

How Well Do Professional Developers Test with Code Coverage Visualizations? An Empirical Study

Joseph Lawrence^{†‡}, Steven Clarke[†], Margaret Burnett[‡], Gregg Rothermel^{*}

[†]Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-8300

[‡]School of Electrical Engineering
and Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

^{*}Department of Computer
Science and Engineering
University of Nebraska-Lincoln
Lincoln, Nebraska 68588-0115

[‡]{lawrance,burnett}@eecs.oregonstate.edu, [†]stevencl@microsoft.com, ^{*}grother@cse.unl.edu

Abstract

Despite years of availability of testing tools, professional software developers still seem to need better support to determine the effectiveness of their tests. Without improvements in this area, inadequate testing of software seems likely to remain a major problem. To address this problem, industry and researchers have proposed systems that visualize “testedness” for end-user and professional developers. Empirical studies of such systems for end-user programmers have begun to show success at helping end users write more effective tests. Encouraged by this research, we examined the effect that code coverage visualizations have on the effectiveness of test cases that professional software developers write. This paper presents the results of an empirical study conducted using code coverage visualizations found in a commercially available programming environment. Our results reveal how this kind of code coverage visualization impacts test effectiveness, and provide insights into the strategies developers use to test code.

1. Introduction

As early as 1968, attendees of the NATO Software Engineering Conference recognized that inadequate testing of software was a problem [8]. Although decades have passed and advancements have been made, inadequate testing is a problem we still face today. In fact, in 2002, NIST estimated that inadequate software tests cost the economy up to \$59.5 billion per year, or about 0.6% of the US GDP [17].

Addressing the problem of inadequate software testing requires a definition of an adequate test. An *adequate test suite* is a set of test cases considered “good enough” by some criterion. Ideally, a test suite is “good enough” when

it exposes every fault and specifies the correct behavior of the program under test. Unfortunately, this criterion is impossible to measure without a complete specification and a list of all faults in the program. As Zhu, et al. [19] point out, one of the first breakthroughs in software testing was Goodenough and Gerhart’s idea of a measurable *test adequacy criterion*, which quantitatively specifies what constitutes an adequate test [9]. Test adequacy criteria provide means to assess the quality of a set of test cases without knowledge of a program’s faults or its specification. A set of test cases that meet a test adequacy criterion are said to provide *coverage* for the program under test.

Code coverage visualizations provide visual feedback of test adequacy [11]. Such visualizations show areas of code exercised by a set of test cases, and areas of code not executed by a set of test cases.

To our knowledge, no previous empirical studies of software testing visualizations have made the specific contributions we make here (see Section 5). This paper makes three contributions. First, this is the first study to our knowledge to investigate the effect of a code coverage *visualization* device on *professional developers’* effectiveness. Second, this is the first study to our knowledge to investigate the effect of a code coverage visualization device when the test adequacy criterion is *block coverage* (see Section 2). Third, this study sheds insights into *human strategy choices* in the presence of code coverage visualization devices.

2. Background

2.1. Test adequacy criteria

Research has produced many test adequacy criteria; [19] summarizes several of these, including the following:

Statement A set of test cases that executes every statement in a program provides statement coverage of the program.

Branch A set of test cases that executes all branches in a program provides branch coverage of the program.

Condition A set of test cases that exercises the true and false outcome of every subexpression in every condition in a program provides condition coverage of the program.

DU A set of test cases that exercises all pairs of data definitions and uses in a program provides definition-use (DU) coverage of the program.

Path A set of test cases that exercises all execution paths from the program's entry to its exit provides path coverage of the program.¹

In practice, of course, some coverage elements cannot be exercised given any program inputs and are thus *infeasible*, so coverage criteria typically require coverage only of feasible elements [5].

Tests serve only as an indirect measure of software quality, demonstrating the presence of faults, not necessarily the correctness of the program under test. Code coverage analysis simply reveals the areas of a program not exercised by a set of test cases. Even if a set of test cases completely exercises a program by some criterion, those test cases may fail to reveal all the faults within the program. For example, a test providing statement coverage may not reveal logic or data flow errors in a program.

2.2. Code coverage analysis

Code coverage analysis tools automate code coverage analysis by measuring coverage. Some coverage analysis tools also depict coverage visually, often by highlighting portions of code unexecuted by a test suite. Code coverage analysis tools include GCT,² Clover,³ and the code coverage tools built into Visual Studio.⁴

GCT measures statement coverage, branch coverage, condition coverage and several more coverage metrics not listed earlier.

