Empirical Studies of Test Case Prioritization in a JUnit Testing Environment

Hyunsook Do* dohy@cs.orst.edu

Gregg Rothermel† grother@cs.orst.edu

Alex Kinneer‡ kinneer@cs.orst.edu

Abstract

Test case prioritization provides a way to run test cases with the highest priority earliest. Numerous empirical studies have shown that prioritization can improve a test suite's rate of fault detection, but whether the results of these studies generalize to other language paradigms is an open question because the studies have all focused on a single procedural language, C, and on only a few specific types of test suites. In particular, Java and the JUnit testing framework are being used extensively in practice, and the effectiveness of prioritization techniques on Java systems tested under JUnit has not been investigated. We have therefore designed and performed a controlled experiment examining whether test case prioritization can be effective on Java programs tested under JUnit, and comparing results to those achieved in earlier studies. Our analyses show that test case prioritization can significantly improve the rate of fault detection of JUnit test suites, but also reveal differences with respect to previous studies that can be related to the language and testing paradigm.

1 Introduction

As a software system evolves, software engineers regression test it to detect whether new faults have been introduced into previously tested code. The simplest regression testing technique is to re-run all existing test cases, but this can require a lot of effort, depending on the size and complexity of the system under test. For this reason, researchers have studied various techniques for improving the cost-effectiveness of regression testing, such as regression test selection [10, 26], test suite minimization [9, 18, 22], and test case prioritization [15, 28, 31].

Test case prioritization provides a way to run test cases that have the highest priority -according to some criterion-earliest, and can yield meaningful benefits, such as providing earlier feedback to testers and earlier detection of faults. Numerous prioritization techniques have been described in the research literature and they have been evaluated through various empirical studies [11, 14, 15, 27, 28, 29, 31]. These studies have shown that several prioritization techniques can improve a test suite's rate of fault detection. All of these studies, however, have focused on a single procedural language, C, and on only a few specific types of test suites, so whether these results generalize to other programming and testing paradigms is an open question. Replication of these studies with populations other than those previously examined is needed, to support a more complete understanding of test case prioritization.

In this work, we set out to perform a replicated study, focusing on an object-oriented language, Java, that is rapidly gaining usage in the software industry, and thus is practically important in its own right. We focus further on the JUnit testing framework, which is increasingly being used by developers to implement test cases for Java programs [1]. In fact, with the introduction of the JUnit testing framework, many software development organizations are building JUnit test cases into their code bases, as is evident through the examination of Open Source Software hosts such as SourceForge and Apache Jakarta [2, 3].

The JUnit framework encourages developers to write test cases, and then to rerun all of these test cases whenever they modify their code. As mentioned previously, however, as the size of a system grows, rerun-all regression testing strategies can be excessively expensive. JUnit users will need methodologies with which to remedy this problem[1].

We have therefore designed and performed a controlled experiment examining whether test case prioritization can be effective on object-oriented systems, specifically those written in Java and tested with JUnit test cases. We examine prioritization effectiveness in terms of rate of fault detection, and we also consider whether empirical results show similarity (or dissimilarity) with respect to the results of previous studies. As objects of study we consider four open source Java programs that have JUnit test suites, and we

*Department of Computer Science, Oregon State University

†Department of Computer Science, Oregon State University

‡Department of Computer Science, Oregon State University

[1] This may not be the case with respect to extreme programming, another development methodology making use of JUnit test cases, because extreme programming is intended for development of modest size projects using between 2 and 12 programmers [30]. However, JUnit is also being used extensively in the testing of Java systems constructed by more traditional methodologies, resulting in large banks of integration and system tests. It is in this context that prioritization can potentially be useful.

examine the ability of several test case prioritization techniques to improve the rate of fault detection of these test suites, while also varying other factors such as the granularity of the prioritization technique and test suite. Our results indicate that test case prioritization can significantly improve the rate of fault detection of JUnit test suites, but also reveal differences with respect to previous studies that can be related to the Java and JUnit paradigms.

In the next section of this paper, we describe the test case prioritization problem and related work. Section 3 describes the JUnit testing framework, and extensions to that framework that allow it to support prioritization. Section 4 presents our experiment design, results, and analysis, describing what we have done in terms of experiment setup to manipulate JUnit test cases. Section 5 discusses our results, and Section 6 presents conclusions and future work.

2 Background and Related Work

2.1 Test Case Prioritization

Test case prioritization techniques [15, 28, 31] schedule test cases in an execution order according to some criterion. The purpose of this prioritization is to increase the likelihood that if the test cases are used for regression testing in the given order, they will more closely meet some objective than they would if they were executed in some other order. For example, testers might schedule test cases in an order that achieves code coverage at the fastest rate possible, exercises features in order of expected frequency of use, or increases the likelihood of detecting faults early in testing.

Depending on the types of information available for programs and test cases, and the way in which those types of information are used, various test case prioritization techniques can be utilized. One way in which techniques can be distinguished involves the type of code coverage information they utilize. Test cases can be prioritized in terms of the number of code statements, basic blocks, or methods they executed on a previous version of the software. For example, a *total block coverage prioritization* technique simply sorts test cases in the order of the number of blocks they covered, resolving ties randomly.

A second way in which techniques can be distinguished involves the use of "feedback". When prioritizing test cases, if a particular test case has been selected as "next best", information about that test case can be used to re-evaluate the value of test cases not yet chosen, prior to picking the next best test case. For example, *additional block coverage prioritization* iteratively selects a test case that yields the greatest block coverage, then adjusts the coverage information for the remaining test cases to indicate coverage of blocks not yet covered, and repeats this process until all blocks coverable by at least one test case have been covered. This process is then repeated on remaining test cases.

A third way in which prioritization techniques can be distinguished involves their use of information on code modifications. For example, the amount of change in a code element can be factored into prioritization, by weighting the elements covered using a measure of change.

Other dimensions along which prioritization techniques can be distinguished, and that have been suggested in the literature [14, 15], include test cost estimates, fault severity estimates, estimates of fault propagation probability, and usage statistics obtained through operational profiles.

