REPORT

INTERIM SEMESTER 2025 - 26

VIT®
BHOPAL
www.vitbhopal.ac.in

CSA2006 – Programming in Java

Slot - A11+A12+A13+D11+D12

Submitted to :

Dr. Garima Jain

Submitted by :

Shamanthula Hasini
24MIM10064

School of Computing Science Engineering and Artificial Intelligence
Academic Year: 2025 – 2026

1. Introduction

Educational institutions need to maintain large amounts of student data such as personal details, academic records, and contact information. Manual record-keeping or simple spreadsheets lead to duplication, data loss, slow retrieval, and human errors.

This project implements a **Console-Based Student Management System using Core Java** that demonstrates proper Object-Oriented Programming (OOP) principles, layered architecture, modular design, input validation, and clean code practices. The system supports full CRUD (Create, Read, Update, Delete) operations on student records stored in memory using ArrayList.

The primary goal is to apply OOP concepts, design patterns (Repository + Service pattern), separation of concerns, and produce well-documented, maintainable code.

2. Problem Statement

Traditional student record management suffers from the following issues:

- Difficulty in quickly locating a specific student's record
- High chances of duplicate entries and human errors
- Time-consuming updates and deletions
- Lack of structured storage and search capability
- No validation of critical fields (age, email, etc.)

An automated, reliable, and user-friendly system is required to overcome these challenges.

3. Objectives

- Develop a fully functional console-based Student Management System
- Implement CRUD operations using Java
- Apply OOP principles (encapsulation, abstraction, single responsibility)
- Design a clean layered architecture (Model → Repository → Service → UI)
- Perform robust input validation and error handling
- Provide complete documentation with UML diagrams and design rationale

- Use Git for version control and host the project on GitHub

4. Functional Requirements

The system provides **three major functional modules**:

Module	Description
1. Student Management	Add new student with auto-generated unique ID
2. View & Search	List all students / Search student by ID
3. Update & Delete	Modify existing student details / Remove student record

Detailed Features

1. **Add Student** – Accept name, age, grade, email; auto-generate ID
2. **View All Students** – Display complete list in readable format
3. **Search Student** – Find student by ID and show details
4. **Update Student** – Modify any field (others remain unchanged if left blank)
5. **Delete Student** – Remove student using ID

5. Non-Functional Requirements

Requirement	Description
Usability	Simple numbered menu; no training required
Performance	All operations execute in O(1) or O(n) with in-memory ArrayList
Reliability	Comprehensive input validation; graceful handling of invalid data
Maintainability	Clear layered architecture and single-responsibility classes

Error Handling	User-friendly messages instead of crashes
Scalability	Code structure allows easy migration to database or GUI in future

