Evaluating Software Testing Strategies

Richard W. Selby, Jr. and Victor R. Basili
Department of Computer Science
University of Maryland
College Park, Maryland 20742
(301) 454-4247

Jerry Page
Computer Sciences Corp., Silver Spring, MD

Frank E. McGarry
NASA/GSFC, Greenbelt, MD

ABSTRACT

This study compares the strategies of code reading, functional testing, and structural testing in three aspects of software testing: fault detection effectiveness, fault detection cost, and classes of faults detected. Thirty two professional programmers applied the three techniques to three unit-sized programs in a fractional factorial experimental design. The major results of this study so far are the following. 1) Code readers detected more faults than did those using the other techniques, while functional testers detected more faults than did structural testers. 2) Code readers had a higher fault detection rate than did those using the other methods, while there was no difference between functional testers and structural testers. 3) Subjects testing the abstract data type detected the most faults and had the highest fault detection rate, while individuals testing the database maintainer found the fewest faults and spent the most effort testing. 4) Subjects of intermediate and junior expertise were not different in number or percentage of faults found, fault detection rate, or fault detection effort; subjects of advanced expertise found a greater number of faults than did the others, found a greater percentage of faults than did just those of junior expertise, and were not different from the others in either fault detection rate or effort. 5) Code readers and functional testers both detected more omission faults and more control faults than did structural testers, while code readers detected more interface faults than did those using the other methods.

Research supported in part by the National Aeronautics and Space Administration Grant NSG-5123 and the Air Force Office of Scientific Research Contract AFOSR-F49620-80-C-001 to the University of Maryland. Computer support provided in part by the facilities of NASA/Goddard Space Flight Center and the Computer Science Center at the University of Maryland.

1. Introduction

The processes of software testing and defect detection continue to challenge the software community. Even though the software testing and defect detection activities are inexact and inadequately understood, they are crucial to the success of a software project. The controlled study presented addresses the uncertainty of how to test software effectively. In this investigation, common testing techniques were applied to different types of software by a representative group of programming professionals. 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.

This paper gives an overview of the testing techniques examined, investigation goals, experimental design, and data analysis. The results presented are from a preliminary analysis of the data; a more complete analysis appears elsewhere [Selby 84, Basili & Selby 85].

2. Testing Techniques

To demonstrate that a particular program actually meets its specifications, professional software developers currently utilize many different testing methods. In functional testing, which is a "black box" approach [Howden 80], a programmer constructs test data from the program's specification through methods such as equivalence partitioning and boundary value analysis [Myers 79]. 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 [Howden 78, Howden 81], 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") [Stucki 77]. The structural coverage criteria used in this study was 100% 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 [Mills 72, Linger, Mills & Witt 79]. The code reader then compares this derived function and the specifications (the intended function).

2.1. Investigation Goals

The goals for this study are to compare the three common testing techniques of code reading, functional testing, and structural testing in terms of 1) fault detection effectiveness, 2) fault detection cost, and 3) classes of faults detected. An example research question in each of these goal areas is as follows. Which testing technique (code reading, functional testing, or structural testing) leads to the detection of the most faults? Which testing technique leads to the highest fault detection rate (#faults/effort)? Which testing techniques capture which classes of faults?

3. Empirical Study

Admittedly, the goals for this study 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 undertaken lends insights into their answers and into the merit and appropriateness of each of the techniques.

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 [Hetzel 76]. Thus, 1) the subjects chosen for the study were professional programmers with a wide range of experience, 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 provide a reasonable evaluation of the testing methods, and to facilitate the generalization of the results to other environments. Note that prior to this experiment, we conducted a similar testing study involving 42 advanced students from the University of Maryland [Basili & Selby 85].

The following sections describe the empirical study undertaken, including the selection of subjects, programs, and experimental design, and the operation of the study.

3.1. Subjects

The 32 subjects in the study were programming professionals from NASA and Computer Sciences Corporation. These individuals were mathematicians, physicists, and engineers that developed ground support software for satellites. They had familiarity with all three testing techniques, but used functional testing primarily. R. W. Selby conducted a three hour tutorial on the testing techniques for the subjects. The subjects were selected to be representative of three different levels of computer science expertise: advanced, intermediate, and junior. Several criteria were considered in the association of a subject with an expertise level, including years of professional experience, degree background, and their manager's suggested assignment. The individuals examined included eight advanced, eleven intermediate, and thirteen junior subjects; these groups had an average of 15.0, 10.9, and 6.1 years of professional experience, respectively, with an overall average of 10.0 (SD = 5.7) years.