2.2 Previous Empirical Work

Studies of cost-effectiveness

Early studies of test case prioritization focused on the cost-effectiveness of individual techniques, the estimation of a technique's performance, or comparisons of techniques [15, 27, 28, 29, 31]. These studies showed that various techniques could be cost-effective, and suggested tradeoffs among them. However, the studies also revealed wide variances in performance, and attributed these to factors involving the programs under test, test suites used to test them, and types of modifications made to the programs.

Studies of factors and generalization

Recent studies of prioritization have begun to examine the factors affecting prioritization effectiveness [11, 21, 24]. Rothermel et al. [24] studied the effects of test suite granularity on regression testing techniques, varying the composition of test suites and examining the effects on cost-effectiveness of test selection and prioritization. While this study did not consider correlating attributes of change with technique performance, Elbaum et al. [11] performed experiments exploring characteristics of program structure, test suite composition, and changes on prioritization, and identified several metrics characterizing these attributes that correlate with prioritization effectiveness.

More recent studies have examined how some of these factors affect the effectiveness and efficiency of test case prioritization, and have considered the generalization of findings through several controlled experiments [12, 16, 25]. These studies exposed tradeoffs and constraints that affect the success of techniques, and provide guidelines for designing and managing prioritization and testing processes.

All of the empirical work just described has concerned C programs and system-level test suites constructed for code coverage, or for partition-based coverage of requirements. In contrast, the study we describe here examines whether prior results generalize, by replicating previous experiments on a new population of programs and test suites (Java and JUnit), and examining whether the results are consistent with those of the previous studies.

3 JUnit Testing and Prioritization

JUnit test cases are Java classes that contain one or more test methods and that are grouped into test suites, as shown in Figure 1. The figure presents a simple hierarchy having only a single test-class level, but the tree can extend deeper using additional nesting of Test Suites. The leaf nodes in such a hierarchy, however, always consist of test-methods, where a test-method is a minimal unit of test code.

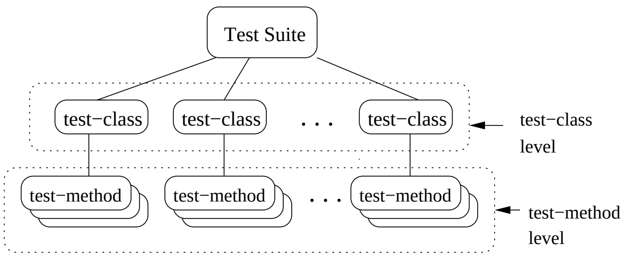

Figure 1. JUnit test suite structure

JUnit test classes that contain one or more test methods can be run individually, or a collection of JUnit test cases (a test suite) can be run as a unit. Running individual test cases is reasonable for small programs, but when the number of test cases increases, it can become prohibitively expensive, because each independent execution of a test incurs startup costs. Thus in practice, developers design JUnit tests to run as sequences of tests invoked through test suites that invoke test classes.

Clearly, choices in test suite granularity (the number of test cases making up a test suite) can affect the cost of running JUnit tests, and we want to investigate the relationship between this factor and prioritization results. To do this, we focus on two levels of test suite granularity: test-class level, a collection of test-classes that represents a coarse test suite granularity, and test-method level, a collection of test-methods that represents a fine test suite granularity. To support this focus we needed to ensure that the JUnit framework allowed us to achieve the following four objectives:

1. treat each TestCase class as a single test case for purposes of prioritization (test-class level);
2. reorder TestCase classes to produce a prioritized order;
3. treat individual test methods within TestCase classes as test cases for prioritization (test-method level);
4. reorder test methods to produce a prioritized order.

Objectives 1 and 2 were trivially achieved as a consequence of the fact that the default unit of test code that can be specified for execution in the JUnit framework is a TestCase class. Thus it was necessary only to extract the names of all TestCase classes invoked by the top level TestSuite for the object program[2] (a simple task) and then execute them individually with the JUnit test runner in a desired order.

Objective 3 was more difficult to achieve, due to the fact that a TestCase class is also the minimal unit of test code that can be specified for execution in the normal JUnit framework. Since a TestCase class can define multiple test methods, all of which will be executed when the TestCase is specified for execution, providing the ability to treat individual methods as test cases required us to extend the JUnit framework to support this finer granularity. Thus the principal challenge we faced was how to design and implement our JUnit extensions so as to provide a means for specifying individual test methods for execution, out of a total set of methods potentially spread across multiple TestCase classes.

Given that the fundamental purpose of the JUnit framework is to discover and execute test methods defined in TestCase classes, the problem of providing test-method level testing reduces to the problem of uniquely identifying each test method discovered by the framework and making them available for individual execution by the tester. We accomplished this task by extending (subclassing) various components of the framework and inserting mechanisms for assigning numeric test IDs to each test method discovered. We then created a SelectiveTestRunner that uses the new extension components. The relationship between our extensions and and the existing JUnit framework is shown in Figure 2, which also shows how the JUnit framework is related to the Galileo system for analyzing Java bytecode (which we used to obtain coverage information for use in prioritization). Our new SelectiveTestRunner checks the numeric ID assigned to each test case against the test numbers provided as input to determine whether a test case should be run, thus achieving the objective of allowing individual subsets of the test methods to be specified for execution.

To achieve objective 4, prioritization at the test-method level, we also needed to provide a way for the test methods to be executed in a tester-specified order. Because the JUnit framework must discover the test methods, and our extensions assign numeric IDs in the order of discovery, to

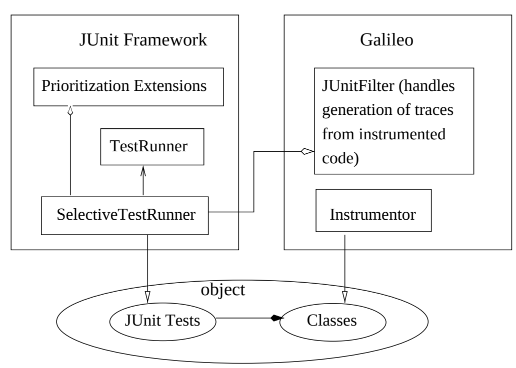

Figure 2. JUnit framework and Galileo

[2]This process may need to be repeated iteratively if Test Suites are nested in other Test Suites.

execute the tests in an order other than that in which they are provided requires that all tests must be discovered prior to execution. We accomplished this objective by using a simple two-pass technique. In the first pass, all the test methods relevant to the system are discovered and assigned numbers. The second pass then uses a specified ordering to retrieve and execute each method by its assigned ID. The tester (or testing tool) is provided a means, via the SelectiveTestRunner, of retrieving the IDs assigned to test methods by the framework for use with prioritization.

