Programming Techniques
S. L. Graham, R. L. Rivest Editors

A Controlled Experiment in Program Testing and Code Walkthroughs/ Inspections

Glenford J. Myers
IBM Systems Research Institute

This paper describes an experiment in program testing, employing 59 highly experienced data processing professionals using seven methods to test a small PL/I program. The results show that the popular code walkthrough/inspection method was as effective as other computer-based methods in finding errors and that the most effective methods (in terms of errors found and cost) employed pairs of subjects who tested the program independently and then pooled their findings. The study also shows that there is a tremendous amount of variability among subjects and that the ability to detect certain types of errors varies from method to method.

Key Words and Phrases: software reliability, program verification, debugging, testing, code walkthroughs, code inspections, personnel selection
CR Categories: 4.6

Author's Address: 219 East 42nd Street, New York City, NY, 10017.

1. Introduction

The introduction of new programming methodologies and tools over the last few years has greatly outpaced our efforts to experiment with alternatives in order to analyze their differences and assess their benefits. One frequently finds statements in the literature such as "Method *A* was found to increase productivity by 39 percent over method *B*," but when one attempts to analyze the underlying evidence, one normally sees that methods *A* and *B* were used on two different projects with different people, different objectives, and literally hundreds of other differing characteristics. In other words, the statement is derived in many instances from a completely uncontrolled environment, rendering it at best misleading.

Unfortunately, controlled experiments in the area of software development tend to be costly and time consuming. Researchers have attempted to circumvent this by performing experiments using "cheap" labor (e.g. the captive audience in an introductory computer science course). However, although much of this work has provided useful insights, there exists the question of whether one can extrapolate, to a typical industrial environment, experimental results obtained from trainee programmers or programmers with only a few years of experience.

A third problem is the lack of data on program-testing methods, which is particularly alarming in light of the fact that program testing consumes approximately half of most organizations' development budgets. For instance, the idea of code walkthroughs or inspections [1] (a semiformal, noncomputer-based method of program testing performed by a team of individuals) has become popular, but no controlled data are available establishing that this method is more effective than traditional computer-based testing. As another example, there are differences of opinion as to whether a program tester should derive test cases based on an examination of the program's logic flow (e.g., to ensure that each conditional transfer has been exercised in all possible directions), but little data exist on which to base these opinions.

This paper presents results of an ongoing research project to answer some of these questions. The purpose of this research is to study, using controlled experiments with highly experienced programmers, the process of program testing, including the relative effectiveness and economics of different testing techniques and the factors that influence a programmer's testing effectiveness. This paper describes the results of an experiment in testing a PL/I program using three approaches and variations thereof: 1) Computer-based testing where the tester has access to only the program's specification, 2) computer-based testing where the tester has access to the program's specification and source-language listing, and 3) non-computer-based testing by teams of programmers employing the walkthrough/inspection method.

As mentioned earlier, research in this area has been

rather limited. The majority of the existing controlled experiments have been in the areas of programming-language design and the use of interactive versus batch-processing systems; the author of [11] reviews many of these efforts. The work of [6] is closely related to the experiment described herein: his experiment employed 39 subjects testing three PL/I programs using three methods: Testing by using only the specification, testing by using the specification and the program listing, and individual desk checking. However, his subjects were mostly students with little programming experience (an average of three years). Among other things, his work showed that the first two methods were equally effective and that the third method was significantly inferior.

Another relevant experiment [5] employed 18 inexperienced programmers testing five Cobol programs. The intent of the experiment was to determine if programmers were more effective in testing only one program at a time, or two or three programs. A small five-programmer experiment was conducted on the Safeguard Project [8], where errors were seeded into a program and individual code reading was used to locate the errors. The work of [3, 4] is also relevant, although it is in the area of debugging rather than testing. (Debugging is distinguished from testing in that testing is the process of showing that a program contains errors, but debugging is the process of finding the precise location of the error within the program [9].) Their experiments used ten moderately experienced Fortran programmers. The subjects were given four Fortran subroutines, told that each contained a one-statement error, and asked to locate the error using several debugging aids. Although their experiments produced several interesting observations, the applicability of their findings is questionable, since debuggers do not start with the knowledge that a program contains only a single one-statement error. Finally [7] has reported encouraging results in the use of symbolic execution as a testing technique.

