Mutation analysis vs. code coverage in automated assessment of students' testing skills

Conference Paper · January 2010

DOI: 10.1145/1869542.1869567 · Source: DBLP

3 authors, including:

Petri Ihantola
Tampere University of Technology
54 PUBLICATIONS **553** CITATIONS

SEE PROFILE

Otto Seppälä
Aalto University
19 PUBLICATIONS **766** CITATIONS

SEE PROFILE

Mutation Analysis vs. Code Coverage in Automated Assessment of Students' Testing Skills

Kalle Aaltonen
kalle.aaltonen@gmail.com

Petri Ihantola Otto Seppälä
Aalto University, Finland
{petri,oseppala}@cs.hut.fi

Abstract

Learning to program should include learning about proper software testing. Some automatic assessment systems, e.g. Web-CAT, allow assessing student-generated test suites using coverage metrics. While this encourages testing, we have observed that sometimes students can get rewarded from high coverage although their tests are of poor quality. Exploring alternative methods of assessment, we have tested mutation analysis to evaluate students' solutions. Initial results from applying mutation analysis to real course submissions indicate that mutation analysis could be used to fix some problems of code coverage in the assessment. Combining both metrics is likely to give more accurate feedback.

Categories and Subject Descriptors K.3.2 [*Computer and Information Science Education*]: Computer science education

General Terms Experimentation, Measurement, Human Factors

Keywords automated assessment, testing, programming assignments, test coverage, mutation analysis, mutation testing

1. Introduction

Students taking introductory programming classes are not usually accustomed to perform their own testing. As a result, they tend to focus on the correctness of the output as specified in the assignment and little else. If the program performs unexpectedly, some manual testing is often done to locate a bug instead of using more systematic approaches. We have observed this effect especially when feedback from automated assessment is available. Spacco and Pugh made similar observations and suggest giving detailed feedback only after students have tested the code also by themselves [14].

Some automated assessment tools allow grading of student tests making it worthwhile for the students to test. At Aalto University (former Helsinki University of Technology) we have used automated assessment of programming assignments at least since 1994 and assessed students' self-written unit tests with Web-CAT[1] [2] since 2006. In Web-CAT, the assessment of student-provided tests is based on the percentage of the student's self-defined tests passing and the structural code coverage (i.e. statement or branch coverage) of these tests. While coverage provides information for the tester about possible places for improvement it might not tell the whole story - this is because good code coverage does not automatically guarantee proper test adequacy. It is well known from the industry that developers can misuse code-coverage-based test adequacy metrics to create a false sense of well tested software [10]. Not too surprisingly we have observed some students to do just the same to please the automated assessment system.

Although Web-CAT performs static analysis to ensure that tests include assertions, getting a good code coverage can be achieved without strong enough assertions or even by checking the assertions before running the code that was to be tested. For example,

- `assertTrue(1 < 2); fibonacci(6);`
- `assertTrue(fibonacci(6) >= 0);`
- `assertEquals(8,fibonacci(6));`

all achieve the same code coverage in automated coverage analysis, although their ability to tell how well the fibonacci method works is quite different. It is even possible that some students do not even see a problem in the first and second examples, which would be even more worrying. At least there are students following approaches similar to all previous examples. When students are rewarded from the code coverage of their tests, they seem to forget the true reason why tests are written.

[1] `http://web-cat.cs.vt.edu/`

SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

2. Research Problem and Method

To tackle the problem of poor test quality in students' written tests, we decided to seek alternative metrics to evaluate the test adequacy.

Mutation analysis is a well known technique performed on a set of unit tests by seeding simple programming errors into the program code to be tested. Each combination of errors applied to the code creates what is called a *mutant*. These "mutants" are generated systematically in large quantities and the examined test suite is run on each of them. The theory is that the test suite that detects more generated defective programs is better than the one that detects less [1]. This makes mutation testing an interesting candidate for use in automatic assessment of student tests. In this paper we apply mutation analysis to real course data to study how this method would perform in an educational setting.

The exact research questions we address are:

Q1: What are the possible strengths and weaknesses of mutation analysis when compared to code coverage based metrics?