4 Experiments

Our experiments address the following research questions:

RQ1: Can test case prioritization improve the rate of fault detection of JUnit test suites?

RQ2: How do the three types of information and information use that distinguish prioritization techniques (type of coverage information, use of feedback, use of modification information) impact the effectiveness of prioritization techniques?

RQ3: Can test suite granularity (the choice of running test-class level JUnit tests versus test-method level tests) impact the cost-effectiveness of prioritization techniques?

In addition to these research questions, we examine whether test case prioritization results obtained from systems written in an object-oriented language (Java) and using JUnit test cases show different trends than those obtained from systems written in a procedural language (C) using traditional coverage- or requirements-based system test cases.

To address our questions we designed several controlled experiments. The following subsections present, for these experiments, our objects of analysis, independent variables, dependent variables and measures, experiment setup and design, threats to validity, and data and analysis.

4.1 Objects of Analysis

We used four Java programs as objects of analysis: Ant, XML-security, JMeter, and JTopas. Ant is a Java-based build tool [4]; it is similar to make, but instead of being extended with shell-based commands it is extended using Java classes. JMeter is a Java desktop application designed to load test functional behavior and measure performance [5]. XML-security implements security standards for XML [6]. JTopas is a Java library used for parsing text data [7]. Several sequential versions of each of these systems were available and were selected for these experiments.

Table 1 lists, for each of our objects, "Versions" (the number of versions),"Size" (the number of classes), "Test Classes" (the number of JUnit test-class level tests), "Test Methods" (the number of JUnit test-method level tests), and "Faults" (the number of faults). The number of classes corresponds to the total number of class files in the final version. The numbers for test-class (test-method) list the number of test classes (test methods) in the most recent version. The number of faults indicates the total number of faults available for each of the objects (see Section 4.3.3).

Table 1. Experiment objects

Subjects	Versions	Size	Test Classes	Test Methods	Faults
Ant	9	627	150	877	21
XML-security	4	143	14	83	6
JMeter	6	389	28	78	9
JTopas	4	50	11	128	5

4.2 Variables and Measures

4.2.1 Independent Variables

Our experiments manipulated two independent variables: prioritization technique and test suite granularity.

Variable 1: Prioritization Technique

We consider nine different test case prioritization techniques, which we classify into three groups to match an earlier study on prioritization for C programs [15]. Table 2 summarizes these groups and techniques. The first group is the control group, containing three "techniques" that serve as experimental controls. (We use the term "technique" here as a convenience; in actuality, the control group does not involve any practical prioritization heuristics; rather, it involves various orderings against which practical heuristics should be be compared.) The second group is the block level group, containing two fine granularity prioritization techniques. The third group is the method level group, containing four coarse granularity prioritization techniques.

Control techniques

- No prioritization (untreated): One control that we consider is simply the application of no technique; this lets us consider "untreated" JUnit test suites.
- Random prioritization (random): As a second control we use random prioritization, in which we randomly order the tests in a JUnit test suite.
- Optimal prioritization (optimal): To measure the effects of prioritization techniques on rate of fault detection, our empirical study utilizes programs that contain known faults. For the purposes of experimentation, we can determine, for any test suite, which test cases expose which faults, and thus we can determine

Table 2. Test case prioritization techniques.

Label	Mnemonic	Description
T1	untreated	original ordering
T2	random	random ordering
T3	optimal	ordered to optimize rate of fault detection
T4	block-total	prioritize on coverage of block
T5	block-addtl	prioritize on coverage of block not yet covered
T6	method-total	prioritize on coverage of method
T7	method-addtl	prioritize on coverage of method not yet covered
T8	method-diff-total	prioritize on coverage of method and change information
T9	method-diff-addtl	prioritize on coverage of method/ change information, and adjusted on previous coverage

an optimal ordering of test cases in a JUnit test suite for maximizing that suite's rate of fault detection. In practice, of course, this is not a practical technique, but it provides an upper bound on the effectiveness of our heuristics.

Block level techniques

- Total block coverage prioritization (block-total): By instrumenting a program we can determine, for any test case, the number of basic blocks (single-entry, single-exit sequences of statements) in that program that are exercised by that test case. We can prioritize these test cases according to the total number of blocks they cover simply by sorting them in terms of that number.
- Additional block coverage prioritization (block-addtl): Additional block coverage prioritization combines *feedback* with coverage information. It iteratively selects a test case that yields the greatest block coverage, adjusts the coverage information on subsequent test cases to indicate their coverage of blocks not yet covered, and repeats this process until all blocks covered by at least one test case have been covered. If multiple test cases cover the same number of blocks not yet covered, they are ordered randomly. When all blocks have been covered, this process is repeated on the remaining test cases, until all have been ordered.

Method level techniques

- Total method coverage prioritization (method-total): Total method coverage prioritization is the same as total block coverage prioritization, except that it relies on coverage measured in terms of methods.
- Additional method coverage prioritization (method-addtl): Additional method coverage prioritization is the same as additional block coverage prioritization, except that it relies on coverage in terms of methods.
- Total diff method prioritization (method-diff-total): Total diff method coverage prioritization uses modification information; it sorts test cases in the order of their coverage of methods that differ textually (as measured by a Java parser that parses pairs of individual Java methods through the Unix "diff" function). If multiple test cases cover the same number of differing methods, they are ordered randomly.
- Additional diff method prioritization (method-diff-addtl): Additional diff method prioritization uses both feedback and modification information. It iteratively selects a test case that yields the greatest coverage of methods that differ, adjusts the coverage information on subsequent test cases to indicate their coverage of methods not yet covered, and then repeats this process until all methods that differ and that have been covered by at least one test case have been covered. If multiple test cases cover the same number of differing methods not yet covered, they are ordered randomly. This process is repeated on remaining test cases, until all test cases that execute methods that differ have been used; additional method coverage prioritization is applied to remaining test cases.

