(CMD,80)
                        LSTEND = ADFRST(POOL,LSTEND,ARG)
                ELSEIF (CODE(CMD,80,CMD2,7)) THEN
                        ARG = GETARG(CMD,80)
                        LSTEND = ADLAST(POOL,LSTEND,ARG)
                ELSEIF (CODE(CMD,80,CMD3,11)) THEN
                        LSTEND = DELFST(POOL,LSTEND)
                ELSEIF (CODE(CMD,80,CMD4,10)) THEN
                        LSTEND = DELLST(LSTEND)
                ELSEIF (CODE(CMD,80,CMD5,7)) THEN
                        ITMP = EMPTY(LSTEND)
                        WRITE(6,100) ITMP
100                FORMAT(' ','   VALUE IS ',I10)
                ELSEIF (CODE(CMD,80,CMD6,5)) THEN
                        ITMP = FIRST(POOL,LSTEND)
                        WRITE(6,110) ITMP
110                FORMAT(' ','   VALUE IS ',I10)
                ELSEIF (CODE(CMD,80,CMD7,10)) THEN
                        ITMP = LSTLEN(LSTEND)
                        WRITE(6,120) ITMP
120                FORMAT(' ','   VALUE IS ',I10)
                ELSEIF (CODE(CMD,80,CMD8,7)) THEN
                        CALL REVERS(POOL,LSTEND)
                ELSEIF (CODE(CMD,80,CMD9,7)) THEN
                LSTEND = NEWLST(LSTEND)
            ELSE
                WRITE(6,210) CMD
210          FORMAT(' ','< UNKNOWN COMMAND: ',50A1,'>')
            ENDIF
C
C PRINT PRINTS THE LIST
C
            CALL PRINT(POOL,LSTEND)
        ENDDO
999     CONTINUE
        RETURN
        END
C
C CODE(CA1,LEN1,CA2,LEN2) RETURNS TRUE IF THE FIRST STRING IN CHARACTER
C   ARRAY CA1 IS EQUIVALENT TO THE FIRST STRING IN CA2.  IT RETURNS FALSE
C   OTHERWISE.
C
        FUNCTION CODE (CA1,LEN1,CA2,LEN2)
        LOGICAL*1 CODE, CA1(120), CA2(120)
        INTEGER LEN1, LEN2
C
        INTEGER I1, I2
        LOGICAL*1 CHAREQ
        I1 = 1
        I2 = 1
        DOWHILE ((I1 .LE. LEN1).AND. CHAREQ(CA1(I1),' ') )
            I1 = I1 + 1
        ENDDO
        DOWHILE ((I2 .LE. LEN2).AND. CHAREQ(CA2(I2),' ') )
            I2 = I2 + 1
        ENDDO
C
        DOWHILE ((CHAREQ(CA1(I1),CA2(I2))) .AND.
   X     (.NOT.(CHAREQ(CA1(I1),' '))).AND.(.NOT.(CHAREQ(CA2(I2),' ')))
   X     .AND.(I1 .LE. LEN1).AND.(I2 .LE. LEN2) )
            I1 = I1 + 1
            I2 = I2 + 1
        ENDDO
        IF (((LEN2 .LE. LEN1).AND.(I2 .GT. LEN2)) .OR.
   X      ((LEN2 .GT. LEN1).AND.(I1 .GT. LEN1)) ) THEN
            CODE = 1
          ELSE
            CODE = 0
        ENDIF
        RETURN
        END
C
C CHAREQ(C1,C2) IS CHARACTER EQUIVALENCE; RETURNS TRUE IF CHARACTER C1
C   EQUALS CHARACTER C2, RETURNS FALSE OTHERWISE
C
        FUNCTION CHAREQ (C1,C2)
        LOGICAL*1 CHAREQ, C1, C2
```

```
C
        IF (C1 .EQ. C2) THEN
                CHAREQ = 1
            ELSE
                CHAREQ = 0
        ENDIF
        RETURN
        END
C
C GETARG RETURNS THE SECOND STRING IN CHARACTER ARRAY CA CONVERTED TO AN
C   INTEGER.
C
        FUNCTION GETARG (CA, LEN)
        INTEGER GETARG, LEN
        LOGICAL*1 CA(120)
C
        INTEGER I, SUM, NEGSUM
        LOGICAL*1 CHAREQ
        I = 1
C STRIP LEADING BLANKS
        DOWHILE ((I .LE. LEN).AND. CHAREQ(CA(I),' ') )
                I = I + 1
        ENDDO
C SKIP OVER COMMAND NAME
        DOWHILE ((I .LE. LEN).AND.(.NOT.(CHAREQ(CA(I),' '))) )
                I = I + 1
        ENDDO
C SKIP BLANKS BETWEEN COMMAND AND ARGUMENT
        DOWHILE ((I .LE. LEN).AND. CHAREQ(CA(I),' ') )
                I = I + 1
        ENDDO
C CALCULATE ARGUMET
        SUM = 0
        NEGSUM = 0
        DOWHILE ((I .LE. LEN).AND.(.NOT.(CHAREQ(CA(I),' '))) )
                IF (CHAREQ(CA(I),'-')) THEN
                        NEGSUM = 1
                    ELSE
                        SUM = SUM * 10 + CA(I) - 48
                ENDIF
                I = I + 1
        ENDDO
        IF (NEGSUM .EQ. 0) THEN
                GETARG = SUM
            ELSE
                GETARG = -SUM
        ENDIF
        RETURN
        END
```

PROGRAM 2 FAULTS

1. FIRST returns 0 when the list has one element.
2. ISEMPTY returns 1 when the list has one element.
3. DELETEFIRST cannot delete the first element when the list has one element.
4. LISTLENGTH returns 1 less than the actual length of the list.
5. ADDFIRST can add more than five elements to the list.
6. ADDLAST can add more than five elements to the list.
7. REVERSE does not reverse the list properly.

VAX & IBM: Structural Testing

Specification

(Note from Rick: a 'file' is the same thing as an IBM 'dataset'.)

The program maintains a database of bibliographic references. It first reads a master file of current references, then reads a file of reference updates, merges the two, and produces an updated master file and a cross reference table of keywords.

The first input file, the master, contains records of 74 characters with the following format:

column	comment
1 - 3	each reference has a unique reference key
4 - 14	author of publication
15 - 72	title of publication
73 - 74	year issued

The key should be a three (3) character unique identifier consisting of letters between A-Z. The next input file, the update file, contains records of 75 characters in length. The only difference from a master file record is that an update record has either an 'A' (capital A meaning add) or a 'R' (capital R meaning replace) in column 75. Both the master and update files are expected to be already sorted alphabetically by reference key when read into the program. Update records with action replace are substituted for the matching key record in the master file. Records with action add are added to the master file at the appropriate location so that the file remains sorted on the key field. For example, a valid update record to be read would be (including a numbered line just for reference)

```
123456789012345678901234567890123456789012345678901234567890123456789012345
BITbaker      an introduction to program testing                          83A
```

The program should produce two pieces of output. It should first print the sorted list of records in the updated master file in the same format as the original master file. It should then print a keyword cross reference list. All words greater than three characters in a publication's title are keywords. These keywords are listed alphabetically followed by the key fields from the applicable updated master file entries. For example, if the updated master file contained two records,

```
ABCkermit     introduction to software testing                            82
DDXJones      the realities of software management                        81
```

then the keywords are introduction, testing, realities, software, and management. The cross reference list should look like

```
introduction
    ABC
management
    DDX
```

realities
    DDX
software
    ABC
    DDX
testing
    ABC

Some possible error conditions that could arise and the subsequent actions include the following. The master and update files should be checked for sequence, and if a record out of sequence is found, a message similar to 'key ABC out of sequence' should appear and the record should be discarded. If an update record indicates replace and the matching key can not be found, a message similar to 'update key ABC not found' should appear and the update record should be ignored. If an update record indicates add and a matching key is found, something like 'key ABC already in file' should appear and the record should be ignored. (End of specification.)

3) Reading Functional Structural   ID# ______

