Project Report

University Course Registration System

Mutation Testing

Submitted by:

Sakina Baranwala (MT2024130)

Yashraj Singh Chauhan (MT2024170)

Date: November 25, 2025

Abstract

This report presents the final results of testing and mutation analysis for a **University Course Registration System**. The system supports student enrollment, course management, prerequisite validation, grade computation, GPA/CGPA calculation, timetable scheduling and clash detection.

A comprehensive **unit test suite** and **integration test suite** were developed using JUnit 5. The effectiveness of these tests was then evaluated using **mutation testing** with the PIT (Pitest) framework. Mutation testing was applied to both unit and integration tests to assess how well they detect artificially injected faults (mutants).

The final PIT report shows:

- **94% line coverage** (356 / 378 lines),
- **87% mutation coverage** (173 / 199 mutants killed),
- **92% test strength** (173 / 188).

These results demonstrate that the test suites are robust and effective at detecting logical faults in the system.

Introduction

The University Course Registration System is designed to support common academic operations, including:

- Managing students and courses,
- Enrolling and dropping courses,
- Applying prerequisite rules,
- Assigning grades and calculating GPA/CGPA,
- Scheduling course time slots and detecting timetable clashes.

To ensure the correctness and reliability of these operations, two main categories of tests were created:

- **Unit tests** for individual classes and methods,
- **Integration tests** for complete workflows across layers.

Mutation testing with PIT was then performed **on top of** these tests, to quantify how resistant the system is to common implementation errors. PIT introduces mutants in the production code and uses the existing JUnit tests (unit and integration) to check whether these mutants are detected (killed) or not (survived).

System Overview

The project follows a layered architecture, with separate model, repository and service layers.

Model Layer

The model layer contains the core domain entities:

- **Student** – student id, name, current semester, CGPA and completed courses.
- **Course** – course code, title, credits, maximum capacity, semester offered and prerequisites.
- **Enrollment** – a student's enrollment in a course with status, marks obtained and grade.
- **GradeReport** – stores semester-wise GPA and total credits for a student.
- **EnrollmentRequest** – simple DTO used when requesting enrollment.
- **TimeSlot** – represents a teaching slot with day and start/end hours.
- **CourseSchedule** – associates a course in a given semester with a list of time slots.

Repository Layer

Repositories provide in-memory storage and querying:

- **StudentRepository**
- **CourseRepository**
- **EnrollmentRepository**
- **GradeReportRepository**

- **CourseScheduleRepository**

They support actions such as saving entities, finding by id or code, and custom queries (e.g., counting enrollments per course and semester).

Service Layer

The service layer implements business logic:

- **StudentService** – updates CGPA from grade reports.
- **GPAService** – calculates GPA based on grades and credits.
- **GradeService** – maps marks to letter grades and grade points.
- **PrerequisiteService** – checks prerequisites, computes prerequisite depth, and detects cycles in prerequisite graphs.
- **RegistrationService** – handles course enrollment and dropping, checking capacity, semester rules, prerequisites and credit limits.
- **TimetableService** – checks for timetable clashes and builds a student's timetable from course schedules and enrollments.

Testing Approach

In this project, a JUnit-based test suite was developed and organized into:

- **Unit test suite:** tests individual components in isolation.
- **Integration test suite:** tests end-to-end workflows across layers.

Mutation testing with PIT was then applied to the combined test suite to measure how well it detects faults at both unit and integration levels.

Unit Test Suite

Unit tests were written for:

- **Model classes:**
  - Constructors, getters and setters,
  - `equals()`, `hashCode()`, and `toString()`,
  - Boundary conditions (e.g., invalid hours in `TimeSlot`).
- **Service classes:**
  - Grade conversion in `GradeService`,
  - GPA and CGPA computations in `GPAService` and `StudentService`,
  - Prerequisite checks and depth in `PrerequisiteService`,
  - Timetable clash detection in `TimetableService`.
- **Repository classes:**
  - Save and retrieve operations,
  - ID assignment,
  - Queries such as counting enrollments for a course in a semester.

Strongly Killed Mutants at Unit Level

Mutation testing showed that the unit tests strongly killed multiple mutants. Three representative examples are:

- **Boundary mutator in `TimeSlot` constructor:** PIT mutated the validation expression to allow invalid hour ranges (e.g., changing `startHour < 0` or `startHour >= endHour`). Unit tests supplying invalid hours (negative, greater than 23, end before start) detected these mutants by expecting an `IllegalArgumentException`.
- **Boolean-return mutant in `TimeSlot.clashesWith()`:** A mutant changed the overlap logic so that overlapping slots returned `false`. Unit tests with overlapping time intervals on the same day asserted that clashes must be reported, thereby killing this mutant.
- **Removed conditional in `Student.equals()`:** Mutants that removed parts of the equality logic (e.g., type checks or id comparisons) were detected by unit tests that compare equal and non-equal `Student` instances, verifying correct behavior of `equals()` and `hashCode()`.