Q2: Can mutation analysis be used to give meaningful grading on student-provided test suites requested in programming assignments?

Suitability of mutation testing for test suite assessment was evaluated with test data from Helsinki University of Technology, Intermediate Courses in Programming L1 and T1, held in Fall 2008. These identical courses both teach object oriented programming in Java and are worth 6 ECTS-credits[2]. They have a 5 ECTS-credits CS1 course as their prerequisite. Automatic assessment is used on all courses but students are not required to do unit testing on the prerequisite course. The exercises count for 30 percent of the course grade. All exercises were originally assessed with Web-CAT. The number of resubmissions was not limited. When the course was given no mutation testing was used.

The research method we applied was to compare the test coverage to the *mutation scores*. Both scores were calculated from existing student solutions to three programming assignments. Mutation scores were calculated using Javalanche [12]. During the course the students were "traditionally" awarded points from test coverage. We further investigated submissions that got full points from the coverage but performed poorly in the mutation analysis. In addition, for test set A, we evaluated the effect of different solutions by calculating the mutation scores of each submitted test against all submitted solutions.

Generating and testing mutants requires processing time. Providing instant feedback using mutation analysis in assessment requires keeping the number of mutants at a reasonable level. Minimum, maximum and average counts of mutants are given for each of the test sets.

[2] European Credit Transfer and Accumulation System

3. Related Research

3.1 Automated Assessment of Testing Skills

Both code coverage based metrics and the ability to find faulty implementations are used to automatically evaluate students' tests. Some of the related research conducted before 2006 is summarized in [14].

Assessing the Code Coverage

ASSYST [8] and Web-CAT [2] are perhaps the most widely used tools that combine automated assessment of correctness and tests. Grading in Web-CAT is based on three factors: percentage of teacher's tests passing, percentage of student's own tests passing and code coverage percentage of student's tests. In ASSYST, statement coverage can affect a grade that is originally based on testing the correctness with teachers's tests. Still another system, Marmoset, has been modified to take into account students' tests when grading and giving feedback [14]. By default, Marmoset has the tests grouped into two sets. Feedback from public tests, including test definitions, are given immediately after submission. After the public tests pass, the students can ask for the release tests to be executed. Feedback from the release tests is both limited and delayed. An enhanced version of Marmoset investigates both the code coverage of release tests and student's own tests. As an incentive to test throughly, information about a release test is provided only if student's own tests cover the same as what the release test covers.

Assessing the Ability to Find Bugs

Goldwasser [4] describes an idea where each student on a course provides both a program and a test set - all combinations of which are tried together. Test sets that reveal a lot of bugs and programs that pass a lot of test sets are both rewarded. It is also possible for the staff to seed a faulty implementation to the competition. This also makes it possible to give immediate feedback as students do not need to wait until the exercise deadline when the competition can be performed. Moreover, this allows students to learn from their mistakes. Other papers with similar concept of a competition include e.g. [6, 11]. Elbaum et. al. have created BugHunt [3], a web based tutorial where students write unit tests to reveal problems from given programs.

3.2 Mutation Analysis

Figure 1 explains the process of mutation analysis. Process inputs are the program to be tested and the test suite to be evaluated. In the next phase mutants are generated from the program, and each mutant is tested with the test set. If a mutant fails in the testing it is killed. If it passes all tests, it is called a live mutant. Live mutants are examined in the next phase by hand and split to equivalent and non-equivalent mutants. For example, Listings 2 and 3 are mutants of Listing 1. Note that Listing 2 is functionally identical with the original (i.e. equivalent) whereas Listing 3 is not.

```
int normFib (int N) {
  int curr=1, prev=0;
  for (int i=0; i<N; i++){
    int temp = curr;
    curr     = curr+prev;
    prev     = temp;
  }
  return prev;
}
```

Listing 1. Original

```
int equalFib (int N) {
  int curr=1, prev=0;
  for (int i=1; i<=N; i++){
    int temp = curr;
    curr     = curr+prev;
    prev     = temp;
  }
  return prev;
}
```

Listing 2. Equivalent mutant

```
int mutFib (int N) {
  int curr=1, prev=0;
  for (int i=0; i<=N; i++){
    int temp = curr;
    curr     = curr+prev;
    prev     = temp;
  }
  return prev;
}
```