3.2. Programs

The three FORTRAN programs used in the investigation were chosen to be representative of several different software types: a text formatter, a numeric abstract data type, and a database maintainer. The programs are summarized in Figure 1. The specifications for the programs and their source code appear in [Selby 84].

Figure 1. The programs tested.					
program	source lines	executable statments	cyclomatic complexity	#routines	#faults
text formatter	169	55	18	3	9
numeric data abstraction	147	48	18	9	7
database maintainer	365	144	57	7	12

There exists some differentiation in size among the programs, and they are a realistic size for unit testing. The first program is a text formatting program, which also appeared in [Myers 78]. A version of this program, originally written by [Naur 69] using techniques of program correctness proofs, was analyzed in [Goodenough & Gerhart 75]. The second 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 [McMullin & Gannon 80]. The third program is a maintainer for a database of bibliographic references. This program was analyzed in [Hetzel 76], and was written by a systems programmer at the University of North Carolina Computation Center.

3.3. Faults

The 28 faults in the programs comprise a reasonable distribution of faults that commonly occur in software [Basili & Weiss 82, Basili & Perricone 84]. All the faults in the database maintainer and the numeric abstract data type were made during the actual development of the programs. The text formatter contains a mix of faults made by the original programmer and faults seeded in the code. Note that this investigation involves only those types of faults occurring in the source code, not other types such as those in the requirements or specifications.

Two abstract classification schemes characterize the faults in the programs. One fault categorization method separates faults of omission from faults of commission. 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. An explanation of these classification schemes appeared in [Basili & Perricone 84], and the faults themselves are described in [Selby 84]. These two classification schemes are intended to distinguish among different reasons that programmers make faults in software development. The consistent application of the two schemes to the faults in the programs resulted in a mutually exclusive and exhaustive categorization; it is certainly possible that another analyst could have a different interpretation (see Figure 2).

Figure 2. Distribution of faults in the programs.			
	Omission	Commission	Total
Initialization	0	2	2
Computation	2	2	4
Control	2	4	6
Interface	2	11	13
Data	2	0	2
Cosmetic	0	1	1
Total	8	20	28

3.4. Experimental Design

The experimental design applied was a fractional factorial design [Cochran & Cox 50, Box, Hunter, & Hunter 78]. All of the subjects tested each of the three programs and used each of the three techniques. Of course, no one tested a given program more than once. The order of presentation of the testing techniques was randomized among the subjects in each level of expertise. A factorial analysis of variance (ANOVA) model supports the analysis of both the main effects (testing technique, software type, programmer expertise) and any interactions among the main effects.

The subjects examined in the study were random samples of programmers from the large population of programmers at each of the levels of expertise. If the samples examined are truly representative of the population of programmers at each expertise level, the inferences from the analysis can then be generalized across the whole population of individuals at each expertise level, not just across the particular subjects in the sample chosen.

3.5. Experimental Operation

The controlled study included five phases: 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. In the testing sessions, the individuals were requested to use the testing techniques to the best of their ability. The subjects' desire for the study's outcome to improve their software testing environment ensured reasonable effort on their part. Note that when the subjects were applying either functional or structural testing, they generated and executed their own test data; no test data sets were provided. At the end of each of the testing sessions, the subjects estimated the amount of time spent detecting faults and the percentage of the faults in the program that they thought were uncovered. The study concluded with a debriefing session for discussing the preliminary results and the subjects' observations.

4. Data Analysis

This section presents the data analysis according to the three goal areas discussed earlier.

4.1. Fault Detection Effectiveness

The first goal area examines the factors contributing to fault detection effectiveness. The following sections present the relationship of fault detection effectiveness to testing technique, software type, programmer expertise, and self-estimate of faults detected.

