

Prioritizing JUnit Test Cases: An Empirical Assessment and Cost-Benefits Analysis

Hyunsook Do, Gregg Rothermel, Alex Kinnear
Computer Science and Engineering Department
University of Nebraska - Lincoln
{dohy,grother,akinnear}@cse.unl.edu

August 2, 2005

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 the extent to which these results generalize is an open question because the studies have all focused on a single procedural language, C, and a few specific types of test suites. In particular, Java and the JUnit testing framework are being used extensively to build software systems 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 the 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. To investigate the practical implications of these results, we present a set of cost-benefits models for test case prioritization, and show how the effectiveness differences observed can result in savings in practice, but vary substantially with the cost factors associated with particular testing processes.

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 great deal 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 [4, 29], test suite minimization [3, 14, 23], and test case prioritization [10, 30, 40].

Test case prioritization provides a way to run test cases that have the highest priority — according to some criterion — earliest in the testing cycle, 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 evaluated through empirical studies [6, 9, 10, 30, 31, 37, 40]. These studies have shown that several prioritization techniques can improve a test suite's rate of fault detection. Most of these studies, however, have focused on a single procedural language, C, and on a few specific types of test suites, so whether their 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 provide a more complete understanding of test case prioritization.

In this work, we set out to perform a replicated study of test case prioritization techniques, 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 a new testing paradigm, the JUnit testing framework, which is increasingly being used by developers to implement test cases for Java programs [20]. 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 [17, 35].

The JUnit framework is being used to support various approaches to testing. As one example, JUnit is being used extensively in the testing of Java systems constructed by traditional development methodologies, resulting in large banks of integration and system tests that are run *in batch* when a new system release is created. As the size of such systems and their banks of test suites grow, retest-all regression testing strategies can be excessively expensive. JUnit users will need methodologies with which to remedy this problem. As a second example, the JUnit framework is also being used to support a very different development approach, in which developers write test cases for classes and their interactions, and then are encouraged to rerun all of these test cases whenever they modify their code. This *incremental* testing approach supports approaches such as *nightly build-and-test* [24], in which modifications made during the day are tested that night, and *extreme programming* [38], in which small sets of tests are rerun following small changes, and is being investigated relative to *continuous testing* [32, 33, 34], in which tests are rerun in between program edits. Prior studies of prioritization have focused primarily on batch testing approaches, evaluating techniques relative to batch testing models; however, incremental approaches must also be considered.

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 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 that affect prioritization effectiveness. 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 practice, prioritization techniques have associated costs, and depending on the testing processes employed and other cost factors, may not reduce overall regression testing costs despite improvements in rates of fault detection. Thus, to investigate issues involving costs and the practical ramifications of our results, we consider our results relative to cost models for assessing cost-benefits of prioritization. Prior research [10, 22]

has presented such cost models; we extend these to consider the different batch and incremental testing approaches described above, and use the resulting models to consider the practical implications for prioritization cost-effectiveness across the different approaches. Our analysis in terms of these cost models lets us consider cases, relative to the data obtained on the particular programs and test suites that we studied, in which prioritization would or would not be cost-effective. The analysis provides further understanding of our particular data, but more important, illustrates how practitioners could apply cost models and prioritization data within their own organization to assess the cost-effectiveness of prioritization in that context.

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 our 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 our cost models and analysis of practical significance. Section 7 presents conclusions and future work.

2 Background and Related Work

2.1 Test Case Prioritization

Test case prioritization techniques [10, 30, 40] schedule test cases in an execution order according to some criterion. The test case prioritization problem is formally defined as follows [30]:

The Test Case Prioritization Problem:

Given: T , a test suite, PT , the set of permutations of T , and f , a function from PT to the reals.

Problem: Find $T' \in PT$ such that $(\forall T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$

In this definition, PT represents the set of all possible prioritizations (orderings) of T , and f is a function that, applied to any such ordering, yields an award value for that ordering.

The purpose of test case 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 employed. One way in which techniques can be distinguished involves the type of code coverage information they use. Test cases can be prioritized in terms of the number of 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 basic blocks (single-entry, single-exit sequences of statements) they covered, resolving ties randomly.

A second way in which prioritization 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 selecting the next 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 about 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 that have been suggested in the literature [9, 10, 21] include test cost estimates, fault severity estimates, estimates of fault propagation probability, test history information, and usage statistics obtained through operational profiles.

2.2 Previous Empirical Work

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 [10, 30, 31, 37, 40]. 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.

Recent studies of prioritization have begun to examine the factors affecting prioritization effectiveness [6, 7, 21, 27]. Kim et al. [21] present a “history-based” prioritization technique, in which information from previous regression testing cycles is used to better inform the selection of a subset of an existing test suite for use on a modified version of a system. Rothermel et al. [27] investigate the effects of test suite design 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 attempt to correlate attributes of change with technique performance, Elbaum et al. [6, 7] report on experiments exploring the effects of program structure, test suite composition, and changes on prioritization, and identify several metrics characterizing these attributes that correlate with prioritization effectiveness.

More recent studies have examined how some of these factors impact the effectiveness and efficiency of prioritization, and have considered the generalization of findings through controlled experiments [11, 28]. These studies expose tradeoffs and constraints that affect the success of techniques, and provide guidelines for designing and managing prioritization and testing processes. Elbaum et al. [11] provide insights into

what types of prioritization techniques are and are not appropriate under specific testing scenarios, and the conditions under which they are or are not appropriate. Further, Rothermel et al. [28] describe controlled experiments examining the effects of two factors in test suite composition: test suite granularity and test input grouping, and report on tradeoffs affecting the relationship between test suite design and regression testing cost-effectiveness.

All of this previous work 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 present 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.

Most recently, Saff and Ernst [32, 33, 34] consider test case prioritization for Java in the context of continuous testing, which uses spare CPU resources to continuously run regression tests in the background as a programmer codes. They propose combining the concepts of test frequency and test case prioritization, and report the results of a study that shows that prioritized continuous testing can reduce wasted development time. However, their prioritization techniques are based on different sources of information than ours, such as history of recent or frequent errors and test cost, rather than code coverage information.

3 JUnit Testing and Prioritization

JUnit testing involves 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 through 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.

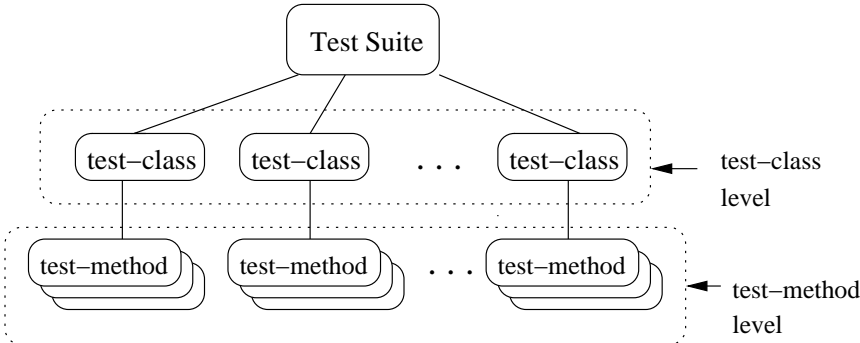


Figure 1: JUnit test suite structure

JUnit test classes can be run individually. Running individual test classes is reasonable for small programs, but for programs having large numbers of test classes it can be expensive, because each independent execution of a test class incurs startup costs. Thus in practice, developers design JUnit test suites that

invoke sequences of test classes.

Studies of C programs have shown that choices in *test suite granularity* (the number and size of the test cases making up a test suite) can affect the cost of executing test suites and the cost-effectiveness of prioritization techniques [28]. As the test hierarchy presented in Figure 1 shows, a test suite granularity choice exists for JUnit tests, as well: we can prioritize JUnit tests at the test-class level, or at the test-method level ignoring groupings in classes. (Prioritization at the test-method level assumes that test methods have no dependencies between one another and thus *can* be ordered arbitrarily, but this is typical practice in JUnit test design, and is also true of the programs that we study empirically in this work). We refer to these levels of focus as *coarse-granularity* and *fine-granularity*, respectively. Following [28], we refer to the minimal units of testing code that can be defined and executed independently at a particular level of focus as *test cases*. We then describe test cases at the coarse-granularity level of JUnit `TestClass` classes as *test-class level test cases*, and test cases at the fine-granularity level of JUnit methods within `TestClass` classes as *test-method level test cases*.

In this work, we wish to investigate the effects of fine- versus coarse-granularity choices in test design on JUnit prioritization, and to do this, we need to ensure that the JUnit framework allows us to achieve the following four objectives:

1. identify and execute each JUnit `TestCase` class individually;
2. reorder `TestCase` classes to produce a (test-class level) prioritized order;
3. identify and execute each individual test method within a JUnit `TestCase` class individually;
4. reorder test methods to produce a (test-method level) prioritized order.

Objectives 1 and 2 can be trivially achieved due to 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 is necessary only to extract the names of all `TestCase` classes invoked by the top level `TestSuite` for the object program¹ (a simple task) and then execute them individually with the JUnit test runner in a desired order.