Listing 3. Non-equivalent mutant

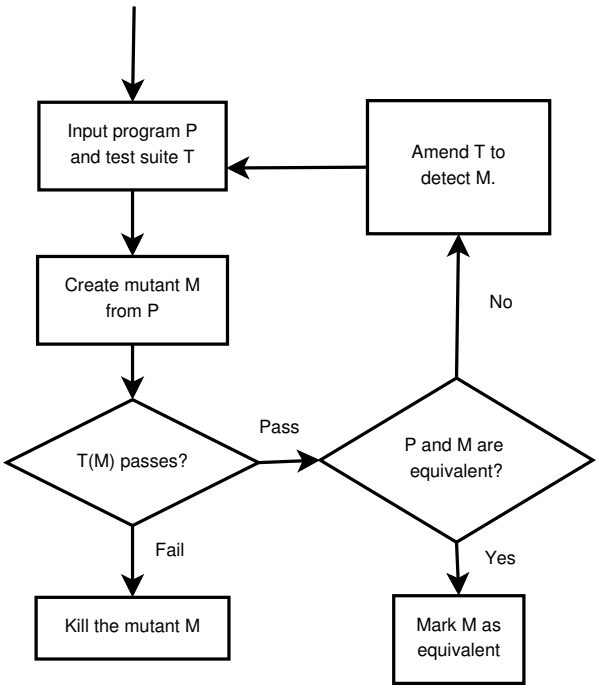


Figure 1. Mutation analysis process

The effectiveness of a test set in mutation analysis is measured by a *mutation score*. This is normally defined as the percentage of non-equivalent mutants killed by the test set. Automated tools often estimate it by dividing the number of *all* live mutants with *all* mutants. The latter is provided by most mutation tools and is also what we have used in this paper.

Mutating Java Programs

Mutants can be generated by modifying a program on different levels – from machine code to interpreted languages with a high abstraction level. Current mutation analysis tools for the Java language generate mutants from the Java source code (e.g. μJava [9]) or from the intermediary bytecode executed by the Java Virtual Machine (e.g. Javalanche [12]), as illustrated in Figure 2. There are pros and cons in both source code level and bytecode level mutants:

- Each examined source code mutant has to be compiled, which is slow.
- Bytecode mutants are difficult to examine afterwards as it's not possible or straightforward to generate the Java source for the mutated bytecode.
- The compiler can eliminate dead code, which in theory can result in fewer equivalent mutants in source code mutants, e.g. if the mutation operation targets a part of the code that's deemed dead by the compiler, the resulting bytecode will be identical with the original.
- Some more advanced operators are significantly easier to implement in Java than in bytecode.

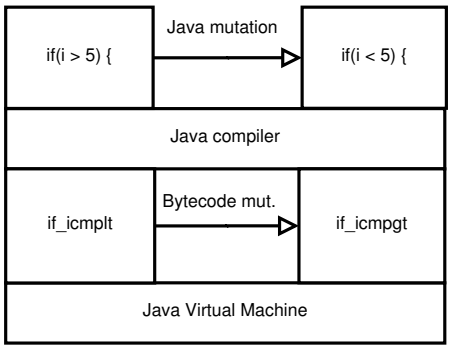


Figure 2. Mutant generation on the Java architecture

μJava[3] is a well known mutation tool for Java, developed since 2003. μJava's mutation operations fall into two distinct classes: method-level mutation operators and class mutation operators. Class level operations are related to encapsulation, inheritance, polymorphism, and some java-specific features (e.g. add or remove keywords like `this` and `static`). Method-level operations, presented in Table 1, are very generic and can be applied to other languages.

