Comparing the Effectiveness of Software Testing Strategies

VICTOR R. BASILI, SENIOR MEMBER, IEEE, AND RICHARD W. SELBY, MEMBER, IEEE

Abstract—This study applies an experimentation methodology to compare three state-of-the-practice software testing techniques: a) code reading by stepwise abstraction, b) functional testing using equivalence partitioning and boundary value analysis, and c) structural testing using 100 percent statement coverage criteria. The study compares the strategies in three aspects of software testing: fault detection effectiveness, fault detection cost, and classes of faults detected. Thirty-two professional programmers and 42 advanced students applied the three techniques to four unit-sized programs in a fractional factorial experimental design. The major results of this study are the following. 1) With the professional programmers, code reading detected more software faults and had a higher fault detection rate than did functional or structural testing, while functional testing detected more faults than did structural testing, but functional and structural testing were not different in fault detection rate. 2) In one advanced student subject group, code reading and functional testing were not different in faults found, but were both superior to structural testing, while in the other advanced student subject group there was no difference among the techniques. 3) With the advanced student subjects, the three techniques were not different in fault detection rate. 4) Number of faults observed, fault detection rate, and total effort in detection depended on the type of software tested. 5) Code reading detected more interface faults than did the other methods. 6) Functional testing detected more control faults than did the other methods. 7) When asked to estimate the percentage of faults detected, code readers gave the most accurate estimates while functional testers gave the least accurate estimates.

Index Terms—Code reading, empirical study, functional testing, methodology evaluation, off-line software review, software measurement, software testing, structural testing.

I. Introduction

THE processes of software testing and fault detection continue to challenge the software community. Even though the software testing and fault detection activities are inexact and inadequately understood, they are crucial to the success of a software project. This paper presents a controlled study where an experimentation methodology was applied to address the uncertainty of how to test software effectively. In this investigation, common testing techniques were applied to different types of software by subjects that had a wide range of professional experience. This controlled study is intended to evaluate different testing methods that are actually used by software developers, "state-of-the-practice" methods, as opposed to state-of-the-art techniques.

This work is intended to characterize how testing effectiveness relates to several factors: testing technique, software type, fault type, tester experience, and any interactions among these factors. The study presented extends previous work by incorporating different testing techniques and a greater number of persons and programs, while broadening the scope of issues examined and adding statistical significance to the conclusions.

There are multiple perspectives from which to view empirical studies of software development techniques, including the study presented in this paper.

- *Experimenter*—An experimenter may view the study as a demonstration of how a software development technique (or methodology, tool, etc.) can be empirically evaluated. Experimenters may examine the work as an example application of a particular experimentation methodology that may be reused in future studies.
- *Researcher*—A researcher may view the study as an empirical basis to refine theories of software testing. Researchers formulate software testing theories that have a horizon across multiple studies. As a consequence, they examine data from a variety of sources and focus on data that either support or refute proposed theories.
- *Practitioner*—A practitioner may view the study as a source of information about which approaches to testing should be applied in practice. Practitioners may focus on the particular quantifications and comparisons provided by the results. They then consider the relationship of the programs and programmers examined to the particular environment or projects in which the results might be applied.

The following sections describe the testing techniques examined, the investigation goals, the experimental design, operation, analysis, and conclusions.

II. Testing Techniques

To demonstrate that a particular program actually meets its specifications, professional software developers currently utilize many different testing methods. The controlled study presented analyzes three common software testing techniques, which will be referred to as functional testing, structural testing, and code reading. Before pre-

Manuscript received May 31, 1985; revised June 30, 1986. This work was supported in part by the Air Force Office of Scientific Research under Contract AFOSR-F49620-80-C-001, the National Aeronautics and Space Administration under Grant NSG-5123, and the University of California Faculty Research Fellowship Program. Computer support was provided in part by the NASA/Goddard Space Flight Center, the Computer Science Center at the University of Maryland, and the University of California.

V. R. Basili is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

R. W. Selby is with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

IEEE Log Number 8717407.

senting the goals for the empirical study comparing the techniques, a description will be given of the testing strategies and their different capabilities (see Fig. 1.). In functional testing, which is a "black box" approach, a programmer constructs test data from the program's specification through methods such as equivalence partitioning and boundary value analysis [42]. The programmer then executes the program and contrasts its actual behavior with that indicated in the specification. In structural testing, which is a "white box" approach [25], [29], a programmer inspects the source code and then devises and executes test cases based on the percentage of the program's statements or expressions executed (the "test set coverage") [52]. The structural coverage criteria used was 100 percent statement coverage. In code reading by stepwise abstraction, a person identifies prime subprograms in the software, determines their functions, and composes these functions to determine a function for the entire program [35], [39]. The code reader then compares this derived function and the specifications (the intended function).

The controlled study presented analyzes, therefore, 1) the functional testing technique of using equivalence class partitioning and boundary value analysis, 2) the structural testing technique of using 100 percent statement coverage criteria, and 3) the code reading technique of reading by stepwise abstraction. Certainly more advanced methods of testing software have been proposed (for example, see [10]). The intention of the controlled study, however, is to apply an experimentation methodology to analyze testing methods that are actually being used by developers to test software [56]. Note that alternate forms exist for each of the three methods described, for example, functional testing that takes into consideration the program design [27], structural testing that uses branch or data flow criteria [16], and code reading in multiperson inspections [14]. With the above descriptions in mind, we will refer to the three testing methods as functional testing, structural testing, and code reading.

A. Investigation Goals

The goals of this study comprise three different aspects of software testing: fault detection effectiveness, fault detection cost, and classes of faults detected. An application of the goal/question/metric paradigm [2], [6] leads to the framework of goals and questions for this study appearing in Fig. 2.

The first goal area is performance oriented and includes a natural first question (I-A): which of the techniques detects the most faults in the programs? The comparison between the techniques is being made across programs, each with a different number of faults. An alternate interpretation would then be to compare the percentage of faults found in the programs (question I-A-1). The number of faults that a technique exposes should also be compared; that is, faults that are made observable but not necessarily observed and reported by a tester (I-A-2). Because of the differences in types of software and in testers' abilities, it is relevant to determine whether the number of the faults detected is either program or programmer dependent (I-B, I-C). Since one technique may find a few more faults than another, it becomes useful to know how much effort that technique requires (II-A). Awareness of what types of software require more effort to test (II-B) and what types of programmer backgrounds require less effort in fault uncovering (II-C) is also quite useful. If one is interested in detecting certain classes of faults, such as in error-based testing [15], [53] it is appropriate to apply a technique sensitive to that particular type (III-A). Classifying the types of faults that are observable yet go unreported could help focus and increase testing effectiveness (III-B).

	code reading	functional testing	structural testing
view program specification	X	X	X
view source code	X		X
execute program		X	X

Fig. 1. Capabilities of the testing methods.

I. Fault detection effectiveness

A. For programmers doing unit testing, which of the testing techniques (code reading, functional testing, or structural testing) detects the most faults in programs?

1. Which of the techniques detects the greatest percentage of faults in the programs (the programs each contain a different number of faults)?

2. Which of the techniques exposes the greatest number (or percentage) of program faults (faults that are observable but not necessarily reported)?

B. Is the number of faults observed dependent on software type?

C. Is the number of faults observed dependent on the expertise level of the person testing?

II. Fault detection cost

A. For programmers doing unit testing, which of the testing techniques (code reading, functional testing, or structural testing) detects the faults at the highest rate (#faults/effort)?

B. Is the fault detection rate dependent on software type?

C. Is the fault detection rate dependent on the expertise level of the person testing?

III. Classes of faults observed

A. For programmers doing unit testing, do the methods tend to capture different classes of faults?

B. What classes of faults are observable but go unreported?

Fig. 2. Outline of goals/subgoals/questions for testing experiment.

III. Empirical Study

Admittedly, the goals stated here are quite ambitious. In no way is it implied that this study can definitively answer all of these questions for all environments. It is intended, however, that the statistically significant analysis presented lends insights into their answers and into the merit and appropriateness of each of the techniques. Note

that this study compares the individual application of the three testing techniques in order to identify their distinct advantages and disadvantages. This approach is a first step toward proposing a composite testing strategy, which possibly incorporates several testing methods. The following sections describe the empirical study undertaken to pursue these goals and questions, including the selection of subjects, programs, and experimental design, and the overall operation of the study. For an overview of the experimentation methodology applied in this study, as well as a discussion of numerous software engineering experiments, see [4].

A. Iterative Experimentation

The empirical study consisted of three phases. The first and second phases of the study took place at the University of Maryland in the Falls of 1982 and 1983, respectively. The third phase took place at Computer Sciences Corporation (Silver Spring, MD) and NASA Goddard Space Flight Center (Greenbelt, MD) in the Fall of 1984. The sequential experimentation supported the iterative nature of the learning process, and enabled the initial set of goals and questions to the expanded and resolved by further analysis. The goals were further refined by discussions of the preliminary results [47], [51]. These three phases enabled the pursuit of result reproducibility across environments having subjects with a wide range of experience.

B. Subject and Program/Fault Selection

A primary consideration in this study was to use a realistic testing environment to assess the effectiveness of these different testing strategies, as opposed to creating a best possible testing situation [23]. Thus, 1) the subjects for the study were chosen to be representative of different levels of expertise, 2) the programs tested correspond to different types of software and reflect common programming style, and 3) the faults in the programs were representative of those frequently occurring in software. Sampling the subjects, programs, and faults in this manner is intended to evaluate the testing methods reasonably, and to facilitate the generalization of the results to other environments.

1) Subjects: The three phases of the study incorporated a total of 74 subjects; the individual phases had 29, 13, and 32 subjects, respectively. The subjects were selected, based on several criteria, to be representative of three different levels of computer science expertise: advanced, intermediate, and junior. The number of subjects in each level of expertise for the different phases appears in Fig. 3.

	Phase			
Level of Expertise	1 (Univ. Md)	2 (Univ. Md)	3 (NASA/CSC)	total
Advanced	0	0	8	8
Intermediate	9	4	11	24
Junior	20	9	13	42
total	29	13	32	74

Fig. 3. Expertise levels of subjects.

The 42 subjects in the first two phases of the study were the members of the upper level "Software Design and Development" course at the University of Maryland in the Falls of 1982 and 1983. The individuals were either upper-level computer science majors or graduate students; some were working part-time and all were in good academic standing. The topics of the course included structured programming practices, functional correctness, top-down design, modular specification and design, step-wise refinement, and PDL, in addition to the presentation of the techniques of code reading, functional testing, and structural testing. The references for the testing methods were [40], [14], [42], [27], and the lectures were presented by V. R. Basili and F. T. Baker. The subjects from the University of Maryland spanned the intermediate and junior levels of computer science expertise. The assignment of individuals to levels of expertise was based on professional experience and prior academic performance in relevant computer science courses. The individuals in the first and second phases had overall averages of 1.7 (SD = 1.7) and 1.5 (SD = 1.5) years of professional experience. The nine intermediate subjects in the first phase had from 2.8 to 7 years of professional experience (average of 3.9 years, SD = 1.3), and the four in the second phase had from 2.3 to 5.5 years of professional experience (average of 3.2, SD = 1.5). The 20 junior subjects in the first phases and the nine in the second phase both had from 0 to 2 years professional experience (averages of 0.7, SD = 0.6, and 0.8, SD = 0.8, respectively).

