The Influence of Test Suite Properties on Automated Grading of Programming Exercises

Benjamin S. Clegg
University of Sheffield

Phil McMinn
University of Sheffield

Gordon Fraser
University of Passau

***Abstract*—Automated grading allows for the scalable assessment of large programming courses, often using test cases to determine the correctness of students' programs. However, test suites can vary in multiple ways, such as quality, size, and coverage. In this paper, we investigate how much test suites with varying properties can impact generated grades, and how these properties cause this impact. We conduct a study on artificial faulty programs that simulate students' programming mistakes and test suites generated from manually written tests. We find that these test suites generate greatly varying grades, with the standard deviation of grades for each fault typically representing ∼84% of the grades not apportioned to the fault. We show that different properties of test suites can influence the grades that they produce, with coverage typically making the greatest effect, and mutation score and the potentially redundant repeated coverage of lines also having a significant impact. We offer suggestions based on our findings to assist tutors with building grading test suites that assess students' code in a fair and consistent manner. These suggestions include ensuring that test suites have 100% coverage, avoiding unnecessarily recovering lines, and checking test suites using real or artificial faults.**

I. Introduction

Automated grading sees widespread use in software engineering education, both in traditional courses and massive open online courses (MOOCs), as it allows tutors to assess programs written by growing numbers of students without demanding more time, as would be the case for manual assessment [1]–[3]. This is especially important in large courses, where assessment would be otherwise impossible. Automated grading also benefits students, by providing them with near instantaneous feedback for formative assessment tasks, even when outside of the conventional teaching context. A common approach to grade programs automatically is to run automated test suites to check the correctness of students' code, and then to calculate a grade based on the fraction of tests that passed [3]. However, such grades may be inaccurate with respect to a task's learning objectives, as test suites constructed by tutors may be deficient in detecting students' errors [4].

Consider the example program in Figure 1a, which is supposed to calculate the absolute value of its input, but erroneously returns a wrong value for negative inputs. If the grade is calculated as the percentage of the tests passing in the JUnit test suite shown in Figure 1b, the resulting grade will be 100% because the erroneous code is not covered at all. In contrast, the test suite in Figure 1c will result in a grade of 0% since only the erroneous code is covered. A more complete assessment would result from Figure 1d, where one test covers the correct branch of the if-condition, and one test covers the erroneous case, resulting in a grade of 50%, suggesting that coverage is a prerequisite for a fair grading test suite.

```
int abs(int x) {
 if(x > 0) {
  return x;
 } else {
  return x; // Incorrect: should be -x
 }
}
```

(a) Example program containing a mistake.

```
@Test void test1() {
 assertEquals(42, abs(42));
}
```

(b) Resulting grade: 100%.

```
@Test void test1() {
 assertEquals(42, abs(-42));
}
```

(c) Resulting grade: 0%.

```
@Test void test1() {
 assertEquals(42, abs(42));
}
@Test void test2() {
 assertEquals(42, abs(-42));
}
```

(d) Resulting grade: 50%.

```
@Test void test1() {
 assertEquals(42, abs(42));
}
@Test void test2() {
 assertEquals(42, abs(-42));
}
@Test void test3() {
 assertEquals(0, abs(0));
}
```

(e) Resulting grade: 66.7%.

```
@Test void test1() {
 assertEquals(42, abs(42));
}
@Test void test2() {
 assertEquals(42, abs(-42));
}
@Test void test3() {
 assertEquals(10, abs(-10));
}
```

(f) Resulting grade: 33.3%.

```
@Test void test1() {
 abs(42);
}
@Test void test2() {
 abs(-42);
}
@Test void test3() {
 abs(0);
}
```

(g) Resulting grade: 100%.

Fig. 1: Example of the grading behaviour of different JUnit test suites on a single faulty method.

However, even when all code is covered, grades may vary based on *additional* tests: Consider Figure 1e and Figure 1f; both achieve 100% code coverage, but nevertheless the resulting grades vary substantially, with one test suite resulting in a grade of 66.7% while the other results in only 33.3%, due to their tests covering some statements more than others. Finally, consider Figure 1g: Although the test suite contains three tests and achieves 100% code coverage, the tests are of low quality and omit the important assertions, thus resulting in an overall grade of 100% as they cannot detect any erroneous implementations. Such quality issues in grading test suites may result in cases where a student's solution is less influential on their grade than the nature of the test suite itself.

In order to understand how test suites influence the resulting

grades, and thus how they can impact fairness and consistency of grades and the quality of feedback, we empirically study the effects of grading test suites on grades. First, we aim to understand whether the effect of grade variation illustrated by the example in Figure 1 can occur in practice:

***RQ1**: How much do grades vary with different test suites?*

Our experiment on a set of Java classes using large numbers of sampled test suites suggest that grades for individual errors can vary significantly, with a standard deviation proportional to ∼84% of the mean grades lost for each fault. Considering this result, we aim to better understand what properties of test suites are influential for the variation in grades:

***RQ2**: Which properties of test suites impact grades?*

Besides the obvious factors of coverage and general test quality (measured using the mutation score), we note that the redundancy of coverage is a further influential factor in grades calculated with automated tests.

We use these findings to suggest strategies that tutors can use to ensure consistency in their grading test suites.

II. Research Methodology

A. Experiment Procedure

To answer the research questions, we require (1) programming assignments; (2) multiple erroneous implementations of these assignments; and (3) test suites with different properties to study their influence on the resulting grade. To analyse these properties in a statistically meaningful way, we used a simulated scenario, where we use real Java classes, but artificial faulty versions (*mutants*, Section II-C), and sample test suites from a larger pool of JUnit tests (Section II-D), in order to gather a large number of datapoints.

Given the test suites and mutants, we assume a simple grading scenario: The grade for a solution is determined by executing all tests, and calculating the percentage of executed tests that passed [3]. We execute each test on every mutant for each subject, as well as the subject class itself, and store the results. We use JaCoCo [5] to gather coverage information for every test execution, allowing us to store which lines are covered and uncovered by each test for every mutant.

a) RQ1: In order to measure the influence of different test suites on grades, we investigate the range of grades. Given a large sample of test suites, it is inevitable that there will be the odd test suite consisting of only failing tests (resulting in a grade of 0%) or of only passing tests (resulting in a grade of 100%) for each mutant. Consequently, we would expect the overall range to be 100% for most mutants. Therefore, instead of the range we look at the standard deviation per mutant.

b) RQ2: In order to measure the influence of different test suite properties on the resulting grade, we perform linear regression on every run of each subject class. Prior to running the linear regression, we normalise the grade and observed properties to allow for the comparison of β coefficients.

