Predicting Test Suite Effectiveness for Java Programs

by

Laura Inozemtseva

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The coverage of a test suite is often used as a proxy for its effectiveness. However, previous studies that investigated the influence of code coverage on test suite effectiveness have failed to reach a consensus about the nature and strength of the relationship between these test suite characteristics. Moreover, many of the studies were done with small or synthetic programs, making it unclear that their results generalize to larger programs. In addition, some of the studies did not account for the confounding influence of test suite size. We have extended these studies by evaluating the relationship between test suite size, block coverage, and effectiveness for large Java programs.

Our test subjects were four Java programs from different application domains: Apache POI, HSQLDB, JFreeChart, and Joda Time. All four are actively developed open source programs; they range from 80,000 to 284,000 source lines of code. For each test subject, we generated between 5,000 and 7,000 test suites by randomly selecting test methods from the program's entire test suite. The suites ranged in size from 3 to 3,000 methods. We used the coverage tool Emma to measure the block coverage of each suite and the mutation testing tool Javalanche to evaluate the effectiveness of each suite.

We found that there is a low correlation between block coverage and effectiveness when the number of tests in the suite is controlled for. This suggests that block coverage, while useful for identifying under-tested parts of a program, should not be used as a quality target because it is not a good indicator of test suite effectiveness.

Acknowledgements

I would like to thank my supervisor Reid Holmes for accepting my transfer to the software architecture group and for his support and guidance over the last year. I particularly appreciate being able to work from a distance while my husband finishes his degree. I would also like to thank Mike Godfrey and Ondřej Lhoták, two of my course professors. I learned a lot from both of them and appreciate the many reference letters they wrote on my behalf. My thesis readers, Ondřej and Patrick Lam, have my gratitude for their helpful comments. Finally, I would like to thank my parents and my husband for their encouragement and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Testing is an important part of producing high quality software, but its effectiveness depends on the quality of the test suite: some suites are better at detecting faults than others. Naturally, developers want their test suites to be good at detecting faults, necessitating a method for measuring the fault detection effectiveness of a test suite. Testing textbooks often recommend coverage as one of the metrics that can be used for this purpose (e.g., [25, 33]). This is intuitively appealing, since it is clear that a test suite cannot find bugs in code it never executes; it is also supported by many studies that have found a relationship between code coverage and fault detection effectiveness [3, 5, 12, 13, 14, 15, 22, 28, 36].

Unfortunately, these studies do not agree on the strength of the relationship between these test suite characteristics. In addition, two issues with the studies make it difficult to generalize their results. First, some of the studies did not control for the size of the suite. Since coverage is increased by adding code to existing tests or by adding new tests to the suite, the coverage of a test suite is correlated with its size. It is therefore not clear that coverage influences effectiveness independently of the number of tests in the suite. Second, the studies were performed with small or synthetic programs, making it unclear that their results hold for the large programs typical of industry.

This thesis explores the relationship between test suite size, coverage and effectiveness for large Java programs. Specifically, we address the following three research questions:

Research Question 1 *For large Java programs, is the effectiveness of a test suite correlated with the number of methods in the suite?*

Research Question 2 *For large Java programs, is the effectiveness of a test suite correlated with its block coverage when the number of methods in the suite is ignored?*

Research Question 3 *For large Java programs, is the effectiveness of a test suite correlated with its block coverage when the number of methods in the suite is held constant?*

Effectiveness refers to a suite's ability to expose faults in a program. It will be defined precisely when we explain our methodology in Chapter 3.

Chapter 2

Background

This chapter describes the previous work that has been done in this area. However, before doing so, we briefly define the coverage measurements used in these studies. Formal definitions, when not given here, can be found in Rapps and Weyuker [34]. We also describe several ideas related to mutation testing. For more details about this topic, see the survey papers by Jia and Harman [23] and Offutt and Untch [31].

2.1 Definitions

The first two types of coverage we define are based on the control flow of a program. To describe them, we need the concept of a *basic block*. Informally, a basic block is a group of statements that are always executed together and in order. Formally, a basic block is a maximal set of statements ordered from 1 to n such that, if $n > 1$, for $i = 2 \ldots n$, the statement s_i is the unique executional successor of statement s_{i-1} and the statement s_{i-1} is the unique executional predecessor of the statement s_i.

Any program can be represented as a graph whose nodes correspond to basic blocks and whose edges correspond to the control flow decisions (branches and jumps) that connect those blocks. A test suite's *block coverage*, sometimes called *node coverage*, is the fraction of the basic blocks in the associated program that are executed by the test suite. *Decision coverage*, also known as *branch coverage* or *edge coverage*, is the fraction of decisions in a program that are executed by its test suite.

The remaining types of coverage that we define are based on the data flow of a program. To describe them, we need a few additional concepts. A *definition* is an assignment to a variable, for example $i = 1$. A *use* is when a variable is read and its value is used in an expression, for example in the predicate $i == 1$. Uses can be divided into two types: when a value is used in a computation, it is referred to as a *c-use*; when a value is used in a predicate, it is referred to as a *p-use*. An *all-use* is either a c-use or a p-use; in other words, "all-use" is synonymous with "use".

Suppose we have one or more paths in a program graph that start at some node a, end at some node b, and meet the following conditions: a variable x is assigned to in node a, its value is used in node b, and it is not redefined between nodes a and b. Each of these paths is called a *def-use path* or *du-path*. The definition at a and the use at b are referred to as a *def-use pair* or *du-pair*.

There are two ways to define the coverage of a du-pair. The first option is to say that the du-pair is covered if the test suite executes at least one of the du-paths between a and b. The second option is to say that the du-pair is covered only if the test suite executes every du-path between a and b. *C-use coverage*, *p-use coverage* and *all-use coverage* use the first option; in addition, c-use coverage only considers du-pairs where the use is a c-use, and p-use coverage only considers du-pairs where the use is a p-use. In other words, to have 100% all-use coverage, for every du-pair in the program, the test suite must execute at least one of the du-paths between the definition and the use. *Def-use* or *DU coverage* uses the second option; that is, it is the fraction of du-paths in a program that are executed by its test suite.

A test suite is called ξ *adequate* if it has complete coverage according to criterion ξ. For example, a suite is decision coverage adequate if it covers 100% of the decisions in the program.

It is possible to organize these definitions into a hierarchy [21, 34] as shown in Figure 2.1. In the figure, a coverage criterion ξ_1 subsumes a criterion ξ_2 if there is an arrow pointing from ξ_1 to ξ_2. Generally speaking, the criteria at the top of the hierarchy are more difficult to satisfy. However, this does not imply anything about the relative quality of test suites that are constructed using different coverage criteria [17, 18]. For example, a suite with 50% c-use coverage is not guaranteed to be more effective than a suite with 50% block coverage.

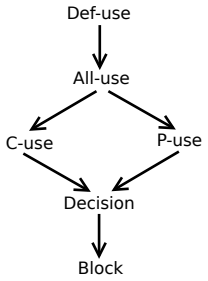


Figure 2.1: Subsumption relationships between different types of coverage. Criterion ξ_1 subsumes criterion ξ_2 if there is an arrow pointing from ξ_1 to ξ_2.

A *mutant* is a new version of a program that is created by making a small syntactic change to the original program. For example, a mutant could be created by modifying a constant, negating a branch condition, or removing a method call. The resulting mutant may produce the same output as the original program, in which case it is called an *equivalent mutant*. An example of this is shown in Listing 2.1. A program's *mutant coverage* is the fraction of non-equivalent mutants that can be detected (or *killed*) by a test suite. Equivalent mutants are excluded because they cannot, by definition, be detected by a unit test.

2.2 Related Work

Having defined the types of coverage that were used in previous studies, we can now discuss their results. All of the studies we will discuss explored the relationship between test suite size, coverage and effectiveness and, with one exception, used the following procedure to do so:

```
int index = 0;
while (true) {
    index++;
    if (index == 10) {
        break;
    }
}
```

Listing 2.1: An example of how equivalent mutants can be produced. If the equality test in this code were changed to `if (index >= 10)`, the new program would be an equivalent mutant.

1. Created faulty versions of the test program(s) by manually seeding faults, reintroducing previously fixed faults, or using a mutation tool.
2. Created a large number of test suites by selecting from a pool of available tests, either randomly or according to some algorithm, until the suite reached either a pre-specified size or a pre-specified coverage level.
3. Measured the coverage of each suite in one or more ways, if suite size was fixed; measured the suite's size if its coverage was fixed.
4. Determined the effectiveness of each suite as the fraction of faulty versions of the program that were detected by the suite.

To use this procedure, the authors of each paper had to make a number of methodological choices. We will discuss their choices as we describe each study.

The earliest work on this topic was done by Frankl and Weiss [13, 14], who wanted to determine the relative effectiveness of all-use adequate suites and decision adequate suites. As a control, the authors generated random suites that contained approximately the same number of test cases as the adequate suites. As test subjects, the authors used nine Pascal programs, ranging from 22 to 78 SLOC[1] in length, that had naturally occurring faults. Suite coverage was measured with the ASSET tool [16]. The authors found that all-use adequate suites were often more effective than decision adequate suites, and decision adequate suites were often more effective than suites without a coverage criterion. However, these results did not control for the effect of test suite size. The authors therefore grouped the suites by size to compare the effectiveness of suites of similar size that were constructed in different ways. When size was fixed, they found that all-use adequate suites were still more effective than decision adequate suites; however, they found that decision adequate suites were no more effective than random suites. This suggests that all-use coverage influences effectiveness independently of size but decision coverage does not. Finally, the authors tried measuring the effectiveness of test suites with less than 100% coverage. They found that coverage was somewhat correlated with effectiveness for both coverage types but that the correlation was not particularly strong and the relationship was highly non-linear. This indicates that, even when coverage does independently influence effectiveness, the quality of a test suite should not be assumed to be proportional to its level of coverage.

[1] In this thesis, source lines of code (SLOC) refers only to executable lines of code, while lines of code (LOC) includes whitespace and comments.

Frankl et al. [15] later extended this study to mutation adequate suites, comparing them with all-use adequate suites and random suites. The test subjects were the same nine Pascal programs used in the earlier study. However, the mutation tool the authors used, Mothra [8], only works with Fortran programs, so the nine subjects were also ported to Fortran. Mutants were generated from the Fortran programs with Mothra; all-use coverage was measured on the Pascal programs with ASSET. The authors found no significant difference between mutation-adequate suites and all-use adequate suites: both were more effective for some of the programs, and for two of the programs there was no significant difference between the two criteria. However, the authors reported that mutation-adequate suites were much more difficult to make. Following this, the authors tried to isolate the effect of size by generating suites with a fixed number of test cases. They found that, while effectiveness improved with coverage, it generally did not improve until very high levels of coverage were reached (greater than 80%), and even then effectiveness did not increase by much.

