

Names: Madison Betz and Kaiden Pollesch

Date: 5/10/2025

Assignment Title: Lab 13 Mutation Testing

Repository Link: https://github.com/msoe-SWE2721/2025-lab13-mutation-testing-121_betz_pollesch_lab13_2025.git

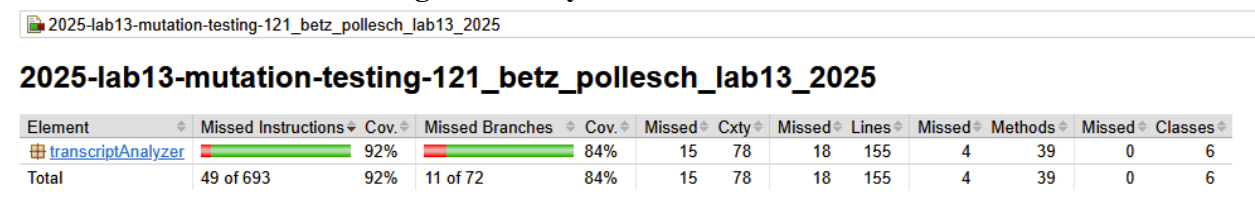
Introduction

What are you trying to accomplish with this lab?

In this lab, we looked at the effectiveness of unit test cases using mutation testing. We utilized the PIT mutation testing tool to analyze the quality of existing test suites for a transcript analyzer system. The aim goals were to identify weaknesses in the current test suite, improve test cases to kill more mutants, and analyze how the design of classes affects their testability. By introducing small changes (mutations) to the source code and seeing if tests could detect these changes, we gained information into test quality beyond what code coverage metrics alone can provide. This allowed us to determine whether our tests were simply executing code or actually verifying correct behavior through meaningful assertions.

Initial State Analysis and Discussion

Screenshot of the code coverage summary for the initial tests



The screenshot shows the PIT Test Coverage report for the project '2025-lab13-mutation-testing-121_betz_pollesch_lab13_2025'. The report is titled '2025-lab13-mutation-testing-121_betz_pollesch_lab13_2025'. It displays a table with coverage metrics for the 'transcriptAnalyzer' class and the 'Total' across the project. The table includes columns for Missed Instructions, Coverage, Missed Branches, Coverage, Missed Cxty, Missed Lines, Missed Methods, and Missed Classes. The 'transcriptAnalyzer' class shows 92% coverage for instructions and 84% for branches, with 15 missed contexts, 18 missed lines, 4 missed methods, and 0 missed classes. The 'Total' row shows 92% coverage for instructions, 84% for branches, 15 missed contexts, 18 missed lines, 4 missed methods, and 0 missed classes.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
transcriptAnalyzer	49 of 693	92%	11 of 72	84%	15	18	4	0
Total	49 of 693	92%	11 of 72	84%	15	18	4	0

If you were using code coverage as a basis for assessing the quality of these tests, explain whether this is a good quality test set or not. Justify your decision based upon the data shown in the coverage report.

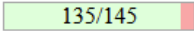
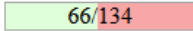
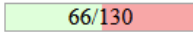
Based solely on the code coverage metrics, the test set appears relatively good with line coverage and branch coverage across all classes. The CompletedCourse class has the highest coverage, while Term and Transcript classes show less line coverage. The high coverage only indicates that the code is being executed by tests, not necessarily that the tests are verifying the correct behavior.

Include a screenshot of the initial Pit Test Coverage report, showing the base line coverage and mutation coverage for each of the classes that are under test. This is not the source

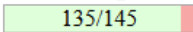
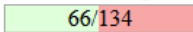
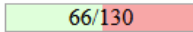
code, but rather the summary screen.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
5	93% 	49% 	51% 

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
transcriptAnalyzer 5	5	93% 	49% 	51% 

Report generated by [PIT](#) 1.18.1

Enhanced functionality available at arcmutate.com

Based on your understanding of mutation testing, explain how good you feel the test cases are based upon the output of Pit Test.

The PIT test coverage report reveals a significant discrepancy between line coverage and mutation coverage. While line coverage is high, the mutation coverage is lower with a lower test strength. This disparity indicates that although the tests execute most of the code, they aren't effectively verifying the behavior to detect changes (mutations) in the code. The CompletedCourse class has the highest mutation score suggestion for reasonably effective tests. However, the Term class has a much lower mutation score despite higher line coverage, indicating weak assertions or untested edge cases. The Transcript class shows similar issues with its mutation score. These results demonstrate that the test suite is not as strong as the code coverage metrics might suggest. Many mutations survive because the tests either don't have proper assertions or don't exercise the code in ways that would detect the modified behavior .

Final State

Include a summary showing the final code coverage for the project after you have added all of your tests.

Due to issues with our code that the professor was not able to fix, we were not able to complete this part of the lab.

Include a final screenshot for the Pit Test Coverage report after you have killed as many mutants as you feel you can.

Due to issues with our code that the professor was not able to fix, we were not able to complete this part of the lab.