2. The Experiment

The experiment discussed in this paper employed 59 highly experienced data processing professionals. The subjects' goal was to test (identify errors in) a small text-formatting program written in PL/I. The subjects were divided into three categories. Subjects in category A were asked to test the program by using a terminal and having only the specification from which to derive test cases. Subjects in category B operated in the same environment, but they were also given a copy of the program's listing from which they could derive additional test cases. The subjects were asked to work independently.

The subjects in category C were grouped into three-person teams. Each team was given the specification and listing of the program and asked to test the program using the manual walkthrough/inspection method.

In general, no time constraints were placed on the subjects. They were asked to test the program until they felt that they found all of the errors (if any). The subjects had no prior knowledge of the number or nature of the errors; they were simply told of a "suspicion that the program is not perfect." The only time constraint was that the walkthrough/inspection sessions were limited to 90 minutes, although the participants were given the materials in advance and could choose the amount of time to be spent in preparing for the session. Other than constraining the methods by the documentation given as described above, all participants were free to choose their testing strategies and methods. For instance, the people in category C were permitted to decide upon the inspection techniques to be used. Most used a combination of mentally walking test-cases through the program's logic and checking the logic for common errors.

Although the three methods were evaluated competitively, one should not draw the conclusion that the purpose of the experiment is to discourage use of the "less than best" methods. Indeed, given the magnitude of today's software reliability problem, as many different error-detection techniques as is feasible should be employed. In addition, one should be cautious in interpreting the results. For instance, the idea of code walkthroughs/inspections is generally recognized as a cost-effective technique in that the earlier that errors can be detected in a project, the lower the cost of repairing the errors and the higher the probability of repairing the errors correctly. This experiment does not address these broad issues; rather, it is a microscopic analysis of the relative effectiveness of the techniques.

2.1 The Subjects

The 59 subjects were students in a course on software reliability at the IBM Systems Research Institute. Their average number of years of experience in the computing field was 11; the range was from 7 to 20 years. Of the 59 participants, 49 were employed as programmers or program testers. The remaining 10 had programming experience, but were not considered to be "professional programmers"; their primary jobs were systems engineers, project managers, documentation writers, and electrical engineers. The subjects were considered to be above-average employees (a requisite for admission to the Systems Research Institute) and were judged to be highly motivated during the experiment. The motivation stemmed from the competitive nature of the situation and the knowledge that someone (the author) knew of the total number of errors in the program.

2.2 The Experimental Design

The crucial part of any experiment is, of course, designing the experiment to avoid biases in the outcome. The technique used in this experiment was to pretest the subjects. Part of the pretesting was done by questionnaire. Each subject was asked to rank his or her prior testing experience on a scale from 1 to 4 (1 = have never tested a program, 2 = have tested programs infrequently,

3 = have tested programs frequently, 4 = primary job is program testing); knowledge of PL/I on a scale from 1 to 3 (1 = could not understand a simple PL/I program, 2 = could understand a simple PL/I program, 3 = very experienced in PL/I); and experience with walkthrough/inspection techniques (0 = none, 1 = some). The subjects were also pretested by giving them a specification for an extremely simple program and asking them to write test cases for it. Their test cases were analyzed with respect to a set of errors in the program to obtain another measure of their testing abilities.

The goal was to place the subjects into the three categories such that no category was biased in terms of the above variables. Such placement proved to be infeasible because all subjects in categories B and C needed a PL/I rating of at least 2 (in order to read the program), but there were not enough subjects with this rating to balance all three categories. (Although they averaged 11 years of experience, many of the participants were Fortran, Cobol, APL, RPG, and assembly language programmers.) Also only 22 of the subjects reported prior walkthrough/inspection experience, but it was deemed necessary that each category-C team contain at least one subject with this experience.

The resultant design was to partition people into the three categories such that each category had the same average testing experience (based on both the questionnaire response and the performance on the pretest program). People with a PL/I rating of 1 were automatically placed into category A, and the teams in category C were organized such that each team contained at least one PL/I expert and one person with prior experience in walkthroughs. Hence the known biases were that category C had an average PL/I rating of 2.4, B a rating of 2.1, and A a rating of 1.5, and category C had an average walkthrough-experience rating of 0.6, B a rating of 0.3, and A a rating of 0.2. A smoothing effect came from the fact that the experiment was performed toward the end of the course using [9] as a text, and several lectures had been presented on program testing.