4.1.1. Testing Technique

The subjects applying code reading detected an average of 5.09 (SD = 1.92) faults per program, persons using functional testing found 4.47 (SD = 1.34), and those applying structural testing uncovered 3.25 (SD = 1.80); the subjects detected an overall average of 4.27 (SD = 1.86) faults per program. Subjects using code reading detected 1.24 more faults per program than did subjects using either functional or structural testing ($\alpha < .0001$, 95% c.i. 0.73 – 1.75).[1] Subjects using functional testing detected 1.11 more faults per program than did those using structural testing ($\alpha < .0007$, 95% c.i. 0.52 – 1.70). Since the programs each had a different number of faults, an alternate interpretation compares the percentage of the programs' faults detected by the techniques. The techniques performed in the same order when percentages are compared: subjects applying code reading detected 16.0% more faults per program than did subjects using the other techniques ($\alpha < .0001$, c.i. 9.9 – 22.1%), and subjects applying functional testing detected 11.2% more faults than did those using structural testing ($\alpha < .003$, c.i. 4.1 – 18.3%). Thus comparing either the number or percentage of faults detected, individuals using code reading observed the most faults, persons applying functional testing found the second most, and those doing structural testing uncovered the fewest.[2]

4.1.2. Software Type

The subjects testing the abstract data type detected an average of 5.22 (SD = 1.75) faults, persons testing the text formatter found 4.19 (SD = 1.73), and those testing the database maintainer uncovered 3.41 (SD = 1.66). The application of Tukey's multiple comparison reveals that subjects detected the most faults in the abstract data type, the second most in the text formatter, and the fewest faults in the database maintainer (simultaneous $\alpha < .05$). This ordering is the same for both number and percentage of faults detected.

4.1.3. Programmer Expertise

Subjects of advanced expertise detected an average of 5.00 (SD = 1.53) faults, persons of intermediate expertise found 4.18 (SD = 1.99), and those of junior expertise uncovered 3.90 (SD = 1.83). Subjects of intermediate and junior expertise were not statistically different in terms of either number or percentage of faults observed ($\alpha > .05$).

---

[1] The probably of Type I error is reported, the probability of erroneously rejecting the null hypothesis. The abbreviation "c.i." stands for confidence interval. The intervals reported are all 95% confidence intervals.

[2] Recall that the individuals used the following techniques: code reading by stepwise abstraction, functional testing using equivalence partitioning and boundary value analysis, and structural testing with 100% statement coverage criteria.

Individuals of advanced expertise detected both a greater number and percentage of faults than did those of junior expertise ($\alpha<.05$). Persons of advanced expertise detected a greater number of faults that did those of intermediate expertise ($\alpha<.05$), but the advanced and intermediate groups were not statistically different in percentage of faults detected ($\alpha>.05$).

4.1.4. Self-Estimate of Faults Detected

At the completion of a testing session, the subjects estimated 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 ($R = .57$, $\alpha<.0001$). Further investigation shows that individuals using certain techniques gave better estimates: code readers gave the best estimates (Pearson $R = .79$, $\alpha<.0001$), structural testers gave the second best estimates ($R = .57$, $\alpha<.0007$), and functional testers gave the worst estimates (no correlation, $\alpha>.05$). This observation suggests that the code readers were more certain of the effectiveness they had in revealing faults in the programs.

4.2. Fault Detection Cost

The second goal area examines the factors contributing to fault detection cost. The following sections present the relationship of fault detection cost to testing technique, software type, and programmer expertise.

4.2.1. Testing Technique

The subjects applying code reading detected faults at an average rate of 3.33 (SD = 3.42) faults per hour, persons using functional testing found faults at 1.84 (SD = 1.06) faults per hour, and those applying structural testing uncovered faults at a rate of 1.82 (SD = 1.24) faults per hour; the subjects detected faults at an overall average rate of 2.33 (SD = 2.28) faults per hour. Subjects using code reading detected 1.49 more faults per hour than did subjects using either functional or structural testing ($\alpha<.0003$, c.l. 0.75 – 2.23). Subjects using functional and structural testing were not statistically different in fault detection rate ($\alpha>.05$). The subjects spent an average of 2.75 (SD = 1.57) hours per program detecting faults. Comparing the total time spent in fault detection, the techniques were not statistically different ($\alpha>.05$). Thus, subjects using code reading detected faults at a higher rate than did those applying functional or structural testing, while the total fault detection effort was not different among the methods.