By collecting the results of tests that are present in each test suite for each mutant, we are able to construct test suite results for every mutant, including the number of passing / failing tests, plus line coverage data. These observations allow us to derive several properties for every pair of mutants and test suites. We compute the generated grade as:

$$\textit{Grade} = \frac{n(P)}{n(T)},$$
$$\text{where } P = \text{passing tests in the suite, and}$$
$$T = \text{all tests in the suite.}$$

We use test adequacy metrics of line coverage and mutation score to capture the quality of test suites:

$$\textit{Coverage} = \frac{n(C)}{n(L)},$$
$$\text{where } C = \text{lines covered by the suite, and}$$
$$L = \text{total lines.}$$
$$\textit{Mutation Score} = \frac{n(D)}{n(M)},$$
$$\text{where } D = \text{other mutants detected by the suite, and}$$
$$M = \text{other mutants.}$$

Additionally, in order to examine the effect of multiple tests covering the same lines, we define a recoverage metric that captures the average number of times each line is recovered:

$$\textit{Recoverage} = \frac{\sum_{l \in C}(f(l) - 1)}{n(T) \cdot n(L)},$$
$$\text{where } f(l) = \text{Number of tests in the suite that cover } l$$

It is possible for other test suite metrics to be considered in future work, such as the number of assertions and lines of code in each test suite.

B. Subject Programs

As a source of example programs and tests, we used nine Java classes contained in a dataset originating from experiments with Code Defenders [7], an educational testing game (Table I). Code Defenders is a mutation testing game where users compete to write faulty mutant variants of a particular program and unit tests to detect these mutants, and has been shown to support the crowdsourcing of mutation testing by Rojas et al. [7], as well as the teaching of software testing techniques by Fraser et al [6]. The subject classes are taken from real open-source Java projects, with the exception of `Lift`, which was written as a simple introduction to Code Defenders [6]. Importantly

TABLE I: Subject classes from the Code Defenders dataset [6].

Subject	LoC	Mutants	Tests	Source Project
ByteVector	154	462	746	Objectweb ASM
CharRange	87	247	896	Commons Lang
Complex	103	294	373	Math4J
IntHashMap	148	343	861	Commons Lang
Lift	53	73	412	(Custom)
Option	260	829	806	Commons CLI
Rational	114	249	675	Dittrich Java Intro
SparseIntArray	161	616	651	Android
XmlElement	221	288	671	Inspirento

TABLE II: Functional mutation operators that we use in this experiment.

Operator	Description
Logic Flow Error	Move contents of a conditional statement's block to outside the block [8].
Incomplete Implementation	Remove all statements in the same scope after a particular point.
String Misspelling	Change or transpose characters in a string [8].
Statement Deletion	Delete a statement [8], [9].
Incorrect Values	Replace a numeric or boolean value with another value of its type (e.g. 1 to -1), or replace a string with an empty string [8], [9].
Incorrect Calculation	Replace an arithmetic operator (+, −, *, etc.) [8], [9].
Logical Op. Replacement	Replace a logical operator (|, &, etc.) [9].
Conditional Op. Replacement	Replace a conditional operator (||, &&, etc.) [9].
Relational Op. Replacement	Replace a relational operator (==, <, etc.) [9].
Shift Operator Replacement	Replace a shift operator (<<, >>, etc.) [9].

for our setting, each of the Java classes comes with a large number of student-written test cases, which we use as a source for grading test suites.

The original dataset includes three additional subject classes, which we excluded from the experiment for one of two reasons. We did not use `CaseInsensitiveString` as some of its AST nodes are incorrectly represented by the JavaParser library, preventing our mutation tool from generating mutants that require the manipulation of a subject's AST.[1] We also excluded `Document` and `Options` (distinct to `Option`), due to incompatibilities with our execution tool.

Each subject class in the dataset includes a set of human-written tests from a series of human studies and classroom trials used in the evaluation of Code Defenders. Some of these tests fail when executed on the original subject class. These tests are invalid, so we removed them from the dataset prior to running the experiment.

C. Grading Candidate Implementations

For this experiment, we use mutation analysis techniques to generate a set of faulty variants for each subject class. In mutation analysis, artificial defects are generated by systematically applying operators that represent different classes of faults. Each resulting mutant differs from the original program by exactly one change. Table II shows the mutation operators that we implemented and used for this experiment. Figure 3 shows some of the faults generated by these operators. Since we aim to investigate how test suites affect automated grading, we implemented some of the mutation operators that we proposed in our previous work as a means to simulate novice programming mistakes [8]. We only used operators which affected the functionality of a class without preventing compilation, since uncompilable mutants would fail on all tests, only generating grades of 0%, which would skew our results. As such, we disabled the *Incorrect Classname* and *Missing Syntax Elements* operators. When implementing our mutation tool, we found that some of our mutation operators subsumed others, for example, *Misspellings in Strings* often results in *Incorrect Filenames*, and *Incorrect Values* can result in *Exceeds Range*. We only implemented the subsuming operators. We redefined our *Incomplete Implementation* operator as removing both a statement and all statements that follow it, to better simulate partially completed solutions. We have retained the original functionality of *Incomplete Implementation*, denoted by *Statement Deletion*, as implemented by Major [9], [10].

Some of our other mutation operators are equivalent to those already implemented by other tools, for example, several operator replacement mutation operators are implemented in Major. In these cases our mutation tool executes Major with the appropriate parameters. For the remaining mutation operators, we used one of two approaches in our tool. For operators that involve simple replacements, such as *String Misspelling*, we locate a line where a change can be made, then apply a string modification to the line. Other operators are more complex, such as *Logic Flow Error* and *Incomplete Implementation*, since they require knowledge of the context in which they can be applied. We implemented these operators by manipulating Java's Abstract Syntax Tree (AST) using JavaParser [11]. For example, to apply the *Incomplete Implementation* operator to a given node (such as a statement), our mutation tool removes the node's succeeding sibling nodes. With the exception of these AST based mutation operators, each operator only modifies a single line of the original program.

Some of the mutation operators can result in non-compilable mutants. For example, our implementation of the *Incomplete Implementation* operator can remove a method's return statement, where the method's declaration requires a value to be returned. Therefore, we ran the Java compiler on every generated mutant, then deleted mutants which failed to compile.

Prior to running our data analysis on each subject, we identify mutants that are potentially equivalent to the original subject class by checking if no tests in the entire set fail on them. We remove all observed executions for these mutants. While these mutants may not be truly equivalent, they will always have a generated grade of 100%, and as such different test suites would not generate varying grades for them. For the same reason, we also remove mutants which cause every test to fail, and therefore only receive grades of 0%.