Objectives 3 and 4 are 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 are executed when that `TestCase` is specified for execution, providing the ability to treat individual methods across a set of `TestCase` classes as test cases required us to extend the JUnit framework.

Since 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

¹This process is repeated iteratively if `Test Suites` are nested in other `Test Suites`.

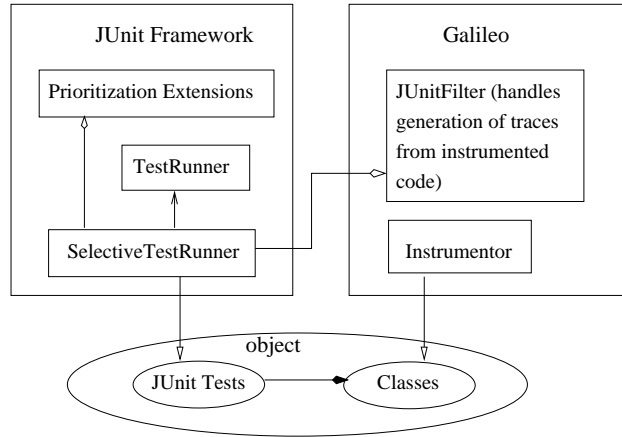


Figure 2: JUnit framework and Galileo

by the tester. We addressed this problem by subclassing various components of the framework and inserting mechanisms for assigning numeric test IDs to each test method identified. Within the JUnit framework, a single test executor class is responsible for the actual execution and collection of the outcomes from all JUnit test methods. To implement ordinal test selection, we introduced a new subclass of the test executor that accepts a list of selected test numbers and uses an internal counter to track the current test number. By checking the current test number against the list of selected tests prior to each test, the extended test executor can determine whether any given test is to be run. A new test runner (`SelectiveTestRunner`) instantiates an instance of the extended test executor in place of the standard test executor.

The relationship between our extensions and the existing JUnit framework is shown in Figure 2, which also shows how the JUnit framework is related to the Galileo² system [13] for analyzing Java bytecode (which we used to obtain coverage information for use in prioritization). Our new `SelectiveTestRunner` is able to access test cases individually using numeric test IDs.

To implement 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 to tests in the order of discovery, to execute the test cases in an order other than the one in which they are provided requires that all test cases be discovered prior to execution. We accomplished this by further extending the executor subclass to perform a simple two-pass technique. When the test executor is instructed to operate in prioritizing mode, the first time it is asked to execute a test suite, it simply passes through all of the test classes (and test methods) in that suite and creates a mapping from test numbers to corresponding test methods. The test runner then issues a special call back to the test executor, at which time the tests are retrieved from the map and executed in exactly the order in which the list of tests is provided to the test executor. In practice, this technique introduces minimal overhead because the first pass by the test executor is extremely fast.

²In subsequent versions, Galileo has been renamed Sofya.

4 Experiments

We wish to 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, and use of modification information) impact the effectiveness of prioritization techniques?

RQ3: Can test suite granularity (the choice of running test-class level versus test-method level JUnit test cases) impact the 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 reveal 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 [2]; 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 [18]. *Xml-security* implements security standards for XML [41]. *Jtopas* is a Java library used for parsing text data [19]. Several sequential versions of each of these programs were available and were selected for these experiments. All of these programs are publically available as part of an infrastructure supporting experimentation [5].

Table 1 lists, for each of our objects, “Versions” (the number of versions), “Classes” (the number of classes), “KLOCs” (the total number of lines of code, excluding comments), “Test Classes” (the number of JUnit test-class level test cases), “Test Methods” (the number of JUnit test-method level test cases), and “Faults” (the number of faults). The number of classes (KLOCs) corresponds to the total number of class files (lines of code) 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 (explained further in Section 4.3.3).

To help assess the representativeness of the objects we selected, we collected size metrics on a number of popular Open Source Java programs available through the SourceForge [35] and Apache Jakarta [17] project

web sites. Using all the projects available through Jakarta, and the Java projects in the top 50 most popular on SourceForge, we obtained 27 projects and measured the numbers of classes and lines of code in those programs. The programs ranged in size from 13 to 1168 classes, and from 2.9 to 157.9 KLOCs, containing 333 classes and 41.2 KLOCs on average. The objects we use in these experiments have a similar average size and range of sizes, and thus are similar in terms of sizes to a significant class of Java programs.

Table 1: Experiment objects

Objects	Versions	Classes	KLOCs	Test Classes	Test Methods	Faults
<i>ant</i>	9	627	80.4	150	877	21
<i>jmeter</i>	6	389	43.4	28	78	9
<i>xml-security</i>	4	143	16.3	14	83	6
<i>jtopas</i>	4	50	5.4	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 [10]. 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 compared.) The other two groups contain the (non-control) techniques that we wish to investigate: the second group is the block level group, containing two fine granularity prioritization techniques, and the third group is the method level group, containing four coarse granularity prioritization techniques.

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 blocks
T5	block-addtl	prioritize on coverage of blocks not yet covered
T6	method-total	prioritize on coverage of methods
T7	method-addtl	prioritize on coverage of methods not yet covered
T8	method-diff-total	prioritize on coverage of methods and change information
T9	method-diff-addtl	prioritize on coverage of methods/change information, and adjusted on previous coverage

Control techniques

- (T1) No prioritization (untreated): One control that we consider is simply the application of no technique; this lets us consider “untreated” JUnit test suites.
- (T2) Random prioritization (random): As a second control we use random prioritization, in which we randomly order the test cases in a JUnit test suite.
- (T3) Optimal prioritization (optimal): To measure the effects of prioritization techniques on rate of fault detection, our empirical study uses programs that contain known faults. For the purposes of experimentation we can determine, for any test suite, which test cases expose which faults, and using this information we can determine an (approximate) optimal ordering of test cases in a JUnit test suite for maximizing that suite’s rate of fault detection. This is not a viable practical technique, but it provides an upper bound on the effectiveness of prioritization heuristics.

Block level techniques

- (T4) Total block coverage prioritization (block-total): By instrumenting a program we can determine, for any test case, the number of basic blocks 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 (and resolving ties randomly).
- (T5) 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

- (T6) Total method coverage prioritization (method-total): Total method coverage prioritization is the same as total block coverage prioritization, except that it relies on coverage in terms of methods.
- (T7) 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.
- (T8) Total diff method coverage 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 extracts pairs of individual Java methods and passes them through the Unix “diff” function). If multiple test cases cover the same number of differing methods, they are ordered randomly.

- (T9) Additional diff method coverage prioritization (method-diff-addtl): Additional diff method coverage prioritization uses both feedback and modification information. It iteratively selects a test case that yields the greatest coverage of methods that differ, adjusts the 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 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 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 [10] 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 test cases. This is a sensible control technique for our study since in practice developers would run JUnit test cases in their original ordering.

Second, the studies with C programs used statement and function level prioritization techniques, where coverage was 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 for system components. In such cases we want to be able to analyze even those class files, and bytecode analysis allows this.

The use of 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. Although basic blocks of bytecode and source code statements represent different types of code entries we can still study the effect that an increase in granularity (from basic blocks to methods) 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 test cases: test-class level and test-method level as described in Section 3. At the test-class level, JUnit test cases are of relatively coarse granularity; each 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 test cases are relatively fine granularity; each test-method is considered to be one test case.

³A 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.

4.2.2 Dependent Variables and Measures

Rate of Fault Detection

To investigate our research questions we need to measure the benefits of 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 [10], 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 that reveals fault i . The APFD for test suite T' is given by the equation:

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

We illustrate this metric using an example reproduced from [10]. Consider a program with a test suite of ten test cases, A through J, such that the program contains eight faults detected by those test cases, as shown by the table in Figure 3.a.

Consider two orders of these test cases, order $T1$: **A-B-C-D-E-F-G-H-I-J**, and order $T2$: **I-J-E-B-C-D-F-G-H-A**. Figures 3.b and 3.c show the percentages of faults detected versus the fraction of the test suite used, for these two orders, respectively. The areas inside the inscribed rectangles (dashed boxes) represent the weighted percentage of faults detected over the corresponding fraction of the test suite. The solid lines connecting the corners of the inscribed rectangles interpolate the gain in the percentage of detected faults. The area under the curve thus represents the weighted average of the percentage of faults detected over the life of the test suite.

On test order $T1$ (Figure 3.b) the first test case executed (**A**) detects no faults, but after running test case **B**, two of the eight faults are detected; thus 25% of the faults have been detected after 0.2 of test order $T1$ has been used. After running test case **C**, one more fault is detected and thus 37.5% of the faults have been detected after 0.3 of the test order has been used. Test order $T2$ (Figure 3.c), in contrast, is a much “faster detecting” test order than $T1$: the first 0.1 of the test order detects 62.5% of the faults, and the first 0.3 of the test order detects 100%. ($T2$ is in fact an optimal ordering of the test suite, resulting in the earliest detection of the most faults.) The resulting APFDs for the two test case orders are 43.75% and 90.0%, respectively.

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.

