ABSTRACT

Title of dissertation:	Accounting for Defect Characteristics in Empirical Studies of Software Testing
	Jaymie Strecker, Doctor of Philosophy, 2009
Dissertation directed by:	Professor Atif M. Memon Department of Computer Science

Software testing is an indispensable activity in quality assurance and an enduring topic of research. For decades, researchers have been inventing new and better techniques to test software. However, no testing technique will ever be a panacea for all software defects. Therefore, while researchers should continue to develop new testing techniques, they also need to deeply understand the abilities and limitations of existing techniques, the ways they complement each other, and the trade-offs involved in using different techniques. This work contends that researchers cannot sufficiently understand software testing without also understanding software defects.

This work is the first to show that simple, automatically-measurable characteristics of defects affect their susceptibility to detection by software testing. Unlike previous attempts to characterize defects, this work offers a characterization that is objective, practical, and proven to help explain why some defects and not others are detected by testing.

More importantly, this work shows that researchers can and should account for defect characteristics when they study the effectiveness of software-testing techniques. An experiment methodology is presented that enables experimenters to compare the effectiveness of different techniques and, at the same time, to measure the influence of defect characteristics and other factors on the results. The methodology is demonstrated in a large experiment in the domain of graphical-user-interface testing.

As the experiment shows, researchers who use the methodology will understand what kinds of defects tend to be detected by testing and what testing techniques are better at detecting certain kinds of defects. This information can help researchers develop more effective testing techniques, and it can help software testers make better choices about the testing techniques to use on their projects. As this work explains, it also has the potential to help testers detect more defects, and more important defects, during regression testing.

Accounting for Defect Characteristics in Empirical Studies of Software Testing

by

Jaymie Strecker

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2009

Dedication

To Mom and Dad.

Acknowledgments

Without the help of many people, this dissertation would not have been possible. I extend my thanks to all of them.

To my advisor, Atif Memon, for making a thousand suggestions that improved this work, for imposing deadlines, for giving me opportunities to learn from others by reviewing their work and meeting them at conferences, and for providing financial support through a research assistantship.

To Dianne O'Leary for going above and beyond the duties of a committee member by reading and commenting on numerous drafts, offering valuable advice, and always treating me with the utmost consideration.

To the rest of my committee—Brian Hunt, Vibha Sazawal, and Marv Zelkowitz—and to the anonymous reviewers of papers I have submitted for showing me how to strengthen this work.

To the National Physical Sciences Consortium, the University of Maryland, and Verizon for financially supporting my graduate studies.

To Eric Slud for providing much-needed guidance on statistics.

To members of the GUITAR research group—especially Penelope Brooks, Cyntrica Eaton, Scott McMaster, Qing Xie, and Xun Yuan—for sharing their expertise on GUI testing and more.

To members of research groups in which I previously participated—especially Vic Basili, Lorin Hochstein, Bill Pugh, and Jaime Spacco—for influencing many of the ideas in this work.

To the professors of mathematics and computer science at the College of Wooster for 4.5 years (and counting) of being excellent teachers and role models for me. Especially to Denise Byrnes and Simon Gray for helping me make progress on my dissertation, and to Denise, Amon Seagull, Dale Brown, and Jon Breitenbucher for advising my first research projects.

To my students at the College of Wooster for giving me a good reason to finish.

To Shirley, my MentorNet mentor, for listening and for providing advice generously and nonjudgmentally.

Finally, thanks to my closest friends and my family for everything. This would not have happened without you.

Table of Contents

Chapter 1

Introduction

Delivering high-quality software is a crucial and difficult undertaking, and the field of software testing is still struggling the meet the challenge. A 2002 NIST report [62] estimates the “annual costs of an inadequate infrastructure for software testing” in the U.S. to be \$22.2 billion to \$59.5 billion. (To put this in perspective, consider that the total software sales in 2000 were \$180 billion [62].) Inadequate testing leads to defective software, and defective software costs society. For every software defect that makes headlines—by crashing an expensive spacecraft, taking down telephone service, or administering lethal doses of radiation [24]—countless more mundane defects waste users’ time and compromise their data, tarnish companies’ reputations, and compel developers to distribute costly patches. Most of those defects could have been prevented from ever reaching users if software testers had been able to test more effectively.

Software testing is an indispensable activity in quality assurance (QA), complementary to other QA activities like formal modeling, static analysis, and inspections. Donald Knuth famously illustrated this when he wrote, “Beware of bugs in the above code; I have only proved it correct, not tried it” [33].

Testing mainly serves two purposes: to establish confidence that the software under test behaves as intended and to detect cases where it does not. Testing is not a single technique, but rather a class of techniques that involve executing the software under some input conditions and observing the result. To fulfill its purposes, any testing technique must be good at detecting defects.

No single software-testing technique—or any QA technique—will ever be a panacea for all defects. Nor will any single testing technique be a "silver bullet" to productivity, instantly slashing the 50% of development effort required for adequate testing [11]. But testers in practice are willing to spend a reasonable amount of time to detect a reasonable number of defects—preferably the most severe ones. For many testers, the goal is "not zero defects but zero defections and positive flow of new customers" [63].

Therefore, while researchers should continue to develop new techniques for software testing, they also need to *deeply* understand the abilities and limitations of existing techniques, the ways they complement each other, and the trade-offs involved in using different techniques. This work contends that researchers cannot sufficiently understand software testing without also understanding software defects.

Thesis statement. Simple, automatically-measurable characteristics of defects affect their susceptibility to detection by at least one form of software testing. Accounting for these characteristics in empirical studies of software testing increases the validity of study results and is feasible to do.

1.1 Basic concepts and terminology

1.1.1 Software testing

Some terms are commonly used in the literature on software testing. A *test case* is an element of the input space of the software under test. For example, one test case for the function

```
int remainder(int dividend, int divisor) {
   if (divisor == 0)
      throw exception;
   else
      return dividend % divisor;
}
```

would be $\langle dividend = 1, divisor = 0\rangle$. Usually, more than one test case is required to test the software thoroughly. A set of test cases to be run together is called a *test suite*. The *oracle* or *oracle information* is the behavior, or an abstraction of the behavior, that should result from a test case. It is specified a priori, often at the time when the test case is written. The *oracle procedure* compares the oracle information to the actual result of a test case to decide whether the test case detects a defect. The process of creating test cases and running the oracle procedure may be as informal as clicking through an application and eyeballing the output. Or it may be as formal as systematically, even algorithmically, identifying program paths to test and comparing the result to an output specification.

Most research on software testing has focused on the inter-related problems of creating test suites and deciding when enough testing has been done. For the problem of creating test suites, techniques have been developed to help human testers systematically write test cases or test specifications [41, 45], to generate test cases automatically [14, 21, 27, 37, 61, 67], and to select a subset of test cases from an existing test suite in order to retest specific parts of the software [2, 28, 54]. For the problem of deciding when to stop testing, much research has focused on *coverage metrics*, which, in various ways, measure the proportion of the software that a test suite executes, or *covers.* For example, *statement coverage* or *line coverage* measures the proportion of executable statements in the source code that a test suite covers. An X*-coverage-adequate* test suite is one that covers every X in the software under test, where X is some kind of program element. For example, one of many possible *statement-coverage-adequate* test suites for the `remainder` function above would be $\{\langle dividend = 1, divisor = 0\rangle, \langle dividend = 1, divisor = 1\rangle\}$.

Many of the steps in testing—generating, executing, and measuring the coverage of test suites—have, to a great extent, been automated. However, the process of testing can never be automated fully because of the *oracle problem.* Someone has to specify the correct behavior that should result from each test case; writing a program to do so would amount to writing a defect-free version of the software under test (although perhaps in a higher-level language) [15]. The best the oracle procedure can do without human assistance is to use heuristics, such as reporting uncaught exceptions [66], looking for unusual software states [19], or comparing the output to that of an earlier version of the software [2].

Usually, a software product is developed in a series of versions. Each version after the initial version may introduce defect fixes, new features, and restructuring of the code to accommodate future changes [38]. In any case, the changes made to the software since the previous version need to be tested. An important part of this process is *regression testing*, which the IEEE Standard Glossary of Software Engineering Terminology [50] defines as follows:

> Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

1.1.2 Software defects

Software defects also have their technical terms. In common parlance, these terms are often used interchangeably, and even among experts they have different meanings in different communities [50]. In this work, *mistakes*, *faults*, and *failures* are treated as distinct, following one version of the definitions in the IEEE Standard Glossary of Software Engineering Terminology [50]. Here, a *mistake* is considered to be a flaw in a programmer's mental conception of a program that leads to one or more problems in the software. Each of those problems, as it is manifested in the source code of the software, is called a *fault*. A fault may cause one or more *failures*, or incorrect behaviors in the software. This work uses the term *defects* to refer generally to problems with software—mistakes, faults, or failures.

When researchers study the abilities of testing techniques to detect defects, they usually need to collect or create some defective software on which to try the techniques. They can accomplish this in several ways. They can collect *natural defects* (sometimes called *real defects*), which are defects made accidentally by the developers of a software product. If the developers have kept good records through a version-control system and a defect-tracking system, then researchers can isolate and reconstruct the defects that have been fixed over time, although this is a painstaking process. Alternatively, researchers can *seed* defects into a software product—that is, insert them intentionally. Defects can either be seeded by hand or by automated mutation. A *mutation fault* is a small, typo-like fault in the code, such as changing a plus sign to a minus sign.

1.2 Contribution: Methodology to account for fault characteristics in empirical studies of software testing

Software-testing techniques need to be good at detecting defects in software. Researchers *evaluate* testing techniques to determine if they are good—relative to other techniques, within some domain of software and defects, by some measurable definition of "good" that considers the resources used and the defects detected.

Anyone who has dealt with software defects before knows that not all defects are equal; some are more susceptible to detection than others. Yet, for decades, evaluations of testing techniques have not been able to take this into account very well. This work offers a remedy: a methodology for empirical evaluations that accounts for the impact that defect characteristics have on evaluation results.

As motivation, consider the well-known experiment on data-flow- and control-flow-based testing techniques by Hutchins et al. [31]. The experiment compared test suites covering all def-use pairs[1], suites satisfying predicate ("edge") coverage[2], and random suites[3]. Test suites were created and run for seven C programs with a total of 130 hand-seeded faults. In order to compare the techniques, each fault was classified according to which kinds of test suites (i.e., techniques) were most likely to detect it. It turned out that few faults were equally likely to be detected by all techniques; instead, most faults clearly lent themselves to detection by just one or two of the techniques. This is a tantalizing conclusion, one that could potentially help testers to choose a testing technique based on the kinds of faults they hope or expect to detect. Unfortunately, the authors "were not able to discern any characteristics of the faults, either syntactic or semantic, that seem to correlate with higher detection by either method."

At least one other empirical study, by Basili and Selby [8], has shown that

[1]Def-use-pair coverage has to do with data flow in a program. A *def-use pair* is a *def* (definition/write) of a variable together with a subsequent *use* (read) of that variable, such that there is a path in the program from the def to the use with no intervening def. A test suite satisfying def-use-pair coverage executes at least one such path for each def-use pair.

[2]Predicate coverage has to do with control flow in a program. It is more rigorous than branch coverage, which requires that each `if` condition be executed at least once with a value of `true` and at least once with a value of `false`. Because an `if` condition may consist of a conjunction or disjunction of individual predicates, predicate coverage additionally requires that each predicate be executed at least once with a value of `true` and at least once with a value of `false`.

[3]Test suites randomly selected from a large set of test cases have often been used in software-testing studies as a control.

different defects can be “harder” to detect with one testing technique and “easier” with another. Based on one’s experience testing software, one might also suspect that some faults are “harder” or “easier” than others in a more general sense. Offutt and Hayes [43] formalized this notion, and they and others have observed it empirically [5, 31, 53].

Thus, the defects against which testing techniques are evaluated can make the techniques look better or worse—both absolutely, in defect-detection rates, and relatively to other techniques. Without understanding defects, it is difficult to integrate results from different experiments [9], and it is usually impossible to explain why one technique outperforms another.

To understand fully how evaluations of testing techniques may depend on the defects used, one must be familiar with the typical procedure for such evaluations. Although some analytical approaches have been proposed [23], by far the most common and practical way to evaluate testing techniques continues to be empirical studies. Typically, these studies investigate hypotheses like “Technique *A* detects more defects than technique *B*” or “*A* detects more than zero defects more often than *B*” [32].

Empirical studies, by their very nature as sample-based evaluations, always face the risk that different samples might lead to different results. An empirical study must select a sample of software to test, a sample of the test suites that can be generated (or recognized) by each technique for the software, and a sample of the defects (typically faults) that may arise in the software. Because different studies usually use different samples, one study might report that technique *A* detects more faults than B, while another would report just the opposite. More and more, published empirical studies of software testing are acknowledging this as a threat to external validity [5, 26, 53].

This threat to external validity can be mitigated by replicating the study with different samples of test suites and fault-ridden software. But, while replicated studies are necessary for scientific progress [9, 68], they are not sufficient. It is not enough to *observe* differing results in replicated studies; it is necessary to *explain* and *predict* them, for several reasons. First, explanation and prediction of phenomena are goals of any science, including the science of software testing. Second, the ability to predict situations in which a software-testing technique might behave differently than in a studied situation would aid researchers by pointing to interesting situations to study in the future [9]. Third, it would aid practitioners by alerting them if a testing technique may not behave as expected for their project.

If evaluators of testing techniques are to explain and predict the techniques' performance outside the evaluation, then they must identify and account for all the characteristics of the studied samples of software, test suite, and faults that can significantly affect the evaluation's results. Some previous work has identified and accounted for characteristics of the software (e.g., size) and the test suites (e.g., granularity) [18, 39, 53, 65]. Characteristics of faults, however, have resisted scrutiny. Few characteristics of faults have been identified, even fewer have been practical and objective to measure, and none of those have been demonstrated to help explain testing techniques' behavior [59]. In the words of Harrold et al. [29], "Although there have been studies of fault categories... there is no established correlation between categories of faults and testing techniques that expose those faults."

Since characteristics of faults must be identified before they can be accounted for, this work provides a more adequate characterization of faults. In this work, a fault is characterized by a vector of characteristics: the fault's method of creation, the faulty code's distance from the program's initial state, the faulty code's tendency to be executed repeatedly, the degrees of freedom in executing the faulty code, and the size of the component containing the faulty code. Each characteristic can be measured automatically, without the need for formal specifications or other documents, making it feasible to use in large empirical studies. While the characterization is not intended to be complete or definitive, it is a reasonable starting point toward explaining faults' susceptibility to detection.

More importantly, this work shows how these fault characteristics—or any vector of one or more fault characteristics—can be accounted for in empirical studies of testing techniques. The challenge here is that, for a given piece of software under test, fault characteristics *and* test-suite characteristics may both affect fault detection. To account for both kinds of characteristics, this work presents a new *methodology*, or template for designing empirical studies. The methodology shows how a study's inputs (test suites and faulty software) can be assembled and analyzed to discover how well different kinds of test suites cover different parts of the software and detect different kinds of faults.

1.3 Contribution: Experiment applying the methodology

This work presents an experiment that applies the methodology, demonstrating that the methodology is viable and useful. Using automated techniques for graphical-user-interface (GUI) testing and for fault seeding, the experiment studies the characteristics and fault-detection properties of 3200 unique $\langle$*test suite, fault*$\rangle$ pairs.

The experiment is among the first ever to statistically model the relationship between characteristics of $\langle$*test suite, fault*$\rangle$ pairs and test suites' ability to detect faults. In addition, it is the first to study empirically the conditional probability that a test suite detects a fault given that it covers the faulty code. The models developed in this experiment show that each of the fault characteristics proposed in this work helps explain differences in faults' susceptibility to detection in one domain of testing.

1.4 Contribution: Practical applications to regression testing

This work describes several ways that statistical models like the ones developed in the experiment can be used to improve regression testing. These are packaged as a set of scenarios, with concrete examples and demonstrations.

Unlike previous research on regression testing, which has typically focused on improving the cost or speed of fault detection without changing the set of faults detected, these scenarios show how to detect more faults or more important faults. The key idea is to use the statistical models, along with feedback from previous versions of the software under test, to predict how effective different kinds of test suites would be on the current version.

1.5 Summary of contributions

In summary, this dissertation makes the following research contributions:

- A methodology to account for both test-suite and fault characteristics in empirical studies of software testing (*Chapter 3*)
- A way to empirically explore the relationship between execution of faulty code and detection of faults (*Chapter 3*)
- An application of the methodology in a large experiment studying the effects of test-suite and fault characteristics on fault detection (*Chapter 4*)
- A simple, practical fault characterization for software-testing studies (*Chapter 4*)
- Evidence that the fault characterization helps explain faults' susceptibility to detection by GUI testing (*Chapter 4*)
- A set of scenarios showing how statistical models like the ones developed in the experiment could be used to improve regression testing (*Chapter 5*)
- A demonstration of the scenarios using models developed in the experiment (*Chapter 5*)

1.6 Intellectual merit and broader impacts

This work has important implications for software-testing researchers. It points out a problem in the way empirical studies of fault detection are typically conducted: characteristics of the sample of faults can impact the results. It also provides a solution: an experiment methodology that accounts for fault characteristics. Using the methodology, researchers will be better able to explain and predict the performance of testing techniques. This will help them interpret the results of empirical studies, choose the contexts in which to replicate empirical studies, and develop new testing techniques that address weaknesses of existing techniques.

This work can also help software-testing practitioners. When evaluations of testing techniques follow the methodology presented here, testers will better understand how the evaluated techniques would perform on their own software projects. This will help them choose the most efficient and effective techniques for their projects. Furthermore, using the scenarios for regression testing presented in this work, testers may be able to detect more faults or more important faults in evolving software. Being able to choose the best testing techniques and to detect more faults in software, testers might recover some of the billions of dollars wasted each year because of inadequate software testing and poor software quality.

Chapter 2

Background and Related Work

This chapter begins with necessary background information on GUI testing (Section 2.1) and on characteristics of faults (Section 2.2), test suites (Section 2.3), and, for completeness, software products and processes (Section 2.4). Section 2.5 goes on to describe models of failure propagation that inspired this work's treatment of faulty-code coverage and fault detection. Finally, Section 2.6 describes approaches to software testing and related activities that, like this work's scenarios for regression testing, are adaptive.

2.1 GUI testing

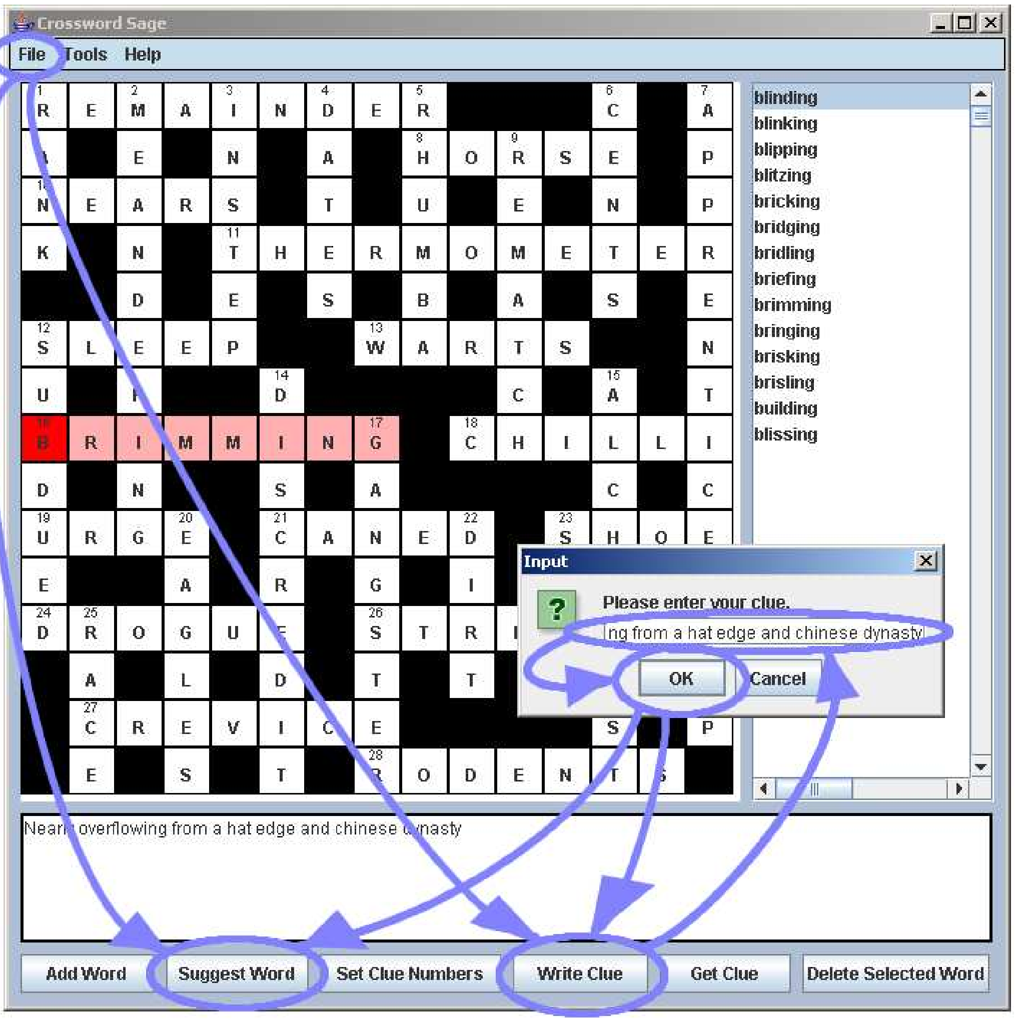
Experiments in software testing have often focused on a particular domain of software (e.g., UNIX utilities) and of testing (e.g., JUnit test cases). This work focuses on GUI-intensive applications and model-based GUI testing [37, 61], a form of system testing. GUI-intensive applications make up a large portion of today's software, so it is important to include them as subjects of empirical studies. Conveniently, model-based GUI testing lends itself to experimentation because test cases can be generated and executed automatically, enabling experimenters to create large samples of test cases. Because test cases are generated automatically and in a model-based way, experiment results are not influenced by the skill of testers.

The basic unit of interaction with a GUI is an *event*. Some examples of events are clicking on a button or typing "hello" in a text box. As the latter example suggests, some events are parameterized—e.g., text entry in a text box is parameterized by the text string entered. In actual GUI testing, only a finite set of parameter values can be tested for each event. This work uses just one parameter value for each event, thus eliminating, in effect, the distinction between unparameterized and parameterized events.

A GUI can be modeled by an *event-flow graph (EFG)*, in which each node represents a GUI event. Figure 2.1 shows a GUI and, superimposed on it, some of the nodes and edges in its EFG. In an EFG, a directed edge to node n_2 from node n_1 means that the corresponding event e_2 can be executed immediately after event e_1. (The term *event* will be used to mean both an actual event and a node representing an event.) In Figure 2.1, for example, the event "click Suggest Word button" in the larger window can be performed immediately after the event "click OK button" in the smaller window. The portion of the application code that executes in response to a GUI event is called the *event handler*.

The state of the GUI when it first appears after the application is launched is called its *initial state*[1]. The EFG has a set of *initial events*, which can be executed in the GUI's initial state. A *length-l test case* consists of any path through l events

[1]While GUIs in general can have more than one initial state—the initial state on a particular run of the application being decided by data outside of the software, such as a configuration file—the GUIs in this work are restricted to just one initial state. Also, in this work, temporary states while starting up the GUI (e.g., splash screens) are ignored.



ng from a hat edge and chinese dynasty

Near overflowing from a hat edge and chinese dynasty

Figure 2.1: A GUI and part of its EFG

in the EFG, starting at an initial event. The *depth* of an event in the EFG is the length of the shortest test case containing the event, counting the event itself; initial events have a depth of 1.

In model-based GUI testing, the oracle information consists of a set of observed properties (e.g., title, background color) of all windows and widgets in the GUI. The *oracle procedure* compares these properties to their expected values after each event in the test case is executed. In contrast to batch-style applications (e.g., compilers), where most of the application's state tends to be hidden from the user, many GUI-based applications, sometimes called *GUI-intensive* applications, expose much of their state during execution.

The main steps in GUI testing—including reverse-engineering an EFG from a GUI, generating and executing test cases, and applying the oracle—have been automated in the GUI Testing Framework (GUITAR)[2], which was developed by the author's advisor and research group. GUITAR is used in the experiment in Chapter 4 and the empirical demonstration in Chapter 5. The applications used in both are GUI-intensive.

2.2 Characterizing faults

How should faults in software-testing studies be characterized? This is an open question. One or more of three approaches to characterizing faults are usually taken:

[2]`http://guitar.sourceforge.net`

1. Characterize faults by their origin (natural, hand-seeded, or mutation). Often, all faults in a study share a common origin, but some studies [5, 16] have compared results for faults of different origins.

2. Describe each fault and report results individually for each one [40]. This is only practical if few faults are used.

3. Calculate the "difficulty" or "semantic size" of each fault relative to the test cases used in the study and compare results for "easier" and "harder" faults [31, 43, 53].

The third approach comes closest to characterizing faults to help explain and predict results from other studies and real situations. But the "difficulty" of a fault can only be calculated relative to a set of test cases. Two different sets of test cases—e.g., a huge test pool in an empirical study and an early version of a test suite in practice, or test sets generated from two different operational profiles—would assign different "difficulty" values, and possibly different "difficulty" rankings, to a set of faults.

A fourth approach has occasionally been used:

4. Characterize faults by some measure intrinsic to them, such as the type of programming mistake [8, 7] or the fault's effect on the program dependence graph [29].

Basili and Selby [8] use a two-dimensional fault classification proposed originally by Basili and Perricone [7]. In one dimension, a fault falls into one of five categories of programming mistakes (which are "only roughly defined") [7]:

- *initialization*—"failure to initialize or reinitialize a data structure properly upon a module's entry/exit",
- *control*—a mistake that "cause[s] an 'incorrect path' in a module to be taken",
- *interface*—a mistake "associated with structures existing outside the module's local environment but which the module used" (e.g., "incorrect subroutine call"),
- *data*—"incorrect use of a data structure" (e.g., "use of incorrect subscripts for an array"), and
- *computation*—a mistake "that cause[s] a computation to erroneously evaluate a variable's value".

In the other dimension, there are two categories of programming mistakes:

- *commissive*—mistakes "present as a result of an incorrect executable statement", and
- *omissive*—mistakes "that are a result of forgetting to include some entity within a module".

With this characterization, faults are classified by manually examining them.

Harrold et al. [29] take a disparate approach. Instead of classifying faults by the type of programming mistake, they consider the change the fault induces on the program dependence graph—a model of the control dependencies and data dependencies in a program. At a coarse level, their taxonomy classifies faults as either *structural*—altering the structure of the program dependence graph—or *statement-level*—altering a statement but leaving the graph structure unchanged. Each category is further broken down by the way the program dependence graph or statement is changed. To study their fault classification, they use a tool that seeds faults of each type into C programs.

Basili and Selby [8] and Harrold et al. [29] each compare the ability of different testing techniques to detect faults, reporting the number of faults of each category detected by each technique. The fault characterization schema used by Basili and Selby [8] proves to be relevant to the testing and inspecting techniques studied—certain techniques better detected certain kinds of faults—but the characterization is labor-intensive and not entirely objective. Conversely, the schema used by Harrold et al. [29] is objective, allowing faults to be seeded automatically, but has not been shown to help explain why some faults were more likely to be detected than others. (Unfortunately, this result could not be re-evaluated in this work because no tools were available for Java software.)

In summary, fault characterization remains an open problem, but, for software-testing studies, objective and quick-to-measure characteristics have certain advantages. When described by such characteristics, large samples of faults can be characterized efficiently, and faults can be grouped into types or clusters that retain their meaning across studies and situations. Section 4.1 identifies several such characteristics, and the experiment of Chapter 4 evaluates their ability to explain why some faults are more likely to be detected than others.

