Software Testing and Quality Assurance Project Report

Course: Software Testing and Quality Assurance - Fall 2025

University: Prince Mohammad Bin Fahd University

College: College of Computer Engineering and Science

Project Team:

Ahmad Al-Dawood 202201115

Mohammed Al-Shammasi 202201348

Abdulhadi Al-Sultan 202201246

Date:

Contents

Project Abstract

This project focuses on the comprehensive testing of a Student Grade Evaluation System, designed to manage student records, course enrollments, and grade calculations. The primary objective is to ensure the reliability and correctness of the system through White-Box and Black-Box testing techniques.

The testing strategy involves the implementation of JUnit test classes for three core components: Student, Course, and GradeCalculator, ensuring high method coverage. Additionally, a specialized driver was developed for the GradeCalculator class to perform Black-Box testing using Equivalence Partitioning and Boundary Value Analysis.

The project deliverables include a detailed IEEE 829-compliant test plan, a statement and code coverage report generated via JaCoCo, and a suite of automated test cases.

1. Testing Plan (IEEE 829 Standard)

1.1 Objectives

The primary objective of this test plan is to verify the functionality, reliability, and correctness of the Student Grade Evaluation System. Specifically, we aim to:

- Validate that all methods in the Student, Course, and GradeCalculator classes function as expected.
- Ensure the GradeCalculator correctly handles all valid and invalid inputs using Black-Box techniques.
- Achieve high code coverage (aiming for 100%) to ensure all logic paths are tested.
- Identify and document any defects or anomalies in the code.

1.2 Test Items

The following software items are the targets of this testing plan:

- Course.java: Handles course details (name, credit hours, letter grade) and grade point conversion.
- Student.java: Manages student identity and course enrollments and calculates GPA.
- GradeCalculator.java: Contains logic for converting numeric scores to letter grades.

1.3 Features to be Tested

- Course Management: Creating courses, validating inputs (credit hours, grades), and retrieving course details.
- Student Enrollment: Enrolling students in courses and maintaining the course list.
- GPA Calculation: Accurately calculating GPA based on enrolled courses and credit hours, including edge cases (no courses, all Fs).
- Grade Calculation: Converting numeric scores (0-100) to letter grades (A-F) and handling invalid inputs.

1.4 Features not to be Tested

- User Interface (UI): The project is a backend logic implementation; no graphical user interface is being tested.
- Database Persistence: The current system uses in-memory data structures (Lists); database integration is out of scope.
- Performance/Load Testing: The focus is on functional correctness, not high-concurrency performance.

1.5 Approach

The testing strategy employs a hybrid approach combining White-Box and Black-Box testing:

- Unit Testing (White-Box)
- Tool: JUnit 5.
- Methodology: Create a test class for each source class (CourseTest, StudentTest, GradeCalculatorTest).
- Goal: Cover all constructors, getters, and logic methods. Use JaCoCo to measure statement and branch coverage.
- Black-Box Testing
- Target: GradeCalculator.calculateLetterGrade(int score).

Methodology:

- Equivalence Partitioning (EP): Divide inputs into valid ranges (A, B, C, D, F) and invalid ranges (<0, >100). Test one value from each.
- Boundary Value Analysis (BVA): Test values at the edges of equivalence classes (e.g., 0, 59, 60, 89, 90, 100).
- Implementation: A custom Java driver (GradeCalculatorDriver) executes these scenarios and reports Pass/Fail status.

1.6 Item Pass/Fail Criteria

- Unit Tests: A test case passes if the actual output matches the expected output defined in the JUnit assertions. The entire suite passes if all JUnit tests execute without failure.
- Black-Box Driver: A test case passes if the calculated letter grade matches the expected grade for the given input.
- Coverage: The project is considered successfully tested if code coverage exceeds 90% for the target classes.

1.7 Test Deliverables

- Test Plan: This document.
- Test Cases: Source code of JUnit tests (*Test.java) and the Driver (GradeCalculatorDriver.java).
- Test Reports:
- JUnit execution results (console output/IDE report).
- Driver execution logs (console output).
- JaCoCo Code Coverage Report.
- Source Code: The complete project source code including main and test directories.

1.8 Testing Tasks

Requirement Analysis: Review the project PDF and understand the grading logic.

Test Design: Identify equivalence classes and boundary values for GradeCalculator.

Implementation:

- Write CourseTest to verify the Course class.
- Write StudentTest to verify the Student class and GPA logic.
- Write GradeCalculatorTest for unit testing.
- Implement GradeCalculatorDriver for Black-Box scenarios.
- Execution: Run all JUnit tests and the Driver.

Reporting: Generate JaCoCo coverage reports and compile the final project report.

1.9 Environmental Needs

Hardware: Test ran on CISC and RISC hardware, no difference should be found.

Operating System: Only tested with windows.

Software:

- Java Development Kit (JDK) 8 or higher.
- Maven (for build and dependency management).
- IDE (IntelliJ IDEA, Eclipse, or VS Code).
- JaCoCo Plugin.
- JUnit

1.10 Responsibilities

Team Members: Responsible for designing test cases, writing code, running tests, and documenting results.

1.11 Schedule

Task	Start Date	End Date	Owner(s)
Project Setup & Analysis	11/24/2025	11/25/2025	All Members
Test Case Design	11/24/2025	11/25/2025	All Members
JUnit Implementation	11/24/2025	11/25/2025	[S1, S2, S3]
Driver Implementation	11/24/2025	11/25/2025	[S4]
Test Execution & Debugging	11/24/2025	11/25/2025	All Members
Report & Presentation Finalization	11/24/2025	11/25/2025	All Members
Final Submission	**01-12-2025**	**01-12-2025**	**All Members**

2. Code Coverage Report (JaCoCo)

This section summarizes the code coverage achieved by our JUnit test suites.

Overall Coverage Summary:

Figure 1 JaCoCo coverage report

Regarding coverage, it is impossible to achieve 100% coverage of this program as there is some dead code that will never be executed. Under Student.java:

```
        if (totalCredits == 0) {
            return 0.0;
        }
```

As well as

```
        if (gpa > 4.0) {
            gpa = 4.0;
        }
        return gpa;
```

These branches and instructions will never be run due to limitations by the constructor and the GPA calculator method.