D. Grading Test Suites

For 30 repetitions of each subject, we constructed a set of test suites by randomly sampling tests from the entire group of human written tests available for it. If a suite is generated that already exists for a run of the experiment, it is discarded and another is generated in its place.

[1] We used JavaParser 3.7.0 in our mutation tool, https://mvnrepository.com/artifact/com.github.javaparser/javaparser-core/3.7.0

To ensure variation in coverage, recoverage, and mutation score, we generated test suites with 10, 20, 40, 60, 80, and 100 tests. This is necessary to control for the possible relationship between suite size and both mutation score and coverage [12]. For the same reason we do not include size as a factor of the linear regression; suite size would likely have a strong degree of covariance with the other factors, which would introduce a significant challenge in identifying the relative impact of individual test suite properties. We generated 1000 suites per repetition of the experiment for each subject, split across each of the six sizes, rounded up (167 suites per size).

E. Threats to Validity

The mutants that we generated only introduce one change each. As such, they will not entirely reflect students' solutions which may contain multiple mistakes. It is possible that for real students' solutions and real grading test suites, the changes to grades may be even greater than what we observe. As such, repeating this experiment with higher-order mutants or real student solutions may provide an even greater insight into how different test suite properties affect grades.

The tests that we use from the Code Defenders dataset may not reflect those written by a tutor. However, they are still tests that are capable of detecting faults in software, and can be used to construct different test suites with varying properties.

The number of mutants generated by each operator varies. As such, our analysis may be skewed by the instances of more common mutation operator classes, which will generally represent smaller faults. However, where a fault appears in a program is what truly impacts its severity. For example, if a small arithmetic change causes a program to attempt to read outside of an array's range in a class' constructor, more tests may fail than if the contents of several methods were removed.

While there is evidence that artificial faults are a suitable substitute for real faults in determining test suite effectiveness [13], it is possible that this may not hold for students' faults, presenting a construct threat. In future work, we will conduct a study using real student faults to verify that mutants are an effective substitute for them.

It is possible that the mutation operators could generate multiple mutants that are functionally identical to one another. This may skew the data towards these duplicated mutants. There are no automated approaches that reliably identify such duplicates, and it is infeasible to manually check for duplication across so many mutants.

III. Results

A. RQ1: To what extent do different test suites generate varying grades?

Figure 2a shows the distribution of grades generated by each suite for every mutant across all 30 repetitions. Figure 2b shows the standard deviation of grades generated by all test suites across the 30 repetitions on a per mutant basis. We find that there is a spread of grades generated for each mutant, despite mutants typically only being a few percentage points short of perfect grade due to their often minute changes to the correct program.

TABLE III: Summary of variance inflation factors (VIFs) for each property across all linear models.

	Coverage	**Recoverage**	**Mutation Score**
Median	1.60	1.43	1.09
Std. Dev.	0.53	0.48	0.40
Max.	4.09	3.07	3.81

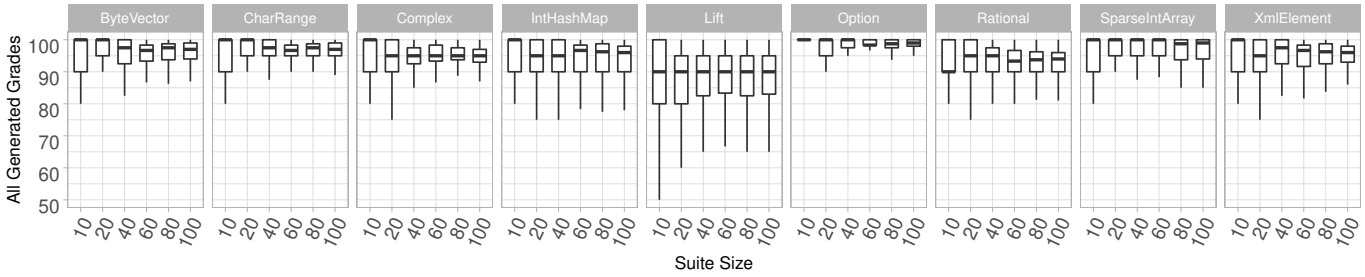
The distributions of generated grades and mutants' standard deviations vary between our observed subject classes. This suggests that the nature of a class itself affects how potential test suites perform on faulty variants. We also note that there is a range of standard deviations across the set of mutants, indicating that some mutants are more prone to a change in grade due to the selected test suite than others. This may be due to how detectable a mutant is with respect to the whole set of available tests. If a mutant is detected by fewer tests it is less likely that a detecting test will be used in a test suite, so the mutant will be more likely to receive a high grade. Additionally, a rare, detecting test would more often be outnumbered by non-detecting tests, so even if it is used, it would have a limited impact on the grade. This especially appears to be the case for `Option`, which when compared to the other subject classes has a very high median grade and low median standard deviation of grades for each mutant.

Across all runs and all subjects, the mean grade is $\sim$96.5%, and the mean standard deviation per mutant is $\sim$2.94%. As each mutant "loses" $\sim$3.5% of the maximum grade, this standard deviation accounts for a very significant amount of the possible range of grades; the standard deviation represents $\sim$84% of the lost grades. This indicates a very strong impact on the inconsistency of grading for single mistakes across different possible test suites. Different test suites are likely to generate varying grades for the same fault. This effect is more prevalent for smaller test suites; for `Lift`, suites with 10 tests typically have a standard deviation of $\sim$9.5% grades for single mutants, while suites with 100 tests have a standard deviation of $\sim$2.6%.

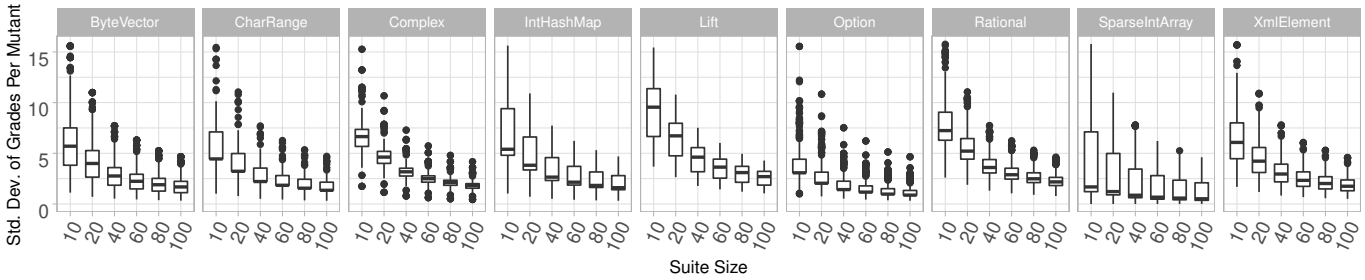
RQ1 Results: Grades can vary greatly across different test suites, with a standard deviation that represents $\sim$84% of the grades lost on average for each individual mutant.