Javalanche[4] is a simple and effective bytecode level mutation analysis tool. It replaces numerical constants ($x \rightarrow x+1|x-1|0|1$), negates jump conditions, omits method calls and replaces arithmetic operators. There are no advanced mutation operators related to visibility, inheritance

[3] `http://cs.gmu.edu/~offutt/mujava/`

[4] `http://www.st.cs.uni-saarland.de/~schuler/javalanche/`

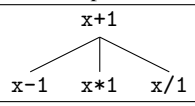
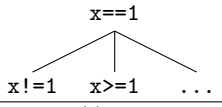
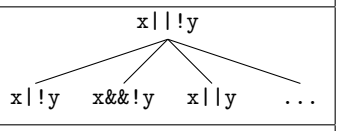
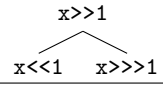
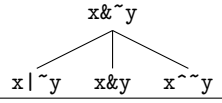
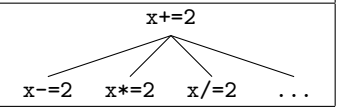
Name	Description	Example
Arithmetic operators	Replace, add, and remove unary and binary arithmetic operators (`+, -, /, *, ++, --`) for both integer and floating point operators.	
Relational operators	replace different comparison operators (`>, >=, <, <=, ==, !=`) within the program.	
Conditional operators	Replace, insert and remove conditional operators (`&&, ||, !`). Bitwise operators `&, |`, and `^` are also used as replacements for these operators as they are very common mistakes.	
Shift operators	Replace bit-wise shifting operators (`<<, >>, >>>`)	
Bitwise operators	Replace, add and move four operators to perform bitwise functions(`&, |, ^, ~`).	
Assignment operators	Replace the convenience assignment operators provided by Java with another (`+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=`).	

Table 1. Method-level mutation operators in μJava. The last column shows a tree diagram where the child nodes are possible mutants generated by this operation applied on the parent.

and polymorphism as μJava has. Javalanche has been successfully used to run mutation analysis on AspectJ, a large open source Java project (with almost 100 thousand lines of code) in under six hours on a single workstation [13].

4. Test Coverage vs. Mutation Score

We analyzed final submissions from each student to three assignments – called test set A, B, and C later on. We failed to apply mutation analysis successfully on some submissions because:

1. The submission did not compile successfully.
2. The test suite did not pass on the unmutated program.
3. Individual tests were not repeatable and independent of the execution order.

This implies that only submissions where all the student's tests passed are analyzed. Table 2 summarizes how many of the submissions were successfully analyzed and how many mutants, on average, were generated from each assignment.

	Mutation analysis			Generated Mutants (per submission)		
Name	All	Applicable	Mutation score avg.	min	max	avg
Set A	158	131 (83.0%)	80.5%	22	90	44.4
Set B	187	174 (90.0%)	73.7%	79	439	106.9
Set C	193	169 (87.6%)	84.9%	12	93	26.4

Table 2. Summary of the test sets used.

4.1 Test set A - Binary search trees

In this exercise the students were instructed to implement a binary search tree by extending an existing binary tree implementation through inheritance. As with all our exercises, unit-testing the solution was required and code coverage was used as the measure of completeness.

Of the 131 analyzed submissions 125 achieved perfect code coverage. This is an expected result, as the students were awarded for reaching perfect coverage, and in the case of this assignment it was relatively easy to achieve.

The mutation analysis yielded an average of 44 mutations per sample, and it took on average about 12 seconds to run per sample. The best work managed to kill 48 of its 49 mutants, resulting in 97.96% mutation score. The one remaining live mutant was identical on the java source code level and thus unkillable, so this can be considered a perfect score. On average the mutation score was 80.48%, and the worst was 40%. The worst mutation score that had reached perfect code coverage was 54.76%. There were several samples where the tested method could be completely commented out, and the test suite would fail to detect this. This would seem to support our assumption that mutation analysis offers better capabilities in identifying weak test sets.

Figure 3 illustrates relationship between mutation score and test coverage in the set as a scatter plot. Histograms on each axis show the distribution of the respective variables. Code coverage (on the X-axis) was 100% for most submissions. This also explains why the correlation between the variables is small ($\rho \approx 0.1628$).

Effect of Implementation to the Generation of Mutants

The binary tree assignment was more restrictive than any of the others. Not only were the students told which methods to implement but they were not allowed to declare any additional instance variables. This was checked automatically. These restrictions made it possible to combine implementations and tests of different students.