2.3 The Program

The PL/I program was based on an Algol program written, using techniques of program-correctness proofs, by Naur [10] and in which six errors were later discovered by Goodenough and Gerhart [2]. (They claim to have found seven, but their sixth error appears to be a duplicate of their third and fifth errors.) The program was translated into a three-procedure structured PL/I program totalling 63 statements, and a few changes were made to the original specification. The last four original errors were retained in the PL/I program, a few additional original errors were found, several errors were made during the conversion to PL/I, and a few typical errors ("typical" based on the experience of the author) were seeded into the program, bringing the total to 15 known errors. (None of the participants found any heretofore unknown errors.) The program is illustrated in Figure 1.

Each subject was given a specification of the program, and subjects in categories B and C were given a copy of the program listing (a compiler listing). The subjects were not told how many errors existed in the program or that the program had any errors. They were asked to find discrepancies (if any) between the program and its specification and to keep track of any errors found and the time expended. Subjects in categories A and B were also given instructions on how to invoke the program from a terminal. The author was present during the walkthrough/inspection sessions to play the role of the original programmer (i.e., the teams could ask questions about the program). The specification that was used is as follows.

Specification

Given an input text 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 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. & 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. A break in the input is reduced to a single blank or start of a new line in the output.

The input text is a single line entered from a terminal similar to an IBM 2741 having a carriage width of 130 characters. When the program is invoked, it prompts the terminal user for a line of input by typing a colon and then skipping to the next line and unlocking the keyboard. The user types the input line, followed by a / (end-of-text) and a carriage return. The program then formats the text and types it on the terminal.

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

The 15 known errors in the program are listed in Table I.

2.4 Data Collection

A questionnaire was distributed at the end of the experiment asking each participant to describe the errors found and to list the time spent in preparing for the test (e.g. studying the specification and designing test cases) and the time spent performing the test (i.e. executing and verifying test runs or participating in the walkthrough session). The raw data from each category is shown in

Fig. 1.

```
FORM: PROCEDURE OPTIONS (MAIN);
% DECLARE LINESIZE FIXED;
% LINESIZE = 31;
DECLARE (K, BUFPOS, FILL) FIXED DECIMAL (9),
      MAXPOS FIXED DECIMAL (9) INIT (LINESIZE),
      CW CHAR,
      BLANK CHAR INIT (' '),
      LINEFEED CHAR INIT ('$'),
      EOTEXT CHAR INIT ('/'),
      MOREINPUT BIT INIT ('1'B),
      BUFFER (LINESIZE) CHAR;
BUFPOS = 0;
FILL = 0;
DO WHILE (MOREINPUT);
  CALL GCHAR (CW);
  IF (CW = BLANK)|(CW = LINEFEED)|(CW = EOTEXT)
    THEN DO;
      IF (CW = EOTEXT) THEN MOREINPUT = '0'B;
      IF ((FILL + 1 + BUFPOS) < = MAXPOS)
        THEN DO;
            CALL PCHAR (BLANK);
            FILL = FILL + 1;
          END;
        ELSE DO;
            CALL PCHAR (LINEFEED);
            FILL = 0;
          END;
      DO K = 1 TO BUFPOS BY 1;
        CALL PCHAR (BUFFER (K));
      END;
      FILL = FILL + BUFPOS;
      BUFPOS = 0;
      END;
    ELSE IF BUFPOS = MAXPOS
        THEN DO;
            MOREINPUT = '0'B;
            DISPLAY ('WORD TO LONG');
          END;
        ELSE DO;
            BUFPOS = BUFPOS + 1;
            BUFFER (BUFPOS) = CW;
          END;
END;
CALL PCHAR (LINEFEED);
GCHAR: PROCEDURE (C);
DECLARE C CHAR,
      BUFFER (130) STATIC CHAR INIT ('Z'),
      INBUF CHAR (130),
      BCOUNT FIXED DECIMAL (3) STATIC INIT (1);
DECLARE SINPUT FILE STREAM;
IF (BUFFER (1) = 'Z')
  THEN DO;
      GET FILE (SINPUT) EDIT (INBUF) (A (130));
      IF (INDEX (INBUF, EOTEXT) = 0)
        THEN DO;
            DISPLAY ('NO END OF TEXT MARK');
            BUFFER (2) = EOTEXT;
          END;
        ELSE STRING (BUFFER) = INBUF;
    END;
  ELSE;
C = BUFFER (BCOUNT);
BCOUNT = BCOUNT + 1;
END;
PCHAR: PROCEDURE (C);
DECLARE C CHAR,
      OUTLINE (LINESIZE) CHAR STATIC INIT ((LINESIZE)
            (' ')),
      I FIXED DECIMAL (3) STATIC INIT (1);
DECLARE SOUTPUT FILE STREAM;
IF (C = LINEFEED)
  THEN DO;
      PUT FILE (SOUTPUT) SKIP EDIT (STRING (OUTLINE))
            (A (LINESIZE));
      OUTLINE = ' ';
      I = 1;
    END;
  ELSE DO;
      OUTLINE (I) = C;
      I = I + 1;
    END;
END;
END;
```