```
C NOTE THAT YOU DO NOT NEED TO VERIFY THE ROUTINES DRIVER, STREQ, WORDEQ,
C   NXTSTR, ARRCPY, CHARPT, BEFORE, CHAREQ, AND WRDBEF. THEIR SOURCE
C   CODE IS DESCRIBED AND INCLUDED AT THE END FOR COMPLETENESS.
C NOTE THAT FORMAT STATEMENTS FOR WRITE STATEMENTS INCLUDE A LEADING
C   AND REQUIRED ' ' FOR CARRIAGE CONTROL
C THE SFORT LANGUAGE CONSTRUCT '.IF (EXPRESSION)' BEGINS A BLOCKED
C   IF-THEN[-ELSE] STATEMENT, AND IT IS EQUIVALENT TO THE F77
C   'IF (EXPRESSION) THEN'.
C
      CALL DRIVER
      STOP
      END
C
C
      SUBROUTINE MAINSB
C
      LOGICAL*1 U$KEY(3),U$AUTH(11),U$TITL(58),U$YEAR(2),U$ACTN(1)
      LOGICAL*1 M$KEY(3),M$AUTH(11),M$TITL(58),M$YEAR(2)
      LOGICAL*1 ZZZ(3), LASTUK(3), LASTMK(3)
      LOGICAL*1 STREQ, CHAREQ, BEFORE, CHARPT
      INTEGER I
C
      LOGICAL*1 WORD(500,12), REFKEY(1000,3)
      INTEGER NUMWDS, NUMREF, PTR(500), NEXT(1000)
      COMMON /WORDS/ WORD, REFKEY, NUMWDS, NUMREF, PTR, NEXT
C
      WRITE(6,290)
290   FORMAT(' ','    UPDATED LIST OF MASTER ENTRIES')
      DO 300 I = 1, 3
          LASTMK(I) = CHARPT(' ')
          LASTUK(I) = CHARPT(' ')
          ZZZ(I) = CHARPT('Z')
300   CONTINUE
C
      NUMWDS = 0
      NUMREF = 0
      CALL GETNM(M$KEY,M$AUTH,M$TITL,M$YEAR,LASTMK)
      CALL GETNUP(U$KEY,U$AUTH,U$TITL,U$YEAR,U$ACTN,LASTUK)
C
      DOWHILE ((.NOT.(STREQ(M$KEY,ZZZ,3))) .OR.
     X         (.NOT.(STREQ(U$KEY,ZZZ,3))) )
          .IF (STREQ(U$KEY,M$KEY,3))
              .IF (.NOT.(CHAREQ(U$ACTN(1),'R')))
                  WRITE(6,100) U$KEY
100               FORMAT(' ','KEY ',3A1,' IS ALREADY IN FILE')
              ENDIF
              CALL OUTPUT(U$KEY,U$AUTH,U$TITL,U$YEAR)
              CALL DICTUP(U$KEY,U$TITL,58)
              CALL GETNM(M$KEY,M$AUTH,M$TITL,M$YEAR,LASTMK)
              CALL GETNUP(U$KEY,U$AUTH,U$TITL,U$YEAR,U$ACTN,LASTUK)
          ENDIF
C
          .IF (BEFORE(M$KEY,3,U$KEY,3))
```

```
              CALL OUTPUT(M$KEY,M$AUTH,M$TITL,M$YEAR)
              CALL DICTUP(M$KEY,M$TITL,58)
              CALL GETNM(M$KEY,M$AUTH,M$TITL,M$YEAR,LASTMK)
          ENDIF
C
          .IF (BEFORE(U$KEY,3,M$KEY,3))
              .IF (CHAREQ(U$ACTN(1),'R'))
                  WRITE(6,110) U$KEY
110               FORMAT(' ','UPDATE KEY ',3A1,' NOT FOUND')
              ENDIF
              CALL OUTPUT(U$KEY,U$AUTH,U$TITL,U$YEAR)
              CALL DICTUP(U$KEY,U$TITL,58)
              CALL GETNUP(U$KEY,U$AUTH,U$TITL,U$YEAR,U$ACTN,LASTUK)
          ENDIF
      ENDDO
C
      CALL SRTWDS
      CALL PRTWDS
      RETURN
      END
C
C
      SUBROUTINE GETNM(KEY,AUTH,TITL,YEAR,LASTMK)
      LOGICAL*1 KEY(3),AUTH(11),TITL(58),YEAR(2),LASTMK(3)
C
      LOGICAL*1 SEQ, INLINE(80)
      LOGICAL*1 BEFORE, CHARPT, CHAREQ
      LOGICAL*1 GO$M, GO$U
      COMMON /DRIV/ GO$M, GO$U
C
      SEQ = 1
      DOWHILE (SEQ)
          .IF (GO$M)
C
C READ FROM THE MASTER FILE
C
              READ(10,200,END=299) INLINE
            ELSE
C
C SEE REMARK ABOUT THE CHARACTER '%' LATER IN THE ROUTINE.
C
              INLINE(1) = CHARPT('%')
          ENDIF
200       FORMAT(80A1)
          DO 210 I = 1, 3
              KEY(I) = INLINE(I)
210       CONTINUE
          DO 220 I = 1, 11
              AUTH(I) = INLINE(3+I)
220       CONTINUE
          DO 230 I = 1, 58
              TITL(I) = INLINE(14+I)
230       CONTINUE
```

```
     DO 240 I = 1, 2
       YEAR(I) = INLINE(72+I)
240  CONTINUE
C
C A METHOD OF SPECIFYING END-OF-FILE IN A FILE IS TO PUT THE CHARACTER '%'
C  AS THE FIRST CHARACTER ON A LINE.  THE DRIVER USES THIS FOR MULTIPLE
C  SETS OF INPUT CASES.
C
     .IF ((.NOT.(CHAREQ(KEY(1),'%'))) .AND.
  X   (BEFORE(KEY,3,LASTMK,3)) )
       WRITE(6,250) KEY
250    FORMAT(' ','KEY ',3A1,' OUT OF SEQUENCE')
      ELSE
       CALL ARRCPY(KEY,LASTMK,3)
       SEQ = 0
     ENDIF
     .IF (CHAREQ(KEY(1),'%'))
       SEQ = 0
       DO 270 I = 1, 3
         KEY(I) = CHARPT('Z')
270    CONTINUE
     ENDIF
   ENDDO
   RETURN
299 CONTINUE
   GO$M = 0
   DO 260 I = 1, 3
     KEY(I) = CHARPT('Z')
260 CONTINUE
   RETURN
   END
C
C
   SUBROUTINE GETNUP(KEY,AUTH,TITL,YEAR,ACTN,LASTUK)
   LOGICAL*1 KEY(3),AUTH(11),TITL(58),YEAR(2),ACTN(1),LASTUK(3)
C
   LOGICAL*1 SEQ, INLINE(80)
   LOGICAL*1 BEFORE, CHARPT, CHAREQ
   LOGICAL*1 GO$M, GO$U
   COMMON /DRIV/ GO$M, GO$U
C
   SEQ = 1
   DOWHILE (SEQ)
     .IF (GO$U)
C
C READ FROM THE UPDATES FILE
C
       READ(11,200,END=299) INLINE
      ELSE
C
C SEE REMARK ABOUT THE CHARACTER '%' LATER IN THE ROUTINE.
C
       INLINE(1) = CHARPT('%')
```

```
          ENDIF
200       FORMAT(80A1)
          DO 210 I = 1, 3
              KEY(I) = INLINE(I)
210       CONTINUE
          DO 220 I = 1, 11
              AUTH(I) = INLINE(3+I)
220       CONTINUE
          DO 230 I = 1, 58
              TITL(I) = INLINE(14+I)
230       CONTINUE
          DO 240 I = 1, 2
              YEAR(I) = INLINE(72+I)
240       CONTINUE
          ACTN(1) = INLINE(75)
C
C A METHOD OF SPECIFYING END-OF-FILE IN A FILE IS TO PUT THE CHARACTER '%'
C   AS THE FIRST CHARACTER ON A LINE.  THE DRIVER USES THIS FOR MULTIPLE
C   SETS OF INPUT CASES.
C
          .IF ((.NOT.(CHAREQ(KEY(1),'%'))) .AND.
     X      (BEFORE(KEY,3,LASTUK,3)) )
              WRITE(6,250) KEY
250           FORMAT(' ','KEY ',3A1,' OUT OF SEQUENCE')
            ELSE
              CALL ARRCPY(KEY,LASTUK,3)
              SEQ = 0
          ENDIF
          .IF (CHAREQ(KEY(1),'%'))
              SEQ = 0
              DO 270 I = 1, 3
                  KEY(I) = CHARPT('Z')
270           CONTINUE
          ENDIF
      ENDDO
      RETURN
299   CONTINUE
      GO$U = 0
      DO 260 I = 1, 3
          KEY(I) = CHARPT('Z')
260   CONTINUE
      RETURN
      END
C
C
      SUBROUTINE OUTPUT(KEY,AUTH,TITL,YEAR)
      LOGICAL*1 KEY(3), AUTH(11), TITL(58), YEAR(2)
C
      WRITE(6,200) KEY, AUTH, TITL, YEAR
200   FORMAT(' ',3A1,11A1,58A1,2A1)
      RETURN
      END
C
```

3) **Reading Functional Structural** ID# ______

