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

Joseph Lawrance[†‡], 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

A technical report submitted to the
School of Electrical Engineering and Computer Science
Oregon State University

Technical Report # CS05-60-02

Completed March 30, 2005

ACKNOWLEDGMENTS

We thank Monty Hammontree, Curt Becker, Wayne Bryan, 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 this study.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

LIST OF TABLES

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

1. INTRODUCTION

As early as 1968, attendees of the NATO Software Engineering Conference recognized that inadequate testing of software was a problem [8]. Decades have passed, advancements have been made, and now software testing is a widespread practice that indirectly measures the quality of software under test. Yet, inadequate testing is a problem we still face. In 2002, NIST estimated that inadequate software testing 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 adequate testing. 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. [20] 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 the faults within a program or the specification for the program. A set of test cases that meet a test adequacy criterion are said to provide adequate testing 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 Chapter 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 Chapter 2). Third, this study reveals 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; [20] 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.

Block A set of test cases that executes all branches and all non-branching sequences of statements in a program provides block 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*. For example, in an if-then-else statement

[1]Since 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.

with a condition that always evaluates to true, it is impossible to execute the else branch. 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 neither of these tools support the additional coverage metrics that GCT supports, both tools measure and visualize "block" (statement and 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

---

[2] http://www.testing.com/tools.html

[3] http://www.cenqua.com/clover/

[4] *Visual 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 2.1. Code coverage visualization: Green: Executed, Red: Unexecuted, Blue: Partial execution

collection of tests to run; visualizations do not accumulate with each successive test run. Figure 2.1 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 2.1). 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. MATERIALS AND METHODS

To gain insight into the effect code coverage visualizations using block coverage have on professional software 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 have revealed?

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, who used C# in 70% of their software development, who were familiar with the term "unit testing," and who 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 (given in Appendix B). 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 [10, 4], we seeded the program 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 would miss. We wanted developers to focus on *testing*, so the program generates no compilation errors.

To help create faults representative of real faults, we performed our seeding using a fault classification similar to published fault classification systems [14, 1]. 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.[1] 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]. Table 3.1 summarizes the faults in the program.

In addition to the faulty program we wrote, we developed two questionnaires for our participants (given in Appendix D and Appendix E). 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

---

[1] A *red herring* is code that draws attention away from the actual faults.

added four test cases providing coverage for two methods under test to help us answer RQ3 (see Appendix C).

TABLE 3.1.: Faults in Program.cs

boolean Contains(string, string)	
Omission	Throws NullReferenceException.
Dependency	Needle contained in empty haystack.
Dependency	Partial matches found in haystack end.
Dependency	Needle contained in shorter haystack.
int IndexOf(string, string)	
Logic error	Fails to throw exception on null needle.
Logic error	Needle found in empty haystack.
Omission	Partial matches found in haystack end.
Logic error	Needles match incorrectly at beginning.
Omission	Matches needles longer than haystack.
boolean IsEmpty(string)	
Logic error	Throws exception on null reference.
boolean IsNotEmpty(string)	
Logic error	Throws exception on null reference.
Logic error	Returns true for empty strings.
double SquareRoot(double)	
Omission	Loops infinitely on negative numbers.
Precision	Loops infinitely on non-square numbers.
Overflow	Loops infinitely on large numbers.
string SubstringAfter(string, string)	
Omission	Throws NullReferenceException.
Omission	Separator not in string throws exception.
Dependency	Empty string throws an exception.
Dependency	Partial match at end throws an exception.
Dependency	Incorrect substring after beginning.
Dependency	Long separator throws an exception.
string SubstringBefore(string, string)	
Omission	Throws NullReferenceException.

Omission	Separator not in string throws exception.
Dependency	Wrong substring from partial end match.
Dependency	Incorrect substring before beginning.
Dependency	Long separator throws an exception.
string ToLower(string)	
Typo	"O" becomes "O", zero becomes "o"
Dependency	"A" remains capitalized.
string ToUpper(string)	
Typo	"yz" transposed to "ZY" in output
Typo	"o" becomes zero "0", not "O"
Dependency	"a" remains in lower case.
string Translate(string, string, string)	
Omission	Throws NullReferenceException.
Logic error	First character not replaced.
Omission	Retains absent replacement characters.
Red herring	Extra code contributes no functionality.