Tables II–IV. Categories A and B had 16 subjects each; category C consisted of nine teams of three people each. A "X" in a column means that that subject (or team) found the corresponding error.

Table V lists the time expended by each subject (or team of subjects for category C). The three numbers listed are the preparation, test, and total times in minutes.

3. Data Analysis

Before turning to statistics, several observations are apparent as a result of inspecting Tables II–IV. These are:

1. There is a tremendous amount of variability in the individual results. For instance, two people in category A found only one error, but five people found seven errors. The variability among student programmers is generally well known, but the high variability among these highly experienced subjects was somewhat surprising.
2. There is a tremendous amount of variability in the errors detected, particularly in categories A and B. Excepting errors 1, 2, 3, and 5, which were detected with some degree of regularity, the detection of individual types of errors varies widely from individual to individual.
3. The walkthrough method (category C) exhibits less variability, particularly in terms of the individual errors found. This suggests that this method is more predictable than the other methods.
4. The overall results are rather dismal; an observation that should not be surprising to anyone in the software engineering field. The mean number of errors found by all efforts was 5.1, or approximately a third of the known errors.
5. The inability to detect some of the seemingly obvious errors is alarming. For instance, error 1 (blank character at the beginning of the first line) was not detected by everyone. The probable reasons are either failing to inspect the output carefully or incorrectly assuming that the condition is not an error. One might expect errors 2, 3, and 5 to be caught by everyone, since these

Table I. Known errors in the text-formatting program.

Error	Symptom
1	A blank is printed before the first word on the first line unless the first word is 30-characters* long.
2	The program assumes that $ (not &) is the new-line character.
3	The program assumes that the line size is 31 characters, not 30 as stated in the specification.
4	If the first character of an input line is "Z", the line is ignored by the program.
5	The program does not condense successive new-line and blank characters.
6	The use of tab characters in the input text causes the 30-character* line limit to be exceeded.
7	A leading blank line is printed when the first word in the input text is 30 characters* long.
8	If an input line is entered without an end-of-text mark, the program prints a message and prompts for input. However, after the new input line is entered with an end-of-text mark, the program changes the first character in the output to "Z".
9	Spelling error in the message "WORD TO LONG."
10	After two successive omissions of end-of-text marks, the program prints a "Z" and terminates.
11	After issuing the word-too-long message, the program also prints whatever is residing in its output buffer.
12	Same situation as number 8, but if the second input line consists of only an end-of-text mark, the program prints 5 blank lines and the word-too-long message.
13	If a too-long word is used, the error message is printed in inconsistent places. For example, if the word would not appear in the first output line, the message is printed at the end of the previous output line, thus exceeding the maximum line size.
14	The program formats underscored words correctly, but it treats each underscored character as three characters when computing the line length.
15	The program's terminal buffer holds only 130 characters, but it is possible to enter a terminal line of more than 130 characters (e.g., by using the backspace key).