```
213: C
214:       SUBROUTINE PRTWDS
215: C
216:       LOGICAL*1 WORD(500,12), REFKEY(1000,3)
217:       INTEGER NUMWDS, NUMREF, PTR(500), NEXT(1000)
218:       COMMON /WORDS/ WORD, REFKEY, NUMWDS, NUMREF, PTR, NEXT
219: C
220: C THE ABOVE GROUP OF DATA STRUCTURES SIMULATES A LINKED LIST.
221: C WORD(I,J) IS A KEYWORD -- J RANGING FROM 1 TO 12
222: C REFKEY(PTR(I),K),K=1,3 IS THE FIRST 3 LETTER KEY THAT HAS AS A
223: C   KEYWORD WORD(I,J),J=1,12
224: C REFKEY(NEXT(PTR(I)),K),K=1,3 IS THE SECOND 3 LETTER KEY THAT HAS
225: C   AS A KEYWORD WORD(I,J),J=1,12
226: C REFKEY(NEXT(NEXT(PTR(I))),K),K=1,3 IS THE THIRD ... ETC.
227: C NEXT(J) IS EQUAL TO -1 WHEN THERE ARE NO MORE 3 LETTER KEYS FOR
228: C   THE PARTICULAR KEYWORD
229: C
230:       INTEGER I, J
231:       LOGICAL*1 FLAG
232: C
233:       WRITE(6,200)
234: 200   FORMAT(' ','    KEYWORD REFERENCE LIST')
235:       DO 210 I = 1, NUMWDS
236:           FLAG = 1
237:           WRITE(6,220) (WORD(I,J),J=1,12)
238: 220       FORMAT(' ',12A1)
239:           LAST = PTR(I)
240:           DOWHILE (FLAG)
241:               WRITE(6,230) (REFKEY(LAST,J),J=1,3)
242: 230           FORMAT(' ','    ',3A1)
243:               LAST = NEXT(LAST)
244:               .IF (LAST .EQ. -1)
245:                   FLAG = 0
246:               ENDIF
247:           ENDDO
248: 210   CONTINUE
249:       RETURN
250:       END
251: C
252: C
253:       SUBROUTINE DICTUP(KEY,STR,STRLEN)
254:       LOGICAL*1 KEY(3), STR(120)
255:       INTEGER STRLEN
256: C
257:       LOGICAL*1 WDLEFT, FLAG, OKLEN, NEXTWD(120), WORDEQ
258:       INTEGER LPTR, NXTSTR, LEN, LAB, I, K
259: C
260:       LOGICAL*1 WORD(500,12), REFKEY(1000,3)
261:       INTEGER NUMWDS, NUMREF, PTR(500), NEXT(1000)
262:       COMMON /WORDS/ WORD, REFKEY, NUMWDS, NUMREF, PTR, NEXT
263: C
264: C THE ABOVE GROUP OF DATA STRUCTURES SIMULATES A LINKED LIST.
265: C WORD(I,J) IS A KEYWORD -- J RANGING FROM 1 TO 12
```

```
C REFKEY(PTR(I),K),K=1,3 IS THE FIRST 3 LETTER KEY THAT HAS AS A
C   KEYWORD WORD(I,J),J=1,12
C REFKEY(NEXT(PTR(I)),K),K=1,3 IS THE SECOND 3 LETTER KEY THAT HAS
C   AS A KEYWORD WORD(I,J),J=1,12
C REFKEY(NEXT(NEXT(PTR(I))),K),K=1,3 IS THE THIRD ... ETC.
C NEXT(J) IS EQUAL TO -1 WHEN THERE ARE NO MORE 3 LETTER KEYS FOR
C   THE PARTICULAR KEYWORD
C
      WDLEFT = 1
      LPTR = 1
C
      DOWHILE (WDLEFT)
          FLAG = 1
          OKLEN = 1
          LEN = NXTSTR(STR,STRLEN,LPTR,NEXTWD,120)
          .IF (LEN .EQ. 0)
              WDLEFT = 0
          ENDIF
C
          .IF (LEN .LE. 2)
              OKLEN = 0
          ENDIF
C
          .IF (OKLEN)
              I = 1
              DOWHILE ((I .LE. NUMWDS).AND. FLAG )
                  .IF (WORDEQ(NEXTWD,I))
                      LAB = I
                      FLAG = 0
                  ENDIF
                  I = I + 1
              ENDDO
              .IF (FLAG)
                  NUMWDS = NUMWDS + 1
                  NUMREF = NUMREF + 1
                  DO 300 K = 1, 12
                      WORD(NUMWDS,K) = NEXTWD(K)
300               CONTINUE
                  PTR(NUMWDS) = NUMREF
                  DO 310 K = 1, 3
                     REFKEY(NUMREF,K) = KEY(K)
310               CONTINUE
                  NEXT(NUMREF) = -1
                ELSE
                  NUMREF = NUMREF + 1
                  DO 320 K = 1, 3
                      REFKEY(NUMREF,K) = KEY(K)
320               CONTINUE
                  NEXT(NUMREF) = PTR(LAB)
                  PTR(LAB) = NUMREF
              ENDIF
          ENDIF
      ENDDO
```

```
C
      RETURN
      END
C
C
      SUBROUTINE SRTWDS
C
      LOGICAL*1 WORD(500,12), REFKEY(1000,3)
      INTEGER NUMWDS, NUMREF, PTR(500), NEXT(1000)
      COMMON /WORDS/ WORD, REFKEY, NUMWDS, NUMREF, PTR, NEXT
C
C THE ABOVE GROUP OF DATA STRUCTURES SIMULATES A LINKED LIST.
C WORD(I,J) IS A KEYWORD -- J RANGING FROM 1 TO 12
C REFKEY(PTR(I),K),K=1,3 IS THE FIRST 3 LETTER KEY THAT HAS AS A
C   KEYWORD WORD(I,J),J=1,12
C REFKEY(NEXT(PTR(I)),K),K=1,3 IS THE SECOND 3 LETTER KEY THAT HAS
C   AS A KEYWORD WORD(I,J),J=1,12
C REFKEY(NEXT(NEXT(PTR(I))),K),K=1,3 IS THE THIRD ... ETC.
C NEXT(J) IS EQUAL TO -1 WHEN THERE ARE NO MORE 3 LETTER KEYS FOR
C   THE PARTICULAR KEYWORD
C
      INTEGER I, J, K, LAB, LOWERB, UPPERB
      LOGICAL*1 WRDBEF, NEXTWD(12)
C
      UPPERB = NUMWDS - 1
      DO 400 I = 1, UPPERB
        LOWERB = I + 1
        DO 410 J = LOWERB, NUMWDS
          .IF (WRDBEF(J,I))
            DO 300 K = 1, 12
              NEXTWD(K) = WORD(I,K)
300         CONTINUE
            LAB = PTR(I)
            DO 310 K = 1, 12
              WORD(I,K) = WORD(J,K)
310         CONTINUE
            PTR(I) = PTR(J)
            DO 320 K = 1, 12
              WORD(J,K) = NEXTWD(K)
320         CONTINUE
            PTR(J) = LAB
          ENDIF
410     CONTINUE
400   CONTINUE
C
      RETURN
      END
C
C
C NOTE: YOU DO NOT NEED TO VERIFY THE FOLLOWING PROCEDURES.  THEIR SOURCE
C   CODE IS INCLUDED JUST FOR COMPLETENESS.
C
C DRIVER CONTINUES TO CALL THE MAIN ROUTINE UNTIL END-OF-FILE HAS BEEN
```

```
C   REACHED ON BOTH THE MASTER AND UPDATES DATASETS.
C
      SUBROUTINE DRIVER
C
      INTEGER EXECNT
      LOGICAL*1 GO$M, GO$U
      COMMON /DRIV/ GO$M, GO$U
C
      EXECNT = 1
      GO$M = 1
      GO$U = 1
      DOWHILE ((GO$M).OR.(GO$U))
          WRITE(6,90) EXECNT
 90       FORMAT(' ','< PROGRAM EXECUTION ',I3,' : >')
          CALL MAINSB
          EXECNT = EXECNT + 1
      ENDDO
      RETURN
      END
C
C WRDBEF(P1,P2) RETURNS 1 IF THE KEYWORD IN WORD(P1,J),J=1,12 COMES
C   ALPHABETICALLY BEFORE THE KEYWORD IN WORD(P2,J),J=1,12.
C   IT RETURNS 0 OTHERWISE.
C
      FUNCTION WRDBEF(PTR1,PTR2)
      LOGICAL*1 WRDBEF
      INTEGER PTR1, PTR2
C
      LOGICAL*1 WORD(500,12), REFKEY(1000,3)
      INTEGER NUMWDS, NUMREF, PTR(500), NEXT(1000)
      COMMON /WORDS/ WORD, REFKEY, NUMWDS, NUMREF, PTR, NEXT
      LOGICAL*1 BEFORE, WORD1(12), WORD2(12)
      INTEGER I
C
      DO 500 I = 1, 12
          WORD1(I) = WORD(PTR1,I)
          WORD2(I) = WORD(PTR2,I)
500   CONTINUE
      .IF (BEFORE(WORD1,12,WORD2,12))
          WRDBEF = 1
        ELSE
          WRDBEF = 0
      ENDIF
      RETURN
      END
C
C CHAREQ(C1,C2) IS CHARACTER EQUIVALENCE; RETURNS TRUE IF CHARACTER C1
C   EQUALS CHARACTER C2, RETURNS FALSE OTHERWISE
C
      FUNCTION CHAREQ (C1,C2)
      LOGICAL*1 CHAREQ, C1, C2
C
      .IF (C1..EQ. C2)
```