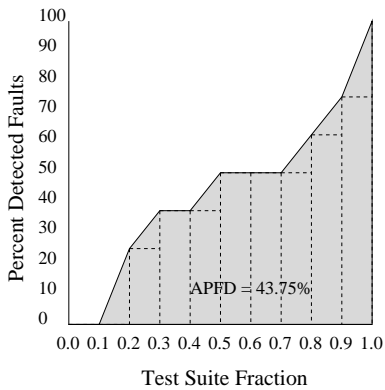
Figure 4 illustrates our experiment process. There were three types of data to be collected prior to applying prioritization techniques: coverage information, fault-matrices, and change information, as follows:

test	fault							
	1	2	3	4	5	6	7	8
A								
B	x	x						
C	x	x	x					
D		x	x					
E	x							x
F								x
G		x						
H				x				x
I	x	x	x	x	x			
J					x	x	x	

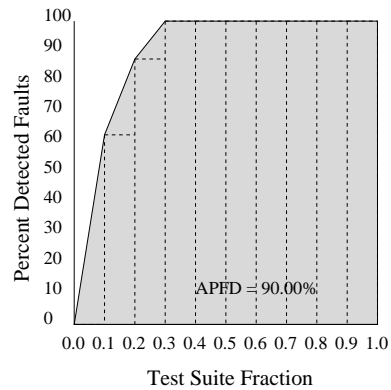
(a) Test suite and faults exposed

Test Case Order T1: A-B-C-D-E-F-G-H-I-J

Test Case Order T2: I-J-E-B-C-D-F-G-H-A



(b) APFD for test case order T1



(c) APFD for test case order T2

Figure 3: Example illustrating the APFD measure.

- We obtained coverage information by running test cases over instrumented object programs using the Galileo system for analysis of Java bytecode in conjunction with a special JUnit adaptor. This information lists which test 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. Since the optimal technique needs to know which test cases expose which faults in advance to determine an optimal ordering of test cases, it uses fault-matrices when the prioritization technique is applied.
- Change information lists which methods differ from those in the preceding version and how many lines of each method were changed (deleted and added methods are also listed). This information is used when method-diff-total (T8) and method-diff-addtl (T9) techniques are applied.

Each prioritization technique uses some or all of this data to prioritize JUnit test suites based on its

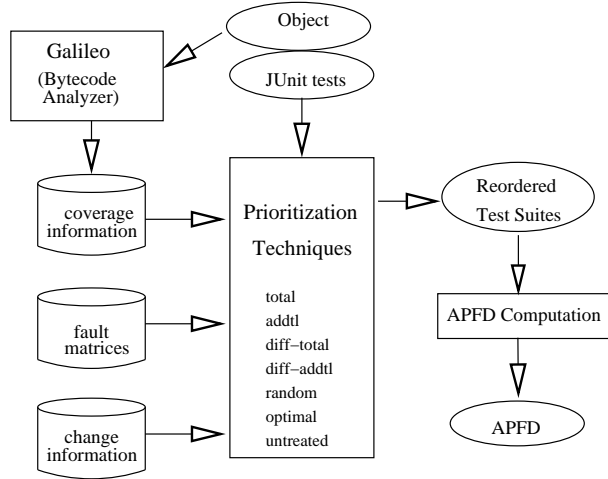


Figure 4: Overview of experiment process

analysis; then APFD scores are obtained from the reordered test suites. The collected scores are analyzed to determine whether the techniques improved the rate of fault detection. For random orderings, we generated random orders from untreated test cases 20 times per version for each object program, collected APFD values from those orderings, and calculated the average of these APFD values.

4.3.1 Object Instrumentation

To perform our experiment, we required that programs be instrumented to support the techniques described in Section 4.2.1. We instrumented class files in two ways: all basic blocks, and all method entry blocks (prior to the first instruction of the method), 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 program’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 JUnit test cases (at the test-class or test-method level), 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 because this produces coverage data for subsequent prioritizations.

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. To obtain faults, we considered two approaches: mutation and fault seeding. The first approach would allow us to generate a large number of faults, but these faults may not be representative of real faults.⁴ The second

⁴A recent study by Andrews et al. [1] suggests otherwise, but its results pertain only to a particular set of C mutants.

approach cannot produce a large number of faults in a cost-effective manner, but it can generate realistic faults. Therefore we chose the second approach, and following the procedure described in [15], used also in the study described in [10], 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. To be consistent with previous studies of C programs to which we wish to compare our results, we excluded any faults that were detected by more than 20% of the test cases at both granularity levels.

4.4 Threats to Validity

In this section we describe the external, internal, and construct threats to the validity of our experiments, and the approaches we used to limit the effects that these threats might have.⁵

External Validity

Three issues limit the generalization of our results. The first issue is object program representativeness. Our programs are of small and medium size, and other industrial programs with different characteristics may be subject to different cost-benefit tradeoffs. The second issue involves testing process representativeness. The test suites our object programs possess are only one type of test suite that could be created by practitioners. On the other hand, as Section 4.1 points out, our programs and test suites do represent an important class of such objects found in practice. The third issue for generalization involves fault representativeness. We used seeded faults that were as realistic as possible, but they were not actual faults found “in the wild”. Future studies will need to consider other fault types.

Internal Validity

The inferences we have made about the effectiveness of prioritization techniques could have been affected by two factors. The first factor involves 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 object programs. Our procedure for seeding faults followed a set process as described in Section 4.3.3, which as mentioned above reduced the chances of obtaining biased faults. Some of our object programs, however (*xml-security* and *jtopas*) ultimately contained a relatively small number of faults, and this could affect the inferences we are able to draw.

Construct Validity

The dependent measure that we have considered, APFD, is not the only possible measure of prioritization effectiveness and has some limitations. For example, 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 be worth measuring. Also, APFD does not account for the possibility that faults and test cases

⁵For a discussion of types of threats to validity, see [39].

may have different costs. Third, APFD focuses on all faults, considering the value added by detecting each fault earlier, but in some situations, engineers may be concerned solely with the first fault detected by a test suite. Future studies will need to consider other measures of effectiveness, such as the cost of analysis, execution, result checking, and test suite maintenance.

4.5 Data and Analysis

To provide an overview of all the collected data we present boxplots in Figure 5. The left side of the figure presents results from test case prioritization applied to test-class level test cases, and the right side presents results from test case prioritization applied to test-method level test cases. Each row presents results per object program. Each plot contains a box for each of the nine prioritization techniques, showing the distribution of APFD scores for that technique, across each of the versions of the object program. (See Table 2 for a legend of the techniques.) Because we apply each technique to each pair of successive versions of each program, the number of data points represented by the boxes for a given program is equal to the number of versions of that program minus 1 (thus, 8 for *ant*, 3 for *xml-security*, 5 for *jmeter*, and 3 for *jtopas*). To further facilitate interpretation, Table 3 presents the mean value and standard deviation of the APFD scores for each boxplot that appears in the figure.

The data sets depicted in Figure 5 served as the basis for our analyses of results. The following sections describe, for each of our research questions in turn, the experiments relevant to that question, and the associated 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 programs.

An initial indication of how each prioritization technique affected our JUnit test suite’s rates of fault detection in this study can be obtained from Figure 5. Comparing the boxplots of untreated (T1) and random (T2) to the boxplot for optimal (T3), it is apparent on each object program and test suite level that they are not close to an optimal prioritization order’s rate of fault detection. Comparing the boxplots of

Table 3: Mean value and standard deviation (SD) of APFD scores in Figure 5.

Objects	Mean & SD	Test-class level									Test-method level								
		T1	T2	T3	T4	T5	T6	T7	T8	T9	T1	T2	T3	T4	T5	T6	T7	T8	T9
<i>ant</i>	Mean	46	59	98	57	83	57	79	59	76	38	64	99	52	87	51	84	54	76
	SD	26	8	1	27	8	27	12	26	22	28	13	1	30	11	29	12	31	28
<i>jmeter</i>	Mean	54	64	97	57	65	57	65	59	62	48	60	99	34	74	34	77	42	55
	SD	34	12	2	37	33	37	33	36	35	37	19	1	38	24	38	18	35	35
<i>xml-sec.</i>	Mean	62	67	95	92	86	95	95	91	91	48	71	99	96	96	97	87	96	96
	SD	43	24	1	3	11	2	2	4	4	34	17	1	3	3	3	12	3	3
<i>jtopas</i>	Mean	32	56	94	41	40	36	37	40	40	35	61	99	68	97	68	97	77	75
	SD	15	1	1	38	21	30	13	15	15	21	12	1	50	2	51	2	19	19