A disadvantage of objective, quick-to-measure characteristics is that they cannot very well describe certain properties of interest to practitioners, such as the type of programming mistake. While the experiment in Chapter 4 does have this disadvantage, the more general approach to experiment design in Chapter 3 does not. It allows for any number of fault characteristics—including, for example, the fault classes defined by Basili and Perricone [7]—to be used.

2.3 Characterizing test suites

Test-suite characteristics and their effects on fault detection have been studied much more intensively than fault characteristics. Probably the most studied characteristic of test suites is the *technique* used to generate or recognize them. In many studies, a sample of test suites from a technique has been used to evaluate the technique empirically against other testing or validation techniques. Techniques that have been compared in this way include code reading, functional testing, and structural testing [8]; data-flow- and control-flow-based techniques [31]; regression test selection techniques [26]; and variations of mutation testing [44].

Often, the technique used to create a test suite is closely tied to the proportion of the software it covers, which in turn may affect the proportion of faults it detects. A study by Morgan et al. [39] finds that the *proportion of coverage* (measured in blocks, decisions, and variable uses) and, to a lesser extent, the *test-suite size* influence fault detection. A study of regression testing by Elbaum et al. [18] finds that, of several test-suite characteristics studied, two related to coverage—the *mean* *percentage of functions executed per test case* and the *percentage of test cases that reach a changed function*—best explain the observed variance in fault detection. In the domain of GUI testing, McMaster and Memon [36] show that some coverage criteria (call-stack and event-pair coverage) are more effective than others (event, line, and method coverage) at preserving test suites' fault-detecting abilities under test-suite reduction. In addition, certain faults are detected more consistently by some coverage criteria than by others.

Another important way in which test suites can differ is in their *granularity*—the amount of input given by each test case. Rothermel et al. [53] show that granularity significantly affects (sometimes increasing, sometimes decreasing) the number of faults detected by several regression-testing techniques. For GUI testing, Xie and Memon [65] have found that more faults are detected by test suites with more test cases, while different faults are detected by suites whose test cases are a different granularity (length). They posit that longer test cases are required to detect faults in more complex event handlers.

In summary, several studies concur that the coverage, size, and granularity of test suites can affect their ability to detect faults. The experiment in Chapter 4 bolsters the empirical evidence about these test-suite characteristics and, for the first time, looks for interaction effects between them and fault characteristics.

2.4 Characterizing software products and processes

Although this work focuses on test-suite and fault characteristics, one other kind of factor can influence fault detection in testing: characteristics of the software under test. Researchers have found that measures of the size and complexity of software help explain why some software products seem to be more testable than others. The studies by Elbaum et al. [18] and Morgan et al. [39], mentioned above for their results on test-suite characteristics, also consider software characteristics. Of the software characteristics studied by Elbaum et al. [18], the *mean function fan-out* and the *number of functions changed* together explain the most variance in fault detection. Morgan et al. [39] find that *software size*—measured in lines, blocks, decisions, or all-uses counts—contributes substantially to the variance in fault detection.

The configuration with which a software product is deployed and the system on which it is deployed can also affect defect detection in testing. Additional defects may arise because of incompatibilities between the software and the system it is running on—a common issue for Web applications, for example [17]. The number of configurations available for a product can also greatly increase the complexity of testing [48].

The process by which the software is developed may also affect fault detection in testing by affecting the faults present in the software to be tested. One would expect fewer faults, and perhaps a different distribution of faults, to exist at the time of testing if other defect-removal or defect-prevention techniques had been applied prior to testing.

2.5 Models of failure propagation

Part of this work is concerned with understanding the conditional probability that a test suite detects a fault, given that the test suite covers the code containing the fault. As Chapter 4 explains, this conditional probability separates the concerns of fault detection and fault coverage. It shows how susceptible a fault is to detection, regardless of whether the code it lies in is frequently or rarely executed.

The relationship between fault detection and coverage has previously been viewed from the perspective of the RELAY model and PIE analysis. The RELAY model of Richardson and Thompson [51] traces the steps by which a fault in source code leads to a failure in execution: from the incorrect evaluation of an expression to unexpected internal states to unexpected output. RELAY is the basis for propagation, infection, and execution (PIE) analysis of program testability proposed by Voas [64]. PIE uses the probability that a given program element is executed and the probability that a fault in that element is detected to determine how testable that element is.

Like RELAY, the current work is concerned with the relationship between faults and failures. This work, however, ignores internal program state. In contrast to this work's empirical approach, Richardson and Thompson use RELAY to compare test adequacy criteria analytically.

PIE differs from this work because it estimates execution probabilities with respect to some fixed input distribution and infection probabilities with respect to some fixed distribution of faults. In contrast, the current work studies how variations in the input distribution (test suite) and the type of fault affect the test suite's likelihood of executing and detecting the fault.

2.6 Adaptive testing techniques

In the proposed scenarios for regression testing in Chapter 5, information about past regression-testing iterations is used to inform and improve future iterations. However, the idea that feedback from iterative processes can be used to improve them is not new. This principle underlies traditional regression-testing techniques that carry over coverage information from one iteration to the next. On a smaller scale, iterative learning has been used to generate test cases. On a larger scale, it is the basis of the Improvement Paradigm.

In regression testing, one key idea has been that test cases being reused from a previous version only need to be re-executed if they exercise part of the software that has changed. As Harrold et al. [28] and Rothermel and Harrold [54], among others, have shown, this can be accomplished by examining coverage data collected during testing of previous versions. Another key idea has been that coverage data can be used to eliminate redundant test cases—those that cover no parts of the program uniquely [28]. In each of the approaches just described, the aim is to reduce the cost of re-executing a test suite without reducing the number of faults it detects. Another approach, test-suite prioritization, aims to re-execute test cases in an order that maximizes the rate at which they detect faults. Described by Rothermel et al. [55], several prioritization techniques similarly use coverage data from previous iterations of regression testing.

In the Improvement Paradigm, the cycle of observing and improving processes occurs at the level of software projects. When planning a new project, an organization draws from its experience with past projects, taking into account the context of the new project (e.g., available resources, product size). During and after the project's lifetime, the knowledge gained about the methods and techniques used by the project is structured for reuse and added to an experience base [6].

Several test-case-generation techniques similarly operate in cycles of observation and improvement. For these, the objects of interest are not software projects or regression-test suites, but individual test cases. Interleaving test execution with test-case generation, these techniques progressively find test inputs to cover application states that have not yet been tested. The selection of test inputs may be guided by various kinds of information gained from test execution, including data dependencies [21], numerical constraints [27], and GUI-state changes [67]. Cai et al. [12] also use feedback from test execution to select additional test cases, but their goal is to accurately estimate defect-detection rates using as few test cases as possible.

Like other adaptive techniques for software testing and software engineering, this work uses feedback from an iterative process to improve later iterations. But unlike existing techniques for regression testing, which try to reduce the cost or time to detect a fixed set of faults, the scenarios for regression testing proposed in this work show how to detect more faults or more important faults.

Chapter 3

Methodology to Account for Fault Characteristics in Empirical Studies of Software Testing

The most important contribution of this work is to provide a new way of doing empirical studies of software testing, particularly evaluations of testing techniques. It is a template for experiment designs, or *methodology* for short. The methodology enables experimenters to compare the effectiveness of different testing techniques and, at the same time, to measure the influence of other factors on the results.

3.1 Requirements

In a given software product, two kinds of factors can influence a testing technique's ability to detect a fault: characteristics of the test suite used (other than testing technique) and characteristics of the fault. Thus, the methodology cannot account for *just* fault characteristics, even though they are the focus of this work. *It must simultaneously account for both test-suite and fault characteristics.* In other words, it must be able to show that certain kinds of test suites generally detect more faults, and certain kinds of faults are generally more susceptible to detection, and certain kinds of test suites are better at detecting certain kinds of faults.

The key observation leading to the methodology is that fault characteristics and test-suite characteristics, including the testing technique, can be accounted for simultaneously with the right kind of multivariate statistical analysis. The experiment in this work uses logistic-regression analysis, although the methodology remains general so that other types of analysis could be substituted, depending on the goals of the experiment. (Possible alternatives include latent class analysis [25] and Bayesian belief networks [20].) Hence, the methodology must at least satisfy the requirements of logistic regression.

As Section 4.2.2 will explain, the input to logistic regression is a data set in which each data point consists of a vector of some independent variables and a binomial dependent variable. The output is a vector of coefficients, which estimate the strength of each independent variable's effect on the dependent variable. If each data point consists of a vector of characteristics of a ⟨*test suite, fault*⟩ pair and a value indicating whether the test suite detects the fault, then the output is just what we want: an estimate of each characteristic's effect on the likelihood that a given test suite detects a given fault. (For example, in the special case where the independent variables are just the testing technique and one fault characteristic, logistic regression would show how likely each technique would be to detect a hypothetical fault having any given value for the fault characteristic.)

Logistic-regression analysis is a flexible technique, able to model different kinds of relationships between the independent variables and the dependent variable. The main requirement is that *each data point must be independent of the other data points*. If each data point consists of characteristics of a ⟨*test suite, fault*⟩ pair, then the same test suite or the same fault cannot be used in more than one data point. This leads to a somewhat unusual (for software-testing studies) experiment design,

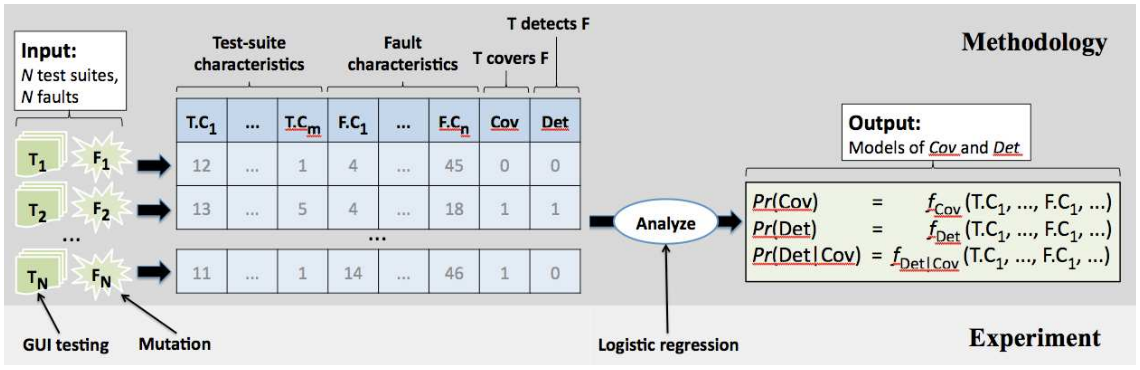


Figure 3.1: Methodology and experiment procedure

in which each test suite is paired with a single fault.

3.2 Methodology

Now that the requirements of the methodology have been described, Figure 3.1 illustrates the methodology (darker gray box) and its instantiation in the experiment in the next chapter (lighter gray box). For now, let us focus on the methodology itself, moving from left to right through the figure. The objects of study are a sample of N test suites $(T_1, \ldots, T_N)$ for one or more software applications and a sample of N faults $(F_1, \ldots, F_N)$ in those applications. Each test suite is paired with exactly one fault (T_1 with F_1, T_2 with F_2, etc.) to form a $\langle$*test suite, fault*$\rangle$ pair.

Each test suite in a pair is run to see whether it (1) executes (covers) the piece of code containing the fault in the pair and (2) if so, whether it detects the fault. These facts are recorded in the dependent variables, Cov (which is 1 if the suite covers the fault, 0 otherwise) and Det (which is 1 if the suite detects the fault, 0 otherwise). In addition, certain characteristics of each fault $(F.C_1, \ldots, F.C_n)$ and each test suite ($T.C_1, \ldots, T.C_m$) are recorded. Determined by the experimenters, these characteristics may include the main variable of interest (e.g., testing technique) as well as factors that the experimenters need to control for. All of these characteristics together comprise the independent variables. As the next part of Figure 3.1 shows, the data collected for the independent and dependent variables form a table structure. For each of the N $\langle$*test suite, fault*$\rangle$ pairs, there is one data point (row in the table) consisting of a vector of values for the independent and dependent variables.

The data points are analyzed, as the right half of Figure 3.1 shows, to build one or more statistical models of the relationship between the independent variables and the dependent variables. The models estimate the probability that a given test suite (i.e., a given vector of values for $T.C_1, \ldots, T.C_m$) covers or detects a given fault (i.e., a given vector of values for $F.C_1, \ldots, F.C_n$) as a function of the test-suite and fault characteristics. Additionally, if only data points with $\mathsf{Cov} = 1$ are considered, then models of Det can be built from them to estimate the conditional probability of fault detection given fault coverage ($\Pr(\mathsf{Det}|\mathsf{Cov})$).

The methodology offers experimenters several choices: in the test-suite and fault characteristics to use as independent variables, in the way the samples of test suites and faults are provided, and in the analysis technique. For this experiment in this work, the choice of independent variables is explained in Section 4.1. The test suites were generated randomly using a GUI-testing technique (Section 4.2.1.2), and the faults were generated by mutation of single lines of source code (Section 4.2.1.3). Because all faults in the experiment were confined to one line, a fault was considered to be covered by a test suite if the line containing it was covered. The chosen analysis technique was logistic regression (Section 4.2.2).

The methodology could be trivially extended to consider a broader class of defect characteristics, including the characteristics of the failures caused by a fault. However, the preceding description emphasized fault characteristics because they are the focus of the experiment in the next chapter.

Chapter 4

Experiment Applying the Methodology

This experiment applies the methodology described in the previous chapter, as shown in Figure 3.1. It serves multiple purposes:

- as a stand-alone experiment, testing hypotheses about the influence of test-suite and fault characteristics on fault detection;
- as a proof of concept, showing that the methodology is viable;
- as a concrete example for potential users of the methodology to follow; and
- as a validation of the fault characterization described in Section 4.1, showing empirically that the fault characteristics chosen *can* affect faults' susceptibility to detection.

The data and artifacts from the experiment have been made available to other researchers as a software-testing benchmark (Section 4.2.1).

This experiment significantly extends, and resolves major problems with, a preliminary study of several test-suite characteristics and just two fault characteristics [58, 60]. (The preliminary study is presented in Appendix A.) The results of the preliminary study raised intriguing questions about the relationship between the execution (coverage) of faulty code and the detection of faults: Are certain kinds of faults more likely to be detected just because the faulty code is more likely to be covered during testing? Or are these faults harder to detect even if the code is covered? This work pursues those questions by studying not just the likelihood of detecting faults, but the likelihood of detecting them *given that* the faulty code has been covered. This perspective echoes existing models of failure propagation (Section 2.5), but its use to study faults empirically is unprecedented; our novel methodology makes it possible.

For each fault characteristic studied in the experiment, the experiment tests the following null hypotheses:

- H1: The characteristic does not affect a fault's likelihood of being detected by a test suite.

- H2: No interaction effect between the characteristic and a test-suite characteristic affects a fault's likelihood of being detected by a test suite. (Certain kinds of faults are not more likely to be detected by certain kinds of test suites.)

- H3: The characteristic does not affect a fault's likelihood of being detected by a test suite, given that the test suite covers the faulty code.

- H4: No interaction effect between the characteristic and a test-suite characteristic affects a fault's likelihood of being detected by a test suite, given that the test suite covers the faulty code.

Analogous hypotheses are tested for each test-suite characteristic. The main concern of the experiment, however, is the fault characteristics because they need to be evaluated to determine whether they really are relevant to software testing.

Like any experiment, this one restricts itself to a limited domain of applications, testing techniques, and faults. In choosing this domain, important factors were the cost and replicability of the experiment. Since existing data did not fit the requirements of this experiment, and since creating test suites and faults by hand is expensive and hard to replicate, test suites and faults needed to be generated automatically. For faults, this led us to choose the domain of mutation faults. For test suites (and consequently applications), the author's experience with automated GUI testing made this domain an obvious choice. As Section 2.1 explained, GUI testing is a form of system testing in which test cases are generated by traversing an event-flow-graph (EFG) model of a GUI. Considering that many computer users today use GUIs exclusively and have encountered GUI-related failures, research on GUIs and GUI testing is timely and relevant.

4.1 Test-suite and fault characterization

The independent variables in this experiment are characteristics of faults and test suites hypothesized to affect the probability of fault detection. Since there is currently no standard way to choose these characteristics, the selection was necessarily somewhat improvised but was driven by earlier research (Sections 2.2 and 2.3). Although the literature does not directly suggest viable fault characteristics, it does clearly point to certain test-suite characteristics. Because the test suites and faults in this experiment were generated automatically, characteristics related to human factors did not need to be considered. To make this experiment practical to perform, replicate, and apply in practice, only characteristics that can be measured objectively and automatically were considered.

It should be noted that, while truly objective measures of faults—measures of intrinsic properties, independent of any test set used to measure them—would be derived from static analysis, static analysis of GUI-based applications is still under development [57]. In this experiment, most characteristic measures were derived from execution data from the *test pool*, the set of all test cases used in the experiment. This is not entirely objective because a different test pool would result in different measurements. Also, it threatens the assumption of independent sampling of data points—an issue examined in Section 4.4.6. However, measures that are closely tied to the test pool (e.g., those averaged across the test cases in the pool) were avoided. The test pool can be seen as an instrument used to estimate the true values of the measures (e.g., the minimum number of GUI events that must be executed to reach the faulty code). As the test pool size increases, the estimates converge to the true values.

Each characteristic describes some property of a fault or a test suite, such as the degrees of freedom in execution of the faulty code or the proportion of the application covered by the test suite. There are often multiple metrics to measure a characteristic, and prior to analyzing the data it is not clear which best predicts Cov or Det. The rest of this section lists the fault and test-suite metrics explored in this experiment, organized by the characteristic they are intended to measure. These are summarized in Table 4.1.

Table 4.1: Fault and test-suite characteristics studied

Characteristic		Metric	Definition
Fault	Method of creation	F.MutType	1 if a method-level mutant, 0 if a class-level mutant
	Distance from initial state	F.CovBef	Est. minimum lines covered before first execution of faulty line, normalized by total lines
		F.Depth	Est. minimum EFG depth of first event executing faulty line in each test case, normalized by EFG depth
	Repetitions	F.SomeRep	1 if est. minimum executions of faulty line by each executing event is > 0, 0 otherwise
		F.AllRep	1 if est. maximum executions of faulty line by each executing event is > 0, 0 otherwise
	Degrees of freedom	F.MinPred	Est. minimum EFG predecessors of events executing faulty line, normalized by total events in EFG
		F.MaxPred	Est. maximum EFG predecessors of events executing faulty line, normalized by total events in EFG
		F.MinSucc	Est. minimum EFG successors of events executing faulty line, normalized by total events in EFG
		F.MaxSucc	Est. maximum EFG successors of events executing faulty line, normalized by total events in EFG
		F.Events	Est. number of distinct events executing faulty line, normalized by total events in EFG
	Size of event handlers	F.MinWith	Est. minimum lines covered in same event as faulty line, normalized by total lines
		F.MaxWith	Est. maximum lines covered in same event as faulty line, normalized by total lines
Test suite	Granularity	T.Len	Length (number of events) of each test case
	Size	T.Events	Number of events, normalized by total events in EFG
	Proportion of coverage	T.Class	Percent of classes covered
		T.Meth	Percent of methods covered
		T.Block	Percent of blocks covered
		T.Line	Percent of lines covered
		T.Pairs	Percent of event pairs in EFG covered
		T.Triples	Percent of event triples in EFG covered

4.1.1 Fault characteristics

One fault characteristic studied is the **method of creation**, which for this experiment is some form of mutation. Although there are too many mutation operators to study each individually, they fall into two categories: class-level (e.g., changing the type of a data member) and method-level (e.g., inserting a decrement operator at a variable use). Class-level and method-level mutations were previously studied by Strecker and Memon [58, 60], whose results were inconclusive but suggested that class-level and method-level faults may be differently susceptible to detection. The label for the metric of mutation type is F.MutType.

Another fault characteristic is the **distance of faulty code from the initial state**. Faults residing in code that is "closer", in some sense, to the beginning of the program are probably easier to cover and may be easier to detect. One metric measuring this is the minimum number of source-code lines that must be covered before the faulty line is executed for the first time (F.CovBef). This can be estimated by running the test pool with program instrumentation to collect coverage data.

In a GUI-based application, a faulty line may lie in the event handler of one or more events. These events can be associated with the line by collecting coverage data for each event in each test case of the test pool. The minimum EFG depth of the events associated with a faulty line (F.Depth) is another way to measure the distance of the line from the initial state.

The **repetitions in which the faulty code is executed** may affect fault detection. Faults that lie in code that, when executed, tends to be executed multiple times by iteration or recursion may be easier to detect. Since, for the applications studied, the exact number of times a line is executed depends closely on the test case, two binomial metrics are studied instead. One is whether the the line is *ever* executed more than once by an event handler (F.SomeRep). The other is whether the line is *always* executed more than once by an event handler (F.AllRep).

Another fault characteristic that may affect fault detection is the **degrees of freedom in execution of the faulty code**. In GUI-based applications, an event handler can typically be executed just after any of several other event handlers. Faulty code executed by an event that can be preceded or succeeded by many other events may be easier to cover, and it is not clear whether it would be more or less susceptible to detection. The minimum or maximum number of event predecessors or successors associated with a faulty line (F.MinPred, F.MaxPred, F.MinSucc, F.MaxSucc) can be estimated by associating coverage data from the test pool with the EFG. Faulty code executed by more events may also be easier to cover and either more or less susceptible to detection. The number of events executing the faulty code (F.Events), too, can be estimated with coverage data from the test pool.

Morgan et al. [39] report that program size affects fault detection in testing, so the **size of the event handler(s) that execute a faulty line** may similarly have an effect. Event-handler size can be measured as the minimum or maximum number of lines covered by each event handler that executes the faulty line (F.MinWith, F.MaxWith).

4.1.2 Test-suite characteristics

For test suites, one interesting characteristic is the **granularity of test cases**—the amount of input provided by each test case. In GUI testing, granularity can easily be measured by the length (number of events) of a test case (T.Len). In this experiment, the length of the test cases in a test suite could be measured either by taking the average of different-length test cases in a suite or by constructing each suite such that its test cases have a uniform length. The latter approach is chosen because it has a precedent in previous work [53, 65] and because the assumption of uniform-length test cases, though unnecessary, is not unrealistic for model-based GUI testing. Suites made up of longer test cases may reach "deeper" program states, enabling them to cover and detect more faults [65].

Clearly, the characteristic of **test-suite size** can affect fault detection: larger test suites are likely to cover and detect more faults. An important question studied in this experiment is whether they do so when other factors, such as the suite's coverage level, are controlled for. In some studies, test-suite size is measured as the number of test cases. But for this experiment, since different suites have different test-case lengths, a more meaningful metric is the total number of events in the suite, which is the product of the test-case length and the number of test cases (T.Events).

Another test-suite characteristic that can affect fault detection is the **proportion of the application covered**. Obviously, the more of an application's code a test suite covers, the more likely it is to cover a specific line, faulty or not. It may also be likely to detect more faults [39]. The proportion of coverage may be measured by any of the myriad coverage metrics proposed over the years. This experiment considers several structural metrics—class (T.Class), method (T.Meth), block (T.Block), and line coverage (T.Line)—because of their popularity and tool support. For GUI-based applications, additional coverage metrics based on the event-flow graph (EFG) are available. Event coverage (coverage of nodes in the EFG) turns out *not* to be a useful metric for this experiment because each suite is made to cover all events. However, coverage of event pairs (EFG edges or length-2 event sequences; T.Pairs) and event triples (length-3 event sequences; T.Triples) is considered. (Longer event sequences could have been considered as well, but length 3 seemed a reasonable stopping point for this experiment. Since this experiment does show coverage of length-2 and length-3 sequences to be influential variables, future experiments can study coverage of longer sequences.)

4.2 Procedure

4.2.1 Data collection

The first stage of the experiment involves building and collecting data from a sample of test suites and a sample of faults. One of the contributions of this work is to make data and artifacts from this experiment—products of thousands of computation-hours and hundreds of person-hours—available to other researchers as a software-testing benchmark[1]. The artifacts—including source code and configuration files for

[1]`http://www.cs.umd.edu/~atif/Benchmarks/UMD2007b.html`

Table 4.2: Applications under test

	Lines	Classes	Events	EFG edges	EFG depth	Data points
CrosswordSage 0.3.5	2171	36	98	950	6	2230
FreeMind 0.7.1	9382	211	224	30146	3	970

the applications under test, GUI models created by GUITAR and tailored by the author, and scripts used by the author—describe the experiment setup more precisely than prose ever could. Furthermore, the test suites, faults, and data about them provided in the benchmark can be reused by researchers to perform new studies.

4.2.1.1 Applications under test

Two medium-sized, open-source applications were studied: CrosswordSage 0.3.5[2], a crossword-design tool; and FreeMind 0.7.1[3], a tool for creating "mind maps". (A screenshot of CrosswordSage was shown in Figure 2.1.) Both are implemented in Java and rely heavily on GUI-based interactions. Table 4.2 gives each application's size as measured by executable lines of code, classes, and GUI events modeled in testing; the depth and number of edges of its EFG; and the number of data points ($\langle$*test suite, fault*$\rangle$ pairs) generated for it.

GUI testing of the applications was performed with tools in the GUITAR suite [65]. To make the applications more amenable to these tools, a few modifications were made to the applications' source code and configuration files (e.g., to make file choosers open to a certain directory and to disable automatic saves). The modified applications are referred to as the *clean-uninstrumented* versions. Each

[2]`http://crosswordsage.sourceforge.net`

[3]`http://freemind.sourceforge.net`

application was made to have the same configuration throughout the experiment. A simple input file was created for each application; it could be opened by performing the correct sequence of events on the GUI. Using GUITAR, an EFG was created for each application. GUI events that could interfere with the experiment (e.g., events involved in printing) were removed from the EFG. The applications, configuration files, input files, and EFG are provided in the software-testing benchmark described at the beginning of Section 4.2.1.

To collect coverage data, each clean-uninstrumented application was instrumented with Instr[4] and Emma[5]. The instrumented applications are referred to as the *clean versions*. Instr reports how many times each source line was executed, while Emma reports (among other information) the proportion of classes, methods, blocks, and lines covered. Coverage reports from Instr were collected after each event in a test case; a report from Emma was collected at the end of the test case.

To identify lines in *initialization code*—code executed before any events are performed—an "empty" test case (with no GUI events) was run on each application and coverage reports were collected. The initialization code was treated as an initial event in the EFG having depth 0, no in-edges, and out-edges extending to all depth-1 events.

[4]`http://www.glenmccl.com/instr/index.htm`

[5]`http://emma.sourceforge.net`

4.2.1.2 Test suites

The sample of test suites used in this experiment should be large and replicable (not influenced by the skill of the tester). These criteria suggest an automated testing technique. For this experiment, a form of automated GUI testing (Section 2.1) is chosen.

Not only should the sample of test suites be large and replicable, but it should also be an independent sample. In other words, the test suites in different ⟨*test suite, fault*⟩ pairs should not be related to one another. For this reason, a unique set of tests was generated for each ⟨*test suite, fault*⟩ pair. (The alternative would be to form each test suite by selecting, with replacement, a subset of a large pool of test cases, as was done in the preliminary study in Appendix A.)

Each test suite satisfies two requirements. First, it covers every event in the application's EFG at least once. This is to avoid obvious conclusions—i.e., that faults in code only executed by one event cannot be covered or detected if the event is not executed. Second, its test cases are all the same length. This is so that test-case length can be studied as an independent variable. The length must be greater than or equal to the depth of the EFG to ensure that all events can be covered. The maximum test-case length studied in this experiment is 20.