Integration Test Suite

Integration tests focus on complete academic workflows involving multiple layers:

- **Enrollment workflow:** verifying that a student can enroll in an existing course when capacity, prerequisites, and semester rules are satisfied, and that the enrollment is persisted correctly.
- **Prerequisite failure scenarios:** checking that attempting to enroll without required completed courses raises the appropriate exception.
- **GPA and CGPA calculation:** validating GPA and CGPA using multiple `GradeReport` entries, ensuring correct aggregation across semesters.
- **Timetable scheduling and clashes:** scheduling multiple courses with time slots and verifying correct clash/no-clash behavior when enrolling in additional courses.

- **End-to-end flows:** combining enrollment, prerequisite validation and timetable clash detection in a single integrated scenario.

Strongly Killed Mutants at Integration Level

Mutation testing also showed that integration tests strongly killed several multi-class mutants. Representative examples are:

- **Null-return mutant in `RegistrationService.enroll()`:** A mutant caused the method to return `null` instead of a valid `Enrollment`. Integration tests that expect a non-null enrollment and then query the repository for its status failed on this mutant, killing it.
- **Removed prerequisite check in `PrerequisiteService/RegistrationService`:** Mutants that omitted prerequisite validation allowed students to enroll without having completed required courses. Integration tests that expect an `InvalidPrerequisiteException` for such cases detected and killed these mutants.
- **Clash-detection mutant in `TimetableService`:** A mutant changed the clash-checking condition to always report no clash. Integration tests that construct overlapping schedules and expect a clash detected this change, killing the mutant.

Mutation Testing on Unit and Integration Tests

PIT was configured to use both:

- Unit test classes (e.g., `*Test`),
- Integration test classes (e.g., `*IT`).

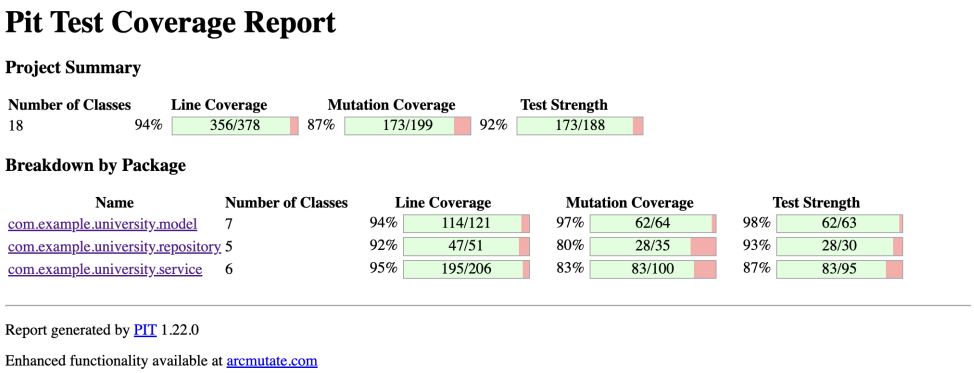
This means that **the same PIT run** evaluates how effectively **both** unit and integration tests kill mutants across the codebase. The results reported in the next section therefore reflect the combined strength of the entire test suite.

Final PIT Mutation Testing Results

Overall Summary

The final PIT run produced the following global metrics:

- **Number of classes mutated:** 18
- **Line coverage:** 356 / 378 = 94%
- **Mutation coverage:** 173 / 199 = 87%
- **Test strength:** 173 / 188 = 92%



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
18	94%	356/378	87%	173/199	92%	173/188

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.example.university.model	7	94%	114/121	97%	62/64	98%	62/63
com.example.university.repository	5	92%	47/51	80%	28/35	93%	28/30
com.example.university.service	6	95%	195/206	83%	83/100	87%	83/95