```
          CHAREQ = 1
        ELSE
          CHAREQ = 0
      ENDIF
      RETURN
      END
C
C BEFORE(S1,L1,S2,L2) RETURNS 1 IF THE STRING IN CHAR ARRAY S1 COMES
C   ALPHABETICALLY BEFORE THE STRING IN CHAR ARRAY S2.  S1 AND S2 ARE
C   OF SIZES L1 AND L2.  IT RETURNS 0 OTHERWISE (INCLUDING WHEN THE
C   STRINGS ARE EXACTLY THE SAME).
C
      FUNCTION BEFORE (STR1,LEN1,STR2,LEN2)
      LOGICAL*1 BEFORE, STR1(120), STR2(120)
      INTEGER LEN1, LEN2
C
      INTEGER PTR
      LOGICAL*1 TIE
      PTR = 1
      TIE = 1
      DOWHILE ( TIE .AND.(PTR.LE.LEN1).AND.(PTR.LE.LEN2))
          .IF (STR1(PTR) .LT. STR2(PTR))
              BEFORE = 1
              TIE = 0
            ELSE
              .IF (STR1(PTR) .GT. STR2(PTR))
                  BEFORE = 0
                  TIE = 0
              ENDIF
          ENDIF
          PTR = PTR + 1
      ENDDO
      .IF (TIE)
          .IF ((PTR .GT. LEN1).AND.(PTR .LE. LEN2))
              BEFORE = 1
            ELSE
              BEFORE = 0
          ENDIF
      ENDIF
      RETURN
      END
C
C CHARPT('C') IS A CHARACTER ASSIGNMENT FUNCTION; IT RETURNS THE
C   CHARACTER PASSED TO IT AS AN ARGUMENT
C
      FUNCTION CHARPT(C)
      LOGICAL*1 CHARPT, C
C
      CHARPT = C
      RETURN
      END
C
C ARRCPY(ARR1,ARR2,ARSIZE) COPIES CHARACTER ARRAY ARR1 TO CHARACTER
```

```
C   ARRAY ARR2. THE ARRAYS ARE OF SIZE ARSIZE
C
      SUBROUTINE ARRCPY (ARR1,ARR2,LEN)
      LOGICAL*1 ARR1(120), ARR2(120)
      INTEGER LEN
C
      INTEGER I
      DO 230 I = 1, LEN
          ARR2(I) = ARR1(I)
230   CONTINUE
      RETURN
      END
C
C NXTSTR(STR,LEN,START,NEXTST,NXTLEN) COPIES THE NEXT STRING IN
C   CHAR ARRAY STR TO CHAR ARRAY NEXTST.  THE FORWARD SEARCH FOR THE
C   NEXT STRING BEGINS AT INDEX START. START IS UPDATED TO BE THE INDEX OF THE
C   CHARACTER IMMEDIATELY FOLLOWING THE STRING IN STR.
C   THE SIZE OF STR IS LEN. THE LENGTH OF THE NEXT STRING IS RETURNED
C   AS THE FUNCTION'S VALUE.  THE SIZE OF THE CHAR ARRAY NEXTST IS NXTLEN.
C
      FUNCTION NXTSTR(STR,LEN,START,NEXTST,NXTLEN)
      INTEGER NXTSTR
      LOGICAL*1 STR(120), NEXTST(120)
      INTEGER LEN, START, NXTLEN
C
      INTEGER I, STRPTR
      LOGICAL*1 CHAREQ, CHARPT
C
      DOWHILE ((START.LE.LEN).AND. CHAREQ(STR(START),' ') )
          START = START + 1
      ENDDO
      STRPTR = 1
      DOWHILE ((START.LE.LEN).AND.(.NOT.(CHAREQ(STR(START),' '))) )
          NEXTST(STRPTR) = STR(START)
          START = START + 1
          STRPTR = STRPTR + 1
      ENDDO
      I = STRPTR
      DOWHILE (I.LE.NXTLEN)
          NEXTST(I) = CHARPT(' ')
          I = I + 1
      ENDDO
      NXTSTR = STRPTR - 1
      RETURN
      END
C
C STREQ(STR1,STR2,LEN) RETURNS 1 IF THE STRING IN CHAR ARRAY STR1 IS
C   EQUIVALENT TO THE STRING IN CHARACTER ARRAY STR2.
C   IT RETURNS 0 OTHERWISE. ARRAYS STR1 AND STR2 ARE OF SIZE LEN.
C
      FUNCTION STREQ(STR1,STR2,LEN)
      LOGICAL*1 STREQ, STR1(120), STR2(120)
      INTEGER LEN
```

