Student Grade Management System

Project Report

```
┌──────────────────────────────────────────────┐
│                                              │
│        STUDENT GRADE MANAGEMENT SYSTEM       │
│                                              │
│            Academic Project Report           │
│                                              │
│                                              │
│             Submitted by: Pakhi Sood         │
│             Reg Number: 24BCE10403           │
│                 Course: B.Tech               │
│          Subject: Programming in Java        │
│               Session: 2025 - 2026           │
│                                              │
│                Date: 24/11/2025              │
│                                              │
│                                              │
│        VITyarthi - Build Your Own Project    │
│                                              │
└──────────────────────────────────────────────┘
```

---

2. INTRODUCTION

2.1 Project Overview

The Student Grade Management System is a console-based Java application designed to help teachers and educational institutions cliently manage student information and academic grades. The system provides a centralized platform for recording grades, calculating averages, and generating comprehensive reports.

2.2 Purpose

This project was developed as part of the course evaluation to demonstrate practical application of object-oriented programming principles, data structure management, file I/O operations, and software design patterns.

2.3 Scope

The application focuses on core grade management functionality including student registration, grade recording across multiple subjects, statistical analysis, and data persistence through file storage.

---

3. PROBLEM STATEMENT

3.1 Current Challenges

Educational institutions face several challenges in managing student grades:

- Manual record-keeping is time-consuming and error-prone
- Di culty in quickly accessing student performance data
- Lack of automated calculation for averages and statistics
- No centralized system for maintaining historical grade records
- Ine cient process for generating student reports

3.2 Impact

These challenges lead to:

- Increased administrative workload for teachers
- Delayed feedback to students
- Higher probability of calculation errors
- Di culty in identifying performance trends
- Poor data organization and accessibility

3.3 Proposed Solution

A digital system that automates grade recording, calculations, and report generation while maintaining data persistence across sessions.

---

4. FUNCTIONAL REQUIREMENTS

4.1 Student Management Module

FR-1: Add Student

- System shall allow adding new students with unique ID, name, and email
- System shall validate student ID for uniqueness
- System shall validate email format

FR-2: View Students

- System shall display list of all registered students
- System shall show student ID, name, email, and grade count

FR-3: Search Student

- System shall allow searching students by ID
- System shall provide error message if student not found

4.2 Grade Recording Module

FR-4: Add Grade

- System shall allow recording grades for specific students
- System shall support multiple subjects per student
- System shall validate grade values (0-100 range)

FR-5: Grade Calculation

- System shall automatically calculate letter grades (A, B, C, D, F)
- System shall calculate individual student averages
- System shall support multiple grades per subject

4.3 Report Generation Module

FR-6: Student Report

- System shall generate individual student reports
- Report shall include all recorded grades with letter grades
- Report shall display calculated average

FR-7: Class Statistics

- System shall generate class-wide statistics
- System shall calculate class average
- System shall identify highest and lowest performing students

- System shall count total students and students with grades

4.4 Data Persistence Module

FR-8: Save Data

- System shall save all data to text file on exit
- System shall create data directory if not exists
- System shall maintain data integrity

FR-9: Load Data

- System shall load existing data on startup
- System shall handle missing file gracefully
- System shall restore complete student and grade records

---

5. NON-FUNCTIONAL REQUIREMENTS

5.1 Performance

NFR-1: The system shall respond to all user operations within 1 second for datasets up to 100 students with 1000 total grades.

5.2 Usability

NFR-2: The system shall provide a simple menu-driven interface requiring no prior training. All operations shall be accessible through numbered menu options with clear prompts.

5.3 Reliability

NFR-3: The system shall persist all data across sessions with 100% accuracy. File corruption shall be prevented through proper exception handling and atomic write operations.

5.4 Maintainability

NFR-4: The system shall follow modular design with clear separation of concerns (models, services, utilities). Code shall be documented with comments explaining complex logic.