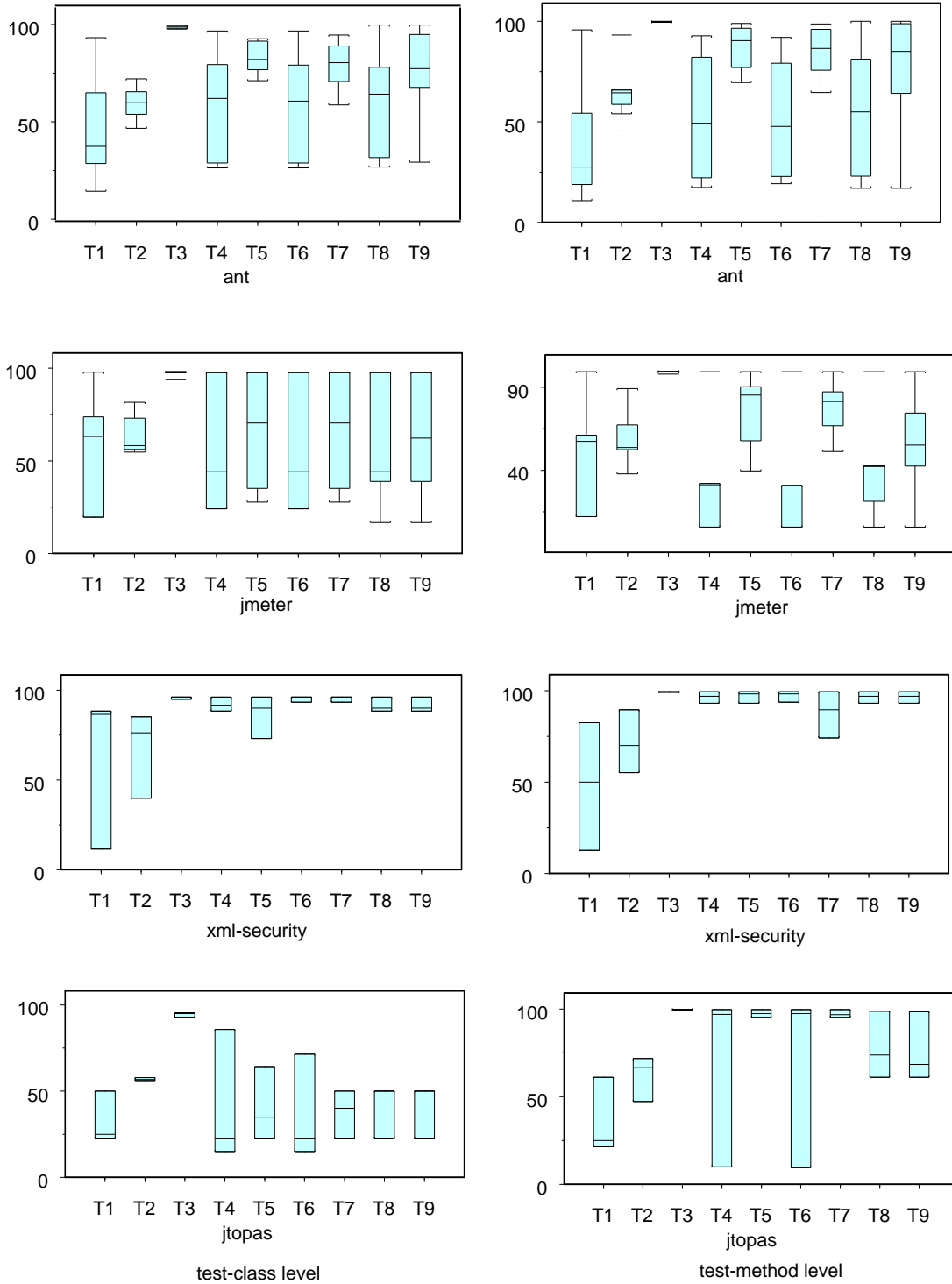


Figure 5: APFD boxplots, all programs. The horizontal axes list techniques, and the vertical axes list APFD scores. The left column presents results for test-class level test cases and the right column presents results for test-method level test cases. See Table 2 for a legend of the techniques.

untreated (T1) to those of random (T2), it appears that random test case orderings outperform untreated test case orderings at the test-method level on all programs other than *jmeter*, and on two of the four programs (*ant* and *jtopas*) at the test-class level. Comparing results for untreated (T1) to results for actual, non-control techniques (T4 – T9) for both test suite levels, it appears that all non-control techniques yield improvement for *ant* and *xml-security* at both test suite levels. However, the results from *jmeter* and *jtopas* show that some techniques yield improvement with respect to untreated while others do not. The comparison of the results of random orderings (T2) with those produced by non-control techniques shows varying results across all programs. For instance, the results from *ant* show that all non-control techniques yield improvement at the test-class level, but at the test-method level, some techniques yield improvement while others do not.

To determine whether the differences observed in the boxplots are statistically significant we performed three sets of analyses, considering test-class and test-method levels independently, for *ant*. (We performed a formal analysis only for *ant* because the other programs provide too few data points to support such an analysis. Formal analyses in RQ2 and RQ3 are also performed only for *ant* for the same reason.) The analyses were:

1. UNTREATED (T1) vs NON-CONTROL (T4-T9): We consider untreated and non-control techniques to determine whether there is a difference between untreated and each non-control technique. We use the Wilcoxon Rank-Sum nonparametric [26] test because the variability between techniques is very different and distributions for some techniques are not normal. We used the Splus statistics package [36] to perform the analysis.
2. RANDOM (T2) vs NON-CONTROL (T4-T9): We perform the same analyses as in (1), against the random ordering.
3. UNTREATED (T1) vs RANDOM (T2): We consider untreated and random techniques to determine whether there is a difference between the two techniques using the Wilcoxon Rank-Sum test.

Table 4 presents the results of analysis (1), for a significance level of 0.05. The Wilcoxon Rank-Sum test⁶ reports that block-addtl (T5), method-addtl (T7), and method-diff-addtl (T9) are significantly different from untreated (T1) at both test suite levels.

Table 5 presents the results of analysis (2). Similar to the first analysis, the Wilcoxon Rank-Sum test reports that block-addtl (T5), method-addtl (T7), and method-diff-addtl (T9) are significantly different from random (T2) at the test-class level, and block-addtl and method-addtl are significantly different from untreated at the test-method level.

For the comparison between untreated and random, the Wilcoxon Rank-Sum test reports that there is no difference between the two test orderings at the test-class level (two-sided p-value = 0.1949), but there is a significant difference at the test-method level (two-sided p-value = 0.0281).

⁶The Wilcoxon Rank-Sum test used here reports a two-sided p-value.

Table 4: Wilcoxon Rank-Sum Test, untreated (T1) vs non-control techniques (T4 – T9), *ant*.

Test-class level						
	untreated vs block-total	untreated vs block-addtl	untreated vs method-total	untreated vs method-addtl	untreated vs method-diff-total	untreated vs method-diff-addtl
p-value	0.5992	0.0148	0.5992	0.02	0.4	0.0379
Test-method level						
	untreated vs block-total	untreated vs block-addtl	untreated vs method-total	untreated vs method-addtl	untreated vs method-diff-total	untreated vs method-diff-addtl
p-value	0.5054	0.0047	0.4418	0.0047	0.3442	0.0379

Table 5: Wilcoxon Rank-Sum Test, random (T2) vs non-control techniques (T4 – T9), *ant*.

Test-class level						
	random vs block-total	random vs block-addtl	random vs method-total	random vs method-addtl	random vs method-diff-total	random vs method-diff-addtl
p-value	0.9591	0.0003	0.9691	0.0047	0.7984	0.0281
Test-method level						
	random vs block-total	random vs block-addtl	random vs method-total	random vs method-addtl	random vs method-diff-total	random vs method-diff-addtl
p-value	0.3282	0.003	0.3282	0.0104	0.5054	0.1049

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 results of block-total (T4) to method-total (T6) and block-addtl (T5) to method-addtl (T7) for both test suite levels for all programs, it appears that the level of coverage information utilized (fine vs coarse) has no effect on prioritization results. Comparing the results of block-total to block-addtl and method-total to method-addtl at both test suite levels for *ant* and *jmeter*, it appears that techniques using feedback outperform those not using feedback. However, in the case of *xml-security* and *jtopas*, no differences between those techniques are apparent. Finally, comparison of the results of method-total (T6) to method-diff-total (T8) and method-addtl (T7) to method-diff-addtl (T9) suggests no apparent effect from using modification information on prioritization effectiveness for all programs.

To determine whether the differences observed in the boxplots are statistically significant we compared each pair of non-control techniques using a Wilcoxon Rank-Sum test, considering test-class level and test-method level independently, for *ant*. Table 6 presents the results of these analyses.

The results show the following with respect to information types and information use:

- Coverage information. The results indicate that there is no difference between comparable block-level and method-level techniques at either test suite level, considering block-total (T4) versus method-total

Table 6: Wilcoxon Rank-Sum Test, all non-control techniques (T4 – T9), *ant*.

Test-class level						
	block-total vs block-addtl	method-total vs method-addtl	block-total vs method-total	block-addtl vs method-addtl	method-total vs method-diff-total	method-addtl vs method-diff-addtl
p-value	0.0499	0.1304	0.9161	0.7525	0.7525	1
Test-method level						
	block-total vs block-addtl	method-total vs method-addtl	block-total vs method-total	block-addtl vs method-addtl	method-total vs method-diff-total	method-addtl vs method-diff-addtl
p-value	0.0281	0.0281	1	0.7909	0.8335	1

(T6) and block-addtl (T5) versus method-addtl (T7). Different levels of coverage information did not impact the effectiveness of prioritization.

