Automated Assessment of Student-written Tests Based on Defect-detection Capability

Zalia Shams

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
In
Computer Science and Applications

Stephen H. Edwards, Chair
Manuel A. Pérez-Quiñones
Dennis Kafura
Eli Tilevich
Jeff Offutt

03/27/2015
Blacksburg, VA

Keywords: Software Testing, Automated Assessment, All-pairs Execution, Mutation Testing, Coverage Criteria, Defect-detection Capability

Automated Assessment of Student-written Tests Based on Defect-detection Capability

Zalia Shams

(ABSTRACT)

Software testing is important, but judging whether a set of software tests is effective is difficult. This problem also appears in the classroom as educators more frequently include software testing activities in programming assignments. The most common measures used to assess student-written software tests are coverage criteria—tracking how much of the student's code (in terms of statements, or branches) is exercised by the corresponding tests. However, coverage criteria have limitations and sometimes overestimate the true quality of the tests. This dissertation investigates alternative measures of test quality based on how many defects the tests can detect either from code written by other students—all-pairs execution—or from artificially injected changes—mutation analysis. We also investigate a new potential measure called checked code coverage that calculates coverage from the dynamic backward slices of test oracles, i.e. all statements that contribute to the checked result of any test. Adoption of these alternative approaches in automated classroom grading systems require overcoming a number of technical challenges. This research addresses these challenges and experimentally compares different methods in terms of how well they predict defect-detection capabilities of student-written tests when run against over 36,500 known, authentic, human-written errors. For data collection, we use CS2 assignments and evaluate students' tests with 10 different measures—all-pairs execution, mutation testing with four different sets of mutation operators, checked code coverage, and four coverage criteria. Experimental results encompassing 1,971,073 test runs show that all-pairs execution is the most accurate predictor of the underlying defect-detection capability of a test suite. The second best predictor is mutation analysis with the statement deletion operator. Further, no strong correlation was found between defect-detection capability and coverage measures.

Dedicated to my beloved mom,
Mst. Masuma Khatun

Acknowledgements

All praise and thanks to God for helping me successfully complete all the stages of my Ph.D. My deepest gratitude to my advisor Dr. Stephen H. Edwards. His in-depth knowledge in software engineering and regular feedback on the modification of open source tools that I needed in my research was invaluable. He always encouraged me to participate in various research competitions and doctoral symposiums. He gave me flexibility in investigating research problems and exploring my ideas. Afterwards, he guided me towards practical solutions. This process helped me to grow as a researcher. His immense patience and continuous support made it possible to conduct experiments with more than 5 million test runs. He is a leader in computer science education and I am very fortunate to have him as my advisor.

I would like to thank Dr. Jeff Offutt for his suggestions and thorough comments. His feedback led us to one of the major outcomes of my research. My special thanks to Dr. Manuel A. Pérez-Quiñones and Dr. Dennis Kafura for their intriguing questions, constructive discussions, and comments that helped me polish my research findings. I thank Dr. Eli Tilevich for his inspiration, kind words, and support.

I am particularly in debt to Professor Madhav V. Marathe for funding me as a GRA for about four years. I learned to collaborate with researchers of different subjects while working with this team. As a great researcher, he is an inspiration to me.

My deepest thanks to Dr. James D. Arthur for his mentorship and guidance. My interest in programming languages and compilers grew while taking his courses. He helped me find my Ph.D. research area. I thank Dr. Barbara Ryder for introducing me to program analysis research.

I would like to thank my lab mates—Kevin Buffardi, Tony Allevato, Jason Snyder and Ellen Boyd—for being good friends and for all the help.

My thanks to David Schuler for helping me get the right version of Javalanche and for answering my questions. This helped me understand the design of this tool and how to modify it.

My sincere thanks to Muhammad Mafidul Islam, Mahfuza Yeasmin, Kabir Uddin, Sanjida Jahan Kabir, Julekha Chowdhury and Shamim Javed for being my family in Blacksburg. They were always beside me, in good and bad times, especially during the time of sickness. I also thank Shahriar Kabir, Tozammel Hossain, Sifat Siddique, and Nazmul Hasan Nahid for their support whenever I asked for it.

I extend a special thanks to my mentors M. Shahriar Hossain, Monika Akbar and Shaimma Lazem. They were my trusted friends who I could seek advice in any matter, from academic to personal. My deep thanks to my friend Farhana Dewan who encouraged and empathized me throughout my Ph.D. life.

Finally, I thank my family, far from here, for always supporting me without question. My younger sister, Alia Shams, used to make me laugh when I was sad. My beloved parents instilled in me the inspiration to set high goals and the confidence to achieve them. My mom managed to call me every day to check on me despite a ten hour time difference, a full-time job, and taking care of my ailing dad. She patiently heard my everyday life events, research findings, conference experiences, and what not. Her out-of-the-box thinking always gave me a new perspective. I could not have finished my Ph.D. without her limitless care and continuous support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Testing is an important part of software development. The main concern of software testing is to detect as many defects as possible. Failures are caused by defects in programs, which are accidentally introduced by developers because of the inherent complexity of the code. A study by the National Institute of Standards and Technology [55] from 2002 estimated that software failures cause costs of $59.5 billion annually in the U.S., and that over one third of these costs could be avoided by better software testing strategies. Further, most software industries apply testing in different stages of software development and it accounts for 50% of the cost of development. However, unfortunately, less than 15% of practitioners ever receive any formal training on the subject [24].

Considering the necessity of testing, more educators are including software tests [22, 23] in programming and software engineering courses [27, 37]. The goal of including software testing as a part of programming assignments is to enable students to better test their software, and thus hopefully produce code with fewer defects. An earlier study [24] showed that students who tests their own programs produce 28% fewer defects per thousand lines of code. Further, when students write software tests, they have to articulate their own understanding of what their code is supposed to do, which helps them solidify their conceptual understanding about their own program. Regardless of the benefits of testing, students are not accustomed to formally test their code. They usually focus on output correctness on the instructor's sample data [37] and do less testing on their own [27]. To change this practice, educators are including software testing as a part of programming assignments, where a student is required to submit his program along with test cases. In this approach, the student's total grade depends on the correctness of his program and the quality of his test cases.

To support software testing as a part of regular programming assignments, current classroom assessment systems (e.g., Web-CAT, ASSYST, Marmoset) allow students to turn in their programs along with tests. These automated grading tools evaluate student-

written tests as part of grading. To support software testing as a part of regular programming assignments, current classroom assessment systems (e.g., Web-CAT, ASSYST, Marmoset) allow students to turn in their programs along with tests. These automated grading tools evaluate student-written tests as part of grading. The most common strategy currently used is to employ a structural code coverage metric, such as statement coverage or branch coverage, in order to measure how much of the code under test is executed by a test suite. Statement coverage calculates what percentage of the software's statements are executed by the tests in a test suite and branch coverage measures what proportion of the control flow transfers are executed. Software test adequacy criteria such as these are traditionally viewed as criteria for judging when a test suite is "thorough enough"—as a kind of stopping rule that says when a test suite is sufficiently complete. However, such criteria can also be used in another way: to measure the adequacy or degree of thoroughness of a test suite [60]. When used in this way, they become an indicator of the quality of a test suite. Automated assessment systems use coverage criteria in the same way to evaluate the quality of students' tests.

Coverage criteria focus on executing more code with the rationale that the chances of detecting defects in code is zero if it never gets executed. However, executing defects does not guarantee they will be detected. A software test may cause a fault to be executed, but contain insufficient checks of expected behavior to detect that the code did not behave as intended. Moreover, the test may have used data for behavioral checks that fail to detect failures. Thus, it is not sufficient to cover the error; we also need a means to detect it.

Two main goals of software testing are:

1) Executing the defects in code in a way that causes failure, and
2) Recognizing or detecting those failures.

Coverage criteria give information about only part of the first goal of testing: what percentage of the code features is being executed. To examine if tests are executing code in a way to cause failures and to detect those failures, we need to assess oracle quality. Test oracles are checks that determine whether a test case passes or fails. Assessing tests based on oracles provides information on the sufficiency of checks as well as the detection of

observable failures. Currently, oracle quality assessment approaches are not used in any classroom grading tools.

Classroom grading tools have different requirements and challenges than traditional industry tools. Student-written programs are generally small in size, so unit testing is used for assessment. No systemic defect-tracking system is available as professionals use in industry. Common pair-review process may not be used or applicable in introductory classes. Thus, beginners depend on the feedback from teachers, TAs and automated grading tools if available. For large classes or MOOCs (Massive online open course)s, providing the human efforts to execute and assesses student-written tests becomes unrealistic. Thus, incorporating effective test quality measures in classroom assessment tools is a growing need. In this research, we investigate different oracle quality assessment approaches for automated grading of students' tests.

1.1 Research Problem

To overcome the problems of assessing students' tests using statement or branch coverage, we investigate alternative approaches based on the defect-detection ability of the tests. Recently, several measures have been proposed as more sensitive measures [6, 50] of test quality than statement or branch coverage. Among them ***all-pairs execution***, ***mutation analysis*** and ***checked code coverage*** have shown promising results on evaluating test quality. In all-pairs execution, each student's test cases are run against all other students' programs. Mutation testing, on the other hand, involves modifying the programs, with the intent to seed artificial errors into code and to check whether a test suite can find them. These two measures evaluate test quality based on how many defects a test suite can detect, whereas checked code coverage measures the percentage of executed code that influences on the checked results. Technical obstacles of automating these approaches for an educational environment raise the question of whether it is feasible to use them in classroom assessment systems. We also need to find out which one of these alternative measures is more accurate, in terms of predicting the defect-detection capability of a given test suite. This thesis focuses on ***discovering the best test quality measure based on defect detection capability for automatic assessment of student written tests***. The specific

research problem is subdivided into the following three questions for the alternative measures: all-pairs execution, mutation testing with four different sets mutation operator and checked-coverage:

1. **Are these measures feasible to use for automated classroom grading systems?**
2. **Are they better indicators of defect detection capability?**
3. **Which of these test quality measures is the *best*?**

Here, we can define *best* as: the most accurate predictor of how likely a test suite is to reveal actual defects. We identified technical challenges of using all-pairs execution, mutation analysis with four different sets of mutant operators, and checked coverage for automated classroom grading tools. We also devised novel technical solutions to overcome those challenges. We designed and conducted an experiment with two CS2 assignments from two different semesters, to compare all-pairs execution, mutation analysis with four different sets of mutants, and checked code coverage using a massive dataset collected from actual student-written tests. The same comparison was extended for four coverage criteria: statement coverage, branch coverage, object instruction coverage, and object branch coverage. The goal of this experiment was to find out how these measures correlate with defect-detection capability.

1.2 Contribution

The contributions of this research include:

- We designed a study to compare 10 different assessment approaches in terms of their defect-detection capabilities: all-pairs execution, mutation analysis with four different sets of mutation operators, checked code coverage, statement coverage, branch coverage, object instruction coverage, and object branch coverage. Completion of the study determines the best test quality measure to automatically assess students' tests.
- We analyzed technical obstacles of all-pairs execution and provided automated solutions for them.

- We introduced a set of techniques, including detection of mutants incrementally, to apply mutation analysis in educational settings.
- We investigated four different sets of mutation operators to determine which mutation operator produces mutants that are most correlated with defect-detection capability. The four sets of operators are sufficient mutation operators listed by Offutt [46], Javalanche mutation operators [49], statement deletion operator [21], and the combination of variable, constant and operator deletion operators [21]. We choose these sets because they are effective and practical for grading tools.
- We provided a novel way to remove compile time dependencies between JUnit tests and their solutions using late binding to support test evaluation against correct and erroneous solutions that are likely to be from different authors or automatically generated.
- We developed a semi-automated system to calculate a more sensitive form of code coverage using the dynamic slices of a program executed from checked assertions.

1.3 Outline

In Chapter 2, we describe potential test quality measures: coverage criteria, all-pairs execution, mutation testing, and checked code coverage. We also discuss strengths, weaknesses, and popular automated tools available to apply the measures. Our discussion also includes automated assessment systems and their evaluation metrics for assessing students' tests. In Chapter 3, we analyze in detail the obstacles to applying all-pairs execution, mutation testing and checked code coverage, present our solution to the problems, and evaluate the feasibility of our solution. Chapter 4 includes an experiment to compare defect-detection capability with all-pairs execution, mutation testing with four sets of mutation operators, checked coverage, and four coverage criteria. Finally, we conclude in Chapter 5 with plans for future work.

Chapter 2

Background

During the development process software is tested at different stages to confirm it produces expected results under specified conditions. Different levels of testing corresponds to different level of development activity. Unit testing exercises the software at the lowest level. It is designed to assess the produced code during implementation phase. It checks the implementation modules or parts of a program in isolation. A number of test cases are executed by the system under test with specified conditions and inputs to check results by test oracles. A test oracles determines whether the test cases pass or fail.

In this thesis, we focus on evaluating student-written unit tests by automated grading systems. Many educators include software testing activities in programming assignments, so there is a growing demand for appropriate methods of assessing the quality of student-written software tests. If students are taught how to test their code well, they will improve the quality of their code in terms of correctness. In fact, Edwards [24] showed that students who tested their own programs reduced defects by 28% per thousand lines of code.

While tests can be hand-graded, some educators also use objective performance metrics to assess software tests. Several metrics are used to evaluate test quality. Statement and branch coverage, all-pairs execution and mutation testing have been either used or proposed for assessment of student-written tests. Recently, a variation of code coverage—called checked code coverage—has been introduced as a more sensitive measure of test quality than mutation testing and statement coverage. We discuss all four types of measures in the following subsections.

2.1 Automated Grading Systems and Their Evaluation Measures

Students often consider software testing to be boring. They usually focus on output correctness on the instructor's sample data [27] and do less testing on their own [26]. Goldwasser [32] first proposed the idea of executing student-written tests against other

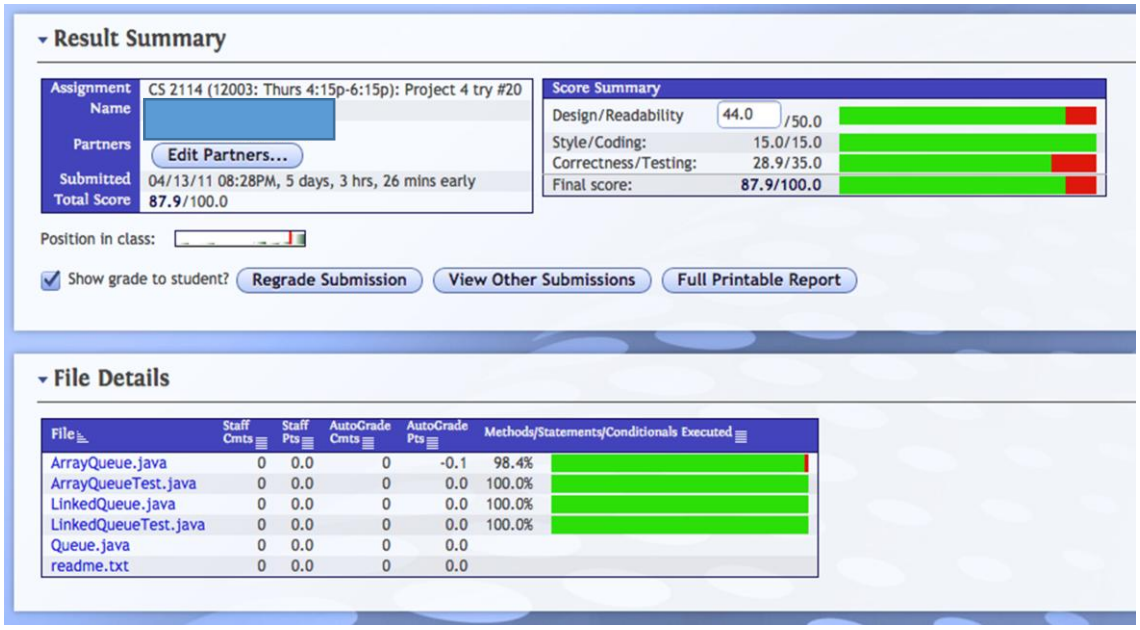


Figure 1: Screen shot of a result summary given to a student by Web-CAT.

students programs. Widely used automated assessment tools (e.g.,Web-CAT [25], ASSYST [36] and Marmoset [54]) evaluate students' codes along with their software tests.

Web-CAT and ASSYST are the most popular assessment tools. Web-CAT can measure statement and branch coverage of student-written tests submitted as a part of an assignment. Web-CAT can also execute instructor-provided reference tests, and for some languages (such as Java) it provides static analysis tools that can assess code style,

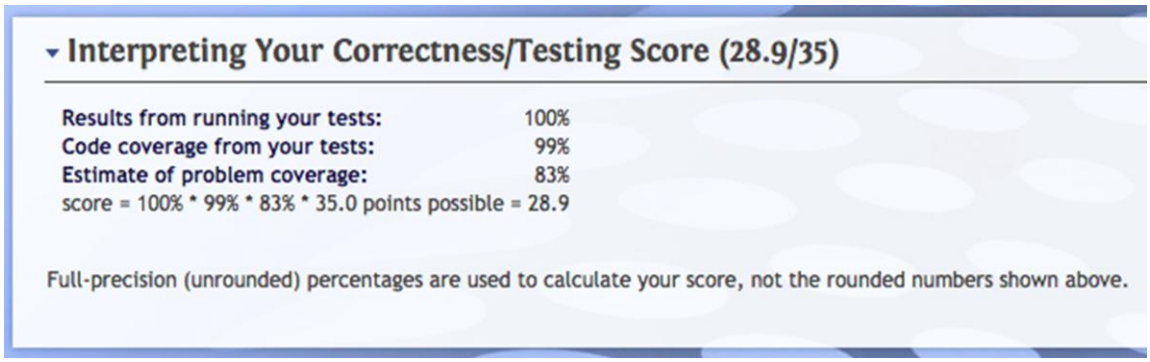
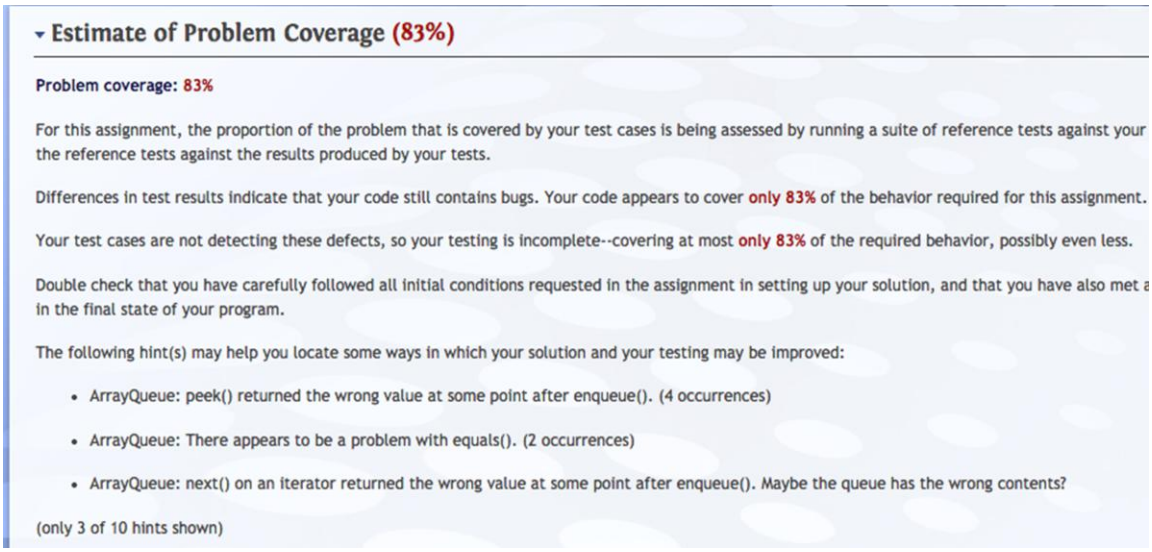


Figure 2: Screen shot of a testing score report from Web-CAT.



Estimate of Problem Coverage (83%)

Problem coverage: 83%

For this assignment, the proportion of the problem that is covered by your test cases is being assessed by running a suite of reference tests against your the reference tests against the results produced by your tests.

Differences in test results indicate that your code still contains bugs. Your code appears to cover only 83% of the behavior required for this assignment.

Your test cases are not detecting these defects, so your testing is incomplete--covering at most only 83% of the required behavior, possibly even less.

Double check that you have carefully followed all initial conditions requested in the assignment in setting up your solution, and that you have also met a in the final state of your program.

The following hint(s) may help you locate some ways in which your solution and your testing may be improved:

- ArrayQueue: peek() returned the wrong value at some point after enqueue(). (4 occurrences)
- ArrayQueue: There appears to be a problem with equals(). (2 occurrences)
- ArrayQueue: next() on an iterator returned the wrong value at some point after enqueue(). Maybe the queue has the wrong contents?

(only 3 of 10 hints shown)

Figure 3: Screen shot of a problem coverage report from Web-CAT.

adherence to coding standards, and some structural aspects of commenting conventions. Depending on the instructor's preferences, a student's grade can depend on 1) design, 2) coding style, and 3) correctness or testing score. The top right corner of Figure 1 gives a summary of the score. Figure 2 illustrates how a testing score is calculated from the statement coverage score calculated from his test suite, percentage of his own test cases

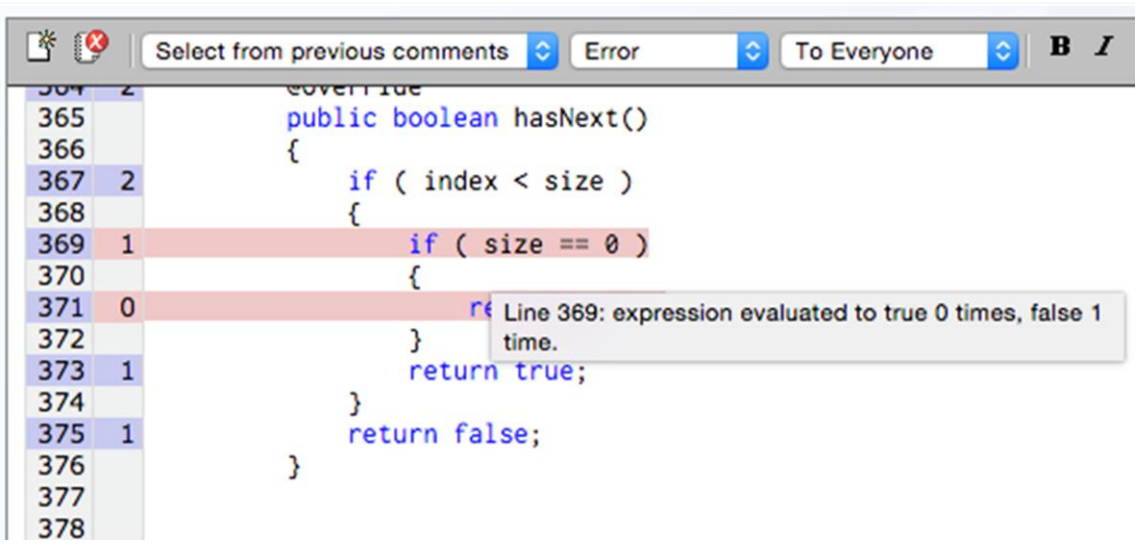


Figure 4: Screen shot of a visual statement coverage report from Web-CAT.

passed, and problem coverage—percentage of required features completed by the student. An additional report is also available to a student explaining problem coverage of his code as shown in Figure 3. This report gives hints on where or in which methods reference tests have found defects. Moreover, a visual report on the statement coverage (Figure 4) from the student's tests is given to the student. As a result, a student knows exactly which lines are uncovered in his code and can try to increase coverage by adding or modifying his test cases.