Report generated by PIT 1.22.0

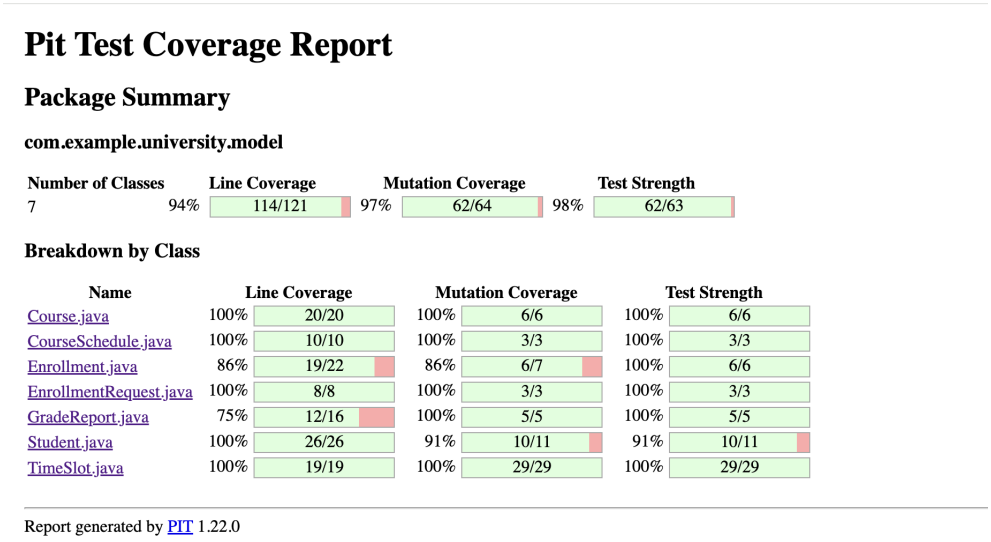
Enhanced functionality available at arcmutate.com

Figure 1: PIT overall mutation testing summary.

Coverage by Package

Model Package

- Line coverage: 114 / 121 = 94%
- Mutation coverage: 62 / 64 = 97%
- Test strength: 62 / 63 = 98%



Pit Test Coverage Report

Package Summary

com.example.university.model

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
7	94%	114/121	97%	62/64	98%	62/63

Breakdown by Class

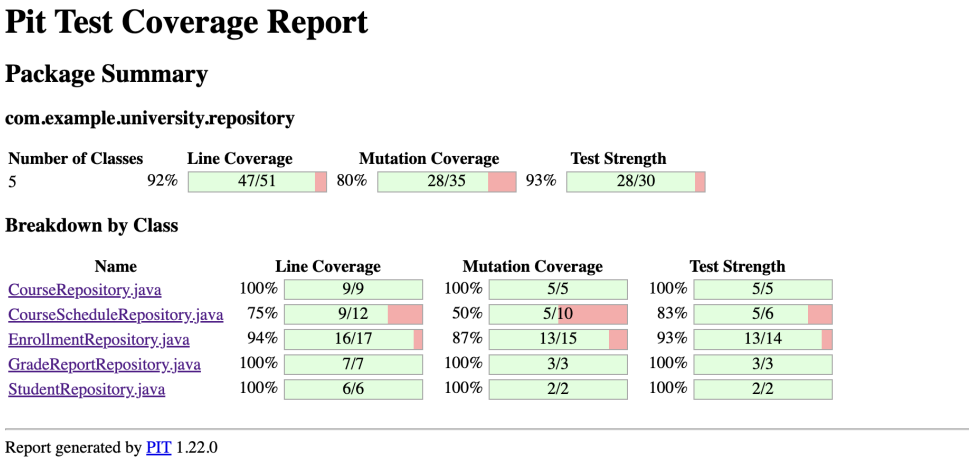
Name	Line Coverage		Mutation Coverage		Test Strength	
Course.java	100%	20/20	100%	6/6	100%	6/6
CourseSchedule.java	100%	10/10	100%	3/3	100%	3/3
Enrollment.java	86%	19/22	86%	6/7	100%	6/6
EnrollmentRequest.java	100%	8/8	100%	3/3	100%	3/3
GradeReport.java	75%	12/16	100%	5/5	100%	5/5
Student.java	100%	26/26	91%	10/11	91%	10/11
TimeSlot.java	100%	19/19	100%	29/29	100%	29/29

Report generated by PIT 1.22.0

Figure 2: PIT report for `com.example.university.model`.