6.System Architecture

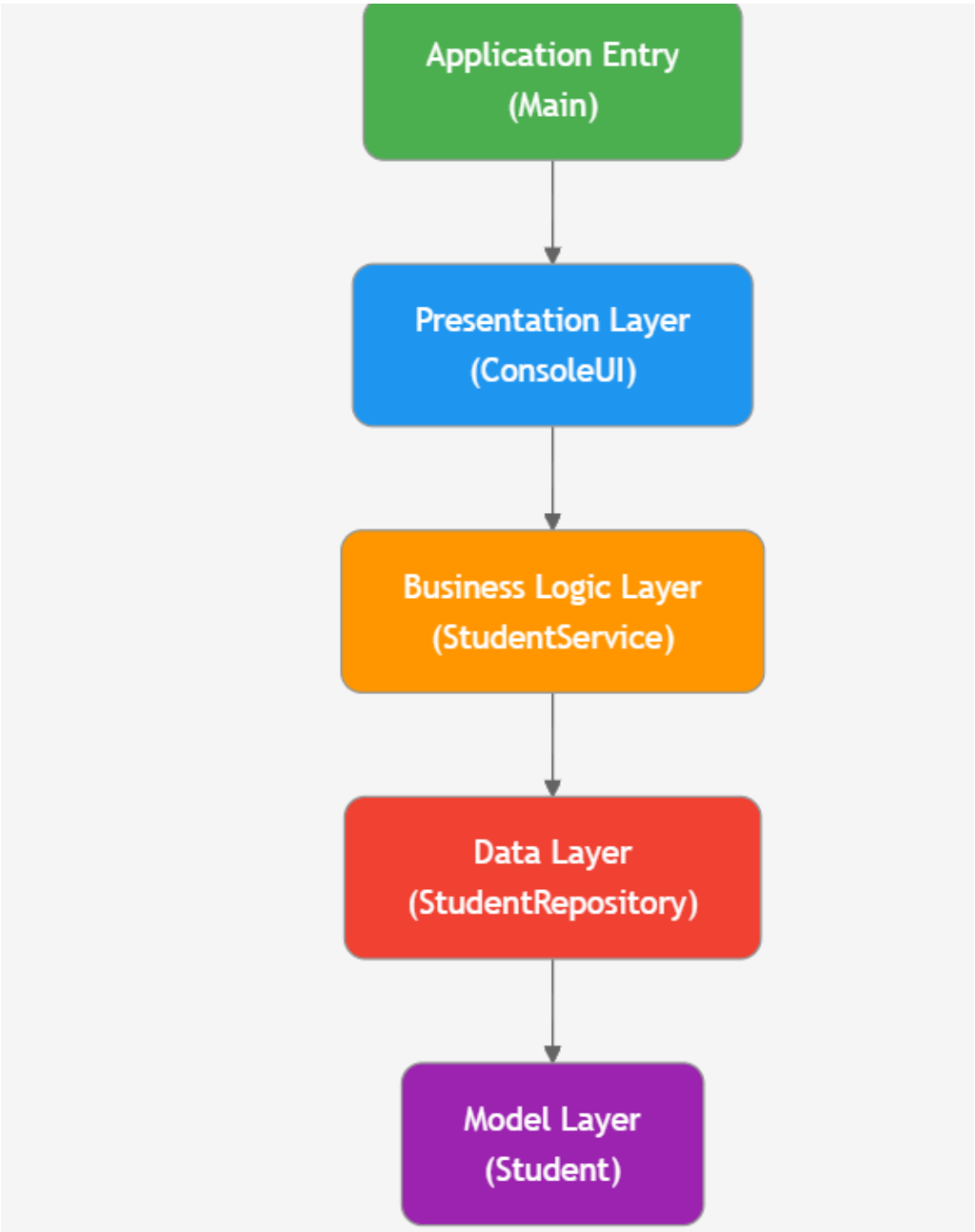

Fig 6.1 Architecture Diagram

The system follows a clean **four-layer architecture** that separates concerns and makes the code professional, maintainable, and extensible.

1. **Presentation Layer (ConsoleUI)** This is the user interface. It displays the menu, reads input from the keyboard, shows results and messages, and forwards user requests to the next layer. It contains no business rules or data storage logic.
2. **Business Logic Layer (StudentService)** This layer acts as the brain of the application. It validates all inputs (name not blank, age > 0, valid email format), enforces business rules, and decides what operations are allowed. It never stores data itself; it only coordinates with the layer below.
3. **Data Access Layer (StudentRepository)** This layer is responsible for actual storage and retrieval. It maintains an in-memory ArrayList<Student>, automatically generates unique student IDs using an auto-increment counter, and provides methods to add, view, search, update, and delete records.
4. **Model Layer (Student)** A simple POJO that represents one student record with the fields id, name, age, grade, and email, along with getters, setters, and a proper toString() method.

Data Flow

ConsoleUI → StudentService → StudentRepository → Student (and back with results).

This strict separation ensures that each class has only one responsibility, making the system easy to test, debug, and extend (e.g., replacing the repository with a database or adding a GUI later would require changes in only one layer). The design follows standard industry practices and demonstrates the application of clean architecture principles in a console-based Java application.