```
C
      INTEGER I, START, I1, I2, NXTSTR
      LOGICAL*1 CHAREQ
      LOGICAL*1 WORD1(120), WORD2(120)
C
      START = 1
      I1 = NXTSTR(STR1,LEN,START,WORD1,120)
      START = 1
      I2 = NXTSTR(STR2,LEN,START,WORD2,120)
      I = 1
      DOWHILE ((I.LE.I1).AND.(I.LE.I2).AND. CHAREQ(WORD1(I),WORD2(I))
     X    .AND.(.NOT.(CHAREQ(WORD1(I),' ')))
     X    .AND.(.NOT.(CHAREQ(WORD2(I),' '))) )
          I = I + 1
      ENDDO
      .IF (((I.GT.I1).AND.(I.GT.I2)) .OR.
     X   ((CHAREQ(WORD1(I),' ')).AND.(CHAREQ(WORD2(I),' '))) )
          STREQ = 1
        ELSE
          STREQ = 0
      ENDIF
      RETURN
      END
C
C WORDEQ(WORD1,PTR2) RETURNS 1 IF THE KEYWORD IN WORD(PTR2,J),J=1,12 IS
C   EQUIVALENT TO THE WORD IN CHARACTER ARRAY WORD1.
C   IT RETURNS 0 OTHERWISE.
C
      FUNCTION WORDEQ(WORD1,PTR2)
      LOGICAL*1 WORDEQ
      LOGICAL*1 WORD1(12)
      INTEGER PTR2
C
      LOGICAL*1 WORD(500,12), REFKEY(1000,3)
      INTEGER NUMWDS, NUMREF, PTR(500), NEXT(1000)
      COMMON /WORDS/ WORD, REFKEY, NUMWDS, NUMREF, PTR, NEXT
C
      LOGICAL*1 WORD2(12), STREQ
      INTEGER I
C
      DO 700 I = 1, 12
          WORD2(I) = WORD(PTR2,I)
700   CONTINUE
      .IF (STREQ(WORD1,WORD2,12))
          WORDEQ = 1
        ELSE
          WORDEQ = 0
      ENDIF
      RETURN
      END
```

PROGRAM 3 FAULTS

1. Three-character words are treated as keywords.

2. The key 'zzz' is not recognized.

2.[a] Any key greater than 'zzz' causes loop.

3. If action ADD occurs with key already in file, the program acts like REPLACE; the update record is not skipped.

4. If REPLACE key is not found, the program acts like ADD; the update record is not skipped.

5. A maximum of 500 keywords and 1000 reference keys are allowed.

6. Greater than 2 transactions for the same master record produces incorrect results.

7. Keywords greater than 12 characters are truncated and not distinguished.

8. UPDATE transaction with column 80 not an 'R' produces same result as ADD.

9. Keyword indices appear in the opposite order from that shown in specifications.

10. No check is made for unique keys in the master file.

11. Punctuation is made a part of the keyword.

12. A word appearing twice in a title gets two cross-reference entries.

[a]Alternate manifestation of this error.

APPENDIX B - DATA SUMMARY

This appendix contains a set of tables summarizing the results obtained from those subjects who passed the initial screening and actually participated in the experiment. The original materials provided by each subject are reproduced in the Data Supplement.

SUBJECT 02

EXPERIENCE LEVEL: Junior

COMPUTER: IBM

PROGRAM 1:

Verification Method:	Functional
Percent Faults Found:	44
Percent Estimated Faults:	20
Hours To Detect:	4.75
Hours To Correct:	1.25
Hours per Fault:	1.5

PROGRAM 2:

Verification Method:	Reading
Percent Faults Found:	86
Percent Estimated Faults:	75
Hours To Detect:	5
Hours To Correct:	2.5
Hours per Fault:	1.25

PROGRAM 3:

Verification Method:	Structural
Percent Faults Found:	17
Percent Estimated Faults:	30
Hours To Detect:	6.75
Hours To Correct:	1
Hours per Fault:	3.88

SUBJECT 03

EXPERIENCE LEVEL: Intermediate

COMPUTER: IBM

PROGRAM 1:

Verification Method:	Reading
Percent Faults Found:	56
Percent Estimated Faults:	90
Hours To Detect:	3
Hours To Correct:	3
Hours per Fault:	1.2

PROGRAM 2:

Verification Method:	Structural
Percent Faults Found:	100
Percent Estimated Faults:	95
Hours To Detect:	2.5
Hours To Correct:	0.75
Hours per Fault:	0.46

PROGRAM 3:

Verification Method:	Functional
Percent Faults Found:	42
Percent Estimated Faults:	70
Hours To Detect:	3.5
Hours To Correct:	3
Hours per Fault:	1.3

SUBJECT 04

EXPERIENCE LEVEL: Junior

COMPUTER: VAX

PROGRAM 1:

Verification Method: Structural
Percent Faults Found: 33
Percent Estimated Faults: 100
Hours To Detect: 0.75
Hours To Correct: 1
Hours per Fault: 0.58

PROGRAM 2:

Verification Method: Reading
Percent Faults Found: 100
Percent Estimated Faults: 100
Hours To Detect: 2
Hours To Correct: 0.5
Hours per Fault: 0.36

PROGRAM 3:

Verification Method: Functional
Percent Faults Found: 50
Percent Estimated Faults: 70
Hours To Detect: 4
Hours To Correct: 4
Hours per Fault: 1.33

SUBJECT 05

EXPERIENCE LEVEL:	Intermediate
COMPUTER:	VAX

PROGRAM 1:

Verification Method:	Reading
Percent Faults Found:	78
Percent Estimated Faults:	100
Hours To Detect:	3.5
Hours To Correct:	4.25
Hours per Fault:	1.11

PROGRAM 2:

Verification Method:	Structural
Percent Faults Found:	43
Percent Estimated Faults:	100
Hours To Detect:	4
Hours To Correct:	3.5
Hours per Fault:	2.5

PROGRAM 3:

Verification Method:	Functional
Percent Faults Found:	90
Percent Estimated Faults:	43
Hours To Detect:	6.25
Hours To Correct:	1.75
Hours per Fault:	1.60

SUBJECT 06

EXPERIENCE LEVEL: Junior

COMPUTER: VAX

PROGRAM 1:

Verification Method:	Functional
Percent Faults Found:	67
Percent Estimated Faults:	75
Hours To Detect:	1.5
Hours To Correct:	1
Hours per Fault:	0.42

PROGRAM 2:

Verification Method:	Structural
Percent Faults Found:	29
Percent Estimated Faults:	50
Hours To Detect:	0.75
Hours To Correct:	0.25
Hours per Fault:	0.5

PROGRAM 3:

Verification Method:	Reading
Percent Faults Found:	25
Percent Estimated Faults:	50
Hours To Detect:	1.5
Hours To Correct:	0.25
Hours per Fault:	0.58

SUBJECT 08

EXPERIENCE LEVEL: Junior

COMPUTER: IBM

PROGRAM 1:

Verification Method:	Reading
Percent Faults Found:	44
Percent Estimated Faults:	93
Hours To Detect:	2.5
Hours To Correct:	3.5
Hours per Fault:	1.5

PROGRAM 2:

Verification Method:	Functional
Percent Faults Found:	57
Percent Estimated Faults:	85
Hours To Detect:	4.5
Hours To Correct:	1
Hours per Fault:	1.38

PROGRAM 3:

Verification Method:	Structural
Percent Faults Found:	17
Percent Estimated Faults:	80
Hours To Detect:	3.75
Hours To Correct:	0.75
Hours per Fault:	2.25

SUBJECT 10

EXPERIENCE LEVEL: Advanced

COMPUTER: IBM

PROGRAM 1:

Verification Method:	Functional
Percent Faults Found:	78
Percent Estimated Faults:	50
Hours To Detect:	5.5
Hours To Correct:	2.5
Hours per Fault:	1.14

PROGRAM 2:

Verification Method:	Reading
Percent Faults Found:	100
Percent Estimated Faults:	100
Hours To Detect:	1.25
Hours To Correct:	1.5
Hours per Fault:	0.39

PROGRAM 3:

Verification Method:	Structural
Percent Faults Found:	50
Percent Estimated Faults:	80
Hours To Detect:	3
Hours To Correct:	2
Hours per Fault:	0.83

SUBJECT 11

EXPERIENCE LEVEL: Junior

COMPUTER: VAX

PROGRAM 1:

Verification Method:	Reading
Percent Faults Found:	67
Percent Estimated Faults:	75
Hours To Detect:	2.25
Hours To Correct:	0.5
Hours per Fault:	0.46

PROGRAM 2:

Verification Method:	Structural
Percent Faults Found:	86
Percent Estimated Faults:	100
Hours To Detect:	1.5
Hours To Correct:	0.25
Hours per Fault:	0.29

PROGRAM 3:

Verification Method:	Functional
Percent Faults Found:	42
Percent Estimated Faults:	90
Hours To Detect:	1.5
Hours To Correct:	1.5
Hours per Fault:	0.6

SUBJECT 12

EXPERIENCE LEVEL: Junior

COMPUTER: VAX

PROGRAM 1:

Verification Method: Functional
Percent Faults Found: 56
Percent Estimated Faults: 80
Hours To Detect: 3
Hours To Correct: 2
Hours per Fault: 1