5.5 Security

NFR-5: The system shall validate all user inputs to prevent invalid data entry. Input validation shall cover ID format, name length, email format, and grade ranges.

5.6 Portability

NFR-6: The system shall run on any platform supporting Java 11 or higher (Windows, macOS, Linux) without modification.

---

6. SYSTEM ARCHITECTURE

6.1 Architecture Overview

The Student Grade Management System follows a layered architecture pattern with four distinct layers:

Presentation Layer (Main.java)

- Handles user interaction and console I/O
- Displays menu and processes user choices
- Validates inputs and displays results

Service Layer

- StudentService: Manages student operations (CRUD)
- GradeService: Handles grade-related operations
- ReportService: Generates reports and statistics

Model Layer

- Student: Entity representing student data
- Grade: Entity representing individual grade records

Utility Layer

- FileHandler: Manages data persistence (save/load)
- Validator: Provides input validation methods

6.2 Architecture Diagram

6.3 Design Rationale

- **Layered Architecture:** Chosen for clear separation of concerns and maintainability
- **Service Pattern:** Encapsulates business logic away from presentation

- **File-based Storage:** Simple, lightweight solution suitable for small-scale deployment
- **Console Interface:** Quick development, no external dependencies

---

7. DESIGN DIAGRAMS

7.1 Use Case Diagram

Description:
This diagram illustrates the interaction between the Teacher (primary actor) and the system. The teacher can perform five main use cases:

1. Add Student - Register new students in the system
2. View Students - Display list of all students
3. Add Grade - Record grades for students

N. Generate Report - View individual student performance

U. View Statistics - Analyze class-wide performance

All use cases are contained within the system boundary and are directly accessible to the teacher without any complex relationships or dependencies.

7.2 Class Diagram

Description:
The class diagram depicts seven classes organized in a modular structure:

Model Classes:

- **Student:** Contains studentId, name, email, and a list of grades. Provides methods for calculating average and managing grade collection.
- **Grade:** Stores subject, score, and studentId. Includes getLetterGrade() method for automatic grade conversion.

Service Classes:

- **StudentService:** Manages student collection using HashMap. Handles CRUD operations and data persistence coordination.
- **GradeService:** Provides grade addition and average calculation functionality.
- **ReportService:** Generates formatted reports and statistical summaries.

Utility Classes:

- **FileHandler:** Implements save/load operations using text file format.

- **Validator:** Static utility methods for input validation.

Relationships:

- Student HAS-MANY Grades (1:* composition)
- Services USE model classes (dependency)
- StudentService HAS-A FileHandler (composition)

7.3 Sequence Diagram

Description:
This sequence diagram illustrates the "Add Grade" operation flow:

1. Teacher selects option 3 from menu
2. Main prompts for student ID, subject, and score
3. Main calls GradeService.addGrade()
N. GradeService requests student object from StudentService
U. StudentService returns Student object
6. GradeService creates new Grade object
7. GradeService adds grade to Student's grade list
8. Success status propagates back to Main
9. Main displays success message to Teacher

This interaction demonstrates the collaboration between presentation, service, and model layers to complete a business operation.

7.4 System Architecture Diagram

Description:
The architecture diagram shows five layers stacked vertically with clear separation:

Layer 1 - Presentation: Main.java handles all console interaction
Layer 2 - Service: Three service classes handle business logic
Layer 3 - Model: Student and Grade entities represent domain objects
Layer 4 - Utility: FileHandler and Validator provide supporting functions
Layer 5 - Data Storage: students.txt file persists data

Data flows from top to bottom during write operations and bottom to top during read operations. Each layer communicates only with adjacent layers, maintaining loose coupling.

7.5 ER Diagram

Description:
The ER diagram shows the data model with two entities:

Student Entity:

- studentId (Primary Key) - Unique identifier
- name - Student's full name
- email - Contact email address

Grade Entity:

- studentId (Foreign Key) - Links to Student entity
- subject - Subject name (e.g., Maths, Java)
- score - Numeric grade value (0-100)

Relationship: One-to-Many

- One student can have multiple grades (0..*)
- Each grade belongs to exactly one student (1)

This simple schema effectively captures the relationship between students and their academic performance across multiple subjects.