Repository Package

- Line coverage: 47 / 51 = 92%
- Mutation coverage: 28 / 35 = 80%
- Test strength: 28 / 30 = 93%



Pit Test Coverage Report

Package Summary

com.example.university.repository

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
5	92%	47/51	80%	28/35	93%	28/30

Breakdown by Class

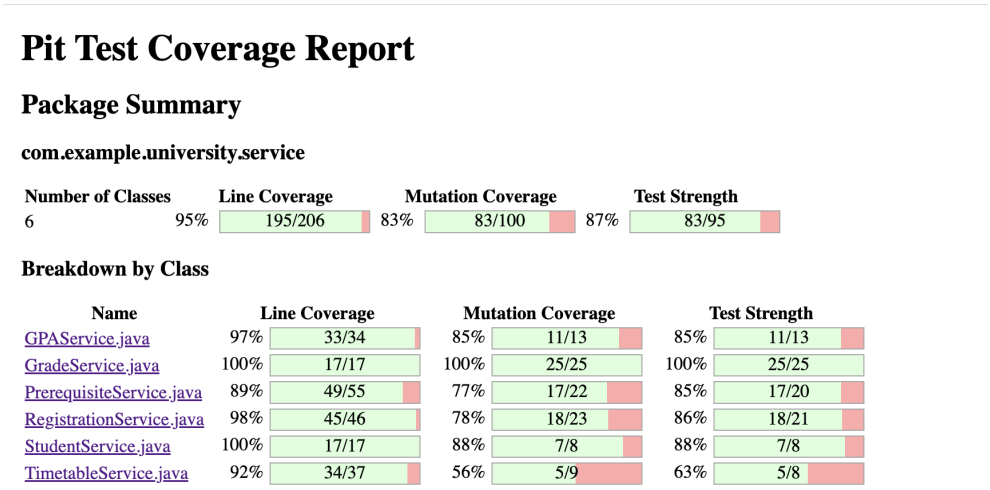
Name	Line Coverage		Mutation Coverage		Test Strength	
CourseRepository.java	100%	9/9	100%	5/5	100%	5/5
CourseScheduleRepository.java	75%	9/12	50%	5/10	83%	5/6
EnrollmentRepository.java	94%	16/17	87%	13/15	93%	13/14
GradeReportRepository.java	100%	7/7	100%	3/3	100%	3/3
StudentRepository.java	100%	6/6	100%	2/2	100%	2/2

Report generated by PIT 1.22.0

Figure 3: PIT report for `com.example.university.repository`.

Service Package

- Line coverage: 195 / 206 = 95%
- Mutation coverage: 83 / 100 = 83%
- Test strength: 83 / 95 = 87%



Pit Test Coverage Report

Package Summary

com.example.university.service

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
6	95%	195/206	83%	83/100	87%	83/95

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
GPAService.java	97%	33/34	85%	11/13	85%	11/13
GradeService.java	100%	17/17	100%	25/25	100%	25/25
PrerequisiteService.java	89%	49/55	77%	17/22	85%	17/20
RegistrationService.java	98%	45/46	78%	18/23	86%	18/21
StudentService.java	100%	17/17	88%	7/8	88%	7/8
TimetableService.java	92%	34/37	56%	5/9	63%	5/8

Report generated by PIT 1.22.0

Figure 4: PIT report for `com.example.university.service`.

Mutator Statistics and Logs

PIT applied several mutation operators, including:

- **CONDITIONALS_BOUNDARY** – modifies comparison operators such as `<`, `<=`, `>`, `>=`.
- **NEGATE_CONDITIONALS** – negates boolean conditions.
- **MATH** – changes arithmetic operators (e.g., `+` to `-`, `/` to `*`).
- **PRIMITIVE_RETURNS** – replaces primitive return values with constants (e.g., 0, 1, -1).
- **TRUE_RETURNS** / **FALSE_RETURNS** – forces boolean methods to always return `true` or `false`.
- **EMPTY_RETURNS** / **NULL_RETURNS** – returns empty collections or `null`.
- **REMOVE_CONDITIONALS** – removes conditional branches.
- **VOID_METHOD_CALL** – removes calls to void methods (e.g., repository `save()` calls).