PROGRAM 2:

Verification Method: Structural
Percent Faults Found: 43
Percent Estimated Faults: 75
Hours To Detect: 3.5
Hours To Correct: 3
Hours per Fault: 2.17

PROGRAM 3:

Verification Method: Reading
Percent Faults Found: 17
Percent Estimated Faults: 10
Hours To Detect: 6
Hours To Correct: 1
Hours per Fault: 3.5

SUBJECT 13

EXPERIENCE LEVEL: Junior

COMPUTER: VAX

PROGRAM 1:

Verification Method:	Structural
Percent Faults Found:	11
Percent Estimated Faults:	60
Hours To Detect:	1
Hours To Correct:	4
Hours per Fault:	5

PROGRAM 2:

Verification Method:	Functional
Percent Faults Found:	57
Percent Estimated Faults:	80
Hours To Detect:	1.5
Hours To Correct:	2
Hours per Fault:	0.88

PROGRAM 3:

Verification Method:	Reading
Percent Faults Found:	0
Percent Estimated Faults:	5
Hours To Detect:	4.5
Hours To Correct:	1.5
Hours per Fault:	-

SUBJECT 14

EXPERIENCE LEVEL: Junior

COMPUTER: IBM

PROGRAM 1:

Verification Method:	Code Reading
Percent Faults Found:	89
Percent Estimated Faults:	90
Hours To Detect:	4
Hours To Correct:	1
Hours per Fault:	0.63

PROGRAM 2:

Verification Method:	Structural
Percent Faults Found:	43
Percent Estimated Faults:	90
Hours To Detect:	3
Hours To Correct:	3.25
Hours per Fault:	2.08

PROGRAM 3:

Verification Method:	Functional
Percent Faults Found:	33
Percent Estimated Faults:	80
Hours To Detect:	4
Hours To Correct:	3.5
Hours per Fault:	1.88

SUBJECT 15

EXPERIENCE LEVEL: Intermediate

COMPUTER: VAX

PROGRAM 1:

Verification Method:	Functional
Percent Faults Found:	67
Percent Estimated Faults:	75
Hours To Detect:	4.5
Hours To Correct:	3.5
Hours per Fault:	1.33

PROGRAM 2:

Verification Method:	Reading
Percent Faults Found:	100
Percent Estimated Faults:	75
Hours To Detect:	2.5
Hours To Correct:	1
Hours per Fault:	0.50

PROGRAM 3:

Verification Method:	Structural
Percent Faults Found:	33
Percent Estimated Faults:	60
Hours To Detect:	3.5
Hours To Correct:	2
Hours per Fault:	1.38

SUBJECT 17

EXPERIENCE LEVEL: Junior

COMPUTER: IBM

PROGRAM 1:

Verification Method:	Structural
Percent Faults Found:	33
Percent Estimated Faults:	40
Hours To Detect:	3
Hours To Correct:	1.5
Hours per Fault:	1.17

PROGRAM 2:

Verification Method:	Functional
Percent Faults Found:	57
Percent Estimated Faults:	60
Hours To Detect:	3
Hours To Correct:	0.5
Hours per Fault:	0.88

PROGRAM 3:

Verification Method:	Reading
Percent Faults Found:	25
Percent Estimated Faults:	30
Hours To Detect:	5
Hours To Correct:	1
Hours per Fault:	2

SUBJECT 21

EXPERIENCE LEVEL: Junior

COMPUTER: VAX

PROGRAM 1:

Verification Method:	Functional
Percent Faults Found:	33
Percent Estimated Faults:	90
Hours To Detect:	1.25
Hours To Correct:	1.5
Hours per Fault:	0.92

PROGRAM 2:

Verification Method:	Structural
Percent Faults Found:	71
Percent Estimated Faults:	100
Hours To Detect:	1.25
Hours To Correct:	0.5
Hours per Fault:	0.35

PROGRAM 3:

Verification Method:	Reading
Percent Faults Found:	17
Percent Estimated Faults:	20
Hours To Detect:	5
Hours To Correct:	0
Hours per Fault:	2.5

SUBJECT 23

EXPERIENCE LEVEL: Intermediate

COMPUTER: IBM

PROGRAM 1:

Verification Method:	Structural
Percent Faults Found:	11
Percent Estimated Faults:	2
Hours To Detect:	0.5
Hours To Correct:	0.5
Hours per Fault:	1

PROGRAM 2:

Verification Method:	Reading
Percent Faults Found:	71
Percent Estimated Faults:	90
Hours To Detect:	0.5
Hours To Correct:	0.5
Hours per Fault:	0.2

PROGRAM 3:

Verification Method:	Functional
Percent Faults Found:	8
Percent Estimated Faults:	60
Hours To Detect:	3
Hours To Correct:	1
Hours per Fault:	4

SUBJECT 25

EXPERIENCE LEVEL: Intermediate

COMPUTER: IBM

PROGRAM 1:

Verification Method:	Reading
Percent Faults Found:	44
Percent Estimated Faults:	70
Hours To Detect:	3.5
Hours To Correct:	2
Hours per Fault:	1.25

PROGRAM 2:

Verification Method:	Functional
Percent Faults Found:	100
Percent Estimated Faults:	100
Hours To Detect:	3.25
Hours To Correct:	0.25
Hours per Fault:	0.5

PROGRAM 3:

Verification Method:	Structural
Percent Faults Found:	0
Percent Estimated Faults:	0
Hours To Detect:	0.75
Hours To Correct:	0.5
Hours per Fault:	-

SUBJECT 26

EXPERIENCE LEVEL:	Intermediate
COMPUTER:	VAX

PROGRAM 1:

Verification Method:	Structural
Percent Faults Found:	22
Percent Estimated Faults:	50
Hours To Detect:	2
Hours To Correct:	0.75
Hours per Fault:	1.38

PROGRAM 2:

Verification Method:	Functional
Percent Faults Found:	86
Percent Estimated Faults:	75
Hours To Detect:	3.25
Hours To Correct:	0.75
Hours per Fault:	0.67

PROGRAM 3:

Verification Method:	Reading
Percent Faults Found:	25
Percent Estimated Faults:	75
Hours To Detect:	2
Hours To Correct:	0.5
Hours per Fault:	0.83

SUBJECT 28

EXPERIENCE LEVEL: Intermediate

COMPUTER: VAX

PROGRAM 1:

Verification Method:	Functional
Percent Faults Found:	44
Percent Estimated Faults:	95
Hours To Detect:	0.75
Hours To Correct:	2
Hours per Fault:	0.69

PROGRAM 2:

Verification Method:	Reading
Percent Faults Found:	100
Percent Estimated Faults:	100
Hours To Detect:	0.75
Hours To Correct:	0.5
Hours per Fault:	0.18

PROGRAM 3:

Verification Method:	Structural
Percent Faults Found:	17
Percent Estimated Faults:	60
Hours To Detect:	2
Hours To Correct:	2.5
Hours per Fault:	2.25

SUBJECT 29

EXPERIENCE LEVEL: Advanced

COMPUTER: IBM

PROGRAM 1:

Verification Method: Structural
Percent Faults Found: 33
Percent Estimated Faults: 80
Hours To Detect: 2
Hours To Correct: 2.25
Hours per Fault: 1.42

PROGRAM 2:

Verification Method: Functional
Percent Faults Found: 43
Percent Estimated Faults: 100
Hours To Detect: 2.75
Hours To Correct: 1.25
Hours per Fault: 1.33

PROGRAM 3:

Verification Method: Reading
Percent Faults Found: 50
Percent Estimated Faults: 90
Hours To Detect: 2
Hours To Correct: 1.5
Hours per Fault: 0.88

SUBJECT 30

EXPERIENCE LEVEL: Intermediate

COMPUTER: VAX

PROGRAM 1:

Verification Method:	Reading
Percent Faults Found:	44
Percent Estimated Faults:	81
Hours To Detect:	3
Hours To Correct:	0.5
Hours per Fault:	0.88

PROGRAM 2:

Verification Method:	Functional
Percent Faults Found:	86
Percent Estimated Faults:	90
Hours To Detect:	2.25
Hours To Correct:	0.75
Hours per Fault:	0.5

PROGRAM 3:

Verification Method:	Structural
Percent Faults Found:	25
Percent Estimated Faults:	80
Hours To Detect:	2.5
Hours To Correct:	1.75
Hours per Fault:	1.42

SUBJECT 35

EXPERIENCE LEVEL: Advanced

COMPUTER: IBM

PROGRAM 1:

Verification Method:	Reading
Percent Faults Found:	67
Percent Estimated Faults:	85
Hours To Detect:	2
Hours To Correct:	2
Hours per Fault:	0.67

PROGRAM 2:

Verification Method:	Structural
Percent Faults Found:	100
Percent Estimated Faults:	85
Hours To Detect:	3.5
Hours To Correct:	1
Hours per Fault:	0.64

PROGRAM 3:

Verification Method:	Functional
Percent Faults Found:	42
Percent Estimated Faults:	75
Hours To Detect:	2
Hours To Correct:	3
Hours per Fault:	1

SUBJECT 36

EXPERIENCE LEVEL: Intermediate

COMPUTER: VAX

PROGRAM 1:

Verification Method:	Structural
Percent Faults Found:	33
Percent Estimated Faults:	50
Hours To Detect:	0.75
Hours To Correct:	2
Hours per Fault:	0.92

PROGRAM 2:

Verification Method:	Reading
Percent Faults Found:	100
Percent Estimated Faults:	90