ASSYST also provides support for executing instructor-written reference tests against student submissions, as well as support for measuring code coverage (i.e., statement coverage) of student solutions when the instructor-written tests are executed. This system can also evaluate the run-time efficiency of a program by the amount of CPU time consumed, its code complexity, and its style. Moreover, weightings can be assigned to particular aspects of tests using ASSYST.

Marmoset is another automated grading system that focuses on Java assignments. It evaluates student programs against two test sets: 1) public test sets, where students see all results, and 2) "release" test sets provided by the instructor—that is, private instructor-written reference tests. The public test sets are available publicly to students and feedback from running public test sets are provided to students immediately. However, release test sets can be run only if a submission passes all the public tests. The results from release test sets are delayed and limited. Marmoset uses statement coverage from the release test sets and from the students' own test sets. This system also provides students with information about their programs using the static program analysis tool FindBugs [17].

All three of these tools run instructor-provided reference tests against student submissions. However, reference tests written in compiled languages, such as Java, do not compile against solutions that fail to provide all the required features in an assignment or that have incorrect method signatures. As a result, these systems cannot evaluate partial or incomplete submissions and gives no credit in those situations. Edwards et al. presented a solution for assessing partial or incomplete Java programs by applying late binding to test cases [30].

Web-CAT, ASSYST, and Marmoset all provide some form of statement or branch coverage but cannot measure how many observable failures that test cases have detected. As an alternative, Aaltonen et al. [6] proposed using mutation analysis to assess the quality of student-written tests in Java assignments. He compared scores that students get for their tests from mutation testing and statement coverage provided by Web-CAT. The comparison showed that students were able to fool coverage tools, and achieving a high score was harder with mutation testing than statement coverage. However, some limitations may make it impractical to use for real-time feedback generation, although it appears feasible for use in batch-style analysis of student submissions.

Some web based tutorial systems, such as BugHunt [31] are designed for students to learn how to write unit tests. It provides existing code containing known defects for students to "find" using testing. Though this work does not assess student-written tests, it does aim directly at including software testing in courses, and helps students learn from their mistakes through an interactive feedback system. Another tool, JavaFest [34], promotes collaborative learning of Java programming and learning to write good unit tests. This tool is inspired by a team-based version of Goldwasser's idea. It groups students into teams and creates competitions where each team's tests are run against other teams' implementations. The team uncovering the largest number of defects in other teams' code wins the completion. However, this is not an automated assessment tool. TAs and tutors provide feedback and help students to use the right interfaces or code structures so that their tests can be executed against each other.

2.2 Coverage Metrics

Coverage criteria [43] were one of the first methods used for software testing and are still the most widely [50] used metrics to assess test quality. They measure the percentage of code features executed during testing. Code features can be statements, branches or path coverage. The idea behind coverage criteria is that a test suite cannot find any defects in code if it never gets executed. On the other hand, the chances of detecting defects in code are more if the code is executed. *Statement coverage* is the most basic form of code coverage. If a specific statement is executed from at least one test case, then it is

```
1  static int abs(int a) {
2    int absVal;
3    absCalled++;
4    absVal = a;
5    if (a < 0) {
6        absVal = -absVal;
7    }
8    return absVal;
9  }
```

Figure 5: An exmaple for coverage criteria.

considered covered. Executing all the statements is necessary to achieve 100% statement coverage.

Object coverage analysis computes metrics focused on machine-level object code. *Object instruction coverage* (OIC) requires all object instructions to be executed at least once. Sometimes, object coverage measures are safer as compiler-generated object code impacts the execution control flow in a way that is not directly visible from source code.

Branch coverage requires that every conditional statement is evaluated to *true* and *false*. For example, given an `if` statement, both the *true* and *false* branches need to execute to achieve 100% branch coverage. If a specific branch is exercised by at least one test, then it is calculated as covered. Branch coverage subsumes statement coverage as it requires all the statements to be covered, including the branches that contain no statements such as an empty else-block of an if-statement. Thus, test suites that achieve full branch coverage also achieve full statement coverage.

Figure 5 shows a simple method, `abs(int)`, that calculates the absolute value of an integer. Invoking `abs(-3)` will achieve 100% statement coverage. However, to satisfy branch coverage, it is necessary to execute the method at least twice, one with `a>=0` and another with `a<0`, such as `abs(3)` and `abs(-3)`.

Condition coverage requires each simple condition in every decision to be evaluated both true and false at least once, but this does not necessarily guarantee each

branch will be executed. A more stringent form is *modified condition/decision coverage* (MC/DC). MC/DC is defined this way [18]:

> "Every point of entry and exit in the program has been invoked at least once, every condition in a decision has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome."

Unfortunately, MC/DC is sometimes infeasible when conditions are dependent (for example, checking whether a pointer is non-null before using the pointer in a later condition in the same decision). *Masking MC/DC* [16] allows for each condition to be shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions that could affect the outcome. It is weaker than MC/DC. However, an empirical study shows that masking MC/DC requires an equivalent number of tests as MC/DC and is a preferred form of MC/DC [16].

A similar form of branch coverage named *object branch coverage* (OBC) is as strong as masking MC/DC for the majority of code [11]. OBC is calculated by instrumenting a program's object code instead of its source code. However, for languages that use short-circuit evaluation of logical operators, object code instrumentation will result in much more stringent coverage requirements. As an example, consider this `if` statement:

```
if (C1 || C2 && C3)
{
     ...
}
```

This `if` statement contains three simple *conditions* combined with logical operators to form one compound *decision*. With traditional branch coverage, a test suite must ensure that the `if` statement is executed when the entire decision evaluates to true, and also when the entire decision evaluates to false (a total of two branches). However, when compiled, this compound decision results in a series of object code instructions that contain a separate conditional jump instruction for each simple condition. OBC requires that each conditional jump in the object code be executed when the corresponding condition is true, and also when it is false. In effect, this forces the tester to exercise both sides of each simple condition in every decision. For most decisions containing *N* simple

conditions, there are $N + 1$ separate object code branches that must be exercised to achieve full object branch coverage.

As a result, OBC requires that each simple condition be evaluated to both true and false, and also that each decision as a whole be evaluated to both true and false, making it equivalent to *condition/decision coverage*. However, in languages where Boolean operators use short-circuit evaluation, achieving full OBC implies achieving full masking MC/DC for compound decisions with certain properties [11], even though OBC is strictly speaking a weaker criterion. An analysis of two industrial applications by Bordin et al. [40] revealed that more than 99% of decisions obey the necessary properties, meaning that in practice, OBC is frequently as strong as masking MC/DC, a criterion considered sufficient for safety-critical systems [11].

Coverage criteria have limitations. They measure how much code has been exercised from the tests. Executing code is important because if a code never gets executed, chances of detecting defects in it is zero. However, covering a defect is not sufficient to detect it. To evaluate the quality of tests we also need to know if tests are checking the result of executed code and are recognizing observable failures. Moreover, a student's solution may be incomplete because it omits some required behaviors, but coverage criteria does not provide information about these omissions. In fact, coverage criteria is not a true indicator of test quality as developers [42] and students [6] can misuse coverage-based test adequacy metrics to create a false sense of a well-tested program.

There are mature and efficient automated tools for coverage metrics. JaCoCo [3], Clover [1] and Emma [2] are three well known Java code coverage tools. Among them, JaCoCo and Emma are open source whereas Clover is a commercial tool. These tools instrument the application to collect coverage data. Afterwards, the tools execute test cases on the application and record coverage data. Finally, the coverage tools populate different statistics from the data collected during execution. Automated assessment systems typically use statement coverage to evaluate students' written tests.

2.3 All-pairs Execution

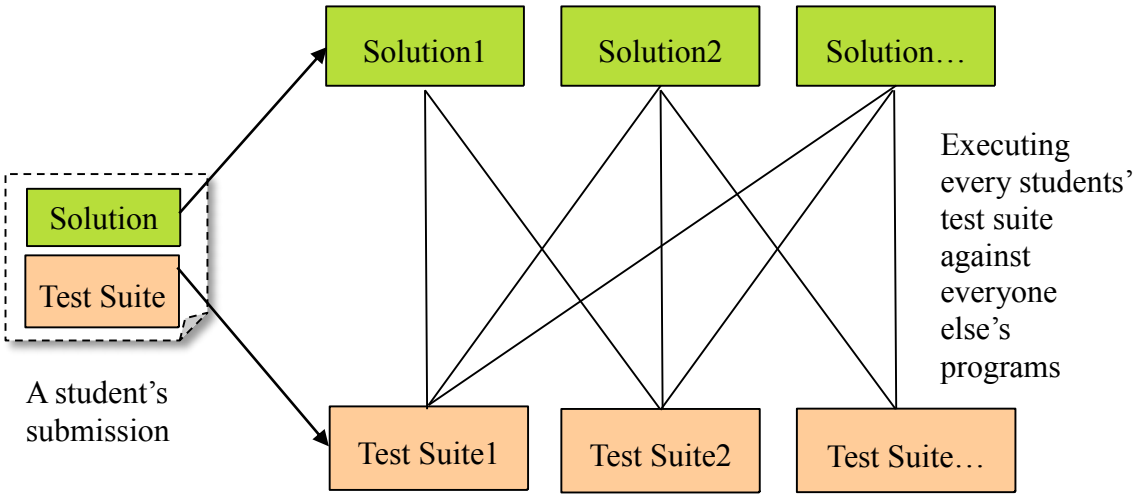


Figure 6: All-pairs execution.

To incorporate software testing as an integral part of programming assignments, Goldwasser [32] proposed the idea of requiring students to turn in tests along with their solutions, and then running every student's tests against every other's program. This *all-pairs* strategy provides a more robust mechanism for evaluating the quality of tests. Figure 6 illustrates this strategy.

A unique feature of this mechanism is assessing by actual defects present in other students' solutions. When students only see the results of their own tests (and possibly their instructor's), they often have an overly inflated sense of the quality of their own programs. This tournament style approach motivates students for software testing in an engaging way. Students take away a greater realization of the density of defects in code they write, as well as their own ability to write tests that find defects in other solutions. Such a metric is beneficial because students see how the class performs overall and how they performed individually—both in terms of how well their own solution passes tests written by others, and in terms of how well their own test suite detects failures in other programs. As a result, students gain a whole new perspective on the value of testing and on the presence of hidden defects in their own code. Moreover, a large collection of student-written tests provides a

diverge environment of software testing. The submitted tests from students may uncover defects in some solutions that are not detected by instructor provided tests [32].

A major disadvantage of all-pairs execution is that the number of tests execution grows quadratically with the number of students. Therefore, this approach is feasible only for fully automated grading systems. However, a key technical problem arose as a barrier to implementing it with modern unit testing frameworks used in the classroom, such as JUnit. The problem is that in such a framework, tests are written in the form of code, alongside the original solutions. A student can test any visible feature (public class, field, method etc.) of his or her solution in any test case. For compiled programming languages such as Java, if any test case within a test class refers to any student-specific feature that is not present in everyone else's solution, then that entire test class will not compile against other students' solutions. This effectively prevents the all-pairs strategy from being used, unless students are constrained to only writing tests using the same common, instructor-specified set of public methods. Edwards *et al.* [30] described a technique to remove compile-time dependencies from the test cases so that they can be compiled once and then run against everyone else's solution, separating out student-specific test cases from those that are more generally applicable at run-time.

2.4 Mutation Analysis

Mutation testing [20] creates a set of modified or mutated versions of the original program with the intent to seed artificial defects and to check if the test cases can detect these injected defects. The modified versions are called *mutants*. If the mutated program produces different results than the original code then it is called a *non-equivalent mutant*. The defective versions, or non-equivalent mutants, are intended to be representative of the faults that programmers are likely to make in practice. The key principle of mutation analysis is complex faults are coupled to simple faults in such a way that a test data set that detects all the simple faults is thorough enough to detect most complex faults [20, 45]. Test suites are run against all the mutants. The effectiveness of a test is measured by its mutation score, which is the percentage of mutants detected or killed by the test.

```
int abs(int a)
{
   int absVal;
   absVal = a;
   if(a < 0){
    absVal=
     -absVal;
   }
   return absVal;
}
```

(a): Original Code

```
int abs(int a)
{
   int absVal;
   absVal = a;
   if(a > 0){
     absVal=
     -absVal;
   }
   return absVal;
}
```

(b): Non-equivalent Mutant

```
int abs(int a)
{
   int absVal;
   absVal = a;
   if(a <= 0){
     absVal=
     -absVal;
   }
   return absVal;
}
```

(c): Equivalent Mutant

Figure 7: Mutant example.

Mutated programs are not always faulty. A mutated program can become a different version of the original solution with no difference in produced results. This type of mutant is called an *equivalent mutant*. Figure 7 (a) shows a simple program fragment that calculates the absolute value of an integer. Figure 7 (b) shows a defective version of the same code where an error has been artificially introduced by changing the comparison operator in the `if` test. This mutant is typical of mutation analysis, where seeded errors represent substitutions or other minor changes to a localized portion of the program. Here, almost any test case that confirmed the code computed the correct absolute value would be likely to discover the mistake. However, in some cases, the change intended to introduce an error might accidentally produce an alternative program that is behaviorally identical to the original, failing to introduce a true defect. Figure 7 (c) shows such a change, where the `if` test has changed from < 0 to <= 0. The effect of this modification will make no difference in outcome since -0 is actually 0 and Figure 7 (c) is behaviorally identical to Figure 7 (a).

When calculating the mutation score for a test suite, equivalent mutants should not be included, since they are behaviorally indistinguishable from the original. However, even

with recent advancement, weeding out equivalent mutants is still a manual process and is labor intensive. Determining equivalent mutants is such a tedious and error-prone activity, that even ignoring equivalent mutants has been considered [47] since the lower mutation score still provides meaningful defect detection information.

Mutants are generated by modifying code at the source or bytecode level. How the code will be modified from the original program is determined based on transformation rules that are called mutation operators. Generally, mutation operators are designed to modify expressions or variables by replacement, insertion, or deletion. Many mutation operators have been proposed. Agrawal et al. [33] provided a list of 76 operators. However, more mutation operators means generating more mutants. To reduce the cost of mutation testing, researchers investigated what set of mutation operators would produce less mutants but would obtain a sufficiently accurate measure of overall mutation adequacy. Such a set is known as a sufficient set [46].

Table 1: Sufficient mutation operators listed by Offutt [46].

Abbreviation	Name	What it does?
ABS	Absolute value insertion	Replaces each arithmetic expression to take on the value 0, a positive value and a negative value
AOR	Arithmetic operator replacement	Replace each arithmetic operator with other syntactically legal operator such as change + to -, *, / etc.
LCR	Logical connector replacement	Replaces each logical connector (AND and OR) with several other kinds of logical connectors
ROR	Relational operator replacement	Replaces relational operators with other relational operators
UOI	Unary operator insertion	Inserts unary operators in front of expressions

Offutt et al. [46] experimentally determined five mutation operators listed in Table 1 as forming a sufficient set of operators. Several other researchers [9, 44, 57] also investigated to determine a sufficient set of operators for different programming languages. A recent study found deletion operators [19] generate fewer mutants but yield tests that are almost as effective as the full set of mutants generated from the sufficient operators. Deletion operators include statement deletion (SDL), operator deletion (ODL), variable deletion (VDL), and constant deletion (CDL). SDL achieves 92% effectiveness [21] for Java. However, combining other deletion operators with SDL achieves 97% of the effectiveness of the sufficient set of mutation operators [19].

Many automated tools are available to generate mutants. Mutants for Java programs can be generated at the source level or at the byte-code level. Both procedures have their pros and cons. Source level mutants are easy to examine but slow in execution as they need to be compiled. Moreover, generated mutants can be from a portion of code that the

Table 2: µJava method level operators [58].

Operator	Description	Example
AOR	Arithmetic Operator Replacement	Replace + with -,*,%,/.
AOI	Arithmetic Operator Insertion	Insert ++, --.
AOD	Arithmetic Operator Deletion	Delete ++, --
ROR	Relational Operator Replacement	Replace > with >=, <, <= .
COR	Conditional Operator Replacement	Replace && with ‖, &, |.
COI	Conditional Operator Insertion	Insert !
COD	Conditional Operator Deletion	Delete !
SOR	Shift Operator Replacement	Replace << with >>
LOR	Logical Operator Replacement	Replace & with |
LOI	Logical Operator Insertion	Insert ^
LOD	Logical Operator Deletion	Delete ^
ASR	Assignment Operator Replacement	Replace += with -=

compiler eliminates as a dead code. Byte-code mutants are faster in execution and more in number, but hard to examine.

There are efficient automated tools to generate both source and byte level mutants. µJava [41] is the most popular source-level mutant generator. It generates mutants from Java programs with two types of operators: class-level mutation operators and method-level mutation operators. Class-level operators [12, 13, 39] are related to encapsulation, inheritance, and polymorphism. In class-level mutants, these operators are changed (e.g., add or remove key word `this`, `static` etc.). Method-level operations include arithmetic, relation, conditional, shift, bitwise and assignment operations. Table 2 gives a brief list of method level operators that are available [58] in µJava. This set includes sufficient operators [46] and deletion operators.

µJava is efficient in generating mutants and easy to use but has some limitations. For example, it uses an old version of Openjava that is not compatible with some features of JDK 1.5 and later. Moreover, it assumes external library (e.g., android jar) classes will be available in source form and provides no way to define a classpath with external jars. It cannot generate mutants for classes that do not have source code. In such cases, µJava does not report the reason for not being able to generate mutants.

Bytecode-level mutation testing is more efficient and scalable than source-level testing. Javalanche [48] is the most widely used byte-code level mutation analysis tool. It generates mutants for operators [49] that are slight modification of operators listed by

Table 3: Javalanche mutation operators [49].

Operator	Description
Numerical Constant Replacement	Replace a numerical constant X by X-1, X+1 or 0
Jump Condition Negation	Negate a conditional jump. This operator is equivalent to negating a conditional statement or subexpression in the source code. This operator negates individual sub conditions also.
Arithmetic Operator Replacement	Replace an arithmetic operator by another one, such as + by -.
Method Call Omission	Suppress a call to a method. If the method has a return value, a default value is used instead, e.g. x = Math:random() is replaced by x = 0:0.

Andrews et al. [8] and was inspired by sufficient mutant operators listed by Offutt *et* al. [46]. Javalanche replaces numerical constants, negates jump conditions, omits method calls and replaces arithmetic operators as listed in Table 3. Javalanche handles JUnit test cases. It executes a series of actions, such as mutation generation, running user supplied test cases, and analyzing coverage from the pass-fail rate of the test cases. However, generating and processing mutants are time consuming and computationally expensive. A comprehensive discussion on mutation testing and uses of mutation testing can be found in the survey papers of Offutt and Untch [47] and Jia and Herman [59].

Aaltonen et al. [6] performed a proof-of-concept evaluation of mutation analysis for assessing the quality of student-written tests in Java assignments. They generated mutants from each student's solution and then ran the mutants against that student's test cases to see how many mutants were detected. Equivalent mutants were weeded out by hand, although that would not be feasible in a classroom situation. This approach appears to provide a deep perspective on how effective the student-written tests are in executing defects that cause failures and detecting these failures. However, some limitations make it impractical to use in the classroom. For example, they could not provide instant feedback to students because the mutation analysis required significant processing time. Moreover, the generated mutants were student-specific. Thus, complex solutions had many mutants, which could lead to an artificially lowered mutation score. Similarly, testing unspecified behaviors that were not part of the assignment could reward a student with a higher mutation score. Furthermore, their approach required manual inspection of the generated mutants to weed out those that were not true defects. In total, their procedure was inefficient and non-uniform, so cannot be used for automated classroom assessment tools.

2.5 Checked Code Coverage

Adequacy of test oracles is important to ensure correct evaluation of programs. Assertions are parts of test oracles that are commonly used for checking program behaviors or outcomes. Test suites are prone to having inadequate assertions [38] that originate from testers' mistakes. Coverage criteria do not reveal that results produced by the executed

code have not been checked by assertions. This may create a false sense of security of good coverage though the program has insufficient checks.

Checked code coverage [50] is an alternative approach for addressing the insufficient oracle issue. It uses dynamic slicing to determine which statements actually contribute to the results checked by test cases. The purpose of this approach is to focus on the code features that actually contribute to the results checked by test cases rather than only considering the code features that are executed in a program. A dynamic backward slice of the program is computed from test cases to determine which statements contribute to the checked result.

A program slice [56] is the set of statements that may influence the variables used in a given location. This approach is used in debugging to locate the source of errors. The slices can be computed statically or dynamically. A static program slice includes the statements that potentially influence the variables at a given point while the dynamic program slice only consists of the variables that actually influence the variables during a concrete program run. Checked coverage uses dynamic slices to calculate the statements that influence the checked results from a concrete execution of a test suite. The influence can be from a *data dependency* or a *control dependency*.

A statement *s* has a data dependency on statement *t* when there is a variable, *v* that is defined (written) in *t* and used (read) in *s* without any redefinition of *v* in between. On the other hand, a statement *s* is control dependent on a statement *t* if and only if *t* is a conditional statement and the execution of *t* depends on the execution of *s*. The checked code coverage uses dynamic backwards slices calculated from assertions.

Figure 8 shows a test (line 8 to 11) that exercises the `dequeue()` method. To compute the checked coverage for this example, the dynamic backward slice from the call to `assertEquals()` (line 10) is built, and only the statements that are on this slice are considered to be covered. The dynamic backwards data and control dependent statements from the `assertEquals()` method call are shown by solid and dashed arrows. Thus, the dynamic slice consists of all the statements except line 4 and 5. However, statement coverage considers line 4 and 5 as covered. To cover line 4 and 5 using checked coverage, a tester needs to check size and front element from assertions.