Analysis: [Provide a brief analysis of the coverage results. For example: "Our test suites achieved 95% statement coverage. The missing 5% corresponds to an unreachable error-handling block, which we have documented as a known limitation."]

3. Black-Box Testing Cases

This section details the specific black-box test cases applied to the calculateLetterGrade method in GradeCalculator, which were executed via the standalone GradeCalculatorDriver.

3.1. Equivalence Partitioning Test Cases

These cases confirm that the system correctly maps typical scores within each functional range to the expected letter grade, including invalid ranges.

Test ID	Test Technique	Input Score	Expected Grade	Description
GC-043	Equivalence Partitioning	90	A	Driver EC1: Boundary Start (A)
GC-044	Equivalence Partitioning	95	A	Driver EC1: Middle of Class (A)
GC-046	Equivalence Partitioning	85	B	Driver EC2: Middle of Class (B)
GC-049	Equivalence Partitioning	75	C	Driver EC3: Middle of Class (C)
GC-052	Equivalence Partitioning	65	D	Driver EC4: Middle of Class (D)
GC-056	Equivalence Partitioning	30	F	Driver EC5: Middle of Class (F)
GC-058	Equivalence Partitioning	-1	Invalid	Driver EC6: Negative Invalid
GC-062	Equivalence Partitioning	150	Invalid	Driver EC7: Positive Invalid

3.2. Boundary Value Analysis Test Cases

These cases focus on values immediately surrounding the boundaries of the defined score ranges to ensure correct transitions.

Test ID	Test Technique	Input Score	Expected Grade	Description
GC-064	Boundary Value Analysis	0	F	BVA1: Valid Minimum Boundary
GC-065	Boundary Value Analysis	-1	Invalid	BVA2: Just Below Valid Range

GC-066	Boundary Value Analysis	59	F	BVA3: Boundary F/D (Lower)
GC-067	Boundary Value Analysis	60	D	BVA3: Boundary F/D (Upper)
GC-068	Boundary Value Analysis	69	D	BVA4: Boundary D/C (Lower)
GC-069	Boundary Value Analysis	70	C	BVA4: Boundary D/C (Upper)
GC-072	Boundary Value Analysis	89	B	BVA6: Boundary B/A (Lower)
GC-073	Boundary Value Analysis	90	A	BVA6: Boundary B/A (Upper)
GC-074	Boundary Value Analysis	100	A	BVA7: Valid Maximum Boundary
GC-075	Boundary Value Analysis	101	Invalid	BVA8: Just Above Valid Range

4.0 Output of Test Executions

This section will include the output of the test executions. This includes the Junit tests along with the Driver Execution Test which is a comprehensive testing suite

4.1 JUnit Test Execution Output

Output of CourseTest

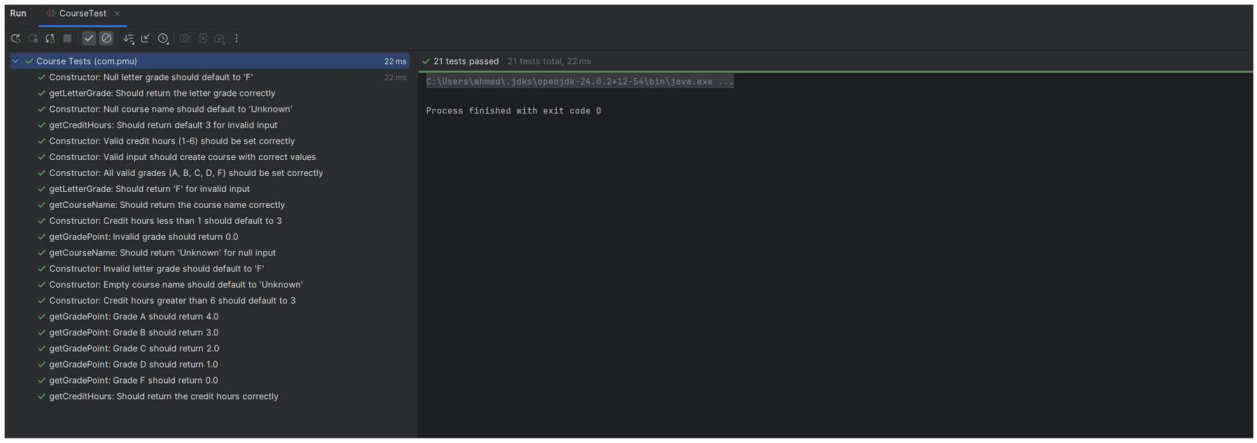

Figure 2 showing Junit CourseTest output.

Output of *GradeCalculator*Test

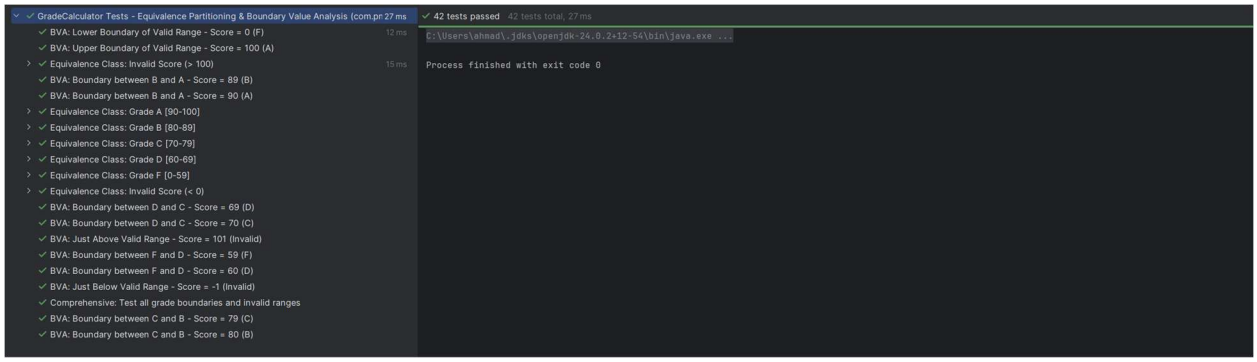

Figure 3 showing Junit GradeCalculatorTest output.