Clover and Visual Studio, on the other hand, measure and *visualize* coverage. Although none of these tools support the additional coverage metrics that GCT supports, both tools measure and visualize “block” (statement and

¹Since the number of execution paths increases exponentially with each additional branch or loop, 100% path coverage is infeasible in all but the most trivial programs.

²<http://www.testing.com/tools.html>

³<http://www.cenqua.com/clover/>

⁴Visual Studio refers to Visual Studio 2005 Beta 2 Team System.

```
public static int IndexOf(string haystack, string needle)
{
    int matchIndex = -1;
    int needleIndex = 0;

    if (IsEmpty(haystack) || IsEmpty(needle))
        return needleIndex;

    for (int i = 0; i < haystack.Length; i++)
    {
        if (needle[needleIndex] == haystack[i])
        {
            needleIndex++;
            if (matchIndex <= 0)
                matchIndex = i;
            if (needleIndex == needle.Length)
                break;
        }
        else
        {
            needleIndex = 0;
            matchIndex = -1;
        }
    }

    return matchIndex;
}
```

Figure 1. Code coverage visualization:
Green: Executed, **Red: Unexecuted**,
Blue: Partial execution

branch) coverage. Clover and Visual Studio color the source code based on the sections of code executed by the *last collection of tests*. That is, the visualization resets every time a developer selects a new collection of tests to run; visualizations do not accumulate with each successive test run. Figure 1 shows that code highlighted in green represents code executed by the test run, whereas code in red represents unexecuted code (refer to the legend in Figure 1). Visual Studio also colors partially executed code with blue highlights. In practice, Visual Studio's coverage tool reserves blue highlights for short-circuited conditions or thrown exceptions.

3. Experiment

To gain insight into the effect code coverage visualizations (using block coverage) have on developers, we investigated the following research questions empirically:

RQ1: Do code coverage visualizations motivate developers to create more effective tests?

RQ2: Do code coverage visualizations motivate programmers to write more unit tests?

RQ3: Do code coverage visualizations lead developers into overestimating how many faults they found?

RQ4: What strategies do developers use in testing, with and without code coverage visualization?

Each of these research questions focuses on how code coverage visualizations affect professional software developers, and serves as a comparison to similar research on

end-user programmers using testing visualizations [16]. The first research question is important because code coverage visualizations are *designed* to motivate developers to write more effective tests by visualizing test adequacy. In addition, code coverage visualizations are supposed to improve developer efficiency or promote more productive testing strategies; we asked research questions two and four to address these points. On the other hand, code coverage visualizations could lead developers to overestimate their test effectiveness; thus, we asked research question three to address this concern.

3.1. Design & Materials

For this study, we recruited a group of 30 professional software developers from several Seattle-area companies. We required developers with two years of experience in the C# programming language, used C# in 70% of their software development, were familiar with the term “unit testing,” and felt comfortable with reading and writing code for a 90-minute period of time.

Our study was a between-subjects design in which we randomly assigned developers to one of two groups. We assigned 15 developers to the treatment group and 15 developers to the control group. The treatment group had code coverage visualizations available to them. The control group had no code coverage visualizations available to them.

Our study required a program for participants to test, so we wrote a class in C# containing a set of 10 methods.¹ We wanted to avoid verbally explaining the class to the developers, so we implemented methods likely to be familiar to most developers. These methods included common string manipulation methods and an implementation of square root. We included descriptive, but sometimes intentionally vague specifications with the methods in the program under test because we did not want the specifications to trivialize the task of writing tests. The program was too complex for participants to test exhaustively in an hour, but it gave us enough leeway for participants who were satisfied with their tests prematurely.

We required faults for our participants to uncover in the program we wrote. Following the lead of previous empirical studies of testing techniques [4, 10], we seeded the program we wrote with faults. We performed this seeding to cover several categories of faults, including faults that code coverage visualizations could reveal and faults that the visualization could miss. That said, we wanted developers to focus on *testing*, so the program generates no compilation errors.