The foregoing set of techniques matches the set examined in [15] in all but two respects. First, we use three control techniques, considering an "untreated" technique in which test cases are run in the order in which they are given in the original JUnit tests. This is a sensible control technique for our study since in practice developers would run JUnit tests in their original ordering.

Second, the studies with C programs used statement and function level prioritization techniques, where coverage is based on source code, whereas our study uses coverage based on Java bytecode. Analysis at the bytecode level is appropriate for Java environments. Since Java is a platform independent language, vendors or programmers might choose to provide just class files to keep the source protected. If many such class files are part of a system under test, we want to be able to analyze even those class files that are provided without source; bytecode analysis allows this.

The choice to use bytecode level analysis does affect our choice of prioritization techniques. As an equivalent to C "function level" coverage, a method level granularity was an obvious choice. As a statement level equivalent, we could use either individual bytecode instructions, or basic blocks of instructions, but we cannot infer a one-to-one correspondence between Java source statements and either bytecode instructions or blocks.[3] We chose the basic block because the basic block representation is a more cost-effective unit of analysis for bytecode. Since basic blocks of bytecode and

[3] A Java source statement typically compiles to several bytecode instructions, and a basic block from bytecode often corresponds to more than one Java source code statement.

source code statements represent different levels of granularity, results obtained on the two are not directly comparable, but we can still study the effect that an increase in granularity has on prioritization in Java, as compared to in C.

Variable 2: Test Suite Granularity

To investigate the impact of test suite granularity on the effectiveness of test case prioritization techniques we considered two test suite granularity levels for JUnit tests: test-class level and test-method level as described in Section 3.

At the test-class level, JUnit tests are of relatively coarse granularity; each test suite and test-class that is invoked by a TestSuite is considered to be one test case, consisting of one or more test-methods. At the test-method level, JUnit tests are relatively fine granularity; each test-method is considered to be one test case.

4.2.2 Dependent Variables and Measures

Rate of Fault Detection

To investigate our research questions we need to measure the benefits of the various prioritization techniques in terms of rate of fault detection. To measure rate of fault detection, we use a metric, APFD (Average Percentage Faults Detected), introduced for this purpose in [15], that measures the weighted average of the percentage of faults detected over the life of a test suite. APFD values range from 0 to 100; higher numbers imply faster (better) fault detection rates. More formally, let T be a test suite containing n test cases, and let F be a set of m faults revealed by T. Let TF_i be the first test case in ordering T$'$ of T which reveals fault i. The APFD for test suite T$'$ is given by the equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{nm} + \frac{1}{2n}$$

Examples and empirical results illustrating the use of this metric are provided in [15].

4.3 Experiment Setup

To perform test case prioritization we required several types of data. Since the process used to collect this data is complicated and requires significant time and effort, we automated a large part of the experiment.

Figure 3 illustrates our experiment process. There were three types of data to be collected prior to applying prioritization techniques: coverage information, fault-matrices, and change information. We obtained coverage information by running tests over instrumented objects using the Galileo system for analysis of Java bytecode in conjunction with a special JUnit adaptor. This information lists which test

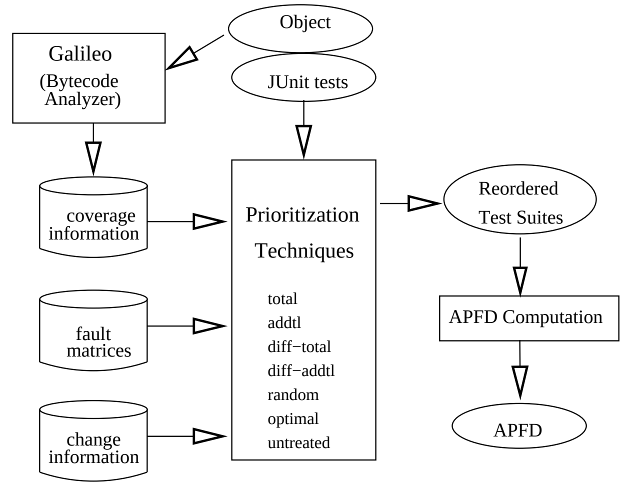

Figure 3. Overview of experiment process

cases exercised which blocks and methods; a previous version's coverage information is used to prioritize the current set of test cases. Fault-matrices list which test cases detect which faults, and are used to measure the rate of fault detection for each prioritization technique. Change information lists which methods differ from those in the preceding version, and how many lines of each such method changed (deleted and added methods are also listed.).

Each prioritization technique utilizes some or all of this data to prioritize JUnit test suites based on its analysis. Then APFD scores are obtained from the reordered test suites, and the collected scores are analyzed to determine whether the techniques improved the rate of fault detection.

4.3.1 Object Instrumentation

To perform our experiment, we required objects to be instrumented to support the techniques described in Section 4.2.1. We instrumented object class files in two ways: all basic blocks, and all method entry blocks, using the Galileo bytecode analysis system (see Figure 2).

4.3.2 Test Cases and Test Automation

As described previously, test cases were obtained from each object's distribution, and we considered test suites at the two levels of granularity previously described. To execute and validate test cases automatically, we created test scripts that invoke test cases, save all outputs from test execution, and compare outputs with those for the previous version. As shown in Figure 2, JUnit test cases are run through JUnitFilter and TestRunner (SelectiveTestRunner) over the instrumented classes.

4.3.3 Faults

We wished to evaluate the performance of prioritization techniques with respect to detection of regression faults. The object programs we obtained were not supplied with any such faults or fault data; thus, following the procedure described in [19], and also utilized for the study in [15], we seeded faults. Two graduate students performed this fault seeding; they were instructed to insert faults that were as realistic as possible based on their experience with real programs, and that involved code inserted into, or modified in each of the versions. We excluded any faults that were detected by more than 20% of the test cases for both granularity levels; our assumption is that such easily detected faults represent coding errors that competent programmers would not be expected to make in practice.