B. RQ2: Which properties of test suites impact grades?

We performed multivariate linear regression on each of the 30 runs for every subject and test suite size, in order to determine if the coverage, recoverage, and mutation score of test suites have an impact on generated grades. Table IV shows the mean results of the linear regressions across all 30 runs. We find that all of the overall linear models are statistically significant, with p-values of $<$0.001. However, models generated for some subjects have a very low adjusted R^2 (R^2_{adj}), most notably `CharRange`, `Complex`, and `Option`, suggesting that the model fails to explain most of the variability in generated grades across our observations. In these cases, differences in grades are more likely to be due to the mutants themselves



(a) All generated grades across all 30 repetitions. For ease of presentation, we removed the outliers.



(b) Standard deviation of grades generated by all suites for each mutant across all 30 repetitions.

Fig. 2: Generated grade statistics of mutants for each subject class.

and how they interact with the tests for the subject class. However, for some of the other subjects our models do explain a significant variability in grades, especially for `IntHashMap` and `SparseIntArray`, indicating that the observed test suite properties do have a considerable impact on generated grades for these subjects. As such, while the magnitude of the effect varies between subject classes, observable test suite properties can have an impact on grades generated by test suites.

Table IV also shows the standardised coefficients (β) and significance values that each property has in our linear models. Only 5 observed coefficients for the properties have a mean significance of $>$0.05. We observe that the remaining property coefficients typically achieve a high significance, with $p<0.01$ in most cases. This indicates that each of the properties has a significant contribution to the linear models, and therefore has some influence on the generated grades for our mutants.

Across all of our models, the properties have low variance inflation factors (VIF), as shown in Table III. This indicates that the properties each have a low degree of multicollinearity, and that the impact of correlations between the properties on the variance of their β coefficients is limited [14]. This allows us to use the β coefficients to determine which factors contribute the most to a model with a higher degree of reliability, thus providing an estimate of the relative impact of each property on generated grades.

On average across the models for each subject, we find that the β coefficient of *Coverage* has the greatest magnitude (0.39), followed by *Mutation Score* (-0.22) and *Recoverage* (0.2). As such, *Coverage* generally influences grades the most, but the other properties also have a significant impact.

The β coefficients for some properties change with different test suite sizes. Across all subjects, as the test suite size increases, so does the magnitude of the β coefficient for *Recoverage*. This could be due to larger test suites being more likely to recover both more and a greater variety of lines of code, which would increase the likelihood of some mutants being detected. Conversely, the magnitude of *Mutation Score* decreases as the test suite size increases. This may be a result of larger test suites being more likely to detect mutants than smaller test suites. This is illustrated by Figures 2a and 2b, where the median and upper quartile grades tend to be lower for larger test suites, whilst the standard deviation of grades for each mutant is also smaller. In these cases, grades are more consistently lower with larger test suites since they have a greater fault detection capability.

There is generally a fluctuation of magnitude for *Coverage* as test suite size varies, perhaps due to different suite sizes achieving different coverage levels. Small test suites may have lower coverage, and very large suites may rarely have low coverage. We also observe that *Mutation Score* usually has a negative β coefficient, and where it is positive, it is very close to zero. This suggests that test suites with a greater fault detection capability will generally generate lower grades.

The overall mean β coefficients are not reflected for some subjects. For example, *Mutation Score* tends to have a very low impact on `IntHashMap` for larger test suites, and *Coverage* has a low impact on `Option`. We examined individual mutants for these subjects, and found that grades for individual mutants did not change with respect to these respective properties. In the case of `IntHashMap`, the low impact of mutation score may be due to different mutants not exhibiting similar behaviour, perhaps due to low method interdependency, the relatively high number of conditional statements, and a low average of mutants per line ($\sim$2.3). For `Option` we found that the subject uses classwide variables extensively, so any mutants in the class are likely to modify its state in an easily observable manner. As such, for most mutants, larger test suites are likely to contain

TABLE IV: Summary of linear regression models built for each subject and test suite size, mean values across 30 repetitions, rounded to 2 decimal places.