- 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 (T4) versus block-addtl (T5) and method-total (T6) versus method-addtl (T7). At the test-class level, there is a significant difference between block-total and block-addtl, but not between method-total and method-addtl.
- Modification information. The results indicate no significant difference between techniques that use modification information and those that do not use modification information, namely method-total (T6) versus method-diff-total (T8) and method-addtl (T7) versus method-diff-addtl (T9), at either test suite level.

4.5.3 RQ3: Test suite granularity effects

Our third research question considers the impact of test suite granularity, comparing test-class level test suites to test-method level test suites. The boxplots and the analysis related to our first two research questions suggest that there is a difference between the two levels of test suite granularity, thus we performed Wilcoxon Rank-Sum tests for non-control techniques and *ant* comparing test-method level to test-class level. Table 7 presents the results of these analyses. The results indicate that test suite granularity did not affect the rate of fault detection for all non-control techniques.

Table 7: Wilcoxon Rank-Sum Test, all non-control techniques (T4 – T9), test-class level vs test-method level, *ant*.

	block-total	block-addtl	method-total	method-addtl	method-diff-total	method-diff-addtl
p-value	0.5054	0.4418	0.4619	0.2786	0.5737	0.8785

5 Discussion

Our results support the conclusion that some of test case prioritization techniques can improve the rate of fault detection for JUnit test suites applied to Java programs.⁷ Considering results from *ant*, at both test suite levels, there were prioritization techniques that outperformed both untreated and randomly ordered test suites.

We also observed that random test case orderings outperformed untreated test case orderings at the test-method level. We conjecture that this difference is due to the construction of the JUnit test cases supplied with the programs used in this study. It is typical in practice for developers to add new test cases at the end of a test suite. Since newer test cases tend to exercise new code, these new test cases may be more likely to be fault-revealing than previous test cases. Randomly ordered test cases, on average, achieve orders that do not put the new test cases later. The practical implication of this result, then, is that the worst thing JUnit users can do is leave their test suites untreated.

To further investigate the foregoing effect and conjecture, we measured the rate of fault detection achieved by reversing the order of test cases supplied with the programs studied, and our results were supportive of the conjecture. The average APFDs for all programs were 82 for test-method level and 75 for test-class level, much better than the original untreated order (and better, even, than random order). Thus, a minimalist approach to improving JUnit testing in future versions of JUnit might involve offering an option by which its execution mechanism can be reversed.

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 [8, 10, 27, 28, 30] showed that prioritization techniques can improve the rate of fault detection of test suites compared to 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 test cases. There were some sources of variation 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 found, however, that block-level techniques were not significantly different overall from method-level techniques. 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 because a block may combine a sequence of consecutive statements into a single unit in a control flow graph.

A related explanatory factor 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

⁷Although we applied formal analyses only to *ant*, examination of the boxplots for other programs suggest that results on these programs are consistent with this conclusion.

concise than C functions, possibly due to object-oriented language characteristics [16] and code refactoring [12], which tend to result in methods that contain small numbers of blocks. Constructors, “get” methods, and “set” methods are examples of methods that frequently contain only one basic block. A study reported in [25] supports this interpretation, providing evidence that the sizes of the methods called most frequently in object-oriented programs are between one and nine statements on average, which generally corresponds (in our measurements on our object programs) to only one or two basic blocks.

To further investigate this result, we measured the number of instrumentation points per method in the four Java programs used in this study and the number of instrumentation points per function in 13 C programs studied in other prioritization research [10, 11]. Tables 8 and 9 show the resulting data for the initial version of each program. Columns 2 through 7 present the number of methods or functions that contain the listed range of instrumentation points. The additional parenthetical number denotes the percentage of the total number of methods (functions) that contain that number of points. In *ant*, for example, 609 methods contain zero or one instrumentation point, and in *printtokens*, four functions contain three instrumentation points. As the data shows, the majority of Java methods tend to contain a small number of instrumentation points (in all cases, over 46% contain two or fewer), whereas C functions tend to include larger numbers (4-9, 10-19, and 20+).

Table 8: Instrumentation points per method in Java

Objects	Blocks					
	0-1	2	3	4-9	10-19	20+
<i>ant</i>	609 (37.99%)	305 (19.00%)	95 (5.92%)	353 (22.00%)	141 (8.80%)	100 (6.29%)
<i>jmeter</i>	403 (27.13%)	421 (28.35%)	140 (9.42%)	341 (22.96%)	134 (9.02%)	46 (3.12%)
<i>xml-security</i>	398 (19.26%)	563 (27.25%)	202 (9.77%)	598 (28.94%)	202 (9.77%)	103 (5.01%)
<i>jtopas</i>	62 (31.00%)	48 (24.00%)	18 (9.00%)	48 (24.00%)	16 (8.00%)	8 (4.00%)

Table 9: Instrumentation points per function in C

Objects	Nodes					
	0-1	2	3	4-9	10-19	20+
<i>printtokens</i>	0 (0.00%)	0 (0.00%)	4 (22.22%)	7 (38.89%)	5 (27.78%)	2 (11.11%)
<i>printtokens2</i>	0 (0.00%)	2 (10.53%)	3 (15.79%)	7 (36.84%)	3 (15.79%)	4 (21.05%)
<i>replace</i>	2 (9.50%)	4 (19.10%)	0 (0.00%)	4 (19.10%)	8 (38.10%)	3 (14.20%)
<i>schedule</i>	0 (0.00%)	1 (5.60%)	2 (11.10%)	11 (61.10%)	3 (16.70%)	1 (5.50%)
<i>schedule2</i>	1 (6.25%)	0 (0.00%)	0 (0.00%)	10 (62.50%)	4 (25.00%)	1 (6.25%)
<i>tcas</i>	4 (44.45%)	0 (0.00%)	0 (0.00%)	3 (33.33%)	1 (11.11%)	1 (11.11%)
<i>totinfo</i>	0 (0.00%)	1 (14.29%)	1 (14.29%)	0 (0.00%)	3 (42.85%)	2 (28.57%)
<i>space</i>	1 (0.74%)	7 (5.15%)	7 (5.15%)	17 (12.50%)	44 (32.35%)	60 (44.11%)
<i>flex</i>	1 (0.70%)	15 (10.80%)	8 (5.80%)	45 (32.37%)	36 (25.90%)	34 (24.43%)
<i>grep</i>	14 (10.60%)	10 (7.60%)	3 (2.27%)	40 (30.30%)	27 (20.45%)	38 (28.78%)
<i>gzip</i>	1 (1.20%)	4 (4.80%)	1 (1.20%)	28 (33.73%)	13 (15.70%)	39 (43.37%)
<i>make</i>	10 (5.37%)	15 (8.00%)	6 (3.20%)	42 (22.58%)	41 (22.00%)	72 (38.85%)
<i>sed</i>	2 (2.70%)	10 (13.70%)	3 (4.10%)	29 (39.70%)	11 (15.00%)	18 (24.80%)

Because instrumentation at the method level is less expensive than instrumentation at the basic-block level, if these results generalize, they suggest that practitioners may not lose much, when working with Java programs, by selecting method-level rather than block-level instrumentation approaches.

Where test suite granularity effects are concerned, an early study of C programs [27] revealed no differences between fine and coarse granularity prioritization techniques, but a subsequent study [28] found that fine granularity could increase prioritization effectiveness compared to coarse granularity. Like the early C study, this study also revealed no difference between fine and coarse granularity prioritization techniques. Since the scope of each test case in a test-class level test suite is limited to a specific class under test,⁸ one possible explanation for this result is that, in our object programs, faults are located only in a few of the classes under test, and the number of faults in several cases is relatively small.

6 Cost-Benefits Analysis

Our results show that there can be differences in the rates of fault detection achieved by various test case prioritization techniques applied to Java programs and JUnit test cases. However, the improvements in the rates of fault detection demonstrated by certain techniques do not guarantee the *practical cost-effectiveness* of those techniques, because the techniques also have associated costs, and because the benefits of early fault detection can vary. In general, detecting faults earlier may have practical advantages only when the ratio of benefits of early detection relative to prioritization costs is sufficiently high [10].

In this section, we provide further understanding of the implications of our empirical results for practice, via a cost-benefits analysis accounting for the factors that affect the costs and benefits of the prioritization techniques considered. To do this we require (1) cost models that capture the important factors, and (2) test process models that consider the different settings in which prioritization may be performed. In the next subsection, we present such models. Then, in Section 6.2, we (1) present a general strategy for analyzing the cost-effectiveness of prioritization techniques based on the models, and (2) an analysis of the specific data obtained from our study.

6.1 Prioritization Cost Models

Malishevsky et al. [22] and Elbaum et al. [10] present cost models and methodologies for assessing the practical cost-benefits of prioritization techniques, considering the factors affecting those costs and benefits. In this section we review and build on that work.

Let P be a program, let P' be a modified version of P , let T be the test suite for P , and consider the application of prioritization techniques relative to P and P' . We define the following variables related to the cost of applying prioritization to P , P' , and T :

⁸This 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.

$Ca(T)$ is the cost of analysis needed to support prioritization.

$Cp(T)$ is the cost of using a prioritization technique itself.