¹The materials we developed and the data we collected are available online at: <ftp://ftp.cs.orst.edu/pub/burnett/TR.codeCoverage-professionalDevelopers.pdf>

To help create faults representative of real faults, we performed our seeding using a fault classification similar to published fault classification systems [1, 14]. Types of faults considered under these systems include mechanical faults, logical faults and omission faults; we also included faults caused by method dependencies and a red herring.² Mechanical faults include simple typographical errors. Logical faults are mistakes in reasoning and are more difficult to detect and correct than mechanical faults. Omission faults include code that has never been included in the program under test, and are the most difficult faults to detect [14].

In addition to the faulty program we wrote, we developed two questionnaires for our participants. We wrote the first questionnaire to assess the programming and unit testing experience of our participants, and to assess the homogeneity of the two groups. This first questionnaire also included measures of self-efficacy to serve as a baseline for the follow-up questionnaire. The follow-up questionnaire included measures of self-efficacy as well other measures to help us answer our research questions.

To assess our materials, we observed four developers in a pilot study. The pilot revealed that we needed to clarify some questions in our questionnaires. It also revealed that we needed to re-order the methods in the program under test. Some developers in the pilot study devoted most of the session to understanding and testing a single method. Since we were not interested in stumping developers, we sorted the methods roughly in ascending order of testing difficulty. We also added four test cases providing coverage for two methods under test to help us answer research question three.

3.2. Procedure

We conducted our experiment one person at a time, one-on-one for 90 minutes. We familiarized developers with the task they were to perform and had them complete the baseline questionnaire. After the orientation, we observed developers as they wrote unit tests for the methods we gave them. Finally, we gave them a follow-up questionnaire.

We trained developers to use a test development tool to create unit test cases for each method in the program we provided. We explained that clicking “Generate” in the test development tool (shown in Figure 2) produces unit test methods (shown in Figure 3) for every selected method. We described how unit tests pass parameters to a method, expect a result, and how the Assert class compares the expectations with the result of the method call. We stressed that we were looking for depth as opposed to breadth in the tests they created. That is, we asked developers to create

²A *red herring* is code that draws attention away from the actual faults.

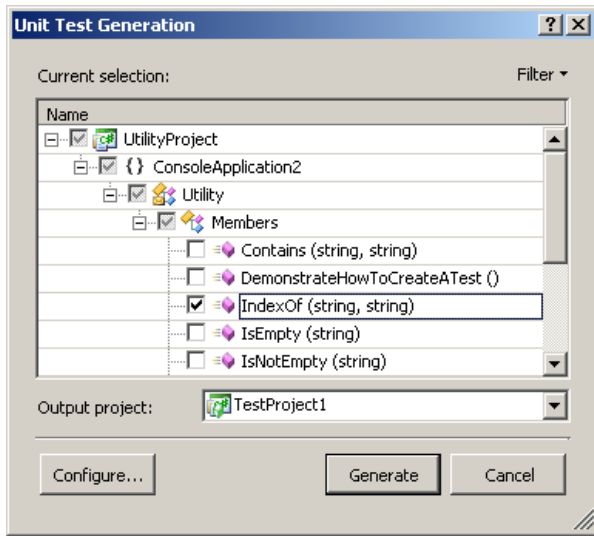


Figure 2. Test development tool

what they believed to be the most effective set of test cases for each method in the program under test before testing the next method.

We explained that we were interested in the tests that they wrote, not in the faults that they fixed. We told them not to fix faults unless they felt confident they could easily fix the fault. We stressed that test case failure was acceptable, but we also mentioned that a test case failure can reveal a problem with the test expectations. We answered any questions they had in relation to the tools or to their task. For participants in the treatment group, we described what the code coverage visualizations meant. After the orientation, we asked participants to complete the baseline questionnaire before we asked them to start writing tests.

While each developer wrote tests, we observed the developer behind a one-way mirror. We answered any questions developers had during the experiment through an intercom system. We had to answer some of the developers' questions carefully so as not to bias the result of the experiment. If developers asked us whether their test cases looked adequate, we told them: "Move on to the next method when you feel you have created what you believe to be the most effective set of test cases for this method." If developers asked for our feedback on the tests repeatedly, we told them: "Feel free to move on to the next method when you feel confident in the tests that you have created."