Model-based GUI testing has the advantage of being automated, but this is tempered by the fact that existing tools for generating and executing GUI test cases are immature. Also, the EFG is only an approximation of actual GUI behavior; because of enabling/disabling of events and other complex behavior in the actual

GUI, not every test case generated from the EFG model is executable [67]. For these reasons, each test suite must be generated carefully to ensure that every test case runs properly.

Each test suite was generated in two stages.

- *Stage 1:* First, a test-case length L between the EFG depth and 20 (inclusive) is randomly chosen. The list E of events that remain to be covered is initialized to include all events in the EFG. A length-L test case is generated to cover a randomly-selected event $e \in E$. Then the test case is run on the application. If it runs successfully, then e and all other events it covers are removed from E; otherwise, it is discarded and a new test case is generated. Test cases continue to be generated until E is empty.

- *Stage 2:* The mean and variance of the total number of events in the test suites generated in Stage 1 scales with test-case length—an undesirable feature for this experiment, in which the number of events and the test-case length should be independent. Stage 2 adds random test cases to the suite to make test-suite size and test-case length independent. In preparation for the experiment, 100 test suites of each test-case length were generated for each application using the procedure in Stage 1. The number of events per suite was observed to be approximately normally distributed for each length; a mean and variance for each normal distribution was estimated from these test suites. During the experiment, Stage 2 for each test suite begins by calculating the quantile on the normal distribution for length L corresponding to the suite's number of events after Stage 1. The number of events corresponding to the same quantile on the normal distribution for length 20 is then found; this becomes the target number of events for the suite. Test cases are generated by randomly traversing the EFG and are added to the suite to reach the target number.

It should be noted that, while both stages of test-case generation were randomized, they favored certain events. Because of the complex structures of the EFGs of the applications under test, some events were more likely than others to be encountered in a random traversal. For example, events available in the initial state of the GUI appeared in more test cases, and more often per test case, than other events.

4.2.1.3 Faults

An important consideration in any empirical study of fault detection is whether to use natural, manually-seeded, or automatically-seeded faults [5, 16]. To achieve this experiment's large sample size (Table 4.2) with the resources available, using automatically-seeded faults was the only feasible option. Even apart from resource considerations, automatically-seeded faults offer some advantages for experimentation: unlike natural or hand-seeded faults, automatically-seeded faults are not influenced by the person (accidentally or intentionally) seeding the fault. The tool MuJava[6] was used to seed mutation faults (syntactically-small changes to the source code, such as replacing one operator or identifier with another; a full list of the mutation types is available at the referenced URL). Although the use of mutation faults

[6] `http://www.ise.gmu.edu/~ofut/mujava/`

is a threat to external validity, it should be noted that in at least some cases mutation faults turn out to be about equally difficult to detect as natural faults [5, 16], and a fault's syntactic size has little to do with its difficulty of detection [43].

Using MuJava, all possible faults within MuJava's parameters were generated for each application. Of those, faults that spanned multiple lines and faults in application classes corresponding to events deliberately omitted from the application's EFG (e.g., `crosswordsage.PrintUtilities`; see Section 4.2.1.1) were omitted. Faults not inside methods (i.e., in class-variable declarations and initialization) were also omitted because their coverage is not tracked by Emma or Instr and because most extra-method faults turned out to be either trivially detectable (e.g., removing a necessary variable initialization) or not faults at all (e.g., removing an unnecessary variable initialization). For CrosswordSage, all 2230 of the remaining faults were used. For FreeMind—which requires much more time to generate and run each test suite because of its larger GUI—1000 of the 5729 single-line faults in acceptable classes were initially selected at random, and of those the 970 faults located inside methods were used. (It should be noted that the number 970 is no more arbitrary than 1000.)

For each fault, MuJava generates a source file, which differs from the clean-uninstrumented source code only by the single mutation, and the corresponding class file(s). A *faulty version* of the application can be created by substituting (in the classpath) one fault's class file(s) for the clean class file(s). Having just one fault per faulty version is of course unrealistic, but it is a ubiquitous practice in empirical studies of software testing because it makes it easy to determine whether a test case detects a particular fault. While it is not yet clear how much this practice affects experiment results, a somewhat related study showed that test suites that detected a high percentage of a sample of mutation faults also detected a high percentage of more complex faults (composed of two mutations along the same program path) [42].

4.2.1.4 Measurement of independent and dependent variables

A test suite was generated for each fault in the sample. For each ⟨*test suite, fault*⟩ pair, each test case was executed on the clean version of the application and, if it covered the line containing the fault, on the faulty version. Test cases were executed by GUITAR on a cluster of Linux machines. Most of the computation time for the experiment was spent running test cases. With the parameters set for GUITAR, a test case of length L took at least $5 + 2.5L$ seconds to run on the clean version. For CrosswordSage, test suites consisted of 18 to 101 test cases (306 to 680 events); for FreeMind, 45 to 343 test cases (770 to 1178 events).

To determine whether a test suite covered a faulty line (Cov), the coverage report from Instr was examined. To determine whether a test suite that covered a faulty line also detected the fault (Det), the oracle information collected by GUITAR for the clean and faulty versions was compared.

When the experiment was run, some false reports of fault detection were anticipated. Because of timing problems in the current version of the test-case-replayer component of GUITAR (an issue in other GUI-testing tools as well [1]), test cases sometimes fail to run to completion, making it appear as if a fault has been detected when really it has not been. In addition, GUITAR by default detects even trivial differences in oracle information, such as in the coordinates of GUI components, which may not actually indicate fault detection.

To reduce false reports of fault detection, usually-trivial differences (e.g., any changes to the size or absolute position of GUI components) were ignored. The *test-step number* (e.g., 4 for the 4th event in the test case) of the first non-trivial difference was determined. To find and fix false reports of fault detection, the test-step number at which fault detection was reported was checked against coverage reports to make sure that no fault was supposedly detected before its line had been covered. For CrosswordSage, only one test case supposedly detected a fault before covering it (but this did not affect Det for the suite), and no false reports of fault detection were found in a manual inspection of the first 100 ⟨*test suite, fault*⟩ pairs, so no further checking was done. For FreeMind, there were more false reports of fault detection, so all ⟨*test suite, fault*⟩ pairs with $\mathsf{Det} = 1$ were manually inspected and corrected.

Some metrics of the faults and test suites could be measured before test execution. For faults, the mutant type was apparent from MuJava's output. The test-case length, size, event-pair coverage, and event-triple coverage of test suites were also known prior to execution. The remaining metrics were calculated from the coverage reports generated by Instr and Emma during execution. To allow comparison between results for CrosswordSage and FreeMind, metrics that vary with application size (e.g., number of events in a test suite, number of lines covered before a faulty line) were normalized (e.g., by number of events in the EFG, by number of executable lines in the application).

4.2.2 Data analysis

The goal of this experiment is to evaluate the strength and significance of the effects of test-suite and fault characteristics on coverage (Cov) and detection (Det) of faults. For data of this structure, logistic regression [3, 25, 52] is the most popular analysis technique (although other techniques, such as Bayesian belief networks, are available [20]). Logistic regression is commonly used to evaluate the effects of several explanatory variables on a binomial response variable (e.g., the effects of race and medication usage on the presence of AIDS symptoms [3]). It has occasionally been used in software-testing research [10, 22], although never to study test-suite and fault characteristics simultaneously. Given a data set, logistic-regression analysis finds the function (in a certain class of functions) that best describes the relationship between the explanatory (independent) variables and the probability of the response (dependent) variable for that data set.

Logistic regression is so named because it uses the *logit* function,

$$\text{logit}(x) = \log\left(\frac{x}{1-x}\right),$$

to map probabilities—values between 0 and 1—onto the entire range of real numbers. For a dependent variable Y and a vector of independent variables $\vec{X}$, the logistic-regression model has the form

$$\text{logit}(\Pr(Y)) = \alpha + \vec{\beta} \cdot \vec{X}. \tag{4.1}$$

The intercept term α is related to the overall probability of Y, and, as explained

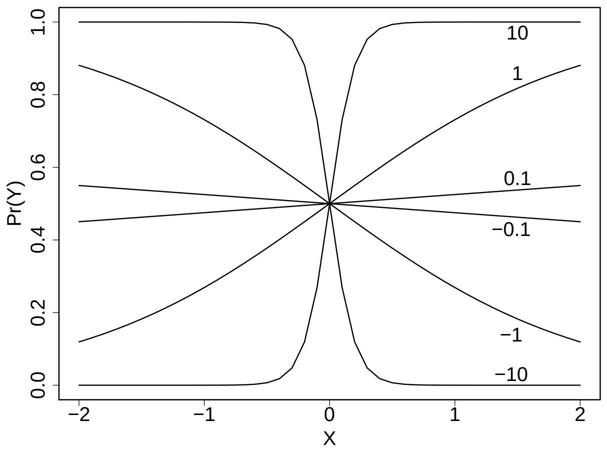


Figure 4.1: Predicted probabilities for example logistic-regression models

below, the coefficients $\vec{\beta}$ show the strength of relationship between each element of $\vec{X}$ and Y. Note that logit(Pr(Y)) equals the log of the odds of Y. Rewritten, Equation 4.1 expresses the probability of the dependent variable as a function of the independents:

$$\Pr(Y) = \frac{\exp(\alpha + \vec{\beta} \cdot \vec{X})}{1 + \exp(\alpha + \vec{\beta} \cdot \vec{X})}. \tag{4.2}$$

Figure 4.1 plots this function for $\vec{X} = X$, $\alpha = 0$, and $\vec{\beta} = \beta \in \{-10, -1, -0.1, 0.1, 1, 10\}$.

In logistic-regression analysis, a data set consists of a set of data points, each a vector of values for $\vec{X}$ (here, test-suite and fault characteristics) paired with a value for Y (here, Cov or Det). The goal of logistic regression is to find values for the intercept α and coefficients $\vec{\beta}$ that maximize the likelihood that the set of observed values of Y in the data set would have occurred given α, $\vec{\beta}$, and the observed values of $\vec{X}$. The process of choosing values for α and $\vec{\beta}$ is called *model fitting* and is accomplished by a maximum-likelihood-estimation algorithm.

Coefficients in a logistic-regression model indicate the magnitude and direction of each independent variable's relationship to the log of the odds of the dependent variable. If $\beta_i = 0$, then there is no relationship between X_i and Y; if $\beta_i < 0$, then the odds and probability of Y decrease as X_i increases; if $\beta_i > 0$, then the odds and probability of Y increase as X_i increases. However, the increase or decrease in the odds of Y is multiplicative, not additive; it is a factor not of β_i but of $\exp(\beta_i)$. More precisely, the quantity

$$OR = \exp(\beta_i \Delta) \tag{4.3}$$

(called the *odds ratio*) is an estimate, calculated during model fitting, of the ratio of the odds of Y when $X_i = x_i + \Delta$ to the odds of Y when $X_i = x_i$, when all other $X_j \in \vec{X}$ are held constant.

Associated with each coefficient is a p-value for the chi-square test of deviance, a statistical test of whether the independent variable is actually related to the dependent variable, or whether the apparent relationship (non-zero coefficient) is merely due to chance. A p-value ≤ 0.05, indicating that there is only a 5% chance of seeing a fitted coefficient value that extreme under the null hypothesis, is typically considered to be statistically significant. For exploratory analysis (as in this experiment), p-values ≤ 0.10 may also be considered.

4.2.2.1 Data sets

For each application, three data sets were constructed. These are summarized in Table 4.3. The first data set, AllFaults, includes all ⟨*test suite, fault*⟩ pairs and all test-suite metrics, but only those fault metrics that can be computed without using execution data. This is because some faulty lines were not covered by any

Table 4.3: Data sets

Data set	Description
AllFaults	All ⟨*test suite, fault*⟩ pairs All test-suite metrics Fault metrics not requiring execution data
PoolCovFaults	Pairs where test pool covers fault All test-suite metrics All fault metrics
SuiteCovFaults	Pairs where test suite covers fault All test-suite metrics All fault metrics

test case in the experiment, so no execution data was available for them. (Even though the test suites cover every GUI event, they do not cover every line of code.) The second data set, PoolCovFaults, includes all test-suite and fault metrics but only those ⟨*test suite, fault*⟩ pairs where the fault was covered by at least one test case in the test pool (not necessarily in that test suite). The third data set, SuiteCovFaults, includes all test-suite and fault metrics but only those ⟨*test suite, fault*⟩ pairs where the test suite covered the faulty line. This data set is used to understand the conditional probability of Det given that $\mathsf{Cov} = 1$.

4.2.2.2 Model fitting

For each data set, two kinds of logistic-regression models were fitted: univariate and multivariate. Each univariate model, which assesses the effect of an independent variable X_i by itself on the dependent variable Y, has the form

$$\text{logit}(\Pr(Y)) = \alpha + \beta_i X_i.$$

Each multivariate model, which assesses the contribution of each independent variable toward explaining the dependent variable in the context of the other independent variables, has the form in Equation 4.1, with $\vec{X}$ consisting of some subset of the independent variables for the data set. For the AllFaults and PoolCovFaults data sets, Y can be either Cov or Det; for SuiteCovFaults, Y can only be Det since Cov is always 1. Before model-fitting, all non-categorical data was centered—the mean was subtracted. Model-fitting and other statistical calculations were performed with the R software environment[7].

A potential problem in fitting multivariate logistic-regression models—for which other studies of fault detection have been criticized [20]—is that strongly correlated ("multicolinear") independent variables can result in models with misleading coefficients and significance tests. If two or more multicolinear variables are included as parameters in a multivariate model, then none may appear to be statistically significant even if each is significant in its respective univariate model. Also, the fitted coefficients may be very different from those in the univariate models, even changing signs. Although some correlation among model parameters is acceptable—indeed, the intention of multivariate analysis is to control for such correlation—for the purposes of this work it is desirable to avoid serious multicolinearity.

To avoid multicolinearity, as well as to provide multivariate models that are small enough to comprehend, a subset of the metrics in Table 4.1 needs to be selected for each model. There is no standard way to do this; to some extent, it is a process of trial and error [56]. Only metrics that are statistically interesting in their univariate models ($p <= 0.10$) are considered. Correlations among these metrics turn out to be complex. The strongest correlations are, not surprisingly, within groups of

[7]`http://www.r-project.org/`

metrics measuring the same characteristic, but not all variables in each group are strongly correlated. For example, T.Class, T.Meth, T.Block, and T.Line are strongly correlated with each other but not as correlated with T.Pairs. Correlations also arise between groups, for example between T.Pairs and T.Len, and vary in strength across data sets. To re-group the data according to correlation, principal-component analysis was tried, but it proved unhelpful: many metrics did not fall neatly into one component or another. For each data set, then, the problem of selecting the subset of metrics that form the best-fitting multivariate model for the dependent variable, while not being too strongly correlated to each other, was an optimization problem. Because there were few enough metrics, a brute-force solution could be applied. A program was written in R to fit logistic-regression models using various combinations of metrics, and the model with the lowest AIC (see Section 4.2.2.3) and without severe multicolinearity—i.e., unexpected coefficient signs—was chosen. This is the *main-effects* model.

Each main-effects model was expanded to include interaction effects of interest—namely, those between a test-suite metric and a fault metric. The models were then reduced by stepwise regression based on AIC to eliminate independent variables and interactions whose contribution toward explaining the dependent variable is negligible. The result is the *interaction-effects* model. This is the multivariate model presented in the next section.

4.2.2.3 Goodness of fit

To evaluate the multivariate models' goodness of fit to the data, several measures were used. One was deviance, which indicates how much of the variance in the dependent variable fails to be explained by the model[8]; lower deviance means better fit. (R^2, a similar measure for linear-regression models that is more directly related to variance, is not applicable to logistic-regression models.) The deviance can be compared to the null deviance, which is the deviance of a model consisting of just a constant term (intercept). Another goodness-of-fit measure used was Akaike's "an information criterion" (AIC), a function of the deviance and the number of independent variables in the model that balances model fit with parsimony (fewer variables); lower AIC is better.

Other measures of goodness of fit used, which may be more familiar outside the statistics community, were sensitivity and specificity. These have to do with the number of correct classifications ("predictions") made by the model on the data to which it was fit. Although probabilities predicted by logistic-regression models may fall anywhere between 0 and 1, they can be classified as 0 ("negative") or 1 ("positive") using the sample mean of the dependent variable as the cutoff. For example, for the AllFaults data set for CrosswordSage, 27.0% of Det values are 1, so model predictions < 0.270 are considered to be 0 and those > 0.270 are considered

[8]More precisely, deviance is a function of "the probability that the observed values of the dependent may be predicted by the independents" [25]. This probability is called the *likelihood*. The deviance of a model is actually $-2(L_M - L_S)$, where L_M is the log of the likelihood for the model and L_S is the log of the likelihood for a perfectly-fitting model.

Table 4.4: Data summary: AllFaults

	CrosswordSage			FreeMind 0.7.1		
	Min.	Mean	Max.	Min.	Mean	Max.
F.MutType	0	0.474	1	0	0.705	1
T.Len	6	13.0	20	3	11.6	20
T.Events	3.12	5.00	6.94	3.44	4.35	5.26
T.Class	0.556	0.697	0.750	0.749	0.795	0.815
T.Meth	0.328	0.454	0.490	0.521	0.569	0.595
T.Block	0.361	0.509	0.552	0.419	0.485	0.524
T.Line	0.345	0.484	0.525	0.423	0.488	0.525
T.Pairs	0.192	0.272	0.338	0.0141	0.0230	0.0286
T.Triples	0.0198	0.0337	0.0499	0.0000554	0.0001540	0.0002132

to be 1. Sensitivity is the proportion of actual positives (e.g., Det = 1) correctly classified as such. Specificity is the proportion of actual negatives correctly classified as such.

4.3 Results

Tables 4.4, 4.5, and 4.6 summarize the three data sets to be analyzed, which were described in Table 4.3. These tables list the minimum, mean, and maximum of each independent variable in each data set. Understanding the range of the data will help to interpret the logistic-regression models presented below. Some key observations about the data sets can also be made.

One observation is that the values for test-suite metrics are nearly the same for the AllFaults and PoolCovFaults data sets. Recall that the AllFaults data set includes all ⟨*test suite, fault*⟩ pairs, while the PoolCovFaults data set includes only the ⟨*test suite, fault*⟩ pairs where the faulty line is covered by the test pool. It makes sense, then, for the two data sets to differ in their fault metrics but not in their test-suite metrics. Certainly the PoolCovFaults data set includes more

Table 4.5: Data summary: PoolCovFaults

	CrosswordSage			FreeMind 0.7.1		
	Min.	Mean	Max.	Min.	Mean	Max.
F.MutType	0	0.448	1	0	0.759	1
F.CovBef	0.0000	0.0736	0.2165	0.000	0.138	0.292
F.Depth	0.000	0.291	0.833	0.000	0.397	1.000
F.SomeRep	0	0.323	1	0	0.745	1
F.AllRep	0	0.304	1	0	0.176	1
F.MinPred	0.0000	0.0293	0.2041	0.00000	0.02450	0.77679
F.MaxPred	0.0000	0.1560	0.2041	0.00000	0.67410	0.77679
F.MinSucc	0.0102	0.0592	0.2041	0.00446	0.14032	0.75893
F.MaxSucc	0.0204	0.1660	0.2041	0.01786	0.66597	0.75893
F.Events	0.0102	0.0698	0.4388	0.00446	0.54611	0.96429
F.MinWith	0.000461	0.060423	0.171350	0.000107	0.041343	0.143253
F.MaxWith	0.00322	0.12423	0.17181	0.0576	0.1552	0.1768
T.Len	6	12.9	20	3	11.6	20
T.Events	3.12	5.02	6.89	3.44	4.34	5.22
T.Class	0.556	0.694	0.750	0.749	0.795	0.815
T.Meth	0.328	0.453	0.487	0.521	0.568	0.595
T.Block	0.361	0.508	0.552	0.420	0.484	0.524
T.Line	0.345	0.483	0.525	0.424	0.487	0.525
T.Pairs	0.192	0.272	0.336	0.0141	0.0230	0.0286
T.Triples	0.0198	0.0337	0.0460	0.0000554	0.0001545	0.0002132

metrics, and for the one metric the two data sets have in common—F.MutType—there is a small but perhaps significant difference. (Statistical analysis will tell for sure whether observed differences are indeed significant.)

A second observation is that there *are* noticeable differences in the test-suite metrics of the SuiteCovFaults data set and the other two data sets. (The differences are small but generally greater than those between the other two data sets.) This, too, is as expected. Recall that the SuiteCovFaults data set includes only the ⟨*test suite, fault*⟩ pairs where the test suite covers the faulty line. Thus, the test suites in SuiteCovFaults are generally better at covering the application code.

Another observation is that among all three data sets some of the fault metrics

Table 4.6: Data summary: SuiteCovFaults

	CrosswordSage			FreeMind 0.7.1		
	Min.	Mean	Max.	Min.	Mean	Max.
F.MutType	0	0.429	1	0	0.711	1
F.CovBef	0.0000	0.0716	0.2165	0.000	0.116	0.238
F.Depth	0.000	0.283	0.833	0.000	0.330	0.667
F.SomeRep	0	0.319	1	0	0.731	1
F.AllRep	0	0.306	1	0	0.166	1
F.MinPred	0.0000	0.0277	0.2041	0.00000	0.00931	0.02679
F.MaxPred	0.0000	0.1631	0.2041	0.00000	0.65850	0.77679
F.MinSucc	0.0102	0.0589	0.2041	0.00446	0.12396	0.75893
F.MaxSucc	0.0204	0.1718	0.2041	0.01786	0.65450	0.75890
F.Events	0.0102	0.0729	0.4388	0.00446	0.59915	0.96429
F.MinWith	0.000461	0.057481	0.171350	0.000107	0.038661	0.137178
F.MaxWith	0.00322	0.12132	0.17181	0.0666	0.1596	0.1768
T.Len	6	12.9	20	3	11.5	20
T.Events	3.12	5.02	6.89	3.44	4.34	5.22
T.Class	0.556	0.703	0.750	0.763	0.796	0.815
T.Meth	0.328	0.457	0.487	0.521	0.569	0.595
T.Block	0.361	0.514	0.552	0.420	0.486	0.524
T.Line	0.345	0.489	0.525	0.424	0.489	0.525
T.Pairs	0.192	0.272	0.336	0.0141	0.0230	0.0286
T.Triples	0.0198	0.0337	0.0460	0.0000553	0.0001539	0.0002132

differ notably. Since additional coverage constraints are placed on the faults from the AllFaults data set to the PoolCovFaults data set (the test pool must cover them) and from the PoolCovFaults data set to the SuiteCovFaults data set (the test suite paired with them must cover them), the three data sets represent progressively more "coverable" sets of faults. It is not surprising that their metrics differ.

It is also worth noting that the levels of coverage in this experiment are rather low, by research standards if not by industrial standards. Even though each test suite executes each event in the EFG an average of 3 to 7 times (T.Events), class coverage (T.Class) falls at less than 82% and line coverage (T.Line) at less than 53%.

Table 4.7: Data summary: Cov and Det

	CrosswordSage						
	Cov			Det			Total
	0	1	Mean	0	1	Mean	
AllFaults	1147	1083	0.486	1629	601	0.270	2230
PoolCovFaults	101	1083	0.915	583	601	0.508	1184
SuiteCovFaults	0	1083	1.000	482	601	0.555	1083
	FreeMind 0.7.1						
	Cov			Det			Total
	0	1	Mean	0	1	Mean	
AllFaults	506	464	0.478	800	170	0.175	970
PoolCovFaults	137	464	0.772	431	170	0.283	601
SuiteCovFaults	0	464	1.000	294	170	0.366	464

With a few tweaks in test-case generation, coverage could have been improved (e.g., by providing more input files or forcing more test cases to open input files), but only at the expense of the randomness and replicability of the test suites. Since fault detection seems to increase super-linearly with coverage [31], different results might be obtained with greater coverage levels.

Table 4.7 summarizes the dependent variables, Cov and Det, for each data set, giving the frequency of each value (0 or 1), the mean (proportion of 1 values), and the total number of data points. With each successive data set, from AllFaults to PoolCovFaults to SuiteCovFaults, more data points with $\mathsf{Cov} = 0$ (and therefore $\mathsf{Det} = 0$), are eliminated, so the proportion of Cov and Det grows.

In the rest of this section, the three data sets are used to test the hypotheses listed at the beginning of this chapter by fitting logistic-regression models to them. For hypotheses H1 and H2, the question is whether fault and test-suite characteristics, or interactions between them, affect the likelihood that a test suite detects a fault ($\mathsf{Det} = 1$). For hypotheses H3 and H4, the question is whether these same variables affect the likelihood that a test suite detects a fault ($\mathsf{Det} = 1$), given that the test suite covers the faulty line ($\mathsf{Cov} = 1$). Both of these questions are addressed, in Sections 4.3.2 and 4.3.3, respectively. But first, as a stepping stone toward answering the other questions, Section 4.3.1 explores whether fault and test-suite characteristics affect the likelihood that a test suite covers a faulty line (Cov). In addition, that section explains how the logistic-regression models should be interpreted.

4.3.1 What characteristics affect whether a test suite covers a faulty line?

For this work, understanding the fault and test-suite characteristics that affect coverage of faulty lines is not an end in itself. Rather, it assists in understanding the characteristics that affect detection of faults.

In fact, a study whose end goal was to understand characteristics affecting coverage of lines would probably want to study each line of code in the applications under test. This experiment, in contrast, studies only the sample of lines where faults happen to be seeded, and the sample is biased toward lines with more opportunities for seeding. Therefore, while the results in this section provide information about coverage that proves helpful in later sections, the information should be used with caution outside the experiment.

For some metrics, it is obvious even without empirical evidence that they affect coverage of faulty lines. The probability of Cov is bound to increase, on average, as T.Class, T.Meth, T.Block, and T.Line increase, and it cannot decrease as

Table 4.8: Univariate models: AllFaults, Cov

	CrosswordSage			FreeMind 0.7.1		
	Int.	Coef.	Sig.	Int.	Coef.	Sig.
F.MutType	0.108	-0.349	***	-0.126	0.0558	
T.Len	-0.0575	-0.0131		-0.0867	-0.00588	
T.Events	-0.0574	0.0901		-0.0867	-0.107	
T.Class	-0.0581	2.16	***	-0.0874	12.3	*
T.Meth	-0.0582	4.28	***	-0.0868	4.85	
T.Block	-0.0582	3.28	***	-0.0869	4.27	
T.Line	-0.0582	3.54	***	-0.0869	4.28	
T.Pairs	-0.0574	1.15		-0.0867	0.265	
T.Triples	-0.0574	2.45		-0.0867	-76.4	

T.Events, T.Pairs, and T.Triples increase. Certainly F.CovBef, F.Depth, F.MinPred, and F.MaxPred affect coverage as well: all faults with a value of 0 for these variables are guaranteed to be covered, since they lie in initialization code that is executed by every test case. For other metrics, like those measuring the size of event handlers (F.MinWith and F.MaxWith), their relationship to coverage can only be discovered by analyzing the data.

Two of the data sets—AllFaults and PoolCovFaults—can be used to show which characteristics affect whether a test suite covers a faulty line. AllFaults includes all of the ⟨*test suite, fault*⟩ pairs but only one fault metric, while PoolCovFaults includes all fault metrics but a subset of ⟨*test suite, fault*⟩ pairs that is biased toward easier-to-cover faulty lines. Whether the independent variables come from AllFaults or PoolCovFaults, the dependent variable of interest is Cov.