4.2.2. Software Type

The subjects testing the abstract data type detected faults at an average rate of 3.70 (SD = 3.26) faults per hour, persons testing the text formatter found faults at 2.15 (SD = 1.10) faults per hour, and those testing the database maintainer uncovered faults at a rate of 1.14 (SD = 0.79) faults per hour. Applying Tukey's multiple comparisons, the fault detection rate was higher in the abstract data type than it was for either the text formatter or the database maintainer, while the text formatter and the database maintainer were not statistically different (simultaneous $\alpha<.05$). The overall time spent in fault detection also differed among the programs. Subjects spent more time testing

the database maintainer than they spent on either the text formatter or the abstract data type, while the time spent on the text formatter and the abstract data type was not statistically different (simultaneous $\alpha < .05$). Thus, subjects uncovered faults at the fastest rate in the abstract data type, and spent the most time testing the database maintainer.

4.2.3. Programmer Expertise

Subjects of advanced expertise detected faults at an average rate of 2.36 (SD = 1.61) faults per hour, subjects of intermediate expertise found faults at 2.53 (SD = 2.48) faults per hour, and subjects of junior expertise uncovered faults at a rate of 2.14 (SD = 2.48) faults per hour. Programmer expertise level had no relation to either fault detection rate or total effort spent in fault detection (both $\alpha > .05$).

4.3. Characterization of Faults Detected

The third goal area focuses on determining what classes of faults are detected by the different techniques. An earlier section characterized the faults in the programs by two different classification schemes: omission or commission, and initialization, control, data, computation, interface, or cosmetic.

When the faults are partitioned according to the omission/commission scheme, a distinction surfaces among the techniques. Subjects using either code reading or functional testing observed more omission faults than did individuals applying structural testing, while there was no difference between code reading and functional testing. Since a fault of omission occurs as a result of some segment of code being left out ("omitted"), you would not expect structurally generated test data to find such a fault.

Dividing the faults according to the second fault classification scheme reveals a few distinctions among the methods. Subjects using code reading detected more interface faults than did those applying either of the other methods, while there was no difference between functional and structural testing. This suggests that code reading by abstracting and composing program functions across modules must be an effective technique for finding interface faults. Individuals using either code reading or functional testing detected more control faults than did persons applying structural testing. Recall that subjects applying structural testing determined the execution paths in a program and then generated test data that executed 100% of the program's statements. One would expect that more control path faults would be found by such an approach. However, structural testing did not do as well as the others in this fault class, suggesting the inadequacy of statement coverage criteria.

5. Preliminary Conclusions

This study compares the strategies of code reading, functional testing, and structural testing in three different aspects of software testing: fault detection effectiveness, fault detection cost, and classes of faults detected. Each of the three testing techniques showed merit in this evaluation. The investigation was intended to compare the different testing strategies in a representative testing situation, using professional programmers, different software types, and common software faults.

The major results of this study so far are the following. 1) Code readers detected more faults than did those using the other techniques, while functional testers detected more faults than did structural testers. 2) Code readers had a higher fault detection rate than did those using the other methods, while there was no difference between functional testers and structural testers. 3) Subjects testing the abstract data type detected the most faults and had the highest fault detection rate, while individuals testing the database maintainer found the fewest faults and spent the most effort testing. 4) Subjects of intermediate and junior expertise were not different in number or percentage of faults found, fault detection rate, or fault detection effort; subjects of advanced expertise found a greater number of faults than did the others, found a greater percentage of faults than did just those of junior expertise, and were not different from the others in either fault detection rate or effort. 5) Code readers and functional testers both detected more omission faults and more control faults than did structural testers, while code readers detected more interface faults than did those using the other methods.

A comparison of professional programmers using code reading with novice and junior programmers using the technique suggests a possible learning curve. In a testing study similar to this one, using a group of advanced students, code readers and functional testers were equally effective in fault detection while structural testers were either equally effective or inferior [Basili & Selby 85]. Also, the three techniques were not different in fault detection rate. Further comparison of this study with other testing studies, including [Hetzel 76, Myers 78, Hwang 81], appears in [Basili & Selby 85].

Investigations related to this work include studies of fault classification [Basili & Weiss 82, Johnson, Draper & Soloway 83, Ostrand & Weyuker 83, Basili & Perricone 84] and Cleanroom software development [Selby, Basili & Baker 84]. In the Cleanroom software development approach, techniques such as code reading are used in the development of software completely off-line. In the above study, 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.