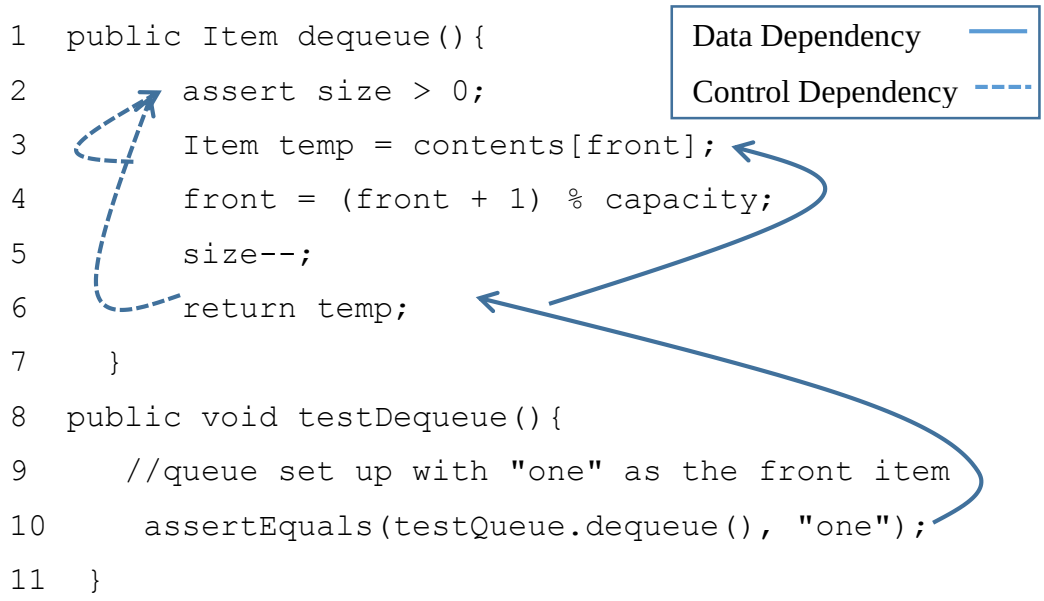


Figure 8: Data and control dependencies in a method call.

Checked coverage is computationally cheaper than mutation analysis or all-pairs execution. Insufficient checks from tests (assert calls) results in low coverage as opposed to statement coverage that counts all code executed from tests. As the executed code that does not affect outcome from behavioral checks are considered uncovered, testers focus on checking as many results as possible. Thus, the possibility of detecting errors increases with higher coverage. Moreover, checked coverage requires running test suites once, whereas mutation analysis requires repeatedly running the tests for all the mutants.

Checked coverage is not a mature test quality measure, however Schuler and Zeller [50] provided a proof-of-concept evaluation of this metric. They used JavaSlicer [4], a home grown tool in their lab, to calculate dynamic slices from seven open-source projects for checked coverage. The limitations of their approach mainly resulted from the limitations of JavaSlicer. The current implementation of JavaSlicer cannot access native method calls, so—many dependencies get lost. Therefore, if the project under test includes native code, it will get lower coverage than actual. Moreover, JavaSlicer currently cannot trace methods from java.lang.System, java.lang.Object and java.lang.String Class. The consequence is that the dependencies through method calls in these classes cannot be reconstructed, leading to incomplete slices. Besides the limitations of JavaSlicer, checked

```
try {
    methodShouldThrowException();
    fail("No exception catched");
}catch(Exception e){
    //Exception occurred
}
```

Figure 9: Common JUnit style to check exceptions.

coverage has its own weakness too. For example, a common practice to check for exceptions is to call a method under test in a try-catch block where try-block includes a `fail()` method call as shown in Figure 9. However, checked coverage will not be able to detect this style of checking and cover statements that contribute to the exception.

Chapter 3

Feasibility of Alternative Measures

Current automated assessment systems evaluate student-written tests using statement or branch coverage, which is imperfect. They measure the percentage of executed code but do not check if the executed code has been tested against expected behavior. Moreover, they do not provide information on how many observable failures have been detected. We will investigate all-pairs execution, mutation analysis, and checked code coverage as alternative measures to assess tests based on defect detection ability of the tests.

Among the three alternative measures, all-pairs execution has never been applied to test cases written in object oriented languages due to technical problems. Checked coverage has been recently proposed but has not been used in industry or academic projects. Well-developed tools are available to apply mutation testing in industrial systems but they may not be feasible for evaluating students' tests. In fact, an educational environment is different from industrial systems in two main ways:

1) Automated assessment systems require instructors to submit a reference solution and tests covering all aspects of the assignments. Industry settings do not have a correct and complete implementation of the project to compare against. The benefit of having a reference solution is that it can be used for checking the accuracy and completeness of students' solutions. Assessment tools can take advantage of a complete solution to screen students' tests.
2) Students must get immediate feedback for their work. Their tests can fail to detect defects in other students' solutions (in all-pairs execution) or mutants generated from a reference solution. Directly revealing the corresponding code section in other students' solutions or in a reference solution is not desirable. However, in industrial projects referring to the source code of the software under test is not an issue at all. Effective feedback generation on students' tests in such a situation is a non-trivial problem.

We investigated technical obstacles of applying the alternative measures in automated assessment systems. We also devised solutions for technical problems of all-pairs execution, mutation analysis and checked coverage.

3.1 All-pairs Execution

All-pairs execution evaluates every student's tests against everyone else's solution. The unique feature of this mechanism is to assess tests in terms of actual defects that students make in their code. However, automation of all-pairs execution is challenging, especially for object oriented languages. In the following subsections, we explain the problems, our solution approach and evaluation process.

3.1.1 Obstacles

All-pairs execution was proposed more than a decade ago when XUnit style testing frameworks were not in practice. The primary technical obstacle preventing all-pairs test execution in many of today's classrooms stems from the aspects of XUnit-style testing frameworks. In this framework, tests are written as additional classes in the same programming language as the software under test, and they may refer to any visible feature of the code itself. This means that students may write tests that refer any public features (class, method or field), whether or not those features are required by the assignment or arise from their personal design decisions (e.g., helper methods). Thus, a student's test may refer a feature that is not present in all solutions. Many such differences may be specific to a particular class or module within the program rather than to the program as a whole. However, software tests that include dependencies on features of the software being tested that are not present in *all* solutions typically *will not even compile* against other student solutions where those features are missing (or are designed differently). This is true in JUnit tests written for Java.

For example, say a student submits an assignment where he writes a LinkedQueue class (Figure 10) and a JUnit test class for LinkedQueue named LinkedQueueTest (Figure 11). This LinkedQueue class defines common methods such as enqueue(), dequeue(), size(), etc. that are present in every student's program. However, this student also added an isEmpty() method as his own design choice. His JUnit test class LinkedQueueTest tests

```
1    public class LinkedQueue<Item>
2    {
3       /* method body and variable declarations are
4        *  omitted for brevity
5        */
6
7        // ~ Constructor
8        public LinkedQueue() {//method body}
9
10       // ~ Public methods
11
12       // add item in the queue
13       public void enqueue(Item value) {//method body}
14
15       // remove item in the queue
16       public Item dequeue() {//method body}
17
18       // check if the queue is empty
19       public boolean isEmpty() {//method body}
20
21       // Get the number of items in this dequeue
22       public int size() {//method body}
23
24       /* Get the item at the front (the head) of
25       * the queue. Does not alter the
26       * queue.
27       */
28       public Item peek() {//method body}
29
30       // Other methods
31   }
```

Figure 10: A sample LinkedQueue class.

each method individually. This student tests his isEmpty() method with a test case call testIsEmpty() in the LinkedQueueTest class. Now, his LinkedQueueTest class cannot be compiled (and so will not run) against other students' programs because they do not provide an isEmpty() method in their LinkedQueue classes. As a result, his tests cannot be evaluated by all-pairs execution. This problem is common for JUnit tests because tests are written in program form in the same language as the software under test and may refer to fields or calls methods of the class under test.

```
1    public class LinkedQueueTest
2                   extends TestCase
3    {
4        // variable for queue used for testing.
5        private Queue<String> linkedQueue;
6
7        // method bodies are omitted for brevity
8
9        // ~ Constructor
10       public LinkedQueueTest() {//method body}
11
12       // ~ Public methods
13       public void setUp()
14       {
15           linkedQueue = new LinkedQueue<String>();
16       }
17
18       //Test the enqueue() method
19       public void testEnqueue() {//method body}
20
21       // Test the dequeue() method
22       public void testDequeue() {//method body}
23
24       // Tests the isEmpty() method
25       public void testIsEmpty()
26       {
27           assertEquals( linkedQueue.isEmpty(), true);
28       }
29
30       // Tests the size() method
31       public void testSize() {//method body}
32
33       //Tests the peek() method
34       public void testPeek() {//method body}
35
36       // Other methods
37   }
```

Figure 11: JUnit test class for LinkedQueue.

The JUnit testing framework for Java is the most well-known Unit testing approach, and it has seen growing use in the classroom in the past decade. Normally, writing such tests requires no significant expertise beyond that required for writing the

software being tested, and tests can be written and managed with the same tools used to produce the source code to be tested. This has caused a significant change in the way software tests are written in many classrooms. Unfortunately, the nature of such software tests violates one of Goldwasser's principal observations [2]:

> Because students will be submitting test sets that are to be run (automatically) on other students' submissions, a standardized format for describing the tests must be established in the assignment description. The overwhelming conclusion is to rely on a textual interface for program input.

In other words, student tests must be written to a consistent interface or API in order to be "interchangeable" across submissions. They must be expressed in a "black box" form that is devoid of any dependencies on how a particular solution is constructed. Goldwasser notes the potential for tragic failure if a student makes mistakes in understanding or implementing the test input format, or the part of their solution that handles reading/parsing such input. He even suggests that where possible the instructor provide all students with an appropriate front-end, implementing the standardized testing interface to preempt such issues.

Requiring student submissions to take the form of a complete program with "a textual interface for program input" provides a uniform mechanism for executing tests, regardless of the internal design choices made by any particular student. Furthermore, it allows tests to be run against any solution, even if that solution happens to be missing significant required features entirely, or only a partial implementation of others. JUnit (or, more generally, XUnit) tests do not work that way because of their nature. Thus, to execute each student's tests against every other student's solution when students are writing JUnit-style test cases for their own Java solutions, we must devise a way to ensure *a uniform interface* against which tests can be executed *regardless any differences between solutions or divergence from the assignment requirements*.

3.1.2 Solution Approach and Implementation

3.1.2.1 Strategy

A trivial way to resolving this issue can be providing students a common interface that everyone implements as proposed by Helmick [35]. However, this approach limits the opportunity for students to learn how to translate problem solutions into program modules or methods. A novel way to resolve the issue of compile time dependency between a student's tests and his solution in Java is transforming the student-written tests so that they use reflection to defer binding to specific features of a solution until run-time. JUnit tests that perform all manipulation of student classes and objects using reflection have no direct compile-time dependencies on any particular solution. We transform the bytecode of the test cases to students' submissions. Note that test sets that depend on the internal details of one particular solution can still be *compiled*—if compiled against the particular solution they were written for. For example, one student's tests will compile against his or her own code if they compile at all, and so we need not worry about syntactically invalid test sets. Similarly, instructors typically provide their own implementation to double-check reference test sets, so the reference tests will compile against this. Thus, the strategy involves the following three steps:

1. Compile JUnit tests against the solution provided alongside them (i.e., written by the same author) when they are first received (an action that existing automated grading systems already perform).
2. *Transform* the bytecode in the .class files from the compiled version of a test set so that it performs all manipulation of the solution's class(es) using reflection. In other words, use a tool to automatically convert the "plain" JUnit tests into purely reflective tests.
3. Run transformed versions of test sets against other solutions as needed, now that the test sets have no compile-time dependencies of the software to be tested.

When transformed test cases are run against all students' code, a single test case may result in three possible outcomes (rather than just pass/fail):

1. The test case may pass successfully.

2. The test case may fail inside the reflection infrastructure, indicating that some dependency in the test case was not met correctly by the code being tested, a situation we can call a *reflection mismatch*.

3. The test case may fail in some other way, indicating incorrect behavior of the software under test.

We need to handle reflection mismatches differently depending on the nature of the test set. Distinguishing reflection mismatches from incorrect behaviors is important. Fortunately, in automated assessment systems, instructors provide a reference implementation and reference test cases that are presumably correct and include all the required features of the assignment. A simple way to differentiate reflection mismatches from incorrect behaviors is running the student's tests on the reference implementation. Student tests that pass against the reference solution can be presumed *valid*—that is, they do not possess student-specific dependencies and check for results that are consistent with the assignment. Student tests that produce reflection mismatches against the reference solution can be presumed *student-specific*—since they depend on some features of their original author's code that was missing in the reference solution. Finally, student tests that fail against the reference implementation can be presumed *invalid*, since they encode different behavioral expectations than those embodied in the reference solution. Only the valid tests are used in all-pairs execution. In Section 3.1.2.2, we discuss how we implemented this approach and then in Section 3.1.3 we present an analysis of all-pairs execution on two separate class assignments.

3.1.2.2 Implementation

The key mechanism in the implementation of this strategy is the bytecode transformer. Bytecode rewriting has been employed in a number of aspect-oriented programming tools, and a number of robust libraries exist for this task. For the implementation of our solution, we chose to use Javassist [14, 15], which provides an API that is well suited for the transformation task. Javassist provides a mechanism for expressing bytecode translations in the form of Java source-level code snippets laced with substitution variables. Javassist uses its own compiler internally to produce the new bytecode that corresponds to such a pattern.

Using these features, we implemented a Javassist-based bytecode transformer that can load each of the class files that comprise a test set, replacing each constructor call, method call, and field access on any object of the class(es) under test, with a corresponding action from our streamlined reflection interface, ReflectionSupport [51]. Java supports programmers with facilities to dynamically create objects, invoke methods, access fields, and perform code introspection at runtime. These capabilities come at the cost of reduced readability and writeability, since code written using Java's reflection classes is clunky, bulky and unintuitive. Common tasks such as object creation, method invocation, and field manipulation need to be decomposed into multiple steps that require try-catch blocks to guard against checked exceptions. Type casts and explicit use of class types as parameters make development and maintenance of code difficult, time consuming and error prone.

We developed an open-source library called ReflectionSupport that addresses these problems and makes reflection in Java easier to use. ReflectionSupport provides static helper methods that offer the same reflective capabilities while encapsulating the overhead of coding with reflection. Thus, by using ReflectionSupport we completely encapsulate the details of using reflection. As a result, test cases written using this library remain the same length as their non-reflective equivalent, but they have no compile-time dependencies on the software under test.

The modified form of each class file is then written out to disk for use whenever that test set needs to be executed. Translation can occur at different times. For our purposes it is most convenient to translate the class files for a test set when the code is first submitted and save them, since they will potentially be used multiple times. Alternatively, translation could be performed later. By writing a custom class loader, it is even possible to perform translation dynamically on demand, but such flexibility is not necessary in this context.

3.1.3 Evaluation

To evaluate the practicality of this solution, we initially applied it to two separate assignments in two different courses. One of them was from a CS1 course offered in fall 2007 and another from a CS2 course in spring 2011.

3.1.3.1 A CS1-level Assignment

First, we examined a CS1 assignment from fall 2007 where students implemented their own "virtual pet". The assignment provided an abstract base class and a state machine description of the required behavior. Students wrote their own implementation of the state machine along with their own software tests. They had two weeks for writing code and tests. Students were instructed about basic test design principles such as to check corner cases, loops, and methods individually. However, they were graded based on branch coverage of their tests and they were also given feedback on their work that specifically identified uncovered code. Therefore, naturally they tried to increase branch coverage.

In the semester under consideration, 46 students completed the assignment. We used our bytecode translator to convert all of the corresponding test sets to reflective versions. The 46 test sets consisted of 463 individual test cases. Test sets ranged in size from five test cases to 15, with a mean of 10.1 and median of 9.

The reflective versions of all test sets were run against an instructor-provided reference solution. Of the 463 test cases, 405 (87.5%) passed and were thus valid. 27 test cases (5.8%) were invalid, and 31 (6.7%) were student-specific. In this assignment, 50% of the test cases failed to find any defects, and 63% of the programs passed all the test cases. Most importantly, no compile time failure occurred while executing transformed test cases. Detailed results of this experiment is available in [30].

3.1.3.2 A CS2-level Assignment

To analyze how our solution works on a more complex assignment, we also examined a CS2 assignment from spring 2011. This assignment required students to write two different implementations of a generic queue interface in Java—one using a linked representation and one using an array-based implementation. In addition to basic queue operations, students were also required to implement equals(), hashCode(), clone(), toString(), and iterator(). Students wrote their own software tests as well. They had two weeks to submit their programs and test cases. They were taught test driven development and instructed to write JUnit tests.

In the semester under consideration, 101 students completed the assignment. We used our bytecode translator to convert all of the corresponding test sets to reflective versions. The 101 test sets consisted of 2156 individual test cases. Test sets ranged in size from eight test cases to 33, with a mean of 21.3 and median of 22. The reflective versions of all test sets were run against an instructor-provided reference solution. We first applied all-pairs execution on this assignment in 2012, and published our results in [30] . At that time, of the 2155 test cases, 2002 (92.9%) were valid, 126 test cases (5.8%) were invalid, and 28 (1.3%) were student-specific. Recently we applied all-pairs execution on the same assignment with updated ReflectionSupport, which was more precise in searching methods and fields. As a result, a large number of students' test cases were detected as student-specific. We investigated students' code and found that many students had implemented base classes of their own. Thus "no class found" errors occurred frequently when we ran their tests (having reference to their base classes) against the reference solution. To handle students' own choice of base classes, we manually changed student-specific base class names to the interface name that instructor provided them. This allowed many more many more students' tests to be considered valid, so they could be included in all-pairs execution. The average all-pairs score was 55%. A more detailed discussion is presented in Section 4.3.

In our most recent experiment, we also used another CS2 assignment, "Minesweeper", given in fall 2011. Students were required to implement a MineSweeperBoard class. Students could load a board state, check if they won or lost the game, uncover or flag cells, find out number of rows and columns etc. through method calls. A total of 147 students submitted the assignment including 1540 test cases. Out of 1540 tests, 1148 test cases were valid and the rest were either invalid or student-specific. The average all-pairs score was 61.5%. Further details about the experimental data are presented in Section 4.3.

3.2 Mutation Testing

Mutation analysis is a robust mechanism for test quality assessment. It modifies or mutates programs with the intent to seed artificial errors in programs. Mutated versions

having a different outcome than the original program are defective. Tests are evaluated based on whether they can detect defective versions. This approach has been proposed as an alternative to code coverage. However, a number of practical problems emerged as barriers against applying mutation testing in classroom grading tools. As we mentioned at the beginning of Chapter 3, automated grading is different than a traditional industry environment. Before evaluating how many defects student-written tests can detect, we need to check of if their tests are correct by executing them against the reference solution. Also automated grading systems are evaluating tests submitted by all students for the same assignment and these tests are comparable. An industry environment does not have multiple copies of tests written by different testers for the same problem to compare against. Moreover, mutation testing is used for designing sufficient tests rather than assessing alternative test sets. In the following subsections we discuss the technical issues, our solution approach and experimental results for evaluation of our methods.

3.2.1 Obstacles

Mutation analysis is a stronger indicator of test adequacy and effectiveness than coverage criteria. However, three main obstacles make it impractical to use in a classroom environment where feedback is provided in real-time.

First, to check that a student's tests are effective for the *assigned problem*, one must generate mutants from a complete, and correct solution of the entire problem. The student's own solution may not meet these requirements, which gives rise to one of the limitations of coverage criteria. In effect, lack of completeness or correctness brings the same limitations to bear on mutation analysis. If the student's solution is incomplete, the set of mutants generated from the student's solution will not cover the space of all defects possible in the assignment. If the student's solution contains additional extra features that are not required, the number of mutants generated may be larger than necessary. But most importantly, if the student's solution contains defects already, then some generated mutants that differ from the student's original solution may in fact be correct, and there is no way to reliably distinguish when a non-passing test indicates that a non-equivalent mutant has been discovered, versus when the test itself has behavioral expectations that differ from the

assignment so that it will pass on the student's (buggy) solution. As a result, mutants must be generated from a reliable solution that is known to be complete and correct.

Second, students' tests may have dependencies on their own solutions and may fail to compile against the reference solution.

Third, mutation testing is computationally expensive because not all generated mutants necessarily represent faults and even with the advancement in mutation analysis, determining which mutants are actual defects (non-equivalent mutants) and which mutants are innocuous version of original code (equivalent mutants) is still a manual process. It is infeasible to manually weed out equivalent mutants from hundreds of mutants for each programming assignment. Section 3.2.3 makes this point more clear with evidence of actual number of mutants generated for four assignments. To use mutation analysis in the classroom for automated feedback, it is necessary to devise a practical approach for detecting and weeding out such equivalent mutants.

3.2.2 Solution Approach and Implementation

To move toward a practical mutation analysis approach that can be used in the classroom, the three problems outlined in Section 3.2.1 must be resolved.

3.2.2.1 Using a Reference Implementation

Effective mutation analysis requires that we assess the student's tests against a solution to the problem that is known to be complete and defect-free—such as a reference solution provided by the instructor. Student-written solutions may be incomplete or erroneous. Therefore, an instructor-provided reference solution is a more reliable candidate to be used as the mutation source.

Mutant generation also takes time. Altanon et al. generated mutants from each student's own solution and thus total time for mutant generation become significant. If mutants are generated from a reference solution available when an assignment is created, it is possible to pre-generate the full set of mutants from the reference solution ahead of time, so that mutant generation will not slow down analysis of student-written tests.

We generated mutants from µJava and Javalanche. However, we had to modify both of the tools- µJava and Javalanche. Original version of µJava expects source files all

external library classes such as Android jars to be available for mutant generation. It does not allow a user to only class files or jar files to add in the class path. Students had to use external jars or instructor provided class files as required by assignments and it is a common practice of Java programming. We modified µJava to give users options of external jar or class files. We also modified Javalanche [48] for our experiment. The original version of Javalanche internally generates mutants, runs a list of given test suites, and analyzes coverage of the test suites, all as one action. It does not store mutants; instead, it regenerates them every time a new test suite is analyzed. We modified Javalanche to separate out the mutant generation step and to store the generated mutants so that when a student submits a new test suite, the generated mutants can be reused. This minimizes the overhead of mutation generation, with the aim of supporting real-time feedback to students.

3.2.2.2 Removing Compile-time Dependencies from Test Suites

While the intent is to run student-written tests against the pre-generated mutants from a reference implementation, those tests may not even compile against the mutants. If student-written tests are provided in source code form, they may have implicit or explicit dependencies on specific, individual design decisions present only in that student's solution. A novel way to resolve this issue in Java, as discussed in Section 3.1.2, is to transform the student-written tests so that they use reflection to defer binding to specific features of a solution until run-time. We rewrite the bytecode of a student-written test suite into a pure reflective form using our bytecode translator in the same way we did for all-pairs execution.

The purely reflective test cases will run against any mutant. Individual test cases that depend on features that are missing from the reference implementation fail at run-time, while other test cases run normally. This makes it possible to separate out "student-specific" tests from those that are generally applicable to any solution, and then mutation analysis can be restricted to just those that tests that are generally applicable.

3.2.2.3 Incrementally Identifying Non-equivalent Mutants

While it is not possible to automatically determine if a mutant is equivalent to the original program, we can instead use a conservative approach to classify mutants as *provably different* from the original, or *not provably different*. Initially, all mutants are placed in the *not provably different* category, since we have no evidence they are true defects. When the instructor's reference tests are run against these mutants, any mutants detected can be moved into the provable different category immediately.

Then as each student submits tests, we can screen the tests using the original (non-mutated) reference solution. This allows invalid tests, those that do not correctly capture expectations of the problem, to be weeded out. If the valid tests are run against all mutants, any mutant failing a valid test is then *provably different* from the original. Any mutants remaining in the *not provably different* set can be ignored for the purposes of mutation analysis, since they are potentially equivalent mutants.