Table 4.9: Univariate models: PoolCovFaults, Cov

	CrosswordSage			FreeMind 0.7.1		
	Int.	Coef.	Sig.	Int.	Coef.	Sig.
F.MutType	2.84	-0.875	***	2.50	-1.54	***
F.CovBef	2.48	-14.1	***	2.93	-43.3	***
F.Depth	2.53	-4.34	***	1.91	-6.70	***
F.SomeRep	2.46	-0.255		1.50	-0.361	
F.AllRep	2.35	0.0888		1.28	-0.30	
F.MinPred	2.41	-6.34	***	-1.11	-197	***
F.MaxPred	2.81	13.1	***	1.25	-1.25	**
F.MinSucc	2.37	-1.09		1.23	-0.874	**
F.MaxSucc	2.95	21.7	***	1.24	-1.09	*
F.Events	2.90	27.8	***	1.34	1.99	***
F.MinWith	2.62	-18.4	***	1.24	-6.79	**
F.MaxWith	2.62	-14.7	***	1.33	24.3	***
T.Len	2.37	-0.0164		1.22	-0.0205	
T.Events	2.37	-0.0314		1.22	-0.00992	
T.Class	3.37	19.6	***	1.25	28.7	***
T.Meth	3.28	36.5	***	1.23	14.8	*
T.Block	3.31	28.1	***	1.24	12.5	**
T.Line	3.30	30.2	***	1.24	12.7	**
T.Pairs	2.37	0.86		1.22	-22.3	
T.Triples	2.37	-3.83		1.22	-2150	

4.3.1.1 Univariate models

Univariate logistic-regression models were fit to each of these two data sets. Recall from Section 4.2.2 that a univariate model shows how much an independent variable (fault or test-suite metric) affects the likelihood of the dependent variable (Cov), without controlling for any other independent variables. Tables 4.8 and 4.9 summarize the univariate models, giving intercepts, coefficients, and significance levels. For example, the first row of Table 4.8 for CrosswordSage represents the model

$$\mathrm{logit}(\Pr(\mathsf{Cov})) = 0.108 - 0.349\mathsf{F.MutType}.$$

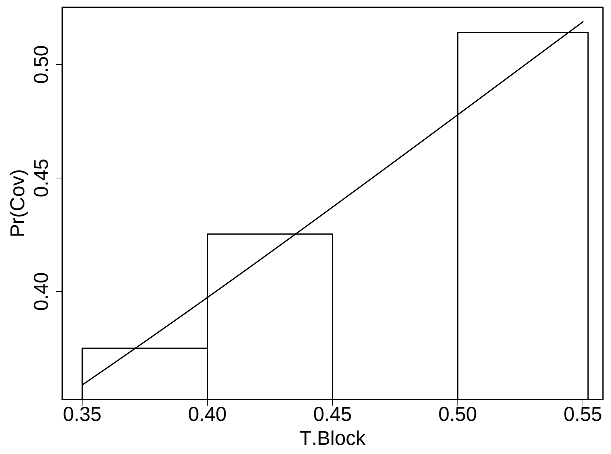


Figure 4.2: Predicted probabilities for CrosswordSage T.Block model in Table 4.8

Metrics with significance levels of "∘", "*", "**", and "***" have p-values less than or equal to 0.1, 0.05, 0.01, and 0.001, respectively. The smaller the p-value, the more likely it is that the theoretical (true) coefficient value is non-zero and has the same sign as the estimated coefficient.

This section first explains how to interpret the models, then describes the main results learned from these models. To understand how to interpret the coefficients in these models, consider T.Block for CrosswordSage in Table 4.8. The coefficient, 3.28, means that the odds of Cov are expected to increase by a factor of $\exp(3.28)^{\Delta}$ when T.Block increases by Δ (Equation 4.3). For example, if T.Block increases from 0.4 to 0.5, the odds of covering a given faulty line are expected to increase by a factor of $\exp(3.28)^{0.5-0.4} \approx 1.39$. Figure 4.2 shows how this translates to probabilities. The plotted curve represents the predicted probability of Cov across the range of T.Block (similar to Figure 4.1). (The near-linear shape of the curve demonstrates the ability of logistic-regression functions to mimic other functions, such as linear functions.) For comparison, the curve is superimposed on a bar plot of the sample probabilities of Cov at different levels of T.Block in the AllFaults data set. (No T.Block values fall between 0.45 and 0.5, so no bar is drawn.) The predicted probability for a particular value of T.Block is found by subtracting the mean of T.Block for the data set (0.509)—since the models are fit to centered data—to get X, and then plugging X, the intercept (-0.0582), and the coefficient (3.28) into Equation 4.2. For instance, for a T.Block value of 0.540, X is $0.540 - 0.509 = 0.031$, and the predicted probability of Cov is $\exp(-0.0582 + 3.28 * 0.031)/(1 + \exp(-0.0582 + 3.28 * 0.031)) = 0.511$.

For certain metrics—F.MutType, F.SomeRep, and F.AllRep—X can only be 0 or 1. In Table 4.9 for FreeMind, for example, the predicted probability of Cov for method-level mutation faults (encoded as 1) is $\exp(2.50 + -1.54 * 1)/(1 + \exp(2.50 + -1.54 * 1)) = 0.724$, whereas for class-level mutation faults (encoded as 0) it is $\exp(2.50 + -1.54 * 0)/(1 + \exp(2.50 + -1.54 * 0)) = 0.924$.

While it is important to understand how to interpret the magnitude of model coefficients—or at least to understand that they do not denote a linear relationship with the probability of the dependent variable—the analysis below focuses on the significance levels, the signs, and occasionally the relative magnitudes of coefficients.

Now let us return to the question at hand: what characteristics affect the likelihood that a test suite covers a faulty line? In Tables 4.8 and 4.9, most of the fault characteristics have statistically significant effects: the method of creation (F.MutType, except for FreeMind in Table 4.8), the distance from the initial state (F.CovBef and F.Depth), the degrees of freedom in execution (F.MinPred, F.MaxPred, F.MinSucc for FreeMind, F.MaxSucc, and F.Events), and the size of enclosing event handlers (F.MinWith and F.MaxWith). For the characteristic of distance from the initial state, the results are as expected and are consistent across applications: the metrics F.CovBef and F.Depth have negative coefficients, indicating that faulty lines lying farther from the initial state are less likely to be covered. For most of the other characteristics, the direction of their effect on coverage is inconsistent: the signs of coefficients differ across metrics or across applications. But for F.MutType, the results are fairly consistent and surprising: more class-level mutation faults than method-level were seeded in code covered by the test pool. This portends (somewhat incorrectly, it turns out) that the logistic-regression models of Det later in this section will show class-level mutation faults to be more susceptible to detection.

One fault characteristic does not affect Cov: the repetitions in which the faulty code is executed (F.SomeRep and F.AllRep). This makes sense because a faulty line only needs to be executed once to be covered.

As for test-suite characteristics, Tables 4.8 and 4.9 show that the code-coverage metrics (T.Class, T.Meth, T.Block, and T.Line) of course affect Cov. The event-coverage metrics (T.Pairs and T.Triples), the length of test cases (T.Len), and the size of the test suite (T.Events) do not have a significant effect on Cov for the ranges of these metrics studied.

4.3.1.2 Multivariate models

Multivariate models were also fit to the AllFaults and PoolCovFaults data sets, again with Cov as the dependent variable, in order to assess each independent

Table 4.10: Multivariate models: AllFaults, Cov

	CrosswordSage		FreeMind 0.7.1	
	Coef.	Sig.	Coef.	Sig.
Intercept	0.105		-0.0874	
F.MutType	-0.346	***		
T.Class			12.3	*
T.Block	3.25	***		
Null deviance	3089.6		1342.9	
Deviance	3056.8		1338.6	
AIC	3062.8		1342.6	
Sensitivity	444/1083 = 0.410		395/464 = 0.851	
Specificity	785/1147 = 0.684		99/506 = 0.196	

variable's contribution to explaining Cov in the context of the other independent variables. Recall from Section 4.2.2.2 that each multivariate model uses the subset of all metrics that provides the best balance of model fit and parsimony (i.e., the lowest AIC) without severe symptoms of multicolinearity. (For each "best" multivariate model, often several similar models are nearly as good; for example, for a model that includes T.Line, models with T.Block, T.Meth, or T.Class in its place often have similar AIC values.) A test-suite or fault metric's coefficient in a multivariate model shows how much it affects Cov, relative to the other metrics in the model, when the other metrics are held constant.

Tables 4.10 and 4.11 summarize the multivariate models for the two data sets. For example, the model for CrosswordSage in Table 4.10 is

$$\mathrm{logit}(\Pr(\mathsf{Cov})) = 0.105 - 0.346\mathsf{F.MutType} + 3.25\mathsf{T.Block}.$$

The lower portion of each table lists several measures of goodness of fit (Section 4.2.2.3).

Let us consider how to interpret the models. As with the univariate models,

Table 4.11: Multivariate models: PoolCovFaults, Cov

	CrosswordSage		FreeMind 0.7.1	
	Coef.	Sig.	Coef.	Sig.
Intercept	10.6		3.19	
F.MutType			-1.16	***
F.CovBef	-142	***	-50.9	***
F.MinPred			-106	***
F.MaxPred	67.2	***		
F.MinWith			-13.7	*
F.MaxWith	-113	***		
T.Class			110	***
T.Block	-39.6	***		
T.Class $\times$ F.CovBef			-1150	***
T.Block $\times$ F.CovBef	2350			
T.Block $\times$ F.MaxPred	-1940	***		
T.Block $\times$ F.MaxWith	3260	***		

	CrosswordSage	FreeMind 0.7.1
Null deviance	690.36	645.22
Deviance	58.81	308.82
AIC	74.81	322.82
Sensitivity	1059/1083 = 0.978	376/464 = 0.810
Specificity	100/101 = 0.990	122/137 = 0.891

predicted probabilities of the dependent variable, Cov, can be calculated by plugging values for the independent variables into Equation 4.2. Coefficients should be interpreted relative to other coefficients. For example, in the model for CrosswordSage in Table 4.11, the coefficient of F.CovBef (-90.2) is about twice that of F.MaxWith (-45.0). Thus, the model estimates that a change of Δ in F.CovBef would increase or decrease the odds of Cov by nearly the same factor that a change of 2Δ in F.MaxWith would. For instance, when F.CovBef increases from 0.05 to 0.10 and all other metrics are held constant, the odds of Cov are predicted to decrease by a factor of $\exp(-90.2)^{0.05} = 0.011$. When F.MaxWith increases from 0.05 to 0.15 and all other metrics are held constant, the odds of Cov are expected to decrease by the same factor, $\exp(-45.0)^{0.10} = 0.011$.

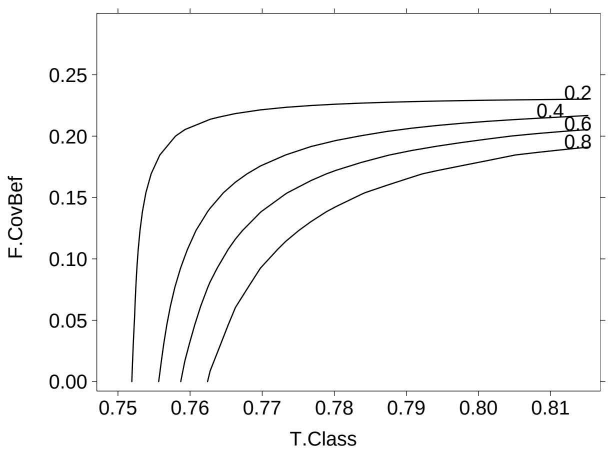


Figure 4.3: Predicted probabilities for FreeMind model in Table 4.11

Interaction effects in a model can be understood by looking at the signs of their coefficients or by calculating predicted probabilities. Consider the interaction between T.Class and F.CovBef (T.Class × F.CovBef) for FreeMind in Table 4.11. The positive coefficient of T.Class and the negative coefficient of F.CovBef imply that the probability of Cov generally increases as T.Class increases or F.CovBef decreases. But when these metrics have opposite signs in the centered data (i.e., one is above its mean and the other is below its mean in the original data), their product is negative, and this multiplied by the negative coefficient of T.Class × F.CovBef is positive—boosting the probability of Cov. Conversely, if T.Class and F.CovBef have the same sign in the centered data, the interaction effect diminishes the probability of Cov. To illustrate, the contour plot in Figure 4.3 shows the predicted probabilities at various levels of T.Class and F.CovBef when F.MutType is held constant at 0 and F.MinPred and F.MinWith are held constant at their respective means. The basic idea is that test suites with high class coverage (T.Class) are especially good at covering lines at a small distance from the initial state (F.CovBef), and test suites with low class coverage are especially bad at covering lines at a great distance from the initial state.

Now, in the multivariate models, which fault and test-suite characteristics affect whether a test suite covers a faulty line? In the model for FreeMind in Table 4.10, the only characteristic included is the proportion of coverage (T.Class); it was the only characteristic significant in the univariate models for that data set. But in the other models in Tables 4.10 and 4.11, it is a combination of faulty-line and test-suite characteristics that best explains Cov. For both applications in Table 4.11, the distance of the faulty line from the initial state (F.CovBef), the degrees of freedom in the faulty line's execution (F.MinPred or F.MaxPred), the size of the event handlers containing the fault line (F.MinWith or F.MaxWith), and the proportion of code coverage (T.Class or T.Block) contribute to explaining Cov. For FreeMind, the type of mutation (F.MutType) also contributes. For CrosswordSage in Table 4.10, the characteristics chosen for the model are the type of mutation (F.MutType) and the proportion of coverage (T.Block). In all of the models in Tables 4.10 and 4.11—with the exception that F.MutType is left out of the model for CrosswordSage in Table 4.11—the set of characteristics included in the models is the same as the set that were significant in the univariate models. Additionally, the multivariate models in Table 4.11 include interaction effects between test-suite coverage and some fault characteristics.

The models for the PoolCovFaults data set (Table 4.11) fit the data well. For CrosswordSage, the fit is nearly perfect, with sensitivity and specificity at 97.8% and 99.0%. For FreeMind, the fit is not quite as good, with sensitivity and specificity

Table 4.12: Univariate models: AllFaults, Det

	CrosswordSage			FreeMind 0.7.1		
	Int.	Coef.	Sig.	Int.	Coef.	Sig.
F.MutType	-1.10	0.211	*	-1.16	-0.577	**
T.Len	-0.997	-0.00597		-1.55	0.0120	
T.Events	-1.00	0.230	**	-1.55	0.362	
T.Class	-0.999	1.07	∘	-1.55	-4.10	
T.Meth	-0.999	2.41	*	-1.55	-0.128	
T.Block	-0.999	1.85	*	-1.55	0.177	
T.Line	-0.999	2.01	*	-1.55	0.154	
T.Pairs	-1.00	5.78	**	-1.55	39.9	
T.Triples	-0.999	23.2	**	-1.55	2520	

at 81.0% and 89.1%. The models for the PoolCovFaults data set, with its additional fault metrics, fit the data much better than those for the AllFaults data set (Table 4.10).

4.3.2 What characteristics affect whether a test suite detects a fault?

Considering the widespread use of code coverage as a predictor of fault detection, one might assume that the same characteristics that affect whether a test suite covers faulty code would also affect whether a test suite detects a fault. To investigate whether this is actually the case, the same data sets (AllFaults and PoolCovFaults) are used as in the previous section, but now the dependent variable is Det instead of Cov.

4.3.2.1 Univariate models

Tables 4.12 and 4.13 summarize the univariate models of Det for the two data sets.

The set of fault characteristics significant in these models of Det is a superset of the fault characteristics significant in the corresponding models of Cov in Tables 4.8

Table 4.13: Univariate models: PoolCovFaults, Det

	CrosswordSage			FreeMind 0.7.1		
	Int.	Coef.	Sig.	Int.	Coef.	Sig.
F.MutType	-0.209	0.536	***	-0.124	-1.12	***
F.CovBef	0.0296	-6.56	***	-1.10	-12.1	***
F.Depth	0.0305	-1.05	**	-1.12	-3.79	***
F.SomeRep	-0.112	0.444	***	-0.606	-0.445	*
F.AllRep	-0.131	0.537	***	-1.03	0.532	*
F.MinPred	0.0293	-3.87	**	-3.07	-144	***
F.MaxPred	0.0292	3.76	***	-0.971	-2.00	***
F.MinSucc	0.0304	0.744		-0.935	-0.531	
F.MaxSucc	0.0202	10.3	***	-0.972	-2.25	***
F.Events	0.0409	7.89	***	-0.953	-0.984	***
F.MinWith	0.0293	-8.83	***	-0.965	10.9	***
F.MaxWith	0.0304	0.699		-0.930	-0.209	
T.Len	0.0304	0.00198		-0.931	0.012	
T.Events	0.0305	0.246	*	-0.936	0.504	◦
T.Class	0.0303	2.30	**	-0.931	-3.16	
T.Meth	0.0302	4.89	***	-0.930	1.43	
T.Block	0.0302	3.70	***	-0.930	1.41	
T.Line	0.0302	4.01	***	-0.930	1.45	
T.Pairs	0.0305	7.35	**	-0.934	41.1	
T.Triples	0.0305	31.0	*	-0.932	2450	

and 4.9; some characteristics affect fault detection without affecting faulty-line coverage. These are the mutation type (F.MutType) for FreeMind in the AllFaults data set and the repetitions in execution (F.SomeRep and F.AllRep) for both applications in the PoolCovFaults data set.

For CrosswordSage, some test-suite characteristics also affect fault detection without affecting faulty-line coverage. These are the size (T.Events) and the coverage as measured by event-based metrics (T.Pairs and T.Triples).

For FreeMind, the opposite happens for test-suite characteristics. The one test-suite characteristic that affects faulty-line coverage, the proportion of test-suite coverage (T.Class), does not affect fault detection.

To some extent, the *significant* coefficients in Tables 4.12 and 4.13 can be compared to those in Tables 4.8 and 4.9 to see whether a metric has a stronger effect on Det or Cov. This must be done cautiously: differences in the magnitude of coefficients are statistically meaningful only if both coefficients are statistically significant and the confidence intervals surrounding the two coefficients do not overlap. (This is implied if the coefficients have opposite signs.) The 95% Wald confidence interval for a coefficient β_i is $\beta_i \pm 1.96SE$, where SE is the standard error for β_i.

For the AllFaults data set for CrosswordSage, the apparent differences in coefficients for the coverage metrics (T.Class, T.Meth, T.Block, and T.Line) between the models of Cov and Det are *not* meaningful. Only the difference for F.MutType is; method-level mutation faults are less likely than class-level ones to be covered but more likely to be detected. For the PoolCovFaults data set, most of the apparent differences *are* statistically meaningful, with the exception of F.MinPred for CrosswordSage and F.MutType, F.MinPred, and F.MaxPred for FreeMind. Thus, for example, F.CovBef really does have a stronger effect on Cov than on Det for both applications. For FreeMind, two metrics—F.Events and F.MinWith—have opposite effects on Cov and Det, but other measures of the same characteristics do not.

4.3.2.2 Multivariate models

Tables 4.14 and 4.15 summarize the multivariate models of Det for the two data sets. Not every characteristic that was significant in the univariate models of Det (Tables 4.12 and 4.13) is included in these models, although most are. In the model for

Table 4.14: Multivariate models: AllFaults, Det

	CrosswordSage		FreeMind 0.7.1	
	Coef.	Sig.	Coef.	Sig.
Intercept	-1.11		-1.16	
F.MutType	0.213	*	-0.577	**
T.Line	0.363	*		
T.Pairs	5.82	**		
T.Line × F.MutType	3.22			
Null deviance	2599.1		900.40	
Deviance	2580.2		889.91	
AIC	2590.2		893.91	
Sensitivity	296/601 = 0.493		68/170 = 0.400	
Specificity	974/1629 = 0.598		582/800 = 0.728	

FreeMind in Table 4.13, the repetitions in which a faulty line is executed (F.SomeRep and F.AllRep) and the size of event handlers containing the faulty line (F.MinWith) are excluded. In the models for CrosswordSage, test-suite size (T.Events) is excluded; it does not contribute to explaining fault detection beyond what is already explained by the test suite's coverage level. (In the model for FreeMind in Table 4.13, test-suite size is included, but perhaps only because no coverage metrics had significant effects on fault detection.) For CrosswordSage, several interaction effects between test-suite coverage (T.Line and T.Pairs) and various fault metrics are also included.

The multivariate models of Det for the AllFaults data set (Table 4.14) fit the data about as well as the corresponding models of Cov (Table 4.10), when goodness of fit is measured as the ratio of deviance to null deviance. But the multivariate models of Det for the PoolCovFaults data set (Table 4.15) fit decidedly worse than the corresponding models of Cov (Table 4.11). Thus, while the set of characteristics in PoolCovFaults is nearly sufficient to explain coverage of faulty lines,

Table 4.15: Multivariate models: PoolCovFaults, Det

	CrosswordSage		FreeMind 0.7.1	
	Coef.	Sig.	Coef.	Sig.
Intercept	-0.561		-1.24	
F.MutType	0.910	***	-0.711	***
F.CovBef	-15.4	***	-7.68	***
F.AllRep	0.387	***		
F.MinPred			-49.9	***
F.MaxSucc	11.7	***		
F.MinWith	-8.94	***		
T.Events			0.506	◦
T.Line	5.17	***		
T.Pairs	9.02	**		
T.Line × F.MutType	4.93	**		
T.Line × F.CovBef	95.2	*		
T.Line × F.MaxSucc	-109	***		
T.Line × F.MinWith	150	***		
T.Pairs × F.CovBef	235	**		
Null deviance	1641.1		715.95	
Deviance	1398.1		588.07	
AIC	1424.1		598.07	
Sensitivity	428/601 = 0.712		102/170 = 0.600	
Specificity	399/583 = 0.684		352/431 = 0.817	

it is not sufficient to explain detection of faults.

4.3.3 What characteristics affect whether a test suite detects a fault, given that the test suite covers the faulty line?

The previous section asked a similar question: What characteristics affect whether a test suite detects a fault? Some of those characteristics might affect the probability that a fault is detected *primarily because* they affect the probability that the faulty code is covered. For example, some faults may be harder to detect simply because they lie in a part of the program that is harder to cover. But other faults may be harder to detect even if code containing them is covered, which is why it is

Table 4.16: Univariate models: SuiteCovFaults, Det

	CrosswordSage			FreeMind 0.7.1		
	Int.	Coef.	Sig.	Int.	Coef.	Sig.
F.MutType	-0.104	0.778	***	0.0299	-0.834	***
F.CovBef	0.221	-3.94	*	-0.596	-9.03	***
F.Depth	0.221	-0.251		-0.607	-2.74	***
F.SomeRep	0.0488	0.554	***	-0.274	-0.380	○
F.AllRep	0.0479	0.582	***	-0.682	0.760	**
F.MinPred	0.221	-2.70	○	-0.602	-116	***
F.MaxPred	0.221	1.25		-0.563	-1.90	***
F.MinSucc	0.221	1.08		-0.548	-0.256	
F.MaxSucc	0.223	6.72	***	-0.564	-2.30	***
F.Events	0.229	5.54	***	-0.586	-2.05	***
F.MinWith	0.223	-6.16	***	-0.568	15.1	***
F.MaxWith	0.222	2.69	**	-0.555	-10.3	**
T.Len	0.221	0.00537		-0.549	0.021	
T.Events	0.222	0.273	*	-0.553	0.583	*
T.Class	0.221	-0.814		-0.551	-17.4	○
T.Meth	0.221	-1.12		-0.548	-4.29	
T.Block	0.221	-0.846		-0.548	-3.40	
T.Line	0.221	-0.891		-0.548	-3.44	
T.Pairs	0.222	7.92	**	-0.552	54.3	○
T.Triples	0.222	34.8	*	-0.55	3540	

interesting to ask the question for this section.

To answer this question, a different data set than in previous sections is needed: SuiteCovFaults. Unlike the AllFaults and PoolCovFaults data sets, this data set includes only the ⟨*test suite, fault*⟩ pairs where the test suite covers the faulty line (Cov = 1). This data set is ideal for understanding the conditional probability that a test suite detects a fault given that it covers the faulty line. The dependent variable of interest is, of course, Det.

4.3.3.1 Univariate models

Table 4.16 summarizes the univariate models of Det given Cov. As in the models of Det for the PoolCovFaults data set (Table 4.13), all fault characteristics in these models have a significant effect on fault detection. Thus, not only do all of these fault characteristics affect faults' likelihood of detection, but they affect faults' likelihood of detection even controlling for whether the faulty code is covered.

The test-suite metrics that turn out to be statistically significant ($p \leq 0.05$) in these models are exactly those that are statistically interesting ($p \leq 0.10$) in the models of Det for the PoolCovFaults data set (Table 4.13) but not in the corresponding models of Cov (Table 4.9). This reinforces the idea that these metrics—T.Events (both applications) and T.Pairs and T.Triples (CrosswordSage)—affect fault detection without affecting code coverage very much.

As in Section 4.3.2, the confidence intervals surrounding coefficients in Table 4.16 (SuiteCovFaults data set) can be compared to those in Table 4.13 (PoolCovFaults data set) to learn whether the coefficients differ meaningfully in magnitude—that is, whether the independent variables have significantly stronger effects on Det or on Det given Cov. The only metrics for which the difference is meaningful are F.MaxSucc for CrosswordSage and F.CovBef and F.Depth for FreeMind (which have a stronger effect on Det) and F.Events for FreeMind (which has a stronger effect on Det given Cov).

Table 4.17: Multivariate models: SuiteCovFaults, Det

	CrosswordSage		FreeMind 0.7.1	
	Coef.	Sig.	Coef.	Sig.
Intercept	-0.310		-0.395	
F.MutType	0.940	***	-0.472	***
F.CovBef	-11.062	***		
F.Depth			-1.552	
F.AllRep	0.503	**	0.669	***
F.MinPred			-53.810	***
F.MaxSucc	5.328	***		
F.MinWith	-7.275	***		
F.MaxWith			-15.226	***
T.Class			-20.190	◦
T.Pairs	8.806	**	90.017	*
T.Pairs × F.CovBef	173.314	◦		
Null deviance	1488.3		609.7	
Deviance	1365.9		526.4	
AIC	1381.9		542.4	
Sensitivity	393/601 = 0.654		105/170 = 0.618	
Specificity	312/482 = 0.647		222/294 = 0.755	

4.3.3.2 Multivariate models

Table 4.17 summarizes the multivariate models. Every fault characteristic takes part in these models. Every test-suite characteristic that was significant in the univariate models (Table 4.16) also takes part, with the exception of test-suite size (T.Events). (Once again, a test suite's size does not explain much about its fault-detecting abilities that is not also explained by its coverage level.) For CrosswordSage, the model also includes an interaction effect between a test suite's event-pair coverage and a fault's distance from the initial state (T.Pairs × F.CovBef).

These multivariate models of Det given Cov fit the data slightly worse than the corresponding models of Det (Table 4.15), when fit is again measured by the ratio of deviance to null deviance. This is consistent with the earlier observation that Cov is better predicted than Det, since Det in the models of Table 4.15 is partially determined by Cov.

4.4 Discussion

Sections 4.4.1, 4.4.2, and 4.4.3 sum up the effects of faults characteristics, test-suite characteristics, and interactions between them on coverage and fault detection. In Section 4.4.4, implications of the fit of the logistic-regression models are considered. Section 4.4.5 discusses differences in the results for CrosswordSage and FreeMind. Finally, Section 4.4.7 considers threats to validity.

4.4.1 Fault characteristics

Every fault characteristic, and every fault metric but F.MinSucc, turned out to significantly affect the likelihood of fault detection in at least one logistic-regression model. Furthermore, there were some cases where the likelihood of fault detection depended *only* on fault characteristics, not on test-suite characteristics (i.e., the models for FreeMind in Tables 4.12 and 4.14). These facts suggest that the kinds of faults used to evaluate the effectiveness of testing techniques can significantly impact the percentage of faults they detect.