4.4 Threats to Validity

In this section we describe the internal, external, and construct threats to the validity of our experiments, and the approaches we used to limit the effects that these threats might have on the validity of our conclusions.

Internal Validity

The inferences we made about the effectiveness of prioritization techniques could have been affected by two factors. The first factor involves the potential faults in our experiment tools. To control for this threat, we validated tools through testing on various sizes of Java programs. The second factor involves the faults seeded in our objects. Our procedure for seeding faults followed a set process as described in Section 4.3.3, which reduced the chances of obtaining biased faults. However some of our objects (XML-security and JTopas) ultimately contained a relatively small number of faults, and larger numbers of faults would be preferable.

External Validity

Two issues limit the generalization of our results. The first issue involves object program representativeness. Our objects are of small and medium size. Complex industrial programs with different characteristics may be subject to different cost-benefit tradeoffs. The second issue involves testing process representativeness. If the testing process we used is not representative of industrial processes, our results might not generalize. Control for these threats can be achieved only through additional studies with a wider population.

Construct Validity

The dependent measure that we have considered, APFD, is not the only possible measure of rate of fault detection and it has some limitations. First, APFD assigns no value to subsequent test cases that detect a fault already detected; such inputs may, however, help debuggers isolate the fault, and for that reason might worth measuring. Second, APFD does not account for the possibility that faults and test cases may have different costs. Third, APFD only partially captures the aspects of the effectiveness of prioritization. Future studies will need to consider other measures for purposes of assessing effectiveness.

4.5 Data and Analysis

To provide an overview of all the collected data we present boxplots in Figure 4. The left side of the figure presents results from test case prioritization applied to test-class level tests, and the right side presents results from test case prioritization applied to test-method level tests. The top row presents results for an all programs total, and other rows present results per object program. Each plot contains a box showing the distribution of APFD scores for each of the nine techniques, across each of the versions of the object program. See Table 2 for a legend of the techniques.

The data sets depicted in Figure 4 served as the basis for our formal analyses of results. The following sections describe, for each of our research questions in turn, the experiments relevant to that question, presenting those analyses.

4.5.1 RQ1: Prioritization effectiveness

Our first research question considers whether test case prioritization can improve the rate of fault detection for JUnit test cases applied to our Java objects.

An initial indication of how each prioritization technique affected a JUnit test suite's rate of fault detection in this study can be determined from Figure 4. Comparing the boxplots of optimal (T3) to those for untreated (T1) and random (T2), it is readily apparent that an optimal prioritization order could provide substantial improvements in rates of fault detection. Comparing results for untreated (T1) to results for actual, non-control techniques (T4 through T9) for both test suite levels, it appears that all techniques yield improvement. However, the comparison of the results of random orderings (T2) with those produced by non-control techniques shows different results: some techniques yield improvement with respect to random while others do not.

To determine whether the differences observed in the boxplots are statistically significant we performed two sets of analyses, considering test-class and test-method levels independently for all programs. The analyses were:

1. UNTREATED vs NON-CONTROL: We consider untreated and non-control techniques to determine whether there is a difference between techniques (using ANOVA), and whether there is a difference between untreated orders and the orders produced by