7.6 Process Flow Diagram

Description:
The flowchart depicts the application's main execution flow:

1. **Start:** Application launches
2. **Load Data:** Existing data loaded from file
3. **Display Menu:** Six options presented to user
N. **Get Choice:** User input captured
U. **Decision Branch:** Flow splits based on choice (1-6)
6. **Process Operation:** Specific function executes (Add Student, View, Add Grade, etc.)
7. **Display Result:** Success/error message shown
8. **Exit Check:** If choice is 6, save and exit; else loop back to menu

The diagram clearly shows the menu-driven loop structure and the conditional logic that routes execution to different functional modules.

---

8. DESIGN DECISIONS & RATIONALE

8.1 Technology Stack Decisions

Choice: Java 11

- **Reason:** Platform independence, strong OOP support, rich standard library
- **Alternative Considered:** Python - rejected due to course requirements
- **Benefit:** Compiled language ensures type safety and performance

Choice: File-based Storage (Text Files)

- **Reason:** Simple, lightweight, no external dependencies
- **Alternative Considered:** Database (MySQL) - rejected due to deployment complexity
- **Benefit:** Easy backup, human-readable format, no setup required
- **Trade-off:** Limited scalability, no concurrent access support

Choice: Console Interface

- **Reason:** Quick development, focus on business logic
- **Alternative Considered:** GUI (JavaFX) - deferred to future enhancement
- **Benefit:** Cross-platform without additional libraries

8.2 Architectural Decisions

Choice: Layered Architecture

- **Reason:** Clear separation of concerns, maintainability
- **Benefit:** Each layer has single responsibility, easy to test
- **Trade-off:** Slight performance overhead from layer boundaries

Choice: Service Pattern

- **Reason:** Encapsulates business logic, reusability
- **Benefit:** Services can be unit tested independently
- **Example:** StudentService manages all student operations in one place

Choice: HashMap for Student Storage

- **Reason:** O(1) lookup performance by student ID
- **Alternative Considered:** ArrayList - rejected due to search ine ciency
- **Benefit:** Fast retrieval for report generation

8.3 Data Structure Decisions

Choice: List in Student

- **Reason:** Preserves order of grade entry, allows duplicates
- **Benefit:** Can track grade history if subject is repeated
- **Trade-off:** Manual iteration for calculations

Choice: Custom toString() Methods

- **Reason:** Clean formatted output for console display
- **Benefit:** Consistent formatting across application

8.4 Design Patterns Used

Pattern: Service Layer Pattern

- Separates business logic from presentation
- Used in StudentService, GradeService, ReportService

Pattern: Static Utility Pattern

- Validator class provides reusable validation methods
- No state, pure functions

Pattern: Data Access Object (DAO)

- FileHandler encapsulates all file operations
- Can be replaced with database implementation later

---

9. IMPLEMENTATION DETAILS

9.1 Technology Stack

- **Language:** Java 11
- **IDE:** Visual Studio Code / IntelliJ IDEA / Eclipse
- **Build Tool:** Manual compilation using javac
- **Version Control:** Git & GitHub
- **Data Storage:** Text files (.txt)
- **Testing:** Manual testing via console

9.2 Key Implementation Highlights

File I/O Implementation:

```
// Custom format for easy parsing
STUDENT:101,Pakhi,pakhi@gmail.com
GRADE:Maths,92.0
GRADE:Java,96.0
END
```

This format allows sequential reading/writing while maintaining data relationships.