The 32 subjects in the third phase of the study were programming professionals from NASA and Computer Sciences Corporation. These individuals were mathematicians, physicists, and engineers that develop ground support software for satellites. They were familiar with all three testing techniques, but had used functional testing primarily. A four hour tutorial on the testing techniques was conducted for the subjects by R. W. Selby. This group of subjects, examined in the third phase of the experiment, spanned all three expertise levels and had an overall average of 10.0 (SD = 5.7) years professional experience. Several criteria were considered in the assignment of subjects to expertise levels, including years of professional experience, degree background, and their manager's suggested assignment. The eight advanced subjects ranged from 9.5 to 20.5 years professional experience (average of 15.0, SD = 4.1). The eleven intermediate subjects ranged from 3.5 to 17.5 years experience (average of 10.9, SD = 4.9). The 13 junior subjects ranged from 1.5 to 13.5 years experience (average of 6.1, SD = 4.4).

2) Programs: The experimental design enables the distinction of the testing techniques while allowing for the effects of the different programs being tested. The four programs used in the investigation were chosen to be representative of several different types of software. The programs were selected specially for the study and were provided to the subjects for testing; the subjects did not test

programs that they had written. All programs were written in a high-level language with which the subjects were familiar. The three programs tested in the CSC/NASA phase were written in Fortran, and the programs tested in the University of Maryland phase were written in the Simpl-T structured programming language [5].[1] The four programs tested were P_1) a text processor, P_2) a mathematical plotting routine, P_3) a numeric abstract data type, and P_4) a database maintainer. The programs are summarized in Fig. 4. There exists some differentiation in size, and the programs are a realistic size for unit testing. Each of the subjects tested three programs, but a total of four programs was used across the three phases of the study. The programs tested in each of the three phases of the study appear in Fig. 5. The specifications for the programs appear in the Appendix, and their source code appears in [3], [48].

program	source lines	executable statements	cyclomatic complexity	#routines	#faults
P_1 - text formatter	169	55	18	3	9
P_2 - mathematical plotting	145	95	32	5	6
P_3 - numeric data abstraction	147	48	18	9	7
P_4 - database maintainer	365	144	57	7	12

Fig. 4. The programs tested.

Program	Phase		
	1 (Univ. Md.)	2 (Univ. Md.)	3 (NASA/CSC)
P_1 - text formatter	X	X	X
P_2 - mathematical plotting	X	X	
P_3 - numeric data abstraction	X		X
P_4 - database maintainer		X	X

Fig. 5. Programs tested in each phase of the analysis.

The first program is a text formatting program, which also appeared in [41]. A version of this program, originally written by [43] using techniques of program correctness proofs, was analyzed in [19]. The second program is a mathematical plotting routine. This program was written by R. W. Selby, based roughly on a sample program in [33]. The third program is a numeric data abstraction consisting of a set of list processing utilities. This program was submitted for a class project by a member of an intermediate level programming course at the University of Maryland [36]. The fourth program is a maintainer for a database of bibliographic references. This program was analyzed in [23], and was written by a systems programmer at the University of North Carolina computation center.

Note that the source code for the programs contains no comments. This creates a worst-case situation for the code readers. In an environment where code contained helpful comments, performance of code readers would likely improve, especially if the source code contained as comments the intermediate functions of the program segments. In an environment where the comments were at all suspect, they could then be ignored.

3) Faults: The faults contained in the programs tested represent a reasonable distribution of faults that commonly occur in software [1], [54]. All the faults in the database maintainer and the numeric abstract data type were made during the actual development of the programs. The other two programs contain a mix of faults made by the original programmer and faults seeded in the code. The programs contained a total of 34 faults: the text formatter had nine, the plotting routine had six, the abstract data type had seven, and the database maintainer had twelve.

a) Fault Origin: The faults in the text formatter were preserved from the article in which it appeared [41], except for some of the more controversial ones [9]. In the mathematical plotter, faults made during program translation were supplemented by additional representative faults. The faults in the abstract data type were the original ones made by the program's author during the development of the program. The faults in the database maintainer were recorded during the development of the program, and then reinserted into the program. The next section describes a classification of the different types of faults in the programs. Note that this investigation of the fault detecting ability of these techniques involves only those types occurring in the source code, not other types such as those in the requirements or the specifications.

b) Fault Classification: The faults in the programs are classified according to two different abstract classification schemes [1]. One fault categorization method separates faults of omission from faults of commission. Faults of commission are those faults present as a result of an incorrect segment of existing code. For example, the wrong arithmetic operator is used for a computation in the right-hand-side of an assignment statement. Faults of omission are those faults present as a result of a programmer's forgetting to include some entity in a module. For example, a statement is missing from the code that would assign the proper value to a variable.

A second fault categorization scheme partitions software faults into the six classes of 1) initialization, 2) computation, 3) control, 4) interface, 5) data, and 6) cosmetic. Improperly initializing a data structure constitutes an initialization fault. For example, assigning a variable the wrong value on entry to a module. Computation faults are those that cause a calculation to evaluate the value for a variable incorrectly. The above example of a wrong arithmetic operator in the right-hand-side of an assignment statement would be a computation fault. A control fault causes the wrong control flow path in a program to be taken for some input. An incorrect predicate in an IF-THEN-ELSE statement would be a control fault. Interface faults result when a module uses and makes assumptions about entities outside the module's local environment. Interface faults would be, for example, passing an

[1] Simpl-T is a structured language that supports several string and file handling primitives, in addition to the usual control flow constructs available, for example, in Pascal.

incorrect argument to a procedure, or assuming in a module that an array passed as an argument was filled with blanks by the passing routine. A data fault are those that result from the incorrect use of a data structure. For example, incorrectly determining the index for the last element in an array. Finally, cosmetic faults are clerical mistakes when entering the program. A spelling mistake in an error message would be a cosmetic fault.

Interpreting and classifying faults in software is a difficult and inexact task. The categorization process often requires trying to recreate the original programmer's misunderstanding of the problem [34]. The above two fault classification schemes attempt to distinguish among different reasons that programmers make faults in software development. They were applied to the faults in the programs in a consistent interpretation; it is certainly possible that another analyst could have interpreted them differently. The separate application of each of the two classification schemes to the faults categorized them in a mutually exclusive and exhaustive manner. Fig. 6 displays the distribution of faults in the programs according to these schemes.

c) Fault Description: The faults in the programs are described in Fig. 7. There have been various efforts to determine a precise counting scheme for "defects" in software [18], [31], [13]. According to the IEEE explanations given, a software "fault" is a specific manifestation in the source code of a programmer "error." For example, due to a misconception or document discrepancy, a programmer makes an "error" (in his/her head) that may result in more than one "fault" in a program. Using this interpretation, software "faults" reflect the correctness, or lack thereof, of a program. A program input may reveal a software "fault" by causing a software "failure." A software "failure" is therefore a manifestation of a software "fault." The entities examined in this analysis are software faults.

C. Experimental Design

The experimental design applied for each of the three phases of the study was a fractional factorial design [7], [12]. This experimental design distinguishes among the testing techniques, while allowing for variation in the ability of the particular individual testing or in the program being tested. Fig. 8 displays the fractional factorial design appropriate for the third phase of the study. Subject S_1 is in the advanced expertise level, and he structurally tested program P_1, functionally tested program P_3, and code read program P_4. Notice that all of the subjects tested each of the three programs and used each of the three techniques. Of course, no one tests a given program more than once. The design appropriate for the third phase is discussed in the following paragraphs, with the minor differences between this design and the ones applied in the first two phases being discussed at the end of the section.

1) Independent and Dependent Variables: The experimental design has the three independent variables of testing technique, software type, and level of expertise. For the design appearing in Fig. 8, appropriate for the third phase of the study, the three main effects have the following levels:

	Omission	Commission	Total
Initialization	0	2	2
Computation	4	4	8
Control	2	5	7
Interface	2	11	13
Data	2	1	3
Cosmetic	0	1	1
Total	10	24	34

Fig. 6. Distribution of faults in the programs.

1) testing technique: code reading, functional testing, and structural testing.

2) software types: (P_1) text processing, (P_3) numeric abstract data type, and (P_4) database maintainer.

3) level of expertise: advanced, intermediate, and junior.

Every combination of these levels occurs in the design. That is, programmers in all three levels of expertise applied all three testing techniques on all programs. In addition to these three main effects, a factorial analysis of variance (ANOVA) model supports the analysis of interactions among each of these main effects. Thus, the interaction effects of testing technique * software type, testing technique * expertise level, software type * expertise level, and the three-way interaction of testing technique * software type * expertise level are included in the model. There are several dependent variables examined in the study, including number of faults detected, percentage of faults detected, total fault detection time, and fault detection rate. Observations from the on-line methods of functional and structural testing also had as dependent variables number of computer runs, amount of cpu-time consumed, maximum statement coverage achieved, connect time used, number of faults that were observable from the test data, percentage of faults that were observable from the test data, and percentage of faults observable from the test data that were actually observed by the tester.

2) Analysis of Variance Model: The three main effects and all the two-way and three-way interactions effects are called fixed effects in this factorial analysis of variance model. The levels of these effects given above represent all levels of interest in the investigation. For example, the effect of testing technique has as particular levels code reading, functional testing, and structural testing; these particular testing techniques are the only ones under comparison in this study. The effect of the particular subjects that participated in this study requires a little different interpretation. The subjects examined in the study were random samples of programmers from the large population of programmers at each of the levels of expertise. Thus, the effect of the subjects on the various dependent variables is a random variable, and this effect therefore is called a random effect. If the samples examined are truly representative of the population of subjects at each ex-