Frankl and Iakounenko [12] later extended Frankl and Weiss's work [13, 14] to a C program with 11,640 LOC. Faulty versions of the program were created by reintroducing faults that had been previously found and fixed. Test sets of various sizes were created by randomly selecting tests from a large test pool. The ATAC tool [26], which accounts for the use of pointers in C, was used to measure the suites' all-use and decision coverage. The authors found that, when the number of test cases in the suite was fixed, effectiveness was correlated with both decision coverage and all-use coverage. This contradicts their earlier result that decision coverage does not independently influence effectiveness; however, they again found that high levels of coverage were needed to see a large increase in effectiveness.

Wong et al. [36] also studied this topic, evaluating ten C programs with a total of 2,310 SLOC. Test suites were generated by random selection from a pool; their block coverage was measured with ATAC. Faulty versions of the test subjects were made by asking graduate students to manually inject faults. The authors found that effectiveness was more highly correlated with block coverage than with the number of test cases in the suite; in other words, knowing the block coverage of a suite tells you more about its effectiveness than knowing its size does.

Hutchins et al. [22] studied this question using seven C programs ranging from 141 to 512 SLOC. Faulty versions were generated by asking ten experienced programmers to manually seed faults. The authors used two different coverage measurements: decision coverage and a slight variant of all-use coverage that accounts for the use of pointers in C. These were both measured with the Tactic tool [32]. The authors found that effectiveness does rise with increased coverage, but that high coverage does not guarantee a high level of effectiveness. They suggest instead that low coverage should be taken as a sign that the test suite is inadequate, even if it seems comprehensive. They also found that effectiveness increased more rapidly when coverage was already at a very high level; in other words, increasing the coverage of the test suite from 40% to 50% had less impact than increasing it from 90% to 100%. While this paper did not focus on the influence of suite size, the authors did note that suites with high coverage tend to be large and that this may be a confounding factor. They therefore took test sets with coverage in the 90 to 100% range, generated random suites that contained the same number of test cases, and compared the fault detection ability of these two types of suites. The high coverage suites were quite a bit more effective, suggesting that decision coverage and all-use coverage both influence effectiveness independently of size.

Cai and Lyu [5] studied this question using 21 C programs that ranged from from 1,455 to 4,512 SLOC. These programs were written by fourth year university students as part of a software engineering course. Faulty versions of the programs were made by reintroducing faults that had been found and fixed during development. Before a program was accepted at the end of the course, it had to pass 1,200 tests, 800

of which were designed using the program's operational specification and 400 of which were randomly generated. These tests were reused on the mutated versions of the programs during the study. The authors used ATAC to measure block, decision, p-use and c-use coverage. They found that effectiveness was moderately correlated with all four types of coverage; however, they found that the magnitude of the correlation depended on the nature of the test. Specifically, the authors found that the correlation between coverage and effectiveness was somewhat higher for the 800 operational tests than it was for the 400 random tests, though the difference was not statistically significant. In addition, the correlation was quite a bit stronger for tests designed to test erroneous situations than for those designed to test normal operation.

A later comprehensive study by Andrews et al. [3] addressed a number of questions related to mutation testing. They studied a C program with approximately 5,000 SLOC and used ATAC to measure four different kinds of coverage: c-use, p-use, decision and block. Test suites were generated in two different ways. One set of suites was generated by randomly selecting tests from a pool. The other set was also created by random selection from a pool, but with the added condition that every test increase the overall coverage. In other words, if a randomly selected test did not increase the suite's coverage, it was not added; a different test was selected instead. Plotting effectiveness against the number of test cases in the suite for both types of suites and all four coverage measures showed that suites built to maximize coverage were more effective. The authors confirmed this result by running a regression analysis with size and coverage as covariates, revealing that both were statistically significant and showed a positive regression coefficient. This suggests that coverage influences effectiveness independently of size. Interestingly, in sharp contrast to the results described earlier, the authors found that effectiveness rose steadily with coverage, meaning that it may be valid to assume that effectiveness is proportional to coverage.

A recent study by Namin and Andrews [28] worked with the Siemens suite of seven C programs, which range from 137 to 513 SLOC. The suite includes a comprehensive test pool for each program, so the authors generated test suites by randomly selecting tests from this pool. The authors used Proteum [7] to generate mutants and ATAC to measure four different types of coverage: block, decision, c-use and p-use. They found that, for all coverage types, both coverage and the number of tests in the suite independently influenced effectiveness, but the correlations were not always very strong. Using linear regression, they found that the best predictor of effectiveness was a combined measurement: log(size) + coverage. The authors then attempted to replicate their findings on two additional test programs, one in C with 5,680 SLOC and one in C++ with 966 SLOC. They found that their results held for these programs, suggesting that their conclusions may generalize to slightly larger programs and other programming languages.

A study by Briand and Pfhal [4] was the only experiment that did not use the procedure outlined earlier. Instead, they proposed a general statistical method to determine the relationship between coverage, size and effectiveness. Specifically, they suggested using Monte Carlo simulation to estimate the effectiveness of a test suite, but without using any information about the coverage of the suite. This estimate is then compared to the actual effectiveness. If the difference between the two is statistically significant, then coverage has influenced effectiveness. The authors tested this method on data from an earlier study [21] that used twelve versions of a C program ranging from 900 to 4,000 SLOC. This study measured decision, block, c-use and p-use coverage with ATAC. Using their statistical method, Briand and Pfhal found that none of the four coverage measures influenced the effectiveness of the test suites independently of the number of test cases in the suite. This paper is particularly relevant to our study because we reached the same conclusion about block coverage using a different methodology.

To summarize, we have described nine studies that considered the relationship between the coverage and the effectiveness of a test suite. Eight of them found that at least one type of coverage has at least some influence on effectiveness independently of size. However, not all studies found a strong correlation, and most found that the relationship was highly non-linear. In addition, some found that the relationship only appeared at very high levels of coverage. We have summarized the results of all nine studies in Table 2.1. As can be seen from the table, the largest program used in any of the studies was an 11,640 LOC C program [12]. All of the studies used Pascal, C or Fortran, with one exception: [28] used a 966 SLOC C++ program.

As the above discussion shows, it is still not clear how test suite size, coverage and effectiveness are related. Most studies conclude that effectiveness depends on coverage, but there is little agreement about the strength of the relationship.

Citation	Languages	Largest Program	Coverage Types	Findings
[13, 14]	Pascal	78 SLOC	All-use, decision	All-use influences effectiveness independently of size; decision does not; relationship is highly non-linear
[15]	Fortran Pascal	78 SLOC	All-use, mutation	Effectiveness improves with coverage but not until coverage reaches 80%; even then increase is small
[12]	C	11,640 LOC	All-use, decision	Effectiveness is correlated with both all-use and decision coverage; increase is small until high levels of coverage are reached
[36]	C	<2,310 SLOC	Block	Effectiveness is more highly correlated with block coverage than with size
[22]	C	512 SLOC	All-use, decision	Effectiveness is correlated with both all-use and decision coverage; effectiveness increases more rapidly at high levels of coverage
[5]	C	4,512 SLOC	Block, c-use, decision, p-use	Effectiveness is moderately correlated with all four coverage types; magnitude of the correlation depends on the nature of the tests
[3]	C	5,000 SLOC	Block, c-use, decision, p-use	Effectiveness is correlated with all four coverage types; effectiveness rises steadily with coverage
[28]	C C++	5,680	Block, c-use, decision, p-use	Effectiveness is correlated with all four coverage types but the correlations are not always strong
[4]	C	4,000 SLOC	Block, c-use, decision, p-use	None of the four coverage types influence effectiveness independently of size

Table 2.1: Summary of the findings from previous studies.

Chapter 3

Methodology

The goal of our study was to answer the research questions posed in Chapter 1. To do this, we followed the general procedure outlined in Section 2.2, which required us to select test subjects, a method of generating faulty versions of the test subjects, a method of generating test suites, a coverage metric, and an effectiveness metric. In this chapter, we describe and explain the decisions we made.

3.1 Test Subjects

We used four test subjects for this study from a variety of application domains. The first, Apache POI[1], is an open source API for Microsoft documents. The second, HSQLDB[2], is an open source relational database management system. The third, JFreeChart[3], is an open source library for producing charts. The fourth, Joda Time[4], is an open source replacement for the Java `date` and `time` classes.

We used a number of criteria to select these projects. First, to help ensure the novelty and generalizability of our study, we required that the projects be reasonably large (on the order of 100,000 SLOC), written in Java, and actively developed. We also required that the projects have a fairly large number of test methods (on the order of 1,000) so that we would be able to generate reasonably sized random test suites. Finally, we required that the projects use Ant as a build system and JUnit 3 as a test harness. This allowed us to automate data collection, and since Ant and JUnit are both popular tools, these requirements were not difficult to meet.

The salient characteristics of our test programs are summarized in Table 3.1. The first row gives the approximate size of each program in SLOC, while the second row indicates how many of those lines are part of the program's test suite. These properties were measured with `SLOCCount`[5]. The third row indicates how many test methods the program has. The fourth row indicates the block coverage of the program's

[1] `http://poi.apache.org`
[2] `http://hsqldb.org`
[3] `http://jfree.org/jfreechart`
[4] `http://joda-time.sourceforge.net`
[5] `http://dwheeler.com/sloccount`

entire test suite as measured with the tool Emma. The remaining rows provide information related to mutation testing and will be explained in Section 3.2.

Property	Apache POI	HSQLDB	JFreeChart	Joda Time
Java SLOC (total)	283,845	178,018	125,659	80,462
Java SLOC (tests only)	68,932	18,425	44,297	51,444
Number of test methods	1,415	643	1,769	3,857
Block coverage (%)	73	37	57	89
Number of mutations	70,471	30,893	66,312	23,117
Equivalent mutants	30,145	29,290	50,748	7,090
Equivalent mutants (%)	43	95	77	31

Table 3.1: Salient characteristics of our subject programs.

3.2 Generating Faulty Programs

We created faulty versions of our subject programs with the open-source tool Javalanche [35]. Javalanche is a mutation testing tool that repeatedly modifies a Java program at the bytecode level and runs a user-specified test suite to see if the mutation can be detected. By default, Javalanche produces a boolean result: either the mutant can or cannot be detected by the suite. Since we wanted a list of every test case that can detect a given mutant, rather than a boolean result, we modified the tool to make it run the entire test suite for each mutant and report which test cases could kill it. The modifications to the source code required to achieve this are described in Appendix A.