In order to show how much variation does the implementation itself (excluding the tests) cause to the mutation score, we selected 4 example test suites from the ones that had achieved 100% test coverage; the worst, the best, and two random ones. We ran mutation analysis on all the previously analyzed submissions with each of these test sets, and results are shown in Figure 4. Each box plot presents one of the four selected test suites. Labels on the X-axis are the original mutation scores and the box plot visualizes how the mutation score varied when the test was executed against implementation of all the other students.

If we are to use the mutation score as an indication to the adequacy of the test set, this score should not be affected by the implementation, but the implementation does affect the number of equivalent mutants created, which makes the mutation scores of two test suites on two different implementations incomparable. It should be noted that in Figure 4 the distribution of the first two test sets seems to be very similar even though the original score is very different. This is something that needs to taken into account if the students are ever rewarded on basis of their mutation score.

4.2 Test set B - Hashing

This was the first exercise on the course and its main idea was to acquaint the students with unit testing. The code to be tested was a pre-implemented hash table implementation using double hashing. The students only had to add in it a method for finding prime numbers – other than that it was all about testing.

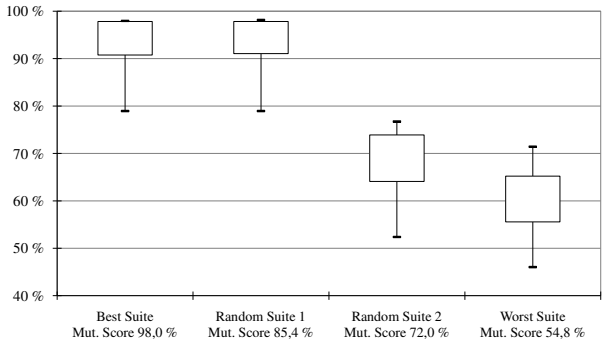


Figure 4. Distribution of mutation scores for the selected test suites as box plot. Displayed are the minimum, maximum 90th and 10th percentiles. X-axis labels are the mutation scores that were achieved when running the mutation analysis on its respective implementation.

Of the 174 analyzed submissions 135 achieved perfect code coverage. This exercise was more complex than test set A, as it yielded on average 106 mutations per sample. Mutation scores of the submissions that achieved perfect coverage score ranged from 42.7% to 89.39% with 73.73% being the average. Mutation scores of the submissions that didn't reach perfect coverage ranged from 24.44% to 85.96% with 63.52% being the average.

The distribution and the relationship between code coverages and mutation scores can be seen in Figure 5.

The best sample reached 100% code coverage and managed to kill 100 out of 112 mutants.

We also examined the sample with the worst mutation score that had reached perfect code coverage: The sample managed to kill 38 of the 89 total mutations, reaching a mutation score of 42.70%. The student generated portion of the test suite had only a single assertion, and half of the unit tests didn't contain any assertions. The test suite is clearly inadequate despite it having perfect branch coverage.