$C(T) = Ca(T) + Cp(T)$ is the total cost associated with the use of a prioritization technique.

In the foregoing equations, $Cp(T)$ varies with the prioritization technique used; for example, techniques that use feedback perform more computation than techniques that do not. $Ca(T)$ varies with technique, as well, but its constituent costs can include the costs of program instrumentation (Ca_{inst}), analysis of changes between old and new versions ($Ca_{changes}$), collection of execution traces (Ca_{traces}), and other non-test dependent activities (Ca_{nt}). For example, the analysis cost for a prioritization technique that uses modification information can include all four of these constituent costs:

$$Ca(T) = Ca_{inst} + Ca_{traces} + Ca_{changes} + Ca_{nt}.$$

In practice, however, the constituent costs that are practically significant for a technique in a given context depend on the regression testing process that is in use. Whereas Malishevsky et al. [22] consider only one regression testing process (batch), we consider two: a batch process and an incremental process, as follows.

6.1.1 Batch Process

In a batch regression testing process, a period of software maintenance lasting several days, weeks, or months occurs until a release is ready for test, and then regression testing begins. In this process, we distinguish two different phases of regression testing - the preliminary and critical phases - which correspond to the times before and after the release is available for testing. Prioritization costs are distributed across these phases.

Preliminary phase costs

Preliminary phase costs are incurred as maintenance proceeds. During this phase, analysis required for prioritization for the upcoming release can be accomplished with the following costs:

$$Ca(T) = Ca_{inst} + Ca_{traces} + Ca_{nt}$$

The cost for prioritization in this phase, $C_{prelim}(T)$, is then:

$$C_{prelim}(T) = Ca(T) = Ca_{inst} + Ca_{traces} + Ca_{nt}$$

Critical phase costs

Critical phase costs, in contrast to preliminary phase costs, occur in the time period designated specifically for regression testing. At this point, product development is essentially complete. In this phase, the primary cost incurred is the cost of prioritization ($Cp(T)$), but depending on the prioritization technique applied, additional analysis costs, such as the cost of analysis of code modifications required for the method-diff and method-diff-addtl techniques, may also be incurred. Thus, the phase involves the following costs:

$$Ca(T) = Ca_{changes}$$

$$C_{crit}(T) = Ca(T) + Cp(T) = Ca_{changes} + Cp(T)$$

6.1.2 Incremental Process

A batch process may not suit typical development practices associated with the use of JUnit test cases, and thus we also consider an incremental process. In the incremental process, no distinction is made between preliminary and critical phases because regression testing is performed very frequently (iteratively) and thus the testing and product development phases cannot reasonably be treated separately. In this case all prioritization costs are incurred in each testing step, and may be characterized as follows (modulo adjustments in analysis cost constituents for specific prioritization techniques):

$$Ca(T) = Ca_{inst} + Ca_{traces} + Ca_{changes} + Ca_{nt}$$

$$C(T) = Ca(T) + Cp(T) = Ca_{inst} + Ca_{traces} + Ca_{changes} + Ca_{nt} + Cp(T)$$

6.1.3 Discussion of Processes and Models

Depending on which regression testing process an organization uses, the cost model associated with prioritization differs. Under the batch process, the data needed for prioritization techniques (with the exception of Ca_{change}) can potentially be collected automatically in the preliminary phase, with little or no effect on release time. Prioritization techniques can then be applied during the critical phase with relatively low analysis costs. Under the incremental process, all analysis costs can affect release time; thus, it may not be appropriate to apply stronger (but more expensive) techniques.

Note further that the incremental model covers a wide range of development and testing processes that are currently in use or under study. For example, many software organizations use nightly build-and-test processes [24], in which modifications made during the day are tested overnight, all directed toward a product release which may itself then be tested under a batch process. More recently, Extreme Programming [38] processes advocate testing following every modification or small set of related modifications, at a rate much more frequent than nightly. Finally, researchers have also recently begun to investigate continuous testing [32, 33, 34], in which tests are run in the time periods (potentially just a few minutes or seconds long) between program edits.

Although other regression testing processes could be considered, the two that we have described represent two ends of a spectrum that exist in practice. It might also be beneficial in practice to use a combined approach, such as employing weaker (but less expensive) prioritization techniques during periods in which testing is frequent, and employing stronger prioritization techniques during the major release testing cycles. The analyses that we perform in the rest of this section can be applied to such alternative processes, as well.

6.1.4 Quantifying Costs

To use the foregoing cost models to assess prioritization cost-effectiveness, we need to quantify the costs involved. In general, where improving the rate of fault detection is the goal, the decision to use a certain prioritization technique depends on the benefits of discovering faults sooner versus the cost of the technique itself. One procedure for savings quantification is to translate the cumulative cost of waiting for each fault to be exposed while executing test suite T under order O , defined as $delays^o$ by Malishevsky et al. [22], to a meaningful value scale based on the assessment of the benefits of detecting faults early.

Given an order O of test suite T , where T contains n test cases and detects m faults, $delays^o$ is defined as follows:

$$delays^o = \sum_{i=1}^m \left(\left(\sum_{k=1}^{TF_i^o} e_k^o \right) \times f_i \right)$$

In this equation, TF_i^o is the test number under order O that first detects fault i , e_k^o is the cost associated with the time required to run and validate test k in suite T under order O , and f_i is the cost of waiting a unit of time for a fault i to be exposed.

The cost savings due to the application of a prioritization technique that creates order O'' at cost $C(T)$, relative to some other test order O' obtained at no cost (i.e. original test order), is:

$$delays^{o'} - delays^{o''} - C(T).$$

From this equation, we can state that the prioritization technique producing test order O'' is cost-effective compared to some original test order O' if and only if:

$$C(T) < delays^{o'} - delays^{o''}$$

When comparing two prioritization techniques 1 and 2, with costs $C_1(T)$ and $C_2(T)$, respectively, and that produce test orders O_1 and O_2 , respectively, technique 1 is cost-beneficial with respect to technique 2 if and only if:

$$C_1(T) - C_2(T) < delays^{o_2} - delays^{o_1} \quad (1)$$

Note that as presented, this cost model is “unit-less”: the delays measure is defined in terms of the costs associated with test execution and waiting for faults to be exposed. In a particular application, these costs must be defined by the organization in a manner appropriate to their situation. We further consider this issue in the following sections.

6.2 Applying the Models to Our Data

To better understand our empirical data in light of the foregoing models, we apply those models to that data and discuss their implications for the use of prioritization in that context.

Our prioritization tools and the analysis tools on which they depend are prototypes, and are not implemented for efficiency. Thus, measurements of their run-time costs are not indicative of the potential run-times of such techniques in practice. Moreover, the (delays) variables on the right hand sides of our formulas for assessing the cost savings due to prioritization techniques involve many organization-specific factors, such as testers' salaries, the impact of a late release of a system, or the reliability expectations for the system. A direct comparison of such cost-related data as could be gathered in our study supports only a limited examination of tradeoffs.

We thus begin our investigation of implications by using an approach that is more general than the use of specific data allows. This approach, used originally in [10], uses savings factors to establish relationships between delays and a savings scale. The approach also provide a general method by which practitioners could assess prioritization cost-effectiveness in their particular application setting.

Following this more general illustration, we then consider an analysis of our particular data.

6.2.1 General Analysis

Let A and B be prioritization techniques with costs $C_A(T)$ and $C_B(T)$, respectively, let the test orders produced by A and B be O_A and O_B , respectively, and let the *delays* associated with A and B be $delays^{O_A}$ and $delays^{O_B}$, respectively. The decision to use technique A rather than B can be framed as one of determining whether $C_A(T) - C_B(T) < delays^{O_B} - delays^{O_A}$. Even when we do not have specific cost values for the techniques (left hand side), we can complete the right hand side of this equation to compute potential savings; the result constitutes an upper bound on possible cost savings that could be achieved given favorable conditions. Furthermore, the resulting analysis applies for either of the cost models presented in Section 6.1.

Recall that our “delays” measure is unit-less, and to be applied in practice must be translated into a concrete measure of benefit. A savings factor (SF) is a weight that translates a reduction in delays into such a measure of benefit. For example, it can associate benefits such as time or dollars saved with reductions in delays. The greater the SF, the greater the benefits generated by a reduction in delays.

By applying a range of scaling factors to a specific difference in delays values between two test orders A and B , we can obtain a view of the potential payoffs associated with that range of savings factors. This view can help us assess the break-even point at which A (and the technique that produced it) truly becomes more cost-effective than B (and the technique that produced it).

Tables 10, 11, and 12 show the results of applying this approach to our data, presenting (i) comparisons between untreated test suites and test suites ordered by heuristics, (ii) comparisons between randomly ordered test suites and test suites ordered by heuristics, and (iii) comparisons between various heuristics,

respectively, for each of the four object programs considered in our studies. Comparisons are made at the test-method level, in which overall improvements in rates of fault detection were observed. For comparisons (i) and (ii), we consider only those heuristics for which statistically significant differences were reported in Tables 4, 5 and 6. (Although, for programs other than *ant*, we do not have any statistical evidence of significant differences involving those techniques, we investigate them to see if their results have any practical implications.) For comparisons between heuristics we consider two cases that showed statistically significant differences: block-total vs block-addtl, and method-total vs method-addtl.