Making this change also allowed us to make our study more efficient. Javalanche is quite slow, and if we had tried to run it for each of the 24,000 random test suites, the study would have been prohibitively time consuming. Instead, we ran Javalanche once with the program's entire test suite, producing for every mutant a list of the tests that covered it and a list of the tests that killed it. We stored these results in a database using the test's full name[6] as the key. When we generated a test suite, we determined which mutants it covered and which it killed by referring to the database entries for the test cases in the suite. Using this procedure will give the same results as running Javalanche if all of the tests in the suite are deterministic. To the best of our knowledge, the tests for our subject programs are; however, even if the suites contain non-deterministic tests, our results still make sense, since we have simply converted any non-deterministic tests to deterministic ones by "memorizing" the results of one run.

3.3 Generating Test Suites

For each subject program, we used Java's reflection API to identify all of the no-argument test methods in the program's suite. We then generated new test suites of fixed size by randomly selecting a subset of

[6]For example, `org.joda.time.convert.TestCalendarConverter.testGetPartialValues`. These are unique since we only consider no-argument test methods.

these methods without replacement. We made 1,000 suites of each of the following sizes: 3 methods, 10 methods, 30 methods, 100 methods, and so on, up to the largest number following this pattern that was less than the total number of test methods. Each suite was created as a JUnit suite so that the necessary set-up and tear-down code was run for each test method. Given this procedure for creating suites, in this thesis the size of our suites should always be understood as the number of methods they contain.

3.4 Measuring Coverage

A number of different coverage measurements exist, as discussed in Section 2.1. We used block coverage in this study for several reasons. First, very few coverage tools for Java can measure data flow coverage. One exception is Coverlipse[7], which can measure all-use coverage, but can only be used as an Eclipse plugin. To the best of our knowledge, there are no open-source coverage tools for Java that can measure other data flow coverage criteria or that can be used from the command line. Since developers use the tools they have, many only measure the block or statement coverage of their test suites. Using the same measurement means that our results will more accurately reflect current development practice.

Second, since block coverage is one of the weaker types of coverage, in that it is subsumed by other types, if it influences effectiveness independently of size, then stronger coverage types will almost certainly do the same. Of course, by the same reasoning, a negative result for block coverage does not mean that other types of coverage do not tell us anything about the quality of a test suite.

We measured the block coverage of each test suite with the open-source tool Emma[8]. We used Emma's exclusion patterns to exclude Emma's own classes and the test classes themselves from the measurement.

3.5 Measuring Effectiveness

Before we can give a precise definition of effectiveness, we must address one issue. Recall that *equivalent mutants* alter the syntax of a program but do not change its output and so cannot be killed by a unit test. If a mutant is not killed by a test suite, manual inspection is required to determine if it is equivalent or if it was simply missed by the suite[9]. Since this is time consuming and expensive, a possible solution is to assume that any mutant that cannot be detected by the test subject's entire test suite is an equivalent mutant. This technique tends to overestimate the number of equivalent mutants; despite this, it is common because it allows us to study much larger programs. Of the three studies in Section 2.2 that used mutation, two of them used this approach [3, 28]; in order to study large projects we have done the same.

Having decided how to handle equivalent mutants, we define a test suite's effectiveness or *kill score* as follows: a test suite's effectiveness is the number of mutants it detected divided by the number of non-equivalent mutants that are covered by the suite. Note that the number of non-equivalent mutants covered by the suite is the maximum number of mutants it could possibly detect, so our effectiveness value ranges from 0 to 1. Note also that the master test suite has an effectiveness score of 1, since we decided that any mutants it did not kill are equivalent.

[7] `http://coverlipse.sourceforge.net/`

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

[9] Manual inspection is required because automatically determining whether or not a mutant is equivalent is undecidable [30].

Alternatively, we could measure the effectiveness of a suite as the number of mutants it detected divided by the number of non-equivalent mutants in the entire program. Unfortunately, this measurement does not allow us to draw meaningful conclusions. Suppose we are comparing suite A, with 50% coverage, to suite B, with 60% coverage. If we use the alternative definition, suite B will almost certainly have a higher effectiveness score, since it covers more code and will therefore almost certainly kill more mutants. Thus, our study would reach the trivial conclusion that suites with higher coverage kill more mutants. Our original definition, which uses the number of covered mutants in the denominator, puts the test suites on an even footing. That is, if suite A kills 80% of the mutants that it covers, while suite B kills only 70% of the mutants that it covers, suite A will have a higher effectiveness score even though it runs less code. Our definition thus captures the idea that an effective suite is very good at finding faults in the code that it runs. It also allows us to draw more interesting conclusions about the relationship between coverage and effectiveness.

We can now describe the remaining rows of Table 3.1. The fifth row shows how many mutants Javalanche generated for each project. The sixth row shows how many of those mutants could not be detected by the entire test suite and were therefore assumed to be equivalent. The last row gives the equivalent mutants as a percentage of the total number of mutants.

3.5.1 Generalizability of the Effectiveness Measurement

Every mutant that Javalanche generates simulates a real fault, so we have used the number of mutants a suite can detect as a measurement of its fault detection effectiveness. This measurement has been shown to be highly representative of the suite's ability to detect real faults [2, 3, 6]. Moreover, it has been shown that if a test suite detects a large number of simple faults, caused by a single incorrect line of source code, it will also detect a large number of harder, multi-line faults [29, 24]. This implies that if a test suite can kill a large proportion of mutants, it can also detect a large proportion of the more difficult faults in the software. The mutant detection rate of a suite has thus been established as a valid measurement of its effectiveness.

Chapter 4

Results

In this chapter, we answer the three research questions posed in Chapter 1. As Chapter 3 explained, we collected the data to answer these questions by generating test suites of fixed size, measuring their coverage with Emma, and measuring their effectiveness with Javalanche.

4.1 Does Size Influence Effectiveness?

Research question 1 asked if the effectiveness of a test suite is influenced by the number of methods it contains. Figure 4.1 shows the data we collected to answer this question. In each subfigure, the x axis indicates suite size on a log scale while the y axis shows the range of effectiveness values we computed. The red line on each plot was fit to the data with linear regression. Specifically, we modelled effectiveness with the following equation:

$$e = a_0 + a_1 \ln s + a_2 \sqrt{s} + a_3 s$$

where e is a suite's effectiveness, s is its size, and a_0 through a_3 are constants. We did not include super-linear terms in our model since it seems unlikely that effectiveness increases with the square or higher power of size. Since we used machine learning to create the models, and it is difficult to explain our methodology without assuming some knowledge of this topic, we leave the details to Appendix B.

The presence of the constant term a_0 may be counterintuitive, since a suite of size zero should have zero effectiveness, but it is necessary due to our definition of effectiveness. Recall that we measured effectiveness as the number of mutants a suite kills divided by the number of non-equivalent mutants it covers, meaning that even suites with one method can have very high effectiveness scores. Forcing a zero intercept resulted in a curve that fit the data poorly because it consistently underestimated the effectiveness of small suites. The non-zero intercept therefore tells us something about the data: the typical effectiveness of a single test method. Since the intercept provides useful information, and the logarithmic term makes the function undefined for suites of size zero anyway, we elected to keep the non-zero intercept.

The r^2 value for each regression line is shown in the bottom right corner of each plot. These values range from 0.66 to 0.86, implying that the correlation coefficient r ranges from 0.81 to 0.93. This indicates

that there is a high to very high correlation between effectiveness and size for these projects[1].

Answer 1 *Our results suggest that, for large Java programs, there is a high to very high correlation between the effectiveness and the size of a test suite.*

4.2 Does Coverage Influence Effectiveness?

Research question 2 asked if the effectiveness of a test suite is correlated with the block coverage of the suite when we ignore the influence of suite size. Figure 4.2 shows the data we collected to answer this question. Each panel shows the results we obtained for one project and one suite size. The project name is given at the top of each column, while the suite size is given at the right of each row. The bottom row is a margin plot that shows the results for all sizes, while the rightmost column is a margin plot that shows the results for all projects. The bottom row of plots suggests that, when size is ignored, effectiveness improves with coverage, but the strength of the relationship varies by project. We verified this by measuring the Kendall τ correlation coefficient between coverage and effectiveness for each project. Kendall's τ is similar to the more common Pearson coefficient but does not assume that the variables are linearly related or that they are normally distributed. Rather, it measures how well an arbitrary monotonic function could fit the data. We used it instead of the Pearson coefficient to avoid introducing unnecessary assumptions. The results are shown in Table 4.1. The second column gives the correlation coefficient and the third column gives the statistical significance level of the coefficient. The fourth column relates to the validity of the results and will be explained further in Chapter 6.

Answer 2 *Our results suggest that, for large Java programs, there is a moderate to high correlation between the effectiveness and the block coverage of a test suite when the influence of suite size is ignored.*

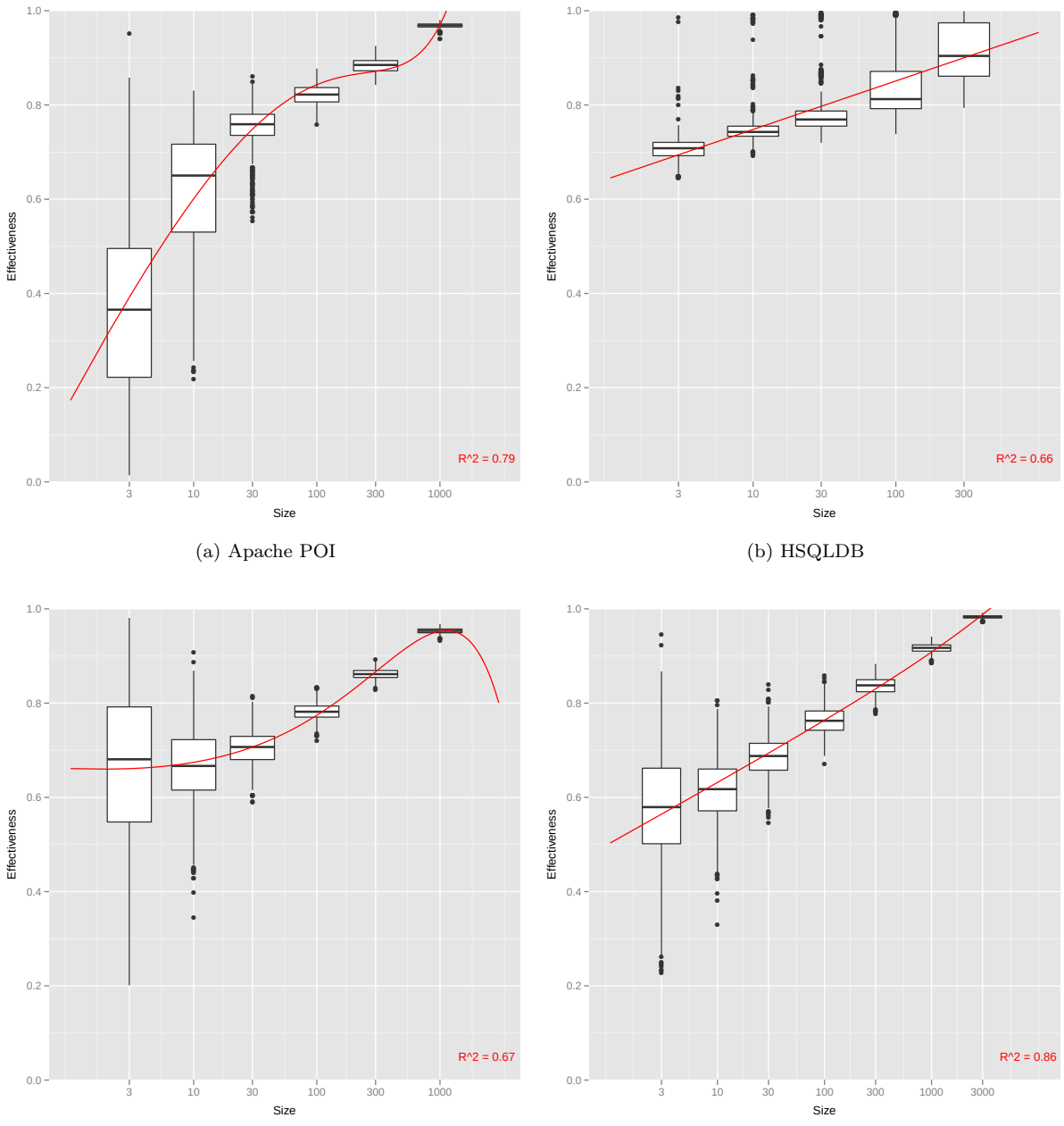
Project	Correlation	Significance	Ties (%)
Apache POI	0.81	99.9	0.00001
HSQLDB	0.68	99.9	0.059
JFreeChart	0.61	99.9	0.0006
Joda Time	0.79	99.9	0.001