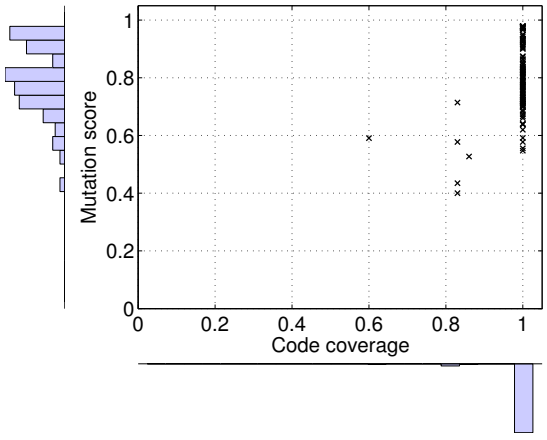


Figure 3. Scatter plot of code coverage and mutation score of the test set A

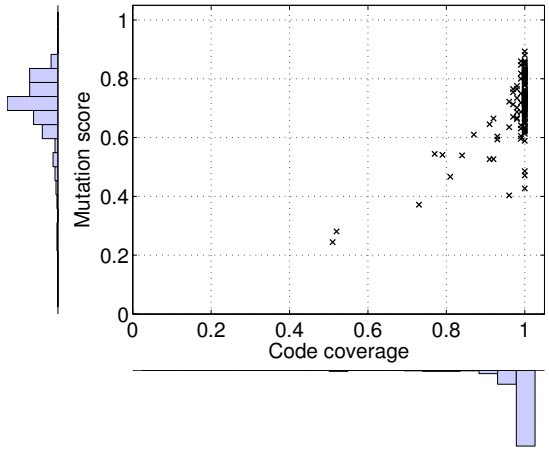


Figure 5. Scatter plot of code coverage and mutation score of the test set B

The distribution seems to be very similar to the distribution seen in test set A (Figure 3), except that the number of samples is higher and there is more variance in the code coverages, $\sigma \approx 0,064$. Pearson product-moment correlation coefficient for the dataset is $\rho \approx 0,669$ indicating a clear positive correlation, unlike in test set A. Major difference with test set A is the maximum mutation score, which was under 90% compared to the practically perfect mutation score achieved in A.

From some submissions we analyzed all mutants that were not killed. Not even the best student generated test suite managed to kill all the non-equivalent mutants.

4.3 Test set C - Disjoint sets

In this exercise the students were instructed to build a simple union-find structure. The exercise allows extracting code shared by different methods into helper methods and has fairly simple recursive and iterative solutions.

Mutation analysis was successfully performed on 169 submissions, of which 144 achieved perfect coverage. Each submission yielded on average 26 mutations. The amount of generated mutants ranged from 12 to 93. Mutation scores of the submissions that achieved perfect coverage score ranged from 38.46% to 95.00% with 84.88% being the average, which is the highest in all the data sets. Mutation scores of the submissions that didn't reach perfect coverage ranged from 21.43% to 90.48% with 67,84% being the average.

The distribution and the relationship between code coverages and mutation scores can be seen in Figure 6.

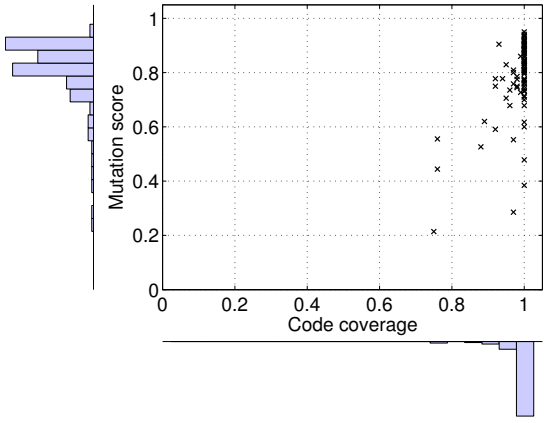


Figure 6. Scatter plot of code coverage and mutation score of the test set C

The best sample had 100% code coverage and killed 19 of its 20 mutations reaching a mutation score of 95.00%, and the remaining mutant was equivalent so this sample should considered mutation adequate.

The submission with perfect code coverage and worst mutation score managed to kill 10 of its 26 total mutations reaching a mutation score of 38.46%.

The distribution seems to be very similar to the distribution seen in the previous test sets (Figure 3 and 5). Pearson product-moment correlation coefficient for the dataset is $\rho \approx 0.6034$ indicating a clear positive correlation.

5. Analysis of Weak Test Sets

We manually examined bottom four samples by mutation score from the samples that had reached perfect coverage. Our initial assumption was that these should be of poor quality by having badly tested or untested functionality. We were not investigating other aspects of test quality, such as style and structural considerations. We also examined the percentage of the teacher's tests passing. It should be noted that all students' tests had to pass their own implementation so that we were able to run the mutation analysis.

5.1 Test set A

All the examined samples had significant problems. For example, none of them tested `printInorder`, at all – although it was fully covered by the tests. Table 3 summarizes our findings from Test set A.

	Mutants			teacher's
Name	generated	detected	score	tests
weak 1	42	23	54.76%	57%
weak 2	54	30	55.56%	86%
weak 3	45	26	57.78%	71%
weak 4	49	29	59.18%	100%

Table 3. Samples with perfect code coverage and bad mutation score of the test set A

5.2 Test set B

All the samples had plenty of untested functionality. Three of the analyzed test sets had only one trivial (although meaningfull) assertion generated by a student. Numerical results from this test set are in Table 4. It should be noted that the most important part of this assignment was to test code that was given. This is why all of the teacher's tests are passing.