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 (using the script given in Appendix A) 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 3.1) produces unit test methods (shown in Figure 3.2) for every selected method. We described how unit

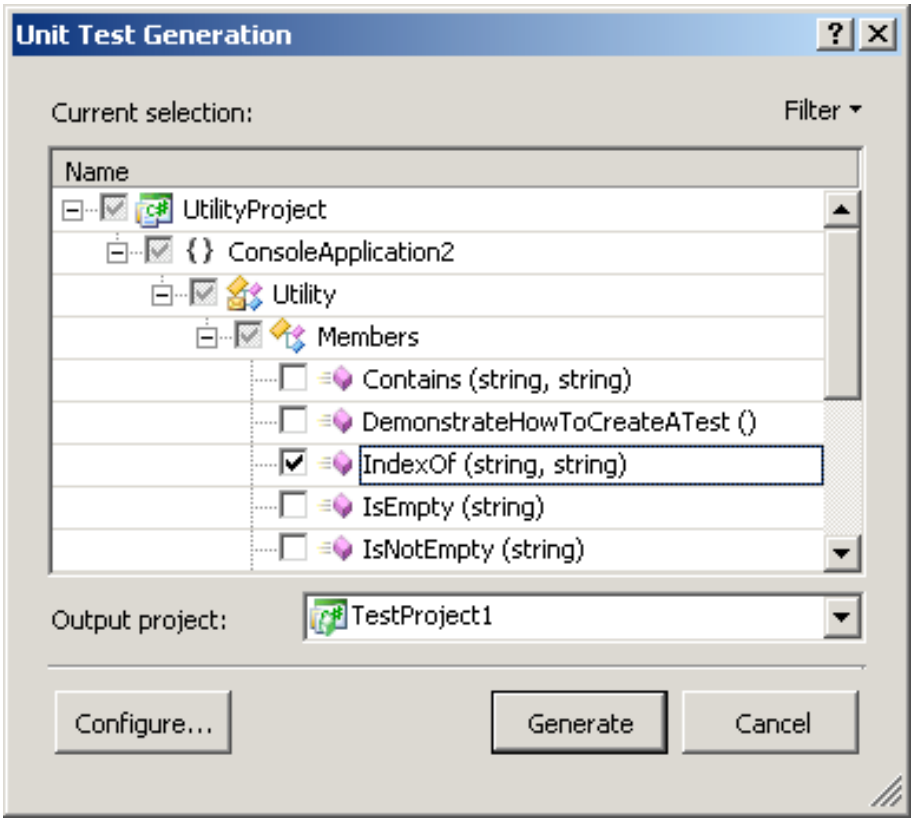

FIGURE 3.1. Test development tool

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 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

```
/// <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 3.2. Generated test method

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 were careful to answer the developers' questions in a way that would avoid biasing 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.

3.3. Threats to Validity

The conclusions drawn from any experiment depends on the validity of the experiment itself. Many factors can threaten experimental validity. Wohlin, et al. [18] grouped these factors into four categories: conclusion validity, construct validity, internal and external validity.

3.3.1. Internal validity

Many other studies of software testing are conducted in large groups. Such studies are plagued by the problem of participants revealing the details of the experiment to others. To avoid these threats, we 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 could have compared the results of the baseline questionnaire with the follow-up questionnaire, but we noticed significant differences between the control group and the code coverage visualization group in the baseline questionnaire. Because we randomly assigned participants to one of two groups, we anticipated that the control group would not differ from the code coverage group in their answers to the baseline questionnaire taken 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 that the control group assessed their own efficacy significantly higher than the code coverage group in the pre-session questionnaire. 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. Consequently, we did not compare the results of the baseline questionnaire to the follow-up questionnaire.

3.3.2. Construct validity

The metrics we used to determine how code coverage visualizations affected test effectiveness, developer efficiency and overestimation of test effectiveness may not accurately reflect the true effect code coverage visualizations had on developers. In particular, in our metrics for overestimation, we compared estimates of faults *found* with the percentage of faults *revealed* through the tests developers wrote. Therefore, our overestimation metric did not account for developers who recognized potential problems in the program but never wrote tests to re-

veal those problems. Future studies could address these concerns by employing a wider variety of metrics to determine the effect of code coverage visualizations on developers.

3.3.3. External validity

We seeded the program that developers tested with several faults. Some faults in the program arose naturally as a side effect of creating the program. In fact, we had not realized they were present until developers in the pilot study exposed these faults for us. That said, we placed most of the faults in the program intentionally. Consequently, the number of faults we placed in the program may not have corresponded to the number of faults our participants were expecting to encounter during the study. 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. Thus, the faults found in the program we provided may not reflect the kind of faults developers typically encounter while testing.

Our study addressed a threat common to most other studies of software testing. Unlike most other studies of software testing, our study did not involve students as participants. Since most studies are run on populations of students, the results of these studies do not necessarily generalize to developers in industry. To avoid this threat to validity, we recruited software developers from several companies.

3.3.4. Conclusion validity

Had we used a within-subjects design, the participant's first treatment might influence their habits on the second treatment, threatening the validity of that design. We avoided this threat through a between-subjects design. Our design could not account for individual differences in programmer efficiency. Thus, a within-subjects design would be a good follow-up design to see if the results we gathered would be replicated. One can reduce both the threats of individual differences and the effect of prior treatments by running experiments using both designs, and through replications of this study with new samples.