Average Calculation Algorithm:

```
public double calculateAverage() {
    if (grades.isEmpty()) return 0.0;
    double sum = 0;
    for (Grade grade : grades) {
        sum += grade.getScore();
    }
    return sum / grades.size();
}
```

Input Validation Strategy:

- ID: Max 20 characters, non-empty
- Name: Minimum 2 characters
- Grade: Range 0-100
- Email: Must contain '@' symbol

Error Handling Approach:

- Try-catch blocks for file operations
- Validation before processing
- Graceful degradation (start fresh if file missing)
- Clear error messages to user

9.3 Module-wise Implementation

Main.java (150 lines)

- Menu loop with switch-case
- Input capture and validation
- Coordination between services
- Scanner management for different input types

StudentService.java (50 lines)

- HashMap for O(1) student lookup
- Delegation to FileHandler for persistence
- Collection view for iteration

GradeService.java (20 lines)

- Simple delegation to Student model
- Validation before grade addition

ReportService.java (65 lines)

- Formatted output using printf
- Statistical calculations (min, max, average)
- Null-safe operations

FileHandler.java (55 lines)

- Custom text format parsing
- Directory creation if missing
- Resource management with try-with-resources

Validator.java (15 lines)

- Static methods for reusability
- Boolean return for easy condition checking

Student.java (60 lines)

- Encapsulation with private fields
- Business logic (calculateAverage)
- Formatted toString() for display

Grade.java (40 lines)

- Letter grade conversion logic
- Immutable after creation (no setters used in practice)

9.4 Data Flow Example

Adding a New Student with Grade:

1. User selects "Add Student" → Main validates input → StudentService creates Student → HashMap stores reference
2. User selects "Add Grade" → Main captures input → GradeService retrieves Student → Creates Grade → Student adds to list

3. User selects "Save and Exit" → StudentService calls FileHandler → Iterates students and grades → Writes to file

---

10. SCREENSHOTS / RESULTS

10.1 Main Menu

D:\java\VITyarthi

Description: The main interface presents six options to the user. The menu is displayed after loading existing data from the file system.

10.2 Adding Students

Description: Successfully added three students (Ekjot, Yash, Anshuman) with their IDs and email addresses. The system validates each input and confirms successful addition.

10.3 Viewing All Students

Description: Displays all registered students in a formatted list showing ID, name, email, and number of grades recorded.

10.4 Adding Grades

Description: Adding grades for student 101 in subjects Maths (92.0) and Java (96.0). The system validates the score range and confirms successful addition.

10.5 Student Report

Description: Individual report for student Pakhi (ID: 101) showing:

- Student details (ID, name, email)
- All grades with letter grades (Maths: 92.00 (A), Java: 96.00 (A))
- Calculated average: 94.00

10.6 Class Statistics

Description: Class-wide statistics showing:

- Total Students: 5
- Students with Grades: 5
- Class Average: 87.20
- Highest Average: 94.00 (Pakhi)
- Lowest Average: 80.00

This demonstrates the system's ability to aggregate data and provide meaningful insights.

10.7 Data Persistence

```
STUDENT:101,Pakhi,pakhisood@gmail.com
GRADE:Maths,92.0
GRADE:Java,96.0
END
STUDENT:102,Ekjot,ekjotsingh@gmail.com
GRADE:Maths,90.0
GRADE:java,95.0
END
...
```

Description: The text file format showing how data is persisted. Each student's information is clearly delimited with grades nested under the student record.

---

11. TESTING APPROACH

11.1 Testing Strategy

Manual testing was conducted through the console interface covering:

- **Functional Testing:** Verify all features work as expected
- **Validation Testing:** Test input validation rules
- **Boundary Testing:** Test edge cases and limits
- **Data Persistence Testing:** Verify save/load functionality

11.2 Test Cases Executed

Test Case 1: Add Valid Student

- **Input:** ID="101", Name="Pakhi", Email="pakhi@gmail.com"

- **Expected:** Success message displayed
- **Actual:** ✓ PASS - Student added successfully
- **Evidence:** Screenshot 02

Test Case 2: Add Duplicate Student

- **Input:** ID="101" (already exists), Name="Test", Email="test@test.com"
- **Expected:** Error message "Student with this ID already exists"
- **Actual:** ✓ PASS - Correct error message displayed