Output of StudentTest

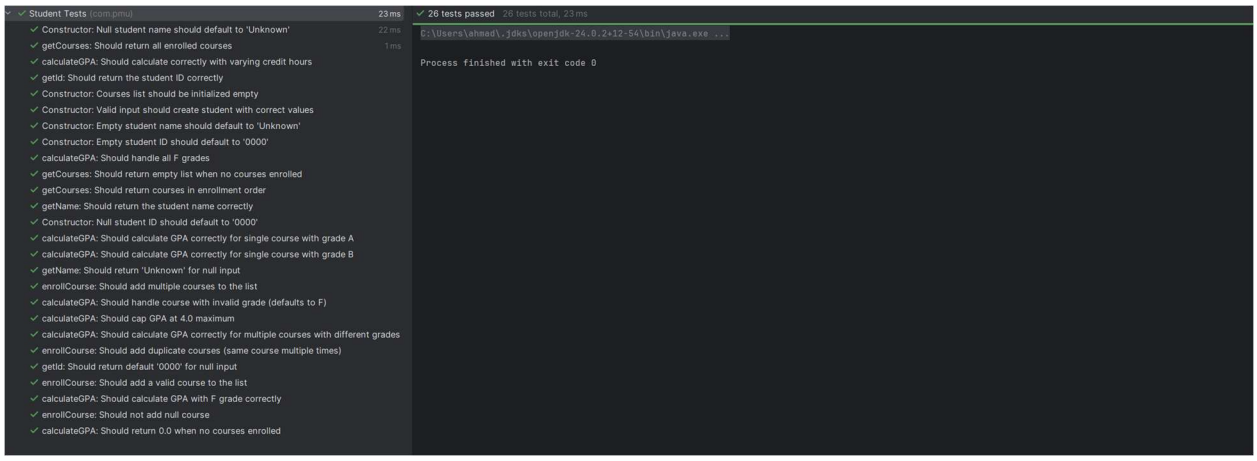

Figure 4 showing Junit StudentTest output.

4.2 Driver Execution Output

```
Grade Calculator - Black Box Testing Driver
```

```
Testing: calculateLetterGrade(int score)
================================================================================

================================================================================
1. EQUIVALENCE PARTITIONING TESTING
================================================================================

Valid Equivalence Classes:
--------------------------------------------------------------------------------
EC1: Grade A [90-100]
  PASS: Grade A - Score: 90 | Expected: A | Got: A
  PASS: Grade A - Score: 95 | Expected: A | Got: A
  PASS: Grade A - Score: 100 | Expected: A | Got: A

EC2: Grade B [80-89]
  PASS: Grade B - Score: 80 | Expected: B | Got: B
  PASS: Grade B - Score: 85 | Expected: B | Got: B
  PASS: Grade B - Score: 89 | Expected: B | Got: B

EC3: Grade C [70-79]
  PASS: Grade C - Score: 70 | Expected: C | Got: C
  PASS: Grade C - Score: 75 | Expected: C | Got: C
  PASS: Grade C - Score: 79 | Expected: C | Got: C

EC4: Grade D [60-69]
  PASS: Grade D - Score: 60 | Expected: D | Got: D
  PASS: Grade D - Score: 65 | Expected: D | Got: D
  PASS: Grade D - Score: 69 | Expected: D | Got: D

EC5: Grade F [0-59]
  PASS: Grade F - Score: 0 | Expected: F | Got: F
  PASS: Grade F - Score: 30 | Expected: F | Got: F
  PASS: Grade F - Score: 59 | Expected: F | Got: F


--------------------------------------------------------------------------------
Invalid Equivalence Classes:
--------------------------------------------------------------------------------
EC6: Invalid Score (< 0)
  PASS: Negative score - Score: -1 | Expected: Invalid | Got: Invalid
  PASS: Negative score - Score: -50 | Expected: Invalid | Got: Invalid
  PASS: Negative score - Score: -100 | Expected: Invalid | Got: Invalid

EC7: Invalid Score (> 100)
  PASS: Score > 100 - Score: 101 | Expected: Invalid | Got: Invalid
  PASS: Score > 100 - Score: 150 | Expected: Invalid | Got: Invalid
  PASS: Score > 100 - Score: 200 | Expected: Invalid | Got: Invalid

================================================================================
2. BOUNDARY VALUE ANALYSIS TESTING
================================================================================

Boundary Values and Transitions:
--------------------------------------------------------------------------------
BVA1: Lower Boundary of Valid Range
  PASS: Score = 0 (Valid/F) | Expected: F | Got: F

BVA2: Just Below Valid Range
  PASS: Score = -1 (Invalid) | Expected: Invalid | Got: Invalid

BVA3: Boundary between F and D
  PASS: Score = 59 (F) | Expected: F | Got: F
  PASS: Score = 60 (D) | Expected: D | Got: D
```

```
BVA4: Boundary between D and C
  PASS: Score = 69 (D) | Expected: D | Got: D
  PASS: Score = 70 (C) | Expected: C | Got: C

BVA5: Boundary between C and B
  PASS: Score = 79 (C) | Expected: C | Got: C
  PASS: Score = 80 (B) | Expected: B | Got: B

BVA6: Boundary between B and A
  PASS: Score = 89 (B) | Expected: B | Got: B
  PASS: Score = 90 (A) | Expected: A | Got: A

BVA7: Upper Boundary of Valid Range
  PASS: Score = 100 (Valid/A) | Expected: A | Got: A

BVA8: Just Above Valid Range
  PASS: Score = 101 (Invalid) | Expected: Invalid | Got: Invalid

================================================================================
TEST SUMMARY
================================================================================
Total Tests Run: 33
Tests Passed:     33 (100.0%)
Tests Failed:     0 (0.0%)
================================================================================
All tests passed! The calculateLetterGrade method is working correctly.
================================================================================
```

5.0 Conclusion

This report documents the complete testing lifecycle for the student grade evaluation system. The implemented test suites, combining White-Box and Black-Box techniques, have thoroughly validated the system's functionality. The high code coverage achieved with JaCoCo confirms the effectiveness of the tests. All objectives outlined in the test plan have been successfully met.