4. RESULTS

TABLE 4.1. Background statistics ($\overline{x}$ = mean, s = standard deviation)

Metric	**Control**	**Treatment**
Programming experience (years)	$\overline{x} = 11$ $s = 5.13$	$\overline{x} = 11.06$ $s = 6.76$
Unit testing experience (7 point Likert scale)	$\overline{x} = 2.62$ $s = 1.3$	$\overline{x} = 2.37$ $s = 0.91$

Table 4.1 statistically describes the programming and unit testing background our study's participants. In the following sections we present the hypotheses that we investigated using statistical methods, and we discuss these results in relation to each of our research questions in turn.

4.1. RQ1: Test effectiveness

We recorded the test cases that our participants wrote and determined which test cases revealed faults in the program under test. We also recorded the programs that developers modified and determined which changes fixed faults in the program under test. We define test effectiveness as the number of unique faults each participant revealed with their test cases. To investigate how code coverage visualizations influenced test effectiveness, we compared the number of faults revealed between each group. To investigate how code coverage visualizations influenced developers fixing faults, we compared the number of faults fixed between each group. The null hypotheses are:

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

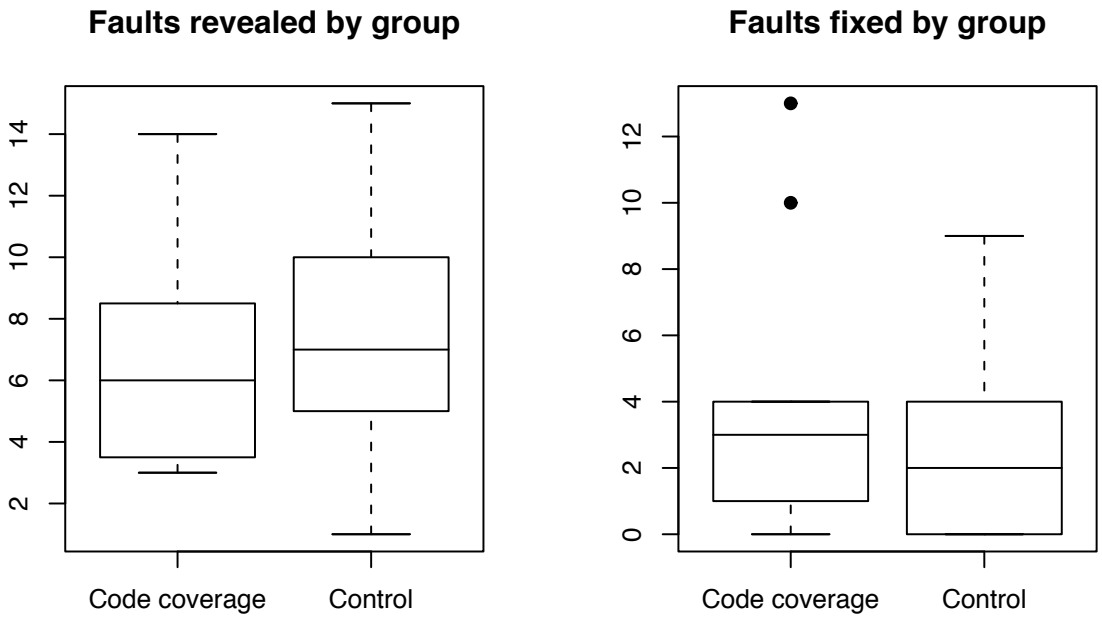

FIGURE 4.1. Faults revealed and fixed by group

H2: The number of faults fixed does not differ between between the control group and the treatment group.

Figure 4.1 displays the distribution of the number of faults revealed by test cases for each group, and shows the distribution of the number of faults fixed by each group using box plots.[1]

To test H1, we ran the Mann-Whitney test on the code coverage group ($n = 15$) and the control group ($n = 15$), $U = 100.5, p = 0.63$. Thus, the test

[1]A 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.

TABLE 4.2. Faults revealed by fault category

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

provided no evidence to suggest a difference in the number of faults revealed between the control group and the treatment group.

To test H2, we ran the Mann-Whitney test, $U = 142.5$, $p = 0.21$. Thus, the test provided no evidence to suggest that code coverage visualizations affected the number of faults developers fixed.

Discussion. Code coverage visualizations using block coverage did not affect the number of faults developers revealed 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 revealed. Likewise, code coverage visualizations using block coverage did not affect the number of faults developers fixed. 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 [19, 13].

Table 4.2 shows the faults revealed by fault category. The table suggests that developers using code coverage visualizations tended to find fewer omissions.

4.2. RQ2: Amount of testing

We used three metrics to compare the tests that the two groups wrote. We looked at the number of test cases, the number of blocks covered, and the number of blocks exercised redundantly by each group. To compare the amount of tests the two groups wrote, we looked at the mean and variance in the number of test cases between each group. To investigate how code coverage visualizations affected coverage, we compared the number of blocks covered between each group. We also compared how often tests covered blocks redundantly (more than once) between each group. The null hypotheses are:

H3: The number of test cases developed does not differ between the control group and the treatment group.

H4: The variance in the number of test cases developed does not differ between the control group and the treatment group.

H5: The number of blocks covered does not differ between between the control group and the treatment group.

H6: Redundant coverage does not differ between between the control group and the treatment group.

H7: The variance of redundant coverage does not differ between between the control group and the treatment group.

Figure 4.2 displays the distribution of the number of test cases written per group, the distribution of the blocks covered per group, and the distribution of redundant coverage per group using box plots. The leftmost box plot in Figure 4.2 suggests that developers using code coverage visualizations produced slightly fewer