Table 4.1: The Kendall τ correlation between coverage and effectiveness when suite size is ignored.

4.3 Does Coverage Influence Effectiveness When Size is Fixed?

Research question 3 asked if the effectiveness of a test suite is correlated with its block coverage when we control for the number of tests in the suite. We answered this question in two ways. First, we computed

[1]Here we use the Guildford scale [20] for verbal description, in which correlations with absolute value less than 0.4 are described as "low", 0.4 to 0.7 as "moderate", 0.7 to 0.9 as "high", and over 0.9 as "very high".



(a) Apache POI

(b) HSQLDB

(c) JFreeChart

(d) Joda Time

Figure 4.1: Effectiveness scores plotted against size for all test subjects. Size is on a logarithmic axis.

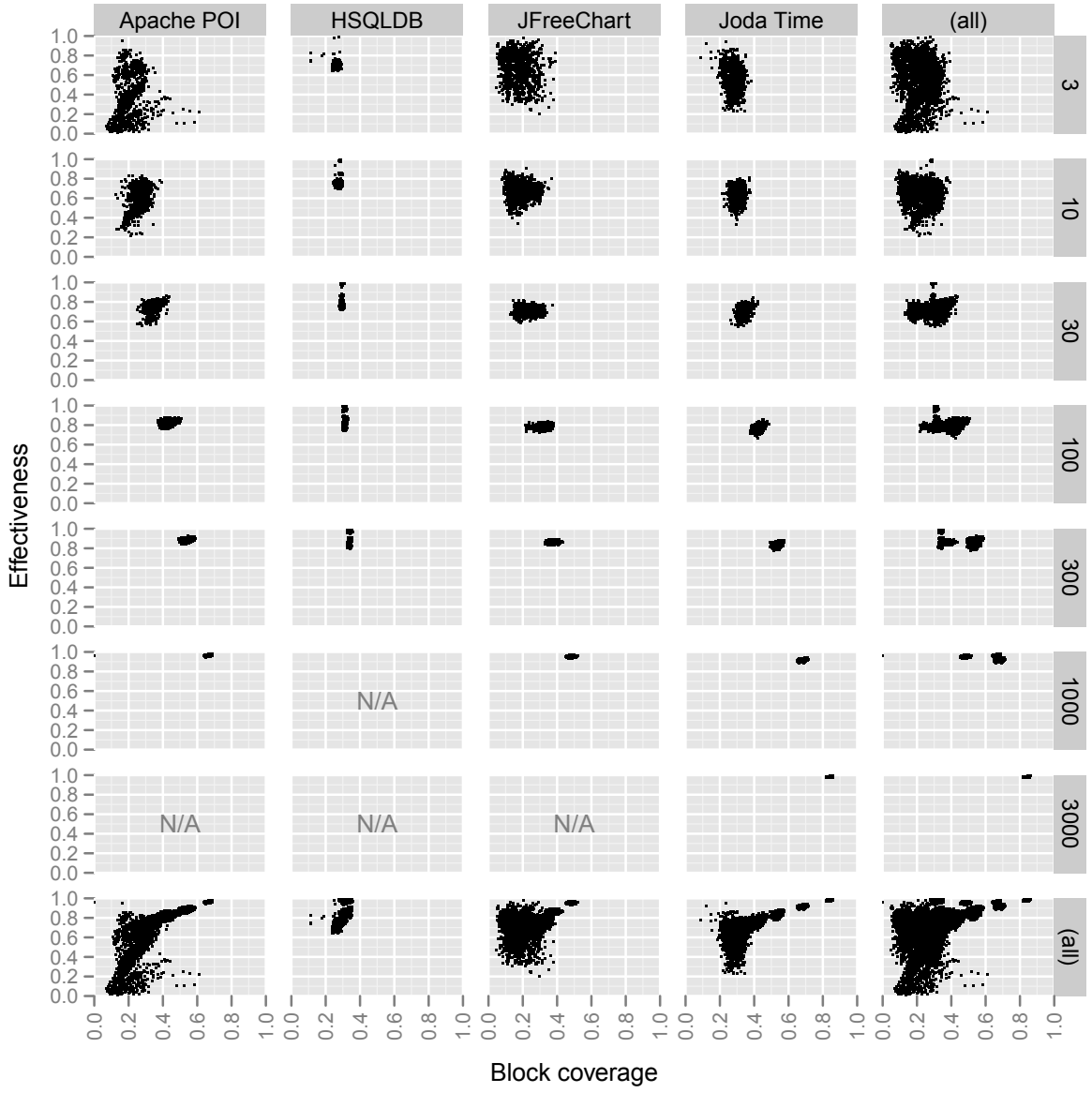


Figure 4.2: Effectiveness scores plotted against coverage for all test subjects. Each row shows the results for one suite size; each column shows the results for one project.

the Kendall τ correlation coefficient between coverage and effectiveness for each project and each suite size. In other words, we computed a correlation value for each of the non-margin plots in Figure 4.2. Next, we used linear regression to model the relationship between size, coverage and effectiveness. We discuss

these two analyses in turn.

4.3.1 Kendall Tau Correlation

The coefficients that we computed are shown in Table 4.2. The first number in each cell is the correlation coefficient; the second number is the significance level of the coefficient. For most projects and suite sizes, we found a statistically significant low correlation between block coverage and effectiveness when size is fixed. A notable exception is the size 1,000 suites for Apache POI, which showed a moderate correlation of 0.42 that is significant at the 99.9% level. Table 4.3 relates to the validity of the results in Table 4.2 and will be explained further in Chapter 6.

Suite Size	Apache POI	HSQLDB	JFreeChart	Joda Time
3	0.23/99.9	0.15/99.9	−0.10/99.9	−0.11/99.9
10	0.23/99.9	−0.01/<95	−0.01/<95	0.07/99.9
30	0.10/99.9	−0.03/<95	0.01/<95	0.16/99.9
100	0.09/99.9	0.10/99.9	0.10/99.9	0.20/99.9
300	0.29/99.9	0.06/99	0.05/99	0.29/99.9
1,000	0.42/99.9	—	0.06/99	0.31/99.9
3,000	—	—	—	0.21/99.9

Table 4.2: The Kendall τ correlation between coverage and effectiveness when suite size is fixed.

Suite Size	Apache POI	HSQLDB	JFreeChart	Joda Time
3	0.0000	0.69	0.0064	0.0096
10	0.0000	0.35	0.0014	0.0008
30	0.0000	0.08	0.0006	0.0004
100	0.0002	0.02	0.0004	0.0002
300	0.0002	0.01	0.0004	0.0002
1,000	0.0000	—	0.0008	0.0050
3,000	—	—	—	0.0290

Table 4.3: The percentage of comparisons that resulted in a tie when computing the Kendall τ coefficients in Table 4.2.

4.3.2 Linear Regression

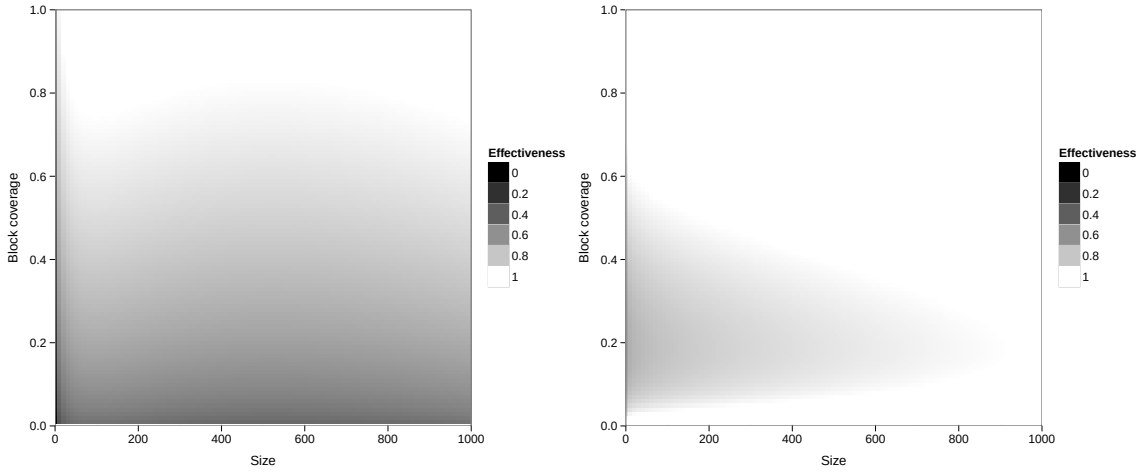
After computing the correlation coefficients, we verified our results by using linear regression to model the relationship between size, coverage and effectiveness. Specifically, we modelled effectiveness with the following equation, where e is a suite's effectiveness, s is its size, c is its coverage, and the a's are constants:

$$e = a_0 \ln s + a_1 \sqrt{s} + a_2 s + a_3 \ln c + a_4 \sqrt{c} + a_5 c$$

Since we used machine learning to create the models, and it is difficult to explain our methodology without assuming some knowledge of this topic, we leave the details to Appendix B. Here, we simply visualize the models as heatmaps in Figure 4.3. In these plots, size is on the x axis and coverage is on the y axis. The predicted effectiveness of a test suite containing s test cases and having block coverage c is given by the colour of the tile at the coordinate (s, c). The colours are consistent across plots; that is, a given colour corresponds to the same effectiveness value in each subfigure. The models for Apache POI and Joda Time predict effectiveness values greater than 1 for large suites with high coverage; since this is not meaningful, we have capped the output at 1 to indicate that, according to the model, further increases in size or coverage will not be beneficial.