Appendix: Test Code

5.1 GradeCalculatorDriver.java

```java
package com.pmu;

/**
 * Driver Program for GradeCalculator Testing
 *
 * This driver tests the calculateLetterGrade method using:
 * 1. Equivalence Partitioning (Black-box testing)
 * 2. Boundary Value Analysis (Black-box testing)
 *
 * Test Strategy:
 * - Divides the input range (0-100) into equivalence classes
 * - Tests boundary values to ensure correct transitions between grades
 * - Tests invalid inputs (< 0 and > 100)
 */
```

```java
public class GradeCalculatorDriver {

    private static int totalTests = 0;
    private static int passedTests = 0;
    private static int failedTests = 0;

    public static void main(String[] args) {
        printSeparator('=', 80);
        System.out.println("Grade Calculator - Black Box Testing Driver");
        System.out.println("Testing: calculateLetterGrade(int score)");
        printSeparator('=', 80);
        System.out.println();

        GradeCalculator calculator = new GradeCalculator();

        // Run all test categories
        testEquivalencePartitioning(calculator);
        System.out.println();
        testBoundaryValueAnalysis(calculator);
        System.out.println();

        // Print summary
        printTestSummary();
    }

    private static void printSeparator(char c, int count) {
        for (int i = 0; i < count; i++) {
            System.out.print(c);
        }
        System.out.println();
    }

    /**
     * Test using Equivalence Partitioning
     * Divides input space into equivalence classes and tests one value from each
     */
    private static void testEquivalencePartitioning(GradeCalculator calculator) {
        printSeparator('=', 80);
        System.out.println("1. EQUIVALENCE PARTITIONING TESTING");
        printSeparator('=', 80);
        System.out.println();

        // Valid Equivalence Classes
        System.out.println("Valid Equivalence Classes:");
        printSeparator('-', 80);

        // Class 1: Grade A [90-100]
        System.out.println("EC1: Grade A [90-100]");
        testScores(calculator, new int[]{90, 95, 100}, "A", "Grade A");

        // Class 2: Grade B [80-89]
        System.out.println("\nEC2: Grade B [80-89]");
        testScores(calculator, new int[]{80, 85, 89}, "B", "Grade B");

        // Class 3: Grade C [70-79]
        System.out.println("\nEC3: Grade C [70-79]");
        testScores(calculator, new int[]{70, 75, 79}, "C", "Grade C");

        // Class 4: Grade D [60-69]
        System.out.println("\nEC4: Grade D [60-69]");
        testScores(calculator, new int[]{60, 65, 69}, "D", "Grade D");

        // Class 5: Grade F [0-59]
```

```
        System.out.println("\nEC5: Grade F [0-59]");
        testScores(calculator, new int[]{0, 30, 59}, "F", "Grade F");

        // Invalid Equivalence Classes
        System.out.println("\n");
        printSeparator('-', 80);
        System.out.println("Invalid Equivalence Classes:");
        printSeparator('-', 80);

        // Class 6: Score < 0
        System.out.println("EC6: Invalid Score (< 0)");
        testScores(calculator, new int[]{-1, -50, -100}, "Invalid", "Negative score");

        // Class 7: Score > 100
        System.out.println("\nEC7: Invalid Score (> 100)");
        testScores(calculator, new int[]{101, 150, 200}, "Invalid", "Score > 100");
    }

    /**
     * Test using Boundary Value Analysis
     * Tests critical values at the boundaries between equivalence classes
     */
    private static void testBoundaryValueAnalysis(GradeCalculator calculator) {
        printSeparator('=', 80);
        System.out.println("2. BOUNDARY VALUE ANALYSIS TESTING");
        printSeparator('=', 80);
        System.out.println();

        System.out.println("Boundary Values and Transitions:");
        printSeparator('-', 80);

        // Boundary: Start of valid range (0 = F)
        System.out.println("BVA1: Lower Boundary of Valid Range");
        testScore(calculator, 0, "F", "Score = 0 (Valid/F)");

        // Boundary: Just below valid range (-1 = Invalid)
        System.out.println("\nBVA2: Just Below Valid Range");
        testScore(calculator, -1, "Invalid", "Score = -1 (Invalid)");

        // Boundary: Between F and D (59/60)
        System.out.println("\nBVA3: Boundary between F and D");
        testScore(calculator, 59, "F", "Score = 59 (F)");
        testScore(calculator, 60, "D", "Score = 60 (D)");

        // Boundary: Between D and C (69/70)
        System.out.println("\nBVA4: Boundary between D and C");
        testScore(calculator, 69, "D", "Score = 69 (D)");
        testScore(calculator, 70, "C", "Score = 70 (C)");

        // Boundary: Between C and B (79/80)
        System.out.println("\nBVA5: Boundary between C and B");
        testScore(calculator, 79, "C", "Score = 79 (C)");
        testScore(calculator, 80, "B", "Score = 80 (B)");

        // Boundary: Between B and A (89/90)
        System.out.println("\nBVA6: Boundary between B and A");
        testScore(calculator, 89, "B", "Score = 89 (B)");
        testScore(calculator, 90, "A", "Score = 90 (A)");

        // Boundary: End of valid range (100 = A)
        System.out.println("\nBVA7: Upper Boundary of Valid Range");
        testScore(calculator, 100, "A", "Score = 100 (Valid/A)");
```

```
        // Boundary: Just above valid range (101 = Invalid)
        System.out.println("\nBVA8: Just Above Valid Range");
        testScore(calculator, 101, "Invalid", "Score = 101 (Invalid)");
    }

    /**
     * Test a single score and verify the result
     */
    private static void testScore(GradeCalculator calculator, int score, String
expectedGrade, String description) {
        totalTests++;
        String result = calculator.calculateLetterGrade(score);
        boolean passed = result.equals(expectedGrade);

        if (passed) {
            passedTests++;
            System.out.printf("  PASS: %s | Expected: %s | Got: %s%n", description,
expectedGrade, result);
        } else {
            failedTests++;
            System.out.printf("  FAIL: %s | Expected: %s | Got: %s%n", description,
expectedGrade, result);
        }
    }

    /**
     * Test multiple scores and verify they all return the expected grade
     */
    private static void testScores(GradeCalculator calculator, int[] scores, String
expectedGrade, String description) {
        for (int score : scores) {
            testScore(calculator, score, expectedGrade, description + " - Score: " +
score);
        }
    }

    /**
     * Print test summary statistics
     */
    private static void printTestSummary() {
        printSeparator('=', 80);
        System.out.println("TEST SUMMARY");
        printSeparator('=', 80);
        System.out.printf("Total Tests Run:  %d%n", totalTests);
        System.out.printf("Tests Passed:     %d (%.1f%%)%n", passedTests, (passedTests
* 100.0 / totalTests));
        System.out.printf("Tests Failed:     %d (%.1f%%)%n", failedTests, (failedTests
* 100.0 / totalTests));
        printSeparator('=', 80);

        if (failedTests == 0) {
            System.out.println("All tests passed! The calculateLetterGrade method is
working correctly.");
        } else {
            System.out.println("Some tests failed. Please review the
implementation.");
        }
        printSeparator('=', 80);
    }
}
```