As more and more students submit tests, the *provably different* set can increase in size, including every mutant to date that some valid test has definitively shown is observably different from the reference solution. This allows the strength of the mutation analysis to increase over time. This automatic mutant detection process is conservative because there may be cases where no test suites find behavioral differences for a given mutant that is actually a defect. However, initial results from the evaluation presented in Section 3.2.3 indicate that this is not a significant issue.

3.2.2.4 Fixing Some µJava Issues

The above mentioned approach has been used to generate mutants from µJava using an instructor-provided solution and to categorize non-equivalent mutants. However, we faced some practical challenges while generating mutants using µJava. The three main problems and our solution for them are discussed below:

1. µJava does not allow the use of external jar files. It expects source files to contain all the required code. We modified µJava so that external library or jar file paths can be provided in the configuration file.

2. µJava internally uses an old version of Openjava which is not fully compatible with JDK 1.5. As a result, it cannot handle “assert” statements. We modified the reference solution by commenting out all assert statements. Most of these assertions are part of test oracles. So, we added those assertions as a patch in generated mutants.
3. µJava does not create mutants for abstract classes. We used one CS2 assignment where the reference solution had an abstract class. So, we had to modify the reference solution to change the abstract class into a concrete class so that all the code can be mutated.

3.2.3 Evaluation

To evaluate the practicality of this approach, we applied our solution with Javalanche to seven assignments originally given in CS1 and CS2 courses, where students were required to write their own software tests for each of their programs. Out of the 7 projects, 4 were from CS1 where the number of generated mutants were 47, 45, 43, and 27. The number of students completing the CS1 assignments varied from 42-47. Average mutation scores were much lower than the average statement or branch coverage scores for all of the projects.

In the same experiment, we used 3 CS2 assignments where the number of students submitting the assignments were approximately 99. The total number of Javalanche mutants varied from 109 to 315. Average mutation scores were 68.5%, 75.9% and 42.2% whereas average statement coverage scores were 94.9%, 96.9% and 95%, respectively. This shows achieving a high coverage score was easier than achieving a high mutation score. Detailed experimental results are available in [52].

Recently we used one new CS2 assignment (Project 1: Minesweeper) from fall 2011 and reused one CS2 assignment (Project 4: Two Queues) from spring 2011 for mutation testing. The same assignments were used in all-pairs execution as well. We used both Javalanche and µJava to create mutants. We generated µJava mutants for sufficient mutant operators, SDL, and a combination of VDL, CDL and ODL. For the Project 1 assignment, the number of Javalanche generated mutants was 319. For the same project, the number of µJava mutants varied based on mutation operators. The total number of

sufficient mutants were 124, SDL mutants were 85, and combined VDL, CDL and ODL mutants were 110. The average mutation score for Javalanche mutants was 77.6%, for µJava sufficient mutants was 33.4%, for SDL mutants was 70%, and for the combination of VDL,CDL and ODL was 32.5%. The distribution and analysis of these scores are presented in Section 4.3.

For the Project 4 assignment, the total number of mutants generated from Javalanche was 316 whereas the total number of mutants generated from µJava was 627. Out of 627 mutants, 388 were from sufficient operators, 131 were from SDL and 108 were from the combination of VDL, CDL and ODL. Average mutation scores were: Javalanche mutants 55.7%, µJava sufficient mutants 46.5%, µJava SDL mutants 46.6%, and µJava VDL+CDL+ODL mutants 31.7%. A detailed data analysis is presented in Section 4.3.

3.3 Checked Code Coverage

Checked coverage is a more sensitive measure [50] than statement coverage. For checked coverage, we are interested in the proportion of statements that contribute to the computation of values, that are checked by the test suite. It subsumes statement coverage because in order to reach full checked coverage, every statement has to be on a dynamic slice, and thereby, it also has to be executed at least once. Thus, every test suite that reaches full checked coverage also reaches full statement coverage.

3.3.1 Obstacles

Schuler and Zeller [50] provided a proof-of-concept evaluation of checked coverage using JavaSlicer [4]. However, JavaSlicer has some limitations and it cannot be used readily to calculate checked coverage. The obstacles behind checked coverage calculation are described below:

1) No automated tool is available for measuring checked coverage. Some open source tools can be used for tracing programs and for creating dynamic slices from the traces. However, these open source tools will need tailoring as they are designed mainly for program slicing rather than computing checked coverage in automated assessment tools.

2) The first step of computing checked coverage is to identify all the explicit behavioral checks inside the test suites (e.g., assert()-style method calls in executable tests). For a small program, manually identifying all such statements may be feasible, but an automatic system must be developed to handle large-scale use of this approach.
3) Test runs need to be traced to provide the raw data necessary to compute dynamic slices. Tracing all test runs is time consuming. Therefore, an automated system should be developed for batch style execution of the programs and storing traces for future dynamic slice creation.
4) JavaSlicer collects a trace of a program execution and creates a dynamic slice from one or more lines of the program given as inputs. However, these slices also include code from Java library classes, not just the code under test. As a result, the slices must be filtered in order to measure only the code under test.
5) While JavaSlicer calculates the executed slices, this information by itself is insufficient for measuring checked coverage. One must also measure the portion of the code under test that is not part of any checked slice, so that the checked coverage can be calculated as a percentage of the total (checked and unchecked) code.
6) Finally, JavaSlicer traces each thread separately, and cannot reconstruct data dependencies between different threads. Some JUnit 4 features, such as specifying test case execution timeouts, use Java threads and prevent data dependencies from being tracked. As a result, JavaSlicer will fail to create dynamic slices for JUnit 4-style test cases that use such features.

3.3.2 Solution Approach and Implementation

In order to evaluate checked coverage, we developed a semi-automated system. We use a combination of JaCoCo and JavaSlicer in this process. The whole procedure is executed for each student's tests in the following steps:

1) We run the student's tests against his or her own program using JaCoCo to collect coverage data. This report includes information about how many bytecode instructions are present in each class and each method under test, which is used later in calculating the percentage of code included in checked slices.

2) Our system identifies all executable checks in the test cases. At this stage, all calls to JUnit assert-style methods from a test suite are considered to be checks.
3) The system uses JavaSlicer to trace programs and store traces. Afterwards, a combined dynamic slice for all of the identified executable checks is created from JavaSlicer.
4) Finally, the system uses the report from JaCoCo to filter out Java library instructions. It determines how many instructions from the students' program have been checked by the tests and reports the checked coverage score.

Our system uses a JUnit 3-style test runner so that all test methods run under the main thread and JavaSlicer can create slices without losing data and control dependencies.

3.3.3 Evaluation

We evaluate the feasibility of our solution by applying it to two CS2 assignments: 1) Project 1 of fall 2011 and 2) Project 4 of spring 2011. As mentioned before, Project 1 was Minesweeper where 147 students submitted their work including 1148 valid tests. The Project 4 required students to implement a queue using an array and also using a linked list. 101 students submitted this assignment, including 1476 valid tests. The same assignments were used for all-pairs execution and mutation testing.

Valid student-written tests were executed against their own programs. We collected data using JaCoCo to measure how many bytecode instructions were present in each class and each method under test. Our system also recorded all assert-style method calls from each student's tests. Afterwards, we used JavaSlicer to collect program traces executed from the tests and to create dynamic slices from recorded assert calls. Finally, we filtered out Java library calls and calculated checked coverage using information from the JaCoCo report.

Experimental results show that the average checked coverage score for Project 1 was 48.9% which is lower than Project 4 average score 56.9%. For both the projects, average checked coverage scores were much lower than the average statement coverage, branch coverage, OIC and OBC scores. This indicates achieving high checked coverage was more difficult than achieving high statement or branch coverage, as expected. Further discussion on experimental data is presented in Section 4.3. An earlier version of this result was published in [53].

Chapter 4

Comparison of Test Quality Measures

Evaluating the quality of students' test suites is important so that students can learn how and where to improve. Even though statement coverage, branch coverage, OIC, OBC, all-pairs execution, mutation analysis with different sets of mutants, and checked coverage all have their own benefits and computational overheads, determining which one is a better indicator of actual defects detected in students' code is a non-trivial problem. Moreover, student-written tests are developed by individual students to exercise their own program solutions, even though those solutions might be incorrect or incomplete. Quantifying the defect-detection capability of a test suite written for an incomplete solution is even more difficult.

To compare the different test quality measures discussed in this proposal, we designed an experiment using tests and solutions produced by students. We conducted this experiment for two CS2 assignments. One of them is Project 1 from fall 2011 and the other one is Project 4 from spring 2011. We evaluated both of the assignments with statement coverage, branch coverage, OIC, OBC, all-pairs execution, mutation testing, and checked code coverage. We analyzed four different sets of mutants that appear practical for classroom grading. The four sets are generated by sufficient mutation operators [46], the statement deletion operator, a combination of variable, constant and operator deletion operators, and Javalanche's set of mutation operators [49]. An earlier version of this work was published [28] at ICSE.

4.1 Experimental Setup

To evaluate the different measures for student-written tests, we aim to compare them to the "true" notion of test quality: the likelihood that a test suite will discover any given defect, which we call the suite's *defect-detection capability*. Measuring the actual defect-detection capability of a test suite is both challenging and expensive, however—otherwise, practitioners could simply use this measure itself as the (ideal) indicator of test suite quality. In the experiment, Project 1 had 147 student submissions. The other

assignment, Project 4, had 101 student submissions. Each of these programs contained at least one error and worked as a representative of authentic common human-produced errors. However, a way of identifying the errors present in this collection is still necessary.

Because of the size of the collection of programs, manual defect counting was impractical. Therefore, we used a technique first described by Edwards [24]. This process was divided into two main steps: 1) creating a master test suite that uniquely cover all observed defects from students programs, and 2) determining how many of the tests from the master suite are covered by student written tests to calculate their defect-detection capabilities.

4.1.1 Creating a Master Test Suite

We compiled a comprehensive test suite containing all tests written by all authors, including all the reference tests produced by the instructor. Then this "super-sized" test suite was reduced by removing "redundant" test cases—that is, test cases that produce exactly the same pass/fail outcome on every single program in the collection. After this reduction, for any pair of test cases in the reduced set, there was at least one "witness" among the student programs that passes one but fails the other. Note that this does not ensure that the suite is "orthogonal," that is, ensuring there will be no significant overlap between test cases. Instead, while separate test cases may overlap in the features they check, they still will differ in the specific defect(s) they detect. Thus, while each individual test case in the reduced test suite does not necessarily represent one individual defect, each can be considered to represent a (hopefully small) "equivalence class" of defects, where all defects in the equivalence class cause the corresponding test case to fail in any program where they occur. In an earlier experiment [24], Edwards showed a statistically significant correlation of 0.755 between the number of test case failures in a test suite constructed this way and the true number of defects present in the student code, so considering the test cases in this master test suite to represent (small) distinct and identifiable equivalence classes of defects is justifiable.

		Student Programs								
Valid Student tests and Reference Tests		P_1	P_2	P_3	P_4	P_5	…	…	P_{M-1}	P_M
	T_1	1	0	1	0	0	…	…	1	1
	T_2	0	0	1	1	0	…	…	0	0
	T_3	1	0	0	1	1	…	…	1	0
	T_4	0	1	1	0	0	…	…	0	1
	T_5	…	…	…	…	…	…	…	…	…
	…	…	…	…	…	…	…	…	…	…
	….	…	…	…	…	…	…	…	…	…
	T_{N-1}	1	0	1	0	0	…	…	1	1
	T_N	1	0	0	0	1	…	…	0	1

Pass 1
Fail 0

Equivalence Class

Figure 12: Master suite formation.

Figure 12 demonstrates the concept of the master suite creation. This process is similar to creating a two dimensional matrix where each row represents a valid test case—either from students or from reference tests—and each column represents a student program. If a program passes a specific test case, then the cell gets a "1", otherwise "0". Thus, we create a defect signature consisting of pass-fail values of all the programs for each test case. If two test cases have the same defect signature for all the programs such as T_1 and T_{N-1}, then both of them are detecting the same equivalence class of defects. Only one test case detecting an equivalence class is included in the master suite. As a result, all distinct and identifiable defects that students made in their code are detected by the master suite.

4.1.2 Calculating Defect-detection Capabilities for Students' Tests

Manually calculating the exact number of defects that exist in each student program, and then the number of defects that are detected by each test suite, is prohibitively expensive. However, by creating the master test suite, we had a suitable proxy—each test cast in the master suite would represent one equivalence class of defects that is uniquely identifiable. Further, every defect detectable by any student-written test suite in the

experiment will already be accounted for in the master suite, by virtue of the master suite construction process. Every individual test case in each student test suite must completely overlap one test case in the master suite, in the sense that the two test cases result in exactly the same pass/fail outcome on every student program in the experiment. This made it possible to count the number of unique test cases in the master suite that correspond to test cases in a given student's test suite. Further, we knew how many of the student programs fail each master suite test case—that is, how many student programs contain a corresponding defect from that equivalence class. Therefore, we were able to determine how many test cases of the master suite a student's test cases overlapped. We calculated the number of program failures detected by the overlapped master suite tests. If a master suite test case was overlapped more than once then we counted its' detected failures once. Thus, we computed an approximation of the suite's defect-detection capability—the probability that for any given defect that occurs in any of the student solutions, this suite would detect that defect.

The simple formula to calculate defect-detection capability is:

$$\text{Defect-detection Capability} = \frac{\sum(\text{No. of program failure in master suite that that are overlapped by a student's tests})}{\text{Total no. of program failure detected by all the tests of master}}$$

For example, let's assume a student submits 17 test cases that overlap with 5 test cases of the master suite. The number of program failures or defects detected by the overlapping testcase1 is 5, testcase2 is 19, testcase3 is 28, testcase4 is 8 and testcase5 is 4. Thus, the total number of defects detected by the student is (5+19+28+8+4 =) 64. If the master suite detects 2400 defects by all the test cases, then the student's defect-detection capability score is (64/2400 *100 =) 2.6%.

4.2 Evaluation Approach

The purpose of our experiment is to find which test quality measure most accurately determines defect-detection capability of student written tests. We calculated the defect-detection capabilities for assignments as described in Section 4.1.2. Afterwards, we

performed statistical comparisons of potential test quality measures. In total, 10 test quality measures were evaluated against defect-detection capability. The 10 measures can be categorized into four types:

- All-pairs execution,
- Mutation analysis,
- Checked code coverage, and
- Coverage criteria (4 measures).

In mutation analysis, we used mutants generated by four sets of mutation operators that appear practical to use in classroom tools. These sets are: Javalanche's mutation operators, the sufficient mutant operators identified by Offutt [46], SDL, a combination of VDL,CDL and ODL. For evaluating coverage criteria, we used statement coverage, branch coverage, OIC and OBC. Each of these measures were compared against each student-written test suite's defect-revealing capability. Two different projects or assignments are used for experiment. Comparing the correlation between scores for both projects provide us information about which of the measures are significantly correlated. We analyze the relationship (correlation) between test quality measures with defect-revealing capability using Spearman's ρ.

4.2.1 Quasi-Independent Variables

This study was a quasi-experiment. We did not exercise randomization rather chose projects that were realistic enough to represent practical CS2 classroom assignments. Our independent variables were assignments, authors of the test cases (i.e., students), and defect-detection capability.

We considered two CS2 assignments that were designed to teach students programming concepts (e.g., object-oriented features), data structures, algorithms, etc. The assignments were given in two different semesters. Participation of different student groups in different assignments from different semesters addressed or nullified the effects of personal interest, changes in the knowledge of testing, and expertise for specific types of assignments, such as GUI assignments. Thus, the results of our experiment were likely not to have bias. The defect-detection capability was used as a benchmark. We measured

performance of the dependent variables based on how accurate they were in predicting defect-detection capability scores of the test cases.

4.2.2 Dependent Variables

Dependent variables in our study were the scores calculated using 10 different test quality measures. In this experiment, our dependent variables were continuous. The scores from potential test measures were analyzed against the defect-detection capability of the tests.

4.2.3 Statistical Analysis

In this experiment, our quasi-independent variables were categorical and dependent variables were continuous. We observed that the distribution of the scores from some measures were not normal. However, we did not perform any statistical tests for normalcy. Spearman's ρ is an appropriate method for distributions that may not be normal. So, we compared all the pairs in one step and calculate correlation using the Spearman's ρ method. We applied a Bonferroni correction [10] to reduce the potential risk of the type I error . We adjusted significance level at 0.0045 to apply the Bonferroni correction for the calculation of correlations. An analysis of the results is presented in Section 4.3.4.

4.3 Experimental Results from Two Assignments

Evaluating all 10 test measures for both of the assignments in this study, required a significant effort—it involved compiling and analyzing the results of approximate 1,971,073 test case executions. The same assignments were used to evaluate all-pairs execution and mutation analysis in Section 3.1.3 and Section 3.2.3 respectively.

4.3.1 Master Test Suite Data

Project 1 had 147 submissions containing 1148 valid student written tests. For this assignment, the reference solution had 27 tests. Thus, the total number of test cases from students and from the instructor was 1175. Removing redundant tests reduced this to 540 test cases. Out of 540 test cases, 8 tests were solely from the reference tests and 496 were from students (these may have overlapped with some reference tests as well). The total

Table 4: Master suite summary.

	Project 1	**Project 4**
Submissions	147	101
Student-written Test cases	1540	2176
Valid tests	1148 (74.54%)	1476 (67.83%)
Reference Tests	27	82
Redundant Tests	670 (57%)	811 (52%)
Master Suite	505	747
Total test case runs	74235	75447
Total failures observed	19712 (26.55%)	16783 (22.24%)
Failures observed by student-written tests	19303(97.92%)	15451(92.06%)

number of test case failures detected by students were 19,303. The average number of program failures detected by a student-written test case was 38.9. The 8 reference tests in the master suite were not overlapped by any student-written tests. They detected 409 defects. The average number of program failures observed these tests was 51.2.

In Project 4, the total number of student submissions was 101 with 1,475 valid tests. The instructor provided 82 reference test cases. After removing redundant tests we found 747 tests: 681 from students (with overlapping reference tests) and 66 solely from reference tests. By applying the bytecode transformation strategy to the master test suite, it was possible to apply all of its tests to every student program. Every student program contained at least one defect. The total number of defects detected by students' tests, and the reference tests (having no overlap with students' test cases) of the master suite were 15,451 and 1,332 respectively. The average number of defects detected by student-written tests was 22.6 and by the un-overlapped reference tests was by 20.1, as summarized in Table 4.

It may appear that student-written tests may miss some possible defects. However, valid student test cases detected all the mutants generated from four all different sets of

mutation operators. This implies that student-written tests were strong enough to cover all the defects from mutation analysis as well as all-pairs execution. However, some equivalence class of defects were not detected by students' tests. The master suite covered those defects (with the reference tests) also.

4.3.2 Approximating the Number of Detected Defects

To calculate defect-detection capability we computed a weighted average over all master suite test cases that are duplicated within one student-written test suite. Figure 13 shows the distribution of defect-detection capability scores for Project 1. Note no student suites had scores above 3%, which is surprisingly low. Also, about half of the students scored 0.75% to 1.25%. Only 12 students out of 137 achieved above 2%.

Upon careful examination, such low scores makes sense. The total number of failures observed by the master suite was 19,712 whereas on average a student's suite observed 195.8 failures. Thus, the average defect-detection capability score was 0.99%. However, the set combining all the student-written test cases observed 97.9% of all the failures observed by the master suite. This result implies that students wrote diverse test cases but their individual tests were not strong enough to detect all the failures they made.

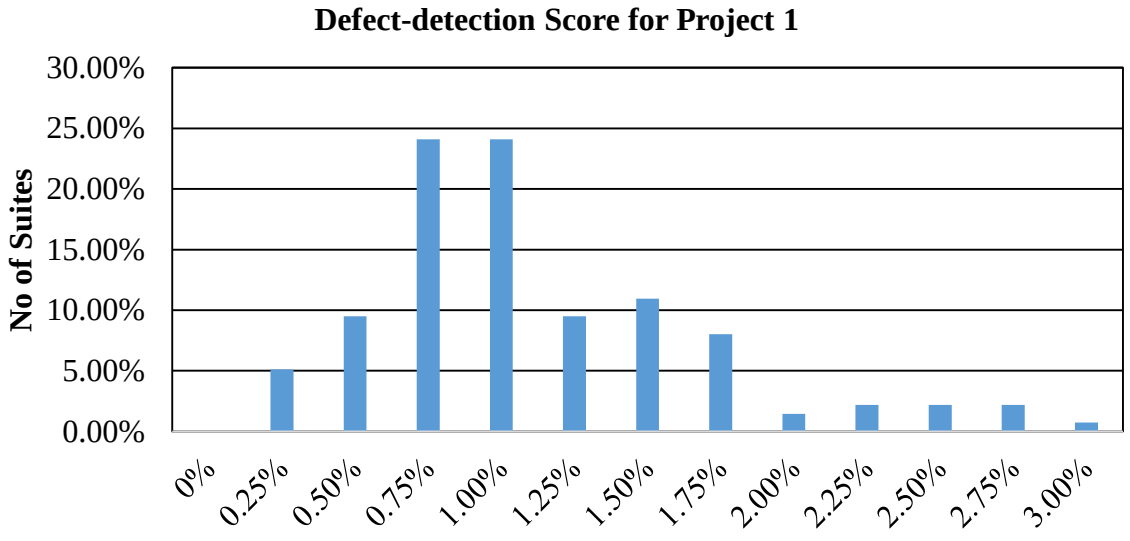


Figure 13: Distribution of defect-detection capability for Project 1.

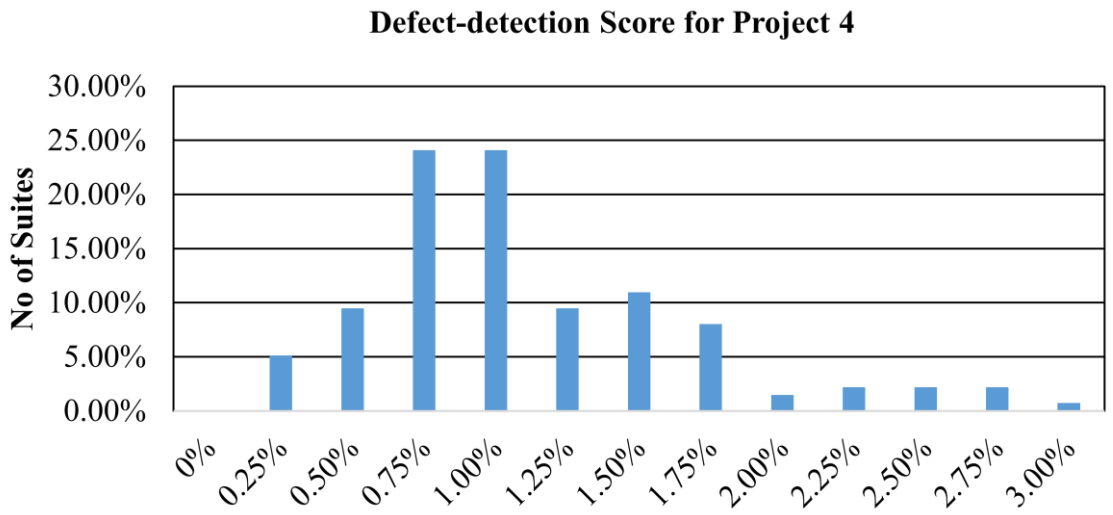


Figure 14: Distribution of defect-detection capability for Project 4.