Fault	Program	Omission/ Commission	Class	Description
a	P1	omission	control	a blank is printed before the first word on the first line unless the first word is 30 characters long; in the latter case, a blank line is printed before the first word
b	P1	commission	initialization	the character & (not $) is the new-line character
c	P1	commission	initialization	the line size is 31 characters (not 30); this fault causes the references to the number 30 in the other faults to be actually the number 31
d	P1	commission	interface	since the program pads an empty input buffer with the character "z," it ignores a valid input line that has a "z" as a first character
e	P1	omission	control	successive break characters are not condensed in the output
f	P1	commission	cosmetic	spelling mistake in the error message "••• word to long •••"
g	P1	commission	computation	after detecting a word in the input longer than 30 characters, the message "••• word to long •••" is printed once for every character over 30, and the processing of the text does not terminate
h	P1	omission	interface	after detecting a word in the input longer than 30 characters, the program prints whatever is residing in its output buffer
i	P1	commission	control	after detecting an input line without an end-of-text character, the program erroneously increments its buffer pointer and replaces the first character of the next input line with a "z"
j	P3	commission	interface	routine FIRST returns zero (0) when the list has one element
k	P3	commission	interface	routine ISEMPTY returns true (1) when the list has one element
l	P3	commission	interface	routine DELETEFIRST can not delete the first list element when the list has only one element
m	P3	commission	interface	routine LISTLENGTH returns one less than than the actual length of the list
n	P3	commission	interface	routine ADDFIRST can add more than the specified five elements to the list
o	P3	commission	interface	routine ADDLAST can add more than the specified five elements to the list
p	P3	omission	computation	routine REVERSE does not reverse the list properly when the list has more than one element
q	P4	commission	computation	words greater than or equal to three characters (not strictly greater than) are treated as cross reference keywords
r	P4	commission	interface	since the program uses the key "ZZZ" as an end-of-input sentinel, it does not process a valid record with key "ZZZ" and ignores any following records
s	P4	commission	control	update action add with the error condition "key already in the master file" replaces the existing record; the update record is not ignored
t	P4	commission	control	update action replace with the error condition "k · not found in the master file" adds the record; the update record is not ignored
u	P4	omission	data	the number of references and number of words in the dictionary are not checked for overflow
v	P4	omission	computation	two or more update transactions for the same master record give incorrect results
w	P4	commission	interface	keywords longer than 12 characters are truncated and not distinguished
x	P4	commission	control	an update record with column 80 neither an add action "A" nor replace action "R" acts like an add transaction
y	P4	commission	interface	keyword indices appear in reverse alphabetical order
z	P4	omission	interface	no check is made for unique keys in the master file
A	P4	commission	interface	punctuation is made a part of the keyword
B	P4	omission	data	words appearing twice in a title get two cross reference entries
C	P2	commission	computation	the x and y axes are mislabeled
D	P2	omission	computation	points with negative y-values are not processed and do not appear on the graph
E	P2	commission	control	the origin (0,0) appears on the graph regardless of whether it is an input point
F	P2	commission	data	no points can appear on the vertical axis
G	P2	commission	computation	the vertical and horizontal scaling for the pixels are calculated incorrectly, causing some points not to appear in the proper pixel
H	P2	omission	computation	when more than one point would appear in a given pixel, only an asterisk (*) appears, not an appropriate integer

Fig. 7. Fault classification and description.

		Code Reading	Functional Testing	Structural Testing
		$P_1\ P_2\ P_3$	$P_1\ P_2\ P_3$	$P_1\ P_2\ P_3$
Advanced Subjects	S_1	——X	—X—	X——
	S_2	—X—	X——	——X
	.		...	
	.		...	
	S_9	X——	——X	—X—
Intermediate Subjects	S_9	—X—	X——	——X
	S_{10}	——X	—X—	X——
	.		...	
	.		...	
	S_{19}	X——	——X	—X—
Junior Subjects	S_{20}	—X—	X——	——X
	S_{21}	X——	——X	—X—
	.		...	
	.		...	
	S_{32}	——X	—X—	X——

Fig. 8. Fractional factorial design.

pertise level, the inferences from the analysis can then be generalized across the whole population of subjects at each expertise level, not just across the particular subjects in the sample chosen. Since this analysis of variance model contains both fixed and random effects, it is called a mixed model. The additive ANOVA model for the design appearing in Fig. 8 is given below [7], [12].

$$\gamma_{ijkl} = \mu + \alpha_i + \beta_j + \gamma_k + \delta_{kl} + \alpha\beta_{ij} + \alpha\gamma_{ik} + \beta\gamma_{jk} + \alpha\beta\gamma_{ijk} + \epsilon_{ijkl}$$

where

- γ_{ijkl} is the observed response from subject l of expertise level k using testing technique i on program j.
- μ is the overall mean response.
- α_i is the main effect of testing technique i ($i = 1, 2, 3$).
- β_j is the main effect of program j ($j = 1, 3, 4$).
- γ_k is the main effect of expertise level k ($k = 1, 2, 3$).
- δ_{kl} is the random effect of subject l within expertise level k, a random variable ($l = 1, 2, \cdots, 32$: $k = 1, 2, 3$).

$\alpha\beta_{ij}$ is the interaction effect of testing technique i with program j ($i = 1, 2, 3$; $j = 1, 3, 4$).

$\alpha\gamma_{ik}$ is the interaction effect of testing technique i with expertise level k ($i = 1, 2, 3$; $k = 1, 2, 3$).

$\beta\gamma_{jk}$ is the interaction effect of program j with expertise level k ($j = 1, 3, 4$; $k = 1, 2, 3$).

$\alpha\beta\gamma_{ijk}$ is the interaction effect of testing technique i program j with expertise level k ($i = 1, 2, 3$; $j = 1, 3, 4$; $k = 1, 2, 3$).

ϵ_{ijkl} is the experimental error for each observation, a random variable.

The tests of hypotheses on all the fixed effects mentioned above are referred to as F-tests [46]. The F-tests use the error (residual) mean square in the denominator, except for the test of the expertise level effect. The expected mean square for the expertise level effect contains a component for the actual variance of subjects within expertise level. In order to select the appropriate term for the denominator of the expertise level F-test, the mean square for the effect of subjects nested within expertise level is chosen. The parameters for the random effect of subjects within expertise level are assumed to be drawn from a normally distributed random process with mean zero and common variance. The experimental error terms are assumed to have mean zero and common variance.

The fractional factorial design applied in the first two phases of the analysis differed slightly from the one presented above for the third phase.[2] In the third phase of the study, programs P_1, P_3, and P_4 were tested by subjects in three levels of expertise. In both phases one and two, there were only subjects from the levels of intermediate and junior expertise. In phase one, programs P_1, P_3, and P_2 were tested. In phase two, the programs tested were P_1, P_2, and P_4. The only modifications necessary to the above explanation for phases one and two are 1) eliminating the advanced expertise level, 2) changing the program P subscripts appropriately, and 3) leaving out the three way interaction term in phase two, because of the reduced number of subjects. In all three of the phases, all subjects used each of the three techniques and tested each of the three programs for that phase. Also, within all three phrases, all possible combinations of expertise level, testing techniques, and programs occurred.

The order of presentation of the testing techniques was randomized among the subjects in each level of expertise in each phase of the study. However, the integrity of the results would have suffered if each of the programs in a given phase was tested at different times by different subjects. Note that each of the testing sessions took place on a different day because of the amount of effort required. If different programs would have been tested on different days, any discussion about the programs among subjects between testing sessions would have affected the future performance of others. Therefore, all subjects in a phase tested the same program on the same day. The actual order of program presentation was the order in which the programs are listed in the previous paragraph.

[2]Although the data from all the phases can be analyzed together, the number of empty cells resulting from not having all three experience levels and all four programs in all phases limits the number of parameters that can be estimated and causes nonunique Type IV partial sums of squares.

D. Experimental Operation

Each of the three phases were broken into five distinct pieces: training, three testing sessions, and a follow-up session. All groups of subjects were exposed to a similar amount of training on the testing techniques before the study began. As mentioned earlier, the University of Maryland subjects were enrolled in the "Software Design and Development" course, and the NASA/CSC subjects were given a four-hour tutorial. Background information on the subjects was captured through a questionnaire. Elementary exercises followed by a pretest covering all techniques were administered to all subjects after the training and before the testing sessions. Reasonable effort on the part of the University of Maryland subjects was enforced by their being graded on the work and by their needing to use the techniques in a major class project. Reasonable effort on the part of the NASA/CSC subjects was certain because of their desire for the study's outcome to improve their software testing environment. All subjects' groups were judged highly motivated during the study. The subjects were all familiar with the editors, terminals, machines, and the programs' implementation language.

The individuals were requested to use the three testing techniques to the best of their ability. Every subject participated in all three testing sessions of his/her phase, using all techniques but each on a separate program. The individuals using code reading were each given the specification for the program and its source code. They were then asked to apply the methods of code reading by stepwise abstraction to detect discrepancies between the program's abstracted function and the specification. The functional testers were each given a specification and the ability to execute the program. They were asked to perform equivalence partitioning and boundary value analysis to select a set of test data for the program. Then they executed the program on this collection of test data, and inconsistencies between what the program actually performed and what they thought the specification said it should perform were noted. The structural testers were given the source code for the program, the ability to execute it, and a description of the input format for the program. The structural testers were asked to examine the source and generate a set of test cases that cumulatively execute 100 percent of the program's statements. When the subjects were applying an on-line technique, they generated and executed their own test data; no test data sets were provided. The programs were invoked through a test driver that supported the use of the multiple input data sets. This test driver, unbeknown to the subjects, drained off the input cases submitted to the program for the ex-

perimenter's later analysis; the programs could only be accessed through a test driver.

A structural coverage tool calculated the actual statement coverage of the test set and which statements were left unexecuted for the structural testers.[3] After the structural testers generated a collection of test data that met (or almost met) the 100 percent coverage criteria, no further execution of the program or reference to the source code was allowed. They retained the program's output from the test cases they had generated. These testers were then provided with the program's specification. Now that they knew what the program was intended to do, they were asked to contrast the program's specification with the behavior of the program on the test data they derived. This scenario for the structural testers was necessary so that "observed" faults could be compared.

At the end of each of the testing sessions, the subjects were asked to give a reasonable estimate of the amount of time spent detecting faults with a given testing technique. The University of Maryland subjects were assured that this had nothing to with the grading of the work. There seemed to be little incentive for the subjects in any of the groups not to be truthful. At the completion of each testing session, the NASA/CSC subjects were also asked what percentage of the faults in the program that they thought were uncovered. After all three testing sessions in a given phase were completed, the subjects were requested to critique and evaluate the three testing techniques regarding their understandability, naturalness, and effectiveness. The University of Maryland subjects submitted a written critique, while a two hour debriefing forum was conducted for the NASA/CSC individuals. In addition to obtaining the impressions of the individuals, these follow-up procedures gave an understanding of how well the subjects were comprehending and applying the methods. These final sessions also afforded the participants an opportunity to comment on any particular problems they had with the techniques or in applying them to the given programs.

IV. Data Analysis

The analysis of the data collected from the various phases of the experiment is presented according to the goal and question framework discussed earlier.

A. Fault Detection Effectiveness

The first goal area addresses the fault detection effectiveness of each of the techniques. Fig. 9 presents a summary of the measures that were examined to pursue this goal area. A brief description of each measure is as follows; an asterisk (*) means only relevant for on-line testing.

a) Number of faults detected = the number of faults detected by a subject applying a given testing technique on a given program.

b) Percentage of faults detected = the percentage of a program's faults that a subject detected by applying a testing technique to the program.

c) Number of faults observable (*) = the number of faults that were observable from the program's behavior given the input data submitted.

d) Percentage of faults observable (*) = the percentage of a program's faults that were observable from the program's behavior given the input data submitted.

e) Percentage detected/observable (*) = the percentage of faults observable from the program's behavior on the given input set that were actually observed by a subject.

f) Percentage faults felt found = a subject's estimate of the percentage of a program's faults that he/she thought were detected by his/her testing.

g) Maximum statement coverage (*) = the maximum percentage of a program's statements that were executed in a set of test cases.