* Because of error 3, read as 31 instead of 30.

conditions are emphasized in the specification, but the expectations prove false. Not detecting error 9 is an example of the "eye seeing what it wants to see"; it is likely that everyone used a test case that caused this message to be issued, but most people failed to recognize the incorrect spelling in the message.

3.1 Comparing the Three Methods

Table VI summarizes some comparative statistics of the three methods. The first row (mean number of errors found) is the most important, but the nonparametric Kruskal-Wallis test indicates that there is no difference between the three methods (i.e., it is incorrect to conclude that method A is the worst). Rows 2–5 also indicate no large differences among the three methods. As a note, since the idea of group walkthroughs or inspections is relatively new, implying that people have less experience with this than with traditional testing methods, it is encouraging to see that this method held its own against the other methods.

In addition to this observation, two conclusions can be drawn from Table VI. One is that *none* of the methods, when used alone, is very good, since they detected only about a third of the errors in this small and simple program. An implication here is that the walkthrough/inspection technique should be viewed as a supplement to, rather than a replacement for, traditional testing methods. Another conclusion is that method C has a higher labor cost. The Kruskal-Wallis test shows that the difference in the mean man-minutes per error is highly significant; although the three methods were approximately equal in terms of the number of errors detected, the labor cost per error was much higher for the walkthrough/inspection method. On a macroscopic level, however, such differences in labor costs should prove to be insignificant when compared to the

Table II. Raw Data—Category A (Computer-Based Testing Using the Specification).

		Subjects															
		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Error	1	X	X		X		X		X	X		X		X	X	X	X
	2		X		X	X	X	X	X	X	X	X	X	X	X	X	X
	3	X	X		X		X	X		X	X	X			X	X	X
	4																
	5	X			X	X		X	X			X			X	X	X
	6					X											
	7			X				X								X	
	8		X		X	X		X			X	X			X		
	9				X		X		X			X					X
	10		X		X		X	X									X
	11		X									X					
	12																
	13		X														X
	14						X										
	15						X										
Total found		3	7	1	7	4	7	6	4	3	3	7	1	2	5	5	7
PL/I experience		1	1	2	2	2	2	2	2	1	1	1	2	1	1	2	1
Test experience		3	3	2	3	2	4	3	3	2	1	2	3	2	2	3	3
Walkthrough experience		0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0

labor normally expended in later stages (e.g. system test and maintenance). Machine time used in methods A and B was not considered because 1) it was insignificant in this experiment and 2) machine costs are dropping but labor costs are rising. As an aside, cost-per-error-found is usually an invalid measure when comparing results from two programs, because the program with the most errors tends to look the best. However, it is a valid measure here, since the three methods were used to test the same program.

3.2 Analysis of Individual Variations

Since the most visible result so far is the wide variation in individual performance, an analysis of the underlying causes is warranted. Correlation coefficients were calculated, across all methods and for each individual method, between the number of errors found and the subjects' PL/I experience, prior testing experience, performance on the pretest testing problem, walkthrough experience, total testing time, preparation time, test execution time, and the fraction of time spent in preparation. However, because correlation coefficients do not necessarily imply cause-effect relationships and they have little meaning when computed from small sample sizes, and because most of the calculated coefficients were not statistically significant and the significant ones were rather small, they provided no useful insight and are not included here.

On the other hand, correlation coefficients are occasionally useful in pointing out questions for future re-

Table III. Raw Data—Category B (Computer-Based Testing Using Specification and Code).

	Subjects															
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Error 1	X	X	X			X	X	X	X		X		X	X		X
2	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X
3	X	X	X	X		X	X	X	X	X	X		X			X
4			X	X	X				X				X			
5	X		X		X			X	X	X	X		X		X	
6																
7	X					X										X
8		X	X					X	X	X	X	X				X
9		X	X	X						X						
10		X	X		X	X		X	X			X	X			X
11										X						
12		X	X					X	X							
13	X															
14								X					X			
15	X												X			
Total found	7	7	9	3	4	5	3	8	8	6	5	3	8	2	2	6
PL/I experience	2	2	2	2	2	2	2	2	3	2	2	2	2	2	2	3
Test experience	3	2	3	3	3	1	3	3	3	3	3	2	2	2	4	4
Walkthrough experience	0	0	0	0	0	0	1	1	1	0	0	0	0	1	1	0

Table IV. Raw Data—Category C (Walkthrough/Inspection).