Project 4 defect-detection capability scores have a similar distribution, though more skewed towards lower scores (Figure 14). The highest score achieved by students is 4%. About one third of the students scored 0.5% to 1.5%. The average defect-detection capability score was 1.2%. No student-written test case observed more than 692 failures (4.1%). We observed somewhat similar results in our earlier study. Students do "happy path" testing [29] by writing tests to show that their code worked rather than designing tests to execute code in a way to cause failures and to detect these failures. This is not surprising because our students are generally beginning testers described as "level 1" testers [7] by Ammann and Offutt. Their purpose of testing is to show correctness.

4.3.3 Analysis of Scores from Different Measures

We collected statement and branch coverage data from Web-CAT for Project 1 from fall 2011 and for Project 4 given during Spring 2011 using Clover [1]. Object Instruction coverage (OIC) and object branch coverage (OBC) data for both the projects have been calculated from xml data obtained from JaCoCo [3]. We generated mutants using four different sets of operators: Javalanche's mutation operators, the sufficient mutation operators, SDL, and a combination of VDL+CDL+ODL operators as described in Section 3.2.3. We generated sufficient, SDL, VDL+CDL+ODL mutants from µJava. Checked coverage scores for both the projects have been collected from our semi-

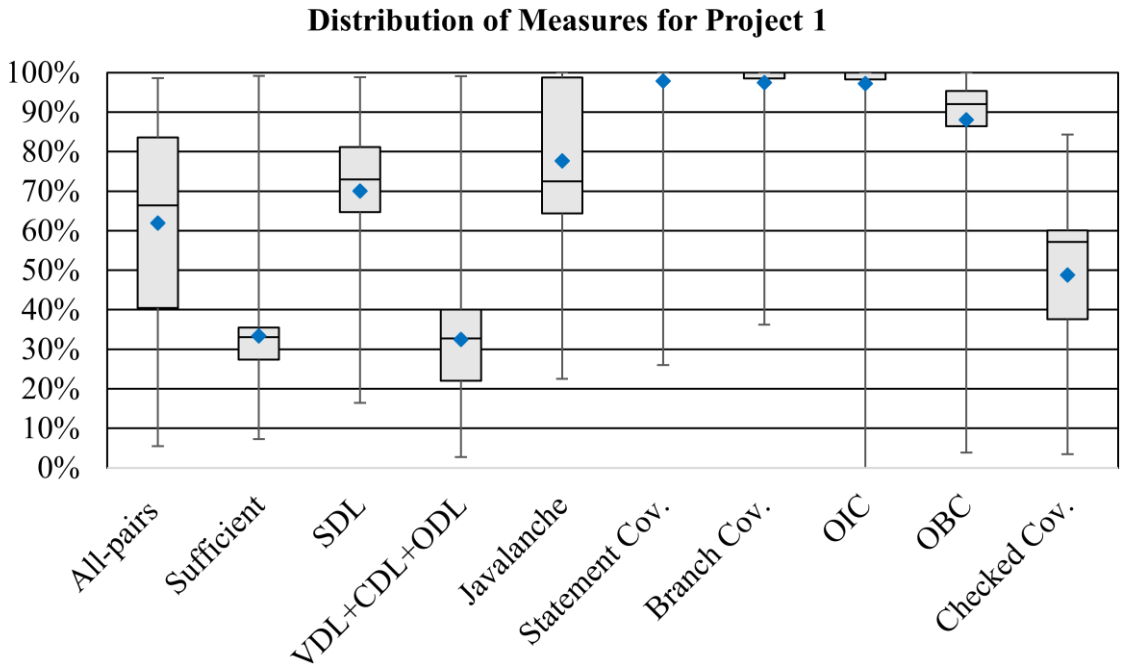


Figure 15: Distribution of Scores from different measures for Project 1.

automated system using JavaSlicer. We calculated all-pairs scores as demonstrated in Section 3.1.3.

Figure 15 shows the distribution of scores from different measures for Project 1. Displayed values are max, min, median, mean, 75th and 25th percentile. As we can see, the lowest and the highest all-pairs scores are 2% and 98% respectively. However, two-thirds of the students scored 40% to 70%. Sufficient mutation scores have similar max and min values although two thirds of the population is distributed in a very small range (from 25% to 35%). SDL scores are higher than sufficient and VDL+CDL+ODL mutation scores. Javalanche's mutation scores are higher than all other mutation operators. This result implies that student-written tests are good in detecting mutants generated from the modification of statements but not effective against mutants generated by the modification of operators or variables.

Coverage scores are much higher than mutation scores. Statement and branch coverage scores have almost no distribution as they have the same value, 100% as mean, median, 75th, and 25th percentile value. This is expected as students were graded on

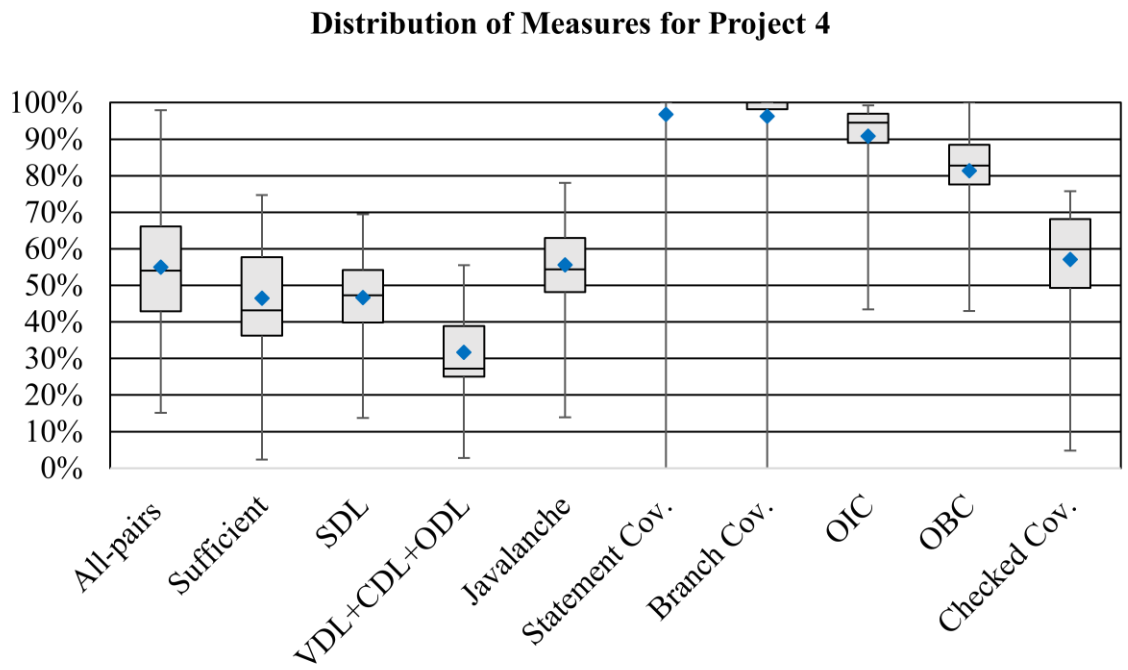


Figure 16: Distribution of different measures for Project 4.

statement coverage. OIC and OBC scores are lower than statement and branch coverage scores. Checked coverage score distribution is different from any other measures. Two thirds of the population scored 35% to 60% and no one achieved 100%. The scores for other coverage measures and checked coverage imply that students executed almost all of their code from their tests but have checked only half of their executed code against the expected results. We discuss each distribution individually later.

The distributions of scores for Project 4 calculated by different measures are depicted in Figure 16. All-pairs distribution has lower scores for mean and median values than Project 1. It implies that students detected fewer failures in Project 4 than in Project 1. This makes sense as Project 4 is more complex than Project 1 and students are likely produce more defects in complex projects. The four coverage scores and checked code coverage scores have very similar distributions to Project 1. However, the distributions of all four sets of mutation scores are much different from Project 1. No one scored 100% in any one of the mutation scores. SDL and Javalanche scores have similar spans. The distribution of each measure is described later with an individual chart.

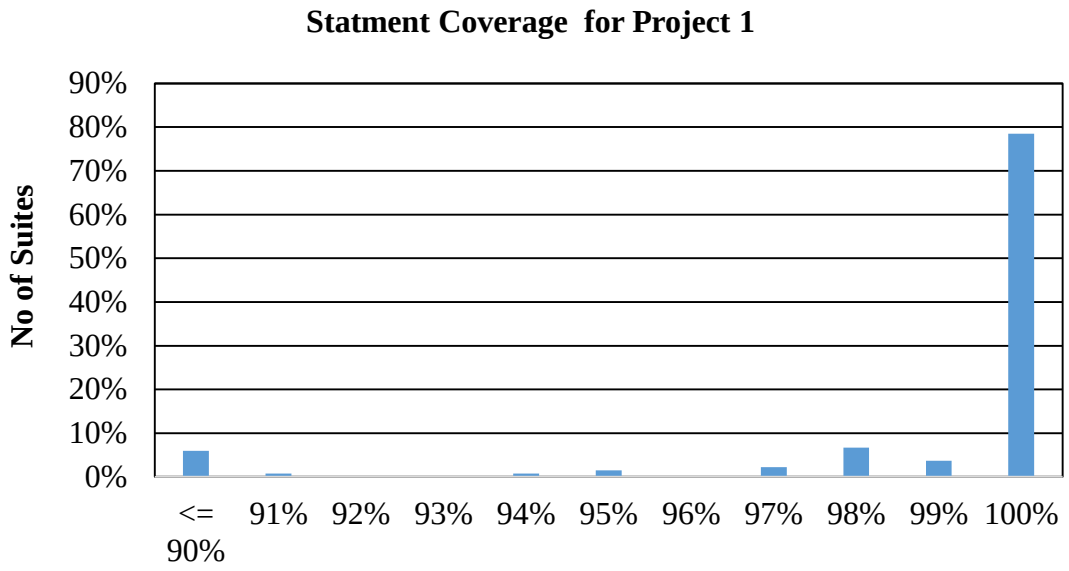


Figure 17: Distribution of statement coverage scores for Project 1.

Figure 17 shows the distribution of statement coverage for Project 1. This distribution is heavily skewed towards 100% coverage. Only 8 students out of 137 scored below 90%. The average score is 98.9%. Branch coverage and object instruction coverage distributions are presented in Figure 18 and Figure 19. As we can see these distributions are almost the same as the distribution of statement coverage. The average branch coverage

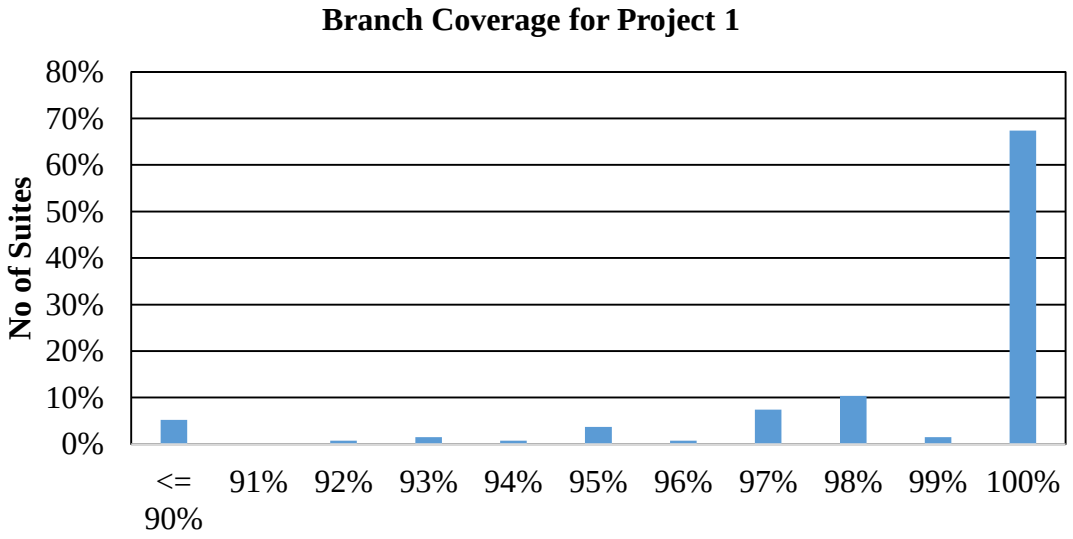


Figure 18: Distribution of branch coverage scores achieved by student-written test suites for Project 1.

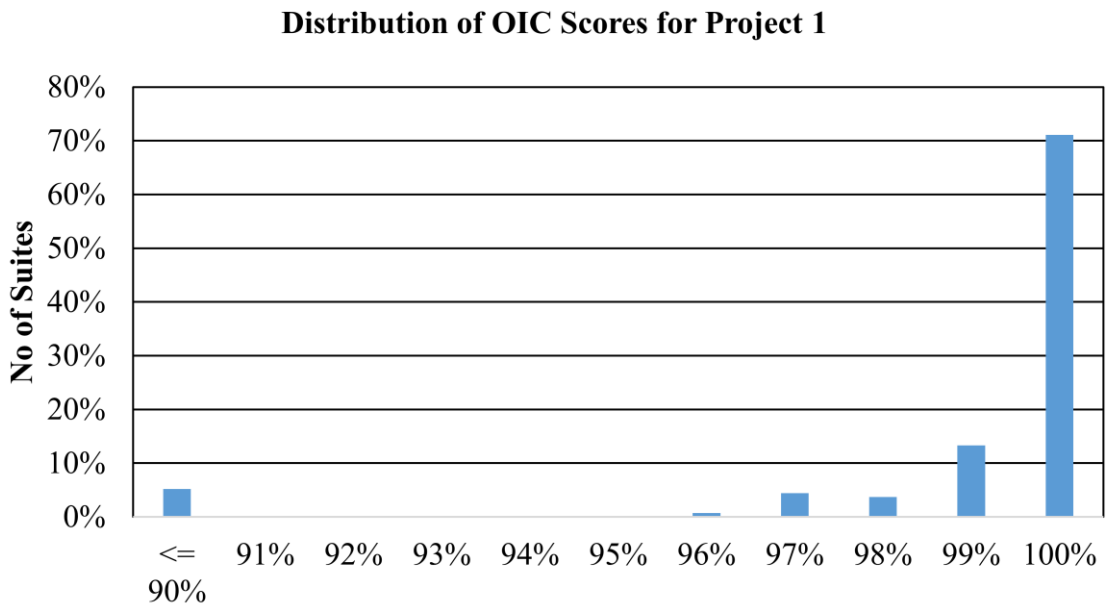


Figure 19: Distribution of OIC scores achieved by student-written test suites for Project 1.

score is 97.4% and the average OIC score is 97.2%. This is not unexpected as object instruction coverage requires each bytecode instruction to be executed to be counted as covered and branch coverage requires execution of each branch with both true and false values. Thus, when most of the branches are covered (high coverage) it is likely that corresponding bytecode instructions will also be covered. For both branch and OIC coverage only 7 students scored below 90%. About one fifth of the students scored 91% to 98% in branch coverage and the remaining students scored 100%. In OIC about one fifth of the students scored 99% and exactly 96 students scored 100%.

Figure 20 summarizes the distribution of statement coverage scores for Project 4. Note the extremely skewed bucketing in the histogram, where the first column represents all suites with coverage score less than or equal to 90%. More than half of the students achieved 100% statement coverage. This should not be surprising, since students got feedback on statement coverage, and it was used directly in scoring their work—students needed to maximize their statement coverage to get the maximum grade on the assignment. However, about 5% students achieved less than 10% coverage regardless of the feedback.

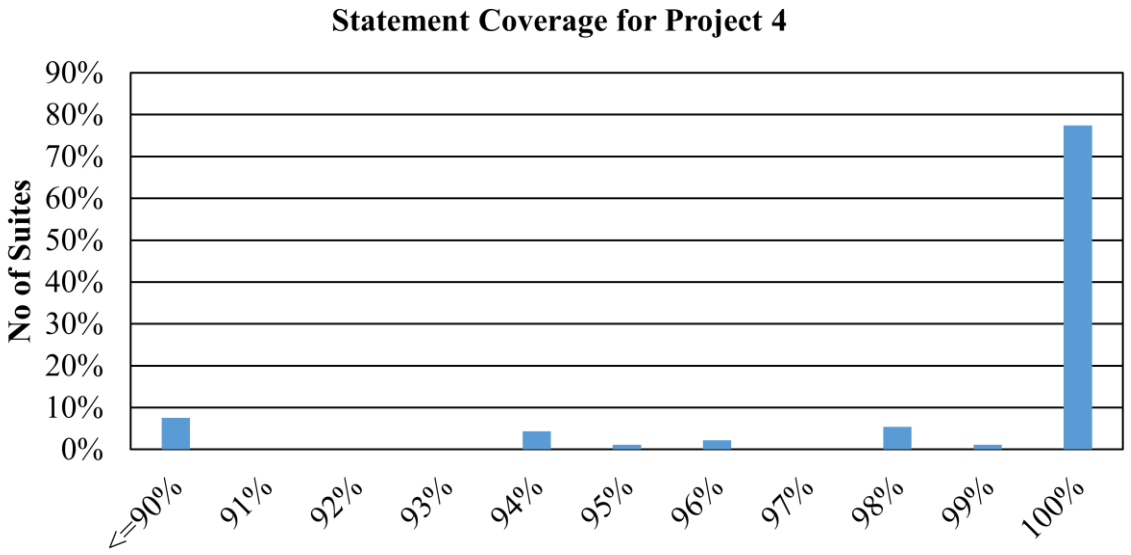


Figure 20: Distribution of statement coverage scores achieved by student-written test suites for Project 4.

This distribution is very similar to the statement coverage distribution for Project 1.

Branch coverage score distribution (Figure 21) for Project 4 is almost similar to the distribution of statement coverage. More than 60% of the students scored 100% and about one fifth of the students scored 91% to 98%. Only 7 students scored below 90%.

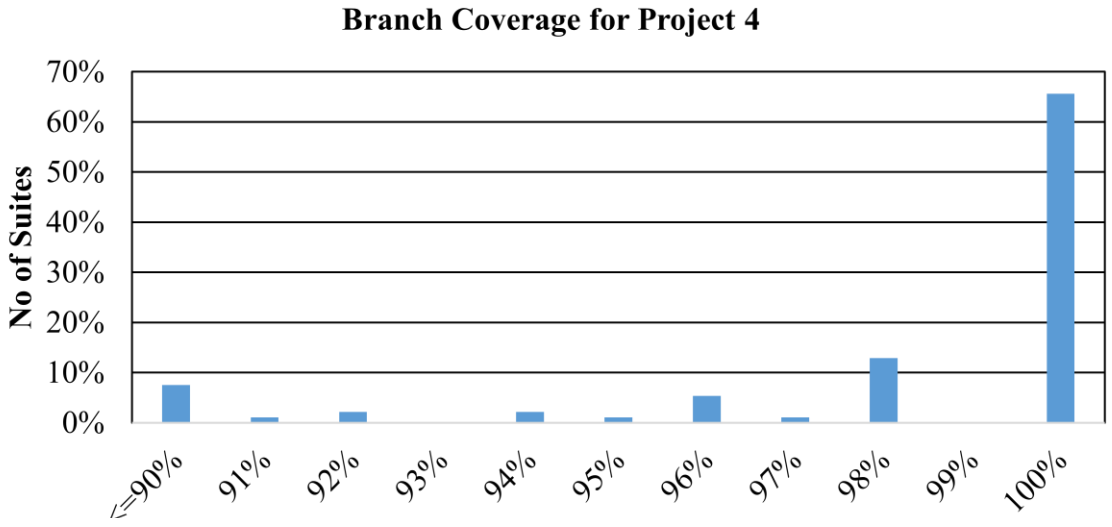


Figure 21: Distribution of branch coverage scores achieved by student-written test suites for Project 4.

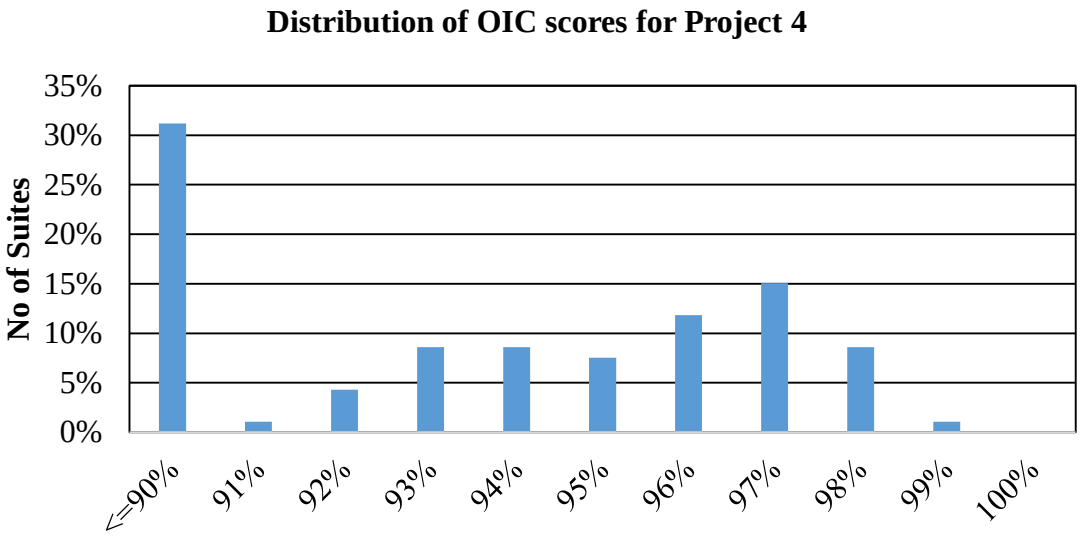


Figure 22: Distribution of OIC coverage scores for Project 4.

The distribution of OIC for Project 4 is presented in Figure 22. About 60% students achieved 91% or higher. However, no one was able to score 100%. One third of the students scored below 90%. We used the same scale to present distribution as we did for statement and branch coverage. So, it is not visible in the scale but no student scored below 40%. The average score was 90.8% which is expected, as students received feedback on

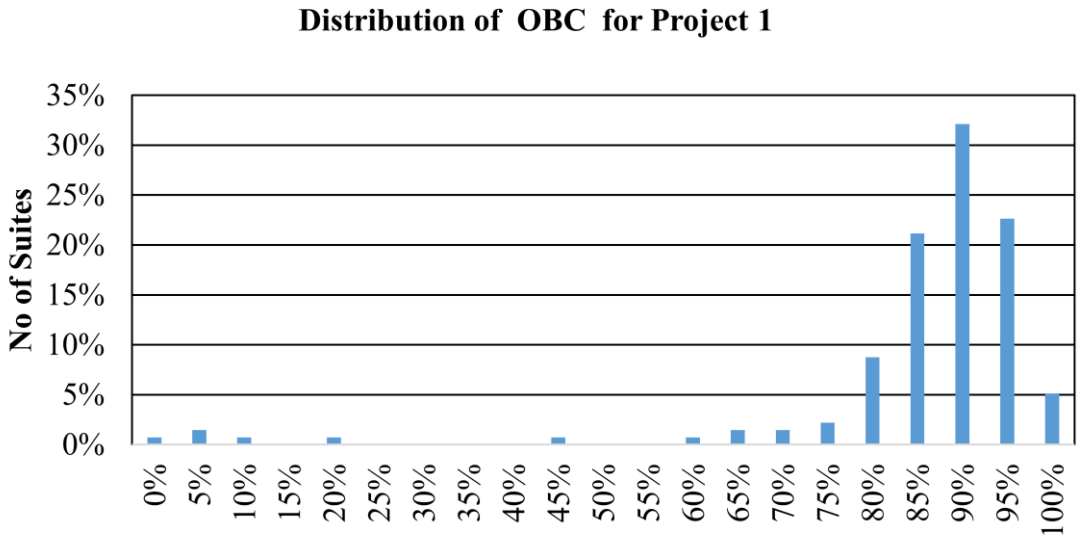


Figure 23: Distribution of OBC scores achieved by students-written tests for Project 1.