Phase	#Subj.	Measure	Mean	SD	Min.	Max.
1	29	# Faults detected	3.94	1.92	0.00	7.00
1	29	% Faults detected	54.79	26.11	0.00	100.00
1	29(*)	# Faults observable	5.28	1.51	3.00	9.00
1	29(*)	% Faults observable	74.39	20.54	33.33	100.00
1	29(*)	% Detected/observable	70.29	24.01	0.00	100.00
2	13	# Faults detected	3.23	1.96	0.00	7.00
2	13	% Faults detected	39.53	27.25	0.00	100.00
3	32	# Faults detected	4.27	1.96	0.00	9.00
3	32	% Faults detected	49.82	27.44	0.00	100.00
3	32	% Faults felt found	75.10	24.07	0.00	100.00
3	32(*)	# Faults observable	5.81	1.52	3.00	9.00
3	32(*)	% Faults observable	82.11	18.36	25.00	100.00
3	32(*)	% Detected/observable	69.87	27.14	0.00	100.00
3	32(*)	Max. % stmt. covered	97.02	7.83	46.00	100.00
Ave	74	# Faults detected	3.97	1.88	0.00	9.00
Ave	74	% Faults detected	49.96	27.29	0.00	100.00
Ave	61(*)	# Faults observable	5.5	1.5	3.00	9.00
Ave	61(*)	% Faults observable	89.0	20.3	25.0	100.0
Ave	61(*)	% Detected/observable	70.3	25.6	0.0	100.0

Fig. 9. Overall summary of detection effectiveness data. Note: some data pertain only to on-line techniques (*), and some data were collected only in certain phases.

1) Data Distributions: The actual distribution of the number of faults observed by the subjects appears in Fig. 10, broken down by phase. From Figs. 9 and 10, the large variation in performance among the subjects is clearly seen. The mean number of faults detected by the subjects is displayed in Fig. 11, broken down by technique, program, expertise level, and phase.

2) Number of Faults Detected: The first question under this goal area asks which of the testing techniques detected the most faults in the programs. The overall F-test of the techniques detecting an equal number of faults in the programs is rejected in the first and third phases of the study ($\alpha < 0.024$ and $\alpha < 0.0001$, respectively; not rejected in phase two, $\alpha > 0.05$). Recall that the phase three data was collected from 32 NASA/CSC subjects, and the phase one data was from 29 University of Maryland subjects. With the phase three data, the contrast of "reading − 0.5 * (functional + structural)" estimates

[3]Program statements within the body of a WHILE statement were considered unexecuted if the Boolean condition of the WHILE statement was false. Having the Boolean condition of the WHILE statement become true at some point was a prerequisite for executing the statements with the body of the WHILE.

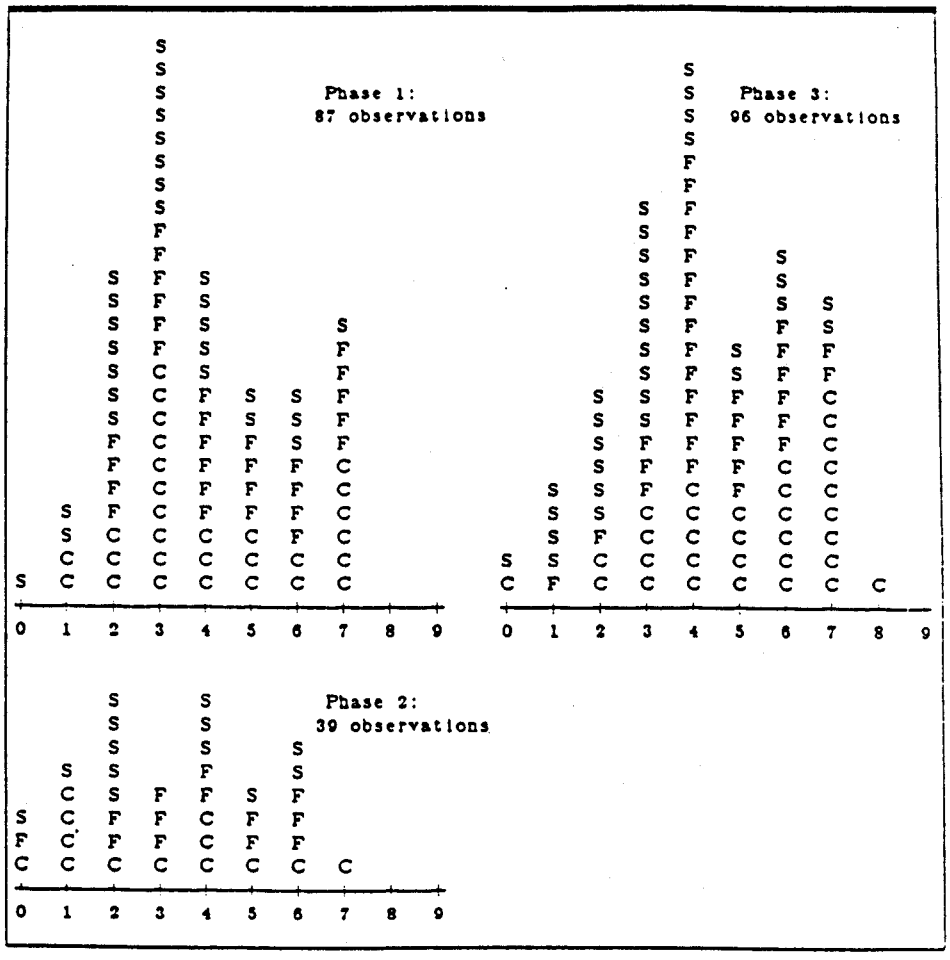

Fig. 10. Distribution of the number of faults detected broken down by phase. Key: code readers (C), functional testers (F), and structural testers (S).

		Phase		
		1	2	3
Effect	Level	Mean(SD)	Mean(SD)	Mean(SD)
Technique	Reading	4.10 (1.93)	3.00 (2.20)	5.09 (1.92)
	Functional	4.45 (1.70)	3.77 (1.53)	4.47 (1.34)
	Structural	3.28 (1.67)	3.09 (1.89)	3.25 (1.80)
Program	Formatter	4.07 (1.62)	3.23 (2.20)	4.19 (1.73)
	Plotter	3.48 (1.45)	3.31 (1.97)	. (.)
	Data type	4.28 (2.25)	. (.)	5.22 (1.75)
	Database	. (.)	3.31 (1.84)	3.41 (1.66)
Expertise	Junior	3.88 (1.89)	3.04 (2.07)	3.90 (1.83)
	Intermed.	4.07 (1.59)	3.53 (1.64)	4.18 (1.99)
	Advanced	. (.)	. (.)	5.00 (1.53)

Fig. 11. Overall summary for number of faults detected (SD = std. dev.).

that the technique of code reading by stepwise abstraction detected 1.24 more faults per program than did either of the other techniques ($\alpha < 0.0001$, c.i. 0.73–1.75).[4] Note that code reading performed well even though the professional subjects' primary experience was with functional testing. Also with the phase three data, the contrast of "functional − structural" estimates that the technique of functional testing detected 1.11 more faults per program than did structural testing ($\alpha < 0.0007$, c.i. 0.52–1.70). In the phase one data, the contrast of "0.5 * (reading + functional) − structural" estimates that the technique of structural testing detected 1.00 fault less per program than did either reading or functional testing ($\alpha < 0.0065$, c.i. 0.31–1.69). In the phase one data, the contrast of "reading − functional" was not statistically different from zero ($\alpha > 0.05$). The poor performance of structural testing across the phases suggests the inadequacy of using statement coverage criteria. The above pairs of contrasts were chosen because they are linearly independent.

3) Percentage of Faults Detected: Since the programs tested each had a different number of faults, a question in the earlier goal/question framework asks which technique detected the greatest percentage of faults in the programs. The order of performance of the techniques is the same as above when the percentage of the program's faults detected are compared. The overall F-tests for phases one and three were rejected as before ($\alpha < 0.037$ and $\alpha < 0.0001$, respectively; not rejected in phase two, $\alpha > 0.05$). Applying the same contrasts as above: a) in phase three, reading detected 16.0 percent more faults per program than did the other techniques ($\alpha < 0.0001$, c.i. 9.9–

[4]The probability of Type I error is reported, the probability of erroneously rejecting the null hypothesis. The abbreviation "c.i." stands for 95 percent confidence interval.

22.1), and functional detected 11.2 percent more faults than did structural ($\alpha < 0.003$, c.i. 4.1–18.3); b) in phase one, structural detected 13.2 percent fewer of a program's faults than did the other methods ($\alpha < 0.011$, c.i. 3.5–22.9), and reading and functional were not statistically different as before.

4) Dependence on Software Type: Another question in this goal area queries whether the number or percentage of faults detected depends on the program being tested. The overall F-test that the number of faults detected is not program dependent is rejected only in the phase three data ($\alpha < 0.0001$). Applying Tukey's multiple comparison on the phase three data reveals that the most faults were detected in the abstract data type, the second most in the text formatter, and the least number of faults were found in the database maintainer (simultaneous $\alpha < 0.05$). When the percentage of faults found in a program is considered, however, the overall F-tests for the three phases are all rejected ($\alpha < 0.027$, $\alpha < 0.01$, and $\alpha < 0.0001$ in respective order). Tukey's multiple comparison yields the following orderings on the programs (all simultaneous $\alpha < 0.05$). In the phase one data, the ordering was (data type $\simeq$ plotter) $>$ text formatter; that is, a higher percentage of faults were detected in either the abstract data type or the plotter than were found in the text formatter; there was no difference between the abstract data type and the plotter in the percentage found. In the phase two data, the ordering of percentage of faults detected was plotter $>$ (text formatter $\simeq$ database maintainer). In the phase three data, the ordering of percentage of faults found in the programs was the same as the number of faults found, abstract data type $>$ text formatter $>$ database maintainer. Summarizing the effect of the type of software on the percentage of faults observed: 1) the programs with the highest percentage of their faults detected were the abstract data type and the mathematical plotter, the percentage detected between these two was not statistically different; 2) the programs with the lowest percentage of their faults detected were the text formatter and the database maintainer; the percentage detected between these two was not statistically different in the phase two data, but a higher percentage of faults in the text formatter was detected in the phase three data.

5) Observable Versus Observed Faults: One evaluation criteria of the success of a software testing session is the number of faults detected. An evaluation criteria of the particular test data generated, however, is the ability of the test data to reveal faults in the program. A test data set's ability to reveal faults in a program can be measured by the number or percentage of a program's faults that are made observable from execution on that input.[5] Distinguishing the faults observable in a program from the faults actually observed by a tester highlights the differences in the activities of test data generation and program behavior examination. As shown in Fig. 8, the average number of the programs' faults observable was 68.0 percent when individuals were either functional testing or structural testing. Of course, with a nonexecution-based technique such as code reading, 100 percent of the faults are observable. Test data generated by subjects using the technique of functional testing resulted in 1.4 more observable faults ($\alpha < 0.0002$, c.i. 0.79–2.01) than did the use of structural testing in phase one of the study; the percentage difference of functional over structural was estimated at 20.0 percent ($\alpha < 0.,0002$, c.i. 11.2–28.8). The techniques did not differ in these two measures in the third phase of the study. However, just considering the faults that were observable from the submitted test data, functional testers detected 18.5 percent more of these observable faults than did structural testers in the phase three data ($\alpha < 0.0016$, c.i. 8.9–28.1); they did not differ in the phase one data. Note that all faults in the programs could be observed in the programs' output given the proper input data. When using the on-line techniques of functional and structural testing, subjects detected 70.3 percent of the faults observable in the program's output. In order to conduct a successful testing session, faults in a program must be both revealed and subsequently observed.