5.2 CourseTest.java

src/test/java/com/pmu/CourseTest.java

```
package com.pmu;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.DisplayName;

import static org.junit.jupiter.api.Assertions.*;

/**
 * Test class for Course
 * Tests all methods: constructor, getCourseName, getCreditHours,
getLetterGrade, getGradePoint
 */
@DisplayName("Course Tests")
class CourseTest {

    private Course course;

    @BeforeEach
    void setUp() {
        // Initialize a valid course for each test
        course = new Course("Java Programming", 3, "A");
    }

    // ============ Constructor Tests ============

    @Test
    @DisplayName("Constructor: Valid input should create course with correct
values")
    void testConstructorWithValidInput() {
        Course testCourse = new Course("Data Structures", 4, "B");
        assertEquals("Data Structures", testCourse.getCourseName());
        assertEquals(4, testCourse.getCreditHours());
        assertEquals("B", testCourse.getLetterGrade());
    }

    @Test
    @DisplayName("Constructor: Null course name should default to 'Unknown'")
    void testConstructorNullCourseName() {
        Course testCourse = new Course(null, 3, "A");
        assertEquals("Unknown", testCourse.getCourseName());
    }

    @Test
    @DisplayName("Constructor: Empty course name should default to
'Unknown'")
    void testConstructorEmptyCourseName() {
        Course testCourse = new Course("", 3, "A");
        assertEquals("Unknown", testCourse.getCourseName());
    }

    @Test
    @DisplayName("Constructor: Credit hours less than 1 should default to 3")
```

```
    void testConstructorCreditHoursTooLow() {
        Course testCourse = new Course("Java", 0, "A");
        assertEquals(3, testCourse.getCreditHours());
    }

    @Test
    @DisplayName("Constructor: Credit hours greater than 6 should default to
3")
    void testConstructorCreditHoursTooHigh() {
        Course testCourse = new Course("Java", 7, "A");
        assertEquals(3, testCourse.getCreditHours());
    }

    @Test
    @DisplayName("Constructor: Valid credit hours (1-6) should be set
correctly")
    void testConstructorValidCreditHours() {
        for (int hours = 1; hours <= 6; hours++) {
            Course testCourse = new Course("Java", hours, "A");
            assertEquals(hours, testCourse.getCreditHours());
        }
    }

    @Test
    @DisplayName("Constructor: Invalid letter grade should default to 'F'")
    void testConstructorInvalidLetterGrade() {
        Course testCourse = new Course("Java", 3, "Z");
        assertEquals("F", testCourse.getLetterGrade());
    }

    @Test
    @DisplayName("Constructor: Null letter grade should default to 'F'")
    void testConstructorNullLetterGrade() {
        Course testCourse = new Course("Java", 3, null);
        assertEquals("F", testCourse.getLetterGrade());
    }

    @Test
    @DisplayName("Constructor: All valid grades (A, B, C, D, F) should be set
correctly")
    void testConstructorAllValidGrades() {
        String[] validGrades = {"A", "B", "C", "D", "F"};
        for (String grade : validGrades) {
            Course testCourse = new Course("Java", 3, grade);
            assertEquals(grade, testCourse.getLetterGrade());
        }
    }

    // ============ getCourseName() Tests ============

    @Test
    @DisplayName("getCourseName: Should return the course name correctly")
    void testGetCourseName() {
        assertEquals("Java Programming", course.getCourseName());
    }

    @Test
```

```java
@DisplayName("getCourseName: Should return 'Unknown' for null input")
void testGetCourseNameUnknown() {
    Course testCourse = new Course(null, 3, "A");
    assertEquals("Unknown", testCourse.getCourseName());
}

// ============ getCreditHours() Tests ============

@Test
@DisplayName("getCreditHours: Should return the credit hours correctly")
void testGetCreditHours() {
    assertEquals(3, course.getCreditHours());
}

@Test
@DisplayName("getCreditHours: Should return default 3 for invalid input")
void testGetCreditHoursDefault() {
    Course testCourse = new Course("Java", -1, "A");
    assertEquals(3, testCourse.getCreditHours());
}

// ============ getLetterGrade() Tests ============

@Test
@DisplayName("getLetterGrade: Should return the letter grade correctly")
void testGetLetterGrade() {
    assertEquals("A", course.getLetterGrade());
}

@Test
@DisplayName("getLetterGrade: Should return 'F' for invalid input")
void testGetLetterGradeInvalid() {
    Course testCourse = new Course("Java", 3, "InvalidGrade");
    assertEquals("F", testCourse.getLetterGrade());
}

// ============ getGradePoint() Tests ============

@Test
@DisplayName("getGradePoint: Grade A should return 4.0")
void testGetGradePointA() {
    Course testCourse = new Course("Java", 3, "A");
    assertEquals(4.0, testCourse.getGradePoint());
}

@Test
@DisplayName("getGradePoint: Grade B should return 3.0")
void testGetGradePointB() {
    Course testCourse = new Course("Java", 3, "B");
    assertEquals(3.0, testCourse.getGradePoint());
}

@Test
@DisplayName("getGradePoint: Grade C should return 2.0")
void testGetGradePointC() {
    Course testCourse = new Course("Java", 3, "C");
    assertEquals(2.0, testCourse.getGradePoint());
```

```
    }

    @Test
    @DisplayName("getGradePoint: Grade D should return 1.0")
    void testGetGradePointD() {
        Course testCourse = new Course("Java", 3, "D");
        assertEquals(1.0, testCourse.getGradePoint());
    }

    @Test
    @DisplayName("getGradePoint: Grade F should return 0.0")
    void testGetGradePointF() {
        Course testCourse = new Course("Java", 3, "F");
        assertEquals(0.0, testCourse.getGradePoint());
    }

    @Test
    @DisplayName("getGradePoint: Invalid grade should return 0.0")
    void testGetGradePointInvalid() {
        Course testCourse = new Course("Java", 3, "Invalid");
        assertEquals(0.0, testCourse.getGradePoint());
    }
}
```