Columns 2 and 3 of Tables 10, 11 and 12 present the *delays* values calculated for our techniques using the formula for calculating delays presented in Section 6.1.4, averaged across the versions of the object programs, and column 4 presents the differences between those average *delays*. Note that in this usage, these delays values are calculated relative to our test case orders and fault detection information in which the values concerned are test indices, not times; thus, these delays values measure delays solely in terms of numbers of tests spent waiting for faults to be revealed.

Columns 5 through 11 of the tables show the potential savings that result from these differences in delays for each object program, for each of seven SFs: 1, 5, 10, 50, 100, 500, 1000. Here, the values in the table represent the delays values, which in this case have been calculated in terms of numbers of tests spent waiting to reveal faults, multiplied by scaling factors, but these scaling factors can be thought of as assigning more meaningful cost-benefit values to the differences in delays. For example, a scaling factor may be taken to represent the savings, in dollars, associated with the removal of one unit. In such cases, for *ant*, in Table 10, if a one unit difference in delays is worth one dollar to the organization, then block-addtl prioritization saves \$445.7 in comparison to an untreated order, and if a one unit difference in delays is worth \$1000, it saves the organization \$445,760.

As we would expect, given that we have used only test indices in calculating delay values, the data is similar to the APFD differences observed for the techniques: with the exception of the results on *xml-security* in Table 12, all results favor the technique in the third column over the technique in the second column. That is, Tables 10 and 11 show that the prioritization techniques considered are better than untreated and random techniques. Table 12 shows that techniques using feedback information perform better than techniques without feedback information. In particular *ant*, which has the largest number of classes and tests among our object programs, has large weighted *delays* values on all comparisons, (For *xml-security*, which has negative values in Table 12, the superiority of method-total holds since the cost of method-total is less than that of method-addtl.)

More important, the use of various scaling factors in the tables lets us examine the extent to which these differences in *delays* might translate, in practice, into savings that are greater than the additional cost incurred by prioritization techniques (or than the costs incurred by the superior prioritization technique in the case of comparison between two prioritization techniques). For sufficiently large SF values, prioritization

Table 10: Comparisons: untreated vs. heuristics

Objects	Delays			SF*(delays(T1)-delays(T5))						
	untreated (T1)	block-addtl (T5)	difference	1	5	10	50	100	500	1000
<i>ant</i>	694.3	248.5	445.7	445.7	2228.8	4457	22288	44576	222880	445760
<i>jmeter</i>	57.7	26.7	31.0	31.0	155.0	310	1550	3100	15500	31000
<i>xml-security</i>	81.6	8.0	73.6	73.6	368.3	736	3683	7360	36830	73600
<i>jtopas</i>	121.1	5.8	115.3	115.3	576.5	1153	5765	11530	57650	115300

Objects	Delays			SF*(delays(T1)-delays(T7))						
	untreated (T1)	method-addtl (T7)	difference	1	5	10	50	100	500	1000
<i>ant</i>	694.3	267.8	426.5	426.5	2132.5	4265	21325	42650	213250	426500
<i>jmeter</i>	57.7	23.4	34.3	34.3	171.5	343	1715	3430	17150	34300
<i>xml-security</i>	81.6	18.3	63.3	63.3	316.5	633	3165	6330	31650	63300
<i>jtopas</i>	121.1	5.8	115.3	115.3	576.5	1153	5765	11530	57650	115300

Objects	Delays			SF*(delays(T1)-delays(T9))						
	untreated (T1)	method-diff-addtl (T9)	difference	1	5	10	50	100	500	1000
<i>ant</i>	694.3	338.8	355.5	355.5	1777.5	3555	17775	35550	177750	355500
<i>jmeter</i>	57.7	45.5	12.2	12.2	61.0	122	610	1220	6100	12200
<i>xml-security</i>	81.6	8.3	73.3	73.3	366.5	733	3665	7366	36650	73300
<i>jtopas</i>	121.1	39.5	81.6	81.6	408.3	816	4083	8160	40830	81600

Table 11: Comparisons: random vs. heuristics

Objects	Delays			SF*(delays(T2)-delays(T5))						
	random (T2)	block-addtl (T5)	difference	1	5	10	50	100	500	1000
<i>ant</i>	444.2	248.5	195.7	195.7	978.5	1957	9785	19570	97850	195700
<i>jmeter</i>	45.1	26.7	18.4	18.4	92.0	184	920	1840	9200	18400
<i>xml-security</i>	46.8	8	38.8	38.8	194.0	388	1940	3880	19400	38800
<i>jtopas</i>	67.3	5.8	61.5	61.5	307.5	615	3075	6150	30750	61500

Objects	Delays			SF*(delays(T2)-delays(T7))						
	random (T2)	method-addtl (T7)	difference	1	5	10	50	100	500	1000
<i>ant</i>	444.2	267.8	176.4	176.4	882.0	1764	8820	17640	88200	176400
<i>jmeter</i>	45.1	23.4	21.7	21.7	108.5	217	1085	2170	10850	21700
<i>xml-security</i>	46.8	18.3	28.5	28.5	142.5	285	1425	2850	14250	28500
<i>jtopas</i>	67.3	5.8	61.5	61.5	307.5	615	3075	6150	30750	61500

Table 12: Comparisons: between heuristics

Objects	Delays			SF*(delays(T4)-delays(T5))						
	block-total (T4)	block-addtl (T5)	difference	1	5	10	50	100	500	1000
<i>ant</i>	521.4	248.5	272.9	272.9	1364.5	2729	13645	27290	136450	272900
<i>jmeter</i>	82.0	26.7	55.3	55.3	276.5	553	2765	5530	27650	55300
<i>xml-security</i>	8.3	8.0	0.3	0.3	1.5	3	15	30	150	300
<i>jtopas</i>	77.8	5.8	72.0	72.0	360.0	720	3600	7200	36000	72000

Objects	Delays			SF*(delays(T6)-delays(T7))						
	method-total (T6)	method-addtl (T7)	difference	1	5	10	50	100	500	1000
<i>ant</i>	530.9	267.8	263.1	263.1	1315.5	2631	13155	26310	131550	263100
<i>jmeter</i>	82.2	23.4	58.8	58.8	294.0	588	2940	5880	29400	58800
<i>xml-security</i>	7.3	18.3	-11.0	-11.0	-55.0	-110	-550	-1100	-5500	-11000
<i>jtopas</i>	77.8	5.8	72.0	72.0	360.0	720	3600	7200	36000	72000

techniques may yield large reductions in delays (and corresponding savings) even for a program like *jmeter*, on which, considering untreated versus method-diff-addtl prioritization (Table 10), a small difference in *delays* (12.2) translates into 12200 when SF is 1000. If SF is small, however, even large differences in *delays* may not translate into savings; for example, on *ant*, an SF of 1 yields a difference of only 445.7 for block-addtl vs untreated (Table 10).

As an illustration, we use the analysis strategy just presented to assess the possible practical implications of our results on the last two versions of *ant*, the largest of our four object programs and the one with the largest number of test cases. Using the cost models from Section 6.1, we measured the costs of applying the method-addtl technique (run-time, in seconds). The costs were: Ca_{inst} : 1156; Ca_{traces} : 589; $Cp(T)$: 1566.⁹ Given these numbers, under the batch process, the costs of prioritization in the preliminary and critical phases are 1745 and 1566 seconds, respectively. Under the incremental process, the cost is 3311 seconds. If we compare these costs with the unweighted difference in *delays* gained by method-addtl, they would not be cost-beneficial, because the unweighted *delays* differences between control orders and method-addtl are less than the cost incurred by the method-addtl prioritization technique: 426.5 in comparison with untreated, and 176.4 in comparison with random. However, somewhere between SF 3 and SF 8, the benefits of method-addtl reach a “break even point” in comparison to untreated orders – a point at which they begin to pay off. For the batch regression testing process with its lower cost (1566, because its preliminary costs can be ignored) the break even point is at $SF = 3.7$, whereas for the incremental process with cost 3311, the break even point is at $SF = 7.7$. As SF grows higher, possible savings increase; if an organization’s estimated SF is 1000, where SF is a measure of the time (in seconds) saved by a one unit difference in *delays*, then the savings gained from the method-addtl technique would be considerably higher (426,500), and thus the costs incurred by the prioritization technique would not be detrimental to savings. Moreover, given prioritization tools that are optimized for industrial use, the cost of prioritization would have less impact even for low values of SFs.

6.2.2 Analysis Relative to Specific Data

The foregoing analysis shows, in general terms, the degree to which certain prioritization techniques might be effective relative to the objects we studied if certain cost-benefits relationships (as captured abstractly by appropriate scaling factors) were to hold. Whether these particular relationships would in fact hold in practice for the particular objects we studied and under a particular testing process, however, is an open question. The test suites for the object programs we studied are not particularly long-running, and for many development and testing processes (such as batch testing, or overnight incremental testing) could not be cost-effectively prioritized. For shorter testing cycles, however, gains from prioritization are at least theoretically possible even for these suites. To investigate whether this possibility holds for our objects, as well as to show how the analysis strategy we have presented can be applied in practice, we next turn to

⁹ Ca_{nt} was too small to be measured, and thus, can be omitted.