6) Dependence on Program Coverage: Another measure of the ability of a test set to reveal a program's faults is the percentage of a program's statements that are executed by the test set. The average maximum statement coverage achieved by the functional and structural testers was 97.0 percent. The maximum statement coverage from the submitted test data was not statistically different between the functional and structural testers ($\alpha > 0.05$). Also, there was no correlation between maximum statement coverage achieved and either number or percentage of faults found ($\alpha > 0.05$).

7) Dependence on Programmer Expertise: A final question in this goal area concerns the contribution of programmer expertise to fault detection effectiveness. In the phase three data from the NASA/CSC professional environment, subjects of advanced expertise detected more faults than did either the subjects of intermediate or junior expertise ($\alpha < 0.05$). When the percentage of faults detected is compared, however, the advanced subjects performed better than the junior subjects ($\alpha < 0.05$), but were not statistically different from the intermediate subjects ($\alpha > 0.05$). The intermediate and junior subjects were not statistically different in any of the three phases of the study in terms of number or percentage faults observed. When several subject background attributes were correlated with the number of faults found, total years of professional experience had a minor relationship (Pearson $R = 0.22$, $\alpha < 0.05$). Correspondence of performance with background aspects was examined across all observations, and within each of the phases, including previous academic performance for the University of Maryland

[5]Test data "reveal a fault" or "make a fault observable" by making a fault be manifested as a program failure (see the explanation in the earlier section entitled Fault Description). Since the analysis is focusing on the number of distinct software faults revealed—and for purposes of readability—this paragraph uses the single word "fault."

subjects. Other than the above, no relationships were found.

8) Accuracy of Self-Estimates: Recall that the NASA/CSC subjects in the phase three data estimated, at the completion of a testing session, the percentage of a program's faults they thought they had uncovered. This estimation of the number of faults uncovered correlated reasonably well with the actual percentage of faults detected ($R = 0.57$, $\alpha < 0.0001$). Investigating further, individuals using the different techniques were able to give better estimates: code readers gave the best estimates ($R = 0.79$, $\alpha < 0.0001$), structural testers gave the second best estimates ($R = 0.57$, $\alpha < 0.0007$), and functional testers gave the worst estimates (no correlation, $\alpha > 0.05$). This last observation suggests that the code readers were more certain of the effectiveness they had in revealing faults in the programs.

9) Dependence on Interactions: There were few significant interactions between the main effects of testing technique, program, and expertise level. In the phase two data, there was an interaction between testing technique and program in both the number and percentage of faults found ($\alpha < 0.0013$, $\alpha < 0.0014$, respectively). The effectiveness of code reading increased on the text formatter. In the phase three data, there was a slight three-way interaction between testing technique, program, and expertise level for both the number and percentage of faults found ($\alpha < 0.05$, $\alpha < 0.04$ respectively).

10) Summary of Fault Detection Effectiveness: Summarizing the major results of the comparison of fault detection effectiveness: 1) in the phase three data, code reading detected a greater number and percentage of faults than the other methods, with functional detecting more than structural; 2) in the phase one data, code reading and functional were equally effective, while structural was inferior to both—there were no differences among the three techniques in phase two; 3) the number of faults observed depends on the type of software; the most faults were detected in the abstract data type and the mathematical plotter, the second most in the text formatter, and (in the case of the phase three data) the least were found in the database maintainer; 4) functionally generated test data revealed more observable faults than did structurally generated test data in phase one, but not in phase three; 5) subjects of intermediate and junior expertise were equally effective in detecting faults, while advanced subjects found a greater number of faults than did either group; 6) self-estimates of faults detected were most accurate from subjects applying code reading, followed by those doing structural testing, with estimates from persons functionally testing having no relationship.

B. Fault Detection Cost

The second goal area examines the fault detection cost of each of the techniques. Fig. 12 presents a summary of the measures that were examined to investigate this goal area. A brief description of each measure is as follows; an asterisk (*) means only relevant for on-line testing. All

Phase	#Subj.	Measure	Mean	SD	Min.	Max.
1	29	# Faults / hour	1.63	1.28	0.00	7.00
1	29	Detection time (hrs)	3.33	2.09	0.75	10.00
2	13	# Faults / hour	0.99	0.81	0.00	3.00
2	13	Detection time (hrs)	4.70	3.02	1.00	14.00
3	32	# Faults / hour	2.33	2.29	0.00	14.00
3	32	Detection time (hrs)	2.75	1.57	0.50	7.25
3	32(*)	Cpu-time (sec)	45.2	56.1	3.0	283.0
3	32(*)	Cpu-time (sec: norm.)	38.5	51.7	2.9	314.4
3	32(*)	Connect time (min)	65.83	50.21	3.50	214.00
3	32(*)	# program runs	5.45	5.00	1.00	24.00
Ave	74	# Faults / hour	1.82	1.80	0.00	14.00
Ave	74	Detection time (hrs)	3.32	2.19	0.50	14.00

Fig. 12. Overall summary of fault detection cost data. Note: some data pertain only to on-line techniques (*), and some data were collected only in certain phases.

of the on-line statistics were monitored by the operating systems of the machines.

a) Number of faults/hour = the number of faults detected by a subject applying a given technique normalized by the effort in hours required, called the fault detection rate.

b) Detection time = the total number of hours that a subject spent in testing a program using a technique.

c) Cpu-time (*) = the cpu-time in seconds used during the testing session.

d) Normalized cpu-time (*) = the cpu-time in seconds used during the testing session, normalized by a factor for machine speed.[6]

e) Connect time (*) = the number of minutes that a individual spent on-line while testing a program.

f) Number of program runs (*) = the number of executions of the program test driver; note that the driver supported multiple sets of input data.

1) Data Distributions: The actual distribution of the fault detection rates for the subjects appears in Fig. 13, broken down by phase. Once again, note the many-to-one differential in subject performance. Fig. 14 displays the mean fault detection rate for the subjects, broken down by technique, program, expertise level, and phase.

2) Fault Detection Rate and Total Time: The first question in this goal area asks which testing technique had the highest fault detection rate. The overall F-test of the techniques having the same detection rate was rejected in the phase three data ($\alpha < 0.0014$), but not in the other two phases ($\alpha > 0.05$). As before, the two contrasts of "reading − 0.5 * (functional + structural)" and "functional − structural" were examined to detect differences among the techniques. The technique of code reading was estimated at detecting 1.49 more faults per hour than did the other techniques in the phase three data ($\alpha < 0.0003$, c.i. 0.75–2.23). The techniques of functional and structural testing were not statistically different ($\alpha > 0.05$). Comparing the total time spent in fault detection, the techniques were not statistically different in the phase two and three data; the overall F-test for the phase one data

[6]In the phase three data, testing was done on both a VAX 11/780 and an IBM 4341. As suggested by benchmark comparisons [11], the VAX cpu-times were divided by 1.6 and the IBM cpu-times were divided by 0.9.

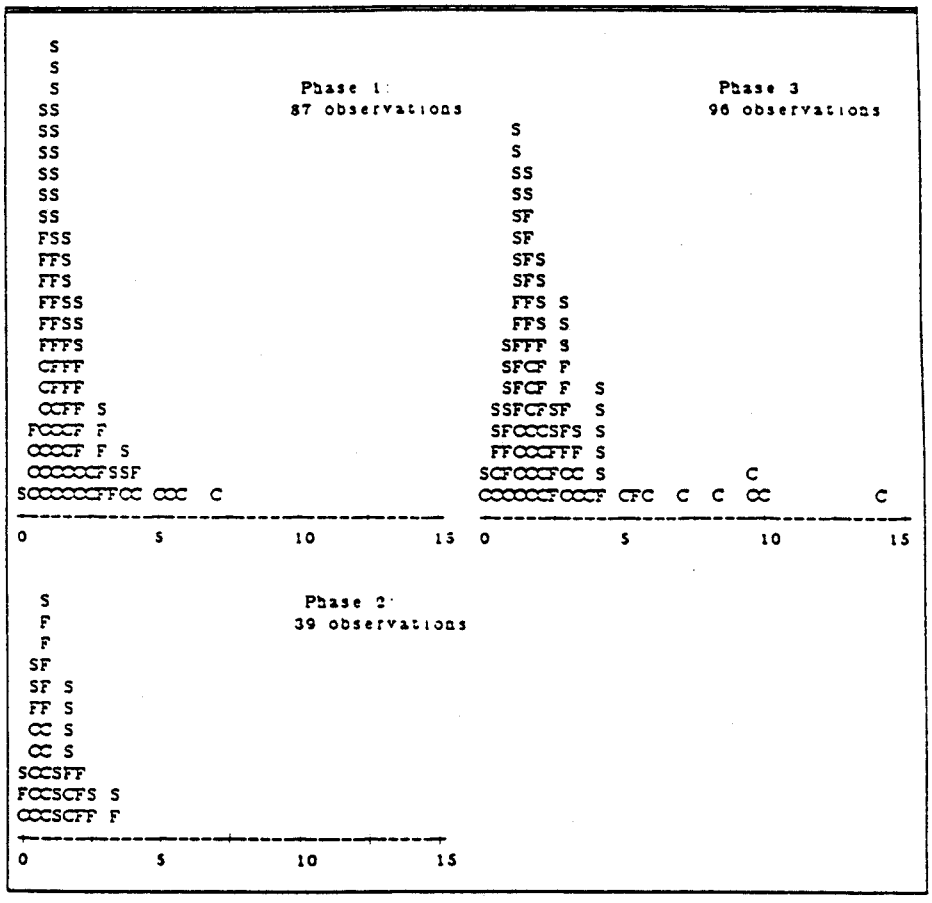

Fig. 13. Distribution of the fault detection rate (number of faults detected per hour) broken down by phase. Key: code readers (C), functional testers (F), and structural testers (S).

		Phase		
		1	2	3
Effect	Level	Mean(SD)	Mean(SD)	Mean(SD)
Technique	Reading	1.90 (1.83)	0.56 (0.46)	3.33 (3.42)
	Functional	1.38 (0.90)	1.22 (0.91)	1.84 (1.06)
	Structural	1.40 (0.87)	1.19 (0.84)	1.82 (1.24)
Program	Formatter	1.60 (1.39)	0.98 (0.67)	2.15 (1.10)
	Plotter	1.19 (0.93)	0.92 (0.71)	. (.)
	Data type	2.09 (1.42)	. (.)	3.70 (3.28)
	Database	. (.)	1.05 (1.04)	1.14 (0.79)
Expertise	Junior	1.36 (0.87)	1.00 (0.85)	2.14 (2.48)
	Intermed.	2.22 (1.56)	0.96 (0.74)	2.53 (2.48)
	Advanced	. (.)	. (.)	2.36 (1.61)

Fig. 14. Overall summary for fault detection rate (number of faults detected per hour) (SD = std. dev).

was rejected ($\alpha < 0.013$). In the phase one data, structural testers spent an estimated 1.08 hours less testing than did the other techniques ($\alpha < 0.004$, c.i. 0.39–1.78), while code readers were not statistically different from functional testers. Recall that in phase one, the structural testers observed both a lower number and percentage of the programs' faults than did the other techniques.