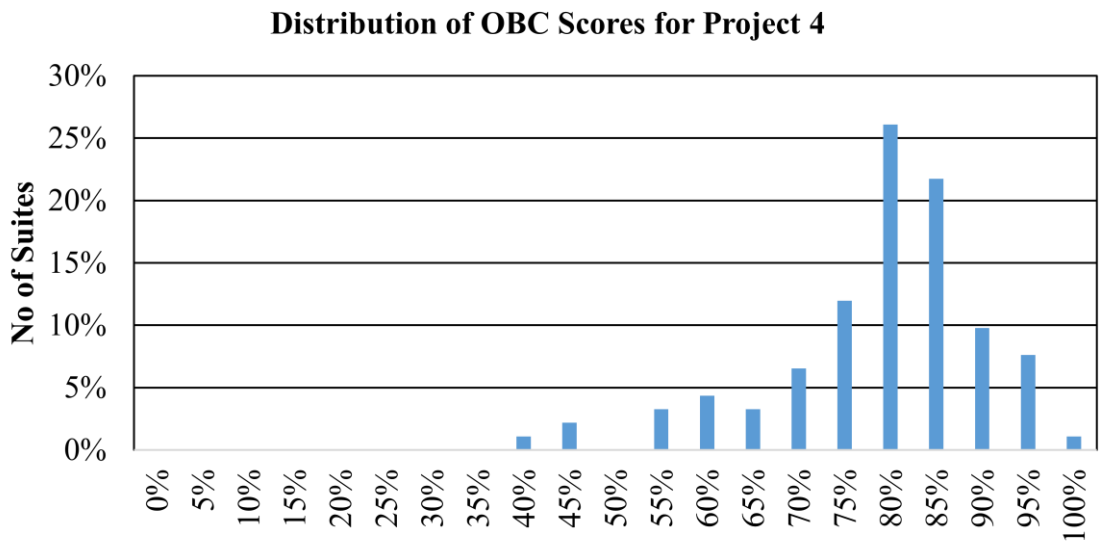


Figure 24: Distribution of OBC scores for Project 4.

their statement coverage and made efforts to achieve high statement coverage. Generally, high coverage on statement results in high coverage in bytecode instruction coverage also.

The distribution of OBC scores for Project 1 is shown in Figure 23. Out of 137 students, 7 students achieved 100%. The average score was 87.47%. One third of the students scored 90% to 95%. Though the average score was high, one student scored 0 and 16 students got less than 75%. However, keeping in mind that this was the first programming assignment and achieving a good OBC score is harder than achieving good statement coverage, students on average performed well.

Figure 24 shows OBC score distribution calculated from JaCoCo for Project 4. Although only one test suite in this group achieved 100% coverage, three quarters of the test suites achieved 88% or better (mean of 81%). Achieving high OBC score is more difficult than getting high statement coverage. However, as mentioned before students got feedback on statement coverage and tried to increase coverage. So, it is likely that they will increase OBC scores also. Interestingly, no student achieved lower score than 42% unlike Project 1.

Figure 25 summarizes the distribution of all-pairs scores achieved by student-written test suites for Project 1. The average score was 61.5%. Two-thirds of the students achieved above 50%, but no one scored 100%. This distribution is much different than the

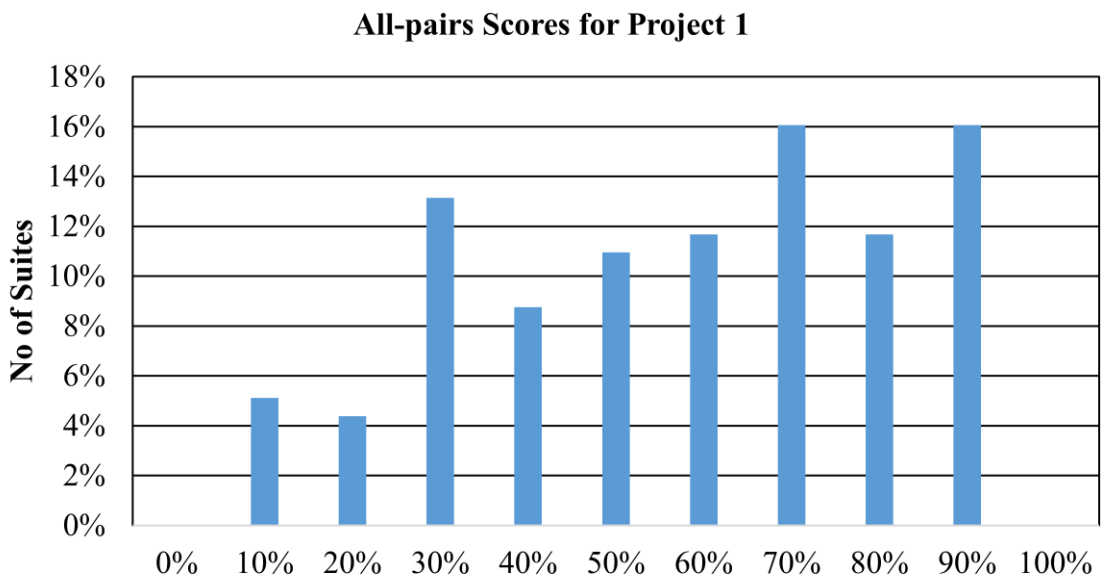


Figure 25: Distribution of all-pairs scores for Project 1.

distribution for Project 4 as shown in Figure 26. In Project 4, one third of the students achieved above 50%. The average score was 55% which is less than Project 1. We find the appearance of a normal distribution around this mean. This seems reasonable as Project 4 was more complex than Project 1. Thus students are likely to make more mistakes.

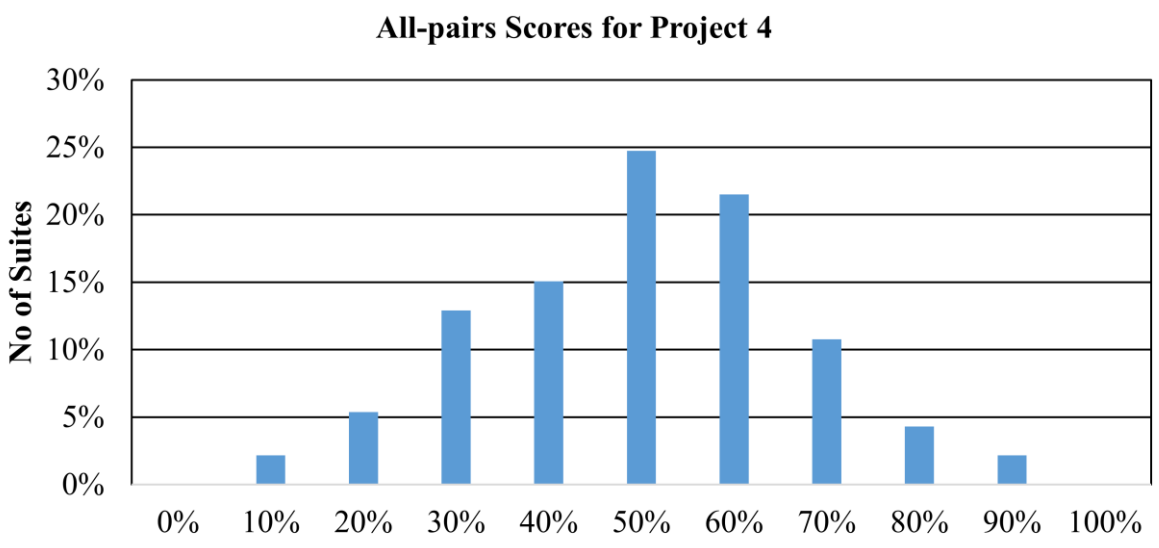


Figure 26: Distribution of all-pairs scores achieved by student-written test suites for Project 4.

From Figure 25 and Figure 26, it seems that the all-pairs measure is more sensitive than OBC or statement coverage, since the scores are somewhat lower. Further, the all-pairs measure is somewhat more selective, since no students achieved the maximum value. Most importantly, it seems that this measure most closely matches the goal of determining how well a suite can detect real-world defects, since it is a direct measure of how many defective programs each suite was able to detect (in this case, at least, since all programs contained observable defects).

It is also worthwhile to examine why, even though the defect-detection capability score is based on the same programs as the all-pairs analysis, the results for Project 1 in Figure 13 and Figure 25 and for Project 4 Figure 14 and Figure 26 are so different. To understand these differences, remember that Figure 13 and Figure 14 show the approximated probability of a test suite detecting any specific defects (or failures within a small equivalence class) in a program. Figure 25 and Figure 26, on the other hand, corresponds to whether the test suite detected any failure for the same program. The defect-detection capability corresponds to a test suite's ability to detect each of these failures in isolation, rather than any one of them when considered all together. So, while any given test suite might have a low probability of finding a specific failure, the suite still will certainly reveal failures covered by the equivalence classes it does include.

We used the mutation testing results for both Project 1 and Project 4. Mutants are generated from two different tools-Javalanche and µJava. As we mentioned in Section 3.2.2 we had to modify both of the tools to generate mutants from the reference solution. In Project 1, total number of Javalanche generated mutants were 160. All the mutants were detected as non-equivalents by at least one valid student-written test. µJava generated 319 mutants: sufficient mutants 124, SDL mutants 85 and combination of VDL+CDL+ODL mutants 110. Students' tests were able detect all the µJava mutants as non-equivalent mutants.

The total number of mutants generated for Project 4 from Javalanche was 316 mutants. During the analysis, every single mutant was empirically determined to be non-

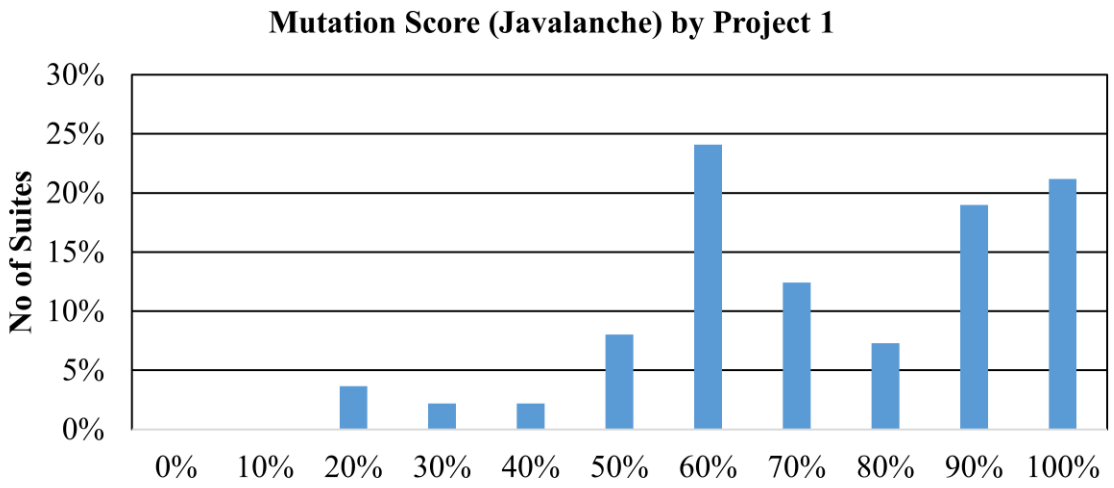


Figure 27: Distribution of Javalanche mutation scores achieved by student-written test suites for Project 1.

equivalent to the original program (i.e., every mutant failed at least one test case). Out of 316 non-equivalent mutants, 59 mutants were generated from modification of "assert" statements in the reference program. Assert statements are part of test oracles. So, we removed these 59 mutants from the total set. For the same assignment, µJava generated 627 mutants in total which is about double the number of Javalanche mutants. Out of 627 mutants, sufficient operator mutants were 388, SDL mutants were 131, and the remaining 108 mutants were generated from VDL, CDL and ODL. All the mutants were categorized as non-equivalents by the valid test cases. As we mentioned in Section 3.2.2 that µJava cannot handle assert statements, and so we commented out asserts from the program during mutants creation. Later we add assert statements as a patch in the mutants. Thus, we ensured that test oracles in asserts were not mutated.

Figure 27 summarizes the results achieved, showing the distribution of Javalanche mutant kill ratios achieved by students' tests written for Project 1. Again, as with the all-pairs strategy, it appears that mutation analysis is more sensitive than coverage criteria. The scores appear to be skewed towards 60% or above. In fact, the average score is 77.2% which is promising for beginners. More than two thirds of the students achieved over 70%. On the other hand, in Project 4 Javalanche mutation kill score (shown in Figure 28) is roughly normally distributed, with an average of 55%. Also like the all-pairs scores, the

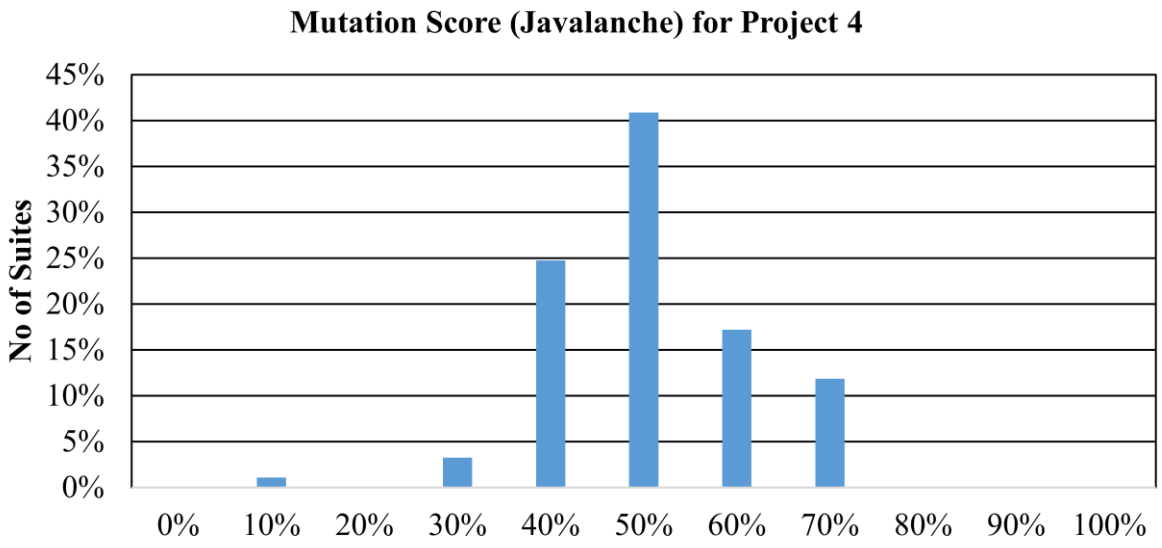


Figure 28: Distribution of Javalanche's mutation scores achieved by student-written test suites for Project 4.

mutation kill ratios appear to be more selective, since no student achieved the maximum value. Also for both Project 1 and Project 4, we found Javalanche mutation kill scores are strongly correlated with all-pairs score. This is reasonable as all-pairs presumably presents mutants created by students and Javalanche provides system generated mutants.

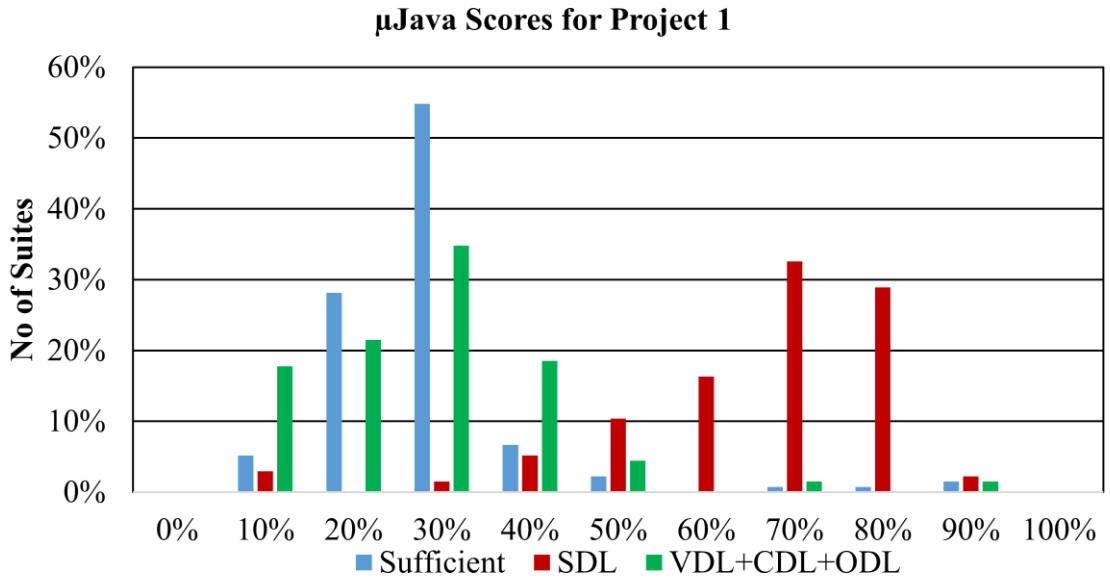


Figure 29: Distribution of µJava mutation scores achieved by student-written test suites for Project 1.

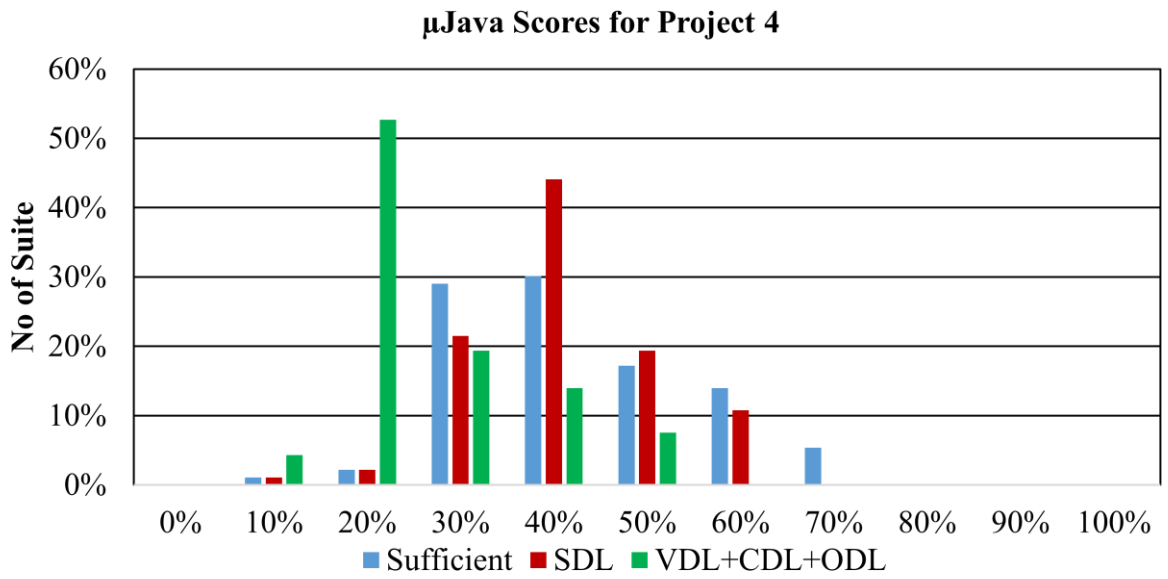


Figure 30: Distribution of µJava mutation scores achieved by student-written test suites in Project 4.

Figure 29 shows the distribution of µJava mutants kill scores for Project 1 achieved by students. The average score for sufficient, SDL, and combination of VDL+CDL+ODL are 33.4%, 70% and 32.5% percent respectively. No one scored 100% in any one of the three sets of mutants. Only 2 students scored above 90% in all three sets. We find some similarities between the distribution of sufficient and VDL+CDL+ODL mutants kill scores. About half of the students scored 20% to 30% in both sufficient and VDL+CDL+ODL mutation. SDL scores looks different than these two sets. Half of the students scored 70% to 80% in SDL.

For Project 4, the average score for sufficient, SDL, the combination of VDL+CDL+ODL operators were 46.5%, 46.6% and 31.6% respectively. The distribution of all the types of mutation kill scores are similar as shown in Figure 30. Both for µJava and Javalanche mutants, no students achieved above 80% in this project which suggested achieving higher score in mutation become harder for complex programs.

Checked coverage scores were then calculated as described in Section 3.3. Figure 31 and Figure 32 show the distribution of checked coverage scores for Project 1 and Project 4 assignments respectively. As expected, checked coverage scores were lower than other coverage scores, including OIC, and OBC. Moreover, the distribution of scores for the two

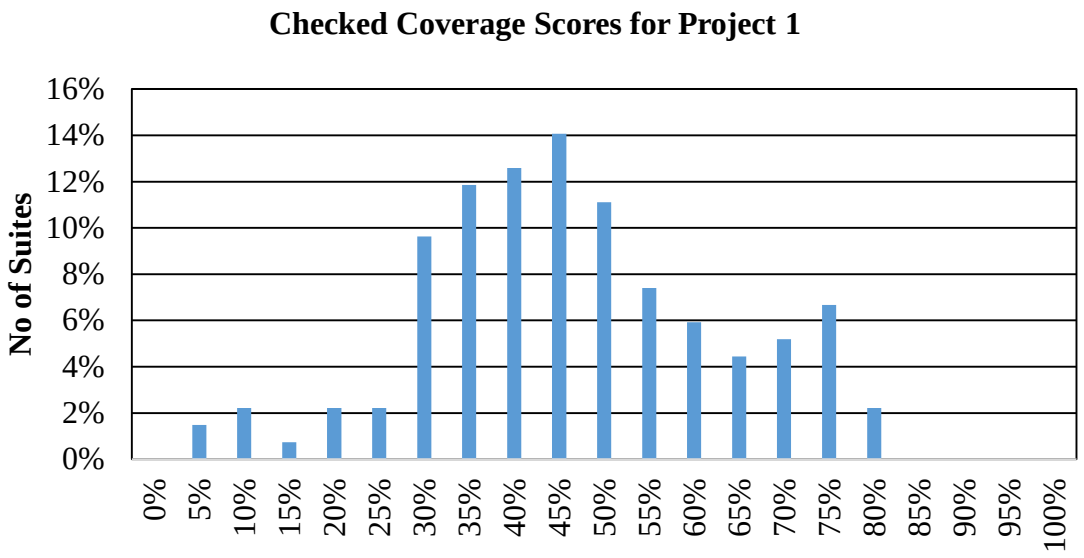


Figure 31: Distribution of checked coverage scores for Project 1.

projects are not similar. In Project 1, over half of the students' scores fall in range 30% to 55% where as in Project 4, one half of the suites scored between 50%-68%. The average score for Project 1 was 48.7%, whereas for Project 4 average was 57.1%. This observation is important because for both OIC and OBC we found the average Project 1 scores were higher than the average Project 4 scores. This result suggests that beginners' tests include less assertions than students who are more accustomed to testing.