It is interesting to note how featureless the plots are. The logarithmic terms in the model lead to some asymptotic behaviour as size and coverage go to zero, but for larger values of size and coverage there is not much variation in effectiveness. Effectiveness does increase with coverage, but the increase is not large, and once 60 to 80% coverage is reached further increases have little to no impact on effectiveness. These results suggest that coverage is not a good way of measuring the quality of a test suite, confirming the results of our first analysis.

Answer 3 *Our results suggest that, for large Java programs, there is a low correlation between the effectiveness and the block coverage of a test suite when the number of tests in the suite is controlled for.*



(a) Apache POI

(b) HSQLDB

(c) JFreeChart

(d) Joda Time

Figure 4.3: Visualization of the linear regression equations.

Chapter 5

Discussion

We found that there is a moderate to high correlation between block coverage and effectiveness when the size of a test suite is not considered, but only a low correlation when the size of the suite is taken into account. This suggests that block coverage alone is not a good predictor of test suite effectiveness; its apparent prediction value is mainly due to the fact that high coverage suites contain more tests.

It is interesting to note that, whether or not size is considered, the strength of the correlation varies by project: Joda Time and Apache POI tend to have higher correlation values than JFreeChart and HSQLDB. Looking at Table 3.1, we can also see that Joda Time and Apache POI have high coverage suites (89% and 73%, respectively) while JFreeChart and HSQLDB have lower coverage suites (57% and 37%, respectively). This seems to agree with earlier studies that found that coverage is only a good indicator of effectiveness once very high levels have been reached [12, 15, 22]. However, even at the high coverage levels reached by Joda Time and Apache POI, there was a low correlation between coverage and effectiveness. Thus, while the correlation may increase when coverage is high, it seems that it is still not strong enough to make coverage a good indicator of effectiveness.

Our results have implications for how developers should use block coverage. While it is useful for identifying under-tested parts of a program, and low block coverage may indicate that a test suite is inadequate, high block coverage does not indicate that a test suite is effective. This means that using a fixed coverage value as a quality target is unlikely to produce an effective test suite. While members of the testing community have previously made this point [11, 27], it has been difficult to evaluate their suggestions due to a lack of studies that considered production-quality programs.

While interpreting our results, it is important to remember that we have only studied one type of coverage in this thesis. Though block coverage is widely used, it is subsumed by several other types of coverage, as described in Section 2.1. It is therefore still possible that stricter coverage measurements are correlated with effectiveness independently of size and could be used as quality targets.

Chapter 6

Threats to Validity

In this chapter, we discuss the threats to the construct validity, internal validity, and external validity of our study.

6.1 Construct Validity

To perform this study, we had to measure the size, block coverage and effectiveness of random test suites. Size and block coverage are straightforward to measure, but effectiveness is more nebulous as we are attempting to predict the fault-detection ability of a suite that has never been used. As we described in Section 3.2, several studies have established that a suite's ability to kill mutants is a valid measurement of its ability to detect real faults [2, 3, 6]. This suggests that, in the absence of equivalent mutants, this metric has high construct validity. Unfortunately, our treatment of equivalent mutants introduces a threat to the validity of this measurement. Recall that we assumed that any mutant that could not be detected by the program's entire test suite is equivalent. For programs with small test suites, this meant that we classified a large number of mutants as equivalent – as high as 95% for HSQLDB. In theory, these mutants are a random subset of the entire set of mutants, so ignoring them should not affect our results. However, this may not be true: for example, if the developers frequently test for off-by-one errors, mutants that simulate this error will be detected more often and thus will be less likely to be classified as equivalent.

6.2 Internal Validity

Our conclusions about the relationship between size, coverage and effectiveness depend on our calculations of Kendall's τ. This introduces a threat to the internal validity of the study. Kendall's original formula for τ assumes that there are no tied ranks in the data; that is, if the data were sorted, no two rows could be exchanged without destroying the sorted order. When ties do exist, two issues arise: first, since the original formula does not handle ties, a modified one must be used; we used the version proposed by Adler [1]. Second, ties make it difficult to compute the statistical significance of the correlation coefficient.

It it possible to show that, in the absence of ties, τ is normally distributed, meaning we can use Z-scores to evaluate significance in the usual way. However, when ties are present, the distribution of τ changes in a way that depends on the number and nature of the ties. This can result in a non-normal distribution [19]. To determine the impact of ties on our calculations, we counted both the number of ties that occurred and the total number of comparisons done to compute each τ. The fourth column of Table 4.1 shows the percentage of comparisons that resulted in a tie when computing the τ values in the second column. Table 4.3 shows the percentage of comparisons that resulted in a tie when computing the corresponding τ values in Table 4.2. As can be seen from these tables, ties rarely occurred: for the worst calculation, 0.69% of the comparisons resulted in a tie, but for most calculations this percentage was smaller by several orders of magnitude. Since there were so few ties, we have assumed that they will have a negligible effect on the normal distribution.

Another threat to internal validity stems from the possibility of duplicate test suites: if two or more suites contain the same subset of test methods, this might skew our results. Fortunately, we can use the data in Tables 4.1 and 4.3 to evaluate the severity of this threat. Since duplicate suites would naturally have identical coverage and effectiveness scores, the number of tied comparisons provides an upper limit on how many identical suites were compared. Since the number of ties was so low, the number of duplicate suites must be similarly low, and so we have ignored the small skew they may have introduced to avoid increasing the memory requirements of our study unnecessarily.

For research questions 1 and 3, our results also depend on the quality of our machine learning models. The primary threat to the validity of these models is overfitting. We evaluated this threat by graphing the learning curves of the models (see Appendix B) and found that three of the four projects display no sign of overfitting. Apache POI was the exception; for both regressions, the learning curve suggests that the model may have overfit the data slightly. That said, in both cases the prediction error on the test set was still very low, so the problem does not seem to be serious.

6.3 External Validity

There are five main threats to the external validity of our study. First, previous work suggests that the relationship between size, coverage and effectiveness depends on the difficulty of detecting faults in the program [3]. Since our test subjects are mature programs with substantial test suites, any remaining faults will be fairly difficult to detect. Our results may therefore not be accurate during a program's initial development, which is unfortunately when many tests are written. Moreover, some of the previous work was done with hand-seeded faults, which have been shown to be harder to detect than both mutants and real faults [2]. While this does not affect our results, it does make it harder to compare them with those of earlier studies.

Second, in object-oriented systems, most faults are usually found in just a few of the system's components [10]. This means that the relationship between size, coverage and effectiveness may vary by class within the system. It is therefore possible that block coverage is correlated with effectiveness in classes with specific characteristics, such as high churn. However, our conclusions still hold for the common practice of measuring the coverage of a program's entire test suite.

Third, previous work suggests that the size of a class can affect the validity of object-oriented metrics [9]. While we controlled for the size of each test suite in this study, we did not control for the size of the class each test method came from.

Fourth, as discussed in Section 3.1, our test subjects had to meet certain criteria: the programs had to be reasonably large, be written in Java, be actively developed, have a fairly large number of test methods, use Ant as a build system, and use JUnit 3 as a test harness. This means that our test subjects have a great deal in common, and so our results may not generalize to programs that do not meet these criteria. We attempted to mitigate this threat by selecting programs from different application domains, thereby ensuring a certain amount of variety in the subjects.

Finally, while our test subjects were considerably larger than the programs used in previous studies, they are still not large by industrial standards. Additionally, all of the projects were open source, so our results may not generalize to closed source or even larger software.

Chapter 7

Future Work

Our study focussed on block coverage on the grounds that this reflects current development practice, making our results more generalizable. However, block coverage is subsumed by a number of other coverage criteria, so it is possible that these stricter criteria are better predictors of test suite effectiveness. This possibility should be investigated in a future study.

Future work might also look into the relationship between coverage and effectiveness at the level of individual test methods. As can be seen in Figure 4.1, there is a great deal of variation in the effectiveness of the smallest test suites. In addition, Figure 4.2 contains several outlier suites that have very high coverage but very low effectiveness or vice versa. These results indicate that there is a great deal of variance in the effectiveness of individual test cases. Exploring the reasons for this may provide insight into the properties that make tests effective.

Finally, it may be helpful to do a longitudinal study that considers how the coverage and effectiveness of a program's test suite change over time. By cross-referencing coverage information with bug reports, it would be possible to isolate those bugs that were covered by the test suite but were not immediately detected by it. Examining these bugs may provide insight into which bugs are the most difficult to detect and how we can improve our chances of detecting them.

Chapter 8

Conclusion

In this thesis, we studied the relationship between the number of methods in a program's test suite, the suite's block coverage, and the suite's ability to kill mutants. We concluded that, for large Java programs, there is a low correlation between the block coverage of a test suite and its effectiveness when its size is controlled for. This implies that block coverage is a poor predictor of effectiveness and should not be used as a quality target.

References

[1] L. M. Adler. A modification of Kendall's tau for the case of arbitrary ties in both rankings. *Journal of the American Statistical Association*, 52(277), 1957.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the International Conference on Software Engineering*, 2005.

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8), 2006.

[4] L. Briand and D. Pfahl. Using simulation for assessing the real impact of test coverage on defect coverage. In *Proceedings of the International Symposium on Software Reliability Engineering*, 1999.

[5] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *Proceedings of the International Workshop on Advances in Model-Based Testing*, 2005.

[6] M. Daran and P. Thévenod-Fosse. Software error analysis: a real case study involving real faults and mutations. In *Proceedings of the International Symposium on Software Testing and Analysis*, 1996.

[7] M. E. Delamaro and J. C. Maldonado. Proteum – a tool for the assessment of test adequacy for C programs. In *Proceedings of the Conference on Performability in Computing Systems*, 1996.

[8] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King. An extended overview of the Mothra software testing environment. In *Proceedings of the Workshop on Software Testing, Verification, and Analysis*, 1988.

[9] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7), 2001.

[10] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8), 2000.

[11] M. Fowler. Test coverage. `http://martinfowler.com/bliki/TestCoverage.html`, 2012.

[12] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the International Symposium on Foundations of Software Engineering*, 1998.

[13] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, 1991.

[14] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8), 1993.

[15] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3), 1997.

[16] P. G. Frankl, S. N. Weiss, and E. J. Weyuker. *ASSET: A System to Select and Evaluate Tests.* Courant Institute of Mathematical Sciences, New York University, 1985.