We recorded transcripts and video of each session. Using the transcripts, we made qualitative observations of developer behavior. When the time for the participants was up, we instructed the participants to complete a follow-up questionnaire. We archived the program and the test cases the developers wrote for our data analysis.

```

/// <summary>
/// A test case for IndexOf (string, string)
/// </summary>
[TestMethod()]
public void IndexOfTest()
{
    // TODO: Initialize to an appropriate value
    string haystack = null;
    string needle = null;

    int expected = 0;
    int actual;

    actual = TestProject1.
    ConsoleApplication2_UtilityAccessor.
    IndexOf(haystack, needle);

    Assert.AreEqual(expected, actual,
    "ConsoleApplication2_Utility.Indexof did not"
    + " return the expected value.");

    Assert.Inconclusive("Verify the correctness"
    + " of this test method.");
}

```

Figure 3. Generated test method

3.3. Threats to Validity

Many other studies of software testing involve software engineering students, and are conducted in large groups. Since such studies are run on populations of students, the results of these studies do not necessarily generalize to developers in industry (external validity). Such studies are also plagued by the problem of participants revealing the details of the experiment to other students (internal validity). To avoid these threats, we recruited software developers from several companies and ran the study one participant at a time. Participants were given non-disclosure agreements as a condition for participating in this study. Although the possibility exists that participants could have discussed the study with colleagues who also participated in the study, almost all participants did not know each other. Because we studied professional developers individually, subjects were aware that we were observing them behind a one-way mirror, which may have changed their behavior.

We intended to compare the results of the baseline questionnaire with the follow-up questionnaire to determine how code coverage visualizations affected self-efficacy. Because we randomly assigned participants to one of two groups, we anticipated that the control group would not differ from the code coverage group before the testing task. Although the code coverage and control groups did not differ in their programming or unit testing experience according to the baseline questionnaire, we noticed the control group assessed their own efficacy significantly higher than

Table 1. Descriptive statistics

Metric	Control	Treatment
Logic errors found	39	43
Omissions found	39	28
Dependency faults found	17	11
Typos found	2	7
Precision errors found	6	2
Overflows found	3	1
Red herrings found	1	1

the code coverage group. Because we gave the baseline questionnaire after explaining how to use the testing tools (which included a tutorial on code coverage for the treatment group), it is possible that either our sample is not random, or the tutorial itself had an effect on developers' sense of efficacy. To avoid this threat to validity, we do not compare the results of the baseline questionnaire with the follow-up questionnaire.

We seeded the program that developers tested with faults. Although we made an effort to produce "natural" faults, most of the errors in the program were there intentionally. Also, the program housing these seeded faults may not have corresponded to the kind of programs our participants were accustomed to writing. For example, some of our participants were accustomed to writing database applications, web applications, or GUI-based applications.

4. Results and Discussion

Table 1 provides descriptive statistics from our study's data, and the next subsections present the hypotheses that we investigated using statistical methods. In the following subsections we describe these results in relation to each of our research questions in turn.

4.1. RQ1: Test effectiveness

To investigate how code coverage visualizations influenced test effectiveness, we compared the number of faults revealed between each group. The null hypothesis is:

H1: The number of faults revealed between the control group and the code coverage group does not differ.

To test H1, we ran a two-sample unpaired t-test on the code coverage group ($n = 15$) and the control group ($n = 15$), $t = -0.561$, $p = 0.58$. Thus, the test provided no evidence to suggest a difference in the number of faults revealed between the control group and the treatment group.

Discussion. Code coverage visualizations using block coverage did not affect the number of faults developers found in the program we provided in the time provided:

developers in the code coverage group did not differ from developers in the control group in terms of the number of faults found. This result is consistent with research done on software engineering students that compared the effectiveness of a reading technique with a structural testing technique using similar coverage criteria [13, 18].

4.2. RQ2: Amount of testing

To compare the amount of tests the two groups wrote, we looked at the mean and variance in the number of test cases between the control group and the treatment group.

H2: There is no difference in the number of test cases between the control group and the treatment group.

H3: There is no difference in the variance of test cases between the control group and the treatment group.

Figure 4 displays the distribution of the number of test cases written per group using a box plot.¹ The boxes in Figure 4 suggest that developers using code coverage visualizations produced slightly fewer test cases and varied less in the amount of test cases than developers without code coverage visualizations.

To test H2, we ran a two-sample unpaired t-test, $t = -0.89$, $p = 0.38$. Thus, the test provided no evidence to suggest the number of test cases between the control group and treatment group differed.