	Subjects								
	A	B	C	D	E	F	G	H	I
Error 1	X	X		X	X	X	X		
2	X	X	X	X	X	X	X	X	X
3	X	X	X	X	X	X	X		X
4	X	X	X	X	X	X	X	X	X
5	X	X		X	X		X	X	X
6									
7									X
8		X				X	X		X
9			X	X			X		X
10							X		
11					X				
12									
13							X		
14									
15									
Total found	5	6	4	6	6	5	9	3	7
PL/I experience	322	332	322	322	322	322	322	322	322
Test experience	433	333	432	432	332	332	333	433	332
Walkthrough experience	111	110	100	110	100	100	100	110	100

Table V. Times expended.

Category A	Category B	Category C
A 50/50/100	A 60/100/160	A 180/270/450
B 20/80/100	B 170/130/300	B 190/270/460
C 60/90/150	C 180/120/300	C 45/225/270
D 60/90/150	D 80/70/150	D 30/210/240
E 30/90/120	E 60/60/120	E 40/270/310
F 40/150/190	F 30/10/40	F 140/270/410
G 50/40/90	G 75/45/120	G 180/240/420
H 90/60/150	H 60/40/100	H 210/270/480
I 40/100/140	I 50/30/80	I 150/270/420
J 95/60/155	J 120/60/180	
K 20/50/70	K 60/55/115	
L 70/10/80	L 40/40/80	
M 20/20/40	M 120/100/220	
N 30/40/70	N 20/60/80	
O 85/45/130	O 60/60/120	
P 40/50/90	P 45/45/90	

Table VI. Comparative statistics.

	Method A	Method B	Method C
Mean no. of errors found	4.5	5.4	5.7
Variance	4.8	5.5	3.0
Median no. of errors found	4.5	5.5	6
Range of errors found	1–7	2–9	3–9
Cumulative errors found	13	14	11
Man-minutes per error	37	29	75

search. One example was a negative correlation (−0.58) between subjects' prior walkthrough/inspection experience and their performance in using method C. The sample is too small to draw conclusions, but further research seems warranted to explore the hypothesis that, because this method is more mentally taxing than the others, people are initially highly motivated but then tend to tire of it after multiple experiences.

3.3 Analysis of Error-Type Variations

Another way of using the data is to analyze the types of errors found by each method. Table VII shows, for each of the 15 errors, the percentage of subjects in each category that found the error and the percentage of total subjects that found each error (counting each category C team as one subject). The entries marked with an "*" appear to represent significant deviations.

The marked errors in the last column represent the errors that were infrequently detected by anyone. Errors 6, 14, and 15 would only be detected if the tester explored the use of special terminal characters (tab and backspace). Even if these special characters were used, error 15 would only be detected if the tester used the backspace character in conjunction with a very long input line.

Error 7 is only seen on a special input situation (when the first word is exactly equal to the output-line length) and when the tester carefully inspects the output (to spot the leading blank line). Error 11 is difficult to spot,

Table VII. Percentage of subjects finding each error type.

Error	Method A	Method B	Method C	All Methods
1	69	69	67	68
2	88	94	100	93
3	69	75	89	76
4	0*	31	100*	34
5	56	56	78	61
6	6	0	0	2*
7	19	19	11	17*
8	44	50	44	46
9	31	25	44	32
10	31	56*	11*	37
11	12	6	11	10*
12	0	25*	0	10*
13	12	6	11	10*
14	6	12	0	7*
15	6	12	0	7*

* These entries appear to represent significant deviations.

because the program's output buffer would contain blanks unless the "too long" word was not the first word in the input. Error 13 would only be spotted under similar conditions. Error 12 involved a special series of input conditions, and hence was infrequently detected.

The conclusion here is that the testers focused too much attention on "normal" test cases and insufficient attention on erroneous-input and special-case conditions. Notice that method C proved to be worse than average on these seven errors, implying that the walkthrough/inspection method focused too heavily on the program's logic at the expense of considering its input/output anomalies.