Table 13: Costs and savings data for prioritization on our object programs, considering block-addtl and method-addtl techniques versus untreated test orders and test-method level test suites.

Objects	Avg. run time per test (sec) (AVG)	Delay differences		AVG * delay differences (sec)		Prioritization time (sec)	
		untreated vs block-addtl (D1)	untreated vs method-addtl (D2)	AVG * D1	AVG * D2	block-addtl (T5)	method-addtl (T7)
<i>ant</i> (v1-v2)	0.30	90.4	90.4	27.12	27.12	9.56	10.42
<i>ant</i> (v3-v4)	0.59	143.9	88.4	84.90	52.15	46.15	51.22
<i>ant</i> (v5-v7)	0.78	560.0	601.6	436.80	469.24	447.46	465.31
<i>ant</i> (v8)	0.64	1416.0	1248.0	905.24	798.72	1460.42	1566.94
<i>jmeter</i>	0.37	31.0	34.3	11.47	12.69	5.92	6.89
<i>xml-security</i>	0.42	73.6	63.3	30.91	26.58	8.82	8.74
<i>jtopas</i>	0.87	115.3	115.3	100.31	100.31	1.21	1.24

our specific data and examine it relative to the incremental testing process in which testing increments are shortest: continuous testing [32, 33, 34].

We consider two prioritization techniques that resulted in differences (statistically significant on *ant*) in prioritization results, block-addtl and method-addtl, and compare them to untreated test case orderings, at the test-method level. In our analysis, we consider three of our object programs (*jmeter*, *xml-security*, and *jtopas*) in terms of average costs across versions, because these programs possess tests suites that remain relatively constant across versions. For *ant*, whose test suites vary considerably in size, we examine averages across four subsets of versions that possess similar test suite sizes: v1-v2 (around 100 test cases), v3-v4 (around 200 test cases), v5-v7 (around 500 test cases), and v8 (877 test cases).

Table 13 presents, for each program or subset of *ant* versions, the average time required to execute a test case for that program or subset of versions (column 2), and the delays differences calculated for that program or subset of versions for the comparisons of block-addtl and method-addtl to untreated orders, respectively (columns 3 and 4). The time was measured on a machine running SuSE Linux 9.1 with 1G RAM and with a 3 GHZ processor. Columns 5 and 6 translate delays values into actual delays differences in seconds – effectively using the average time required to execute a test as the scaling factor introduced in Section 6.2.1, multiplying that number by the delays value. Columns 7 and 8 present the average time required to prioritize the test cases for each program or subset of versions, using the block-addtl and method-addtl techniques, respectively.

Given this data, we can assess the cost-benefits of prioritization during continuous testing as follows. Using formula (1) from Section 6.1.4, we know that a prioritization heuristic is more cost-effective than the untreated ordering if and only if the following constraint is satisfied:¹⁰

$$\text{prioritization time} < \text{AVG} * \text{delay_differences}$$

¹⁰On the left side of the equation, the cost of prioritizing for the untreated order is zero, and we assume that the development environment includes facilities for instrumenting code during compilation, and executes tests on that instrumented code regardless of the prioritization method used; thus, the costs of instrumentation for the prioritization heuristic and the untreated order are equivalent and cancel out. Note that using time as the component by which delays are translated into values makes sense in the context of continuous testing, where testing is vying with short edit cycles for attention. Also, another measure of interest in this context might be the delay until exposure of a first fault, which could be of particular interest in the continuous testing context. We retain our focus on delays, however, as they capture more fully the potential benefits of a complete prioritized order.

With respect to our data, this constraint is satisfied for *jmeter*, *xml-security*, and *jtopas* when considering the block-addtl and method-addtl heuristics. Among these cases, the largest savings are observed for *jtopas*: 99.1 and 99.07 seconds for block-addtl and method-addtl, respectively. Results on *ant* vary across subsets of versions. On the first and second subsets (v1-v2 and v3-v4) prioritization produces some savings: 17.56 and 16.7 seconds for block-addtl and method-addtl, respectively, on the first subset, and 38.75 and 0.93 seconds on the second. However, the third and fourth subsets do not result in savings; for these, the costs of applying prioritization exceeds the potential reduction in delays. (Again, our prioritization tools are just non-optimized prototypes, and in practice we expect that industrial-strength implementations could perform better.)

In assessing the implications of these results, it is important to recognize the iterative nature of the continuous testing process. In their empirical study of continuous testing [32], Saff et al.'s subjects utilized, on average, 266 and 116 individual test runs on the two programs they considered. In such situations, even small savings on individual test runs such as those we see on some of our programs can be magnified across the project cycle. For instance, in a development cycle of 266 iterations, the 99.1 second savings per *jtopas* test run would translate to 26361 seconds (438 minutes) across the entire cycle.

7 Conclusions and Future Work

We have presented a study of prioritization techniques applied to JUnit test suites provided with four Java programs. Although several studies of test case prioritization have been conducted previously, most have 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 [8, 10, 27, 28, 30], while also revealing some differences regarding effects of prioritization technique granularity and test suite granularity. As discussed in Section 5, these differences can be explained in relation to characteristics of the Java language and JUnit testing paradigm. We also investigated the practical impact of our results with respect to differences in *delays* values, relating these to the many cost factors involved in regression testing and prioritization processes, and illustrating the approach with respect to our particular data, relative to a continuous testing process.

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, a wider range and distribution of faults, and different types of faults, such as faults seeded by mutation. Second, contrary to our expectations, 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 responsible for this outcome. Third, because our analysis revealed a sizable performance gap between prioritization heuristics and optimal orders, we are investigating alternative techniques.

Through the results reported in this paper, and 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

Steve Kachman of the University of Nebraska - Lincoln Statistics Department provided assistance with our statistical analysis. This work was supported in part by NSF under Awards, CCF-0080900 and CCF-0306023 to Oregon State University, and CNS-0454203 to University of Nebraska - Lincoln. Zirui Zhu helped prepare the *jtopas* object. We thank the anonymous reviewers for suggestions that materially improved the paper.

References

- [1] 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*, pages 402–411, May 2005.
- [2] <http://ant.apache.org>.
- [3] T.Y. Chen and M.F. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, March 1996.
- [4] Y.F. Chen, D.S. Rosenblum, and K.P. Vo. TestTube: A system for selective regression testing. In *Proceedings of the International Conference on Software Engineering*, pages 211–220, May 1994.
- [5] H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 60–70, August 2004.
- [6] S. Elbaum, D. Gable, and G. Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings of the International Software Metrics Symposium*, pages 169–179, April 2001.
- [7] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Journal of Software Testing, Verification and Reliability*, 12(2), 2003.
- [8] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112, August 2000.

- [9] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the International Conference on Software Engineering*, pages 329–338, May 2001.
- [10] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.
- [11] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3), 2004.
- [12] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
- [13] <http://csce.unl.edu/~galileo/pub/galileo>.
- [14] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.
- [15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering*, pages 191–200, May 1994.
- [16] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Sukanuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation and evaluation of optimisations in a just-in-time compiler. In *ACM 1999 Java Grande Conference*, pages 119–128, June 1999.
- [17] <http://jakarta.apache.org>.
- [18] <http://jakarta.apache.org/jmeter>.
- [19] <http://jtopas.sourceforge.net/jtopas>.
- [20] <http://www.junit.org>.
- [21] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering*, pages 119–129, May 2002.
- [22] A. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Proceedings of the International Conference on Software Maintenance*, pages 204–213, November 2002.
- [23] J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of the International Conference on Testing Computer Software*, pages 111–123, June 1995.

- [24] K. Onoma, W-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, May 1988.
- [25] 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.
- [26] F. L. Ramsey and D. W. Schafer. *The Statistical Sleuth: A Course in Methods of Data Analysis*. Duxbury Press, Belmont, CA, 1st edition, 1997.
- [27] 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 *Proceedings of the International Conference on Software Engineering*, pages 230–240, May 2002.
- [28] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering and Methodology*, 13(3):277–331, July 2004.
- [29] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.
- [30] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization. *IEEE Transactions on Software Engineering*, 27(10), October 2001.
- [31] G. Rothermel, R.H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188, August 1999.
- [32] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 281–292, November 2003.
- [33] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 76–85, July 2004.
- [34] D. Saff and M. D. Ernst. Continuous testing in Eclipse. In *Proceedings of the 2nd Eclipse Technology Exchange Workshop*, March 2004.
- [35] <http://sourceforge.net>.
- [36] <http://www.insightful.com/products/splus>.
- [37] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceeding of the International Symposium on Software Testing and Analysis*, July 2002.

- [38] D. Wells. Extreme Programming: A Gentle Introduction. <http://www.extremeprogramming.org>, January 2003.
- [39] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Boston, 2000.
- [40] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 230–238, November 1997.
- [41] <http://xml.apache.org/security>.