Test Case 3: Invalid ID (Empty)

- **Input:** ID="", Name="Test", Email="test@test.com"
- **Expected:** Error message "Invalid ID format"
- **Actual:** ✓ PASS - Validation prevented empty ID

Test Case 4: Invalid Name (Too Short)

- **Input:** ID="999", Name="A", Email="test@test.com"
- **Expected:** Error message "Invalid name"
- **Actual:** ✓ PASS - Validation enforced minimum 2 characters

Test Case 5: Add Valid Grade

- **Input:** StudentID="101", Subject="Maths", Score=92.0
- **Expected:** Success message, grade added to student
- **Actual:** ✓ PASS - Grade added successfully
- **Evidence:** Screenshot 04

Test Case 6: Invalid Grade (Out of Range)

- **Input:** StudentID="101", Subject="Maths", Score=150
- **Expected:** Error message "Grade must be between 0 and 100"
- **Actual:** ✓ PASS - Validation prevented invalid grade

Test Case 7: Add Grade to Non-existent Student

- **Input:** StudentID="999" (doesn't exist), Subject="Maths", Score=85
- **Expected:** Error message "Student not found"
- **Actual:** ✓ PASS - Correct error handling

Test Case 8: Calculate Average (Single Grade)

- **Input:** Student with 1 grade: Maths=88.0
- **Expected:** Average = 88.0
- **Actual:** ✓ PASS - Correct calculation

Test Case 9: Calculate Average (Multiple Grades)

- **Input:** Student with grades: Maths=92.0, Java=96.0
- **Expected:** Average = 94.0
- **Actual:** ✓ PASS - Correct calculation
- **Evidence:** Screenshot 05

Test Case 10: Class Statistics

- **Input:** 5 students with various grades
- **Expected:** Correct totals, averages, highest, lowest
- **Actual:** ✓ PASS - All calculations accurate
- **Evidence:** Screenshot 06

Test Case 11: Save Data

- **Input:** Exit application (Option 6)
- **Expected:** Data written to data/students.txt
- **Actual:** ✓ PASS - File created with correct content

Test Case 12: Load Data

- **Input:** Restart application
- **Expected:** All students and grades loaded
- **Actual:** ✓ PASS - "Data loaded successfully. Total students: 5"

Test Case 13: Empty System Behavior

- **Input:** View students when none exist
- **Expected:** Message "No students found"
- **Actual:** ✓ PASS - Graceful handling

Test Case 14: Generate Report (No Grades)

- **Input:** Student with no grades
- **Expected:** Message "No grades recorded"
- **Actual:** ✓ PASS - Handled correctly

11.3 Test Results Summary

- **Total Test Cases:** 14
- **Passed:** 14
- **Failed:** 0
- **Pass Rate:** 100%

11.4 Boundary Value Testing

- Grade = 0.0 ✓ (minimum)
- Grade = 100.0 ✓ (maximum)
- Grade = -1.0 ✗ (rejected by validation)
- Grade = 101.0 ✗ (rejected by validation)
- Student ID length = 20 characters ✓ (maximum)
- Student ID length = 21 characters ✗ (rejected)

11.5 Testing Limitations

- No automated unit tests implemented
- No concurrent access testing (file locking)
- No performance testing with large datasets (1000+ students)
- No security testing (file permission handling)

---

12. CHALLENGES FACED

12.1 File I/O Complexity

Challenge: Designing a text file format that could represent the one-to-many relationship between students and grades.

Solution: Created a custom format with delimiters (STUDENT:, GRADE:, END) that allows sequential parsing while maintaining data relationships.

Learning: Understanding the importance of structured data formats and parsing logic.

12.2 Scanner Input Handling

Challenge: Scanner's nextInt() leaves a newline character in the buffer, causing subsequent nextLine() calls to return empty strings.

Solution: Added scanner.nextLine() after each nextInt() and nextDouble() to consume the leftover newline.