In examining method A, one sees that it did significantly poorer than the average on error 4. The program initialized the first character in its input buffer to "Z", and used this as an indication that its input buffer was empty. The error was relatively easy to spot if one examined the code, but it would only be detected by method-A subjects as a result of a test case such as "Zebras are animals./".

In examining method B, one finds that it was more likely to detect errors 10 and 12 than the other two methods. No explanation of this is offered. Method C was much more effective than the average in detecting error 4 (the "Z" in the buffer). This follows from the rigorous analysis of the program's logic that was performed by the teams employing method C. Method C was significantly poorer than the average on error 10. The error is extremely subtle and not likely to be discovered by reading the program. Since it is triggered by an easy-to-make input error, it was probably discovered by accident by the subjects using methods A and B.

4. Variations on the Experiment

Since the three methods were equally effective in finding errors (although method C was more costly), one might wish to determine if some combination of the methods might be more effective. For instance, is there

Table VIII. Comparative Statistics of Methods A–G.

	Method						
	A	B	C	D	E	F	G
Mean no. of errors found	4.5	5.4	5.7	7.3	8.3	7.2	7.6
Variance	4.8	5.5	3.0	3.6	2.8	3.4	4.3
Median no. of errors found	4.5	5.5	6	8	8	7.5	8
Range of errors found	1–7	2–9	3–9	4–10	6–11	3–10	5–10
Cumulative errors found	13	14	11	13	14	15	14
Mean man-minutes per error	37	29	75	34	34	37	75

any benefit from having two people independently test the program, using method A, and then pooling the errors that they find?

To study these combinations, four additional methods were defined. Method D incorporates two people independently testing the program using method A, where they pool their results when completed (and, of course, duplicate errors are not counted twice). Method E is similar, but the independent testers use method B. Method F is defined as employing two independent testers, one using method A and the other using method B. Method G is a combination of methods A and C; three people use method C (a walkthrough) and the fourth person independently uses method A to test the program.

Rather than running a separate set of experiments, data for these four methods were acquired by combining data from Tables II–V. That is, eight pairs of subjects were simulated for method D by pooling the results of the 16 subjects in Table II (pairing them in the order A-B, C-D, E-F, G-H, etc.). In a similar fashion, data were obtained for eight pairs of subjects using method E, 16 pairs of subjects using method F, and nine quadruplets of subjects using method G (pairing the first nine subjects in Table II to the nine groups in Table IV).

The comparative statistics for all seven methods are shown in Table VIII. As is indicated, the pooling of independent results (particularly in method E) reduces some of the unpredictability and risk (i.e. variance in errors found) associated with methods A and B. A Kruskal-Wallis test on the seven mean-number-of-errors-found now indicates that we reject the hypothesis that the means vary only by chance, implying that methods D-G are better than methods A-C. However a test of the means for methods D-G does not indicate any difference between these four methods, implying that we cannot conclude that method E is best.

Again there is a significant difference in the cost-per-error-found statistic; methods A, B, D, E, and F are significantly better than methods C and G. The interesting result, however, is the observation that the labor cost of methods D–F is approximately the same as that of methods A and B. In other words, the pooling of independent test results was not, as might be expected, more costly because of the detection of duplicate errors. The explanation for this is the large variability in the types of errors found by different individuals. This observation is perhaps the most significant result of the experiment, because it implies that a more cost-effective way to test a program is to employ two testers who work independently of one another (and based on a visual inspection of Table VIII, one might be swayed toward method E, although statistical analysis will not bear this conclusion out).

5. Discussion

One basic result of the experiment was that the walkthrough/inspection method had a higher labor cost than the other methods. This deserves further analysis to avoid being misconstrued. One reason for the popularity of the walkthrough/inspection method is that it gets people other than the program's programmer involved in the testing process, and there is reason to believe that programmers are relatively unsuccessful in testing their own programs (see Chapter 10 of [9] for a discussion of this). However in this experiment we did not compare the walkthrough to a programmer testing his or her program, but to a programmer testing someone else's program. Thus the walkthrough method is likely to be more effective than a programmer testing his or her own program, but less effective in terms of labor costs than a third party testing the program using computer-based methods.

One observation that was made during the experiment is that the walkthroughs and inspections focused too much on the logic of the program, at the expense of focusing attention on the input and output data. Perhaps this method could be improved by training programmers to focus their attention on the data handled by the program, rather than solely on the program's logic.