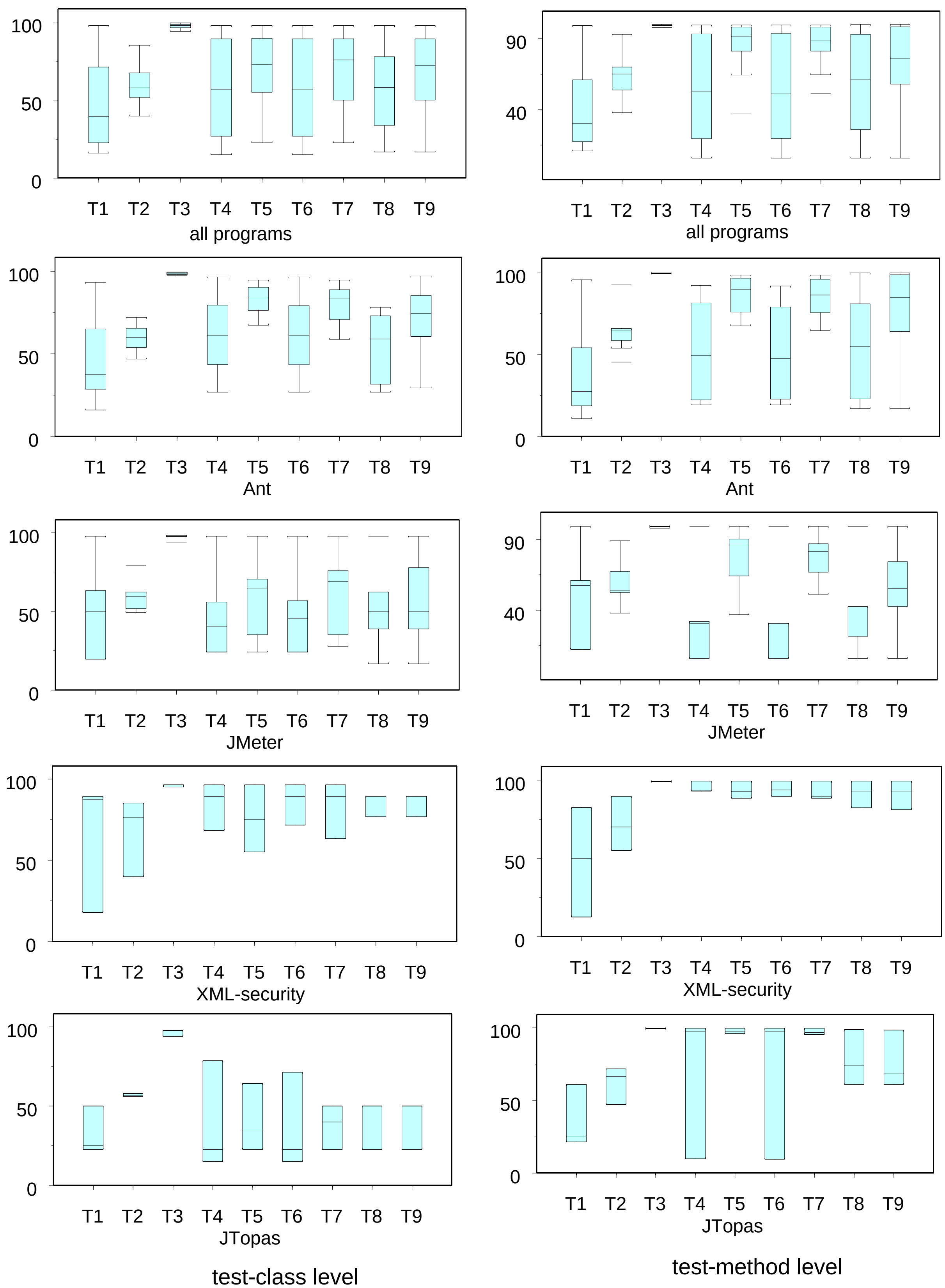

Figure 4. APFD boxplots, all programs. The horizontal axes list techniques, and the vertical axes list APFD scores. The left column presents results for test-class level tests and the right column presents results for test-method level tests.

each non-control technique (using Dunnett's method for multiple comparison with a control [8]).

2. RANDOM vs NON-CONTROL: The same analyses as in (1) are performed, against a different control technique, random.

Table 3 presents the results of analysis (1), for significance level of 0.05. The ANOVA results indicate that there is no difference between the techniques at test-class level (p-value = 0.0733), but there is a significant difference at test-method level (p-value < 0.0001). Dunnett's method reports that block-addtl and method-addtl are different from untreated at test-class level, and block-addtl, method-addtl, and method-diff-addtl are different from untreated at test-method level. Cases that were statistically significant are marked by "****" (which indicates confidence intervals that do not include zero).

test-class level					
Source	SS	d.f	MS	F-stat	p-value
Technique	6259	6	1043	1.98	0.073
Program	22663	3	7221	13.71	< 0.0001
Residuals	64776	123	526		
multiple comparison with a control by Dunnett's method					
critical point: 2.603					
	Estimate	Std. Err	Lower Bound	Upper Bound	
T4-T1	10.0	7.45	-9.34	29.4	
T5-T1	20.5	7.45	1.10	39.9	****
T6-T1	10.1	7.45	-9.30	29.5	
T7-T1	20.7	7.45	1.28	40.0	****
T8-T1	8.54	7.45	-10.8	27.9	
T9-T1	16.4	7.45	-2.94	35.8	
test-method level					
Source	SS	d.f	MS	F-stat	p-value
Technique	30991	6	5165	7.16	< 0.0001
Program	17171	3	5723	7.93	< 0.0001
Residuals	88730	123	721		
multiple comparison with a control by Dunnett's method					
critical point: 2.603					
	Estimate	Std. Err	Lower Bound	Upper Bound	
T4-T1	14.8	8.71	-7.84	37.5	
T5-T1	44.1	8.71	21.5	66.8	****
T6-T1	14.2	8.71	-8.5	36.9	
T7-T1	43.8	8.71	21.1	66.5	****
T8-T1	18.5	8.71	-4.14	41.2	
T9-T1	31.0	8.71	8.30	53.7	****

Table 3. ANOVA and Dunnett's method, untreated vs heuristics, all programs.

Table 4 presents the results of analysis (2). Similar to the first analysis, the ANOVA results indicate that there is no difference between the techniques at test-class level (p-value = 0.3687), but there is a significant difference at test-method level (p-value < 0.0001). Dunnett's method reports that no individual prioritization technique is different from random at test-class level, but block-addtl and method-addtl are different from random at test-method level.

test-class level					
Source	SS	d.f	MS	F-stat	p-value
Technique	2953	6	492	1.09	0.368
Program	19151	3	6383	14.2	< 0.0001
Residuals	55258	123	449		
multiple comparison with a control by Dunnett's method					
critical point: 2.603					
	Estimate	Std. Err	Lower Bound	Upper Bound	
T4-T2	-2.44	6.88	-20.3	15.5	
T5-T2	8.0	6.88	-9.9	25.9	
T6-T2	-2.39	6.88	-20.3	15.5	
T7-T2	8.19	6.88	-9.71	26.1	
T8-T2	-3.94	6.88	-21.8	14	
T9-T2	3.96	6.88	-13.9	21.9	
test-method level					
Source	SS	d.f	MS	F-stat	p-value
Technique	19029	6	3171	5.13	< 0.0001
Program	18851	3	6283	10.17	< 0.0001
Residuals	75973	123	617		
multiple comparison with a control by Dunnett's method					
critical point: 2.603					
	Estimate	Std. Err	Lower Bound	Upper Bound	
T4-T2	-6.97	8.06	-28.0	14.0	
T5-T2	22.3	8.06	1.34	43.3	****
T6-T2	-7.63	8.06	-28.6	13.4	
T7-T2	22.0	8.06	0.97	42.9	****
T8-T2	-3.28	8.06	-24.3	17.7	
T9-T2	9.16	8.06	-11.8	30.2	

Table 4. ANOVA and Dunnett's method, random vs heuristics, all programs.

4.5.2 RQ2: The effects of information types and use on prioritization results

.

Our second research question concerns whether differences in the types of information and information use that distinguish prioritization techniques (type of coverage information, use of feedback, type of modification information) impact the effectiveness of prioritization.

Comparing the boxplots (Figure 4) of block-total to method-total and block-addtl to method-addtl for both test suite levels, it appears that the level of coverage information utilized (fine vs coarse) has no effect on techniques. Comparing the results of block-total to block-addtl and method-total to method-addtl for both test suite levels, it appears that techniques using feedback do yield improvement over those not using feedback. In particular, the difference from using the test-method level is distinguishable from the test-class level.

Finally, comparison of the results of method-total to method-diff-total, and method-addtl to method-diff-addtl, shows no apparent effect from using modification information on prioritization effectiveness.

To determine whether the differences observed in the boxplots are statistically significant we performed an ANOVA and all-pairwise comparison using Tukey's method [8], considering test-class level and test-method level independently, for all non-control techniques and all programs. Table 5 presents the results of these analyses.