To test H3, we ran a one-sided F test, $F = 0.25$, $p = 0.015$. The ratio of variances in the number of test cases between groups was not equal to one. Thus, evidence suggests that code coverage visualizations reduced the variability in the amount of tests cases developers wrote.

Discussion. The reduced variability in the amount of test cases suggests that code coverage visualizations were powerful enough to affect the developers' testing behavior. With code coverage visualizations, developers stopped testing when they achieved coverage, and wrote more tests cases when they did not achieve coverage. Since test adequacy criteria are supposed to make people continue testing until they achieve coverage and then stop, the testing visualization's closing up of the variance suggests that it performed exactly as it should have.

In contrast, developers in the control group had no cues about the effectiveness of their tests. Such developers had only their own individual talents to draw on in determining the effectiveness of their tests, which probably explains the wide variability in the control group.

¹A boxplot is a standard statistical device for representing data distributions. In the boxplots presented in this paper, each data set's distribution is represented by a box. The box's height spans the central 50% of the data, and its ends mark the upper and lower quartiles. The horizontal line partitioning the box represents the median element in the data set. The vertical lines attached to the ends of the box indicate the tails of the distribution. Data elements considered outliers are depicted as circles.

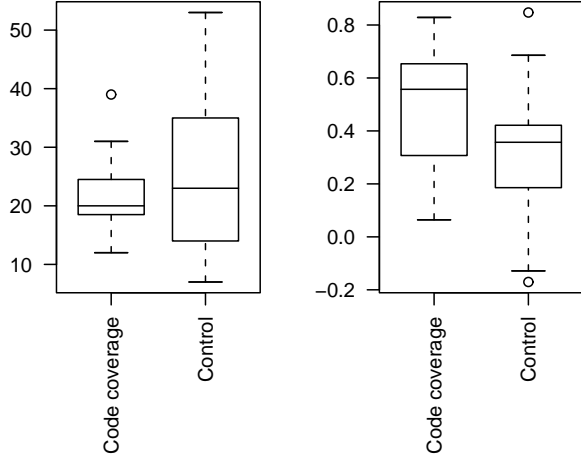


Figure 4. RQ2: Test cases written by group (left), RQ3 (H4): Overestimation by group (right)

4.3. RQ3: Overestimation of test effectiveness

Developers commonly determine when code is ready to ship based on their estimates of the correctness of the code. Thus, to test the possibility that code coverage visualizations using block coverage criteria lead developers into overestimating the correctness of the program, we asked developers to estimate the percentage of faults that they found. Using our measure of test effectiveness, we devised the following two formulas to measure the actual percentage of faults found, where *total faults* is the number of faults in the program (35), and *total faults possible* is the total number of faults in the methods each developer tested. Then, using each measure of the actual percentage of faults found, we compared overestimation by group.

H4: Overestimation did not differ by the following:

$$\text{estimate} - \left(\frac{\text{test effectiveness}}{\text{total faults}} \right)$$

H5: Overestimation did not differ by the following:

$$\text{estimate} - \left(\frac{\text{test effectiveness}}{\text{total faults possible}} \right)$$

Figure 4 displays the distribution of overestimation measured in H4 by group. The box plots suggest that developers in the code coverage group overestimated their effectiveness more than developers in the control group.

To test H4, we ran a two-sample one-sided unpaired t-test, $t = 1.96, p = 0.03$. However, we used a non-parametric test for H5 because the metric did not follow a normal distribution by the Shapiro-Wilk normality test at the 0.10 level ($W = 0.9307, p = 0.05134$). To test H5, we ran the Mann-Whitney test, $U = 160, p = 0.026$. The results of H4 and H5 both provide evidence to suggest

that developers using code coverage visualizations overestimated their effectiveness more than developers without code coverage visualizations.

Discussion. The overestimation we observed suggests that developers using code coverage visualizations interpreted the green coloring as correct code.

Comparing our results with the results of research using the stronger definition-use test adequacy criterion [16] suggests that the choice of test adequacy criterion can be critical. Since it was trivial to achieve complete block coverage of the program we provided, it was trivial for developers who may equate coverage with correctness to overestimate their test effectiveness. Using stronger coverage criteria, the task of writing tests that achieve complete coverage becomes less trivial. That is, the stronger the coverage criteria, the harder the testing task, and the less coverage each test provides. Thus, for developers who may equate coverage with correctness, stronger coverage criteria would imply lesser overestimations of test effectiveness. This implies that the choice of test adequacy criterion may play a critical role in developers’ estimates of their own effectiveness.