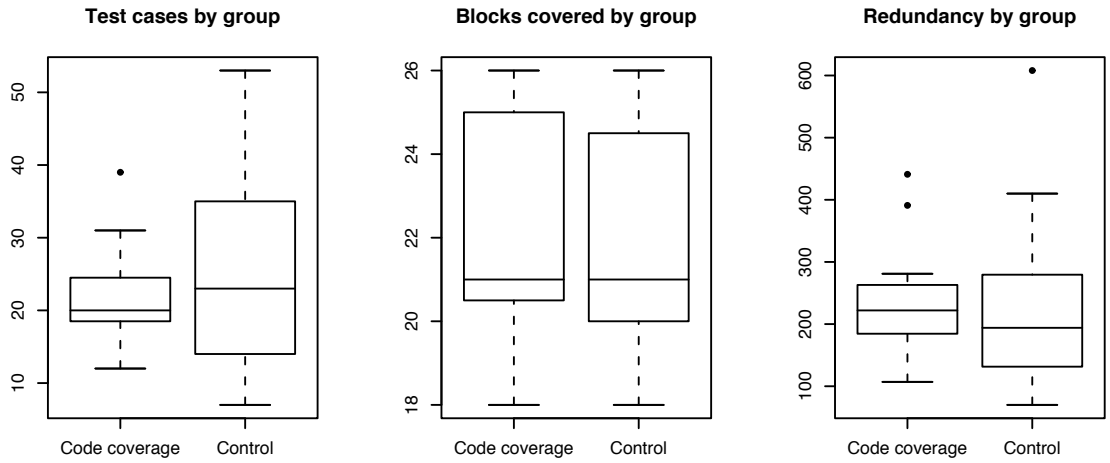

FIGURE 4.2. Test cases, blocks covered, and redundant coverage by group

test cases and varied less in the amount of test cases than developers without code coverage visualizations. The middle box plot in Figure 4.2 suggests that the coverage did not differ between the two groups. The rightmost box plot in Figure 4.2 suggests that code coverage developers may have exercised slightly more blocks redundantly than the control group, but varied less in the amount of blocks exercised redundantly.

To test H3, we ran the Mann-Whitney test, $U = 100, p = 0.62$. Thus, the test provided no evidence to suggest the number of test cases between the control group and treatment group differed.

To test H4, we ran the Levine test, $\mathbf{F = 6.42}$, $\mathbf{p = 0.017}$. 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.

To test H5, we ran the Mann-Whitney test, $U = 113, p = 1$. Thus, the test provided no evidence to suggest that code coverage visualizations affected the number of blocks covered.

To test H6, we ran the Mann-Whitney test, $U = 123.5, p = 0.66$. Thus, the test provided no evidence to suggest that code coverage visualizations affected the number of blocks covered redundantly.

To test H7, we ran the Levine test, $F = 2.11$, $p = 0.157$. Thus, the test provided no evidence to suggest that code coverage visualizations reduced the variability in the number of blocks covered redundantly.

Discussion. The reduced variability in the number 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.

The result of testing H5 suggests that code coverage visualizations did not impact coverage. Because it is relatively trivial to write test cases that achieve high coverage by the block coverage criteria, perhaps the result of H5 should not be a surprise. That said, the result is ironic considering that code coverage visualizations are supposed to improve test coverage.

The boxplots in Figure 4.2 and the results of testing H6 and H7 suggest that code coverage visualizations did not influence the number of blocks exercised redundantly as strongly as code coverage visualizations influenced the number of test cases developed.

4.3. RQ3: Overestimation of correctness

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 led developers into overestimating the correctness of the program, we asked developers: "Please give your estimate of the percent of faults that you found." Using our measure of the number of faults revealed, we devised the following two formulas to measure the actual percentage of faults revealed, 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 revealed, we compared overestimation by group.

H8: Overestimation did not differ by the following:

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

H9: Overestimation did not differ by the following:

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

Figure 4.3 displays the distribution of overestimation measured by group. The box plots suggest that developers in the code coverage group overestimated the percentage of faults revealed more than developers in the control group.

To test H8, we ran the Mann-Whitney test, $U = 151.5$, $p = 0.052$. To test H9, we ran the Mann-Whitney test, $\mathbf{U = 160}$, $\mathbf{p = 0.026}$. The result of H9 provides evidence to suggest that developers using code coverage visualizations overestimated the percentage of faults revealed more than developers without code coverage visualizations.

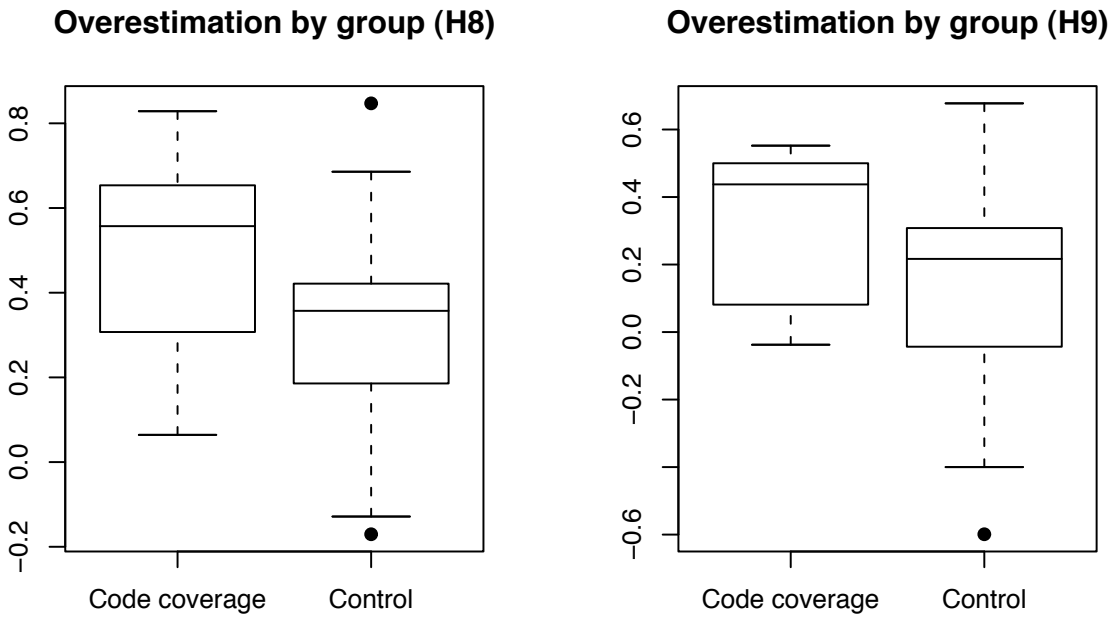

FIGURE 4.3. Overestimation by group

Discussion. Both groups overestimated the percentage of faults they revealed, indicating both groups did not have a true sense of how many faults they revealed with their test cases. This result is not surprising, considering the pervasive human tendency toward overconfidence [14]. However, the overestimation we observed in this study suggests that code coverage visualizations using block coverage criteria did not help developers attain a truer assessment of how many faults they found. In fact, developers who used visualizations of block coverage had even less sense of how many faults they found.