3) Dependence on Software Type: Another question in this area focuses on how fault detection rate depends on software type. The overall F-test that the detection rate is the same for the programs is rejected in the phase one and phase three data ($\alpha < 0.01$ and $\alpha < 0.0001$, respectively); the detection rate among the programs was not statistically different in phase two ($\alpha > 0.05$). Applying Tukey's multiple comparison on the phase one data finds that the fault detection rate was greater on the abstract data type than on the plotter, while there was no difference either between the abstract data type and the text formatter or between the text formatter and the plotter (simultaneous $\alpha < 0.05$). In the phase three data, the fault detection rate was higher in the abstract data type than it was for the text formatter and the database maintainer, with the text formatter and the database maintainer not being statistically different (simultaneous $\alpha < 0.05$). The overall effort spent in fault detection was different among the programs in phases one and three ($\alpha < 0.012$ and $\alpha < 0.0001$, respectively), while there was no difference in phase two. In phase one, more effort was spent testing the plotter than the abstract data type, while there was no statistical difference either between the plotter and the text formatter or between the text formatter and the abstract data type (simultaneous $\alpha < 0.05$). In phase three, more time was spent testing the database maintainer than was spent on either the text formatter or on the abstract data type, with the text formatter not differing from the abstract data type (simultaneous $\alpha < 0.05$). Summarizing the dependence of fault detection cost on software type,

1) the abstract data type had a higher detection rate and less total detection effort than did either the plotter or the database maintainer, the latter two were not different in either detection rate or total detection time; 2) the text formatter and the plotter did not differ in fault detection rate or total detection effort; 3) the text formatter and the database maintainer did not differ in fault detection rate overall and did not differ in total detection effort in phase two, but the database maintainer had a higher total detection effort in phase three; 4) the text formatter and the abstract data type did not differ in total detection effort overall and did not differ in fault detection rate in phase one, but the abstract data type had a higher detection rate in phase three.

4) Computer Costs: In addition to the effort spent by individuals in software testing, on-line methods incur machine costs. The machine cost measures of cpu-time, connect time, and the number of runs were compared across the on-line techniques of functional and structural testing in phase three of the study. A nonexecution-based technique such as code reading, of course, incurs no machine time costs. When the machine speeds are normalized (see measure definitions above), the technique of functional testing used 26.0 more seconds of cpu-time than did the technique of structural testing ($\alpha < 0.016$, c.i. 7.0–45.0). The estimate of the difference is 29.6 seconds when the cpu-times are not normalized ($\alpha < 0.012$, c.i. 9.0–50.2). Individuals using functional testing used 28.4 more minutes of connect time than did those using structural testing ($\alpha < 0.004$, c.i. 11.7–45.1). The number of computer runs of a program's test driver was not different between the two techniques ($\alpha > 0.05$). These results suggest that individuals using functional testing spent more time on-line and used more cpu-time per computer run than did those structurally testing.

5) Dependence on Programmer Expertise: The relation of programmer expertise to cost of fault detection is another question in this goal section. The expertise level of the subjects had no relation to the fault detection rate in phases two and three ($\alpha > 0.05$ for both F-tests). Recall that phase three of the study used 32 professional subjects with all three levels of computer science expertise. In phase one, however, the intermediate subjects detected faults at a faster rate than did the junior subjects ($\alpha < 0.005$). The total effort spent in fault detection was not different among the expertise levels in any of the phases ($\alpha > 0.05$ for all three F-tests). When all 74 subjects are considered, years of professional experience correlates positively with fault detection rate ($R = 0.41$, $\alpha < 0.0002$) and correlates negatively with total detection time ($R = -0.25$, $\alpha < 0.03$). These last two observations suggest that persons with more years of professional experience detected the faults faster and spent less total time doing so. Several other subject background measures showed no relationship with fault detection rate or total detection time ($\alpha > 0.05$). Background measures were examined across all subjects and within the groups of NASA/CSC subjects and University of Maryland subjects.

6) Dependence on Interactions: There were few significant interactions between the main effects of testing technique, program, and expertise level. There was an interaction between testing technique and software type in terms of fault detection rate and total detection cost for the phase three data ($\alpha < 0.003$ and $\alpha < 0.007$, respectively). Subjects using code reading on the abstract data type had an increased fault detection rate and a decreased total detection time.

7) Relationships Between Fault Detection Effectiveness and Cost: There were several correlations between fault detection cost measures and performance measures. Fault detection rate correlated overall with number of faults detected ($R = 0.48$, $\alpha < 0.0001$), percentage of faults detected ($R = 0.48$, $\alpha < 0.0001$), and total detection time ($R = -0.53$, $\alpha < 0.0001$), but not with normalized cpu-time, raw cpu-time, connect time, or number of computer runs ($\alpha > 0.05$). Total detection time correlated with normalized cpu-time ($R = 0.36$, $\alpha < 0.04$) and raw cpu-time ($R = 0.37$, $\alpha < 0.04$), but not with connect time, number of runs, number of faults detected, or percentage of faults detected ($\alpha > 0.05$). The number of faults detected in the programs correlated with the amount of machine resources used: normalized cpu-time ($R = 0.47$, $\alpha < 0.007$), raw cpu-time ($R = 0.52$, $\alpha < 0.002$), and connect time ($R = 0.49$, $\alpha < 0.003$), but not with the number of computer runs ($\alpha > 0.05$). The correlations for percentage of faults detected with machine resources used were similar. Although most of these correlations are weak, they suggest that 1) the higher the fault detection rate, the more faults found and the less time spent in fault detection; 2) fault detection rate had no relationship with use of machine resources; 3) spending more time in detecting faults had no relationship with the amount of faults detected; and 4) the more cpu-time and connect time used, the more faults found.

8) Summary of Fault Detection Cost: Summarizing the major results of the comparison of fault detection cost: 1) in the phase three data, code reading had a higher fault detection rate than the other methods, with no difference between functional testing and structural testing; 2) in the phase one and two data, the three techniques were not different in fault detection rate; 3) in the phase two and three data, total detection effort was not different among the techniques, but in phase one less effort was spent for structural testing than for the other techniques, while reading and functional were not diffferent; 4) fault detection rate and total effort in detection depended on the type of software: the abstract data type had the highest detection rate and lowest total detection effort, the plotter and the database maintainer had the lowest detection rate and the highest total detection effort, and the text formatter was somewhere in between depending on the phase; 5) in phase three, functional testing used more cpu-time and connect time than did structural testing, but they were not different in the number of runs; 6) in phases two and three, subjects across expertise levels were not different in fault detection rate or total detection time, in phase one intermediate subjects had a higher detection rate; and 7) there

was a moderate correlation between fault detection rate and years of professional experience across all subjects.

C. Characterization of Faults Detected

The third goal area focuses on determining what classes of faults are detected by the different techniques. In the earlier section on the faults in the software, the faults were characterized by two different classification schemes: omission or commission; and initialization, control, data, computation, interface, or cosmetic. The faults detected across all three study phases are broken down by the two fault classification schemes in Fig. 15. The entries in the figure are the average percentage (with standard deviation) of faults in a given class observed when a particular technique was being used. Note that when a subject tested a program that had no faults in a given class, he/she was excluded from the calculation of this average.

1) Omission Versus Commission Classification: When the faults are partitioned according to the omission/commission scheme, there is a distinction among the techniques. Both code readers and functional testers observed more omission faults than did structural testers ($\alpha < 0.001$), with code readers and functional testers not being different ($\alpha > 0.05$). Since a fault of omission occurs as a result of some segment of code being left out, you would not expect structurally generated test data to find such faults. In fact, 44 percent of the subjects applying structural testing found zero faults of omission when testing a program. A distribution of the faults observed according to this classification scheme appears in Fig. 16.

2) Six-Part Fault Classification: When the faults are divided according to the second fault classification scheme, several differences are apparent. Both code reading and functional testing found more initialization faults than did structural testing ($\alpha < 0.05$), with code reading and functional testing not being different ($\alpha > 0.05$). Code reading detected more interface faults than did either of the other methods ($\alpha < 0.01$), with no difference between functional and structural testing ($\alpha > 0.05$). This suggests that the code reading process of abstracting and composing program functions across modules must be an effective technique for finding interface faults. Functional testing detected more control faults than did either of the other methods ($\alpha < 0.01$), with code reading and structural testing not being different ($\alpha > 0.05$). Recall that the structural test data generation criteria examined is based on determining the execution paths in a program and deriving test data that execute 100 percent of the program's statements. One would expect that more control path faults would be found by such a technique. However, structural testing did not do as well as functional testing in this fault class. The technique of code reading found more computation faults than did structural testing ($\alpha < 0.05$), with functional testing not being different from either of the other two methods ($\alpha > 0.05$). The three techniques were not statistically different in the percentage of faults they detected in either the data or cosmetic fault classes ($\alpha > 0.05$ for both). A distribution of the faults observed according to this classification scheme appears in Fig. 17.

3) Observable Fault Classification: Fig. 18 displays the averge percentage (with standard deviation) of faults from each class that were observable from the test data

	Code Reading	Functional Testing	Structural Testing	Overall
Omission	55.6 (40.1)	61.0 (39.5)	39.2 (41.6)	52.0 (41.3)
Commission	54.3 (32.1)	53.5 (25.4)	44.3 (26.6)	50.7 (28.4)
Total	54.1 (29.2)	54.6 (24.5)	41.2 (26.1)	50.0 (27.3)
Initial.	64.6 (40.3)	75.0 (36.1)	46.2 (39.8)	61.5 (40.2)
Control	42.8 (36.6)	66.7 (34.9)	48.8 (36.5)	52.8 (37.2)
Data	20.7 (36.6)	28.3 (44.9)	26.8 (41.9)	25.3 (41.0)
Computat.	70.9 (37.0)	64.2 (40.8)	58.8 (43.5)	64.6 (40.6)
Interface	46.7 (38.5)	30.7 (33.5)	24.6 (29.4)	34.1 (35.1)
Cosmetic	16.7 (38.1)	9.3 (29.2)	7.7 (27.2)	10.8 (31.3)
Total	54.1 (29.2)	54.6 (24.5)	41.2 (26.1)	50.0 (27.3)

Fig. 15. Characterization of the faults detected. Mean (and std. dev.) of the percentage of faults in each class that were detected.