Overall, the results indicate that there is a significant difference between techniques at the test-method level (p-value = 0.00015); but no significant difference between techniques at the test-class level (p-value = 0.317).

The all-pairwise comparison shows the following results regarding information types and information use:

- Coverage information. The results indicate that there is no difference between block-level and method-level techniques at either test suite level, considering block-total versus method-total (T4 - T6) and block-addtl versus method-addtl (T5 - T7). Different levels of coverage information did not impact the effectiveness of prioritization; that is, there was no prioritization technique granularity effect observed.

- Use of Feedback. The results indicate a significant difference between techniques that use feedback and those that do not use feedback at the test-method level, namely block-total versus block-addtl (T4 - T5) and method-total versus method-addtl (T6 - T7). However, there was no significant difference at the test-class level. Techniques that use feedback information provided a more significant improvement in rate of fault detection than those that did not, but only at the finer granularity test level.

- Modification information. The results indicate no significant difference between techniques that use modification information and those that do not use modification information at either test suite levels, namely method-total versus method-diff-total (T6 - T8) and method-addtl versus method-diff-addtl (T7 - T9).

4.5.3 RQ3: Test suite granularity effects

Our third research question considers the impact of test suite granularity, comparing test-class level to test-method level. The boxplots and the analysis related to our first research question suggest that there is a difference between the two levels of test suite granularity, thus we performed an ANOVA for non-control techniques and all programs comparing test-method level to test-class level. Figure 6 presents the results, which indicate that there is suggestive evidence that test suite granularity affected the rate of fault detection (p-value = 0.0437).

test-class level					
Source	SS	d.f	MS	F-stat	p-value
Technique	2895	5	579	1.192	0.3177
Program	21071	3	7023	14.46	< 0.0001
residuals	50971	105	485		
all pair comparison by Tukey's method					
critical point: 2.903					
	Estimate	Std. Err	Lower Bound	Upper Bound	
T4-T5	-10.400	7.15	-31.20	10.3	
T4-T6	-0.046	7.15	-20.80	20.7	
T4-T7	-10.600	7.15	-31.40	10.1	
T4-T8	1.500	7.15	-19.30	22.3	
T4-T9	-6.400	7.15	-27.20	14.4	
T5-T6	10.400	7.15	-10.40	31.1	
T5-T7	-0.185	7.15	-20.90	20.6	
T5-T8	11.900	7.15	-8.81	32.7	
T5-T9	4.040	7.15	-16.70	24.8	
T6-T7	-10.600	7.15	-31.30	10.2	
T6-T8	1.550	7.15	-19.20	22.3	
T6-T9	-6.350	7.15	-27.10	14.4	
T7-T8	12.100	7.15	-8.63	32.9	
T7-T9	4.230	7.15	-16.50	25.0	
T8-T9	-7.900	7.15	-28.70	12.9	
test-method level					
Source	SS	d.f	MS	F-stat	p-value
Technique	18456	5	3691	5.5	0.00015
Program	20415	3	6805	10.14	< 0.0001
Residuals	70426	105	670		
all pair comparison by Tukey's method					
critical point: 2.903					
	Estimate	Std. Err	Lower Bound	Upper Bound	
T4-T5	-29.300	8.4	-53.700	-4.91	****
T4-T6	0.654	8.4	-23.700	25.00	
T4-T7	-28.900	8.4	-53.300	-4.54	****
T4-T8	-3.700	8.4	-28.100	20.70	
T4-T9	-16.100	8.4	-40.500	8.25	
T5-T6	30.000	8.4	5.560	54.40	****
T5-T7	0.370	8.4	-24.000	24.80	
T5-T8	25.600	8.4	1.210	50.00	****
T5-T9	13.200	8.4	-11.200	37.60	
T6-T7	-29.600	8.4	-54.000	-5.19	****
T6-T8	-4.350	8.4	-28.700	20.00	
T6-T9	-16.800	8.4	-41.200	7.60	
T7-T8	25.200	8.4	0.841	49.60	****
T7-T9	12.800	8.4	-11.600	37.20	
T8-T9	-12.400	8.4	-36.800	12.00	

Table 5. ANOVA and Tukey's method, all heuristics , all programs.

Source	SS	d.f	MS	F-stat	p-value
Suite	3408	1	3408	4.1047	0.0437
Residuals	234999	283	830		

Table 6. ANOVA analysis, test-class level vs test-method level, all programs.

5 Discussion

Our results strongly support the conclusion that test case prioritization techniques can improve the rate of fault detection for JUnit test suites applied to Java systems. The prioritization techniques we examined outperformed both untreated and randomly ordered test suites, as a whole, at the test-method level. Overall, at the test-class level, prioritization techniques did not improve effectiveness compared to untreated or randomly ordered test suites, but individual comparisons indicated that techniques using additional coverage information did improve the rate of fault detection.

We also observed that random test case orderings outperformed untreated test case orderings at both test suite levels. We conjecture that this difference is due to the construction of the JUnit tests supplied with the objects used in this study. It is typical in practice for developers to add new tests at the end of a test suite. Since newer test cases tend to exercise new code, we conjecture that these new test cases are more likely to be fault-revealing than previous test cases. Randomly ordered test cases redistribute fault-revealing test cases more evenly than original, untreated test case orders. Of course prioritization techniques, in turn, typically outperform random orderings. The practical implication of this result, then, is that the worst thing JUnit users can do is not practice some form of prioritization.

We now consider whether the results we have observed are consistent with those from the previous studies with C programs and coverage- or requirements-based test suites. Both this study (at the test-method level) and previous studies [13, 15, 24, 28] showed that prioritization techniques improved the rate of fault detection compared to both random and untreated orderings. Also, both this study and earlier studies found that techniques using additional coverage information were usually better than other techniques, for both fine and coarse granularity tests. There were sources of variance between the studies, however.

Regarding the impact of granularity, previous studies on C showed that statement-level techniques as a whole were better than function-level techniques. This study, however, found that block-level techniques were not significantly different overall from method-level techniques. There are two possible reasons for this difference. First, this result may be due to the fact that the instrumentation granularity we used for Java programs differs from that used for C programs, as we explained in Section 4.2.1. Block-level instrumentation is not as sensitive as statement-level instrumentation since a block may combine a sequence of consecutive statements into a single unit in a control flow graph.