Recall that test adequacy criteria are designed to tell people when to stop testing. 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 may be critical. With the block coverage criterion, it was trivial to achieve complete coverage of the program we provided. Using stronger coverage

criteria, the task of writing tests that achieve complete coverage becomes less trivial. Thus, the stronger the criterion, the harder the testing task, and the less coverage each test provides.

The results of RQ3 taken together with previous findings for definition-use coverage in spreadsheets implies that the choice of test adequacy criterion may influence developers' estimates of their own effectiveness at testing, which in turn plays a critical role in determining when code is ready to ship. However, given the human tendency toward overconfidence, it is unlikely that testing visualizations could solve the problem of developers overestimating the percentage of faults that they revealed.

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 4.3. We also classified strategies developers used in response to a test run as productive or counterproductive using the categorization shown in Table 4.4.

We performed Fisher's exact test to test whether code coverage visualizations influenced whether the test follow-up strategies developers used were productive or counterproductive, $p = 0.21$. 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 coun-

TABLE 4.3. 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

[a]Developers employed several of these strategies simultaneously.

terproductive strategies listed in Table 4.4. This suggests that even professional developers need assistance to avoid counterproductive strategies. Some counter-

TABLE 4.4. 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

productive 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 without making any changes. Some common strategies, such as copying and pasting test code or debugging carried potential risks or consumed time. We did not classify these strategies as counterproductive, since neither of these strategies consistently posed an immediate threat to the testing task like the counterproductive strategies we identified.

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 have given 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, 19, 16]. In each of these studies, like our own study, experimenters gave subjects faulty programs and compared the subjects' test effectiveness based on the testing technique. In [2, 13, 19], 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 [19] 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 did.

Although our results were consistent with results of experiments in which statement and branch coverage were used as test adequacy criteria, they were *not* consistent with the results of [16]. In [16], subjects using the stronger 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 [10, 6, 7, 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] implies that the test adequacy criterion may be 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 toward 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.

6.1. 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., Sitges, Spain, 1995, pp. 362–383.

[14] R. Panko, "What we know about spreadsheet errors." *Journal of End User Computing*, vol. 10, no. 2, 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] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, V. R. Basili, Ed. Kluwer Academic Publishers, 2000.

[19] 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.

[20] 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.

APPENDICES

APPENDIX A. Software Testing Study

Thank you for volunteering to participate in today's study. We appreciate the time you have taken out of your schedule to provide us with your feed back.

For today's study, imagine you work for Acme Inc., a software development company that develops and sells class libraries written in C#. You work for Acme as a software developer, and you have just completed the implementation of some new Utility class. Your task today will be to create an effective set of test cases for the methods in that class.

1. First, we'll ask some questions about your experiences with testing software. Please double click on pre.qst and complete the survey to the best of your ability.

2. Starting with whatever method you choose, generate what you would consider to be an effective set of test cases for that method. Feel free to read the code to understand what it does. Be sure to observe any failures or faults you uncover aloud. Generate and run test cases until you are confident that you have found all the bugs in each method before moving on to the next method.