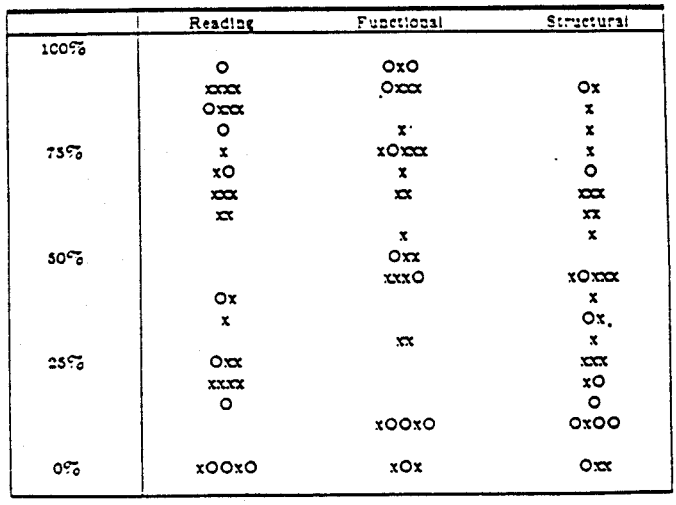

Fig. 16. Characterization of faults detected by the three techniques: 10 omission (0) versus 24 commission (x). The vertical axis is the percentage of persons using the particular technique that detected the fault.

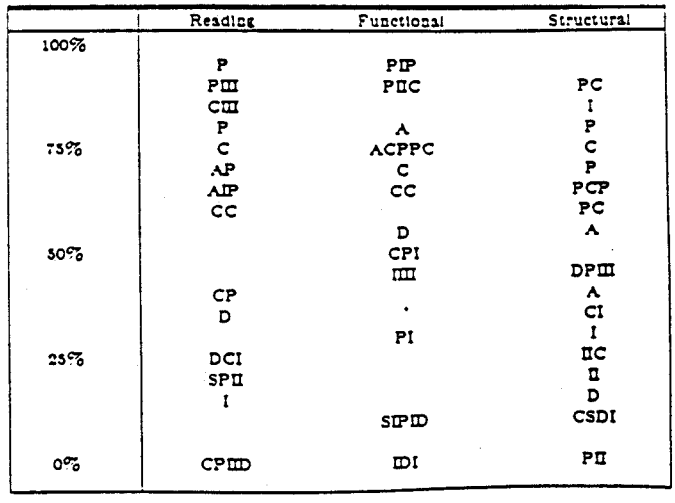

Fig. 17. Characterization of faults detected by the three techniques. Initialization (2-A), computation (8-P), control (7-C), data (3-D), interface (13-I), and cosmetic (1-S). The vertical axis is the percentage of the persons using the particular technique that detected the fault.

	Functional Testing	Structural Testing	Overall
Omission	15.7 (25.4)	21.3 (31.8)	19.5 (28.9)
Commission	19.1 (20.0)	20.1 (16.6)	19.6 (18.3)
Total	18.1 (17.8)	19.9 (16.8)	19.0 (17.3)
Initial.	5.0 (15.4)	14.3 (32.2)	9.8 (25.5)
Control	20.3 (30.6)	21.1 (31.4)	20.7 (30.8)
Data	28.6 (43.5)	7.5 (24.5)	18.3 (36.7)
Computat.	16.0 (31.3)	20.1 (37.6)	18.0 (34.5)
Interface	16.1 (20.0)	20.3 (21.5)	18.2 (20.9)
Cosmetic	60.0 (50.3)	85.7 (35.9)	73.2 (44.9)
Total	18.1 (17.8)	19.9 (16.8)	19.0 (17.3)

Fig. 18. Characterization of the faults observable but not reported. The mean (and std. dev.) of the percentage of such faults in each class are given. (With the appropriate inputs, all faults could be made observable in the program output. The faults included here are those that were observable given the program inputs selected by the testers yet were unreported.)

submitted, yet were not reported by the tester.[7] The two on-line techniques of functional and structural testing were not different in any of the faults classes ($\alpha > 0.05$). Note that there was only one fault in the cosmetic class.

4) Summary of Characterization of Faults Detected: Summarizing the major results of the comparison of classes of faults detected: 1) code reading and functional testing both detected more omission faults and initialization faults than did structural testing; 2) code reading detected more interface faults than did the other methods; 3) functional testing detected more control faults than did the other methods; 4) code reading detected more computation faults than did structural testing; and 5) the on-line techniques of functional and structural testing were not different in any classes of faults observable but not reported.

V. Conclusions

This study compares the strategies of code reading by stepwise abstraction, functional testing using equivalence class partitioning and boundary value analysis, and structural testing using 100 percent statement coverage. The study evaluates the techniques across three data sets in three different aspects of software testing: fault detection effectiveness, fault detection cost, and classes of faults detected. The three data sets involved a total of 74 programmers applying each of the three testing techniques on unit-sized software; therefore, the analysis and results presented were based on observations from a total of 222 testing sessions. The investigation is intended to compare the different testing strategies in representative testing situations, using programmers with a wide range of experience, different software types, and common software faults.

In this controlled study, an experimentation methodology was applied to compare the effectiveness of three testing techniques; for an overview of the experimentation methodology, see [4]. Based on our experience and observation [56], the three testing techniques represent the high end of the range of testing methods that are actually being used by developers to test software. The techniques examined correspond, therefore, to the state-of-the-practice of software testing rather than the state-of-the-art. As mentioned earlier, there exist alternate forms for each of the three testing methods.

There are several perspectives from which to view empirical studies of software development techniques. Three example perspectives given were that of the experimenter, researcher, and practitioner. One key aspect of the study presented, especially from an experimenter's perspective, was the use of an experimentation methodology and a formal statistical design. The actual empirical results from the study, which are summarized below, may be used to refine a researcher's theories about software testing or to guide a practitioner's application of the techniques.

Each of the three testing techniques showed some merit in this evaluation. The major empirical results of this study are the following. 1) With the professional programmers, code reading detected more software faults and had a higher fault detection rate than did functional or structural testing, while functional testing detected more faults than did structural testing, but functional and structural testing were not different in fault detection rate. 2) In one University of Maryland (UoM) subject group, code reading and functional testing were not different in faults found, but were both superior to structural testing, while in the other UoM subject group there was no difference among the techniques. 3) With the UoM subjects, the three techniques were not different in fault detection rate. 4) Number of faults observed, fault detection rate, and total effort in detection depended on the type of software tested. 5) Code reading detected more interface faults than did the other methods. 6) Functional testing detected more control faults than did the other methods. 7) When asked to estimate the percentage of faults detected, code readers gave the most accurate estimates while functional testers gave the least accurate estimates.

The results suggest that code reading by stepwise abstraction (a nonexecution-based method) is at least as effective as on-line functional and structural testing in terms of number and cost of faults observed. They also suggest the inadequacy of using 100 percent statement coverage criteria for structural testing. Note that the professional programmers examined preferred the use of functional testing because they felt is was the most effective technique; their intuition, however, turned out to be incorrect. Recall that the code reading was performed on uncommented programs, which could be considered a worst-case scenario for code reading.

In comparing the results to related studies, there are mixed conclusions. A prototype analysis done at the University of Maryland in the Fall of 1981 [30] supported the belief that code reading by stepwise abstraction does as well as the computer-based methods, with each strategy having its own advantages. In the Myers experiment [41], the three techniques compared (functional testing, 3-person code reviews, control group) were equally effective. He also calculated that code reviews were less cost-effective than the computer-based testing approaches. The first

[7]The standard deviations presented in the figure are high because of the several instances in which all observable faults were reported.

observation is supported in one study phase here, but the other observation is not. A study conducted by Hetzel [23] compared functional testing, code reading, and "selective" testing (a composite of functional, structural, and reading techniques). He observed that functional and "selective" testing were equally effective, with code reading being inferior. As noted earlier, this is not supported by this analysis. The study described in this analysis examined the technique of code reading by stepwise abstraction, while both the Myers and Hetzel studies examined alternate approaches to off-line (nonexecution-based) review/reading. Other studies that have compared the effectiveness of software testing strategies include [22], [32], [21], [20], [24], [8], [26], [28], [55], [38], [45], [17].

A few remarks are appropriate about the comparison of the cost-effectiveness and phase-availability of these testing techniques. When examining the effort associated with a technique, both fault detection and fault isolation costs should be compared. The code readers have both detected and isolated a fault; they located it in the source code. Thus, the reading process condenses fault detection and isolation into one activity. Functional and structural testers have only detected a fault; they need to delve into the source code and expend additional effort in order to isolate the fault. Moreover, the code reading process corresponds more closely to the activity of program proving than do the other methods. Also, a nonexecution-based reading process can be applied to any document produced during the development process (e.g., high-level design document, low-level design document, source code document). While functional and structural execution-based techniques may only be applied to documents that are executable (e.g., source code), which are usually available later in the development process.

Investigations related to this work include studies of fault classification [54], [34], [44], [1] and Cleanroom software development [50]. In the Cleanroom software development approach, techniques such as code reading are used in the development of software completely off-line (i.e., without program execution). In [50], systems developed using Cleanroom met system requirements more completely and had a higher percentage of successful operational test cases than did systems developed with a more traditional approach.

The work presented in this paper differs from previous studies in several ways. 1) The nonexecution-based software review technique used was code reading by stepwise abstraction. 2) The study was based on programmers—including professionals—having varying expertise, different software types, and programs having a representative profile of common software faults. 3) A very sensitive statistical design was employed to account for differences in individual performance and interactions among testing technique, software type, and subject expertise level. 4) The study was conducted in multiple phases in order to refine experimentation methods. 5) The scope of issues examined was broadened (e.g., observed versus observable faults, structural coverage of functional testing, multiple fault classification schemes).

The empirical study presented is intended to advance the understanding of how various software testing strategies contribute to the software development process and to one another. The results given were calculated from a set of individuals applying the three techniques to unit-sized programs—the direct extrapolation of the findings to other testing environments is not implied. Further work applying these and other results to devise effective testing environments is underway [49].

Appendix
The Specifications for the Programs

Program 1[8]

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 line, followed by a / (end-of-text) and a carriage return. The program then echoes the text input and formats 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 properly terminated line of text.

Program 2

Given ordered pairs (x, y) of either positive or negative integers as input, the program plots them on a grid with a horizontal x-axis and a vertical y-axis which are appropriately labeled. A plotted point on the grid should appear as an asterisk (*).

The vertical and horizontal scaling is handled as follows. If the maximum absolute value of any y-value is less than or equal to 20, the scale for vertical spacing will be one line per integral unit [e.g., the point (3, 6) should be plotted on the sixth line, two lines above the point (3, 4)]. Note that the origin [point (0, 0)] would correspond to an asterisk at the intersection of the axes (the x-axis is

[8]Note that this specification was rewritten in [37].

referred to as the 0th line). If the maximum absolute value of any x-value is less than or equal to 30, the scale for horizontal spacing will be one space per integral unit [e.g., the point (4, 5) should be plotted four spaces to the right of the y-axis, two spaces to the right of (2, 5)]. However, if the maximum absolute value of any y-value is greater than 20, the scale for vertical spacing will be one line per every (max absolute value of y-values)/20 rounded-up. [e.g., If the maximum absolute value of any y-value to be plotted is 66, the vertical line spacing will be a line for every 4 integral units. In such a data set, points with y-values greater than or equal to eight and less than twelve will show up as asterisks in the second line, points with y-values greater than or equal to twelve and less than sixteen will show up as asterisks in the third line, etc. Continuing the example, the point (3, 15) should be plotted on the third line, two lines above the point (3, 5).] Horizontal scaling is handled analogously.