A second related factor that might account for this result is that the instrumentation difference between blocks and methods is not as pronounced in Java as is the difference between statements and functions in C. Java methods tend to be more concise than C functions, possibly due to object-oriented language characteristics [20] and code refactoring [17], which tend to result in methods containing small numbers of blocks. For example, constructors, "get" methods, and "set" methods frequently contain only one basic block. A study reported in [23] supports this interpretation, providing evidence that the sizes of the methods called most frequently in object-oriented programs is between one and nine statements on average, which generally corresponds (in our measurements on our objects) to only one or two basic blocks.

Where test suite granularity effects are concerned, contrary to the previous study [24], this study showed that fine granularity test suites (test-method level) were more supportive of prioritization than coarse granularity test suites (test-class level). Although some techniques (block-addtl and method-addtl) at the test-class level improved rate of fault detection, overall analysis showed that test-class level test suites did not support improvements in APFD. Since the scope of each test case in a test-class level test suite is limited to a specific class under test,[4] one possible explanation for this result involves the fact that, in our objects, faults are located only in a few of the classes under test, and the number of faults in several cases is relatively small. The test-class level boxplots for the Ant program (Figure 4), which has a larger number of faults (21) overall, show a different trend from the other plots, supporting this conjecture.

6 Conclusions and Future Work

We have presented a study of prioritization techniques applied across four Java programs, to JUnit test suites provided with those programs. Although several studies of test case prioritization have been conducted previously, those studies all focused on a single procedural language, C, and on only a few specific types of test suites. Our study, in contrast, applied prioritization techniques to an object-oriented language (Java) tested under the JUnit testing framework, to investigate whether the results observed in previous studies generalize to other language and testing paradigms.

Our results regarding the effectiveness of prioritization techniques confirm several previous findings [13, 15, 24, 28], while also revealing some differences regarding prioritization technique granularity effects and test suite granularity effects. As discussed in Section 5, these differences can be explained in relation to characteristics of the Java language and JUnit testing.

The results of our studies suggest several avenues for future work. First, we intend to perform additional studies using larger Java programs and additional types of test suites,

[4] This is true in the objects that we considered; in general, however, the scope of a unit test could be subject to a developers' specific practices.

and a wider range and distribution of faults. Second, since our studies, unlike previous studies, did not find differences between prioritization technique granularity levels, it would be interesting to investigate what types of attributes caused this result through a further analysis of the relationship between basic block size and method size, and the ratio of execution of simple methods to complex methods. Third, contrary to our expectation, code modification information did not improve cost-effectiveness significantly. This result was also observed in C programs, and it is worth investigating what types of factors other than fault distribution in a code base could be involved in this outcome. Fourth, because our analysis revealed a sizable performance gap between prioritization heuristics and optimal prioritization, we are investigating alternative techniques. One such alternative involves employing measures of static and dynamic dependencies in the code to estimate locations where faults can reside.

Through the results reported in this paper, and this future work, we hope to provide software practitioners, in particular practitioners who use Java and the JUnit testing framework, with cost-effective techniques for improving regression testing processes through test case prioritization.

Acknowledgements

This work was supported in part by NSF under Awards, CCR-0080900 and CCR-0306023 to Oregon State University. Ziru Zhu helped prepare the JTopas object.

References

[1] http://www.junit.org.
[2] http://sourceforge.net.
[3] http://jakarta.apache.org.
[4] http://ant.apache.org.
[5] http://jakarta.apache.org/jmeter.
[6] http://xml.apache.org/security.
[7] http://jtopas.sourceforge.net/jtopas.
[8] Splus statistics software.
[9] T. Chen and M. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, Mar. 1996.
[10] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proc. Int'l. Conf. Softw. Eng.*, pages 211–220, May 1994.
[11] S. Elbaum, D. Gable, and G. Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proc. Int'l. Softw. Metrics Symp.*, pages 169–179, Apr. 2001.
[12] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *J. Softw. Testing, Verif., and Rel.*, 12(2), 2003.
[13] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proc. Int'l. Symp. Softw. Testing and Anal.*, pages 102–112, Aug. 2000.
[14] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proc. Int'l. Conf. Softw. Eng.*, pages 329–338, May 2001.
[15] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, Feb. 2002.
[16] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. Technical Report 03-01-01, University of Nebraska – Lincoln, Jan. 2003.
[17] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
[18] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. and Meth.*, 2(3):270–285, July 1993.
[19] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. Int'l. Conf. Softw. Eng.*, pages 191–200, May 1994.
[20] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation and evaluation of optimisations in a just-in-time compiler. In *ACM 1999 Java grande conf.*, pages 119–128, June 1999.
[21] J. Kim and A. Porter. A histroy-based test prioritization technique for regression testing in resource constrained environments. In *Proc. Int'l. Conf. Softw. Eng.*, May 2002.
[22] J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *Proc. Int'l. Conf. Testing Comp. Softw.*, pages 111–123, June 1995.
[23] J. Power and J. Waldron. A method-level analysis of object-oriented techniques in Java applications. Technical Report NUM-CS-TR-2002-07, National University of Ireland, July 2002.
[24] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proc. Int'l. Conf. Softw. Eng.*, May 2002.
[25] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. Technical Report 03-60-04, Oregon State University, Nov. 2003.
[26] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Meth.*, 6(2):173–210, Apr. 1997.
[27] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proc. Int'l. Conf. Softw. Maint.*, pages 179–188, Aug. 1999.
[28] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization. *IEEE Trans. Softw. Eng.*, 27(10), Oct. 2001.
[29] A. Srivastava and T. J. Effectively prioritizing tests in development environment. In *Proc. Int'l. Symp. Softw. Testing and Anal.*, July 2002.
[30] D. Wells. Extreme programming: A gentle introduction. http://www.extremeprogramming.org, Jan. 2003.
[31] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proc. Int'l. Symp. Softw. Rel. Engr.*, pages 230–238, Nov. 1997.