5.3 GradeCalculatorTest.java

src/test/java/com/pmu/GradeCalculatorTest.java

```
package com.pmu;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.*;

/**
 * Test class for GradeCalculator
 *
 * Testing Strategy: Equivalence Partitioning + Boundary Value Analysis
(Black-box testing)
 *
 * Equivalence Classes:
 * 1. Valid Score Range: 0-100
 *    - A: [90-100]
 *    - B: [80-89]
 *    - C: [70-79]
 *    - D: [60-69]
 *    - F: [0-59]
 * 2. Invalid Score Range: < 0 or > 100
 *
 * Boundary Values:
 * - Score < 0 (e.g., -1, -100)
 * - Score = 0 (boundary for F)
 * - Score = 59 (boundary between F and D)
```

```
 * - Score = 60 (boundary for D)
 * - Score = 69 (boundary between D and C)
 * - Score = 70 (boundary for C)
 * - Score = 79 (boundary between C and B)
 * - Score = 80 (boundary for B)
 * - Score = 89 (boundary between B and A)
 * - Score = 90 (boundary for A)
 * - Score = 100 (boundary for A/valid)
 * - Score > 100 (e.g., 101, 150)
 */
@DisplayName("GradeCalculator Tests - Equivalence Partitioning & Boundary
Value Analysis")
class GradeCalculatorTest {

    private GradeCalculator gradeCalculator;

    @BeforeEach
    void setUp() {
        gradeCalculator = new GradeCalculator();
    }

    // ============ EQUIVALENCE PARTITIONING TESTS ============

    @DisplayName("Equivalence Class: Grade A [90-100]")
    @ParameterizedTest
    @ValueSource(ints = {90, 91, 95, 99, 100})
    void testEquivalenceClassA(int score) {
        assertEquals("A", gradeCalculator.calculateLetterGrade(score),
                "Score " + score + " should return grade A");
    }

    @DisplayName("Equivalence Class: Grade B [80-89]")
    @ParameterizedTest
    @ValueSource(ints = {80, 81, 85, 89})
    void testEquivalenceClassB(int score) {
        assertEquals("B", gradeCalculator.calculateLetterGrade(score),
                "Score " + score + " should return grade B");
    }

    @DisplayName("Equivalence Class: Grade C [70-79]")
    @ParameterizedTest
    @ValueSource(ints = {70, 71, 75, 79})
    void testEquivalenceClassC(int score) {
        assertEquals("C", gradeCalculator.calculateLetterGrade(score),
                "Score " + score + " should return grade C");
    }

    @DisplayName("Equivalence Class: Grade D [60-69]")
    @ParameterizedTest
    @ValueSource(ints = {60, 61, 65, 69})
    void testEquivalenceClassD(int score) {
        assertEquals("D", gradeCalculator.calculateLetterGrade(score),
                "Score " + score + " should return grade D");
    }

    @DisplayName("Equivalence Class: Grade F [0-59]")
    @ParameterizedTest
```

```
@ValueSource(ints = {0, 1, 30, 50, 59})
void testEquivalenceClassF(int score) {
    assertEquals("F", gradeCalculator.calculateLetterGrade(score),
            "Score " + score + " should return grade F");
}

@DisplayName("Equivalence Class: Invalid Score (< 0)")
@ParameterizedTest
@ValueSource(ints = {-1, -10, -100})
void testEquivalenceClassInvalidNegative(int score) {
    assertEquals("Invalid", gradeCalculator.calculateLetterGrade(score),
            "Score " + score + " should return Invalid");
}

@DisplayName("Equivalence Class: Invalid Score (> 100)")
@ParameterizedTest
@ValueSource(ints = {101, 110, 150, 200})
void testEquivalenceClassInvalidPositive(int score) {
    assertEquals("Invalid", gradeCalculator.calculateLetterGrade(score),
            "Score " + score + " should return Invalid");
}

// ============ BOUNDARY VALUE ANALYSIS TESTS ============

@Test
@DisplayName("BVA: Lower Boundary of Valid Range - Score = 0 (F)")
void testBVALowerBoundaryValid() {
    assertEquals("F", gradeCalculator.calculateLetterGrade(0));
}

@Test
@DisplayName("BVA: Just Below Valid Range - Score = -1 (Invalid)")
void testBVAJustBelowValid() {
    assertEquals("Invalid", gradeCalculator.calculateLetterGrade(-1));
}

@Test
@DisplayName("BVA: Boundary between F and D - Score = 59 (F)")
void testBVABoundaryFD_Lower() {
    assertEquals("F", gradeCalculator.calculateLetterGrade(59));
}

@Test
@DisplayName("BVA: Boundary between F and D - Score = 60 (D)")
void testBVABoundaryFD_Upper() {
    assertEquals("D", gradeCalculator.calculateLetterGrade(60));
}

@Test
@DisplayName("BVA: Boundary between D and C - Score = 69 (D)")
void testBVABoundaryDC_Lower() {
    assertEquals("D", gradeCalculator.calculateLetterGrade(69));
}

@Test
@DisplayName("BVA: Boundary between D and C - Score = 70 (C)")
void testBVABoundaryDC_Upper() {
```

```
        assertEquals("C", gradeCalculator.calculateLetterGrade(70));
    }

    @Test
    @DisplayName("BVA: Boundary between C and B - Score = 79 (C)")
    void testBVABoundaryCB_Lower() {
        assertEquals("C", gradeCalculator.calculateLetterGrade(79));
    }

    @Test
    @DisplayName("BVA: Boundary between C and B - Score = 80 (B)")
    void testBVABoundaryCB_Upper() {
        assertEquals("B", gradeCalculator.calculateLetterGrade(80));
    }

    @Test
    @DisplayName("BVA: Boundary between B and A - Score = 89 (B)")
    void testBVABoundaryBA_Lower() {
        assertEquals("B", gradeCalculator.calculateLetterGrade(89));
    }

    @Test
    @DisplayName("BVA: Boundary between B and A - Score = 90 (A)")
    void testBVABoundaryBA_Upper() {
        assertEquals("A", gradeCalculator.calculateLetterGrade(90));
    }

    @Test
    @DisplayName("BVA: Upper Boundary of Valid Range - Score = 100 (A)")
    void testBVAUpperBoundaryValid() {
        assertEquals("A", gradeCalculator.calculateLetterGrade(100));
    }

    @Test
    @DisplayName("BVA: Just Above Valid Range - Score = 101 (Invalid)")
    void testBVAJustAboveValid() {
        assertEquals("Invalid", gradeCalculator.calculateLetterGrade(101));
    }

    // ============ COMPREHENSIVE TESTS ============

    @Test
    @DisplayName("Comprehensive: Test all grade boundaries and invalid
ranges")
    void testComprehensiveAllRanges() {
        // Invalid negative
        assertEquals("Invalid", gradeCalculator.calculateLetterGrade(-50));

        // F range
        assertEquals("F", gradeCalculator.calculateLetterGrade(0));
        assertEquals("F", gradeCalculator.calculateLetterGrade(30));
        assertEquals("F", gradeCalculator.calculateLetterGrade(59));

        // D range
        assertEquals("D", gradeCalculator.calculateLetterGrade(60));
        assertEquals("D", gradeCalculator.calculateLetterGrade(65));
        assertEquals("D", gradeCalculator.calculateLetterGrade(69));
```

```
        // C range
        assertEquals("C", gradeCalculator.calculateLetterGrade(70));
        assertEquals("C", gradeCalculator.calculateLetterGrade(75));
        assertEquals("C", gradeCalculator.calculateLetterGrade(79));

        // B range
        assertEquals("B", gradeCalculator.calculateLetterGrade(80));
        assertEquals("B", gradeCalculator.calculateLetterGrade(85));
        assertEquals("B", gradeCalculator.calculateLetterGrade(89));

        // A range
        assertEquals("A", gradeCalculator.calculateLetterGrade(90));
        assertEquals("A", gradeCalculator.calculateLetterGrade(95));
        assertEquals("A", gradeCalculator.calculateLetterGrade(100));

        // Invalid positive
        assertEquals("Invalid", gradeCalculator.calculateLetterGrade(150));
    }
}
```