[17] P. G. Frankl and E. J. Weyuker. Assessing the fault-detecting ability of testing methods. In *Proceedings of the Conference on Software for Critical Systems*, 1991.

[18] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3), 1993.

[19] J. D. Gibbons. *Nonparametric Measures of Association.* Sage Publications, 1993.

[20] J. P. Guilford. *Fundamental Statistics in Psychology and Education.* McGraw-Hill, 1942.

[21] J. R. Horgan, S. London, and M. R. Lyu. Achieving software quality with testing coverage measures. *Computer*, 27(9), 1994.

[22] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering*, 1994.

[23] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5), 2010.

[24] K. Kapoor. Formal analysis of coupling hypothesis for logical faults. *Innovations in Systems and Software Engineering*, 2(2), 2006.

[25] E. Kit. *Software Testing in the Real World: Improving the Process.* ACM Press, 1995.

[26] M. R. Lyu, J. R. Horgan, and S. London. A coverage analysis tool for the effectiveness of software testing. *IEEE Transactions on Reliability*, 43(4), 1994.

[27] B. Marick. How to misuse code coverage. `http://www.exampler.com/testing-com/writings/coverage.pdf`, 1997.

[28] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2009.

[29] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1), 1992.

[30] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In *Proceedings of the Conference on Computer Assurance*, 1996.

[31] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation Testing for the New Century.* Kluwer Academic Publishers, 2001.

[32] T. J. Ostrand and E. J. Weyuker. Data flow-based test adequacy analysis for languages with pointers. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, 1991.

[33] W. Perry. *Effective Methods for Software Testing.* Wiley Publishing, 2006.

[34] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4), 1985.

[35] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, 2009.

[36] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *Proceedings of the International Symposium on Software Reliability Engineering*, 1994.

APPENDICES

Appendix A

Javalanche Modifications

As discussed in Section 3.2, we modified the mutation testing tool that we used in this study. The changes are given below as a diff against the version of Javalanche that was available at the end of January 2012, specifically, commit 2fa6e1a48ad84777d352554a1587c2e08dc16557[1]. Long lines in the diff have been wrapped to fit the margins of this thesis.

A.1 Allowing Multiple Results Per Mutation

By default, Javalanche only records one result per mutation, as discussed in Section 3.2. To allow more than one result per mutation, we made changes to three files: `MutationsForRun.java`, `MutationSwitcher.java` and `MutationTestDriver.java`.

```
diff -ru javalanche-2fa6e1a/src/main/java/de/unisb/cs/st/javalanche/mutation/
javaagent/MutationsForRun.java
javalanche-src/src/main/java/de/unisb/cs/st/javalanche/mutation/
javaagent/MutationsForRun.java
--- javalanche-2fa6e1a/src/main/java/de/unisb/cs/st/javalanche/mutation/
javaagent/MutationsForRun.java 2012-01-31 12:21:47.000000000 -0500
+++ javalanche-src/src/main/java/de/unisb/cs/st/javalanche/mutation/
javaagent/MutationsForRun.java 2012-04-15 21:59:16.408717258 -0400
@@ -59,7 +59,7 @@
   *         specified at the command line.
   */
  public static MutationsForRun getFromDefaultLocation() {
-return getFromDefaultLocation(true);
+ return getFromDefaultLocation(false); // allow more than one result per mutation
                                                        by making filter false
```

[1] http://github.com/david-schuler/javalanche/commit/2fa6e1a48ad84777d352554a1587c2e08dc16557

```
  }

  public static MutationsForRun getFromDefaultLocation(boolean filter) {
@@ -127,9 +127,9 @@
  logger.warn("Mutation file does not exist: " + idFile);
  }

-if (filter) {
-filterMutationsWithResult(mutationsToReturn);
-}
+ //if (filter) {
+ // filterMutationsWithResult(mutationsToReturn);
+ //} don't remove mutants that have already been killed once
  return mutationsToReturn;
  }

@@ -161,7 +161,7 @@
   * @param mutations
   *            the list of mutations to be filtered.
   */
-private static void filterMutationsWithResult(List<Mutation> mutations) {
+ /*private static void filterMutationsWithResult(List<Mutation> mutations) {
  if (mutations != null) {
  // make sure that we have not got any mutations that have already an
  // result
@@ -180,7 +180,7 @@
  tx.commit();
  session.close();
  }
-}
+ }*/ // this function should not be called, comment it out to be sure

  /**
   * Checks whether a mutation is a mutation for this run (should be applied).

diff -ru javalanche-2fa6e1a/src/main/java/de/unisb/cs/st/javalanche/mutation/
runtime/MutationSwitcher.java
javalanche-src/src/main/java/de/unisb/cs/st/javalanche/mutation/
runtime/MutationSwitcher.java
--- javalanche-2fa6e1a/src/main/java/de/unisb/cs/st/javalanche/mutation/
runtime/MutationSwitcher.java 2012-01-31 12:21:47.000000000 -0500
+++ javalanche-src/src/main/java/de/unisb/cs/st/javalanche/mutation/
runtime/MutationSwitcher.java 2012-04-15 21:59:16.408717258 -0400
@@ -79,11 +79,11 @@
  public Mutation next() {
  while (iter.hasNext()) {
```

```
  currentMutation = iter.next();
-if (currentMutation.getMutationResult() == null) {
+ //if (currentMutation.getMutationResult() == null) {
  return currentMutation;
-} else {
-logger.info("Mutation already got Results");
-}
+ //} else {
+ // logger.info("Mutation already got Results");
+ //} prevent it from skipping mutants that already have results
  }
  return currentMutation;
  }

diff -ru javalanche-2fa6e1a/src/main/java/de/unisb/cs/st/javalanche/mutation/runtime/
testDriver/MutationTestDriver.java
javalanche-src/src/main/java/de/unisb/cs/st/javalanche/mutation/runtime/
testDriver/MutationTestDriver.java
--- javalanche-2fa6e1a/src/main/java/de/unisb/cs/st/javalanche/mutation/runtime/
testDriver/MutationTestDriver.java 2012-01-31 12:21:47.000000000 -0500
+++ javalanche-src/src/main/java/de/unisb/cs/st/javalanche/mutation/runtime/
testDriver/MutationTestDriver.java 2012-04-15 21:59:16.408717258 -0400
@@ -371,7 +371,7 @@
  List<String> allTests = getAllTests();
  int counter = 0;
  int size = allTests.size();
-timeout = 120;
+ timeout = 600;
  if (doColdRun) {
  coldRun(allTests);
  }
@@ -443,9 +443,9 @@
   * carried out and their corresponding tests are run.
   */
  public void runMutations() {
-if (checkMutations()) {
-return;
-}
+ //if (checkMutations()) {
+ // return;
+ //} This exits if all the mutations in this run already have a result in the
                    db, which we don't want
  shutDownThread = new Thread(new MutationDriverShutdownHook(this));
  addMutationTestListener(new MutationObserver());
  addListenersFromProperty();
@@ -617,13 +617,19 @@
```

```
  testName);
  result.setTouched(touched);
  resultsForMutation.add(result);
-if (configuration.stopAfterFirstFail() && !result.hasPassed()) {
+ /*if (configuration.stopAfterFirstFail() && !result.hasPassed()) {
  logger.info("Test failed for mutation not running more tests. Test: "
  + testName);
  TestMessage testMessage = result.getTestMessage();
  logger.info("Message: " + testMessage.getMessage());
  break;
-}
+ }*/ // commented out if and replaced it with the following four lines
+
+ // added the following four lines
+ logger.info("Mutant with id " + currentMutation.getId() +
                                      " is being hunted by test " + currentTestName);
+  if (!result.hasPassed()) {
+      logger.info("Mutant with id " + currentMutation.getId() +
                             " has been killed by test " + currentTestName);
+  }
  }
  currentTestName = "No test name set";
  MutationTestResult mutationTestResult = SingleTestResult
@@ -664,6 +670,7 @@
  service.shutdown();
  String exceptionMessage = null;
  Throwable capturedThrowable = null;
+ timeout = 120;
  try {
  logger.debug("Start  test: ");
  boolean terminated = service.awaitTermination(timeout,
@@ -748,6 +755,7 @@
  thread.start();
  String exceptionMessage = null;
  Throwable capturedThrowable = null;
+ timeout = 600;
  try {
  future.get(timeout, TimeUnit.SECONDS);
  logger.debug("Second timeout");
```

A.2 Changing the Database

By default, Javalanche uses HSQLDB to store its results. However, this database could not handle Unicode strings and was very slow, so we replaced it with MySQL. We also replaced the default connection pool