However, the experiment results do not tell conclusively how to define "kinds of faults". For many characteristics, the effect on fault detection was inconsistent—for the two different applications, for different data sets, or for different metrics measuring the characteristic. This section, as well as the subsequent section on test-suite characteristics, aims to synthesize the often inconsistent results across applications and data sets. It identifies the characteristics and metrics that are relatively consistent predictors of fault detection for the limited set of situations studied in this experiment. But the more important result at this early stage of research is not which metrics best predict fault detection but whether each metric *can* predict fault detection. That fault metrics can predict fault detection, and that they do so often in this experiment, tells us that different "kinds of faults" must exist even though it is not yet clear how to define them.

The characteristics provide a fairly orthogonal classification of faults for studies of fault detection, as most or all fault characteristics were represented in each multivariate model of fault detection. The rest of this section considers each characteristic in turn.

4.4.1.1 Method of creation

F.MutType had a different effect on fault detection for the two applications. For CrosswordSage, method-level mutation faults were easier to detect, while for FreeMind, class-level mutation faults were easier. Strangely, for both applications class-level mutation faults were easier to cover. Actually, the mutation type was not expected to be significant for either Cov or Det; the hypothesis from preliminary work [58, 60] was only that class-level mutation faults in class-variable declarations/initializations would be easier to detect, and such faults were omitted from this experiment. That F.MutType makes a difference in Cov and Det seems to be a quirk of the way CrosswordSage and FreeMind are structured—in providing opportunities to seed mutation faults—not an inherent difference between class- and method-level mutation faults. But it is something for experimenters to be aware of.

4.4.1.2 Distance from initial state

F.CovBef and F.Depth behaved just as expected: faults lying “closer” to the initial state were easier to cover and detect. Interestingly, these faults were easier to detect even given that they had been covered.

4.4.1.3 Frequency of execution

F.AllRep and F.SomeRep significantly affected fault detection, although the direction of that effect was mixed. Faulty lines that, when executed by an event, were only sometimes executed more than once (F.SomeRep $= 1$) were easier to detect for CrosswordSage but harder for FreeMind. Faulty lines that, when executed, were always executed more than once (F.AllRep $= 1$) were easier to detect for both applications. Since F.AllRep was chosen over F.SomeRep for the multivariate models, its effect on fault detection was not only more consistent but also more important.

4.4.1.4 Degrees of freedom

It was not clear at the outset which measures of degrees of freedom in execution of a faulty line—F.MinPred, F.MaxPred, F.MinSucc, F.MaxSucc, or F.Events—would be most closely associated with fault detection. Nor was it clear whether faults in code with more degrees of freedom—code that could be executed in many different contexts—would be easier or harder to cover and detect. For FreeMind, the minimum number of EFG predecessors of a faulty event (F.MinPred) turned out to be the most influential of these metrics, having been selected for each multivariate model of Det and of Det given Cov (Tables 4.15 and 4.17). Faulty code with fewer EFG predecessors—fewer degrees of freedom—was consistently more likely to be covered and detected for both applications. For CrosswordSage, F.MaxSucc was the metric chosen to represent the degrees-of-freedom characteristic in the multivariate models of Det and of Det given Cov (Tables 4.15 and 4.17). In contrast to FreeMind, faulty code with more EFG successors—more degrees of freedom—was consistently more likely to be detected.

4.4.1.5 Size of event handlers

The size of event handlers executing a faulty line could either be measured by the minimum or the maximum number of lines executed in the same event (F.MinWith or F.MaxWith). For CrosswordSage, F.MinWith was the better predictor of fault detection, with faults in larger event handlers being harder to detect. For FreeMind, the results are inconsistent. In the univariate models of the SuiteCovFaults data set, for example, event handlers with a larger F.MinWith are associated with greater fault detection, but event handlers with a larger F.MaxWith are associated with less fault detection.

4.4.2 Test-suite characteristics

4.4.2.1 Proportion of coverage

The code-coverage metrics—T.Class, T.Meth, T.Block, and T.Line—were, of course, associated with the probability of covering a given faulty line. Interestingly, greater coverage did not necessarily increase the likelihood of detecting a given fault—at least for the range of coverage levels considered in this experiment. For FreeMind, in fact, code coverage seemed to add no statistically significant value to testing. Had a broader range of code coverage levels been studied, however, code coverage would almost certainly have been shown to increase the likelihood of fault detection significantly.

The event-coverage metrics—T.Pairs and T.Triples—behaved in nearly the opposite fashion. While they did not increase the likelihood of covering a given faulty line, they did (for CrosswordSage and sometimes for FreeMind) significantly increase the likelihood of detecting a given fault.

4.4.2.2 Size

As expected, test suites with a greater size (T.Events) were more likely to detect a given fault. (They were not any more likely to cover a given faulty line.) An important question in studies of test-suite characteristics is whether code coverage still affects fault detection when test-suite size is controlled for. In this study, the only case where test-suite size influenced fault detection more than coverage (i.e., when T.Events was chosen for a multivariate model) was in the model for FreeMind in Table 4.15, in which no coverage metrics were significant at all.

4.4.2.3 Granularity

Contrary to previous experiments [53, 65], granularity (T.Len) had no effect whatsoever on coverage or detection. In this experiment, the minimum test-case length was equal to the depth of the EFG. In the previous experiment on GUI testing [65], shorter test cases were allowed. Thus, test-case length appears to only affect fault detection to the extent that higher-granularity test suites reach deeper into the EFG; once the granularity exceeds the EFG depth, no further benefit is gained.

4.4.3 Interactions between test-suite and fault characteristics

Interaction effects between test-suite and fault characteristics indicate cases where certain kinds of test suites are better at detecting certain kinds of faults. In practice, this information may be used to design test suites that target kinds of faults that tend to be more common or more severe. Interaction effects can also influence the results of evaluations of testing techniques because, if certain techniques are better at detecting certain kinds of faults, the sample of faults may be biased for or against a technique.

Only for CrosswordSage did interactions between test-suite and fault characteristics significantly affect fault detection. For the PoolCovFaults data set (Table 4.15), method-level mutation faults (F.MutType $= 1$), faults lying farther from the initial state (higher F.CovBef), and faults lying in larger event handlers (higher

F.MinWith) were better targeted by test suites with greater line coverage (T.Line). Both for that data set and for the SuiteCovFaults data set (Table 4.17), faults lying farther from the initial state (higher F.CovBef) were also better targeted by test suites with greater event-pair coverage (T.Pairs).

4.4.4 Model fit

The multivariate models of Cov for the PoolCovFaults data set (Table 4.11) fit the data well, with sensitivity and specificity between 81% and 99%. For CrosswordSage, the fit was nearly perfect. This suggests that the set of metrics studied is nearly sufficient to predict whether a test suite will cover a given line—a prerequisite to predicting whether the suite will detect a fault in the line. But, at least for applications like FreeMind, some influential test-suite or fault metrics remain to be identified.

The multivariate models of Det for the PoolCovFaults data set (Table 4.15) and of Det given Cov for the SuiteCovFaults data set (Table 4.17) did not fit as well, with sensitivity and specificity between 60% and 82%. This shows that predicting whether a test suite will detect a fault is harder than predicting whether the test suite covers the faulty code; more factors are at work.

Finally, the multivariate models of Cov and Det for the PoolCovFaults data set, which includes all fault characteristics, fit much better than those for the AllFaults data set, which includes just one fault characteristic (F.MutType). This provides additional evidence that at least some of the fault characteristics studied besides F.MutType are important predictors of fault coverage and detection.

For different applications and data sets, the fit of the multivariate models was, of course, different. When more fault metrics were available as independent variables—in the PoolCovFaults and SuiteCovFaults data sets, as opposed to the AllFaults data set—the models, not surprisingly, fit better. The models also fit better when the dependent variable was Cov than when the dependent was Det (a variable influenced by Cov), and better when the dependent was Det than when it was Det given Cov (a variable not influenced by Cov); fault coverage is evidently easier to predict than fault detection. As for the two applications, the models of Cov fit the data for CrosswordSage better, while the models of Det and of Det given Cov fit the data for FreeMind better (when fit is measured as the ratio of deviance to null deviance). In all cases but one, however, the difference is small (less than 0.06). The exception is the models of Cov for the PoolCovFaults data set (Table 4.11); the model for CrosswordSage fits markedly better. The reason must be that some additional, unknown test-suite or fault characteristics affect coverage for FreeMind, which is the larger and more complex of the two applications.

Even though the set of fault and test-suite characteristics studied did not result in perfectly-fitting models, the experiment results are still of value. While the characteristics studied do not comprise a complete list of factors that influence fault detection in software testing, they are certainly *some* of the factors that empirical studies of testing should account for.

4.4.5 Differences between applications

There were numerous minor differences between the results for CrosswordSage and FreeMind—cases where the significance, magnitude, or even the direction of a metric's influence on coverage or fault detection differed for the two applications. But there was only one metric, mutation type (F.MutType), that was influential enough to be included in the multivariate models yet influenced fault detection in opposite ways for the two applications. For CrosswordSage, method-level mutation faults were consistently *more* likely to be detected (even though they were less likely to be covered). For FreeMind, method-level mutation faults were *less* likely to be detected. The proportion of faults that were method-level mutations also differed widely between the applications: 47% for CrosswordSage and 70% for FreeMind.

These differences seem to result from the structure of the applications and the nature of the test oracle, with FreeMind having more opportunities to seed method-level mutation faults and with those opportunities lying in code that is less likely to affect the GUI state (as checked by GUITAR). For example, FreeMind contains a method called `ccw` that helps calculate the coordinates of an object in the GUI. Since the test oracle ignored the coordinates of GUI objects, faults in `ccw` were unlikely to be detected if they only changed the calculated coordinate values. And since `ccw` mainly operates on primitive types, nearly all of the mutation opportunities were for method-level faults. Thus, in the experiment, all 21 of the faults sampled from `ccw` were method-level mutations, and only 2 were detected by their corresponding test suite.

Another notable difference between the applications was in the total percentage of faults covered and detected in each data set (Table 4.7). Even though the percentage of all faults covered was nearly the same (49% for CrosswordSage, 48% for FreeMind), the percentage of faults detected was lower for FreeMind in every data set. This suggests at least one of the following explanations:

1. the sample of faults selected for FreeMind, out of the population of mutation faults for that application, happened to be especially hard to detect;
2. the structure of FreeMind makes it less testable, in the sense that randomly seeded faults are harder to detect; or
3. event-coverage-adequate GUI test suites are less effective for FreeMind than for CrosswordSage.

The experiment offers some evidence of the second explanation (e.g., the mean value of F.CovBef is greater for FreeMind), but the other explanations cannot be ruled out.

4.4.6 Assumption of independent sampling

A key assumption of logistic regression is that the error terms of different data points are independent. The error term is the distance between the actual value of the dependent variable for a data point (0 or 1) and the predicted value for that data point (a probability between 0 and 1), and it can be measured in various ways. For logistic regression, the error term is usually *not* measured by the raw residual—which is simply the difference between the predicted and actual values—but rather

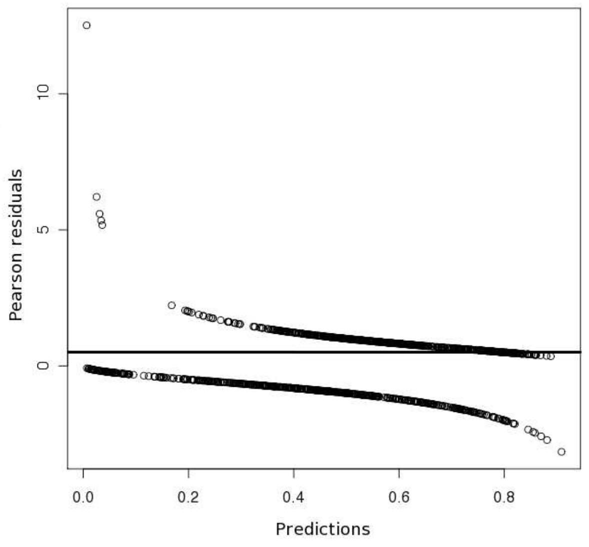


Figure 4.4: Pearson residuals vs. predictions for a model with fault and test-suite metrics

by alternatives such as the Pearson residual—which is the raw residual divided by the square root of the variance. Unlike linear regression, logistic regression does not assume that error terms are normally distributed, only that they are independent.

As Section 4.1 mentioned, the use of the test pool to calculate most of the fault characteristics threatens the assumption of logistic regression that the error terms of different data points are independent: the test-suite characteristics in one data point may be related to the fault characteristics in another data point. The threat, however, is negligible. Each test suite consists of no more than 1% of the test cases in the test pool, and most test suites are at least an order of magnitude smaller. Thus, each test suite's impact on the calculation of fault characteristics is tiny.

For logistic regression, plots of residuals can give some information about

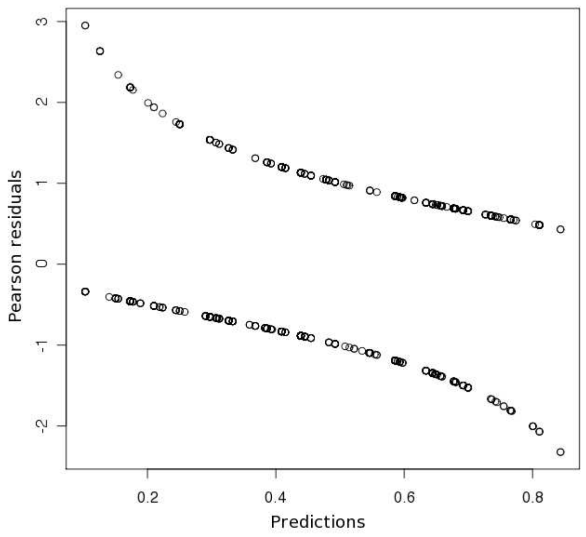


Figure 4.5: Pearson residuals vs. predictions for a model with only fault metrics

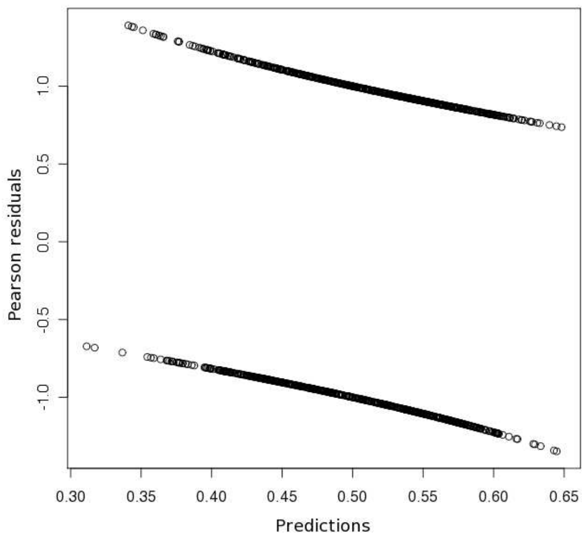


Figure 4.6: Pearson residuals vs. predictions for a model with fault and test-suite metrics

independence of error terms, although they can also be misleading [47]. As an example, Figure 4.4 plots the Pearson residuals from the multivariate model of Det for the PoolCovFaults data set for CrosswordSage (Table 4.15) against the actual values of Det for that same data set. (Plots for other data sets and for FreeMind were similar.) As expected, because of the binary nature of the predicted values, the plot shows two distinct curves. However, the linear trend line for the data is nearly flat, and the mean of the data is close to 0 (actually 0.009), suggesting that the error terms are indeed independent. Since the visually-apparent downward curve of the data is suspicious, two additional plots were drawn. Figure 4.5 plots the Pearson residuals from a multivariate model made up of just the fault metrics from the model in Figure 4.4. Figure 4.6 plots the Pearson residuals from a model made up of just the test-suite metrics from that same model. The downward curvature also appears in these plots, so it must not be a product of any non-independence between test-suite characteristics in some data points and fault characteristics in other data points.

4.4.7 Threats to validity

Four kinds of threats to validity are possible for this experiment.

Threats to internal validity are possible alternative causes for experiment results. Since one aim of the experiment was to identify characteristics of faults that make them easier or harder to detect, threats to internal validity here would cause certain kinds of faults to seem easier or harder to detect when in fact they are not.

For example, it could be that faults seeded nearer to the initial state of the program happened to be easier for reasons independent of their location in the program. But, because faults were seeded objectively and randomly, there is no reason to suspect that this is the case. Furthermore, studying Cov separately from Det (Section 4.3.1) helped to nullify one potential threat to validity: that class-level mutation faults appeared to be significantly easier to cover, even though this is not an inherent property of such faults. One threat to internal validity that remains, however, is that just two applications were studied. Since each had different results, some of those inconsistent results may have arisen by chance or through quirks of one or the other of the applications.

Threats to construct validity are discrepancies between the concepts intended to be measured and the actual measures used. The experiment studied metrics of faults and test suites that were intended to measure some more abstract characteristics (Table 4.1). These characteristics could have been measured in other ways. Indeed, if static analysis had been feasible for the applications studied, then it would have been a better way to measure certain fault metrics than using estimates based on the test pool. However, because care was taken to define the metrics such that they would not depend too much on the test pool (e.g., by using minima and maxima rather than averages), and because the test pool was much larger than any test suite, the author contends that the threats to validity posed by these estimates are not severe.

Threats to conclusion validity are problems with the way statistics are used. Because the experiment was designed to meet the requirements of logistic regression, it does not violate any of those requirements. The only known threat to conclusion validity is the sample size. While it is not recommended that the sample size or power be calculated retrospectively from the experiment data [34], the required theoretical sample size would probably be very large because so many independent variables were used. The sample size for this experiment was chosen not to achieve some desired level of power but to generate as many data points as possible in a reasonable amount of time and then perform an exploratory analysis, focusing on the fault characteristics. Since the fault characteristics frequently showed up as statistically significant, the sample size apparently served its purpose.

Threats to external validity limit the generalizability of experiment results. Any empirical study must suffer from these threats to some degree because only a limited sample of all possible objects of study (here, software, faults, and test suites) can be considered. In this experiment, only GUI-intensive applications, mutation faults, and event-coverage-adequate GUI test suites (with line and block coverage around 50%) were studied; different results might be obtained for different objects of study. Besides the limitations of the objects of study, the experiment is also unrealistic in that some of the faults are trivial to detect (they cause a major failure for every test case) and in real life would likely be eliminated before the system-testing phase. However, it is crucial to note that this experiment is more generalizable in one respect than most other studies of software testing. This work characterized the faults used in the experiment, and it presented the results in terms of those characteristics, to make clear precisely what impact the particular sample of faults had on the results. The same was done for test suites and their characteristics. This

Table 4.18: Calculated sample size at three effect sizes based on the PoolCovFaults data set and Det

	CrosswordSage			FreeMind		
	0.318	0.477	0.953	0.318	0.477	0.953
F.MutType	809	809	809	281	281	281
F.CovBef	576	256	64	824	366	92
F.AllRep	1080	1080	1080	1613	1613	1613
F.MinPred				3118	1386	347
F.MaxSucc	673	299	75			
F.MinWith	612	272	68	838	372	93
T.Events	1167	519	130	1558	693	173
T.Line	508	226	57	833	370	93
T.Pairs	1165	518	130	1856	825	207

Table 4.19: Calculated sample size at three effect sizes based on the SuiteCovFaults data set and Det

	CrosswordSage			FreeMind		
	0.318	0.477	0.953	0.318	0.477	0.953
F.MutType	399	399	399	447	447	447
F.CovBef	580	258	65	2091	929	233
F.AllRep	926	926	926	742	742	742
F.MinPred				2063	917	230
F.MaxSucc	664	295	74			
F.MinWith	593	263	66	1066	474	119
T.Events	1201	534	134	1331	592	148
T.Line	544	242	61	749	333	83
T.Pairs	1198	533	133	1637	728	182

work can serve as a model to help other researchers improve the external validity of their experiments.

4.4.8 Sample size for future experiments

While it is not recommended that the sample size of this experiment be used retrospectively to draw conclusions about its power [34], the data from this experiment can be used to calculate the required sample size at desired levels of significance, power, and effect size for future experiments in similar contexts. Appendix B explains the calculation.

Because the required sample size grows with the number of independent variables, it would make sense in a future experiment to study only the subset of test-suite and fault metrics that most influenced fault detection in this study. Table 4.18 lists one reasonable choice of these metrics. For each characteristic, in most cases, one metric was chosen. Exceptions are the characteristics of test-case length, which is omitted because it never influenced fault detection, and proportion of coverage, which is represented by the two somewhat orthogonal metrics of T.Line and T.Pairs. For some characteristics, other metrics than the ones chosen were nearly as influential. Because these metrics are highly correlated with the chosen metrics, the sample size calculated with them instead would be about the same.

Tables 4.18 and 4.19 show the influential metrics' calculated sample sizes at three reasonable effect sizes. The effect size is the smallest coefficient magnitude of interest, or practical significance, to experimenters. Unlike for power and significance, there is no standard value for effect size to use in experiments; it depends entirely on how the results will be used. In this case, since there is little or no precedent for such effect sizes in the software-testing literature, three effect sizes were chosen somewhat arbitrarily but in a range that seems reasonable. Recall that most of the metrics in Tables 4.18 and 4.19 range from 0 to 1. If incrementing or decrementing any of these metrics by a moderate amount—say, 0.3, 0.2, or 0.1—increases a test suite's odds of detecting a fault by a small but not tiny amount—say, a factor of 1.1—then experimenters may want to discover this effect, but they may not be interested in any weaker effects. Therefore, effect sizes of 0.318, 0.477, and 0.953 were chosen because they correspond to the odds of Det increasing by a factor of 1.10 when the metric in question is incremented or decremented by 0.3, 0.2, or 0.1, respectively. The smaller the effect size, the greater the calculated sample size, and the more likely that weak relationships between the independent and dependent variables will turn up as statistically significant. (For the binomial variables F.MutType and F.AllRep, the effect size is irrelevant.) The sample sizes were calculated using the customary significance and power values of 0.05 and 0.80 [3].

Even though the number of independent variables in these tables is still rather large, the calculated sample sizes are quite attainable, especially at the largest effect size (0.953). The metrics with the greatest calculated sample size at that effect size are F.AllRep and F.MutType because these metrics had uneven proportions of 0 and 1 values in the data sets. A future experiment could reduce the required sample size by better balancing these metrics in the data. When F.AllRep and F.MutType are ignored, the calculated sample sizes at the effect size of 0.953 are well below the actual sizes of the data sets in this experiment.

4.5 Conclusions

The experiment in this work tested hypotheses about faults in GUI testing. It showed that, in the context studied, certain kinds of faults were consistently harder to detect:

- faults in code that lies further from the initial program state ("deep faults"),

- faults in code that is *not* always executed more than once by an event handler, and
- faults in event handlers with more predecessors (in-edges) in the event-flow graph.

These results have implications for stakeholders in GUI testing. First, the results provide a picture of the faults most likely to go undetected. For users of GUI testing, this gives valuable information about reliability. (For example, deep faults are more likely to survive testing, but they may also be less likely to affect users of the software; therefore, additional effort to target them may not be warranted.) For researchers, understanding the faults often missed by GUI testing can guide the development of new testing techniques. The second implication is that the results suggest ways to target harder-to-detect faults. They show that these faults are more likely to reside in certain parts of the code (e.g., event handlers with more predecessors), so testers can focus on these parts. The results also show that increasing the line coverage or event-pair coverage of GUI test suites may boost detection of deep faults.

The experiment also tested hypotheses about GUI test suites, with several surprising results. Test suites with modestly greater code coverage (line, block, method, or class) did *not* necessarily detect more faults. Test suites with greater event-pair and event-triple coverage were *not* more likely to execute faulty code, yet they sometimes *were* more likely to detect faults; evidently, these suites executed more different paths within the covered portion of the code. Test-case granularity did *not* affect fault detection.

While results like these may interest the GUI-testing community, the experiment has broader implications. The results support the claims made by the thesis statement in Chapter 1. First of all, they show that simple, automatically-measurable characteristics of defects affect their susceptibility to detection by at least one form of software testing. This work proposed a new defect characterization. Unlike previous attempts to characterize defects, this work used characteristics that are objective, practical to measure, and—as the experiment shows—can significantly affect defects' susceptibility to detection. While the characterization here is not intended to be complete or definitive, it is an important first step from which further progress can be made.

Secondly, the results show that accounting for defect characteristics in empirical studies of software testing is feasible to do. The experiment serves as a proof of concept that the proposed methodology for accounting for defect characteristics is feasible. While it would be very difficult to compare the effort required for experiments done with different goals by different people in different contexts, it can be noted that this experiment was carried out by one student, with guidance from her advisor, in about half a year. Since many empirical studies are carried out by teams of experienced researchers, and some even have support personnel for tasks such as writing test cases, the amount of effort required for this experiment seems reasonable.

Finally, the results show that accounting for defect characteristics in empirical studies of software testing increases the validity of study results. Because, as the experiment showed, characteristics of the defects used in a study can impact the results, studies that do not account for defect characteristics will suffer threats to internal and external validity. Depending on the sample of defects, a studied testing technique may appear to be better or worse at detecting defects—either absolutely or relative to other techniques.

This is especially a problem when the goal of an empirical study is to evaluate testing techniques. It is imperative that evaluations of testing techniques be as accurate and effective as possible. Effective evaluations can convince practitioners that a new technique is worth adopting, or prevent them from wasting resources if it is not. Effective evaluations can illuminate the strengths and weaknesses of different, possibly complementary, techniques. They can even direct the invention of new techniques. Only if evaluations of testing techniques account for defect characteristics can they satisfactorily explain and predict the performance of testing techniques.

Chapter 5

Practical Applications to Regression Testing

The preceding chapters presented and demonstrated a better way to conduct empirical studies of fault detection in software testing. A key improvement was to use a statistical model, such as a logistic-regression model, to quantify the relationship between test-suite and fault characteristics and a test suite's ability to detect a fault. The statistical model can be used to show which test-suite and fault characteristics have important (statistically significant) effects on fault detection, and the direction and magnitude of those effects. This information can, as Chapter 1 explained, improve the science of testing. It can also, as this chapter shows, improve the practice of testing—specifically, the practice of regression testing.

Regression testing is an important step in the software-maintenance cycle that tests modifications made to previously tested software. An iteration of regression testing is an opportunity to catch changes that break the software. It is also an opportunity to learn how to make the next regression-testing iteration more effective.

To date, research has paid much attention to the problem of constructing a test suite initially but little attention to improving upon the initial suite's fault detection given some experience. Much research has studied the choice of testing technique and other factors that make for an effective test suite, without necessarily considering regression testing (Section 2.3). Research specifically on regression testing has typically focused on selecting or prioritizing test cases from the initial test suite in order to improve the speed or cost of fault detection [28, 55, 54]. But, so far, no one has used feedback from previous iterations of regression testing to design a test suite that detects more faults, or more important faults, in the current iteration.

This chapter shows how feedback—in the form of test-suite and fault histories from previous iterations of regression testing—together with statistical models of fault detection can help testers detect more, and more important, faults as regression testing progresses. The next section describes three such scenarios. To make the descriptions more palpable, each scenario is illustrated with a simple example based on fictitious feedback and models. While fictitious data, in its contrived neatness, helps to clearly explain the scenarios, real data can better show their strengths and weaknesses. That is why Section 5.2 demonstrates the scenarios with data from two versions of a real application. Section 5.3.1 outlines a more comprehensive evaluation of these scenarios and describes several additional scenarios.

5.1 Scenarios

Suppose that a team of testers needs to test version n of a software product. For versions $1, \ldots, n-1$, the test suites used, the faults found by each test suite, and the faults found after testing (e.g., by users) have been recorded. This section describes three ways that testers could use this information with a statistical model of fault detection to better test version n.