Hours To Detect:	0.75
Hours To Correct:	0.5
Hours per Fault:	0.18

PROGRAM 3:

Verification Method:	Functional
Percent Faults Found:	25
Percent Estimated Faults:	75
Hours To Detect:	2
Hours To Correct:	2
Hours per Fault:	1.33

SUBJECT 37

EXPERIENCE LEVEL: Junior

COMPUTER: VAX

PROGRAM 1:

Verification Method:	Functional
Percent Faults Found:	44
Percent Estimated Faults:	80
Hours To Detect:	2
Hours To Correct:	2
Hours per Fault:	1

PROGRAM 2:

Verification Method:	Reading
Percent Faults Found:	86
Percent Estimated Faults:	95
Hours To Detect:	0.75
Hours To Correct:	0.5
Hours per Fault:	0.21

PROGRAM 3:

Verification Method:	Structural
Percent Faults Found:	33
Percent Estimated Faults:	90
Hours To Detect:	6
Hours To Correct:	1.5
Hours per Fault:	1.88

SUBJECT 38

EXPERIENCE LEVEL: Advanced

COMPUTER: IBM

PROGRAM 1:

Verification Method:	Reading
Percent Faults Found:	67
Percent Estimated Faults:	90
Hours To Detect:	6.75
Hours To Correct:	1.5
Hours per Fault:	1.38

PROGRAM 2:

Verification Method:	Functional
Percent Faults Found:	57
Percent Estimated Faults:	90
Hours To Detect:	2.75
Hours To Correct:	1.5
Hours per Fault:	1.06

PROGRAM 3:

Verification Method:	Structural
Percent Faults Found:	17
Percent Estimated Faults:	60
Hours To Detect:	5
Hours To Correct:	1
Hours per Fault:	3

SUBJECT 40

EXPERIENCE LEVEL: Intermediate

COMPUTER: IBM

PROGRAM 1:

Verification Method:	Functional
Percent Faults Found:	44
Percent Estimated Faults:	70
Hours To Detect:	1.5
Hours To Correct:	1.5
Hours per Fault:	0.75

PROGRAM 2:

Verification Method:	Structural
Percent Faults Found:	71
Percent Estimated Faults:	100
Hours To Detect:	2
Hours To Correct:	0.5
Hours per Fault:	0.5

PROGRAM 3:

Verification Method:	Reading
Percent Faults Found:	33
Percent Estimated Faults:	51
Hours To Detect:	3
Hours To Correct:	1
Hours per Fault:	1

SUBJECT 41

EXPERIENCE LEVEL: Junior

COMPUTER: VAX

PROGRAM 1:

Verification Method: Reading
Percent Faults Found: 56
Percent Estimated Faults: 100
Hours To Detect: 3.5
Hours To Correct: 1
Hours per Fault: 0.9

PROGRAM 2:

Verification Method: Functional
Percent Faults Found: 57
Percent Estimated Faults: 60
Hours To Detect: 2.25
Hours To Correct: 1.5
Hours per Fault: 0.94

PROGRAM 3:

Verification Method: Structural
Percent Faults Found: 25
Percent Estimated Faults: 80
Hours To Detect: 2.75
Hours To Correct: 1.5
Hours per Fault: 1.42

SUBJECT 42

EXPERIENCE LEVEL: Advanced

COMPUTER: VAX

PROGRAM 1:

Verification Method:	Structural
Percent Faults Found:	44
Percent Estimated Faults:	90
Hours To Detect:	2
Hours To Correct:	0.5
Hours per Fault:	0.63

PROGRAM 2:

Verification Method:	Functional
Percent Faults Found:	57
Percent Estimated Faults:	100
Hours To Detect:	3.5
Hours To Correct:	1.5
Hours per Fault:	1.25

PROGRAM 3:

Verification Method:	Reading
Percent Faults Found:	33
Percent Estimated Faults:	50
Hours To Detect:	2.5
Hours To Correct:	0.5
Hours per Fault:	0.75

SUBJECT 43

EXPERIENCE LEVEL: Intermediate

COMPUTER: IBM

PROGRAM 1:

Verification Method: Functional

Percent Faults Found: 44

Percent Estimated Faults: 85

Hours To Detect: 1.5

Hours To Correct: 3

Hours per Fault: 1.13

PROGRAM 2:

Verification Method: Structural

Percent Faults Found: 14

Percent Estimated Faults: 90

Hours To Detect: 0.75

Hours To Correct: 0.75

Hours per Fault: 1.5

PROGRAM 3:

Verification Method: Reading

Percent Faults Found: 25

Percent Estimated Faults: 60

Hours To Detect: 3

Hours To Correct: 0.5

Hours per Fault: 1.17

SUBJECT 45

EXPERIENCE LEVEL: Advanced

COMPUTER: VAX

PROGRAM 1:

Verification Method: Functional
Percent Faults Found: 44
Percent Estimated Faults: 90
Hours To Detect: 1.5
Hours To Correct: 2.25
Hours per Fault: 0.94

PROGRAM 2:

Verification Method: Structural
Percent Faults Found: 86
Percent Estimated Faults: 100
Hours To Detect: 1.5
Hours To Correct: 0.5
Hours per Fault: 0.33

PROGRAM 3:

Verification Method: Reading
Percent Faults Found: 42
Percent Estimated Faults: 80
Hours To Detect: 3.25
Hours To Correct: 0.75
Hours per Fault: 0.8

SUBJECT 47

EXPERIENCE LEVEL: Advanced

COMPUTER: VAX

PROGRAM 1:

Verification Method:	Structural
Percent Faults Found:	44
Percent Estimated Faults:	83
Hours To Detect:	1.25
Hours To Correct:	2.5
Hours per Fault:	0.94

PROGRAM 2:

Verification Method:	Reading
Percent Faults Found:	100
Percent Estimated Faults:	100
Hours To Detect:	1.5
Hours To Correct:	1.25
Hours per Fault:	0.39

PROGRAM 3:

Verification Method:	Functional
Percent Faults Found:	50
Percent Estimated Faults:	70
Hours To Detect:	7.25
Hours To Correct:	3.5
Hours per Fault:	1.79

SUBJECT 48

EXPERIENCE LEVEL: Advanced

COMPUTER: IBM

PROGRAM 1:

Verification Method: Structural
Percent Faults Found: 33
Percent Estimated Faults: 50
Hours To Detect: 1.25
Hours To Correct: 0.25
Hours per Fault: 0.5

PROGRAM 2:

Verification Method: Reading
Percent Faults Found: 100
Percent Estimated Faults: 90
Hours To Detect: 1
Hours To Correct: 0.25
Hours per Fault: 0.18

PROGRAM 3:

Verification Method: Functional
Percent Faults Found: 33
Percent Estimated Faults: 80
Hours To Detect: 3.75
Hours To Correct: 2.25
Hours per Fault: 1.56

SUBJECT 50

EXPERIENCE LEVEL: Junior

COMPUTER: IBM

PROGRAM 1:

Verification Method:	Structural
Percent Faults Found:	11
Percent Estimated Faults:	100
Hours To Detect:	2
Hours To Correct:	1
Hours per Fault:	3

PROGRAM 2:

Verification Method:	Reading
Percent Faults Found:	100
Percent Estimated Faults:	100
Hours To Detect:	0.5
Hours To Correct:	2
Hours per Fault:	0.36

PROGRAM 3:

Verification Method:	Functional
Percent Faults Found:	17
Percent Estimated Faults:	80
Hours To Detect:	3
Hours To Correct:	2
Hours per Fault:	2.5

APPENDIX C - FAULTS FOUND

Figures C-1 through C-3 show the distribution of faults found by subjects during the experiment for the three test programs. The subject identification number and verification technique applied are also indicated in the figures.

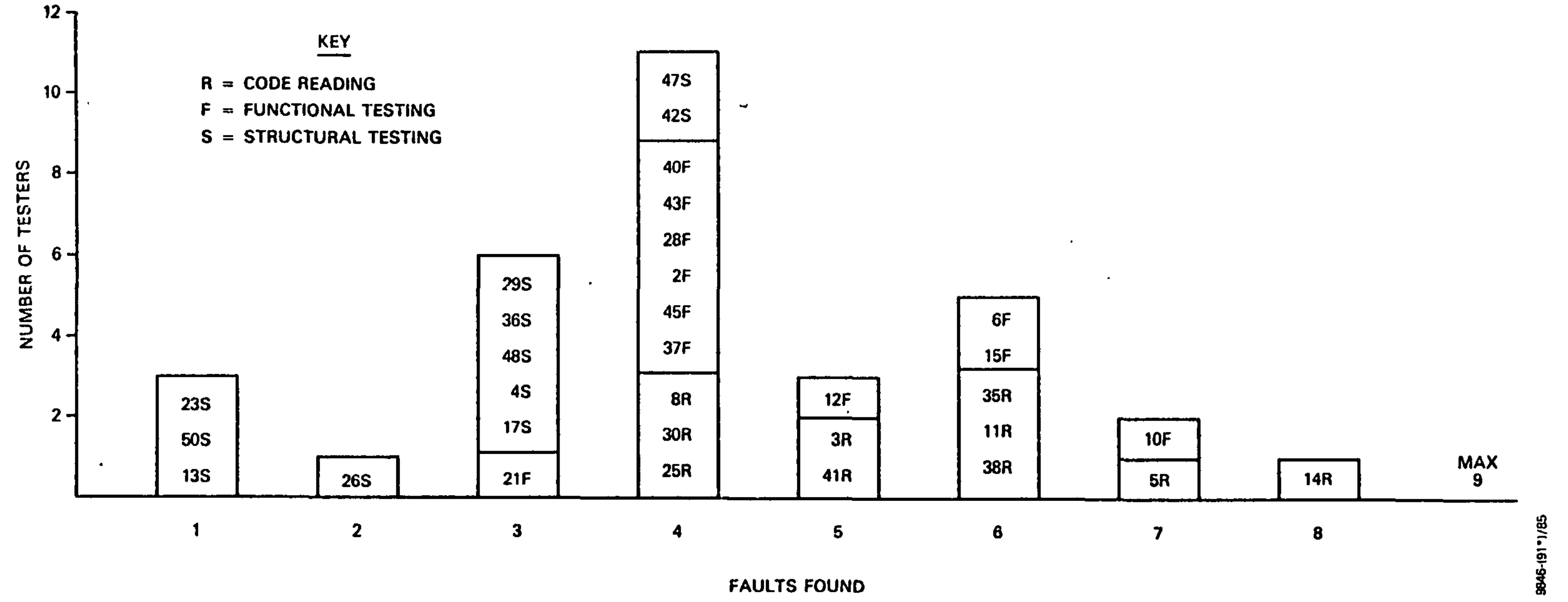


Figure C-1. Results of Testing Program 1

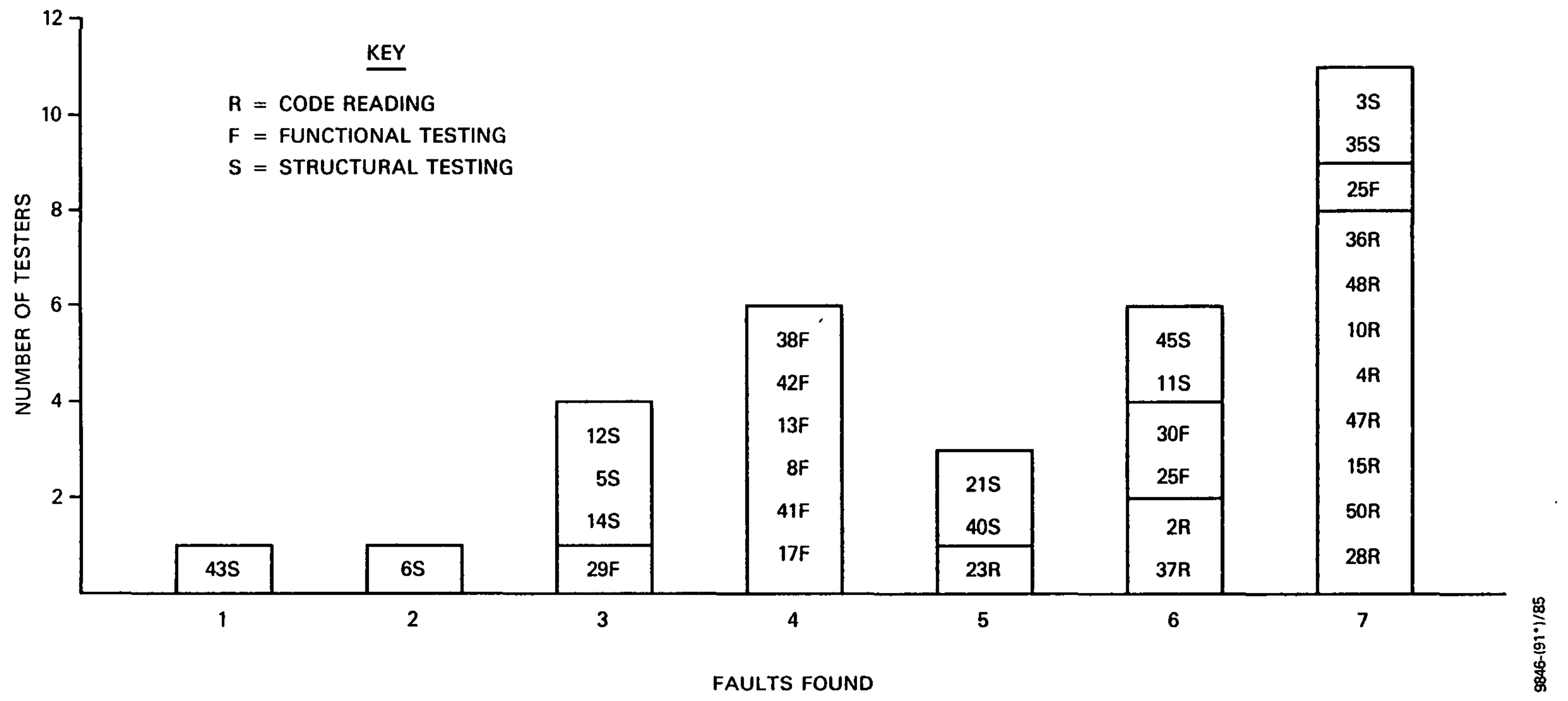


Figure C-2. Results of Testing Program 2

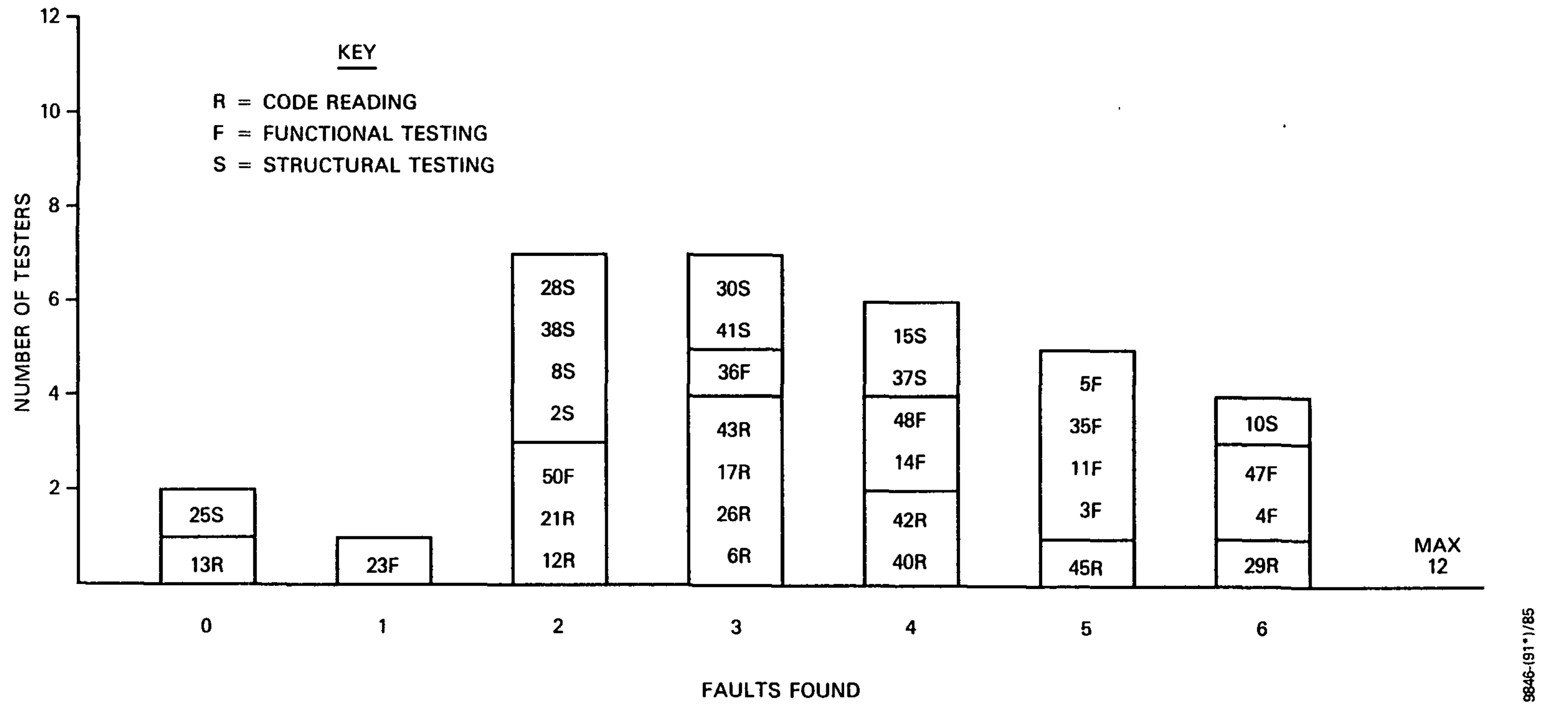


Figure C-3. Results of Testing Program 3

REFERENCES

1. M. S. Deutsch, "Verification and Validation," Software Engineering. New Jersey: Prentice Hall, Inc., 1979

2. M. Dyer and H. D. Mills, "Cleanroom Software Development," Proceedings of the Sixth Annual Software Engineering Workshop, December 1981

3. B. Beizer, Software System Testing and Quality Assurance. New York: Van Nostrand Reinhold, 1984

4. Linger, Mills, and Witt, "Reading Structured Programs," Structured Programming: Theory and Practice. New York: Addison-Wesley, 1979

5. G. J. Myers, "Test-Case Design," The Art of Software Testing. New York: John Wiley & Sons, 1979

6. B. Beizer, Software Testing Techniques. New York: Van Nostrand Reinhold, 1983

7. R. W. Selby, V. R. Basili, G. T. Page, and F. E. McGarry, "Evaluating Software Testing Strategies," Proceedings of the Ninth Annual Software Engineering Workshop, November 1984

8. R. W. Selby, "A Quantitative Approach for Evaluating Software Technologies," University of Maryland, Ph.D. Thesis, December 1984