3. Double click on result.qst and complete the follow-up questionnaire.

APPENDIX B. Program.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication2
{
  class Utility
  {
    public static void Main()
    {
    }
    public bool
    DemonstrateHowToCreateATest()
    {
      //right click inside the method you
      //want to test and choose the create
      //tests... command
      int i;
      i = 42;
      Console.WriteLine("This function " +
              "does nothing useful");
      return (i == 42);
    }
    /// <summary>
    /// Checks if the string is empty ("")
    /// or null
    /// </summary>
    /// <param name="s"></param>
    /// <returns></returns>
    public static bool IsEmpty(string s)
    {
      bool result = false;
      if (s == null)
        result = true;
      if (s.Length == 0)
        result = true;
      return result;
    }
    /// <summary>
    /// Reports the position of needle in
```

```
/// haystack.
/// </summary>
/// <param name="haystack"></param>
/// <param name="needle"></param>
/// <returns>Should behave exactly like
/// the IndexOf method in the String
/// class</returns>
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;
}
/// <summary>
/// Return true if needle is a substring
/// of haystack.
/// </summary>
/// <param name="haystack"></param>
/// <param name="needle"></param>
/// <returns></returns>
public static bool
```

```
Contains(string needle, string haystack)
{
  return IndexOf(haystack, needle) >= 0;
}
/// <summary>
/// Return the square root of the number
/// using Newton's method.
/// </summary>
/// <param name="number"></param>
/// <returns></returns>
public double SquareRoot(double number)
{
  double result = number;
  while (result * result != number)
    result = (result+(number/result))/2.0;
  return result;
}
/// <summary>
/// Replace OriginalCharacters with
/// ReplacementCharacters in str.
/// <example>
/// Translate("Hello"," Ho"," Jy")
/// = "Jelly"
/// Translate("S3KR37"," R3K7"," rect")
/// = "Secret"
/// </example>
/// </summary>
/// <param name="str"></param>
/// <param name="original"></param>
/// <param name="replacement"></param>
/// <returns></returns>
public string Translate(string str,
string Original,
string Replacement)
{
  StringBuilder buf;
  buf  = new StringBuilder(str.Length);
  for (int i = 0; i < str.Length; i++)
  {
    char ch = str[i];
    int index;
    index  = Original.IndexOf(ch);
```

```
    if (index > 0)
    {
      buf.Append(Replacement[index]);
    }
    else
    {
      buf.Append(ch);
    }
  }
  if (Original.Equals(Replacement))
    return str;
  return buf.ToString();
}
/// <summary>
/// Convert a string to upper case
/// </summary>
/// <param name="s"></param>
/// <returns>
/// String s in upper case
/// </returns>
public string ToUpper(string s)
{
  return Translate(s,
  "abcdefghijklmnopqrstuvwxzy",
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
}
/// <summary>
/// Convert a string to lower case
/// </summary>
/// <param name="s"></param>
/// <returns>
/// String s in lower case
/// </returns>
public string ToLower(string s)
{
  return Translate(s,
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
  "abcdefghijklmnopqrstuvwxyz");
}
/// <summary>
/// Get the first substring after the
/// first occurrence of a separator.
```

```
    /// </summary>
    /// <example>
    /// SubstringAfter("abcdef","bc")
    /// = "def"
    /// </example>
    /// <param name="?"></param>
    /// <returns></returns>
    public static string
    SubstringAfter(string str, string separator)
    {
      return str.Substring(IndexOf(str, separator)
      + separator.Length);
    }
    /// <summary>
    /// Get the first substring before the
    /// first occurrence of the separator
    /// </summary>
    /// <example>
    /// SubstringBefore("abcdef","def")
    /// = "abc"
    /// </example>
    /// <returns></returns>
    public static string
    SubstringBefore(string str, string separator)
    {
      return str.Substring(0,
      IndexOf(str, separator));
    }
    /// <summary>
    /// Return the negation of IsEmpty
    /// </summary>
    /// <param name="s"></param>
    /// <returns></returns>
    public static bool IsNotEmpty(string s)
    {
      return (s != null || s.Length > 0);
    }
  }
}
```