This empirical study 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. However, valuable insights have been gained and additional areas of analysis and interpretation appear in [Selby 84, Basili & Selby 85].

6. Acknowledgement

The authors are grateful to the subjects from Computer Sciences Corporation and NASA Goddard for their enthusiastic participation in this study.

7. References

[Basili & Weiss 82]
V. R. Basili and D. M. Weiss, Evaluating Software Development by Analysis of Changes: The Data from the Software Engineering Laboratory*, Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1236, Dec. 1982..

[Basili & Perricone 84]
V. R. Basili and B. T. Perricone, Software Errors and Complexity: An Empirical Investigation, *Communications of the ACM* **27**, 1, pp. 42-52, Jan. 1984.

[Basili & Selby 85]
V. R. Basili and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep., 1985.

[Box, Hunter, & Hunter 78]
G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters,* John Wiley & Sons, New York, 1978.

[Cochran & Cox 50]
W. G. Cochran and G. M. Cox, *Experimental Designs,* John Wiley & Sons, New York, 1950.

[Goodenough & Gerhart 75]
J. B. Goodenough and S. L. Gerhart, Toward a Theory of Test Data Selection, *IEEE Trans. Software Engr.*, pp. 156-173, June 1975.

[Hetzel 76]
W. C. Hetzel, An Expermental Analysis of Program Verification Methods, Ph.D. Thesis, Univ. of North Carolina, Chapel Hill, 1976.

[Howden 78]
W. E. Howden, Algebraic Program Testing, *Acta Informatica* **10**, 1978.

[Howden 80]
W. E. Howden, Functional Program Testing, *IEEE Trans. Software Engr.* **SE-6**, pp. 162-169, Mar. 1980.

[Howden 81]
W. E. Howden, A Survey of Dynamic Analysis Methods, pp. 209-231 in *Tutorial: Software Testing & Validation Techniques, 2nd Ed.,* ed. E. Miller and W. E. Howden, 1981.

[Hwang 81]
S-S. V. Hwang, An Empirical Study in Functional Testing, Structural Testing, and Code Reading/Inspection*, Dept. Com. Sci., Univ. of Maryland, College Park, Scholarly Paper 362, Dec. 1981.

[Johnson, Draper & Soloway 83]
W. L. Johnson, S. Draper, and E. Soloway, An Effective Bug Classification Scheme Must Take the Programmer into Account, *Proc. Workshop High-Level Debugging*, Palo Alto, CA, 1983.

[Linger, Mills & Witt 79]
R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA, 1979.

[McMullin & Gannon 80]
P. R. McMullin and J. D. Gannon, Evaluating a Data Abstraction Testing System Based on Formal Specifications, Dept. Com. Sci., Univ. of Maryland, College Park, Tech. Rep. TR-993, Dec. 1980.

[Mills 72]
H. D. Mills, Mathematical Foundations for Structural Programming, IBM Report FSL 72-6021, 1972.

[Myers 78]
G. J. Myers, A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections, *Communications of the ACM*, pp. 760-768, Sept. 1978.

[Myers 79]
G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, 1979.

[Naur 69]
P. Naur, Programming by Action Clusters, *BIT* **9**, 3, pp. 250-258, 1969.

[Ostrand & Weyuker 83]
T. J. Ostrand and E. J. Weyuker, Collecting and Categorizing Software Error Data in an Industrial Environment, Dept. Com. Sci., Courant Inst. Math. Sci., New York Univ., NY, Tech. Rep. 47, August 1982 (Revised May 1983).

[Selby 84]
R. W. Selby, Jr., A Quantitative Approach for Evaluating Software Technologies, Dept. Com. Sci., Univ. Maryland, College Park, Ph. D. Dissertation, 1984.

[Selby, Basili & Baker 84]
R. W. Selby, Jr., V. R. Basili, and F. T. Baker, CLEANROOM Software Development: An Empirical Evaluation, Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1415, July 1984. (submitted to the *IEEE Trans. Software Engr.*)

[Stucki 77]
L. G. Stucki, New Directions in Automated Tools for Improving Software Quality, in *Current Trends in Programming Methodology,* ed. R. T. Yeh, Prentice Hall, Englewood Cliffs, NJ, 1977.