Statistical model (fabricated, not based on real data):

$\text{logit}(\Pr(\mathsf{Det})) = 0.3\mathsf{T.Events} + 3\mathsf{T.Pairs} - 10\mathsf{F.CovBef} + 40\mathsf{T.Pairs} \times \mathsf{F.CovBef}$

Characteristics of test suites used for versions $1, \ldots, n-1$:

T.Events	T.Pairs
2.0	0.20

Known faults in version 1:

	F.CovBef $= 0.3$	F.CovBef $= 0.7$	Total
Found by GUI testing	30	30	60

Known faults in versions $\mathbf{1, \ldots, n-1}$:

	F.CovBef $= 0.3$	F.CovBef $= 0.7$	Total
Found by GUI testing	200	200	400
Found by users, etc.	100	400	500

Figure 5.1: Statistical model and other data used in examples

Three scenarios are described, and each is illustrated with at least one example. To be concrete, the examples use fictitious data. (The next section demonstrates the scenarios with real data.) While the scenarios make no assumptions about the testing technique or statistical model being used, the examples assume that the testers are using GUI testing and a logistic-regression model similar to the ones developed in the previous chapter. The logistic-regression model and other data used in the examples are summarized in Figure 5.1.

5.1.1 Scenario 1: Varying test-suite characteristics

The coefficients of test-suite characteristics in the model can tell the testers how many more or fewer faults they can expect to detect by varying those characteristics, assuming that the distribution of faults does not change much from version to version. After exploring their options of which test-suite characteristics to vary, the testers can specify that the test suite for version n should have certain values for its characteristics. Whether the testers are creating the test suite for version n manually or automatically, and whether they are adding to the suite from version $n-1$ or generating a new suite, they can use the specified characteristic values as an adequacy criterion for the version-n test suite.

Example: The testers want to detect more faults in version n and are willing to spend up to 25% more time running test cases. One way to accomplish this would be to increase the normalized test-suite size (T.Events) from 2.0 to 2.5 (assuming that the number of GUI events in version n is the same as in version $n-1$). The statistical model predicts that this will increase the odds of fault detection (ratio of faults detected to faults undetected) by at least a factor of $e^{(0.3)(0.5)} \approx 1.2$. (The odds will increase even more if the larger test suite happens to cover additional event pairs.) For versions $1, \ldots, n-1$, the empirical odds of the suite detecting a fault were $400/500 = 0.8$—corresponding to an empirical probability of detection of about 0.45. With the increased test-suite size, the odds of the suite detecting a fault should then be $(1.2)(0.8) = 0.96$—corresponding to a probability of detection of about 0.5.

5.1.2 Scenario 2: Simulating testing

By plugging values for test-suite and fault characteristics into the model, the testers can run simulations to try out different kinds of test suites on different kinds of faults. They can do this descriptively, as the next scenario shows, to understand the correctness of the software. They can also do it prescriptively, as this scenario shows, to compare the effectiveness of hypothetical test suites.

Example: The testers want to simulate what would have happened if different test suites had been applied to versions $1, \ldots, n-1$, in order to help them specify the test suite for version n. They try plugging various combinations of test-suite characteristic values into the model: $\mathsf{T.Events} \in \{1.5, 2, 2.5, 3\}$ and $\mathsf{T.Pairs} \in \{0.15, 0.20, 0.25, 0.30\}$. For each combination of test-suite characteristic values, they calculate the predicted probability of detecting each known fault in versions $1, \ldots, n-1$ by plugging in its characteristic values. For testing version n, the testers might choose a combination of test-suite characteristic values that improves each known fault's detection probability, or one that reduces the cost of the test suite without decreasing the sum of detection probabilities.

5.1.3 Scenario 3: Understanding correctness

Using the model and the history of known faults, the testers can better understand the correctness of the tested software. The coefficients of fault characteristics show which kinds of faults are less likely to be detected during testing and, therefore, if present in the software are more likely to remain there. This is not a reliability estimate, since it considers faults rather than failure probabilities. But, like a reliability estimate, it helps the testers understand how incorrect the software is.

Example: The testers want to evaluate the correctness of version 1 after it has been GUI-tested but before other techniques (such as beta testing) have been applied. Because this is the first version of the software, the testers cannot use previous versions as a guide. Instead, they turn to the information they have for version 1 and the statistical model.

The summary of known faults in versions $1, \ldots, n-1$ makes the simplifying assumption that there are just two kinds of faults: shallow faults ($\mathsf{F.CovBef} = 0.3$) and deep faults ($\mathsf{F.CovBef} = 0.7$). The test suite for version 1 found 30 shallow faults and 30 deep faults. According to the model, the probability of detecting a shallow fault with this test suite is about 0.65 and the probability of detecting a deep fault is about 0.45. Thus, while the numbers of shallow and deep faults found by GUI testing were equal, the model predicts that about 16 shallow faults and about 37 deep faults remain in the software.

Example: This example makes the simplifying assumption that, to test versions $2, \ldots, n-1$, the testers have gradually augmented the test suite from version 1 rather than generating a new test suite for each version. (This avoids the problem of calculating a fault's joint probability of detection by multiple test suites.)

Suppose that version $n-1$ has been tested and released. In versions $1, \ldots, n-1$, testing found 200 of the 300 known shallow faults and 200 of the 400 known deep faults. The model estimates that there were originally about 307 shallow faults and 444 deep faults in total in versions $1, \ldots, n-1$. The testers can use this information in one of two ways, depending on whether they are more confident that the model is accurate or that all faults have been found. If they believe the model is accurate, then they can presume that some faults, especially deep faults, remain in version $n-1$ of the software, to be carried over to version n. If, on the other hand, they trust that virtually all faults in versions $1, \ldots, n-1$ are known, then they may be able to adjust the model to better fit their situation; this is an area for future research.

5.2 Empirical demonstration

This section demonstrates the scenarios as applied to the regression testing of a real application. The purpose of the demonstration is not to evaluate the effectiveness of these scenarios—evaluation lies beyond the scope of this dissertation, though it is discussed in Section 5.3.1—but simply to illustrate them with concrete data and explore their current limitations.

5.2.1 Data collection

This demonstration shows how testers might use the proposed scenarios to test a new version of FreeMind. It simulates the testing of two versions of FreeMind by using automatically-generated GUI test suites and seeded faults. The statistical models used in the demonstration come from Section 4.3. This section describes in more detail the hypothetical test situation in the demonstration and the actual data involved.

Table 5.1: Applications under test

	Lines	Classes	Events	EFG edges	EFG depth
FreeMind 0.7.1	9382	211	224	30146	3
FreeMind 0.8.0	15692	332	541	56102	4

Univariate model with T.Pairs (from Table 4.16):

$\text{logit}(\Pr(\mathsf{Det}|\mathsf{Cov})) = 54.3\mathsf{T.Pairs}$

Multivariate model (from Table 4.17):

$\text{logit}(\Pr(\mathsf{Det}|\mathsf{Cov})) = -0.395 - 0.472\mathsf{F.MutType} - 1.55\mathsf{F.Depth} + 0.669\mathsf{F.AllRep} - 53.8\mathsf{F.MinPred} - 15.2\mathsf{F.MaxWith} - 20.2\mathsf{T.Class} + 90.0\mathsf{T.Pairs}$

Figure 5.2: Statistical models used in demonstration

5.2.1.1 Application under test and statistical models

Section 4.3 presented several logistic-regression models, each fit to data from one of two applications: CrosswordSage (version 0.3.5) or FreeMind (version 0.7.1). Because these models are immature—the study in Chapter 4 not having been replicated yet—they cannot be expected to make accurate predictions about other applications. Therefore, it makes sense for this demonstration to consider only CrosswordSage or FreeMind. FreeMind is chosen because, with more versions and a larger user base, it is arguably the more interesting application.

The testers in this demonstration need to test FreeMind version 0.8.0. They have data on the test suite and known faults from the previously released version, 0.7.1. Table 5.1 gives vital statistics on these two versions.

The testers realize, of course, that a test suite will fail to detect any faults in code it does not cover. They are more interested in whether a test suite will fail to detect faults in code it does cover. (This assumption greatly reduces the number of

Table 5.2: Summary of test suites and fault detection

	T.Events	T.Pairs	T.Class	Pr(Det)	odds(Det)
FreeMind 0.7.1	2.70	0.0142	0.763	27/91 = 0.297	27/64 = 0.422
FreeMind 0.8.0 baseline	2.32	0.0142	0.765	30/93 = 0.323	30/63 = 0.476
FreeMind 0.8.0 improved	4.34	0.0216	0.777	32/93 = 0.344	32/61 = 0.525

seeded faults required for the demonstration.) Therefore, of the statistical models presented in Section 4.3, the models of the conditional probability of fault detection given coverage are most appropriate. Figure 5.2 shows the models that are relevant to this demonstration, which are a subset of those shown in Tables 4.16 and 4.17.

5.2.1.2 Test suites

For FreeMind 0.8.0, the testers want to compare the predicted effectiveness of a hypothetical *baseline test suite*, similar to the suite used for FreeMind 0.7.1, to that of a hypothetical *improved test suite*. The testers will then choose to create an actual test suite for FreeMind 0.8.0 whose characteristic values match either the baseline suite or the improved suite. Table 5.2 gives the characteristics of each suite.

This demonstration provides actual examples of the FreeMind 0.7.1 suite, the FreeMind 0.8.0 baseline suite, and the FreeMind 0.8.0 improved suite. The purpose is to compare the predicted effectiveness of the FreeMind 0.8.0 suites, relative to the FreeMind 0.7.1 suite, to their actual effectiveness. The demonstration only shows what happens in one case—one of many possible choices of test suites having the characteristic values of the FreeMind 0.7.1, FreeMind 0.8.0 baseline, and FreeMind 0.8.0 improved suites. (Section 5.3.1 describes a more effective evaluation in which multiple instantiations of the test suites are created and their average fault-detection

Table 5.3: Summary of faults (average characteristic values)

	F.MutType	F.Depth	F.AllRep	F.MinPred	F.MaxWith
FreeMind 0.7.1	0.637	0.253	0.198	0.01246	0.152
FreeMind 0.8.0	0.366	0.226	0.194	0.00495	0.133

abilities are compared.)

The improved suite was generated first, following the procedure in Section 4.2.1.2 and aiming for a normalized size (T.Events) equal to the normalized size for the FreeMind 0.7.1 SuiteCovFaults data set (the basis for the logistic-regression models in Figure 5.2). The baseline suite was created by reducing the improved suite, preserving event coverage. In other words, certain test cases from the improved suite were chosen to be copied to the baseline suite, using a greedy algorithm that covers all events with nearly as few test cases as possible [35]. The test suite for FreeMind 0.7.1 was constructed to resemble the FreeMind 0.8.0 baseline suite in event coverage and event-pair coverage. Like the FreeMind 0.8.0 baseline suite, it was constructed by again following the procedure in Section 4.2.1.2 and then reducing the suite until its event-pair coverage (T.Pairs) was the same as for the baseline suite.

5.2.1.3 Faults

To compare the effectiveness of the example test suites in this demonstration, faults in FreeMind 0.7.1 and 0.8.0 needed to be found or seeded. Seeded faults, as opposed to natural faults, were chosen for two reasons. First, the models used in the demonstration can only deal with a limited class of faults: those confined to a single line of source code. Many natural faults would fall outside this class. Second, the FreeMind project does not provide an easy way to reconstruct natural faults; the version-control system does not closely correspond with the defect-tracking system. (Section 5.3.1 describes a more effective evaluation that uses natural faults.)

Because the logistic-regression models used in this demonstration model the conditional probability of fault detection given fault coverage, faults were only seeded in lines covered by the FreeMind 0.7.1 suite or the FreeMind 0.8.0 baseline suite. And because new faults in version 0.8.0 would only arise in parts of the code that had been changed since version 0.7.1, faults were only seeded in changed files. With those restrictions, faults were seeded in FreeMind 0.7.1 and 0.8.0 following the procedure in Section 4.2.1.3. Table 5.3 summarizes the relevant characteristics of the faults. Assuming that most faults in version 0.7.1 have been found, either during testing or after release, the testers of version 0.8.0 would know the characteristics of faults in version 0.7.1 (other than F.MutType, which is an artifact of this demonstration). In the table, items known to the testers are shaded in gray.

5.2.1.4 Measurement of fault detection

The test suites were run to find out which faults they detected. To remove spurious reports of fault detection, all ⟨*test suite, fault*⟩ pairs where the fault is detected by a minority of test cases in the suite were checked manually. Table 5.2 summarizes the test suites' ability to detect the faults. As above, data items that would actually be available to the testers before testing FreeMind 0.8.0 are shaded in gray. (The testers would not know the exact values for Pr(Det) and odds(Det), but they would be able to estimate them based on faults found after testing.) The rest of the values in the table are used to compare the models' predictions to the actual fault-detecting abilities of the example test suites.

5.2.2 Demonstration of scenario 1: Varying test-suite characteristics

This scenario showed how testers could use a statistical model to estimate how many more or fewer faults they would detect by varying a test suite's characteristics.

This scenario assumes that the fault distribution does not change much from version to version. In reality, this is something that empirical studies or the FreeMind testers themselves would have to determine. In this demonstration, as Table 5.3 and the multivariate model in Figure 5.2 show, the faults in FreeMind 0.8.0 may actually be easier to detect than those in FreeMind 0.7.1 because they have notably smaller averages for F.MutType and F.MinPred.

Nevertheless, suppose that the testers decide to use the models to estimate the effect of varying the event-pair coverage (T.Pairs). They do not assume that the other important test-suite characteristic, T.Class, is held constant; if they vary T.Pairs, they would vary T.Class as a side effect. Therefore, the most appropriate model for them to use is the univariate model for T.Pairs in Figure 5.2.

When T.Pairs increases from 0.0142 to 0.0216, as it does from the FreeMind 0.8.0 baseline suite to the improved suite, the model predicts that the odds of fault detection will increase by a factor of $e^{(54.3)(0.0216-0.0142)} \approx 1.5$. The test suite for FreeMind 0.7.1 has the same event-pair coverage as the baseline suite, and its odds of fault detection are 0.422. Therefore, based on the model, the testers estimate that the baseline suite's odds of detection are also about 0.422, and the improved suite's odds of detection are about $(1.5)(0.422) = 0.633$. For the example baseline and improved suites, the difference in odds turns out to be smaller: 0.476 for the baseline suite and 0.525 for the improved suite.

5.2.3 Demonstration of scenario 2: Simulating testing

This scenario showed how testers could plug values for test-suite and fault characteristics into the model to simulate running different kinds of test suites on different kinds of faults.

The testers would do well to begin by checking the accuracy of the model—comparing the predicted probabilities of fault detection for the FreeMind 0.7.1 known faults against the actual proportion of known faults detected by testing. For the FreeMind 0.7.1 test suite and known faults, using the definition of a correct prediction from Section 4.2.2.3, the testers find that the model correctly predicts Det for 62 of the 91 faults. In reality, the testers might consider the predictions too inaccurate, but suppose for this demonstration that they proceed anyway.

If the testers are assuming that the distribution of faults does not change much from version to version, then they can plug the characteristics of each known fault in FreeMind 0.7.1 and characteristics of the baseline or improved suite into the model to compare the predicted effectiveness of the two suites. For each known fault in FreeMind 0.7.1, the predicted probability of detection turns out to be between 0.01 and 0.11 higher with the improved suite than with the baseline suite.

Table 5.4: Faults detected by the improved suite but not the baseline suite

Fault	F.MutType	F.Depth	F.AllRep	F.MinPred	F.MaxWith
1	1	0.5	0	0.00185	0.164
2	0	0.5	0	0.00185	0.164

If the testers are not assuming that the distribution of faults is stable from version to version, then they can instead plug characteristics of various hypothetical faults into the model. For characteristics they do not care to set, they can assume average values (i.e., 0 for centered, continuous data). For example, the testers can compare the baseline and improved suites' predicted effectiveness at detecting faults lying 2 events deep in the EFG (the maximal value of F.Depth for the FreeMind 0.7.1 faults) and in code that is not executed repeatedly (F.AllRep $= 0$). They find that the improved suite, relative to the baseline suite, is predicted to raise the probability of detecting these rather difficult faults by either 0.09 or 0.10, depending on whether F.MutType is 0 or 1.

Let us compare this prediction to what actually happens with the example baseline and improved suites. The improved suite detects two faults that the baseline suite does not. Their characteristics are shown in Table 5.4. As in the example above, these two faults lie 2 events deep in the EFG (the maximum fault depth for the FreeMind 0.8.0 faults as well as the FreeMind 0.7.1 faults) and in non-repeating code. A total of 22 of the FreeMind 0.8.0 faults have the same values for these characteristics. Of these 22, the baseline suite detects 3 (14%) and the improved suite detects 5 (23%). Thus, the difference in detection rates for these faults is just as predicted by the model (9% to 10%).

Because the statistical model is still immature, its predictions are often not as perfect as that. For example, the model only correctly predicts Det for 37 of 93 faults for the baseline suite and 33 of 93 faults for the improved suite. Some of this inaccuracy could possibly be remedied by adding a term to the model that accounts for the change in testability from version to version. This can be seen in the fact that changing the intercept term in the model from -0.395 to -2.5 (a value chosen by trial and error) increases the number of correct predictions for the baseline and improved suite each to 63 of 93.

5.2.4 Demonstration of scenario 3: Understanding correctness

This scenario showed how testers could use statistical models and the history of known faults to better understand the correctness of the tested software.

As in the demonstration of the previous scenario, consider faults lying 2 events deep in the EFG and in non-repeating code, like the two faults in Table 5.4. If the testers choose to test FreeMind 0.8.0 with the baseline suite used in this demonstration, then they will detect 3 such faults. Depending on the value of F.MutType, the model predicts that the baseline suite will have detected 22% to 31% of such faults. If instead they use the improved suite, then they will detect 5 such faults. The model predicts that the improved suite will have detected 31% to 41% of such faults. Thus, for both test suites, the model predicts that about 7 to 11 such faults remain undetected.

As mentioned in the demonstration of the previous scenario, for this demonstration there are actually 22 such faults in FreeMind 0.8.0. Thus, while the model correctly predicts that neither test suite has detected the majority of such faults, it underestimates the number of such faults that remain undetected.

5.3 Extensions of this work

This section describes two logical extensions of this work: a more comprehensive evaluation of Scenarios 1–3 (Section 5.3.1) and a list of additional scenarios that could be implemented and evaluated (Section 5.3.2).

5.3.1 Evaluation

The previous section demonstrated some of the practical applications of this work in a simulated iteration of regression testing. This simulation differed from reality in several major ways:

- The faults were seeded mutation faults instead of real mistakes made by developers.
- The faults were spread fairly evenly across the files changed between versions 0.7.1 and 0.8.0. In reality, faults tend to be concentrated in a few components [46, 49, 63].
- The testing technique was similar to the one used in the experiment in Chapter 4: GUI testing by random traversal of the EFG (optionally with test-suite reduction). Real testers might use a more systematic traversal of the EFG, and they would probably use other techniques besides GUI testing, such as unit testing.

- The test suites for version 0.8.0 try to cover the whole application, whereas real testers might only try to test the portions changed since version 0.7.1.
- The statistical model had been fit to data from FreeMind 0.7.1, one of the versions of FreeMind used in the demonstration. Real testers would probably need to use a model fit to data from software products other than their own.
- The statistical model's predictions had limited accuracy. To be useful in real situations, it would need to make better predictions.

The rest of this section outlines a more realistic evaluation of the proposed regression-testing scenarios. The next chapter lays out future work needed to make this evaluation possible.

The evaluation of the proposed regression-testing scenarios would need to address two perspectives often encountered in software engineering: validation and verification [63]. Validation asks: Did we build the right thing? That is, would these scenarios really help testers? Verification asks: Did we build the thing right? That is, do the scenarios work?

The scenarios could be *verified* with one or more case studies of the following design. The case study would look at a real, industrial software product with an actual history of test suites and known faults. Each test suite would be linked to the version of the software on which it was run, and each fault would be linked to each version of the software in which it appeared. A mature statistical model, built from software products other than the one being tested, would be available.

Like the demonstration in the previous section, the case study would compare the performance of "baseline" test suites to "improved" test suites. The study would answer the following research question: Are the improved suites better than the baseline suites at detecting faults that testers most want to detect, to the degree that the model predicts?

The researchers conducting the study would have a choice as to how they would create the baseline and improved suites. In the simplest case, they would use the actual test suite for each version as the baseline and create an improved suite to pair with each baseline suite. The improved suite would be predicted by the model to detect some greater percentage of faults, or of important faults, than the baseline suite. The researchers would then compare the model's predictions to the actual differences in fault detection between the baseline and improved suite. Alternatively, instead of creating just one baseline and one improved suite per software version, the researchers could see what happens in the aggregate by creating multiple baseline and improved suites. The baseline suites would include the original suite for the version and a number of "equivalent" suites—equivalent in the sense that the model predicts them to have similar fault-detecting abilities. The improved suites would consist of another set of equivalent suites, each predicted to improve similarly upon the fault detection of the baseline suites.

The case study would answer other important questions: How much does the distribution of faults change from version to version? How does the accuracy of the model change as the software evolves? The answers would help in understanding limitations to the way the model can be used. They might also point to ways to overcome these limitations, such as adding parameters to the model to account for software characteristics that change over time.

The scenarios would need to be *validated* by real testers in realistic situations. This part of the evaluation would need to ask: What is the learning curve for using these scenarios? Can tools be created to aid in carrying out the scenarios? How accurate must the statistical models be to help testers?

Another crucial issue in validation is whether these scenarios would complement or compete with existing regression-testing techniques, such as test selection and test prioritization. It may be that each have their niche: the test-suite-improvement scenarios for automatically-generated, "throw-away" test suites and the traditional regression-testing techniques for test suites that have been run on previous versions. Or maybe they can be used together. For example, when testing a version that adds a new feature, test cases from the previous version could be selected and prioritized, and the addition of new test cases to exercise the feature could be guided by a statistical model. The validation of the scenarios would need to look at the regression-testing process used by real testers to understand which of these possibilities would benefit them.

Before the scenarios can be verified and validated, major challenges must be overcome. Described more fully in the next chapter, they include developing more mature statistical models of fault detection and finding software projects with suitable histories of faults and test suites.

5.3.2 Additional scenarios

The three regression-testing scenarios described and demonstrated earlier in this chapter are just a few of the ways that statistical models and feedback from previous iterations of regression testing may improve regression testing. This section lists several additional scenarios. Like Scenarios 1–3, the examples for these scenarios refer to the testing situation and statistical model in Figure 5.1.

5.3.2.1 Scenario 4: Varying alternative test-suite characteristics

The relative magnitudes of coefficients of test-suite characteristics in the model can help testers evaluate alternative ways to modify a test suite. They can find that varying one characteristic by some amount is expected to have the same effect on fault detection as varying another characteristic by some other amount. The testers can choose which characteristic to manipulate.

Example: The testers wonder how much of an increase in event-pair coverage (T.Pairs) would be required to effect the same improvement in fault detection as a 25% increase in test-suite size; they may be able to increase event-pair coverage by this amount without increasing the test-suite size as much (or at all). The model estimates that increasing the normalized test-suite size from 2.0 to 2.5 (holding event-pair coverage constant) has the same effect as increasing the event-pair coverage from 0.20 to 0.25 (holding test-suite size constant). This is because $e^{(0.3)(2.5-2.0)} = e^{(3)(0.25-0.20)} = 1.2$.

5.3.2.2 Scenario 5: Targeting common or elusive kinds of faults

Interaction effects in the model can tell testers how to vary test-suite characteristics to improve detection of certain kinds of faults. Using the history of known faults, the testers can learn which kinds of faults should be targeted. They can look for characteristic values that arose commonly among all known faults, or among faults that eluded their previous test suites. Then they can look at interaction effects between fault characteristics and test-suite characteristics in the model to see how varying those test-suite characteristics would improve the odds of detecting those common or elusive faults. Alternatively, if no reasonable variation in test-suite characteristics would sufficiently improve those faults' odds of detection, then the testers can decide to target those faults with other techniques, such as inspections.

Example: Recall that the summary of known faults in versions $1, \ldots, n-1$ makes the simplifying assumption that there are just two kinds of faults: shallow faults ($\mathsf{F.CovBef} = 0.3$) and deep faults ($\mathsf{F.CovBef} = 0.7$). Deep faults, in this example, turn out to be the more common and the more elusive kind. Therefore, the testers want to improve detection of deep faults. Looking at the model, they see that they can achieve this by increasing event-pair coverage ($\mathsf{T.Pairs}$) because it has a positive interaction effect with fault depth. Increasing a test suite's event-pair coverage should increase the likelihood of detecting any fault, but it should give an extra boost to deeper faults.

Using the model and Equation 4.2, the testers can predict the probability that a shallow fault or a deep fault would be detected by various test suites. If the testers start with a test suite like the one for versions $1, \ldots, n-1$ and increase its event-pair coverage to 0.25, the predicted probability of detecting a shallow fault increases from 0.65 to 0.79. The predicted probability of detecting a deep fault sees a greater increase, from 0.45 to 0.79. Thus, with event-pair coverage at 0.25, the deep faults ($\mathsf{F.CovBef} = 0.7$) should no longer be any more elusive than the shallow faults ($\mathsf{F.CovBef} = 0.3$).

5.3.2.3 Scenario 6: Targeting severe faults

Some fault characteristics may be related to the severity of the fault. As in the previous scenario, interaction effects in the model can show how to target more severe faults or how to avoid wasting resources on less severe faults.

Example: The testers consider shallow faults to be more severe because users are more likely to encounter them. Since the interaction effect between event-pair coverage (T.Pairs) and fault depth (F.CovBef) in the model only shows the testers how to target less severe faults—those with higher F.CovBef values—they may choose not to take advantage of it like they did in the previous scenario's example. Instead, they may increase event-pair coverage or test-suite size just enough to reach a targeted detection probability for shallow faults (e.g., 0.70 with $\mathsf{T.Pairs} = 0.215$) without worrying about reaching a corresponding probability for deeper faults.

5.3.2.4 Scenario 7: Optimizing test-suite characteristics

It may be possible to optimize the function expressed by the statistical model to find the best values for the test-suite characteristics under given cost constraints. This will require cost functions for test-suite characteristics (What is the cost of increasing the characteristic by X amount?) and fault characteristics (What is the cost of failing to detect a fault with characteristic value X, and how likely is such a fault to occur?).

5.3.2.5 Scenario 8: Targeting fault-prone areas of the software

In other research, models have been developed to predict which areas of a software product are most fault-prone [46, 49]. Some of the logistic-regression models developed in the previous chapter predict the probability that a test suite covers a piece of code with certain characteristic values. It may be possible to combine these two lines of research: interaction terms in the model could show testers how to adjust their test suites to target fault-prone code.

5.4 Summary

This chapter presented potentially the first approach to regression testing that uses feedback from previous iterations of regression testing to detect more, and more important, faults in the current iteration. Before this potential can be realized, however, more mature statistical models of fault detection must be developed and other challenges must be overcome.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Recall this dissertation's thesis statement:

> Simple, automatically-measurable characteristics of defects affect their susceptibility to detection by at least one form of software testing. Accounting for these characteristics in empirical studies of software testing increases the validity of study results and is feasible to do.

Do simple, automatically-measurable characteristics of defects affect their susceptibility to detection? Yes. The experiment in Chapter 4 proposed and studied a set of such defect characteristics. It showed that the defect characteristics studied—each of which was simple and automatically measurable—affect defects' susceptibility to detection in at least one domain of testing. This set of characteristics improves upon the state of the art in defect characterization because it is practical to apply—whether in empirical studies or in practice—and proven to help explain defect detection in software testing.

Is accounting for defect characteristics in empirical studies of software testing feasible? Yes. The experiment applied the general methodology presented in Chapter 3, which showed how to control for defect characteristics and test-suite characteristics simultaneously in empirical studies of software testing. The experiments provided a proof of concept that the methodology works, as well as an example to its potential users.