APPENDIX C. ProgramTest.cs

```
using Microsoft.VisualStudio.QualityTools.
UnitTesting.Framework;
namespace TestProject1
{
  ///<summary>
  ///This is a test class for
  ///ConsoleApplication2.Utility and is
  ///intended to contain all
  ///ConsoleApplication2.Utility Unit Tests
  ///</summary>
  [TestClass()]
  public class UtilityTest
  {
    private TestContext testContextInstance;
    ///<summary>
    ///Gets or sets the test context which
    ///provides information about and
    ///functionality for the current test run.
    ///</summary>
    public TestContext TestContext
    {
      get { return testContextInstance;  }
      set { testContextInstance = value; }
    }
    ///<summary>
    ///Initialize() is called once during test
    ///execution before test methods in this
    ///test class are executed.
    ///</summary>
    [TestInitialize()]
    public void Initialize() {}
    ///<summary>
    ///Cleanup() is called once during test
    ///execution after test methods in this
    ///class have executed unless this test
    ///class' Initialize() method throws an
    ///exception.
    ///</summary>
    [TestCleanup()]
    public void Cleanup() {}
```

```
///<summary>
///A test case for IndexOf (string, string)
///</summary>
[TestMethod()]
public void IndexOfTest()
{
  string haystack = "Hello, world!";
  string needle = "world";
  int expected = 7;
  int actual;
  actual = TestProject1.
  ConsoleApplication2_UtilityAccessor.
  IndexOf(haystack, needle);
  Assert.AreEqual(expected, actual,
  "ConsoleApplication2.Utility.IndexOf did"
  + " not return the expected value.");
}
///<summary>
///A test case for IndexOf (string, string)
///</summary>
[TestMethod()]
public void IndexOfTest1()
{
  string haystack = "Hello, world!";
  string needle = "";
  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.");
}
///<summary>
///A test case for Translate
///(string, string, string)
///</summary>
[TestMethod()]
public void TranslateTest()
{
  object target = TestProject1.
```

```
  ConsoleApplication2_UtilityAccessor.
  CreatePrivate();
  TestProject1.
  ConsoleApplication2_UtilityAccessor accessor
  = new TestProject1.
  ConsoleApplication2_UtilityAccessor(target);
  string str = "Hello";
  string Original = " Ho";
  string Replacement = " Jy";
  string expected = "Jelly";
  string actual;
  actual = accessor.Translate(str,
  Original, Replacement);
  Assert.AreEqual(expected, actual,
  "ConsoleApplication2.Utility.Translate did"
  + " not return the expected value.");
}
///<summary>
///A test case for Translate
///(string, string, string)
///</summary>
[TestMethod()]
public void TranslateTest1()
{
  object target = TestProject1.
  ConsoleApplication2_UtilityAccessor.
  CreatePrivate();
  TestProject1.
  ConsoleApplication2_UtilityAccessor accessor
  = new TestProject1.
  ConsoleApplication2_UtilityAccessor(target);
  string str = "Hello, world!";
  string Original = "asdf";
  string Replacement = "asdf";
  string expected = "Hello, world!";
  string actual;
  actual = accessor.Translate(str,
  Original, Replacement);
  Assert.AreEqual(expected, actual,
  "ConsoleApplication2.Utility.Translate did"
  + " not return the expected value.");
}
```

```
    }
}
```