manager C3P0 with BoneCP due to known issues with C3P0. We did this by making the following changes to `hibernate.cfg.xml`. We also changed one of the column mappings in `TestMessage.java` to allow the column to store four byte Unicode characters.

```
diff -ru javalanche-2fa6e1a/src/main/resources/hibernate.cfg.xml
javalanche/src/main/resources/hibernate.cfg.xml
--- javalanche-2fa6e1a/src/main/resources/hibernate.cfg.xml
2012-01-31 12:21:47.000000000 -0500
+++ javalanche/src/main/resources/hibernate.cfg.xml
2012-04-17 01:51:21.795533873 -0400
@@ -5,51 +5,40 @@
 <hibernate-configuration>
     <session-factory>

-        <!--
-        <property name="hibernate.connection.driver_class">
             com.mysql.jdbc.Driver</property>
-        <property name="hibernate.connection.url">
             jdbc:mysql://localhost:3308/mutation_test</property>
-        <property name="hibernate.connection.username">mutation</property>
-        <property name="hibernate.connection.password">mu</property>
-        <property name="hibernate.dialect">
             org.hibernate.dialect.MySQLDialect</property>
-        <property name="hibernate.jdbc.batch_size">20</property>
-
-        -->
-        <property name="hibernate.transaction.factory_class">
             org.hibernate.transaction.JDBCTransactionFactory</property>
-
-        <property name="hibernate.connection.driver_class">
             org.hsqldb.jdbcDriver</property>
-        <property name="hibernate.connection.url">
             jdbc:hsqldb:hsql://localhost/mt</property>
-        <property name="hibernate.connection.username">sa</property>
-        <property name="hibernate.dialect">
             org.hibernate.dialect.HSQLDialect</property>
-        <property name="hibernate.jdbc.batch_size">1</property>
-
-
-        <property name="hibernate.c3p0.min_size">5</property>
-        <property name="hibernate.c3p0.max_size">10</property>
-        <property name="hibernate.c3p0.timeout">1000</property>
-        <!--property name="hibernate.c3p0.max_statements">50</property-->
-        <property name="hibernate.c3p0.max_statements">0</property>
-        <property name="hibernate.c3p0.idle_test_period">3000</property>
-
```

```
-
-
-
-        <!-- SQL to stdout logging
-        <property name="show_sql">true</property>
-        <property name="format_sql">true</property>
-        <property name="use_sql_comments">true</property>
-        -->
-
-
-        <mapping class="de.unisb.cs.st.javalanche.mutation.results.Mutation"/>
-        <mapping class="de.unisb.cs.st.javalanche.mutation.results.MutationTestResult"/>
-        <mapping class="de.unisb.cs.st.javalanche.mutation.results.TestMessage"/>
-        <mapping class="de.unisb.cs.st.javalanche.mutation.results.TestName"/>
-        <!--mapping class="de.unisb.cs.st.javalanche.mutation.results.AddResult"/-->
-
-        <!--mapping class="de.unisb.cs.st.javalanche.mutation.results.MutationCoverage"/-->
-        <!--mapping class="de.unisb.cs.st.javalanche.mutation.results.Invariant"/-->
-        <!--mapping class="de.unisb.cs.st.javalanche.mutation.results.InvariantAddResult"/-->
-        <!--mapping class="de.unisb.cs.st.javalanche.mutation.results.InvariantSet"/-->
+        <property name="hibernate.transaction.factory_class">
            org.hibernate.transaction.JDBCTransactionFactory</property>
+        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver</property>
+        <property name="hibernate.connection.username">SA</property>
+        <property name="hibernate.connection.password"></property>
+        <property name="hibernate.jdbc.batch_size">0</property>
+        <property name="hibernate.connection.release_mode">after_statement</property>
+
+        <!-- Use these two properties to use the old database -->
+        <!-- <property name="hibernate.connection.url">
            jdbc:hsqldb:hsql://localhost/mt</property> -->
+        <!-- <property name="hibernate.dialect">
            org.hibernate.dialect.HSQLDialect</property> -->
+
+        <!-- Use these five properties to use MySQL -->
+        <property name="hibernate.connection.url">
            jdbc:mysql://localhost/mt?autoReconnect=true</property>
+        <property name="hibernate.dialect">
            org.hibernate.dialect.MySQL5InnoDBDialect</property>
+        <property name="hibernate.connection.useUnicode">true</property>
+        <property name="hibernate.connection.charSet">UTF-8</property>
+        <property name="hibernate.connection.characterEncoding">UTF-8</property>
+
+        <!-- Manage the connection pool with BoneCP -->
+        <property name="hibernate.connection.provider_class">
```

```
            com.jolbox.bonecp.provider.BoneCPConnectionProvider</property>
+        <property name="bonecp.partitionCount">3</property>
+        <property name="bonecp.maxConnectionsPerPartition">20</property>
+        <property name="bonecp.minConnectionsPerPartition">2</property>
+        <property name="bonecp.acquireIncrement">3</property>
+
+        <!-- Debugging: print SQL statements to stdout -->
+        <!-- <property name="show_sql">true</property> -->
+        <!-- <property name="format_sql">true</property> -->
+        <!-- <property name="use_sql_comments">true</property> -->
+
+        <mapping class="de.unisb.cs.st.javalanche.mutation.results.Mutation"/>
+        <mapping class="de.unisb.cs.st.javalanche.mutation.results.MutationTestResult"/>
+        <mapping class="de.unisb.cs.st.javalanche.mutation.results.TestMessage"/>
+        <mapping class="de.unisb.cs.st.javalanche.mutation.results.TestName"/>

     </session-factory>
 </hibernate-configuration>

diff -ru javalanche-2fa6e1a/src/main/java/de/unisb/cs/st/javalanche/mutation/
results/TestMessage.java
javalanche-src/src/main/java/de/unisb/cs/st/javalanche/mutation/
results/TestMessage.java
--- javalanche-2fa6e1a/src/main/java/de/unisb/cs/st/javalanche/mutation/
results/TestMessage.java 2012-01-31 12:21:47.000000000 -0500
+++ javalanche-src/src/main/java/de/unisb/cs/st/javalanche/mutation/
results/TestMessage.java 2012-04-16 12:57:16.701577109 -0400
@@ -24,6 +24,7 @@
 import javax.persistence.GeneratedValue;
 import javax.persistence.Id;
 import javax.persistence.Lob;
+import javax.persistence.Column;

 import org.apache.log4j.Logger;

@@ -60,7 +61,7 @@
   /**
    * Failure or error message of the TestCase.
    */
-@Lob
+ @Column(columnDefinition="longtext character set utf8mb4 collate utf8mb4_general_ci")
   private String message;

   /**
```

A.3 Changing Logging Behaviour

We retrieved information about Javalanche's runs from a log file, which required making the following changes to `log4j.properties`.

```
diff -ru javalanche-2fa6e1a/src/main/resources/log4j.properties
javalanche/src/main/resources/log4j.properties
--- javalanche-2fa6e1a/src/main/resources/log4j.properties
2012-01-31 12:21:47.000000000 -0500
+++ javalanche/src/main/resources/log4j.properties
2012-04-23 02:35:01.944125109 -0400
@@ -1,9 +1,9 @@
-log4j.rootCategory=WARN, A1
-#log4j.rootCategory=DEBUG, A1
-log4j.appender.A1=org.apache.log4j.ConsoleAppender
+log4j.rootCategory=INFO, A1
+log4j.appender.A1=org.apache.log4j.FileAppender
+log4j.appender.A1.file=/home/laura/workspace/coverage-size-corr/test-subjects/mutation.log
 log4j.appender.A1.layout=org.apache.log4j.PatternLayout
 log4j.appender.A1.layout.ConversionPattern=%6r %-5p [%M - %C] -  %m%n

-log4j.logger.com.mchange=WARN
-log4j.logger.org.hibernate=WARN
-log4j.logger.de.unisb.cs.st.javalanche.mutation.results.persistence.QueryManager=WARN
\ No newline at end of file
+log4j.logger.com.mchange=INFO
+log4j.logger.org.hibernate=INFO
+log4j.logger.de.unisb.cs.st.javalanche.mutation.results.persistence.QueryManager=DEBUG
Only in javalanche-2fa6e1a/src/main/resources: log4j-silent.properties
Only in javalanche-2fa6e1a/src/main/resources: MANIFEST.txt
Only in javalanche-2fa6e1a/src/main/resources: mutation-add-tasks.xml
Only in javalanche-2fa6e1a/src/main/resources: report
Only in javalanche-2fa6e1a/src/main/resources: runMutations.sh
Only in javalanche-2fa6e1a/src/: site
Only in javalanche-2fa6e1a/src/: test
```

Appendix B

Linear Regression Details

As discussed in Sections 4.1 and 4.3.2, we used linear regression to answer our first and third research questions. The models were created with machine learning; this section discusses the methodological details. We begin by explaining the procedure we used for research question 1, then discuss how it generalizes to research question 3. We assume some knowledge of multivariate linear regression and machine learning techniques.

B.1 Linear Regression for Research Question 1

In this section, we explain how we used machine learning to model the relationship between effectiveness and size. We trained four models, one for each test subject, so that we could compare the results by project. For each project, we began by randomly reordering our data points. We then divided them into three groups: a training set, containing 60% of the points; a cross validation set, containing the next 20% of the points; and a test set, containing the final 20% of the points.

Next, we prepared the input features that we might want to include in the model. We chose four features: a constant term, the square root of size, the natural logarithm of size, and size. We did not include super-linear functions of size since it seemed unlikely that effectiveness would improve with the square or higher power of size. Since our input features have different ranges, we normalized each one prior to use by dividing it by its standard deviation. This means that every feature ranged between approximately 0 and 6. We did not normalize the mean to zero, although this is commonly done, because having negative inputs would have prevented us from including a logarithmic term in the model. The normalization values are given in Table B.1.

Having four potential input features meant that we had 15 possible models, one for each nonempty subset of the input features. We selected the best subset as follows. First, for each subset, we trained twelve models using only the input features in the subset. Each of the twelve models used a different value of the regularization parameter λ. Once we had those twelve models, we evaluated them on the cross validation set and selected the λ that led to the lowest prediction error. By doing this for each subset, we got 15 possible models, each with a different subset of the input features and a different λ. We then

evaluated these fifteen models on the test set and selected the one with the lowest prediction error as the best model for the project. This procedure is summarized as pseudocode in Listing B.1.

```
for each project P {
    for each subset S of the input features {
        for each lambda L in {0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100} {
            train a model using S and L
            compute its prediction error E_cv using the cross validation set
        }
        choose the best lambda by picking the model M with the lowest E_cv
        compute M's prediction error E_test using the test set
    }
    choose the best model M' by picking the one with the lowest E_test
}
```

Listing B.1: Pseudocode explaining our regression procedure.

Project	**Ln(size)**	**Sqrt(size)**	**Size**
Apache POI	1.9742	10.3469	354.5169
HSQLDB	1.6285	5.6373	111.1499
JFreeChart	1.9742	10.3469	354.5169
Joda Time	2.3029	17.9032	1019.9449

Table B.1: The standard deviation of the input features for each project. These values were used to normalize the features before training the models for research question 1.

The cost (prediction error) function we used is

$$C = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

where the first term inside the square brackets represents the sum of the squared residuals and the second term is an adjustment factor based on λ. Table B.2 shows the prediction error of our models when they were evaluated on the test set. Since the prediction error is very low for all four projects, these models seem to be a good representation of the relationship between size and effectiveness.

We can also evaluate the models by inspecting their learning curves, shown in Figure B.1. A learning curve displays the prediction error of the model on the training and cross validation sets as the number of data points in the training set increases. In general, when there are few data points in the training set, the model will overfit the data. This will result in a small training error, but a large cross validation error, since the model will not generalize to new data. As the number of training points increases, the training error should increase, since it becomes more difficult to fit the model to the data, and the cross validation error should decrease, since the model can more easily generalize to new data. Ideally, the two lines will

Project	Error
Apache POI	0.004
HSQLDB	0.001
JFreeChart	0.003
Joda Time	0.002

Table B.2: The prediction error of the best model for each project when evaluated on the test set for research question 1.

meet or nearly meet at the right side of the graph. This is the case for three of the projects, but the learning curve for Apache POI indicates that the model could be improved. The gap between the training and cross validation error at the right side of the graph suggests that the model is overfitting the training data. This is also indicated by the fact that it has the highest error on the test set, as seen in Table B.2. That said, the error is still fairly small, so the model is reasonably good.