	Mutants			teacher's
Name	generated	detected	score	tests
weak 1	89	38	42.70%	100%
weak 2	104	49	47.12%	100%
weak 3	107	52	48.60%	100%
weak 4	90	53	58.89%	100%

Table 4. Samples with perfect code coverage and bad mutation score of the test set B

5.3 Test set C

Unlike in the other test sets, the samples in the test set C were not all of bad quality. Quality of the test of samples 1 and 2 was poor and comparable to test sets A and B. However, tests of samples 3 and 4 were significantly better and it can be argued they only had some corner cases untested but redundant code caused a large number of mutants to be equivalent with the original. Quantitative data from the samples are presented in Table 5.

Name	Mutants generated	detected	score	teacher's tests
weak 1	26	10	38.46%	100%
weak 2	23	11	47.83%	100%
weak 3	25	15	60.00%	86%
weak 4	21	13	61.76%	100%

Table 5. Samples with perfect code coverage and bad mutation score of the test set C

6. Discussion and Conclusions

In the following two subsections we answer to our first research question: *What are the possible strengths (Section 6.1) and weaknesses of mutation analysis (Section 6.2) when compared to code coverage based metrics?* In Section 6.4 we answer our second question: *can mutation analysis be used to give meaningful grading on student-provided test suites requested in programming assignments*?

6.1 Strengths

Automatically assessed exercises are often criticized for not being creative enough. Assessing the functionality of the solution by unit tests written by the teacher implies exercises where students are given the structure of the code. Greening, for example, argues [5, pp. 53–54]:

> Usually, however, the tasks required of the student are highly structured and meticulously synchronized with lectures, and are of the form that asks the student to write a piece of code that satisfies a precise set of specifications created by the instructor. [...] Although some practical skills are certainly gained, the exercise is essentially one of reproduction.

Mutation analysis combines the correctness of the program to be tested (i.e. it can only be applied when tests pass) and the adequacy of the tests. This lessens the need of unit tests written by the teacher and allows more open ended assignments.

In Section 3.1, we described other approaches and assessment tools that also evaluate test set's ability to detect faulty programs. The benefit of mutation analysis over competitions where students' assignments are executed against each other is the ability to give immediate feedback. Immediate feedback would also be possible if faulty programs were generated beforehand by the teacher – as discussed in Section 3.1. However, manual generation of the mutants would prevent automatically assessing more open ended assignments – which mutation analysis could perform. Qualitative analysis in Section 5 implies that mutation analysis is an effective approach for semi-automatic assessment. It could be used with systems like Web-CAT and ASSYST to post-process the submissions and to identify students that may be trying to fool the assessment systems. These submissions could then be manually assessed to ensure this is not the case.

6.2 Weaknesses

One weakness of mutation analysis is that coverage results are easier to interpret and are therefore simpler to use as feedback and assessment criteria with students. While approaches where mutants are used as counterexamples of weak test sets exist, they should be tested in a real course setting to see what is the best way to apply them.

Complex Solutions can be Over-weighted

Complex code creates many mutants and redundant code can cause large numbers of equivalent mutants. This can cause unfairly low mutation scores but can also be used to cheat automatic assessment based on mutation analysis.

When methods contain redundant code or are very complicated the number of mutants blows up. This can lead to a situation where a significant portion of the mutants are from a small untested functionality. This implies that the penalty of not testing that specific funtionality gets too high as demonstrated in Section 5.3.

If students realize that complex code creates many mutants, they may try to fool the mutation analysis system by seeding irrelevant code into their submissions. For example, Listing 4 simply performs the function $f(x) = x + 63$, but the way it is written blows up the number of mutants. This will distort the mutation score. The large number of analyzed mutants can also result in the grading system performing poorly.

```
public static int dummy(int x) {
    x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;
    x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;
    x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;
    return x;
}
```

Listing 4. Sample of an easily testable dummy method, that yields a large number of mutants, and can be thoroughly tested with `assertEquals(63,dummy(0));`.