Size	Model		Coverage		Recoverage		Mutation Score	
	p	R^2_{adj}	β	p	β	p	β	p
`ByteVector`								
10	<0.01	0.27	0.56	<0.01	0.25	<0.01	−0.62	<0.01
20	<0.01	0.39	0.55	<0.01	0.33	<0.01	−0.56	<0.01
40	<0.01	0.5	0.47	<0.01	0.41	<0.01	−0.4	<0.01
60	<0.01	0.54	0.38	<0.01	0.49	<0.01	−0.32	<0.01
80	<0.01	0.57	0.3	<0.01	0.56	<0.01	−0.23	<0.01
100	<0.01	0.6	0.23	<0.01	0.63	<0.01	−0.21	<0.01
`CharRange`								
10	<0.01	0.04	0.05	<0.01	−0.09	<0.01	−0.18	<0.01
20	<0.01	0.03	0.09	<0.01	−0.13	<0.01	−0.15	<0.01
40	<0.01	0.05	0.16	<0.01	−0.17	<0.01	−0.13	<0.01
60	<0.01	0.07	0.21	<0.01	−0.18	<0.01	−0.12	<0.01
80	<0.01	0.09	0.25	<0.01	−0.2	<0.01	−0.1	<0.01
100	<0.01	0.1	0.27	<0.01	−0.22	<0.01	−0.07	<0.01
`Complex`								
10	<0.01	0.06	0.05	<0.01	0.1	<0.01	−0.3	<0.01
20	<0.01	0.04	0.05	<0.01	0.14	<0.01	−0.24	<0.01
40	<0.01	0.05	0.07	<0.01	0.2	<0.01	−0.21	<0.01
60	<0.01	0.07	0.12	<0.01	0.23	<0.01	−0.19	<0.01
80	<0.01	0.1	0.17	<0.01	0.26	<0.01	−0.19	<0.01
100	<0.01	0.14	0.21	<0.01	0.28	<0.01	−0.17	<0.01
`IntHashMap`								
10	<0.01	0.44	0.67	<0.01	0.02	<0.05	−0.26	<0.01
20	<0.01	0.66	0.76	<0.01	0.08	<0.01	−0.1	<0.01
40	<0.01	0.76	0.7	<0.01	0.21	<0.01	−0.02	<0.05
60	<0.01	0.77	0.6	<0.01	0.33	<0.01	0	<0.05
80	<0.01	0.77	0.52	<0.01	0.41	<0.01	0	0.06
100	<0.01	0.78	0.45	<0.01	0.48	<0.01	0	<0.05
`Lift`								
10	<0.01	0.13	0.43	<0.01	0.08	<0.01	−0.41	<0.01
20	<0.01	0.24	0.5	<0.01	0.1	<0.01	−0.32	<0.01
40	<0.01	0.37	0.55	<0.01	0.13	<0.01	−0.17	<0.01
60	<0.01	0.42	0.55	<0.01	0.16	<0.01	−0.08	<0.01
80	<0.01	0.45	0.54	<0.01	0.2	<0.01	−0.05	<0.05
100	<0.01	0.47	0.52	<0.01	0.23	<0.01	−0.02	0.15
`Option`								
10	<0.01	0.1	0.01	0.17	0.11	<0.01	−0.3	<0.01
20	<0.01	0.08	−0.02	0.06	0.18	<0.01	−0.23	<0.01
40	<0.01	0.09	−0.03	<0.05	0.25	<0.01	−0.17	<0.01
60	<0.01	0.11	−0.03	<0.05	0.32	<0.01	−0.14	<0.01
80	<0.01	0.14	−0.02	<0.05	0.36	<0.01	−0.12	<0.01
100	<0.01	0.16	0	<0.05	0.39	<0.01	−0.1	<0.01
`Rational`								
10	<0.01	0.31	0.75	<0.01	0.06	<0.01	−0.67	<0.01
20	<0.01	0.47	0.78	<0.01	0.07	<0.01	−0.51	<0.01
40	<0.01	0.58	0.76	<0.01	0.07	<0.01	−0.27	<0.01
60	<0.01	0.62	0.74	<0.01	0.09	<0.01	−0.15	<0.01
80	<0.01	0.63	0.72	<0.01	0.12	<0.01	−0.09	<0.01
100	<0.01	0.64	0.69	<0.01	0.16	<0.01	−0.05	<0.01
`SparseIntArray`								
10	<0.01	0.41	0.61	<0.01	0.11	<0.01	−0.27	<0.01
20	<0.01	0.52	0.57	<0.01	0.21	<0.01	−0.15	<0.01
40	<0.01	0.66	0.58	<0.01	0.28	<0.01	−0.07	<0.01
60	<0.01	0.72	0.55	<0.01	0.35	<0.01	−0.04	<0.05
80	<0.01	0.74	0.46	<0.01	0.44	<0.01	−0.02	<0.05
100	<0.01	0.75	0.36	<0.01	0.54	<0.01	−0.01	0.07
`XmlElement`								
10	<0.01	0.13	0.5	<0.01	0.09	<0.01	−0.59	<0.01
20	<0.01	0.16	0.47	<0.01	0.16	<0.01	−0.52	<0.01
40	<0.01	0.23	0.43	<0.01	0.25	<0.01	−0.43	<0.01
60	<0.01	0.29	0.41	<0.01	0.31	<0.01	−0.37	<0.01
80	<0.01	0.33	0.37	<0.01	0.36	<0.01	−0.3	<0.01
100	<0.01	0.36	0.35	<0.01	0.39	<0.01	−0.22	<0.01
Mean (All Subjects & Suite Sizes)								
	<0.01	0.36	0.39	<0.05	0.2	<0.01	−0.22	<0.05

similar proportions of tests that detect the fault, irrespective of how much of the program they cover. From this, we can conclude that the impact of test suite properties on grades can depend on the subject itself, though most subjects do follow a similar trend.

RQ2 Results: *Coverage* generally has the most effect on generated grades ($\beta \approx 0.39$), though *Mutation Score* ($\beta \approx -0.22$) and *Recoverage* ($\beta \approx 0.20$) also have a significant impact.

IV. Discussion

As we observed in Section III, different test suites can produce varying grades, influenced by various properties of the test suites themselves. While the effect of test suite properties on students' grades cannot be directly controlled, some measures can be made by tutors when designing or updating a test suite, in order to improve consistency and fairness. We discuss such measures in this section. For each of our suggestions, we assume that a correct model solution which perfectly reflects a programming task's specification is available when developing a grading test suite.

In this section we demonstrate the effects of each property on grades, using `Lift` as an example, since its β coefficients are similar to the means across all subjects for suites with 10 tests. These means are 0.40, 0.08, and -0.4 for *Coverage*, *Recoverage*, and *Mutation Score*, respectively. We use three mutants to illustrate the effects of these properties. Figure 3 is the source code of `Lift`, including the diffs of the three mutants. Figure 4 shows the relationship between the properties and grades for each of the three mutants, alongside all of the mutants for `Lift`.

We selected the mutants by the relationships between their grades and the properties measured for the executions of each test suite:

- Mutant X (Figure 4b, Line 7-8) exhibits similar grade-property relationships to the overall trend of all `Lift` mutant executions combined.
- Mutant Y (Figure 4c, Line 21-22) has a positive relationship between grades and each property.
- Mutant Z (Figure 4d, Line 61-62) has a negative relationship between grades and each property.

A. Coverage

Overall Observations

Our results in Section III-B show that the coverage of a test suite often has a significant impact on generated grades. Figure 4 shows the grades generated by test suites of varying coverage levels.

The overall trend for `Lift` indicates an increase in grades along with coverage. It is possible that this is due to the mutants only affecting a subset of the lines of code, and as such, suites that have higher levels of coverage may have fewer tests that detect the mutant than suites with lower coverage that only cover lines affected by the mutant.

Individual Mutants

The overall effect is demonstrated by Mutant X, which modifies the initial value of `numRiders`. This mutant only affects three of the nine public non-constructor methods. If a test suite were to evenly cover all nine of these methods, two thirds of its tests would not be capable of detecting the mutant. If a test

```
public class Lift {

 private int topFloor;
 private int currentFloor = 0;
 private int capacity = 10;
 # Mutant X - Incorrect Values
 - private int numRiders = 0;
 + private int numRiders = 1;

 public Lift(int highestFloor) {
  topFloor = highestFloor;
 }

 public Lift(int highestFloor, int maxRiders) {
  this(highestFloor);
  capacity = maxRiders;
 }

 public int getTopFloor() {
  # Mutant Y - Incorrect Values
  - return topFloor;
  + return 0;
 }

 public int getCurrentFloor() {
  return currentFloor;
 }

 public int getCapacity() {
  return capacity;
 }

 public int getNumRiders() {
  return numRiders;
 }

 public boolean isFull() {
  return numRiders == capacity;
 }

 public void addRiders(int numEntering) {
  if (numRiders + numEntering <= capacity)
   numRiders = numRiders + numEntering;
  else
   numRiders = capacity;
 }

 public void goUp() {
  if (currentFloor < topFloor)
   currentFloor++;
 }

 public void goDown() {
  if (currentFloor > 0)
   currentFloor--;
 }

 public void call(int floor) {
  if (floor >= 0 && floor <= topFloor) {
   # Mutant Z - Relational Operator Replacement
   - while (floor != currentFloor)
   + while (true)
   {
    if (floor > currentFloor)
     goUp();
    else
     goDown();
   }
  }
 }
}
```

Fig. 3: The source code of `Lift`, with three example mutants.

suite exclusively covered the three affected methods, none of the tests would face such a restriction, and detecting the fault would be a matter of the tests' quality alone. This effect is even more apparent for Mutant Y, which only affects one public method. It is also possible that this behaviour is correct for the single faults introduced in most mutants; the faults are often relatively minor, so they should not lose many grades. In this sense, suites with higher coverage may provide more accurate grades for most faults.