Comments were solicited from the participants and most of the participants felt that the experiment had a high educational value. At the end of the experiment, each participant was given a copy of the 15 known errors. In the normal environment, one does not have the opportunity to test a program and then receive immediate feedback on the errors overlooked, so perhaps exercises such as this should be incorporated into programming courses.

When asked to report on their testing techniques, most of the participants indicated that they focused their efforts on boundary and invalid-input conditions, but the results do not indicate that this really happened. The two reasons for the relatively poor error-detection rates

on this program appear to be: 1) The participants did not carefully compare the actual output produced by the program to the expected output. Thus many errors that were observable on the output listings were overlooked. 2) The participants focused too much of their attention on "normal" input conditions, and not enough on special cases and invalid-input situations.

Lastly, a few participants using the walkthrough method suggested that situations arise where it would be desirable—during the walkthrough session—to study special cases of interest by invoking the program from a terminal rather than by simulating them mentally. This idea of "computer assisted" walkthroughs seems interesting and warrants additional study; experiments are underway to do so. Likewise, the influence of testing tools deserves further study.

6. Conclusions

One result of the experiment is that the three original methods are equal in terms of error-detection capabilities, although the walkthrough method was not as cost-effective as the computer-based methods in a "unit testing" environment under the condition that the person doing the testing was not the programmer of the program. To repeat an earlier warning, this does not imply that the walkthrough technique, in and of itself, is not cost-effective; experience has shown the opposite to be the case.

Another result was that independent two-party tests tended to find more errors and, because of the large variability in errors found by each individual, were equally cost-effective as the single-person tests.

One finding is the significant variability among individuals, both in terms of the number and types of errors found. Given the high experience level of the participants and their participation in a course on software reliability, the variability was higher than expected. This variability could not be explained by correlating it with prior testing experience and other measured factors. As an example, an individual who found eight errors was employed as a documentation writer, but another individual employed as a program test specialist found only two errors. This variability implies that personnel selection for program-testing roles is of vital importance, but additional experimentation is necessary to determine the factors that contribute to high testing abilities.

Analysis of each of the errors showed that certain types of errors were very difficult to detect (independent of the method used) and that the ability to detect certain types of errors varied somewhat from method to method.

A possible criticism of the experiment is the small size of the program used. However, the results are believed to be applicable to larger programs. In particular, this experiment is analogous to the "unit testing" of individual subroutines or modules in a large program.

Received March 1977; revised October 1977

References

1. Fagan, M.E. Design and code inspections to reduce errors in program development. *IBM Syst. J. 15,* 3 (1976), 182–211.
2. Goodenough, J.B., and Gerhart, S.L. Toward a theory of test data selection. *IEEE Trans. Software Eng. SE-1,* 2 (1975), 156–173.
3. Gould, J.D. Some psychological evidence on how people debug computer programs. *Int. J. Man-Machine Studies 7,* 2 (1975), 151–182.
4. Gould, J.D., and Drongowski, P. An exploratory study of computer program debugging. *Human Factors 16,* 3 (1974), 258–277.
5. Griffith, P.F., and Henry, R.M. An investigatory study into human problem solving capabilities as they relate to programmer efficiency. *Comptr. Personnel 3,* 3 (1972), 10–15.
6. Hetzel, W.C. An experimental analysis of program verification methods. Ph.D. Th., U. of North Carolina, Chapel Hill, 1976.
7. Howden, W.E. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Trans. Software Eng. SE-3,* 4 (1977), 266–278.
8. Jelinski, Z., and Moranda, P.B. Applications of a probability-based model to a code reading experiment. Rec. 1973 IEEE Symp. Comptr. Software Reliability, IEEE, New York, 1973, pp. 78–81.
9. Myers, G.J. *Software Reliability: Principles and Practices.* Wiley-Interscience, New York, 1976.
10. Naur, P. Programming by action clusters. *BIT 9,* 3 (1969), 250–258.
11. Shneiderman, B. Experimental testing in programming languages, stylistic considerations and design techniques. Proc. AFIPS 1975 NCC, AFIPS Press, Montvale, N.J., 1975, pp. 653–656.