The final output of the regression was the following four equations where, as before, e represents effectiveness and s represents size. Recall that the input features must be normalized using the standard deviations in Table B.1 before being substituted into these equations.

$$e_{\text{POI}} = 0.23 + 0.46 \ln s - 0.56\sqrt{s} + 0.30s \tag{B.1}$$

$$e_{\text{HSQL}} = 0.65 + 0.07 \ln s \tag{B.2}$$

$$e_{\text{JFree}} = 0.64 - 0.03 \ln s + 0.25\sqrt{s} - 0.12s \tag{B.3}$$

$$e_{\text{Joda}} = 0.50 + 0.13 \ln s + 0.02\sqrt{s} \tag{B.4}$$

B.2 Linear Regression for Research Question 3

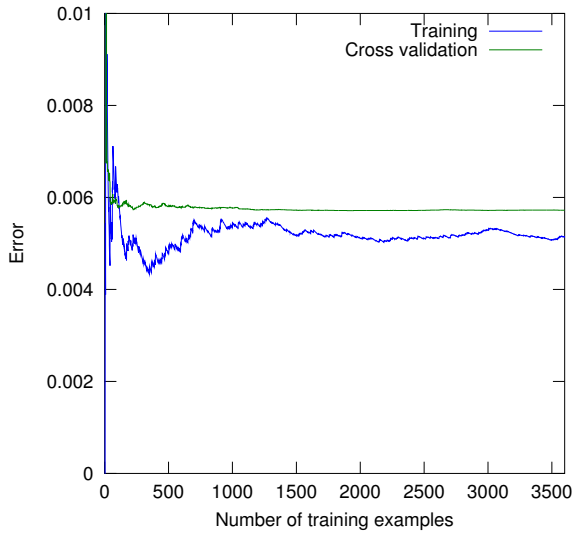
We performed the regression for research question 3 using the same procedure that was used for research question 1. The only difference was the set of possible input features. We included six: the square root of size, the natural logarithm of size, size, the square root of coverage, the natural logarithm of coverage, and coverage. Again, we normalized the features before doing the regression using the standard deviations in Table B.3. Table B.4 shows the prediction error on the test set for all four projects and Figure B.2 shows the learning curves for all four projects; they are analogous to Table B.2 and Figure B.1. They again suggest that the models are accurate, though the model for Apache POI may have overfit the data slightly. The final output was the equations given below.

$$e_{\text{POI}} = 0.39 \ln s - 0.59\sqrt{s} + 0.31s + 0.07\sqrt{c} + 0.02c \tag{B.5}$$

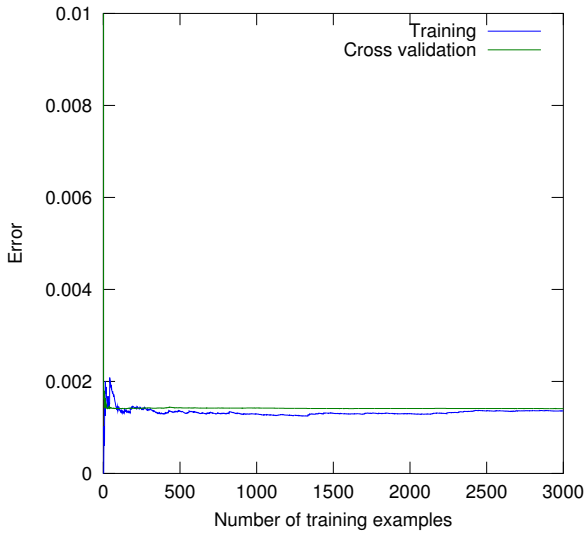
$$e_{\text{HSQL}} = 0.04 \ln s + 0.01\sqrt{s} + 0.02s - 0.02 \ln c + 0.04c \tag{B.6}$$

$$e_{\text{JFree}} = 0.19\sqrt{s} - 0.09s - 0.08 \ln c + 0.08\sqrt{c} \tag{B.7}$$

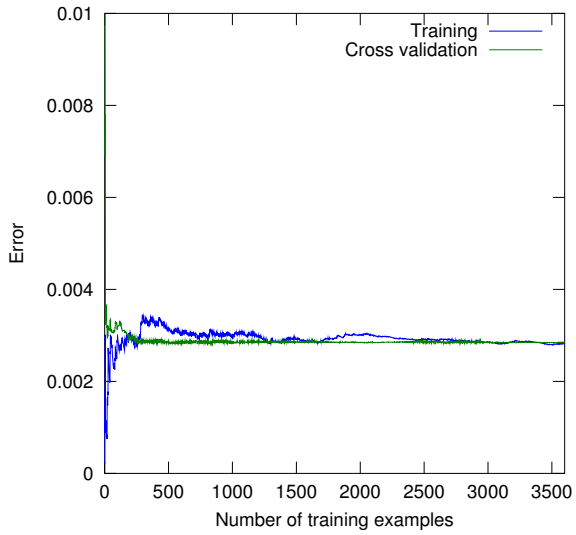
$$e_{\text{Joda}} = 0.15 \ln s - 0.12\sqrt{s} - 0.13 \ln c - 0.11\sqrt{c} + 0.35c \tag{B.8}$$



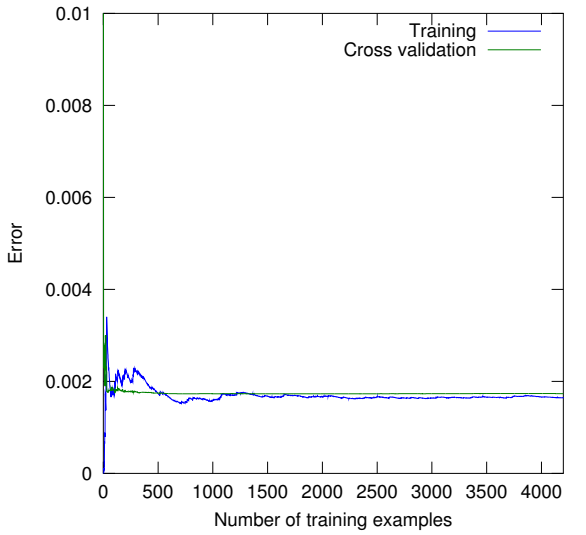
(a) Apache POI



(b) HSQLDB



(c) JFreeChart



(d) Joda Time

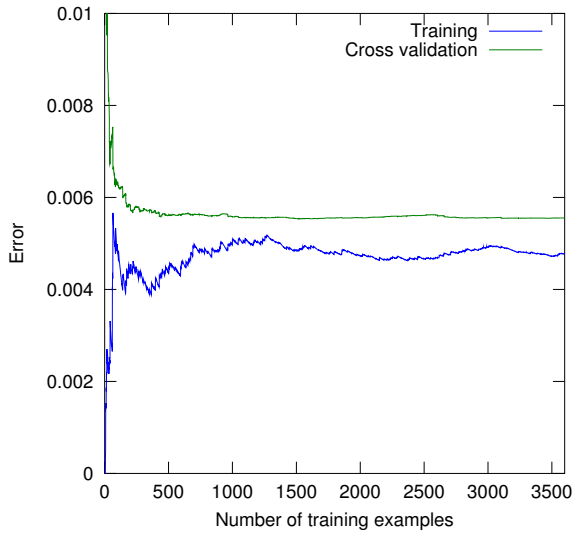
Figure B.1: Learning curves for the four regression models for research question 1.

Project	Ln(size)	Sqrt(size)	Size	Ln(coverage)	Sqrt(coverage)	Coverage
Apache POI	1.9742	10.347	354.5	0.47274	0.12860	0.16044
HSQLDB	1.6285	5.637	111.2	0.09674	0.02591	0.02800
JFreeChart	1.9742	10.347	354.5	0.45895	0.11362	0.11874
Joda Time	2.3029	17.903	1019.9	0.40010	0.13889	0.19864

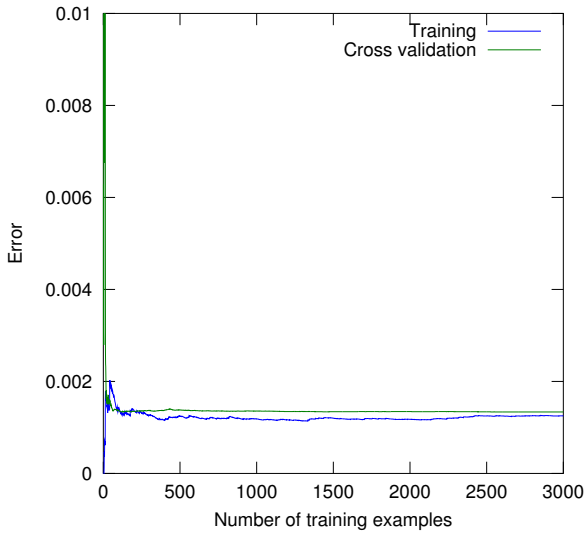
Table B.3: The standard deviation of the input features for each project. These values were used to normalize the features before training the models for research question 3.

Project	Error
Apache POI	0.004
HSQLDB	0.001
JFreeChart	0.003
Joda Time	0.002

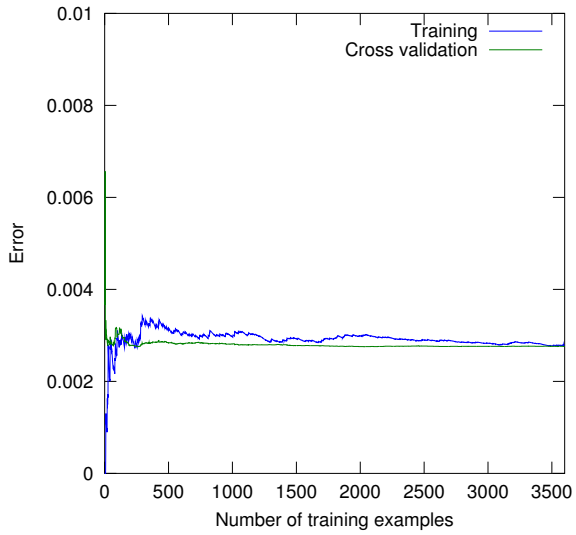
Table B.4: The prediction error of the best model for each project when evaluated on the test set for research question 3.



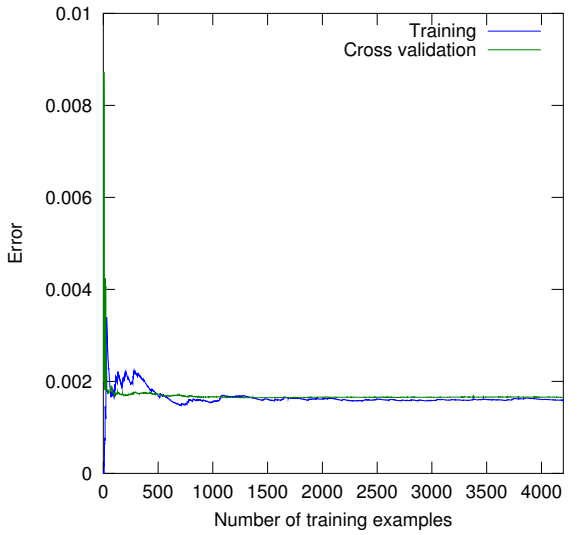
(a) Apache POI



(b) HSQLDB



(c) JFreeChart



(d) Joda Time

Figure B.2: Learning curves for the four regression models for research question 3.