Full coverage does not, however, guarantee that faults are always detected. In Mutant X and Mutant Y, some test suites provide a 100% grade despite achieving 100% coverage on these mutants. For these cases, revealing the fault may require different tests that evaluate results with higher precision, or exercise an edge condition.

These observations do not necessarily occur for all mutants, however. Mutant Z exhibits a trend of falling grades as coverage increases, and has no executions with 100% grades and 100% coverage. This is due to the mutant's introduction of a `while (true)` loop, which would never terminate, and cause any tests that execute it to timeout, revealing the fault. Merely executing such highly fragile faults guarantees that they are detected. Conversely, if any fault is not covered by tests, it cannot be detected. This is shown by the 100% grades for Mutant Z at lower coverage levels; these test suites do not cover the fault. If a fault in a student's solution is not covered, it would prevent them from receiving feedback on why they made a mistake.

Impact Mitigation

Fortunately, limiting the effect of coverage on grades is relatively straightforward, a tutor can write a test suite that achieves 100% coverage on the model solution. Such a test suite may not achieve 100% coverage on some student solutions, but in these cases either a student solution's uncovered code would not improve correctness, or the uncovered code would improve correctness but is not executed due to a fault that the student has introduced elsewhere.

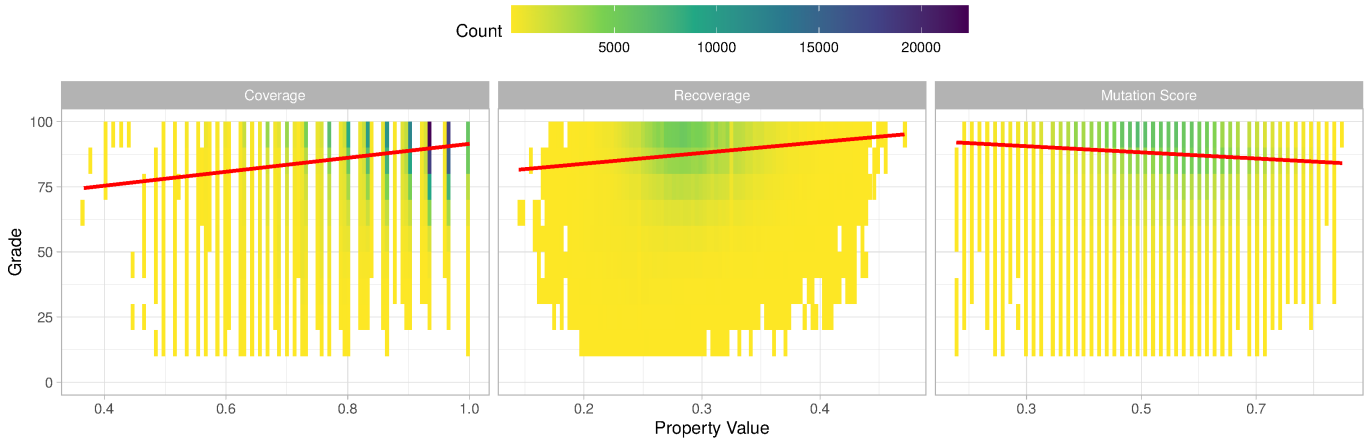
If 100% coverage is difficult to achieve in the model solution, it is possible that the tests alone are not sufficient. For example, the model solution may load test data. In this case, test data should be created that allows for 100% coverage to be achieved. If it remains impossible to gain 100% coverage, there may be problems with either the task's specification or the model solution themselves; a redesign may be required.

Although it is not covered in this paper, branch coverage can also be used to ensure that the test suite sufficiently exercises conditional statements. Most modern IDEs have support for code coverage metrics, and provide coverage highlighting to help identify where code is insufficiently exercised by tests.

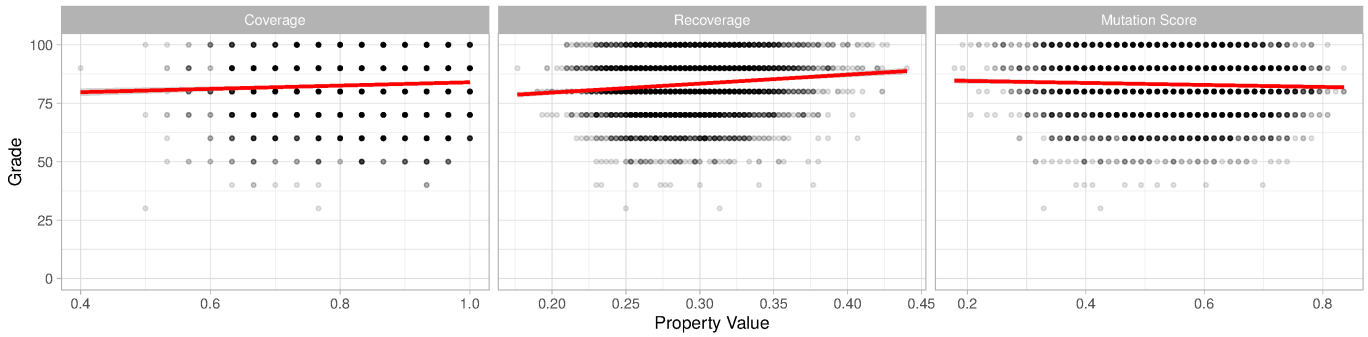
Suggestion 1: Autograding test suites should achieve 100% coverage on a model solution.