The number of mutants generated per submission should also be monitored, as it can indicate this kind of cheating, malicious intent in trying to cause the system to perform poorly, or simply an over-complicated solution from where the student should get feedback.

6.3 Testing Unspecified Behavior

Even with assignments where an exact interface to implement is provided, some details of the implementation can be unspecified. For example, how to use return values of methods can be left for students. For students, leaving such unspecified behavior not tested is natural. However, mutation analysis penalizes from this as mutants are also generated from the unspecified behavior. This forces students to specify the otherwise unspecified features through tests.

6.4 Mutation Analysis in Grading

We conclude that mutation analysis can reveal tests that were created to fool the assessment system. Preliminary results indicate that mutation analysis can provide valuable feedback of how well students have tested their software. While the information is most easily interpreted and used by a teacher, the results could be valuable to the students as well. However, to verify this result, a follow up study where students get feedback based on the mutation analysis is needed. An interesting question is if students fool the mutation analysis just like they do for the coverage.

We should also keep in mind that mutation score is not independent from the implementation. Thus, if the objective is to give separate grades from tests and implementation, raw mutation scores are not the best option as they are not commensurable. We assume that it would be possible to set an exercise-specific threshold to identify certainly poor or suspicious work. However, this is where more research is needed.

Although many students submit their work just before the deadline, we assume mutation analysis to scale up and not to be computationally too expensive. For example, analysing a single submission in test set A took 12 seconds on average (see Section 4.1).

7. Future Research

In the future, we would like to see mutation analysis being used to provide formative feedback, i.e. feedback for learning. For example, if mutants are generated on source code level, programs which did not pass student's tests could be provided as feedback. However how to select which of the live mutants to show in an interesting research problem.

Testing can be made more effective by writing code that is easy to test. There are rules how to write testable code and metrics related to measuring the testability. One such metric is provided by a tool called TestabilityExplorer[5] [7]. In the future, we plan to take our data set and apply traditional code coverage, mutation score and TestabilityExplorer to understand how these three different metrics are related to each other. Follow up studies to see how students behave when the immediate feedback they get is based on mutation analysis and/or testability are needed.

References

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11:34–41, 1978.

[2] 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, California, USA, 26–30 October*, pages 148–155. ACM, New York, NY, USA, 2003. ISBN 1-58113-751-6.

[3] S. Elbaum, S. Person, J. Dokulil, and M. Jorde. Bug hunt: Making early software testing lessons engaging and affordable. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 688–697, Washington, DC, USA, 2007. IEEE Computer Society.

[4] M. H. Goldwasser. A gimmick to integrate software testing throughout the curriculum. *SIGCSE Bull.*, 34(1):271–275, 2002. ISSN 0097-8418.

[5] T. Greening. Emerging constructivist forces in computer science education: Shaping a new future. In T. Greening, editor, *Computer science education in the 21st century*, pages 47–80. Springer Verlag, 1999.

[6] M. Hauswirth, D. Zaparanuks, A. Malekpour, and M. Keikha. The javafest: a collaborative learning technique for java programming courses. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 3–12, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-223-8.

[7] M. Hevery. Testability explorer: using byte-code analysis to engineer lasting social changes in an organization's software development process. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 747–748, New York, NY, USA, 2008. ACM.

[8] D. Jackson and M. Usher. Grading student programs using ASSYST. In *Proceedings of 28th ACM SIGCSE Symposium on Computer Science Education*, pages 335–339, 1997.

[9] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, 2005. ISSN 0960-0833.

[10] B. Marick. How to misuse code coverage. In *Proceedings of the 16th International Conference on Testing Computer Software*, pages 16–18, 1999.

[11] W. Marrero and A. Settle. Testing first: emphasizing testing in early programming courses. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 4–8, New York, NY, USA, 2005. ACM. ISBN 1-59593-024-8.

[12] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for java. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium*, pages 297–298, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2.

[13] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 69–80, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-338-9.

[14] J. Spacco and W. Pugh. Helping students appreciate test-driven development (tdd). In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 907–913, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X.

[5] `http://code.google.com/p/testability-explorer/`