4.4. RQ4: Testing strategies

Developers needed to understand the code we gave them and also create, organize and evaluate their test methods and test cases. As we observed developers, we identified several strategies developers used in each step of the testing process, summarized in Table 2. We also classified strategies developers used in response to a test run as productive or counterproductive (see Table 3).

We performed Fisher’s exact test to test whether code coverage visualizations influenced the test follow-up strategies developers used, $p = 0.2135$. The test provided no evidence to suggest an association between code coverage visualizations and whether developers followed a productive or counterproductive strategy.

Discussion. Recall that all participants were experienced professional developers. Yet, almost one-third of them spent much of their time following counterproductive strategies listed in Table 3. This suggests that even professional developers need assistance to avoid counterproductive strategies. Some counterproductive strategies, such as changing the parameters of a method under test or deleting failed tests amounted to developers throwing away their work. Other counterproductive strategies, such as “fixing” generated test framework code, skipping tests for similar methods, or changing the expectation or specification to match the behavior suggested that some developers didn’t understand where the fault was located. Still other counterproductive strategies wasted developer time, such as creating duplicate tests or repeating the last test run

Table 2. Testing strategies (number of developers)

Strategy	Control	Treatment
Test creation process		
• Batch	9	7
• Incremental	6	8
Test run process		
• Batch	7	8
• Incremental	8	7
Test creation choices		
• Copy/paste	8	7
• Change test case	2	1
• Test development tool	5	5
• Write tests from scratch	0	2
Test organization		
• One test case / test method	9	11
• Many test cases / test method	4	2
• Parameterized test method	2	2
Test follow-up		
• Productive	9	12
• Counterproductive	6	3
Code understanding^a		
• Read specification	15	15
• Read program under test	15	15
• Execute code mentally	15	15
• Debug program under test	9	7
• Examine code coverage	0	15

^aDevelopers employed several of these strategies simultaneously.

without making any changes. Some common strategies, such as copying and pasting test code or debugging code, carried risks or consumed time but we did not classify these strategies as counterproductive.

5. Related Work

Empirical studies of software testing compare testing techniques or test adequacy criteria [12]. Some empirical studies of software testing use faulty programs as subjects; others use humans as subjects [3]. A few empirical studies have also studied the effect of visualizations [11, 16]. This is the first study to our knowledge that focuses exclusively on the effect of code coverage visualizations using block coverage on professional software developers. Although we are not aware of any studies exactly like ours, the results of previous empirical studies of software testing have guided our study design, our hypotheses, and gave us a basis to compare our results with previous work.

We based our study design and hypotheses on empirical studies of humans testing software. Studies of humans include [2, 13, 16, 18]. In each of these studies, like our own study, experimenters gave subjects faulty programs

Table 3. Test follow-up strategies

Productive	Counterproductive
Review, modify or fix method under test	Review, modify or “fix” generated test framework
Create new test	Change test parameters
Change expectations to match the specification	Change expectation or spec to match wrong behavior, leave specification as-is
Review assertion failed messages	Comment out or delete tests that fail
Create similar tests for other methods	Create duplicate tests, skip tests for similar methods
Note the test results and write next test	Repeat last test run without making any changes

and compared the subjects’ test effectiveness based on the testing technique. In [2, 13, 18], experimenters compared statement and branch coverage with other testing and verification techniques and measured the number of faults each subject found. Their results corresponded to our own results, which revealed that subjects were equally effective at isolating faults regardless of test technique. Despite the similarity in test effectiveness, Wood [18] noted that the relative effectiveness of each technique depends on the nature of the program and its faults. This result corroborates our own observation that the code coverage group discovered fewer omissions than the control group.

Although our results were consistent with results of experiments in which statement and branch coverage were used as test adequacy criteria, we were *not* consistent with the results of [16]. In [16], subjects using a definition-use coverage visualization performed significantly more effective testing, were less overconfident and more efficient than subjects without such visualizations.