B. Recoverage

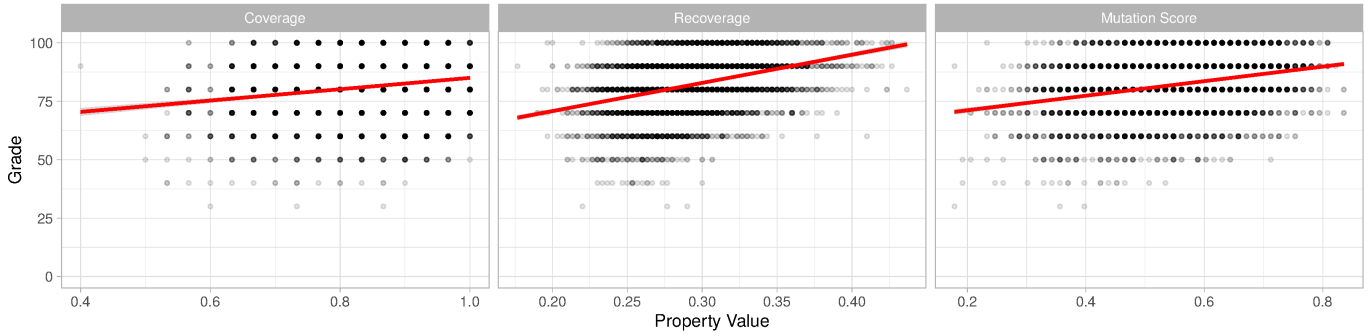
Overall Observations



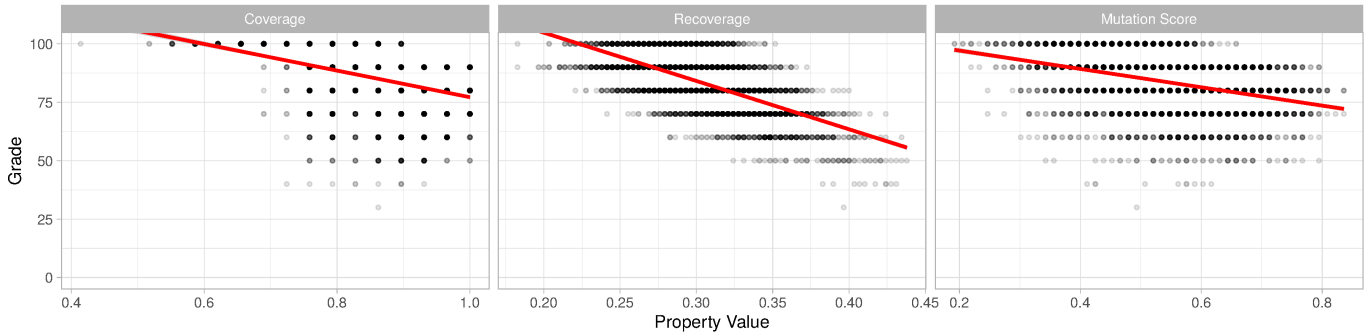
(a) All Mutants



(b) Mutant X (Incorrect Values)



(c) Mutant Y (Incorrect Values)



(d) Mutant Z (Relational Operator Replacement)

Fig. 4: Generated grades vs. each property for individual executions of `Lift` mutants with suites of 10 tests across 30 runs.

Figure 4 shows how suites with varying degrees of average line recoverage generate grades. Recoverage levels are lower than the other two properties for suites of 10 tests, with `Lift`'s maximum recoverage only being ∼48%, despite having multiple test suites that achieve 100% coverage. This shows that these test suites are unlikely to cover most lines of each subject more than once. This may be due to some areas of the code requiring specific conditions to be covered, and the limited size of the test suites.

Across the whole set of mutants, grades tend to increase as recoverage increases. Similarly to coverage, this depends on how tests cover individual mutants, but perhaps to a greater extent on a per mutant basis, where the gradients of the lines of best fit are steeper. At a first glance, this contradicts our findings for RQ2, where *Recoverage* has a very low β coefficient for `Lift` with suites of 10 tests. This may be due to opposite cases (e.g. Mutants Y and Z) effectively cancelling each other out in the set of all mutants. This suggests that, for some faults, recoverage may have a greater impact than our linear models suggest, and as such it should not be ignored.

Individual Mutants

The behaviour typical of the full set of mutants is apparent in Mutant X. Similarly to coverage, the impact of recoverage may be due to how many tests are covering lines affected by the mutant; suites with lower recoverage levels may be recovering only affected lines, while suites with higher recoverage may recover other parts of the code. This indicates that if the faulty lines of a solution are covered by disproportionately many tests, it would lose more grades than if all lines were covered equally. Conversely, if many non-faulty lines are recovered, but a faulty line is only covered once, it may receive an overly high grade. Again, the effect is more prevalent for Mutant Y, likely due to it only having an effect on a single statement.

The reverse effect occurs for Mutant Z, which has grades of ∼50% for suites with high recoverage, and high grades for low recoverage. Since this mutant is always detected if it is covered, and only affects a single method, increased recoverage likely reduces grades as is it more likely that more of the tests in the suite are executing (and thus revealing) the fault. Recovering such fragile faults will have a great impact on the grade that they receive, compared to other faults that may require specific assertions to detect. This may be beneficial, since these faults would greatly impact functionality, but it may also be a source of unfairness if students are not readily able to detect the fault when running the program themselves. Providing students with a set of simple tests to use before submitting their solutions would help to remedy this issue.

Impact Mitigation

Unevenly recovering some lines of code is likely to affect fairness; mistakes that are heavily recovered are more likely to receive lower grades than mistakes that are covered less. While method dependencies and compound branches will make a degree of recoverage inevitable, we recommend that care is taken to avoid needlessly recovering some lines of code more than others. However, for cases where some portions of the code are more important with respect to the task's learning objectives, or where code is more likely to contain subtle faults, having greater recoverage may be beneficial.

We have provided our reference implementation of line recoverage to assist with the identification of recovered code [15].

Suggestion 2: Tutors should be wary of unevenly recovering lines when developing their test suites.

C. Test Quality (Mutation Score)

Overall Observations

Figure 4 shows how mutation score relates to generated grades. One important observation to note in these plots is that no observed test suite achieves a mutation score of 100%, despite typically achieving high levels of coverage. This shows that coverage alone is not sufficient to detect faulty solutions; tests must also make correct and robust assertions, and exercise edge conditions. We find an overall trend of grades falling as mutation score increases.

Individual Mutants

This general behaviour is exhibited by Mutant X. The three methods that this mutant affects may not always demonstrate divergent behaviour when tested:

- `getNumRiders()` would pass tests that expected `numRiders` to be equal to `capacity`.
- `isFull()` would only fail on tests that expect the method to return `false` if `capacity - 1` is added to `numRiders`.
- `addRiders()` cannot reveal this fault alone, but allows for `numRiders` to be manipulated as described above.