9. Software Engineering Laboratory, SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

10. G. E. Box, W. G. Hunter, and J. S. Hunter, Statistics for Experimenters. New York: John Wiley & Sons, 1978

11. S. V. Hwang, An Empirical Study in Functional Testing, Structural Testing, and Code Reading/Inspection, University of Maryland, Scholarly Paper 362, December 1981

12. R. W. Selby, "An Empirical Study Comparing Software Testing Techniques," Presented at the Sixth Minnowbrook Workshop on Software Performance Evaluation, July 1983

13. D. N. Card, F. E. McGarry, and G. T. Page," Evaluating Software Engineering Technologies," Proceedings of the Eighth Annual Software Engineering Workshop, November 1983

14. G. J. Myers, "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," Communications of the ACM, September 1978

15. W. C. Hetzel, "An Experimental Analysis of Program Verification Methods," University of North Carolina, Ph.D. Thesis, 1976

16. E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," IEEE Transactions on Software Engineering, September 1984

17. M. S. Deutsch, "Verification and Validation," Software Engineering. New Jersey; Prentice-Hall, Inc., 1979

18. G. T. Page, F. E. McGarry, and D. N. Card, "A Practical Experience with Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

19. Computer Sciences Corporation, CSC/TM-78/6296, Acceptance Test Methods, J. Niblack, October 1978

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-001, The Software Engineering Laboratory, V. R. Basili, M. V. Zelkowitz, F. E. McGarry, et al., May 1977

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-003, Structured FORTRAN Preprocessor (SFORT), B. Chu and D. S. Wilson, September 1977

SEL-77-004, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-001, FORTRAN Static Source Code Analyzer (SAP) Design and Module Descriptions, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, February 1978

SEL-78-003, Evaluation of Draper NAVPAK Software Design, K. Tasaki and F. E. McGarry, June 1978

SEL-78-004, Structured FORTRAN Preprocessor (SFORT) PDP-11/70 User's Guide, D. S. Wilson and B. Chu, September 1978

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-78-202, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 2), W. J. Decker and W. A. Taylor, April 1985

SEL-79-001, SIMPL-D Data Base Reference Manual, M. V. Zelkowitz, July 1979

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-001, Functional Requirements/Specifications for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, A. L. Green, and C. E. Goorevich, February 1980

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

SEL-80-104, Configuration Analysis Tool (CAT) System Description and User's Guide (Revision 1), W. Decker and W. Taylor, December 1982

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, The Rayleigh Curve As a Model for Effort Distribution Over the Life of Medium Scale Software Systems, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-102, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide Revision 1, P. Lo and D. Wyckoff, July 1983

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-106, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide (Revision 1), W. Taylor and W. J. Decker, May 1985

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page and F. McGarry, December 1983

SEL-81-203, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, P. Lo, June 1984

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-003, Software Engineering Laboratory (SEL) Data Base Reporting Software User's Guide and System Description, P. Lo, September 1982

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1), W. A. Taylor and W. J. Decker, April 1982

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-206, Annotated Bibliography of Software Engineering Laboratory Literature, D. N. Card, Q. L. Jordan, and F. E. McGarry, November 1984

SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-83-104, Software Engineering Laboratory (SEL) Data Base Retrieval System (DARES) User's Guide, T. A. Babst, W. J. Decker, P. Lo, and W. Miller, August 1984

SEL-83-105, Software Engineering Laboratory (SEL) Data Base Retrieval System (DARES) System Description, P. Lo, W. J. Decker, and W. Miller, August 1984

SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, V. E. Church, and F. E. McGarry, April 1984

SEL-84-002, Configuration Management and Control: Policies and Procedures, Q. L. Jordan and E. Edwards, December 1984

SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory (SEL), W. Agresti, V. Church, and F. E. McGarry, December 1984

SEL-85-001, A Comparison of Software Verification Techniques, D. Card, R. Selby, F. McGarry, et al., April 1985

SEL-RELATED LITERATURE

Agresti, W. W., Definition of Specification Measures for the Software Engineering Laboratory, Computer Sciences Corporation, CSC/TM-84/6085, June 1984

[1]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

[2]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

[2]Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

Basili, V. R., "SEL Relationships for Programming Measurement and Estimation," University of Maryland, Technical Memorandum, October 1979

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: Computer Societies Press, 1980 (also designated SEL-80-008)

[2]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?", Journal of Systems and Software, February 1981, vol. 2, no. 1

[2]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

[1]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984, vol. 27, no. 1

[2]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/ Workshop: Quality Metrics, March 1981

[1]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983

Basili, V. R., and J. Ramsey, Structural Coverage of Functional Testing, University of Maryland, Technical Report TR-1442, September 1984

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity and Cost, October 1979

[1]Basili, V.R., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235, December 1982

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

[2]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[2]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: Computer Societies Press, 1978

[2]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

[2]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

[1]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," Proceedings of the Seventh International Computer Software and Applications Conference. New York: Computer Societies Press, 1983

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

[2]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York: Computer Societies Press, 1979

[1]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," Empirical Foundations for Computer and Information Science (proceedings), November 1982

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

[1]This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[2]This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.