Mutator Statistics Screenshot

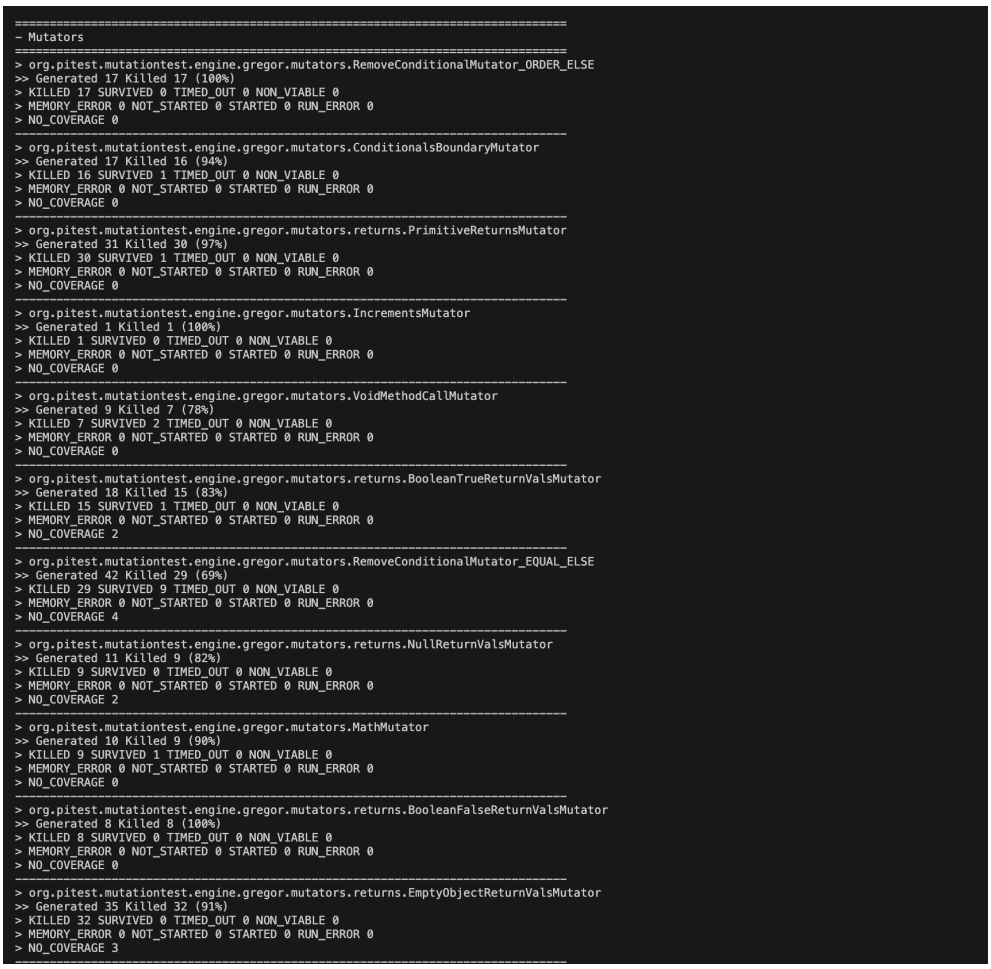


Figure 5: PIT mutator usage and kill statistics.

Final Build Log

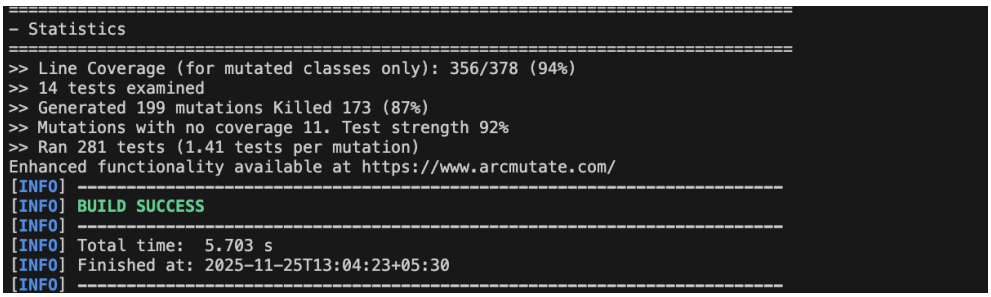


Figure 6: PIT build output showing successful mutation test execution.

Conclusion

The University Course Registration System has been thoroughly validated using a combination of unit and integration tests, with mutation testing applied to both. The final metrics:

- 94% line coverage,
- 87% mutation coverage,
- 92% test strength,

indicate a strong and effective test suite. Representative mutants at both unit and integration levels were strongly killed, demonstrating that the tests detect boundary errors, logic faults, and missing method calls.

Overall, the system shows high resilience to common implementation mistakes and is well-prepared for future extension or deployment.

References

- GeeksforGeeks – Java, Testing, and Software Engineering Concepts
- Javatpoint – Java Programming and OOP References
- PIT Mutation Testing Framework Documentation

Project Repository

The full implementation, source code, unit tests, integration tests, and mutation testing configuration are available at:

- **GitHub Repository: University Course Registration – Mutation Testing**