If two or more of the points to be plotted would show up as the same asterisk in the grid (like the points (9, 13) and (9, 15) in the above example), a number "2" (or whatever number is appropriate) should be printed instead of the asterisk. Points whose asterisks will lie on a axis or grid marker should show up in place of the marker.

Program 3

A list is defined to be an ordered collection of integer 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 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 the integer 0 being returned. The detailed operation descriptions are as below:

ADDFIRST(LIST L, INTEGER I)
Returns the list L with I as its first element followed by all the elements of L. If L is "full" to begin with, L's last element is lost.

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

DELETEFIRST(LIST L)
Returns the list containing all but the first element of L. If L is empty, then an empty list is returned.

DELETELAST(LIST L)
Returns the list containing all but the last element of L. If L is empty, then an empty list is returned.

FIRST(LIST L)
Returns the first element in L. If L is empty, then it returns zero.

ISEMPTY(LIST L)
Returns one if L is empty, zero otherwise.

LISTLENGTH(LIST L)
Returns the number of elements in L. An empty list has zero elements.

NEWLIST(LIST L)
Returns an empty list.

REVERSE(LIST L)
Returns a list containing the elements of L in reverse order.

Program 4

(Note that 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 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 an "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

```
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.)

Acknowledgment

The authors are grateful to F. T. Baker, F. E. McGarry, and G. Page for their assistance in the organization of the study. The authors appreciate the comments from J. D. Gannon, H. D. Mills, and R. N. Taylor on an earlier version of this paper. The authors are grateful to the subjects from Computer Sciences Corporation, NASA Goddard, and the University of Maryland for their enthusiastic participation in the study.

References

[1] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Commun. ACM*, vol. 27, no. 1, pp. 42-52, Jan. 1984.

[2] V. R. Basili and R. W. Selby, "Data collection and analysis in software research and management," in *Proc. Amer. Statist. Ass. and Biometric Soc. Joint Statistical Meetings*, Philadelphia, PA, Aug. 13-16, 1984.

[3] ——, "Comparing the effectiveness of software testing strategies," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1501, May 1985.

[4] V. R. Basili, R. W. Selby, and D. H. Hutchens, "Experimentation in software engineering," *IEEE Trans. Software Eng.*, vol. SE-12, no. 7, pp. 733-743, July 1986.

[5] V. R. Basili and A. J. Turner, *SIMPL-T: A Structured Programming Language*. Paladin House, 1976.

[6] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Trans. Software Eng.*, vol. SE-10, no. 6, pp. 728-738, Nov. 1984.

[7] G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters*. New York: Wiley, 1978.

[8] T. A. Budd, R. J. Lipton, F. G. Sayward, and R. DeMillo, "The design of a prototype mutation system for program testing," *Proc. AFIPS Conf.*, vol. 47, pp. 623-627, 1978.

[9] R. Cailliau and F. Rubin, "ACM forum: On a controlled experiment in program testing," *Commun. ACM*, vol. 22, pp. 687-688, Dec. 1979.

[10] L. A. Clarke, Program Chair, *Proc. Workshop Software Testing*, Banff, Alta., Canada, July 15-17, 1986.

[11] V. Church, "Benchmark statistics for the VAX 11/780 and the IBM 4341," Computer Sciences Corporation, Silver Spring, MD, Internal Memo, 1984.

[12] W. G. Cochran and G. M. Cox, *Experimental Designs*. New York: Wiley, 1950.

[13] J. C. Deprie, "Report from the IFIP Working Group on Terminology," in *Proc. 15th Annu. Int. Symp. Fault Tolerant Computing*, University of Michigan, Ann Arbor, MI, June 19-21, 1985.

[14] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 182-211, 1976.

[15] K. A. Foster, "Error sensitive test cases," *IEEE Trans. Software Eng.*, vol. SE-6, no. 3, pp. 258-264, 1980.

[16] P. G. Frankl and E. J. Weyuker, "Data flow testing in the presence of unexecutable paths," in *Proc. Workshop Software Testing*, Banff, Alta., Canada, July 15-17, 1986, pp. 4-13.

[17] M. R. Girgis and M. R. Woodward, "An experimental comparison of the error exposing ability of program testing criteria," in *Proc. Workshop Software Testing*, Banff, Alta., Canada, July 15-17, 1986, pp. 64-73.

[18] S. A. Gloss-Soler, "The DACS glossary: A bibliography of software engineering terms, Data & Analysis Center for Software," Griffiss Air Force Base, NY, Rep. GLOS-1, Oct. 1979.

[19] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 156-173, June 1975.

[20] J. D. Gould, "Some psychological evidence on how people debug computer programs," *Int. J. Man-Machine Studies*, vol. 7, pp. 151-182, 1975.

[21] J. D. Gould and P. Drongowski, "An exploratory study of computer program debugging," *Human Factors*, vol. 16, no. 3, pp. 258-277, 1974.

[23] W. C. Hetzel, "An expermental analysis of program verification problem solving capabilities as they relate to programmer efficiency," *comput. Personnel*, vol. 3, no. 3, pp. 10-15, 1972.

[23] W. C. Hetzel, "An expermental analysis of program verification methods," Ph.D. dissertation, Univ. North Carolina, Chapel Hill, 1976.

[24] W. E. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," *IEEE Trans. Software Eng.*, vol. SE-3, no. 4, pp. 266-278, 1977.

[25] ——, "Algebraic program testing," *Acta Inform.*, vol. 10, 1978.

[26] ——, "An evaluation of the effectiveness of symbolic testing," *Software—Practice and Experience*, vol. 8, pp. 381-397, 1978.

[27] ——, "Functional program testing," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 162-169, Mar. 1980.

[28] ——, "Applicability of software validation techniques to scientific programs," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 3, pp. 307-320, July 1980.

[29] ——, "A survey of dynamic analysis methods," in *Tutorial: Software Testing & Validation Techniques*, 2nd ed., E. Miller and W. E. Howden, Eds. Washington, DC: IEEE Computer Society Press, 1981, pp. 209-231.

[30] S-S. V. Hwang, "An empirical study in functional testing, structural testing, and code reading inspection*," Dep. Comput. Sci., Univ. Maryland, College Park, Scholarly Paper 362, Dec. 1981.

[31] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, New York, Rep. IEEE-STD-729-1983, 1983.

[32] Z. Jelinski and P. B. Moranda, "Applications of a probability-based model to a code reading experiment," in *Proc. IEEE Symp. Computer Software Reliability*, New York, 1973, pp. 78-81.

[33] K. Jensen and N. Wirth, *Pascal User Manual and Report*, 2nd ed. New York: Springer-Verlag, 1974.

[34] W. L. Johnson, S. Draper, and E. Soloway, "An effective bug classification scheme must take the programmer into account," in *Proc. Workshop High-Level Debugging*, Palo Alto, CA, 1983.

[35] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*. Reading, MA: Addison-Wesley, 1979.

[36] P. R. McMullin and J. D. Gannon, "Evaluating a data abstraction testing system based on formal specifications," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-993, Dec. 1980.

[37] B. Meyer, "On formalism in specifications," *IEEE Software*, vol. 2, pp. 6-26, Jan. 1985.

[38] E. Miller and W. E. Howden, *Tutorial: Software Testing & Validation Techniques*, 2nd ed., IEEE Catalog No. EHO 180-0. Washington, D.C: IEEE Computer Society Press, 1981.

[39] H. D. Mills, "Mathematical foundations for structural programming," IBM Rep. FSL 72-6021, 1972.
[40] ——, "How to write correct programs and know it," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, 1975, pp. 363-370.
[41] G. J. Myers, "A controlled experiment in program testing and code walkthroughs inspections," *Commun. ACM*, pp. 760-768, Sept. 1978.
[42] ——, *The Art of Software Testing*. New York: Wiley, 1979.
[43] P. Naur, "Programming by action clusters," *BIT*, vol. 9, no. 3, pp. 250-258, 1969.
[44] T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment*," *J. Syst. Software*, vol. 4, pp. 289-300, 1984.
[45] D. J. Panzl, "Experience with automatic program testing," in *Proc. NBS Trends and Applications*, Nat. Bureau Standards, Gaithersburg, MD, May 28, 1981, pp. 25-28.
[46] H. Scheffe, *The Analysis of Variance*. New York: Wiley, 1959.
[47] R. W. Selby, "An empirical study comparing software testing techniques," in *Proc. Sixth Minnowbrook Workshop Software Performance Evaluation*, Blue Mountain Lake, NY, July 19-22, 1983.
[48] ——, "Evaluations of software technologies: Testing, CLEANROOM, and metrics," Ph.D. dissertation, Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1500, 1985.
[49] ——, "Combining software testing strategies: An empirical evaluation," in *Proc. Workshop Software Testing*, Banff, Alta., Canada, July 15-17, 1986, pp. 82-91.
[50] R. W. Selby, V. R. Basili, and F. T. Baker, "Cleanroom software development: An empirical evaluation," *IEEE Trans. Software Eng.*, vol. SE-13, pp. 1027-1037, Sept. 1987.
[51] R. W. Selby, V. R. Basili, J. Page, and F. E. McGarry, "Evaluating software testing strategies," in *Proc. Ninth Annu. Software Eng. Workshop*, NASA/GSFC, Greenbelt, MD, Nov. 1984.
[52] L. G. Stucki, "New directions in automated tools for improving software quality," in *Current Trends in Programming Methodology*, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice Hall, 1977.
[53] P. M. Valdes and A. L. Goel, "An error-specific approach to testing," in *Proc. 8th Annu. Software Eng. Workshop*, NASA/GSFC, Greenbelt, MD, Nov. 1983.
[54] D. M. Weiss and V. R. Basili, "Evaluating software development by analysis of changes: Some data from the software engineering laboratory," *IEEE Trans. Software Eng.*, vol. SE-11, no. 2, pp. 157-168, Feb. 1985.
[55] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience with path analysis and testing of programs," *IEEE Trans. Software Eng.*, vol. SE-6, no. 3, pp. 278-286, May 1980.
[56] M. V. Zelkowitz, R. T. Yeh, R. G. Hamlet, J. D. Gannon, and V. R. Basili, "Software engineering practices in the US and Japan," *Computer*, vol. 17, no. 6, pp. 57-66, June 1984.

Victor R. Basili, for a photograph and biography, see this issue, p. 1217.

Richard W. Selby (S'83-M'85) received the B.A. degree in mathematics and computer science from Saint Olaf College, Northfield, MN, in 1981 and the M.S. and Ph.D. degrees in computer science from the University of Maryland, College Park, in 1983 and 1985, respectively.

He is an Assistant Professor of Information and Computer Science at the University of California, Irvine. His research interests include methodologies for developing and testing software, techniques for empirically evaluating software methodologies, and software environments.

Dr. Selby is a member of the Association for Computing Machinery and the IEEE Computer Scoiety.