Learning: Importance of understanding how input methods interact with the input buffer.

12.3 Grade Average Calculation Edge Case

Challenge: Division by zero error when calculating average for student with no grades.

Solution: Added isEmpty() check before calculation and return 0.0 as default.

Learning: Always handle edge cases and empty collections.

12.4 Data Persistence State Management

Challenge: Deciding when to save data - after every operation (slow) or only on exit (risk of data loss).

Solution: Chose to save only on explicit exit (Option 6) with clear messaging to user.

Learning: Trade-offs between performance and data safety.

12.5 Menu Loop Exit Condition

Challenge: Ensuring clean exit with proper resource cleanup (Scanner, file handles).

Solution: Used boolean flag in while loop and explicit scanner.close() before exit.

Learning: Resource management and proper cleanup in Java.

---

13. LEARNINGS & KEY TAKEAWAYS

13.1 Technical Skills Developed

1. **Object-Oriented Design:** Applied encapsulation, inheritance (through toString override), and composition (Student HAS-A List of Grades)
2. **File I/O Operations:** Learned to handle file creation, reading, writing, and error handling in Java
3. **Data Structures:** Practical use of HashMap (fast lookup), ArrayList (ordered collection), and custom object storage

N. **Exception Handling:** Implemented try-catch blocks for robust error management

U. **Input Validation:** Developed validation logic to ensure data integrity

13.2 Software Engineering Principles

1. **Separation of Concerns:** Understood the value of layered architecture in maintaining clean, modular code
2. **Single Responsibility:** Each class has one clear purpose (Student models data, StudentService manages operations)

3. **DRY (Don't Repeat Yourself):** Created reusable Validator methods instead of repeating validation logic

N. **Code Readability:** Importance of meaningful variable names, comments, and formatting

13.3 Problem-Solving Approach

1. Breaking down complex requirements into manageable modules
2. Iterative development (built model classes first, then services, then UI)
3. Testing after each feature implementation
N. Debugging systematic errors (package structure, Scanner issues)

13.4 Project Management

1. Version control usage (Git commits for each major feature)
2. Documentation alongside development (README, statement.md)
3. Time management across design, implementation, testing, documentation phases

13.5 Key Insights

- **Design Before Code:** Spending time on architecture design saved refactoring time later
- **Test Early and Often:** Finding bugs during development is easier than after completion
- **User Experience Matters:** Even console apps benefit from clear menus and error messages
- **Data Integrity is Critical:** Input validation prevents many downstream issues

14. REFERENCES

14.1 Technical Documentation

1. Oracle Java Documentation - Java SE 11
   https://docs.oracle.com/en/java/javase/11/

2. Java Collections Framework Guide
   https://docs.oracle.com/javase/tutorial/collections/

3. Java I/O Streams Tutorial
   https://docs.oracle.com/javase/tutorial/essential/io/

14.2 Design Patterns and Best Practices

N. "Design Patterns: Elements of Reusable Object-Oriented Software"
Gang of Four (GoF)

U. "Effective Java" by Joshua Bloch
Best practices for Java development

6. Layered Architecture Pattern
https://en.wikipedia.org/wiki/Multitier_architecture

14.3 Software Engineering Resources

7. Clean Code: A Handbook of Agile Software Craftsmanship
Robert C. Martin

8. UML Diagram Standards
https://www.uml.org/

14.4 Tools and Technologies

9. Git Documentation
https://git-scm.com/doc

10. GitHub Guides
https://guides.github.com/

11. draw.io (diagrams.net) for UML diagrams
https://app.diagrams.net/

14.5 Course Materials

12. VITyarthi Course Lecture Notes
[Subject Name] - [Instructor Name]

13. Lab Manuals and Assignment Guidelines
Provided by course instructor

14.6 Online Resources

1N. Stack Overflow - Programming Q&A
https://stackoverflow.com/

1U. GeeksforGeeks - Java Programming
https://www.geeksforgeeks.org/java/