Does accounting for defect characteristics in empirical studies of software testing increase the validity of study results? Yes. Chapter 1 explained the importance of being able to explain and predict the performance of testing techniques, which is only possible if all factors that significantly affect their performance are understood. The experiments showed that defect characteristics are among those factors. Furthermore, Chapter 5 showed how an understanding of defect and test-suite characteristics' effect on defect detection could help improve regression testing. Unlike existing approaches to regression testing, the approach taken in that chapter could help testers detect more faults, and more important faults, in evolving software.

6.2 Future Work

Although this dissertation makes substantial contributions toward understanding the role of defect characteristics in software testing, the field is fairly untrodden and much exploration remains.

6.2.1 Immediate future work

Immediate future work should focus on replicating the experiment in different contexts, honing the methodology, and expanding the fault characterization.

6.2.1.1 Experiments applying the methodology

To lend external validity to the experiment results on GUI testing, the experiment should be replicated with different GUI-based applications. One weakness of this experiment was that it explored only a narrow range of test-suite coverage levels; in the replicated experiment, a different strategy for GUI-test-suite generation, such as augmenting automatically-generated test suites with manually-generated test cases, could be used.

To lend even broader external validity to the experiment, it should be replicated in domains other than GUI testing and with faults other than one-line mutations. Especially interesting would be natural faults made by programmers.

To make this possible, analogues of the GUI-testing-specific metrics used in this experiment will need to be found for other testing domains. One metric, test-case length, already has an analogue: test granularity. For other metrics based on GUI events and the event-flow graph, function calls or "test grains" [53] might be substituted for events. A different graph representation, such as the control-flow graph, might be substituted for the event-flow graph.

The assumption that faults affect just one line of code will also need to be relaxed. This should be straightforward. For example, "coverage of the line containing a fault" could be generalized to "coverage of all the lines altered by a fault".

6.2.1.2 Methodology to account for fault characteristics in empirical studies of software testing

Instantiations or variations of the methodology could be developed to perform experiments more efficiently, requiring fewer test cases or fewer faults. For example, a factorial design may be more appropriate in some situations than this experiment's method of randomly generating data points.

6.2.1.3 Fault characterization

Additional relevant characteristics should be identified to more fully explain faults' susceptibility to detection. In particular, characteristics of the failures caused by a fault should be identified and studied. These may help explain faults' susceptibility to detection in ways that static fault characteristics cannot.

6.2.2 Long-term future work

In the long term, research should focus on further refining the methodology and the fault characterization, as well as bringing them into general use by researchers. Also, research in the future needs to evaluate the effectiveness of the regression-testing scenarios proposed in this work.

6.2.2.1 Practical applications to regression testing

Section 5.3.1 outlined a comprehensive evaluation of the scenarios for regression testing proposed in that chapter. Before the evaluations, and the scenarios themselves, can become a reality, several major challenges must be resolved.

The most important challenge is to continue developing the statistical models presented in this work until they become sufficiently mature. A mature model would be based on data from a wide variety of testing situations, including non-GUI test suites and real faults. It would be parameterized by a set of test-suite, fault, and other characteristics that together predict fault detection with a high degree of accuracy. Developing mature statistical models will require many studies like the one in Chapter 4 and contributions from many researchers, but the benefit to the science and practice of software testing will be worthwhile.

Until the statistical models become sufficiently mature, testers who wanted to try the scenarios could create data points from some version(s) of the software they are testing and fit a model to them. At first glance, it seems possible to use the history of test suites and known faults in an evolving software product as those data points. Unfortunately, this would not work for most software products if logistic regression is used. Logistic regression requires that each data point be independent, but usually the test suites from version to version are offshoots of the previous versions' test suites. Even if the test suites from version to version were independent, each test suite could only be paired with one fault as a data point. If, however, the testers were using an automated testing technique, they could run their own mini-experiment—generating and executing a unique test suite for each known fault in some version(s) of the software. Whether the testers would stand to gain more by spending the human and computational resources to generate those data points, and subsequently using the fitted statistical model, or to spend the same amount of resources just running more test cases on the version under test remains to be seen.

Another temporary solution while statistical models of fault detection remain immature might be to provide a way to calibrate an existing model to a new software product or organization (as with COCOMO II [13]). As the Section 5.2 showed, even just adjusting the intercept term in the model may dramatically improve the model's accuracy.

Besides developing mature statistical models, other challenges remain before an evaluation like the one proposed in Section 5.3.1 can be performed. Software products must be found for which the history of test suites and faults can be readily extracted from code repositories, modification requests or defect trackers, and other sources. Some strategy for grouping versions that have very few faults may need to be devised. If sets of equivalent baseline and improved suites are to be generated, an algorithm must be developed to do this.

6.2.2.2 Methodology to account for fault characteristics in empirical studies of software testing

The methodology should be extended to account for program characteristics as well as test-suite and fault characteristics. One solution *seems* to be to look at ⟨*test suite, fault, program*⟩ triples rather than ⟨*test suite, fault*⟩ pairs. But the solution is not so straightforward: the fundamental rule of logistic regression—independent data points—would dictate that each ⟨*test suite, fault, program*⟩ tuple must refer to a different program. Clearly, a better way to account for program characteristics is needed.

In some cases, it may be more interesting to study test cases than test suites. A variation of the methodology could be developed that replaces test suites with test cases. Since a single test case would not usually be expected to exercise all of the software under test, restrictions might be placed on the faults that may be paired with a test case.

Both for efficiency and for goodness of fit, alternatives to logistic-regression analysis, such as Bayesian networks, may be tried. Last but not least, the methodology should be further validated by using it in evaluations of testing techniques conducted by people other than the author.

6.2.2.3 Fault characterization

Future work for the short term included identifying additional relevant characteristics of faults. Even better for the long term would be to develop an underlying theory of fault types in software testing, rather than the somewhat ad hoc list of characteristics currently being used.

The characterization could be made even more applicable to practice by incorporating fault severity. Although severity is highly subjective, perhaps one or more fault characteristics (such as the distance of faulty code from the initial program state) could approximate it. Characteristics of the failures caused by a fault may help establish a connection to fault severity.

Eventually, the fault characterization, as well as test-suite and software characterizations, should mature to the point where these characteristics can accurately predict whether a given kind of fault in an application will be detected by a given test suite. The more accurate those predictions are, the more effective evaluations of testing techniques can be.

Appendix A

Preliminary Experiment

This appendix describes a first attempt to use the methodology of Chapter 3 in an experiment. The experiment studies the effects of a preliminary set of test-suite and fault characteristics on the probability that a test suite detects a fault. Data points were generated for two GUI-intensive applications using GUI testing.

Section A.1 describes the test-suite and fault characteristics studied. Sections A.2–A.4 present the experiment procedure, results, and conclusions. Throughout the appendix and in Section A.5, weaknesses in the experiment are brought to light. These weaknesses are instructive in understanding the proper use of the methodology, and they motivate the improved experiment in Chapter 4.

Table A.1: Test-suite and fault characteristics studied

	Abbrev.	Description
Test suite	T.Len'	Length of test cases
	T.Events'	Number of events
	T.Pairs'	Event-pair coverage / number of events
	T.Triples'	Event-triple coverage / event-pair coverage
Fault	F.MutType	Mutant type (0 for class-level, 1 for method-level)
	F.InMethod	In method (0 if not inside method, 1 if inside method)
	F.Branch	Branch points in faulty method's bytecode
	F.StmtDet	Estimated probability of detection by statement-coverage-adequate test suite

A.1 Test-suite and fault characterization

The test-suite and fault characteristics studied in this experiment were chosen based on two principles. First, the choice should be informed by previous research. Although the literature has little to say about fault characteristics (Section 2.2), it suggests several potentially relevant test-suite characteristics (Section 2.3). Second, all characteristics chosen should be measurable automatically and without special artifacts such as specifications. Otherwise, use of the characteristics in replications of this experiment or in practice would be impractical.

Table A.1 summarizes the characteristics studied. Some characteristics are marked with a prime (') to distinguish them from characteristics in Chapter 4 that intend to measure the same property but measure it somewhat differently.

A.1.1 Fault characteristics

As the next section explains, all faults in this experiment were mutation faults generated by a tool called MuJava. The characteristics F.MutType is the type of mutant, as classified by MuJava. *Method-level* mutants result from the more traditional mutation operators, which can operate on non-object-oriented code (e.g., inserting a decrement operator in front of a variable use). *Class-level* mutants operate on classes (e.g., changing the type of a data member). Prior to this study, there is no evidence to suggest that one type of fault is more difficult to detect than another, but this is an easy characteristic to include because it comes directly from the output of MuJava. MuJava further classifies each mutant by the specific mutation operator used to create it, but there are too many different mutation operators to study this as a characteristic.

An earlier version of this experiment [58, 60] found the mutation type (F.MutType) to be related to fault detection but hypothesized that the relationship really had more to do with the fault's location in the software—specifically, whether it lies inside a method or not. Class-level mutations are more likely than method-level mutations to occur in parts of a class other than its methods, such as declarations and initializations of data members. Therefore, this version of the experiment tests the hypothesis by including the characteristic F.InMethod, which is 1 if the fault lies inside a method and 0 otherwise.

Another fault characteristic, F.Branch, is the number of branch points in the method containing the fault, as counted in the method's bytecode. (The applications in this study were written in Java.) If the fault is not inside a method, F.Branch is 0. This characteristic is meant to approximate the number of branch points in the event handler(s) containing the fault, since prior work [65] hypothesized that faults in event handlers with more complex branching can only be detected by longer test cases.

Previous work by the author [59] has asserted that an effective way to characterize the faults detected by a testing technique is in terms of their susceptibility to detection by other techniques. Therefore, the third fault characteristic studied is F.StmtDet, a fault's estimated probability of detection by statement-coverage-adequate test suites.

A.1.2 Test-suite characteristics

Earlier research inspired each of the test-suite characteristics studied here. T.Len′, the length or granularity of test cases, has affected fault detection in previous studies [53, 65]. Although it is a characteristic of a test *case*, not necessarily a test *suite*, this experiment studies it by making all test cases in a suite have the same length. In this experiment, the length of a test case is the number of events minus the number of reaching events; this is explained in the next section.

Clearly, T.Events′, the size of a test suite, affects fault detection—as previous studies [39] have confirmed. In this experiment, test-suite size is measured as the total number of events in all of its test cases. This makes more sense, for the applications studied, than measuring the number of test cases because the time required to test them depends mostly on the number of events to be executed.

Many previous empirical studies have looked at the coverage of test suites. Some have found that test-suite coverage is a better predictor of fault detection than test-suite size [39]. Although there are many ways to measure coverage, this experiment considers two coverage metrics: GUI-event-pair coverage and GUI-event-triple coverage. These are, respectively, the number of unique length-two and length-three event sequences executed by a test suite. The characteristic T.Pairs′ is event-pair coverage normalized by test-suite size. T.Triples′ is event-triple coverage normalized by event-pair coverage. These characteristics are normalized to avoid confounding their influence on fault detection with that of test-suite size and each other. Although there was no clear stopping point in the length of event sequences to consider for coverage—this work could have looked at coverage of length-four and longer event sequences—length-two and length-three sequences seemed a reasonable choice for this experiment. If coverage of length-two and length-three sequences turned out to affect fault detection, then coverage of longer sequences could be studied in future experiments.

Because all test suites in this experiment are constructed to cover each GUI event at least once, no information about the events executed by a test suite, such as their complexity or content, is included among the characteristics studied.

A.2 Procedure

The experiment design follows the methodology from Chapter 3. A random sample of ⟨*test suite, fault*⟩ pairs was created, their characteristics and fault detection were measured, and the resulting data points were analyzed using logistic regression. Details follow.

A.2.1 Data collection

Data and other artifacts from this experiment have been packaged as a software-testing benchmark, available for download[1].

Table A.2: Sizes of the applications under test, their test suites, and their test pools

	Application Size			Cases in Suite		Cases in Pool	
	Lines	Classes	Events	Min.	Max.	Len.-2	Len.-20
CrosswordSage 0.3.5	2171	36	98	8	46	402	226
FreeMind 0.8.0	15692	332	541	117	424	1093	455

A.2.1.1 Applications under test

Two medium-sized, open-source applications were studied: CrosswordSage[2] (version 0.3.5), a crossword-design tool; and FreeMind[3] (version 0.8.0), a tool for creating "mind maps". Both are implemented in Java and rely heavily on GUI-based interactions. Table A.2 gives each application's size as measured by executable lines of code and classes, along with information about its test suites that will be explained later.

A.2.1.2 Sample size

For this experiment, an attempt was made to choose a sample size large enough to guarantee, with high probability, that the right test-suite and fault characteristics would be identified as statistically significant. Appendix B explains how sample size is calculated for logistic regression.

The sample-size calculation takes several parameters. For the level of significance and the power, the standard values of 0.05 and 0.80 were chosen [3]. The effect size is typically set at the smallest value of practical significance. Since the software-testing literature offered no precedent, an effect size of 0.3 was chosen be-

[1]`http://www.cs.umd.edu/~atif/Benchmarks/UMD2007b.html`

[2]`http://crosswordsage.sourceforge.net`

[3]`http://freemind.sourceforge.net`

cause it seemed to provide a reasonable balance between identification of practically-significant relationships and feasible sample size.

The sample-size calculation also requires sample data from a pilot study. For this experiment, the pilot study consisted of 100 $\langle$*test suite, fault*$\rangle$ pairs for CrosswordSage. These data points were generated following a procedure very similar to the one described in the rest of this section.

The calculated sample size turned out to be 146. Therefore, 146 $\langle$*test suite, fault*$\rangle$ pairs were generated for the experiment.

Unfortunately, some problems with the calculation were discovered after the experiment was completed. First, errors were found in the implementation of the formulas in Appendix B. In particular, α was not divided by the number of independent variables (test-suite and fault characteristics). These errors were subsequently corrected, and the implementation was tested using the inputs and outputs in [30]. Second, as explained later in this section, the rate of false reports of fault detection in the data was unacceptably high. Some of the same steps taken to correct the experiment data were also taken to correct the pilot-study data: reports of detection were checked against coverage data (if a test case does not cover the faulty code, it should not detect the fault), and the corrected oracle procedure was used. Table A.3 gives the recalculated sample size for each independent variable, using the same parameters as in the original calculation. The recalculated sample size for the experiment is the maximum of these: 13745.

That the actual sample size is much smaller than the recalculated sample size does not invalidate the experiment. It only implies that there is a greater-

Table A.3: Recalculated sample size for each independent variable

T.Len'	T.Events'	T.Pairs'	T.Triples'	F.MutType	F.InMethod	F.Branch	F.StmtDet
4205	13745	3500	7825	394	127	768	12874

than-expected chance of failing to identify test-suite and fault characteristics as statistically significant.

A.2.1.3 Test pool

Following a common practice in software-testing studies (e.g., [65]), this experiment constructed test suites by selecting test cases from a fixed set, called the *test pool*. A test case may appear in more than one test suite. Therefore, while the test pool must be small enough that all of its test cases can be executed in a reasonable amount of time, it must also be large enough that test suites picked from the pool are sufficiently different from each other. The latter requirement depends on the number of test cases per test suite. Table A.2 lists the minimum and maximum number of test cases per test suite for each application studied.

Algorithm 1 shows the algorithm used to construct the test pool, which resulted in 19 "buckets" of test cases, one for each length. Test cases did not exceed 20 events because the current version of the test-case-replayer component of GUITAR, the tool used to execute the test cases, often fails spuriously from timing problems with longer test cases. The minimum test-case length was set at 2, rather than 1, because the targeted number of test cases in each bucket exceeded the number of possible length-1 test cases—yet it was desired that each bucket in the test pool initially contain an equal number of test cases with no duplicates.

The test cases were built such that each length-2 to length-19 test case was a prefix of a length-20 test case, which permitted a time-saving shortcut in test-case execution: the experiment only needed to run the length-20 test cases, checking the GUI state after each intermediate event, to obtain results for the length-2 to length-19 test cases as well. This shortcut has been used previously by Xie and Memon [65].

It was desired that each GUI event be part of $bucket_1$, yet, technically, not every GUI event can be the first in a test case because not every GUI event can be executed in the initial state of the application. Sometimes a sequence of *reaching events* had to be prefixed to an event in $bucket_1$ to bridge the gap between the initial state and the "first" event in the test case. The reaching events did not count toward the length of the test case.

The number of iterations, $iters$, was chosen to limit the probability that two test suites picked from the pool would share more than a small percentage of test cases. After the test cases were executed, however, the pool size had to be reduced. Test cases that failed spuriously on the ith event (as determined by examining statement coverage of lines with seeded faults) were removed from $bucket_i$ to $bucket_{20}$. Table A.2 shows the resulting number of length-2 and length-20 test cases in the pool.

Algorithm 1 Algorithm for constructing the test pool. $succs(event)$ is the set of successors of the event in the EFG. For a test case in $bucket_i$, $last(testCase)$ is the ith (i.e., last) event in the test case. and $covSuccs(testCase)$ is the set of events such that $testCase \circ event \in bucket_{i+1}$.

```
1:  $bucket_1 \leftarrow$ {all events in application}
2:  $i \leftarrow 0$
3:  **while** $i <$iters **do**
4:      **for** $tc \in bucket_1$ **do**
5:          $tc' \leftarrow tc$
6:          **for** $i$ from 2 to 20 **do**
7:              $uncov \leftarrow succs(last(tc')) - covSuccs(tc')$
8:              **if** $uncov = \varnothing$ **then**
9:                  $covSuccs(tc') \leftarrow \varnothing$
10:                 $uncov \leftarrow succs(last(tc'))$
11:             **end if**
12:             $e \leftarrow$ random event in $uncov$
13:             $covSuccs(tc') \leftarrow covSuccs(tc') \cup e$
14:             $tc' \leftarrow tc' \circ e$
15:             $bucket_i \leftarrow bucket_i \cup tc'$
16:         **end for**
17:     **end for**
18:     $i \leftarrow i + 1$
19: **end while**
```

A.2.1.4 Test suites

Each test suite was constructed by randomly selecting test cases from some fixed, randomly chosen bucket in the test pool until the test suite covered all GUI events that the test pool covered. A test case was only added to the suite if it contained some event that the suite did not yet cover.

A.2.1.5 Faults

To obtain a sample of faults, experimenters typically use one of three approaches: identifying actual faults made by developers, seeding faults by hand, or seeding faults mechanically. Each approach has its pros and cons, discussed elsewhere [4].

Because of the large number of faults needed for this study, the third approach, automatic seeding, was chosen. Fault seeding was done with the tool MuJava[4]. MuJava seeds a fault in a Java class by applying a *mutation operator*, which makes a syntactically small change to the source code (e.g., replacing a relational operator with another relational operator or changing the access level of an instance variable). As explained in Section A.1, the mutation operators fall under two main categories: class-level and method-level. Method-level mutants can be seeded inside the methods of a class, while class-level mutants can be seeded either inside methods or elsewhere in a class.

For each application studied, all possible mutants were created, and 146 of those mutants were randomly selected. Correspondingly, 146 faulty versions of each

[4]`http://www.ise.gmu.edu/~ofut/mujava/`

application were created, each containing one fault.

No attempt was made to eliminate *equivalent mutants*—those that do not affect the semantics of the application. In the results, this might affect the intercept term in the logistic-regression models—a fairly minor detail. It would only affect the more important part of the logistic-regression models—the coefficients, specifically the coefficient for F.MutType and any correlated characteristics—if different proportions of the class-level and method-level mutation faults were equivalent mutants.

Since opportunities for seeding different types of mutants (i.e., mutants generated by different mutation operators) are more common for some types of mutants than others, some mutant types in the sample were more common than others. However, this should not affect the experiment.

A.2.1.6 Measurement of independent and dependent variables

Several of the characteristics of $\langle$*test suite, fault*$\rangle$ pairs listed in Table A.1 could be observed before any test cases were executed. This was true of all of the test-suite variables, which were straightforward to measure. F.MutType was easily obtained from the output of MuJava. F.InMethod and F.Branch were both found by identifying the method (if any) in which a fault resides. For F.Branch, the control-flow graph created by Sofya[5] for that method was analyzed.

The dependent variable in this experiment is fault detection, Det. If, for a $\langle$*test suite, fault*$\rangle$ pair, the test suite detects the fault, then Det $= 1$; otherwise, Det $= 0$. To determine this, of course, the test cases in the test pool had to be executed.

[5] `http://sofya.unl.edu`

Using GUITAR (Section 2.1), each test case was executed on the original, or *clean*, version of the application and on each faulty version. If the oracle information gathered by GUITAR when running a test case differed between the clean version and a faulty version, then GUITAR reported that the test case detected that fault. Since length-2 through length-19 test cases in the test pool were merely prefixes of length-20 test cases, the data for each shorter test case were gotten while running the length-20 test case by capturing oracle information after each event in the test case. In addition, for each length-20 test case, a listing of all program statements it covered in the clean version was collected by Instr[6]. Because execution of all test cases in the pool required hundreds of hours of computation time, execution was distributed across multiple computers.

When the experiment was run, some false reports of fault detection were anticipated. Because of timing problems in GUITAR, test cases sometimes fail to run to completion, making it appear as if a fault has been detected when really it has not been. In addition, GUITAR by default detects even trivial differences in oracle information, such as in the coordinates of GUI components, which may not actually indicate fault detection. When this work was originally published [58, 60], several steps had been taken to eliminate false reports of detection. GUITAR had been set to ignore usually-trivial differences in oracle information, such as the coordinates of components. Reports of fault detection had been checked against the statement-coverage data for length-20 test cases; if the test case did not cover the line containing the fault, then it (and the length-2 to length-19 test cases inside it)

[6]`http://www.glenmccl.com/instr/index.htm`

could not have detected the fault. As mentioned in the procedure for constructing the test pool, if the statement-coverage data showed that a length-20 test case failed spuriously on the ith event, then the length-i and longer test cases inside it had been discarded from the test pool.

After the work was published, several contributing factors to the incorrect fault-detection results were diagnosed and mitigated. The timing problems with GUITAR had been exacerbated by running it on a publicly available distributed system called Condor[7], so test cases with questionable results were rerun on a private cluster at the University of Maryland. The component of GUITAR that implemented the oracle procedure, which compares two files of oracle information to see if they differ, had contained logic errors causing it to report non-existent differences, so those were corrected. Also, the fault-detection data were checked more rigorously. A statement-coverage report was collected after each event in each test case, rather than just at the end of each length-20 test case, and reports of fault detection were checked against it. Each supposedly-detected fault was also manually inspected to determine if it really could be detected by GUI testing; if not, reports of its detection were eliminated. For CrosswordSage, 20 of 58 reports of $\mathsf{Det} = 1$ turned out to be incorrect; for FreeMind, 53 of 76. The results reported here use the corrected data for Det.

$\mathsf{F.StmtDet}$ was calculated using the fault-detection data and the statement-coverage data. Recall that this characteristic estimates a fault's susceptibility to detection by statement-coverage-adequate test suites. To make this estimate, 100

[7]`http://www.cs.wisc.edu/condor/`

test suites of length-20 test cases were constructed using a procedure similar to the one for the other test suites in the experiment, except that a different coverage criterion was used: instead of covering all events, these test suites were made to cover all statements covered by the test pool.

A.2.2 Data analysis

For each application under test, the data set of 146 ⟨*test suite, fault*⟩ pairs was analyzed with logistic-regression analysis (Section 4.2.2). Two kinds of logistic-regression models were fit to the data: univariate and multivariate. In each univariate model, one test-suite or fault characteristic was the independent (explanatory) variable and Det was the dependent (response) variable. A multivariate model for each application was created as follows:

1. An initial multivariate model, using all test-suite and fault characteristics as independent variables, was built.

2. The model was reduced using stepwise regression based on AIC.

3. Each two-way interaction between a fault characteristic and a test-suite characteristic in the reduced model was added to the reduced model.

4. The model was reduced again using stepwise regression.

Stepwise regression based on AIC removes superfluous independent variables from the model, leaving the set of independent variables that provides the best balance between model fit and parsimony. Before model-fitting, all non-categorical data was

Table A.4: Data summary

	CrosswordSage			FreeMind		
	Min.	Mean	Max.	Min.	Mean	Max.
T.Len'	2	11.4	20	2	10.6	20
T.Events'	16	308.2	551	824	1776	2580
T.Pairs'	0.694	0.952	2.688	0.774	0.896	0.926
T.Triples'	0.762	1.161	1.348	0.397	0.900	1.043
F.MutType	0	0.411	1	0	0.568	1
F.InMethod	0	0.911	1	0	0.945	1
F.Branch	0	17.6	58	0	16.1	94
F.StmtDet	0.000	0.253	1.000	0.000	0.147	1
Det	0	0.260	1	0	0.158	1

centered (the mean was subtracted). Model-fitting and other statistical calculations were performed with the R software environment[8].

A.3 Results

Before analyzing data, it is worthwhile to look it over. Table A.4 gives an overview of the data collected for CrosswordSage and FreeMind. The table reveals a number of slight differences between the two applications but only a few major differences:

- T.Events', the number of events per test suite, is much greater for FreeMind because it is a larger application. However, when T.Events' is normalized by the number of events per application (Table A.2), FreeMind's test suites turn out to be only slightly larger than CrosswordSage's.

- F.MutType is less than 0.5 for CrosswordSage but greater than 0.5 for FreeMind. This means that the majority of CrosswordSage's faults are class-level mutations, while the majority of FreeMind's faults are method-level mutations.

[8] `http://www.r-project.org/`

Table A.5: Univariate models of Det

	CrosswordSage			FreeMind		
	Int.	Coef.	Sig.	Int.	Coef.	Sig.
T.Len'	−1.06	0.0450		−1.68	0.00767	
T.Events'	−1.06	0.00210		−1.68	0.000235	
T.Pairs'	−1.05	−0.598		−1.71	9.62	
T.Triples'	−1.05	1.66		−1.68	0.758	
F.MutType	−1.01	−0.0910		−1.25	−0.854	◦
F.InMethod	0.154	−1.35	*	−0.511	−1.26	
F.Branch	−1.10	−0.0250	*	−1.70	0.00112	
F.StmtDet	−2.16	8.00	***	−2.50	8.89	***

- The mean values of F.StmtDet and Det are larger for CrosswordSage than for FreeMind, indicating that CrosswordSage's faults are more susceptible to detection.

A.3.1 Univariate models

Table A.5 gives the univariate logistic-regression models that were fitted to the data. Each row of the table holds a compact representation of a univariate model for each application. For example, the first row of the left side of the table represents this model for CrosswordSage:

$$\text{logit}(\Pr(\mathsf{Det})) = -1.06 + 0.0450\mathsf{T.Len}'.$$

The table also shows the level of statistical significance for each independent variable in the model, as determined by a chi-square test of deviance. Independent variables with significance levels of "◦", "*", "**", and "***" have p-values less than or equal to 0.1, 0.05, 0.01, and 0.001, respectively. The smaller the p-value, the more likely it is that the theoretical (true) coefficient value is non-zero and has the same sign as the estimated coefficient.

For both applications, F.StmtDet is the most significant independent variable. (The large magnitude of its coefficient does *not* necessarily indicate that it is more important than other variables, since the variables have different means and ranges.) Its coefficient of 8.00 (CrosswordSage) or 8.89 (FreeMind) indicates that a unit increase in F.StmtDet is predicted to increase the odds of Det by a factor of $e^{8.00}$ or $e^{8.89}$. But, since F.StmtDet only ranges from 0 to 1, it only makes sense to look at smaller increments. If F.StmtDet increases by 0.1, say from 0.3 to 0.4, then the odds of Det are predicted to increase by a factor of $e^{(8.00)(0.1)} \approx 2.23$ (CrosswordSage) or $e^{(8.89)(0.1)} \approx 2.43$ (FreeMind).