5.4 StudentTest.java

src/test/java/com/pmu/StudentTest.java

```
package com.pmu;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.DisplayName;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

/**
 * Test class for Student
 * Tests all methods: constructor, getId, getName, enrollCourse, getCourses,
calculateGPA
 */
@DisplayName("Student Tests")
class StudentTest {

    private Student student;
    private Course courseA;
    private Course courseB;
    private Course courseF;

    @BeforeEach
    void setUp() {
        // Create a valid student
        student = new Student("S001", "mohammed");

        // Create test courses with different grades
        courseA = new Course("Math", 3, "A");      // Grade point: 4.0
        courseB = new Course("English", 4, "B");   // Grade point: 3.0
        courseF = new Course("Science", 2, "F");   // Grade point: 0.0
```

```
    }

    // ============ Constructor Tests ============

    @Test
    @DisplayName("Constructor: Valid input should create student with correct
values")
    void testConstructorWithValidInput() {
        Student testStudent = new Student("S002", "ahmad");
        assertEquals("S002", testStudent.getId());
        assertEquals("ahmad", testStudent.getName());
    }

    @Test
    @DisplayName("Constructor: Null student ID should default to '0000'")
    void testConstructorNullId() {
        Student testStudent = new Student(null, "ahmad");
        assertEquals("0000", testStudent.getId());
    }

    @Test
    @DisplayName("Constructor: Empty student ID should default to '0000'")
    void testConstructorEmptyId() {
        Student testStudent = new Student("", "ahmad");
        assertEquals("0000", testStudent.getId());
    }

    @Test
    @DisplayName("Constructor: Null student name should default to
'Unknown'")
    void testConstructorNullName() {
        Student testStudent = new Student("S001", null);
        assertEquals("Unknown", testStudent.getName());
    }

    @Test
    @DisplayName("Constructor: Empty student name should default to
'Unknown'")
    void testConstructorEmptyName() {
        Student testStudent = new Student("S001", "");
        assertEquals("Unknown", testStudent.getName());
    }

    @Test
    @DisplayName("Constructor: Courses list should be initialized empty")
    void testConstructorCoursesEmpty() {
        assertTrue(student.getCourses().isEmpty());
        assertEquals(0, student.getCourses().size());
    }

    // ============ getId() Tests ============

    @Test
    @DisplayName("getId: Should return the student ID correctly")
    void testGetId() {
        assertEquals("S001", student.getId());
    }
```

```
    @Test
    @DisplayName("getId: Should return default '0000' for null input")
    void testGetIdDefault() {
        Student testStudent = new Student(null, "hadi");
        assertEquals("0000", testStudent.getId());
    }

    // ============ getName() Tests ============

    @Test
    @DisplayName("getName: Should return the student name correctly")
    void testGetName() {
        assertEquals("mohammed", student.getName());
    }

    @Test
    @DisplayName("getName: Should return 'Unknown' for null input")
    void testGetNameDefault() {
        Student testStudent = new Student("S001", null);
        assertEquals("Unknown", testStudent.getName());
    }

    // ============ enrollCourse() Tests ============

    @Test
    @DisplayName("enrollCourse: Should add a valid course to the list")
    void testEnrollCourseValid() {
        student.enrollCourse(courseA);
        assertEquals(1, student.getCourses().size());
        assertEquals(courseA, student.getCourses().get(0));
    }

    @Test
    @DisplayName("enrollCourse: Should add multiple courses to the list")
    void testEnrollMultipleCourses() {
        student.enrollCourse(courseA);
        student.enrollCourse(courseB);
        student.enrollCourse(courseF);
        assertEquals(3, student.getCourses().size());
    }

    @Test
    @DisplayName("enrollCourse: Should not add null course")
    void testEnrollCourseNull() {
        student.enrollCourse(null);
        assertEquals(0, student.getCourses().size());
    }

    @Test
    @DisplayName("enrollCourse: Should add duplicate courses (same course
multiple times)")
    void testEnrollDuplicateCourse() {
        student.enrollCourse(courseA);
        student.enrollCourse(courseA);
        assertEquals(2, student.getCourses().size());