As such, merely covering these methods would not necessarily reveal this mutant. However, if tests are able to detect faults in the same methods that exhibit similar behaviour, they are more likely to detect this mutant. Consider a fault that makes `isFull` return `numRiders == capacity - 1`. Any test detecting such a fault would also reveal Mutant X.

This behaviour is present with a stronger effect for Mutant Z. Since this mutant will be detected by any test which covers it, its relationship between grades and mutation score follows a similar pattern to its relationship between grades and coverage. However, these relationships are by no means identical; the mutation scores of test suites do not directly match their coverage levels due to more subtle mutants only being detected by more complex tests. Furthermore, mutation score may have a stronger relationship for this mutant since the method it is in (`call()`) uses multiple conditionals. Other mutants in this method would only be detected by tests that meet these conditions; tests that detect those mutants would also detect this mutant.

Mutant Y exhibits the inverse behaviour; grades increase as mutation score increases. This is a deviation from its behaviour for the other properties, where it followed the same trend as the combination of all mutants. This may be due to a lack of interaction with other mutants; Mutant Y only has an

effect on one method (`getTopFloor()`), with no method dependencies. Furthermore, it only modifies a single statement, and only the *Incorrect Values* operator can be applied to this statement without affecting compilation. Therefore, there will be proportionally few mutants that also modify this method. As such, suites with higher mutation scores would have a smaller proportion of tests that exercise this mutant; it cannot be detected by most of these tests, receiving a high grade. This behaviour is similar to that of coverage and recoverage for this mutant. However, this is not to say that test suites with a high mutation score are incapable of detecting this mutant; no suites with the maximum mutation score ($\sim$83%) generate a 100% grade for this mutant. It is likely that such test suites detect another mutant that affects the same statement.

Impact Mitigation

As discussed in Section IV-A, merely executing faulty statements does not guarantee that the fault is revealed. An effective means to ensure that tests are capable of detecting faulty solutions would be to execute them on such faulty solutions. This could be approached by using existing students' solutions, allowing a tutor to check the distribution of grades generated by the test suite, and to make adjustments if necessary. However, it may have a significant time cost if the correctness of these students' solutions is not already known. This method would be preferable for any tutors seeking to improve an existing autograding test suite or to convert a manually assessed programming task to be autograded.

Another approach is to use mutation analysis, as we did in this paper. By generating artificial mutants from the model solution, a tutor can easily verify that the test suite is able to identify faulty solutions. Any artificial faults that are not detected by the test suite provide information on how to improve the test suite. This is reflected in our data, where some faults have grades of 100% for many test suites, while none of the test suites detect all of the mutants. If the test suites had a perfect mutation score, they would be capable of detecting more faults.

A key benefit of this approach is that it does not require existing solutions; it is suitable for preparing an autograder for a new programming assessment. This can also be combined with student solutions to provide additional confidence that a test suite is capable of detecting faults. Mutants would however not be beneficial for tuning grade distributions, though this can be performed after collecting students' solutions.

Suggestion 3: Tutors should run their test suites against either real or artificial student programs to ensure that they can detect faults.

V. Conclusions & Future Work

In this paper, we have shown that different test suites can yield vastly different grades for faulty solutions. We have also demonstrated that coverage, and to a lesser extent, mutation score and recoverage can each affect generated grades of faulty solution programs. Additionally, we have provided tutors with suggestions on how they can improve the consistency and fairness of their automated grading test suites. Specifically, we recommend writing test suites that achieve 100% coverage, to avoid unnecessarily recovering statements, and to verify the effectiveness of grading test suites using mutation analysis or existing student solutions, where available.

While it has been established that mutants are representative of real faults in software analysis [13], we will conduct future work with real student faults to validate whether this also holds in a grading context. We will also perform further research into the impact of other test suite properties on higher order mutants and real student solutions. Such additional properties may include diagnosability metrics (e.g., [16]), which aim to assess test suites in terms of how well they support fault localisation.

Acknowledgements

We would like to thank Abdullah Alsharif for his advice when writing our data analysis script.

Phil McMinn is supported in part by the Institute of Coding, funded by the Office for Students (OfS), England.

References

[1] S. Krusche and A. Seitz, "ArTEMiS – An Automatic Assessment Management System for Interactive Learning," in *SIGCSE 2018*, (New York, New York, USA), pp. 284–289, ACM Press, 2018.

[2] D. M. Souza, K. R. Felizardo, and E. F. Barbosa, "A systematic literature review of assessment tools for programming assignments," in *CSEET 2016*, pp. 147–156, IEEE, 2016.

[3] D. Insa and J. Silva, "Automatic assessment of Java code," *Computer Languages, Systems and Structures*, vol. 53, pp. 59–72, 2018.

[4] K. Dewey, P. Conrad, M. Craig, and E. Morozova, "Evaluating Test Suite Effectiveness and Assessing Student Code via Constraint Logic Programming," *ITiCSE 2017*, vol. 6, 2017.

[5] M. R. Hoffmann, E. Mandrikov, and M. Friedenhagen, "JaCoCo Java Code Coverage Library." http://eclemma.org/jacoco/, 2016.

[6] G. Fraser, A. Gambi, M. Kreis, and J. M. Rojas, "Gamifying a software testing course with code defenders," *SIGCSE 2019*, pp. 571–577, 2019.

[7] J. M. Rojas, T. D. White, B. Clegg, and G. Fraser, "Code Defenders: Crowdsourcing Effective Tests and Subtle Mutants with a Mutation Testing Game," in *ICSE 2017*, pp. 677–688, IEEE, 2017.

[8] B. Clegg, S. S. North, P. McMinn, and G. Fraser, "Simulating Student Mistakes to Evaluate the Fairness of Automated Grading," in *ICSE-SEET 2019*, pp. 121–125, IEEE, 2019.

[9] R. Just, "The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java," 2014.

[10] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *ASE 2011*, pp. 612–615, IEEE, 2011.

[11] N. Smith, D. Van Bruggen, and F. Tomassetti, "JavaParser: Visited." https://leanpub.com/javaparservisited, 2019.

[12] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, 2006.

[13] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-Nove, pp. 654–665, 2014.

[14] G. C. S. Wang, "How to handle multicollinearity in regression modeling.pdf," *Journal of Business Forecasting Methods & Systems*, vol. 15, no. 1, pp. 23–27, 1996.

[15] B. Clegg, "CoverWeight Git Repository (BitBucket)." https://bitbucket.org/BenClegg/coverweight/src/master/, 2020.

[16] A. Perez, R. Abreu, and A. Van Deursen, "A Test-Suite Diagnosability Metric for Spectrum-Based Fault Localization Approaches," in *ICSE 2017*, pp. 654–664, IEEE, 2017.