Additional fault characteristics, but no test-suite characteristics, turn out to be statistically interesting. For CrosswordSage, these are F.InMethod and F.Branch: faults not inside methods and faults surrounded by less branching are more likely to be detected. More precisely, the odds of detecting an intra-method fault are $e^{-1.35} \approx 0.259$ times the odds of detecting an extra-method fault, and the odds of detecting a fault decrease slightly, by a factor of $e^{-0.0250} \approx 0.975$, with each additional branch point in the enclosing method. For FreeMind, the other statistically interesting characteristic is F.MutType: the odds of detecting a method-level mutation fault are less than half ($e^{0.854} \approx 0.426$) the odds of detecting a class-level mutation fault.

Table A.6: Multivariate models of Det

	CrosswordSage		FreeMind	
	Coef.	Sig.	Coef.	Sig.
Intercept	-13.4		-7.18	
T.Len'			-0.346	*
T.Events'	0.0588	**		
T.Triples'	-41.6	*	-1.14	*
F.MutType	3.17			
F.InMethod	6.41	*		
F.Branch	-0.0890			
F.StmtDet	21.5	***	101.1	***
T.Triples' $\times$ F.StmtDet			12.2	***
Null deviance	167.4		127.2	
Deviance	28.6		17.5	
AIC	42.6		27.5	
Sensitivity	$37/38 = 0.974$		$23/23 = 1.000$	
Specificity	$101/108 = 0.935$		$117/123 = 0.951$	

A.3.2 Multivariate models

Table A.6 shows the multivariate models and the significance level of each independent variable in them. The multivariate model for CrosswordSage is

$$\begin{aligned}\text{logit}(\Pr(\mathsf{Det})) &= -13.4 + 0.0588\mathsf{T.Events}' + -41.6\mathsf{T.Triples}' + 3.17\mathsf{F.MutType} + \\ &\quad 6.41\mathsf{F.InMethod} + -0.0890\mathsf{F.Branch} + 21.5\mathsf{F.StmtDet}.\end{aligned}$$

For FreeMind, the multivariate model is

$$\begin{aligned}\text{logit}(\Pr(\mathsf{Det})) &= -7.18 + -0.346\mathsf{T.Len}' + -1.14\mathsf{T.Triples}' + 101.1\mathsf{F.StmtDet} + \\ &\quad 12.2\mathsf{T.Triples}' \times \mathsf{F.StmtDet}.\end{aligned}$$

Additionally, the table shows some measures of goodness of fit for each model (Section 4.2.2).

Table A.7: Univariate models of Det with F.StmtDet

	CrosswordSage		FreeMind	
	Coef.	Sig.	Coef.	Sig.
Intercept	−2.16		−2.50	
F.StmtDet	8.00	***	8.89	***
Null deviance	167.4		127.2	
Deviance	56.5		38.8	
AIC	60.5		42.8	
Sensitivity	37/38 = 0.974		18/23 = 0.783	
Specificity	99/108 = 0.917		122/123 = 0.992	

When interpreting multivariate logistic-regression models, it is important to consider strong correlations among the independent variables, a condition known as *multicolinearity*. Multicolinearity can, as in the models in Table A.6, lead to extreme values or unexpected signs for coefficients. For both applications, T.Len', T.Events', T.Pairs', and T.Triples' are all strongly correlated with each other (correlation values of magnitude 0.63 to 0.97)—despite the attempt to normalize T.Pairs' and T.Triples' (Section A.1). F.MutType, F.InMethod, F.Branch, and F.StmtDet are also correlated with each other, though less so (correlation values of magnitude 0.10 to 0.39). Multivariate models suffering symptoms of multicolinearity are not totally invalid, but they are harder to interpret and less likely to be accurate for other data sets.

Different sets of independent variables are chosen for the two applications' multivariate models. For both, a mixture of test-suite and fault characteristics help predict fault detection. Interestingly, the model for FreeMind includes a positive interaction effect between T.Triples' and F.StmtDet, partly offsetting the unexpected negative coefficient of T.Triples'.

The multivariate models fit the data remarkably well, with sensitivity and

Table A.8: Multivariate models of Det without F.StmtDet

	CrosswordSage		FreeMind	
	Coef.	Sig.	Coef.	Sig.
Intercept	-0.179		-145	
T.Events'	0.00234		0.00290	*
T.Pairs'			11200	*
T.Triples'			-21.7	
F.InMethod	-1.03	*	144	
F.Branch	-0.0193		0.0154	
T.Pairs' $\times$ F.InMethod			-11100	**
Null deviance	167.4		127.2	
Deviance	158.0		107.2	
AIC	166.0		121.2	
Sensitivity	$22/38 = 0.579$		$12/23 = 0.522$	
Specificity	$69/108 = 0.639$		$85/123 = 0.691$	

specificity close to 1. This is mostly due to the influence of F.StmtDet, which has a correlation of 0.837 (CrosswordSage) or 0.866 (FreeMind) with Det. When all other variables are dropped from the model, leaving just the univariate model for F.StmtDet, as shown in Table A.7, the sensitivity and specificity remain high: 0.974 and 0.917 (CrosswordSage) or 0.783 and 0.992 (FreeMind), respectively. On the other hand, when a multivariate model is fit without using F.StmtDet, as shown in Table A.8, the sensitivity and specificity drop to 0.579 and 0.639 (CrosswordSage) or 0.522 and 0.691 (FreeMind).

A.4 Discussion

This section interprets the results and discusses their implications.

A.4.1 Threats to validity

The results should be interpreted while keeping in mind possible threats to the validity of the experiment.

Threats to internal validity are possible alternative causes for experiment results. The main threat to internal validity here is the possibility of incorrect data for fault detection (Det). As Section A.2.1 explained, numerous steps were taken to clean the data. However, there were simply too many ⟨*test case, fault*⟩ pairs involved in the calculation of Det to examine every one in detail. A few incorrect values for Det may remain. As long as the characteristics of the ⟨*test suite, fault*⟩ pairs corresponding to those incorrect values are randomly distributed, this should not affect the results. Characteristics for which this may not be the case are F.InMethod—because tracking of coverage of extra-method faulty lines, and therefore corrections of Det results for this faults, were approximate—and the rest of the fault characteristics—because they are modestly correlated with F.InMethod.

Threats to construct validity are discrepancies between the concepts intended to be measured and the actual measures used. Two fault characteristics raise concerns here. One is F.Branch, the number of branch points in the bytecode of the method containing a fault. It is intended to approximate the complexity of the event handler(s) containing a fault. However, there is no definite relationship between methods and event handlers. The other is F.StmtDet, a fault's estimated probability of detection by statement-coverage-adequate test suites. What F.StmtDet actually measures in this experiment is the fault's probability of detection by a test suite *selected from the test pool* that *covers all statements covered by the pool.*

Threats to conclusion validity are problems with the way statistics are used. This experiment faces two main threats to conclusion validity. First, the sample size is much smaller than the sample size required to achieve the desired level of accuracy in results (significance = 0.05, power = 0.80, effect size = 0.3). With the current sample size, it is likely that not all independent variables that "should" be statistically significant are identified as such. This is especially true for independent variables whose coefficients in the logistic-regression models are very small.

The other major threat to conclusion validity is a consequence of the requirement in logistic-regression analysis that all data points be independent. Because the test suites in this experiment are built from a test pool, and furthermore because some test cases in the pool are prefixes of others, the test suites in the experiment are not independent. The problem is compounded by the fact that the pools of longer test cases had to be substantially reduced because of test cases failing to run to completion (Section A.2.1). Consequently, each length-20 test case consisted of a large portion of the length-20 test pool: on average, 10% for CrosswordSage and 40% for FreeMind. This serious problem is corrected in the experiment in Chapter 4.

Threats to external validity limit the generalizability of experiment results. Like any software-testing experiment, this one considers a limited sample of test suites, faults, and software under test. It is not yet clear how conclusions drawn from the study would apply to other forms of testing, other kinds of software, and real faults. However, this experiment has greater external validity than many previous studies of software testing because it characterizes the faults used and shows how they, along with test-suite characteristics, affected the results.

A.4.2 Fault characteristics

By far the independent variable most strongly related to fault detection in this experiment is F.StmtDet, a fault's estimated probability of detection by a statement-coverage-adequate test suite. This fault characteristic is strongly correlated with fault detection. The question is, to what extent is this information useful? It *may* indicate that event-coverage-adequate suites and statement-coverage-adequate suites mostly detect the same faults. However, because of the way F.StmtDet was calculated, we cannot be sure. The statement-coverage-adequate suites used to calculate it were selected from the same test pool as the experiment suites, and they only covered statements that were covered by the test pool. This is just too incestuous: even random test suites from the test pool would have fault-detection abilities not unlike the experiment suites. Thus, the results for F.StmtDet may not tell us much after all.

For CrosswordSage, F.InMethod, whether or not a fault resides inside a method, and F.Branch, the amount of branching surrounding a fault, also appear to affect fault detection. Both are statistically significant in their univariate models (Table A.5), and F.InMethod is significant in the full multivariate model (Table A.6) and the model without F.StmtDet (Table A.8). Except in the full multivariate model (Table A.6), where F.InMethod apparently has an unexpected sign because of multicolinearity with F.StmtDet and F.Branch, F.InMethod has a negative effect on fault detection—faults inside methods are less likely to be detected—and F.Branch has a slight negative effect on fault detection—faults in methods with more branching are slightly less likely to be detected.

For FreeMind, the only fault characteristic that shows up as statistically interesting is F.MutType, and only in its univariate model (Table A.5). That model suggests that class-level mutation faults in FreeMind are more likely to be detected than method-level mutation faults.

Section A.1 suggested that faults in methods with more branching might only be detected by longer test cases. This would have appeared in the logistic-regression models as an interaction effect between F.Branch and T.Len'. However, no such interaction appeared in any of the models.

A.4.3 Test-suite characteristics

Surprisingly, test-suite characteristics in this experiment did not seem to affect fault detection very much. No test-suite characteristics show up as statistically significant in their univariate models (Table A.5). However, T.Events' and perhaps other characteristics must affect fault detection at least some; either the range or the number of test suites studied is apparently insufficient to reveal the true effects of these characteristics. As Table A.3 shows, the calculated sample sizes for the test-suite characteristics are higher than for most of the fault characteristics.

There were a few statistically significant results for test-suite characteristics, although none is consistent across all models. In the full multivariate model for

CrosswordSage (Table A.6) and the multivariate model without F.StmtDet for FreeMind (Table A.8), T.Events', the test-suite size measured in events, has a positive statistically significant effect on fault detection. In the multivariate model without F.StmtDet for FreeMind (Table A.8), T.Pairs', the event-pair coverage, also has a positive, statistically significant effect, although its coefficient is inflated by multicolinearity. In both applications' full multivariate models (Table A.6), T.Triples', the event-triple coverage, has a significantly significant but unexpectedly negative effect on fault detection. This suggests either that multicolinearity is an issue or that dividing by event-pair coverage is not a very good way to normalize event-triple coverage. T.Len', the length of test cases, is not significant in any models.

A.5 Lessons learned

This experiment was the first attempt to show that simple, automatically-measurable characteristics of faults affect their susceptibility to detection. Furthermore, it was the first proof of concept of the experiment methodology presented in Chapter 3.

Of the test-suite and fault characteristics studied, a fault's estimated probability of detection by statement-coverage-adequate test suites turns out to be the best predictor by far of whether a given test suite will detect a given fault. The results for one of the two applications studied suggest that faults inside methods may be less susceptible to detection, and faults in methods with more branching may be slightly less susceptible to detection.

Unfortunately, the validity of this experiment's results is compromised by many weaknesses in its implementation. Each is resolved in the improved experiment presented in Chapter 4.

A.5.1 Choice of fault and test-suite characteristics

Most of the fault characteristics chosen for this experiment turned out to be problematic. F.StmtDet is difficult to estimate—to do it properly, a new test pool of non-GUI test cases would need to be created—and may not help testers in practice. F.InMethod is problematic because extra-method faults make up a small proportion of faults, driving up the sample size. Additionally, their coverage must be approximated because it is not tracked by the available coverage tools, making them more likely to be associated with false reports of fault detection. F.Branch is a promising start toward measuring the degrees of freedom in execution of faulty event handlers, but better measures could be obtained.

For the test-suite characteristics, the main weakness is the lack of commonly-used coverage metrics like statement coverage. Use of coverage metrics other than event-based measures would make the experiment more applicable to those outside the GUI-testing community.

The experiment in Chapter 4 considers a much larger set of fault and test-suite characteristics. Problematic characteristics like F.StmtDet are omitted, and many interesting characteristics are added. Characteristics are also normalized against measures of the application under test to facilitate comparison between applications.

A.5.2 Sample size

Although an attempt was made to choose an adequate sample size for this experiment, the sample size turned out to be too small. Since the sample size grows with the number of independent variables, a statistician consulted for this work [56] recommends that the study of test-suite and fault characteristics be carried out in two phases. In the first phase, an exploratory study of many independent variables is conducted, using as many data points as can be collected with the resources available. In the second phase, replications of the study, with sample sizes calculated from the exploratory study, examine the small subset of independent variables that turned out to be most important in the exploratory study. The experiment in Chapter 4 implements the first phase; the second phase lies beyond the scope of this dissertation.

A.5.3 Handling of false reports of fault detection

False reports of fault detection are inevitable with the current version of GUITAR, and this experiment took several steps to correct them. For example, reports of fault detection were checked against reports of the statements covered by each length-20 test case in the pool. The experiment in Chapter 4 goes even further, by collecting coverage data after each event in each test case and checking reports of fault detection against that finer-grained data. Also, because accurate coverage data cannot be collected for extra-method faults with the tools available, they are omitted.

A.5.4 Construction of test suites

In this experiment, the logistic-regression models may not have been valid because the test suites in the data points were not independent; they were picked from a relatively small test pool. It turns out that, for this experiment design, it is more efficient anyway to generate a new test suite for each fault. This is what the experiment in Chapter 4 does.

Another problem in this experiment was that many test cases failed to run to completion on the clean version of the applications, for two reasons: timing problems in GUITAR and the approximate nature of the EFG model of the GUI. In this experiment, a test pool with desired properties was generated and then run—and then spuriously-failing test cases were excised from the pool, destroying the pool's desirable properties. In the experiment in Chapter 4, the test cases in each test suite are generated and run one at a time, and only added to the suite if they work on the clean version, until the test suite has the desired properties.

A.5.5 Handling of multicolinearity

Strong correlations among independent variables arose in the data and were not handled during data analysis, leading to multivariate models with extreme and unexpected coefficients. In particular, the test-suite characteristics turned out to be strongly correlated with each other, despite attempts to normalize some of them. In the improved experiment, test suites are constructed more carefully to reduce some of these correlations. The data analysis handles the multicolinearity that does arise in the data.

Appendix B

Sample-size calculation for logistic regression

To calculate the sample size for an experiment, several inputs are required:

- α, the level of significance (usually 0.05),
- β, where $1 - \beta$ is the power (usually 0.80),
- β^*, the effect size, which is the smallest coefficient magnitude of interest, and
- data from a pilot study that used the same independent and dependent variables as the experiment.

The values of α and β are, respectively, the probability of Type I and Type II errors in the experiment's statistical tests. Both errors have to do with "unlucky" samples. A Type I error occurs when there is a statistical relationship between an independent variable and the dependent variable in the sample but not in the general population (i.e., the null hypothesis is spuriously rejected). A Type II error occurs when a statistical relationship of magnitude greater than or equal to the effect size exists in the population but does not show up in the sample (i.e., the null hypothesis should be rejected but is not).

The following formulas are due to Hsieh et al. [30] and were implemented for this work in the R software environment[1]. The sample size n required to statistically

[1] `http://www.cs.umd.edu/~strecker/samplesize/`

test one independent variable, X, in the presence of the other independent variables is

$$n = \frac{n_1}{1 - R^2}$$

where n_1 is the sample size that would be required to test X if it were the only independent.

For a continuous independent variable,

$$n_1 = \frac{(Z_{1-\alpha/2} + Z_{1-\beta})^2}{r_{\bar{X}}(1 - r_{\bar{X}})\beta^{*2}}$$

where Z_u is the uth percentile of the standard normal distribution and $r_{\bar{X}}$ is the probability of the dependent variable at the mean value of X. (In Chapter 4 and Appendix A, the value of $r_{\bar{X}}$ is estimated by fitting a univariate logistic-regression model of the dependent with explanatory variable X to the pilot-study data, then plugging the sample mean of X in for X.)

For a dichotomous (Boolean) independent variable,

$$\begin{aligned} n_1 &= \frac{(Z_{1-\alpha/2}V^{1/2} + Z_{1-\beta}W^{1/2})^2}{(r_0 - r_1)^2(1 - s_1)} \text{ , where} \\ V &= \frac{r(1-r)}{s_1} \\ W &= r_0(1 - r_0) + \frac{r_1(1 - r_1)(1 - s_1)}{s_1} \end{aligned}$$

In these formulas, s_1 is the proportion of the pilot-study data with $X = 1$; r_0 and r_1 are the empirical probabilities of fault detection when $X = 0$ and $X = 1$, respectively; and $r = (1 - s_1)r_0 + s_1 r_1$ is the overall empirical probability of fault detection.

This takes care of n_1, leaving R^2, the squared multiple correlation coefficient relating X to the rest of the independents. In the R environment, R^2 is calculated as a side effect of fitting a linear model, in which X is predicted by the rest of the independents, to the pilot-study data.

When V independent variables are studied, the sample size required for each independent variable can be calculated using the formulas above, except that α should be replaced with α/V [56]. The sample size for the study, then, is the maximum value of n for the independent variables.

Bibliography

[1] *First International Workshop on Testing Techniques and Experimentation Benchmarks for Event-Driven Software* (Apr. 2009).

[2] Agrawal, H., Horgan, J. R., Krauser, E. W., and London, S. Incremental regression testing. In *Proceedings of ICSM '93* (Washington, DC, USA, 1993), IEEE Computer Society, pp. 348–357.

[3] Agresti, A. *An introduction to categorical data analysis*, second ed. John Wiley & Sons, 2007.

[4] Andrews, J. H., Briand, L. C., and Labiche, Y. Is mutation an appropriate tool for testing experiments? In *Proceedings of ICSE '05* (2005), pp. 402–411.

[5] Andrews, J. H., Briand, L. C., Labiche, Y., and Namin, A. S. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng. 32*, 8 (2006), 608–624.

[6] Basili, V. R. Software development: a paradigm for the future. In *Proceedings of COMPSAC '89* (1989), IEEE Computer Society, pp. 471–485.

[7] Basili, V. R., and Perricone, B. T. Software errors and complexity: an empirical investigation. *Commun. ACM 27*, 1 (1984), 42–52.

[8] Basili, V. R., and Selby, R. W. Comparing the effectiveness of software testing strategies. *IEEE Trans. Softw. Eng. 13*, 12 (1987), 1278–1296.

[9] Basili, V. R., Shull, F., and Lanubile, F. Building knowledge through families of experiments. *IEEE Trans. Softw. Eng. 25*, 4 (1999), 456–473.

[10] Briand, L. C., Melo, W. L., and Wst, J. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. Softw. Eng. 28*, 7 (2002), 706–720.

[11] Brooks, F. P. *The Mythical Man-Month: Essays on Software Engineering*, anniversary ed. Addison-Wesley Pub. Co., 1995.

[12] Cai, K.-Y., Li, Y.-C., and Liu, K. Optimal and adaptive testing for software reliability assessment. *Information and Software Technology 46*, 15 (2004), 989–1000.

[13] Clark, B., Devnani-Chulani, S., and Boehm, B. Calibrating the COCOMO II post-architecture model. In *Proceedings of ICSE '98* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 477–480.

[14] Clarke, L. A. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng. 2*, 3 (1976), 215–222.

[15] Davis, M. D., and Weyuker, E. J. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM '81 conference* (New York, NY, USA, 1981), ACM, pp. 254–257.

[16] Do, H., and Rothermel, G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Softw. Eng. 32*, 9 (2006), 733–752.

[17] Eaton, C., and Memon, A. M. An empirical approach to testing web applications across diverse client platform configurations. *International Journal on Web Engineering and Technology, Special Issue on Empirical Studies in Web Engineering 3*, 3 (2007), 227–253.

[18] Elbaum, S., Gable, D., and Rothermel, G. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings of METRICS '01* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 169–179.

[19] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of SOSP '01* (New York, NY, USA, 2001), ACM, pp. 57–72.

[20] Fenton, N. E., and Neil, M. A critique of software defect prediction models. *IEEE Trans. Softw. Eng. 25*, 5 (1999), 675–689.

[21] Ferguson, R., and Korel, B. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol. 5*, 1 (1996), 63–86.

[22] Frankl, P. G., and Weiss, S. N. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of TAV4* (New York, NY, USA, 1991), ACM, pp. 154–164.

[23] Frankl, P. G., and Weyuker, E. J. A formal analysis of the fault-detecting ability of testing methods. *IEEE Trans. Softw. Eng. 19*, 3 (1993), 202–213.

[24] Garfinkel, S. History's worst software bugs. *Wired* (Nov. 2005). Available at `http://www.wired.com/software/coolapps/news/2005/11/69355`.

[25] Garson, G. D. Statnotes: Topics in multivariate analysis, 2006. `http://www2.chass.ncsu.edu/garson/PA765/statnote.htm`.

[26] Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol. 10*, 2 (2001), 184–208.

[27] Gupta, N., Mathur, A. P., and Soffa, M. L. Automated test data generation using an iterative relaxation method. In *Proceedings of SIGSOFT '98 / FSE-6* (New York, NY, USA, 1998), ACM, pp. 231–244.

[28] Harrold, M. J., Gupta, R., and Soffa, M. L. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol. 2*, 3 (1993), 270–285.

[29] Harrold, M. J., Offutt, A. J., and Tewary, K. An approach to fault modeling and fault seeding using the program dependence graph. *J. Syst. Softw. 36*, 3 (1997), 273–295.

[30] Hsieh, F. Y., Bloch, D. A., and Larsen, M. D. A simple method of sample size calculation for linear and logistic regression. *Statistics in Medicine 17*, 14 (1998), 1623–1634.

[31] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria.

In *Proceedings of ICSE '94* (Los Alamitos, CA, USA, 1994), IEEE Computer Society, pp. 191–200.

[32] JURISTO, N., MORENO, A. M., AND VEGAS, S. Reviewing 25 years of testing technique experiments. *Empirical Softw. Eng. 9*, 1–2 (2004), 7–44.

[33] KNUTH, D. E. Frequently asked questions. `http://www-cs-faculty.stanford.edu/~knuth/faq.html`, accessed Jun. 2, 2009.

[34] LENTH, R. V. Two sample-size practices that I don't recommend. In *Proceedings of the Section on Physical and Engineering Sciences* (2000), American Statistical Association.

[35] MCMASTER, S. personal communication, 2008.

[36] MCMASTER, S., AND MEMON, A. Call-stack coverage for GUI test suite reduction. *IEEE Trans. Softw. Eng. 34*, 1 (2008), 99–115.

[37] MEMON, A. M. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability 17*, 3 (2007), 137–157.

[38] MOCKUS, A., AND VOTTA, L. G. Identifying reasons for software changes using historic databases. In *Proceedings of ICSM '00* (Washington, DC, USA, 2000), IEEE Computer Society, p. 120.

[39] MORGAN, J. A., KNAFL, G. J., AND WONG, W. E. Predicting fault detection effectiveness. In *Proceedings of METRICS '97* (Washington, DC, USA, 1997), IEEE Computer Society, pp. 82–89.

[40] Myers, G. J. A controlled experiment in program testing and code walkthroughs/inspections. *Commun. ACM 21*, 9 (1978), 760–768.

[41] Myers, G. J., and Sandler, C. *The Art of Software Testing.* John Wiley & Sons, 2004.

[42] Offutt, A. J. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol. 1*, 1 (1992), 5–20.

[43] Offutt, A. J., and Hayes, J. H. A semantic model of program faults. In *Proceedings of ISSTA '96* (New York, NY, USA, 1996), ACM, pp. 195–200.

[44] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., and Zapf, C. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol. 5*, 2 (1996), 99–118.

[45] Ostrand, T. J., and Balcer, M. J. The category-partition method for specifying and generating fuctional tests. *Commun. ACM 31*, 6 (1988), 676–686.

[46] Ostrand, T. J., Weyuker, E. J., and Bell, R. M. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng. 31*, 4 (2005), 340–355.

[47] Pardoe, I., and Cook, R. D. A graphical method for assessing the fit of a logistic regression model. *The American Statistician 56*, 4 (2002), 263–272.

[48] Porter, A., Memon, A., Yilmaz, C., Schmidt, D. C., and Natarajan, B. Skoll: A process and infrastructure for distributed continuous quality assurance. *IEEE Trans. Softw. Eng. 33*, 8 (2007), 510–525.

[49] Porter, A. A., and Selby, R. W. Empirically guided software development using metric-based classification trees. *IEEE Softw. 7*, 2 (1990), 46–54.

[50] Radatz, J., et al. IEEE standard glossary of software engineering terminology. Std 610.12-1990, Standards Coordinating Committee of the Computer Society of the IEEE, 1990.

[51] Richardson, D. J., and Thompson, M. C. An analysis of test data selection criteria using the RELAY model of fault detection. *IEEE Trans. Softw. Eng. 19*, 6 (1993), 533–553.

[52] Rosner, B. *Fundamentals of Biostatistics*, fifth ed. Duxbury, 2000.

[53] Rothermel, G., Elbaum, S., Malishevsky, A. G., Kallakuri, P., and Qiu, X. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol. 13*, 3 (2004), 277–331.

[54] Rothermel, G., and Harrold, M. J. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol. 6*, 2 (1997), 173–210.

[55] Rothermel, G., Untch, R. J., and Chu, C. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng. 27*, 10 (2001), 929–948.

[56] Slud, E. personal communication, 2008.

[57] Staiger, S. Static analysis of programs with graphical user interface. In *Proceedings of CSMR '07* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 252–264.

[58] Strecker, J. Fault-detection effectiveness and fault detectability in GUI testing. Dissertation proposal, University of Maryland, Apr. 2008.

[59] Strecker, J., and Memon, A. M. Faults' context matters. In *Proceedings of SOQUA '07* (New York, NY, USA, 2007), ACM, pp. 112–115.

[60] Strecker, J., and Memon, A. M. Relationships between test suites, faults, and fault detection in GUI testing. In *Proceedings of ICST '08* (Washington, DC, USA, Apr. 2008), IEEE Computer Society, pp. 12–21.

[61] Strecker, J., and Memon, A. M. Testing graphical user interfaces. In *Encyclopedia of Information Science and Technology*, second ed. IGI Global, 2009.

[62] Tassey, G. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, Prepared by RTI for the National Institute of Standards and Technology (NIST), 2002. Available at `http://www.nist.gov/director/prog-ofc/report02-3.pdf`.

[63] Tian, J. *Software quality engineering: Testing, quality assurance, and quantifiable improvement.* Wiley-Interscience, 2005.

[64] Voas, J. M. PIE: A dynamic failure-based technique. *IEEE Trans. Softw. Eng. 18*, 8 (1992), 717–727.

[65] XIE, Q., AND MEMON, A. Studying the characteristics of a "good" GUI test suite. In *Proceedings of ISSRE '06* (Los Alamitos, CA, USA, 2006), IEEE Computer Society, pp. 159–168.

[66] XIE, Q., AND MEMON, A. M. Rapid "crash testing" for continuously evolving GUI-based software applications. In *Proceedings of ICSM '05* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 473–482.

[67] YUAN, X., AND MEMON, A. M. Using GUI run-time state as feedback to generate test cases. In *Proceedings ICSE '07* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 396–405.

[68] ZELKOWITZ, M. V., AND WALLACE, D. Experimental validation in software engineering. *Information and Software Technology 39*, 11 (1997), 735–743.