    }
```

```
    // ============ getCourses() Tests ============

    @Test
    @DisplayName("getCourses: Should return empty list when no courses
enrolled")
    void testGetCoursesEmpty() {
        assertTrue(student.getCourses().isEmpty());
    }

    @Test
    @DisplayName("getCourses: Should return all enrolled courses")
    void testGetCoursesMultiple() {
        student.enrollCourse(courseA);
        student.enrollCourse(courseB);

        List courses = student.getCourses();
        assertEquals(2, courses.size());
        assertTrue(courses.contains(courseA));
        assertTrue(courses.contains(courseB));
    }

    @Test
    @DisplayName("getCourses: Should return courses in enrollment order")
    void testGetCoursesOrder() {
        student.enrollCourse(courseA);
        student.enrollCourse(courseB);

        List courses = student.getCourses();
        assertEquals(courseA, courses.get(0));
        assertEquals(courseB, courses.get(1));
    }

    // ============ calculateGPA() Tests ============

    @Test
    @DisplayName("calculateGPA: Should return 0.0 when no courses enrolled")
    void testCalculateGPANoCourses() {
        assertEquals(0.0, student.calculateGPA());
    }

    @Test
    @DisplayName("calculateGPA: Should calculate GPA correctly for single
course with grade A")
    void testCalculateGPASingleCourseA() {
        student.enrollCourse(courseA);
        // GPA = (4.0 * 3) / 3 = 4.0
        assertEquals(4.0, student.calculateGPA());
    }

    @Test
    @DisplayName("calculateGPA: Should calculate GPA correctly for single
course with grade B")
    void testCalculateGPASingleCourseB() {
        student.enrollCourse(courseB);
        // GPA = (3.0 * 4) / 4 = 3.0
        assertEquals(3.0, student.calculateGPA());
```

```
    }

    @Test
    @DisplayName("calculateGPA: Should calculate GPA correctly for multiple
courses with different grades")
    void testCalculateGPAMultipleCourses() {
        // courseA: 4.0 * 3 credits = 12.0
        // courseB: 3.0 * 4 credits = 12.0
        // Total: 24.0 / 7 credits = 3.43 (approx)
        student.enrollCourse(courseA);
        student.enrollCourse(courseB);

        double expectedGPA = (4.0 * 3 + 3.0 * 4) / 7.0;
        assertEquals(expectedGPA, student.calculateGPA(), 0.0001);
    }

    @Test
    @DisplayName("calculateGPA: Should calculate GPA with F grade correctly")
    void testCalculateGPAWithFGrade() {
        // courseF: 0.0 * 2 credits = 0.0
        // courseA: 4.0 * 3 credits = 12.0
        // Total: 12.0 / 5 credits = 2.4
        student.enrollCourse(courseF);
        student.enrollCourse(courseA);

        double expectedGPA = (0.0 * 2 + 4.0 * 3) / 5.0;
        assertEquals(expectedGPA, student.calculateGPA(), 0.0001);
    }

    @Test
    @DisplayName("calculateGPA: Should cap GPA at 4.0 maximum")
    void testCalculateGPACapped() {
        // Even with all A grades, GPA should not exceed 4.0
        Course courseA2 = new Course("Physics", 3, "A");
        student.enrollCourse(courseA);
        student.enrollCourse(courseA2);

        double gpa = student.calculateGPA();
        assertTrue(gpa <= 4.0);
        assertEquals(4.0, gpa);
    }

    @Test
    @DisplayName("calculateGPA: Should handle course with invalid grade
(defaults to F)")
    void testCalculateGPAInvalidGrade() {
        Course invalidCourse = new Course("History", 3, "InvalidGrade");
        student.enrollCourse(invalidCourse);

        // Invalid grade defaults to F (0.0 points)
        assertEquals(0.0, student.calculateGPA());
    }

    @Test
    @DisplayName("calculateGPA: Should calculate correctly with varying
credit hours")
    void testCalculateGPAVaryingCredits() {
```

```
        Course course1 = new Course("Course1", 1, "A");  // 4.0 * 1 = 4.0
        Course course2 = new Course("Course2", 5, "F");  // 0.0 * 5 = 0.0

        student.enrollCourse(course1);
        student.enrollCourse(course2);

        // GPA = 4.0 / 6 = 0.667
        double expectedGPA = 4.0 / 6.0;
        assertEquals(expectedGPA, student.calculateGPA(), 0.0001);
    }

    @Test
    @DisplayName("calculateGPA: Should handle all F grades")
    void testCalculateGPAAllF() {
        Course f1 = new Course("Course1", 3, "F");
        Course f2 = new Course("Course2", 4, "F");

        student.enrollCourse(f1);
        student.enrollCourse(f2);

        assertEquals(0.0, student.calculateGPA());
    }
}
```