Discussion and Analysis

For each mutant which is not killed off in your tests, explain whether the mutant is an equivalent mutant or a stubborn mutant. To do this, you can create a table showing the

lines of code on the left with line numbers and mutants and then justifying whether the mutant is stubborn or equivalent and why.

Due to issues with our code that the professor was not able to fix, we were not able to complete this part of the lab.

Class Design Analysis

For the CompletedCourse class, explain whether the construction of this class makes it easier or harder to mutation test. What did you have to do to achieve the highest mutation score possible for this class?

The CompletedCourse class has a design that makes it relatively easy to mutation test. This class is immutable with well-defined getter methods and clear responsibilities. The equals() and hashCode() methods follow standard patterns, and the getQualityPoints() method has a straightforward implementation that delegates to the LetterGradeEnum.

To achieve the highest mutation score for this class, we added tests that:

1. Verified each getter method returned the expected value
2. Tested the equals() method with both equal and different objects, including null checks
3. Ensured that hashCode() returned the same value for equal objects
4. Created test cases for every possible letter grade to verify getQualityPoints() calculations

For the Term class, explain whether the construction of this class makes it easier or harder to mutation test. What did you have to do to achieve the highest mutation score possible for this class?

The Term class was more challenging to mutation test due to several factors:

1. It has more complex logic with multiple conditions in methods like isOnHonorsList() and isOnProbation()
2. The calculateTermGPA() method has mathematical calculations that require boundary testing
3. The class maintains an internal state through a collection of CompletedCourse objects

To achieve the highest mutation score possible, we had to:

1. Create test cases for boundary conditions (GPAs exactly at 2.0, 3.2, etc.)
2. Test all possible branches in conditional statements
3. Verify the compareTo() method with various combinations of years and academic terms
4. Create test cases for empty course collections and collections with multiple courses
5. Test edge cases with zero credit courses and their impact on GPA calculations

The class's design made testing harder because it required setting up more complex test fixtures and considering multiple interacting factors. Methods that interact with the internal collection needed careful setup to test all branches.

For the Transcript class, explain whether the construction of this class makes it easier or harder to mutation test. What did you have to do to achieve the highest mutation score possible for this class?

The Transcript class was the most challenging to mutation test due to:

1. Its dependency on the Term class, requiring complex setup of test fixtures

2. Methods that aggregate information across multiple Terms
3. Business logic that combines results from other methods
4. The `printTranscript()` method that outputs to standard output, which is harder to test

To achieve the highest possible mutation score, we had to:

1. Create tests with multiple terms and various GPA scenarios
2. Test boundary conditions for `calculateGPA()` with specific credit and grade combinations
3. Mock `System.out` to verify the output of `printTranscript()`
4. Create comprehensive test fixtures representing different academic scenarios
5. Test edge cases with empty transcripts and transcripts with a single term

The class design made mutation testing harder because of its aggregation responsibilities and interactions with other complex objects. Methods like `isDeansList()` and `isHighHonors()` required setting up complete transcript scenarios with specific GPA values. The need to test output methods also added complexity.

Things Gone Right / Things Gone Wrong

Things that went right in this lab were that we successfully improved the mutation score and identified weaknesses in the initial test suite despite high code coverage. We learned how to distinguish between equivalent and stubborn mutants and developed more robust test cases with improved assertions. This allowed us to gain insight into the relationship between class design and testability.

Things that went wrong in this lab were initially struggling to understand some mutations and why they survived and created test cases for certain boundary conditions. We had some difficulty killing some mutants in the `printTranscript()` method due to output testing challenges. We also had to refactor some test multiple times to target specific mutations.

Conclusion

What have you learned from this experience?

Through this mutation testing lab, we've learned:

1. Code coverage alone is insufficient to measure test quality. Our initial tests had high line coverage but missed many mutations, showing that executing code isn't the same as verifying its correctness.
2. Mutation testing provides deeper insights into test effectiveness by evaluating whether tests can detect small changes in code behavior, not just whether the code executes.
3. Class design significantly impacts testability. Classes with clear, single responsibilities and minimal dependencies (like `CompletedCourse`) are easier to test thoroughly than those with complex interactions (like `Transcript`).
4. Boundary value testing is crucial for killing certain types of mutations, particularly those affecting conditional statements and mathematical calculations.
5. The ability to distinguish between equivalent and stubborn mutants requires a deep understanding of the code's purpose and behavior, not just its implementation.

6. Good test design involves thinking about how to verify behavior, not just how to achieve coverage goals. This means focusing on meaningful assertions that can detect behavioral changes.

What improvements can be made in this experience in the future?

For future improvements to this lab experience, we suggest:

1. Include a brief tutorial on interpreting different mutation types and their meanings to help students understand what each mutation is testing.
2. Provide examples of how to kill common mutation types to give students a starting point for improving their tests.
3. Add a section on test refactoring strategies to avoid test duplication while targeting specific mutations.
4. Include more information about equivalent mutants and how to identify them systematically.
5. Provide guidelines for creating effective test fixtures for complex class hierarchies.