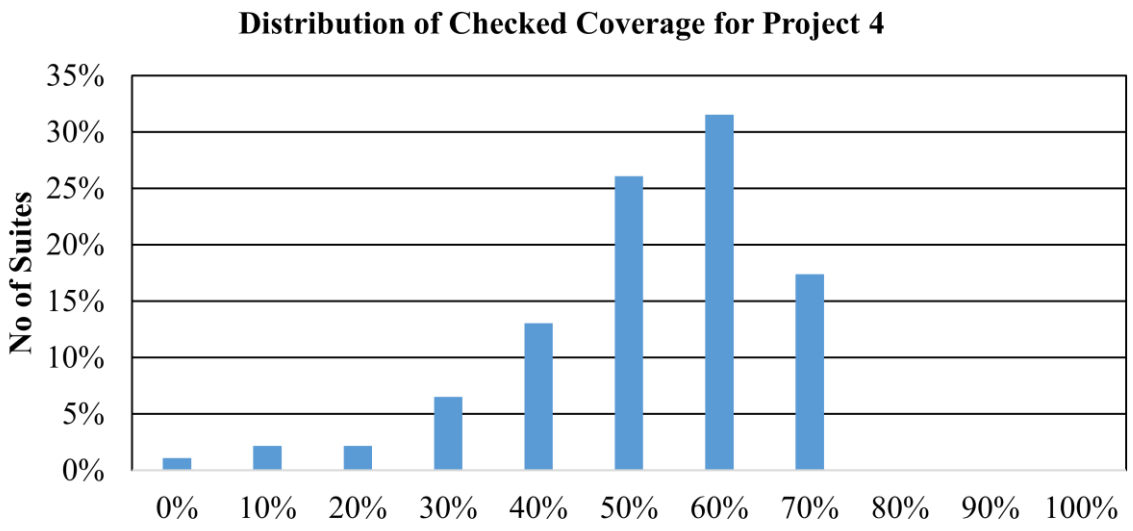


Figure 32: Distribution of checked coverage scores achieved by student-written test suites for Project 4.

Table 5: Correlation (Spearman's ρ) between measures for Project 1. (bold * entries are statistically significant at α < 0.0045).

	St. Cov.	Bra. Cov.	OBC	OIC	Mutation				Checked Cov.	All-pairs	Defect-detection Score
					Javalanche	µJava (suff.)	µJava (SDL)	µJava (VDL /CDL/ ODL)			
St. Cov.	-	**0.78***	0.012	0.08	0.20	0.21	**0.23***	0.16	0.009	**0.35***	0.28
Bra. Cov.		-	0.05	0.15	0.25	0.17	**0.23***	0.12	0.03	**0.26***	**0.26***
OBC			-	**0.80***	0.13	0.14	0.17	0.13	**0.59***	**0.59***	0.16
OIC				-	0.07	0.12	0.08	0.04	**0.63***	0.22	0.18
Java-lanche					-	**0.54***	**0.73***	**0.53***	0.09	**0.32***	**0.42***
µJava (suff.)						-	**0.71***	**0.84***	0.14	0.14	**0.42***
µJava (SDL)							-	**0.64***	0.12	0.12	**0.54***
µJava (VDL/CDL/ODL)								-	0.12	0.12	**0.46***
Checked Cov.									-	0.29	0.22
All-pairs										-	**0.92***
Defect-detection Score											-

4.3.4 Evaluation

Based on the data summarized in Section 4.3.2 and 4.3.3, we performed a statistical comparison of the 10 test quality measures against each student-written test suite's defect-detection capability. Table 5 presents results from Project 1. We calculate correlations with a Bonferroni correction with **α** < 0.0045 to reduce inflation of the potential for type I errors. Of all the measures, all-pairs scores are the most accurate predictor of defect-detection capability with a correlation 0.92. The next best predictor is µJava SDL mutation scores. We also found VDL+CDL+ODL mutation scores correlated with defect-detection score reasonably well compared to other measures. Interestingly, Javalanche mutation scores for

this project seems as good as µJava sufficient mutation scores. Both of them have a correlation value of 0.42 with defect-detection capability. OIC, OBC and statement coverage scores are not statistically significant. Branch coverage scores are significant but not strongly correlated with defect-detection score. Checked coverage is more costly than branch coverage but is not a better predictor of defect-detection score. Though it seems statistically significant with $\alpha < 0.05$, after applying the Bonferroni correction no longer meets the criterion for significance.

We observe OBC has a strong correlation with all-pairs score though it is not statistically significant with defect-detection capability. µJava sufficient mutation scores are strongly correlated with µJava SDL scores and VDL+CDL+ODL scores as expected. Interestingly, Javalanche mutation scores have the strongest correlation with µJava SDL mutation scores. Checked coverage scores are strongly correlated with both OIC and OBC but not with statement or branch coverage.

Figure 33 depicts the correlation between defect-detection capability score vs. all-pairs score. A majority of the suites lie close to the trend line identified in the plot.

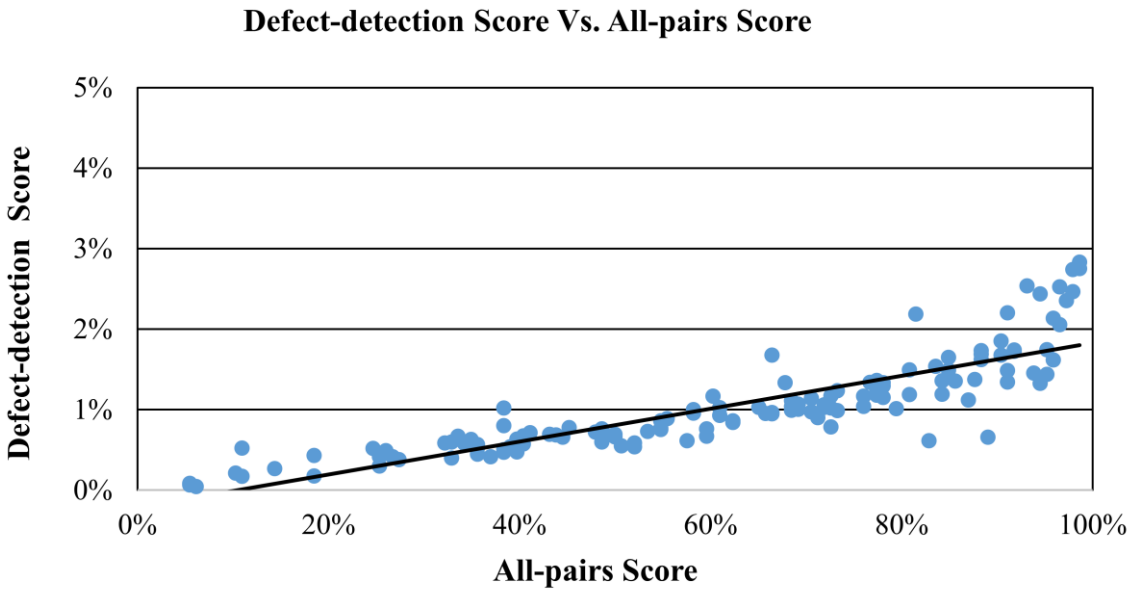
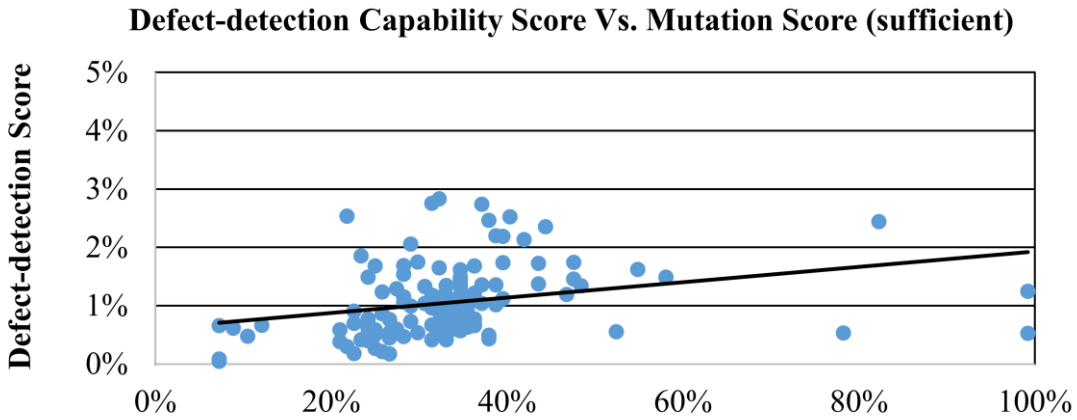
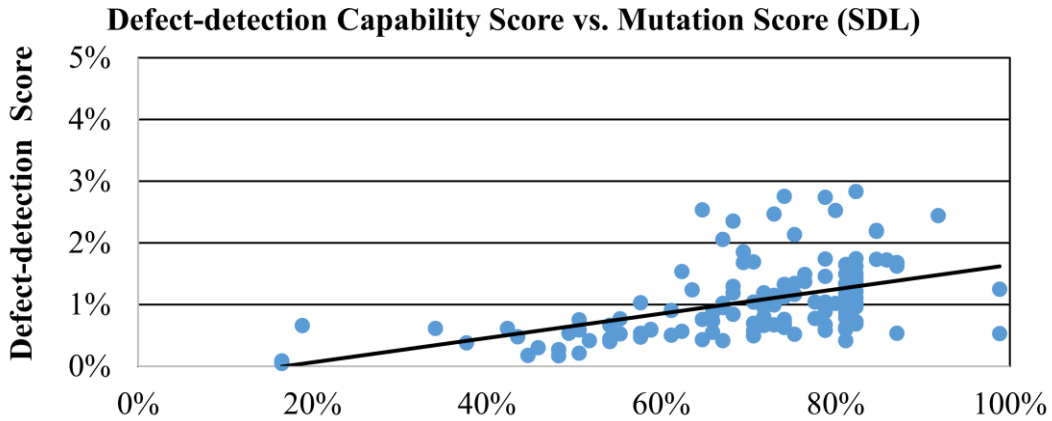


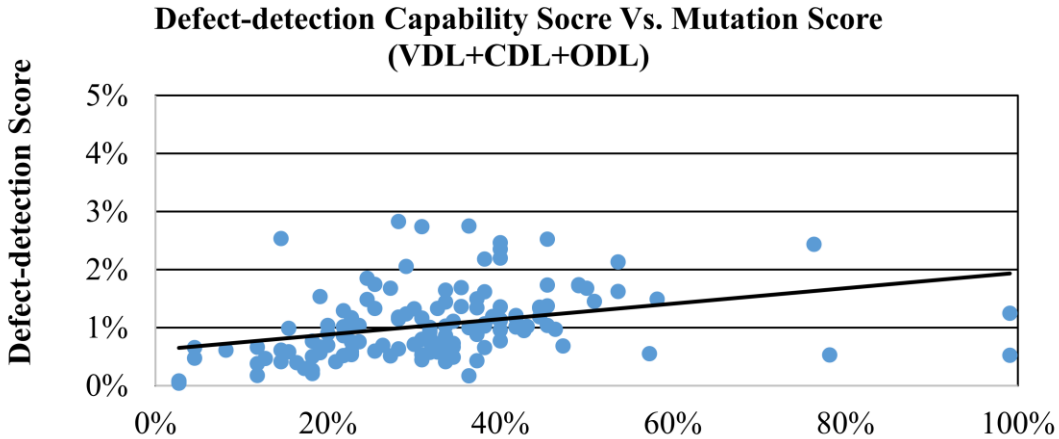
Figure 33: Relationship between defect-detection capability score vs. all-pairs score for Project 1.



(a) Sufficient



(b) SDL



(c) VDL+CDL+ODL

Figure 34: Relation between defect-detection capability estimates and µJava scores for Project 1.

However, a small minority of students achieved very high scores on both all-pairs and defect-detection capability score (of its scale). This group is visually identifiable on the defect-detection capability score (of its scale). This group is visually identifiable on the top right corner. Most importantly we can see the strong correlation between two scores in the figure.

The correlation between defect-detection capability score and sufficient mutation score is depicted in Figure 34(a). Most students' suites achieved mutation scores below 40% and defect-detection capability sores below 2%. The correlation is similar to the correlation of defect-detection capability vs. SDL mutation scores described in Figure 34 (b). However, the population is less clustered in SDL score distribution than sufficient score distribution. Many students' suites scores fall above and below the trend line but a strong correlation between SDL mutation scores and defect-detection capability scores is visible. Figure 34(c) shows the relationship between defect-detection capability and VDL+CDL+ODL mutation scores. Here also a large number of students achieve below 40% in mutation score and below 1.5% in defect-detection score. Out of the three mutation scores, SDL mutation score is the most strongly correlated with defect-detection capability.

Figure 35 shows the correlation between defect-detection capability score and Javalanche mutation score. This plot is much different than the relation plots of µJava

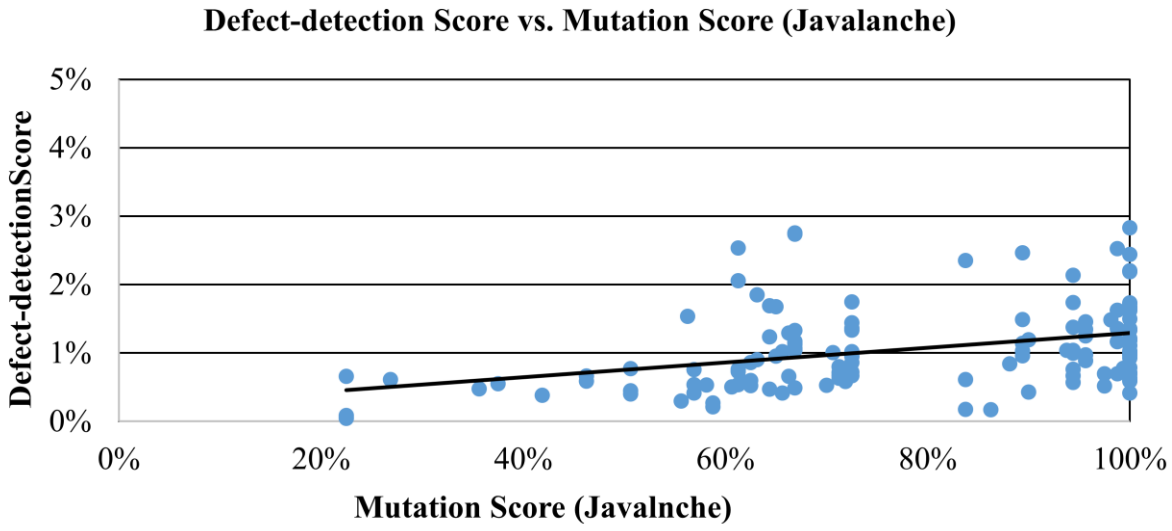


Figure 35: Relationship between defect-detection capability score vs. mutation score (Javalanche) in Project 1.

Table 6: Correlation (Spearman's ρ) between measures for Project 4. (bold * entries are statistically significant at α < 0.0045).

	St. Cov.	Bra. Cov.	OBC	OIC	Mutation				Checked Cov.	All-pairs Score	Defect-detection Score
					Javalanche	µJava (suff.)	µJava (SDL)	µJava (VDL /CDL/ ODL)			
St. Cov.	-	**0.80***	0.01	0.08	0.19	0.14	0.22	0.16	0.008	**0.31***	**0.30***
Bra. Cov.		-	0.05	0.15	0.19	0.14	**0.23**	0.15	0.03	**0.33***	**0.32***
OBC			-	**0.82***	0.13	0.14	0.17	0.13	**0.59***	0.19	0.16
OIC				-	0.07	0.11	0.08	0.04	**0.63***	**0.22**	0.18
Java-lanche					-	**0.88***	**0.96***	**0.88***	0.09	**0.75***	**0.72***
µJava (suff.)						-	**0.82***	**0.93***	0.14	**0.67***	**0.61***
µJava (SDL)							-	**0.84***	0.12	**0.78***	**0.76***
µJava (VDL/ CDL/ ODL)								-	0.12	**0.72***	**0.64***
Checked Cov.									-	**0.29***	0.23
All-pairs Score										-	**0.89***
Defect-detection Score											-

sufficient, SDL, and combined deletion mutation scores. A large number of students' test suites achieved a high score in Javalanche mutation scores, though their defect-detection capability scores vary from 0.05% (low) to 3% (high). However, a linear relationship between the scores of defect-detection capability and Javalanche mutation score is visible in the figure.

Table 6 presents results for Project 4. We applied the same Bonferroni correction for this project data also to reduce the risk of type I errors, and calculated correlations at α < 0.0045. For this project, all-pairs scores are most strongly correlated with defect-detection capability. The next best predictor is µJava SDL mutation scores with a correlation 0.76. So, for both the projects all-pairs performed the best and µJava SDL

mutants performed the second best. However, Javalanche mutation scores become the third best predictor unlike Project 1. We also found VDL+CDL+ODL mutation scores are statistically significant with strong creation of 0.64. Interestingly, µJava sufficient mutation scores did not perform as well as SDL or a combination of VDL+CDL+ODL. It is plausible that the type of the project or the problem students submitted had some impacts on the scores. For example, both Project 1 and Project 4 are data structure projects and students were required to perform more logical computations than arithmetic calculations. Thus, SDL mutants are showing stronger correlation with defect-detection capability than sufficient mutation sets.

Interestingly, both statement coverage scores and branch coverage scores for Project 4 showed statistical significance with defect-detection capability though the correlations are not strong. Checked coverage seemed statistically significant with $\alpha < 0.05$. However, after applying the Bonferroni correction the correlation does not meet the criterion for significance. OIC and OBC scores showed strong correlations with checked code coverage scores.

Figure 36 shows the relationship between defect-detection capability scores and all-pairs scores. As we can see the relationship is linear and strong as most of the scores lie on or close to the trend line. Only a few students scored high in defect-detection capability but

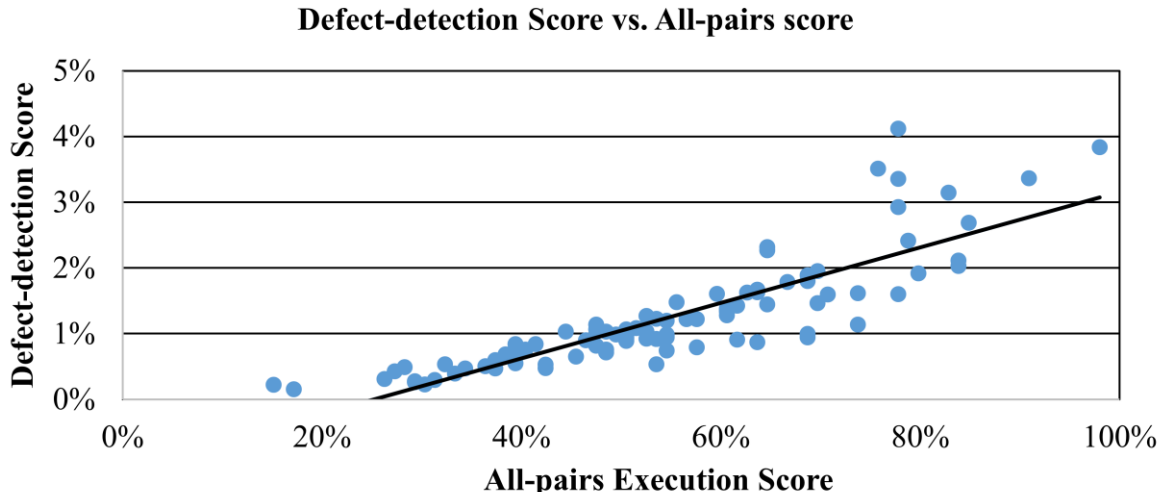


Figure 36: Relationship between defect-revealing capability estimates and all-pairs score for Project 4.

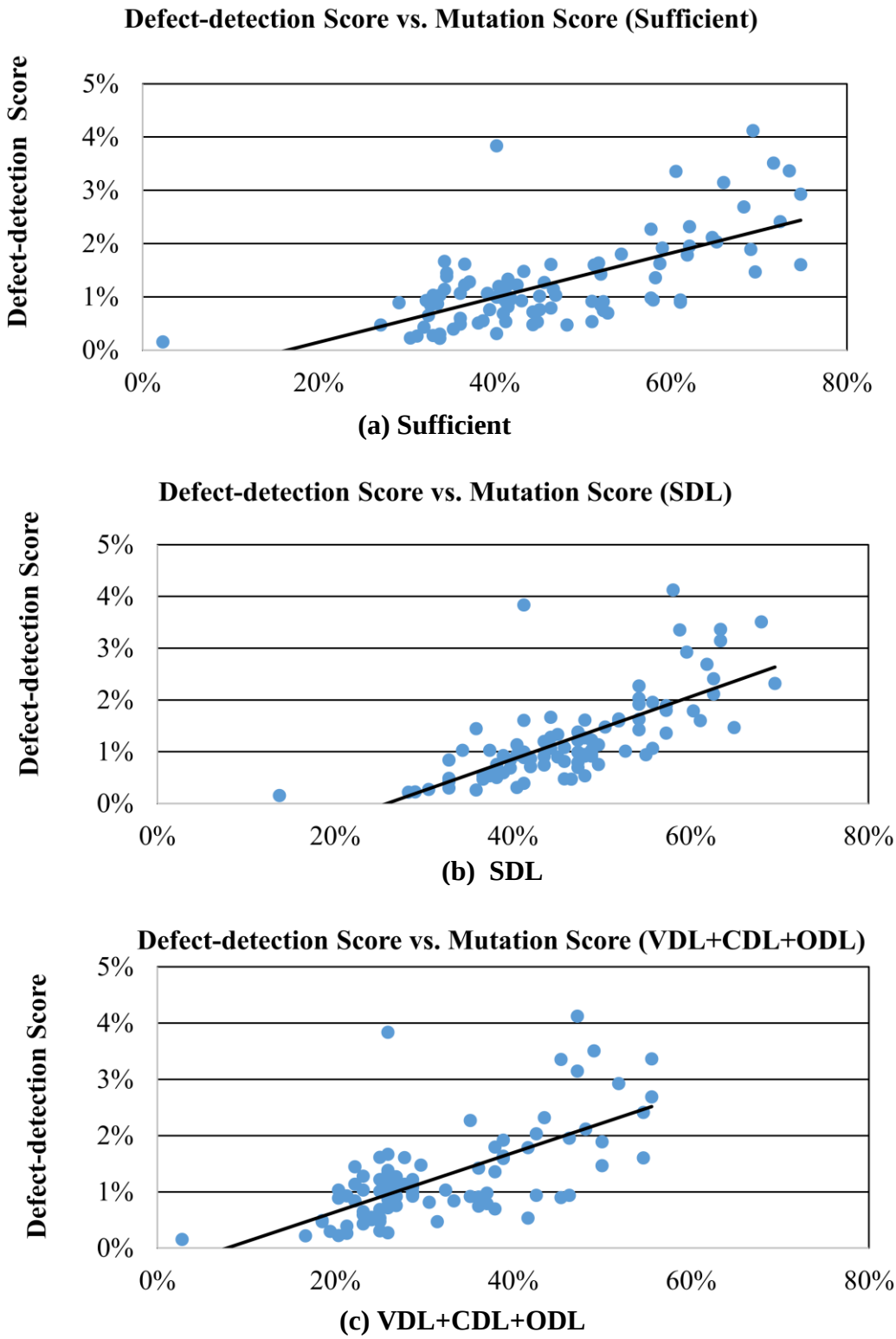


Figure 37: Relation between defect-detection capability estimates and µJava mutation scores for Project 4.