APPENDIX D. Pre Questionnaire

Background survey

1. How many years of C# programming experience do you have?
2. How many years of programming experience do you have?
3. How much experience do you have with C#? (None ... Guru)
4. How often do you write unit tests... (Never ... Always)
   (a) for your own code?
   (b) before writing code?
   (c) while writing code?
   (d) to maintain code?

Pre-session questionnaire

5. How motivated are you to write unit tests? (Not at all ... Very much)
6. I enjoy testing code with unit tests. (Disagree ... Agree)
7. Given a class with familiar methods, I could find errors...
   (Disagree ... Agree)
   (a) if there was no one around to tell me what to do as I go.
   (b) if I only had manuals or online documentation for references.
   (c) if I could ask someone for help if I got stuck.
   (d) if I had a lot of time to complete the task.
   (e) if I had the built-in help facility for assistance.

(f) if I had tested similar classes in the past.

8. In testing your own code, how confident are you in finding bugs?
   (Not at all confident ... Very confident)

9. When creating unit tests for your own code, how confident are you that those unit tests will find more than 80% of the bugs?
   (Not at all confident ... Very confident)

APPENDIX E. Post Questionnaire

1. Please give your estimate of the percent of faults that you found.
2. If you were using the testing tools in this study for testing your own code, how confident would you be in finding 80% of the bugs?
   (Not at all confident ... Very confident)
3. I'd rate the testing tools in the development environment I used as:
   (Harmful ... Helpful)
4. The testing tools I used... (Disagree ... Agree)
   (a) mislead me into believing buggy code was correct.
   (b) mislead me into believing correct code was buggy.
   (c) led me to overlook code that was tested but was still incorrect.
   (d) led me to suspect correct code was incorrect.
   (e) gave me a sense that I've found all the faults.
5. I'd write _____ unit tests if I used the testing tools in this study. (Much fewer ... Much more)
6. How motivated are you to write unit tests? (Not at all ... Very much)
7. I enjoy testing code with unit tests. (Disagree ... Agree)
8. Given a class with familiar methods, I could find errors...
   (Disagree ... Agree)
   (a) if there was no one around to tell me what to do as I go.

(b) if I only had manuals or online documentation for references.

(c) if I could ask someone for help if I got stuck.

(d) if I had a lot of time to complete the task.

(e) if I had the built-in help facility for assistance.

(f) if I had tested similar classes in the past.