Other empirical studies explain why our results differed radically from [16]. Studies of faulty programs have given us a basis to compare our results with previous work [6, 7, 10, 11]. These studies have shown that definition-use coverage can produce test suites with better fault-detection effectiveness than block coverage [10]. Definition-use coverage is a stronger criterion (in terms of subsumption) than statement or branch coverage [15]. Thus, comparing our results with the results of [16] imply that the test adequacy criterion is critical to the outcome of the study.

6. Conclusion

Code coverage visualizations using block coverage neither guided developers toward productive testing strategies, nor did these visualizations motivate developers to write

more tests or help them find more faults than the control group. Nevertheless, code coverage visualizations did influence developers in a few important ways. Code coverage visualizations led developers to overestimate their test effectiveness more than the control group. Yet, these same visualizations reduced the variability in the number of test cases developers wrote by changing the standard developers used to evaluate their test effectiveness.

Thus, the true power of testing visualizations lies not only with the faults that visualizations can highlight; it also lies in how visualizations can change how developers think about testing. Testing visualizations guide developers to a particular standard of effectiveness, so if we want developers to test software adequately, we must ensure that the coverage criteria we choose to visualize leads developers toward a good standard of test effectiveness.

7. Acknowledgments

We thank Curt Becker, Wayne Bryan, Monty Hammon-tree, Ephraim Kam, Karl Melder and Rick Spencer for observing the study with us. We thank Curt Becker and Jesse Lim for developing the logging and reporting tools essential to running this study. We thank Madonna Joyner, I-Pei Lin, and Thomas Moore for their assistance in running the study. We especially thank the developers who took the time to participate in our study.

References

- [1] C. Allwood, "Error detection processes in statistical problem solving." *Cognitive Science*, vol. 8, no. 4, pp. 413–437, 1984.
- [2] V. R. Basili and R. W. Selby, "Comparing the effectiveness of software testing strategies," *IEEE Trans. Software Engineering*, vol. 13, no. 12, pp. 1278–1296, 1987.
- [3] L. Briand and Y. Labiche, "Empirical studies of software testing techniques: Challenges, practical strategies, and future research," *WERST Proceedings/ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 5, pp. 1–3, 2004.
- [4] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.
- [5] P. Frankl and E. Weyuker, "An applicable family of data flow criteria," *IEEE Trans. Software Engineering*, vol. 14, no. 10, pp. 1483–1498, 1988.
- [6] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of branch testing and data flow testing," *IEEE Trans. Software Engineering*, vol. 19, no. 8, pp. 774–787, 1993.
- [7] P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness," in *SIGSOFT '98/FSE-6: 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Lake Buena Vista, Florida, USA, 1998, pp. 153–162.
- [8] B. A. Galler, "ACM president's letter: NATO and software engineering?" *Comm. ACM*, vol. 12, no. 6, p. 301, 1969.
- [9] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," in *International Conf. Reliable Software*, 1975, pp. 493–510.
- [10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *ICSE '94: 16th International Conf. Software Engineering*, 1994, pp. 191–200.
- [11] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE '02: 24th International Conf. Software Engineering*, 2002, pp. 467–477.
- [12] N. Juristo, A. M. Moreno, and S. Vegas, "Reviewing 25 years of testing technique experiments," *Empirical Software Engineering*, vol. 9, no. 1-2, pp. 7–44, 2004.
- [13] E. Kamsties and C. M. Lott, "An empirical evaluation of three defect-detection techniques," in *Fifth European Software Engineering Conference*, W. Schafer and P. Botella, Eds. Lecture Notes in Computer Science Nr. 989, 1995, pp. 362–383.
- [14] R. Panko, "What we know about spreadsheet errors." *Journal of End User Computing*, pp. 15–21, 1998.
- [15] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Software Engineering*, vol. 11, no. 4, pp. 367–375, 1985.
- [16] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel, "WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation," in *ICSE '00: 22nd International Conf. Software Engineering*, 2000, pp. 230–239.
- [17] RTI, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Tech. Rep. 02-3, 2002. [Online]. Available: <http://www.nist.gov/director/prog-ofc/report02-3.pdf>
- [18] M. Wood, M. Roper, A. Brooks, and J. Miller, "Comparing and combining software defect detection techniques: A replicated empirical study," in *ESEC '97/FSE-5: 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 1997, pp. 262–277.
- [19] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, 1997.