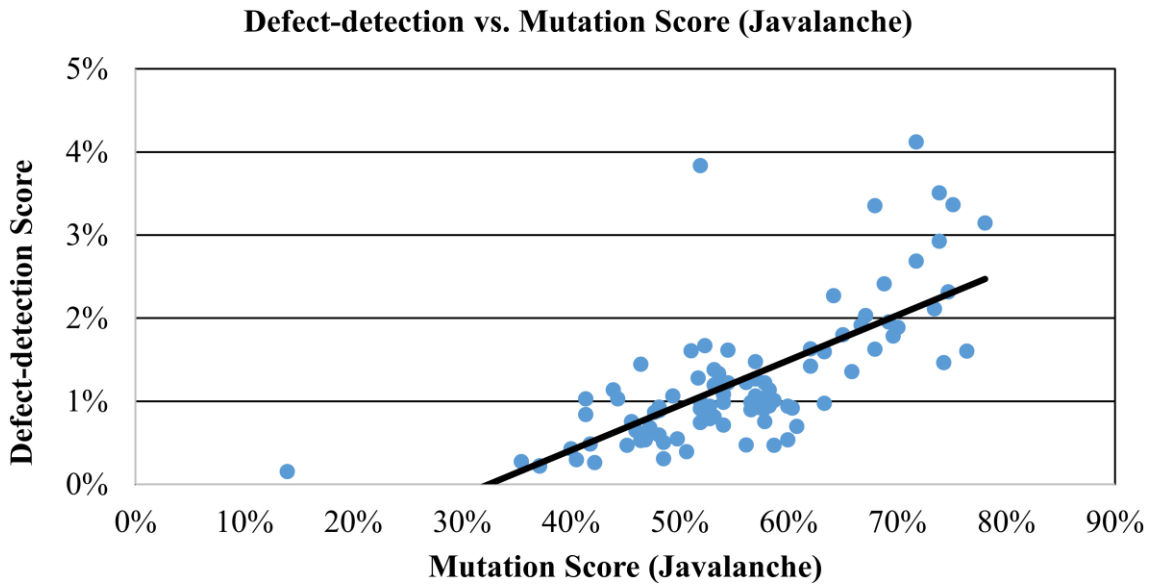


Figure 38: Relation between defect-detection scores vs. Javalanche mutation scores for Project 4.

moderately in all-pairs score. These scores are visible on the top right corner of the picture. Note the all-pairs and defect-detection capability relationship plot for Project 4 is similar to the plot for Project 1 (Figure 33). Figure 37 (a), (b) and (c) present the relationship between defect-detection capability with sufficient mutation score, SDL mutation score and VDL+CDL+ODL mutation score. All three of them have very similar correaltions, though SDL performed better than the other three. Students who achieve high mutation scores on SDL mutation also scored high in defect-detection capability. Only 10 students scored high in defect-detection capability but moderately on SDL mutation score. They are visible on the top right corner. The same students are found on top right of the trend lines in sufficent mutation and VDL+CDL+ODL mutation score plots as well.

The relation between Javalanche's mutation scores and defect-detection capability scores is depicted in Figure 38. The scores are clustered beween 30% to 80%. The sharp trend line suggests a strong linear relationship. However, the students' scores in Figure 38 are not as close to the trend line as they are in SDL mutation score plot shown in Figure 37

(b). We do not include the relationship plots of other measures, as they are not strongly correlated with defect-detection capability scores.

4.3.4 Threats to Validity

There are some limitations in our research. First, we used the same students' programs to calculate all-pairs execution and defect-detection capability. For master suite creation we used valid student-written tests and reference tests. This procedure may create a bias towards the failures that students' programs have. Creating a master suite covering different types of defects, such as SDL mutants, may be a good alternative to overcome this bias.

Second, our research ourcome showed coverage measures were not strongly correlated with defect-detection capability. For both of the assignments, students were graded on their branch coverage. They also got feedback on coverage of their code. It is natural that they put effort into achieving high coverage. In fact, more than one third of the students achieved very high (> 90%) coverage scores. For example, 106 out of 147 students achieved 100% in Project 1. Thus, little (or no) variance between their coverage scores was observed. This affected the correlation between coverage metrics and defect-detection capability. Further investigation is required to find out how student-written tests perform if they are given no feedback or feedback on a different measure such as SDL.

Third, we calculated failures based on the final submissions. It is plausible that students corrected easy defects between their first and final submissions. This raises the question of which defects were present when solutions were in their original, untested, undebugged state? Of course, few (if any) of the students ever had a complete solution in an untested state. Students in this course were taught beforehand to practice TDD and to incrementally develop their tests alongside their code. As a result, it is, impossible in many cases to construct a picture of an individual's code "before testing" to attempt to capture all the defects.

Finally, student-specific tests were not evaluated for all-pairs and mutation analysis. We tried to handle some student-specific designs in Project 4 by manually changing their base class names to the instructor-given interface name. However, about 30% of all the tests in that project were still either invalid or student-specific. Future

research on evaluating student-specific tests in all-pairs execution and mutation analysis (where mutants are generated from the reference solution) is important.

Chapter 5

Conclusion

As educators add software testing to more and more courses, the question of how to best evaluate student-written tests arises. While code (statement and branch) coverage tools are readily available and are already being used by some educators, coverage metrics have known limitations as test quality indicators. Other researchers and educators have proposed alternative measures aimed at addressing these limitations, but until recently, technical obstacles have prevented the use of these approaches. We investigated alternative measures to find the best test quality measure in terms of defect-detection capability of tests for automated assessment tools. We divide this chapter into four sections. Section 5.1 summarizes our research findings. Contributions of this research are described in Section 5.2. Section 5.3 focuses on more general implications of the research outcomes. Finally, future work and related research problems are presented in Section 5.4.

5.1 Research Results

In this thesis, we have analyzed three alternative test quality measures: all-pairs execution, mutation analysis and checked code coverage. We investigated four different sets of mutation operators that appeared most practical. Our research answered the following questions:

1. **Are these measures feasible to use for automated classroom grading systems?**

Yes all of the measures investigated proved feasible. Each of these measures have their own benefits and computational barriers to use them in automated grading systems. We provided novel solutions for the technical problems. We also examined their feasibility with CS2 assignments.

The main technical obstacle behind all-pairs execution is student-written tests may refer to their own personal design features that are not present in other students' code. In such cases, their JUnit tests do not compile against others' code. We modified the bytecode

of students' tests and applied late binding so that dependencies on student-specific design features are resolved at run-time rather than compile-time. As a result, tests referring to student-specific features will compile. Test cases that refer to student-specific features will throw run-time exceptions making those tests fail. However, tests checking common features will execute normally.

Mutation testing required a correct and complete version of the program to as the source for mutants. Students' programs are not good candidates for this purpose as they frequently submit incorrect and incomplete programs. We used an instructor-provided reference solution to resolve this issue. Mutation generation to evaluate each students' tests takes time. Instead, we pre-generated mutants from the reference solution ahead of time and stored them for reuse. Finally, we determined non-equivalent mutants incrementally when at least one valid student-written test fails. This procedure is conservative but guarantees that non-equivalent mutants are observed to produce a different outcome from the original program by valid tests.

Checked code coverage seemed to be promising but no automatic tool was available to calculate it. We developed a semi-automated system that works in five phases. First, it identifies explicit checks (i.e. asserts) inside the test suites. Second, it creates traces from all tests using JavaSlicer. Third, we filter out library code from traces. Fourth, we use JaCoCo to collect information about how many bytecode instructions are present in each class and each method under test. Finally, the system calculates checked coverage from the instructions present in the dynamic slices vs. total instructions in the programs. JavaSlicer traces each thread separately and cannot reconstruct data-dependencies between different threads. We modified JUnit 4 tests into JUnit 3 tests so that all tests run from the main thread. Though our system currently is only semi-automated, with some software engineering effort it can be made automatic.

2. Are they better indicators of defect-detection capability?

We designed a study to compare 10 different assessment approaches in terms of their prediction of defect-detection capability: all-pairs execution, mutation analysis with four different sets of mutation operators, statement coverage, branch coverage, OIC, OBC and checked coverage. To calculate defect-detection capability we created a master suite

representative of defects that students actually make in their code. Then we determined how many student-written tests overlap with the master suite tests and calculate the number of defects detected by those overlapped tests. If a test case of the master suite is overlapped by a student's tests multiple times, then failures detected by the test are counted once. This score shows the likelihood that a test-suite will detect any specific defect that may exist in a program.

Our experimental results show that all-pairs execution and mutation testing with any one of the four sets of mutation operators are better indicators of detect-detection capability than coverage measures. Although checked code coverage was expected to overcome the limitations of conventional coverage criteria, we did not find it to be a better predictor of defect-detection capability than other coverage measures.

3. Which of these test quality measure is the best?

Our research outcome shows all-pairs execution is the best predictor of defect-detection capability. All of the four mutation operator sets we investigated showed strong correlation with defect-detection capability as well, though the SDL operator performed second best. However, all-pairs execution is computationally more expensive than SDL mutation analysis. In all-pairs execution, the number of tests executed grows quadratically with the number of students. The number of test runs are linear in mutation analysis. As an example, compare test executions for both measures in Project 4. In all-pairs execution, 1,476 valid student-written tests were executed against 101 programs for Project 4, resulting in 149,076 test runs. For SDL mutation analysis, valid tests were run against 80 mutants, resulting in 118,080 test executions (30,999 fewer test runs than all-pairs execution). Moreover, SDL generates fewer mutants than sufficient or other deletion operators. Another problem of all-pairs execution is feedback cannot be generated until after the deadline when all the students have submitted their final code and final tests. This may be impractical for classroom grading tools. Thus, though all-pairs execution is the best predictor of defect-detection capability, considering computational cost and feedback delay, SDL mutation analysis is the best alternative for practical use in automated grading.

5.2 Contributions

This research investigates alternative approaches of test quality measures. We designed a study to compare 10 different measures in terms of their ability to predict defect-detection capabilities. This study determined that all-pairs execution is the best predictor of defect-detection capability of tests. We provided a novel way to remove compile time dependencies between JUnit tests and corresponding programs using late binding through reflection. This enables evaluation of complete or incomplete tests against correct and erroneous solutions or programs. The same technique can be used to evaluate incomplete programs which was not possible before.

We analyzed obstacles of all-pairs execution and provided novel solutions to overcome them. We also introduced a set of techniques including incrementally determining non-equivalent mutants and pre-generating mutants from a reference solution to apply mutation analysis in classroom grading tools. To calculate checked coverage, we developed a semi-automated system that calculates dynamic backward slices from checks and determines the percentage of code that influences checked results.

5.3 Broader Impacts

We found that students achieved high coverage scores but scores lower in all-pairs execution. This implies that they executed a large percentage of their code from their tests but did not detect many common failures or defects in other students' programs. We also found that students write diverse test cases. When we combine valid tests from all the students, the collection produced a strong test suite. For example, this collection detected all the mutants (over thousand in total) generated by the four different mutation operators. The same collection detected about 92% of all the errors that students produced naturally in coding. However, individually their test cases were not strong.

This research outcome provides educators and students an insight into the quality of students' testing skills. Implementation of our solution to remove compile-time dependencies from the test cases will enable automated graders to evaluate partial solutions. Application of all-pairs execution, mutation analysis, and checked code coverage will encourage students to practice testing skills in many classes and will give them

concrete feedback on their testing performance. Educators will be able to automatically evaluate students' tests using the test quality measure that predicts defect-detection capability most accurately. As a result, students will have more opportunities to find what defects they make, by getting a better evaluation on their test cases, and may improve the accuracy of their solution.

5.4 Future Work

In our experiments, we used final submissions of students work. It is very likely that students' fixed defects between their first and final submissions. Thus, only the defects that students failed to detect remained in their code. This is a plausible reason behind their low scores in defect-detection capability. We plan to extend our experiments with all the defects—from all of their submissions so that the defect-detection capability score would reflect all types of defects. Exploration of types and frequencies of the defects that students make also would be helpful for instructors to give them better feedback. We used assignments as independent variables in our statistical analysis. Types of assignments, such as GUI assignments, may affect the correlation of different measures with defect-detection scores. We want to include more assignments from different semesters to generalize our research results.

This research showed all-pairs execution is the best predictor, and SDL mutation was the second best predictor for detect-detection capability. We used the same defective programs from students to calculate all-pairs scores and to create master suites (that represent equivalence class of defects). We plan to create a master suite covering SDL mutants and investigate if that changes the correlation between different measures and defect-detection scores. Moreover, we want to categorize Javalanche-generated mutants in a similar way that we used for µJava generated mutants. This will help us to determine which mutation operators of Javalanche are creating more effective mutants. In addition, we will investigate a new tool, PITEST [5], for mutation analysis.

We plan to investigate test cases that constitute the master suite. Not all the tests in the master suite detected the same number of defects. It is possible that some test cases detect more defects than others. We consider test case A is a superset of test case B if 1)

test case A detects more failures than test case B, and 2) test case A detects all the failures that test case B detects. A master suite consisting of the super set tests may be as effective as the complete set but may reduce cost of calculating defect-detection capability.

A major concern of all-pairs execution and mutation analysis is giving students feedback on their tests. We are researching effective ways of generating feedback from the approaches without revealing reference solutions or other students' solutions. In all-pairs execution, tests are evaluated on how many defects they can find in others' code. It is not possible to give direct feedback on what defects a test case failed to detect. Similarly in mutation analysis, mutants are created from the reference solution. Students will not have access to the reference solution. Thus, some indirect but effective form of feedback is needed so that students can understand problems of their tests.

Lastly, implementing practical alternative approaches, such as mutation analysis, in Web-CAT and other classroom assessment tools, and investigating students' performance change with this addition will enable us to bring into reality the benefits of this research.

References

[1] "Clover: Java and Groovy Code Coverage," Available: https://www.atlassian.com/software/clover/overview, accessed 10/19/2013

[2] "EMMA: a free Java code coverage tool," Available: http://emma.sourceforge.net/, accessed 10/19/2013

[3] "JaCoCo Java Code Coverage Library," Available: http://www.eclemma.org/jacoco/, accessed 10/19/2013 2013

[4] "JavaSlicer," Available: http://www.st.cs.uni-saarland.de/javaslicer/, accessed 11/10/2013 2013

[5] "PIT: Real world mutation testing," Available: http://pitest.org/, accessed 3/9/2015

[6] K. Aaltonen, P. Ihantola, and O. Seppälä, "Mutation analysis vs. code coverage in automated assessment of students' testing skills," in *Proceedings of the ACM International Conference Companion on Object-oriented Programming Systems Languages and Applications Companion*, Reno/Tahoe, Nevada, USA, pp. 153-160, 2010.

[7] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1 ed.: Cambridge University Press, 2008.

[8] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, USA, pp. 402-411, 2005.

[9] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for C†," *Software Testing, Verification and Reliability*, vol. 11, pp. 113-136, 2001.

[10] C. E. Bonferroni, *Teoria statistica delle classi e calcolo delle probabilità*: Libreria internazionale Seeber, 1936.

[11] J. G. C. Comar, O. Hainque, T. Quinot, "Formalization and Comparison of MCDC and Object Branch Coverage Criteria," in *In ERTS (Embedded Real Time Software and Systems Conference)*, 2012.

[12] P. Chevalley, "Applying mutation analysis for object-oriented programs using a reflective approach," in *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pp. 267-270, 2001.

[13] P. Chevalley and P. Thévenod-Fosse, "A mutation analysis tool for Java programs," *International journal on software tools for technology transfer*, vol. 5, pp. 90-103, 2003.

[14] S. Chiba, "Javassist," Available: http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/, accessed 3/7/2015 2015

[15] S. Chiba and M. Nishizawa, "An easy-to-use toolkit for efficient Java bytecode translators," in *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, Erfurt, Germany, pp. 364-376, 2003.

[16] J. J. Chilenski, "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion " U.S. Department of Transportation 2001.

[17] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens, "Improving your software using static analysis to find bugs," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA, pp. 673-674, 2006.

[18] S. Cornett, "Code Coverage Analysis," Bullseye Testing Technology, 1996-2014.

[19] M. E. Delamaro, J. Offutt, and P. Ammann, "Designing Deletion Mutation Operators," in *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, pp. 11-20, 2014.

[20] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, pp. 34-41, 1978.

[21] L. Deng, J. Offutt, and N. Li, "Empirical Evaluation of the Statement Deletion Mutation Operator," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 84-93, 2013.

[22] C. Desai, D. S. Janzen, and J. Clements, "Implications of integrating test-driven development into CS1/CS2 curricula," in *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, pp. 148-152, 2009.

[23] T. Dvornik, D. S. Janzen, J. Clements, and O. Dekhtyar, "Supporting introductory test-driven labs with WebIDE," in *Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training*, pp. 51-60, 2011.

[24] S. H. Edwards, "Improving student performance by evaluating how well students test their own programs," *J. Educ. Resour. Comput.*, vol. 3, p. 1, 2003.

[25] S. H. Edwards, "Rethinking computer science education from a test-first perspective," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, USA, pp. 148-155, 2003.

[26] S. H. Edwards, "Using test-driven development in the classroom: Providing students with concrete feedback " in *International Conference on Education and Information Systems: Technologies and Applications*, pp. 421–426, 2003.

[27] S. H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," in *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, ACM*, pp. 26-30, 2004.

[28] S. H. Edwards and Z. Shams, "Comparing test quality measures for assessing student-written tests," in *Companion Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, pp. 354-363, 2014.

[29] S. H. Edwards and Z. Shams, "Do student programmers all tend to write the same software tests?," in *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, Uppsala, Sweden, pp. 171-176, 2014.

[30] S. H. Edwards, Z. Shams, M. Cogswell, and R. C. Senkbeil, "Running students' software tests against each others' code: new life for an old "gimmick"," in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, Raleigh, North Carolina, USA, pp. 221-226, 2012.

[31] S. Elbaum, S. Person, J. Dokulil, and M. Jorde, "Bug Hunt: Making Early Software Testing Lessons Engaging and Affordable," in *Proceedings of the 29th International Conference on Software Engineering*, pp. 688-697, 2007.

[32] M. H. Goldwasser, "A gimmick to integrate software testing throughout the curriculum," in *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, pp. 271-275, 2002.

[33] R. A. D. H. Agrawal, B. Hathaway, W. Hsu, W. Hsu, and R. J. M. E. W. Krauser, A. P. Mathur, and E. Spafford., "Design of mutant operators for the C programming language.," SERC-TR-41-P, 2006.

[34] M. Hauswirth, D. Zaparanuks, A. Malekpour, and M. Keikha, "The JavaFest: a collaborative learning technique for Java programming courses," in *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, Modena, Italy, pp. 3-12, 2008.

[35] M. T. Helmick, "Interface-based programming assignments and automatic grading of java programs," in *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, Dundee, Scotland, pp. 63-67, 2007.

[36] D. Jackson and M. Usher, "Grading student programs using ASSYST," in *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, pp. 335-339, 1997.

[37] D. S. Janzen and H. Saiedian, "Test-driven learning: intrinsic integration of testing into the cs/se curriculum," in *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pp. 254–258,, 2006.

[38] Z. Junji and V. Garousi, "On Adequacy of Assertions in Automated Test Suites: An Empirical Investigation," in *IEEE Sixth International Conference on Software Testing Verification and Validation Workshop*, pp. 382-391, 2013.

[39] S. Kim, J. A. Clark, and J. A. McDermid, "Class mutation: Mutation testing for object-oriented programs," in *Proc. Net. ObjectDays*, pp. 9-12, 2000.

[40] C. C. M. Bordin, T. Gingold, J. Guitton, O. Hainque, and T. Quinot, "Object and Source Coverage for Critical Applications with the COUVERTURE Open Analysis Framework," in *ERTS (Embedded Real Time Sofware and Systems Conference)*, 2010.

[41] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system: Research Articles," *Softw. Test. Verif. Reliab.*, vol. 15, pp. 97-133, 2005.

[42] B. Marick, "How to misuse code coverage," in *Proceedings of the 16th Interational Conference on Testing Computer Software*, pp. 16-18, 1999.

[43] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," *Commun. ACM*, vol. 6, pp. 58-63, 1963.

[44] A. S. Namin, J. H. Andrews, and D. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *Proceedings of the 30th international conference on Software engineering*, pp. 351-360, 2008.

[45] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, pp. 5-20, 1992.

[46] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, pp. 99-118, 1996.

[47] A. J. Offutt and R. Untch, "Mutation 2000: Uniting the Orthogonal," in *Mutation Testing for the New Century*. vol. 24, W. E. Wong, Ed., ed: Springer US, pp. 34-44, 2001.

[48] D. Schuler, "Javalanche," Available: https://github.com/david-schuler/javalanche/, accessed 04/15/2013 2013

[49] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for Java," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, Amsterdam, The Netherlands, pp. 297-298, 2009.

[50] D. Schuler and A. Zeller, "Assessing Oracle Quality with Checked Coverage," in *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation*, pp. 90-99, 2011.

[51] Z. Shams and S. H. Edwards, "RefectionSupport: Java Reflection Made Easy," *The Open Software Engineering Journal, TOSEJ*, vol. 7, pp. 38-52, 2013.

[52] Z. Shams and S. H. Edwards, "Toward practical mutation analysis for evaluating the quality of student-written software tests," in *Proceedings of the Ninth Annual*

International ACM Conference on International Computing Education Research, San Diego, San California, USA, pp. 53-58, 2013.

[53] Z. Shams and S. H. Edwards, "Checked Coverage and Object Branch Coverage: New Alternatives for Assessing Student-Written Tests," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, Kansas City, Missouri, USA, pp. 534-539, 2015.

[54] J. Spacco and W. Pugh, "Helping students appreciate test-driven development (TDD)," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA, pp. 907-913, 2006.

[55] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, RTI Project Number 7007.011, 2002.

[56] M. D. Weiser, "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method," University of Michigan, 1979.

[57] W. E. Wong, "On Mutation and Data Flow," Purdue University, 1993.

[58] a. J. O. Yu-Seung Ma, "Description of Method-level Mutation Operators for Java," Available: http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf, accessed 12/11/2014 2014

[59] J. Yue and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, pp. 649-678, 2011.

[60] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, pp. 366-427, 1997.