7.Design Diagrams

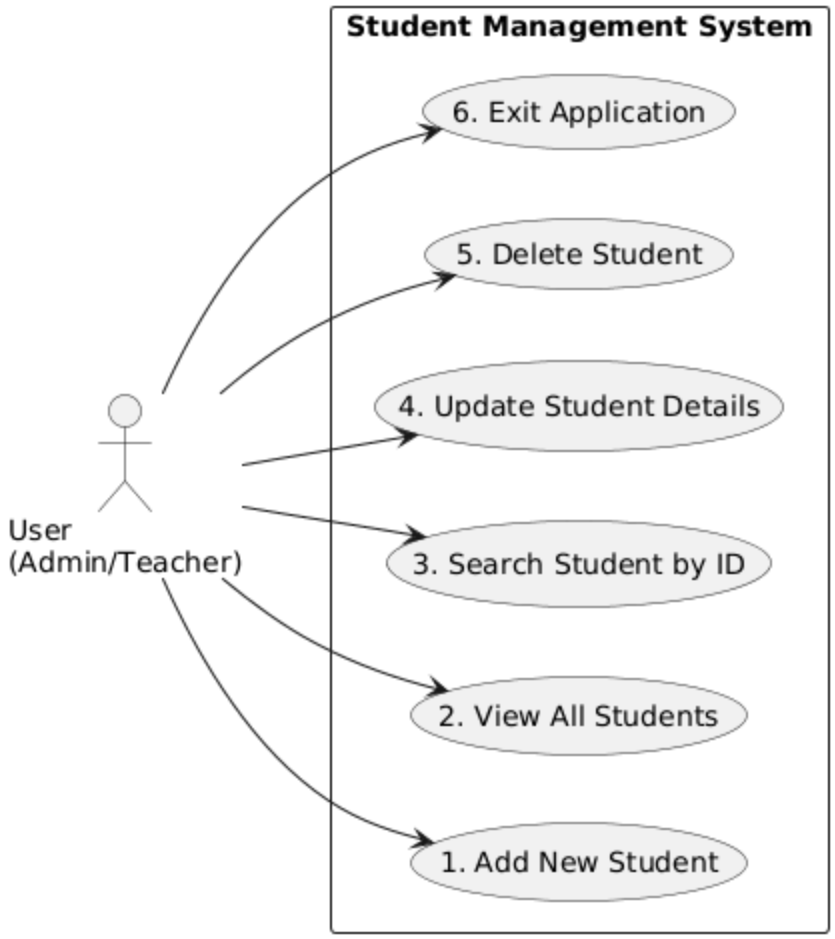

Fig 7.1 Use Case Diagram

The Use Case Diagram captures the functional requirements from the user's perspective. The only actor is the **User** (school admin, teacher, or staff member).

The system provides six primary use cases:

- **Add Student**: Create a new student record with auto-generated ID
- **View All Students**: Display the complete list of registered students
- **Search Student by ID**: Retrieve and display details of a specific student
- **Update Student**: Modify one or more fields of an existing student (partial update allowed)
- **Delete Student**: Permanently remove a student record using the ID
- **Exit**: Close the application

All interactions are initiated by the User through the console menu. This diagram clearly shows that the system offers complete CRUD functionality plus a graceful exit option.

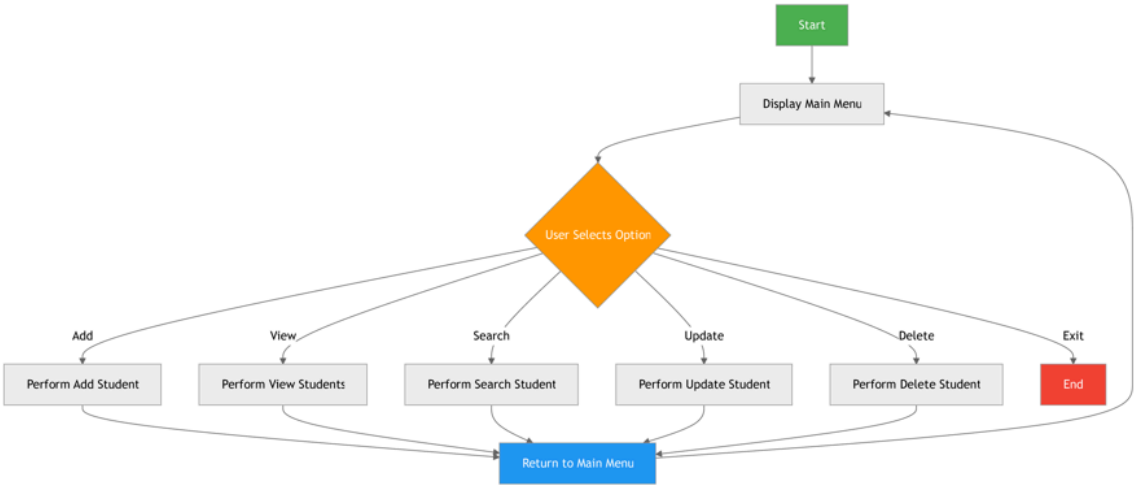

Fig 7.2 Workflow Diagram

The workflow diagram illustrates the overall flow of control inside the application.

It is a continuous loop that represents the main menu cycle:

1. The system displays the numbered menu.
2. The user enters a choice (1–6).
3. Depending on the choice, the corresponding operation (Add, View, Search, Update, Delete, or Exit) is executed.
4. After each operation finishes (except Exit), the program returns to the menu.
5. If an invalid option is entered, an error message is shown and the menu is displayed again.
6. When the user selects Exit, the loop terminates and the application ends.

This diagram clearly depicts the simple, robust, and infinite menu-driven workflow of the console application.

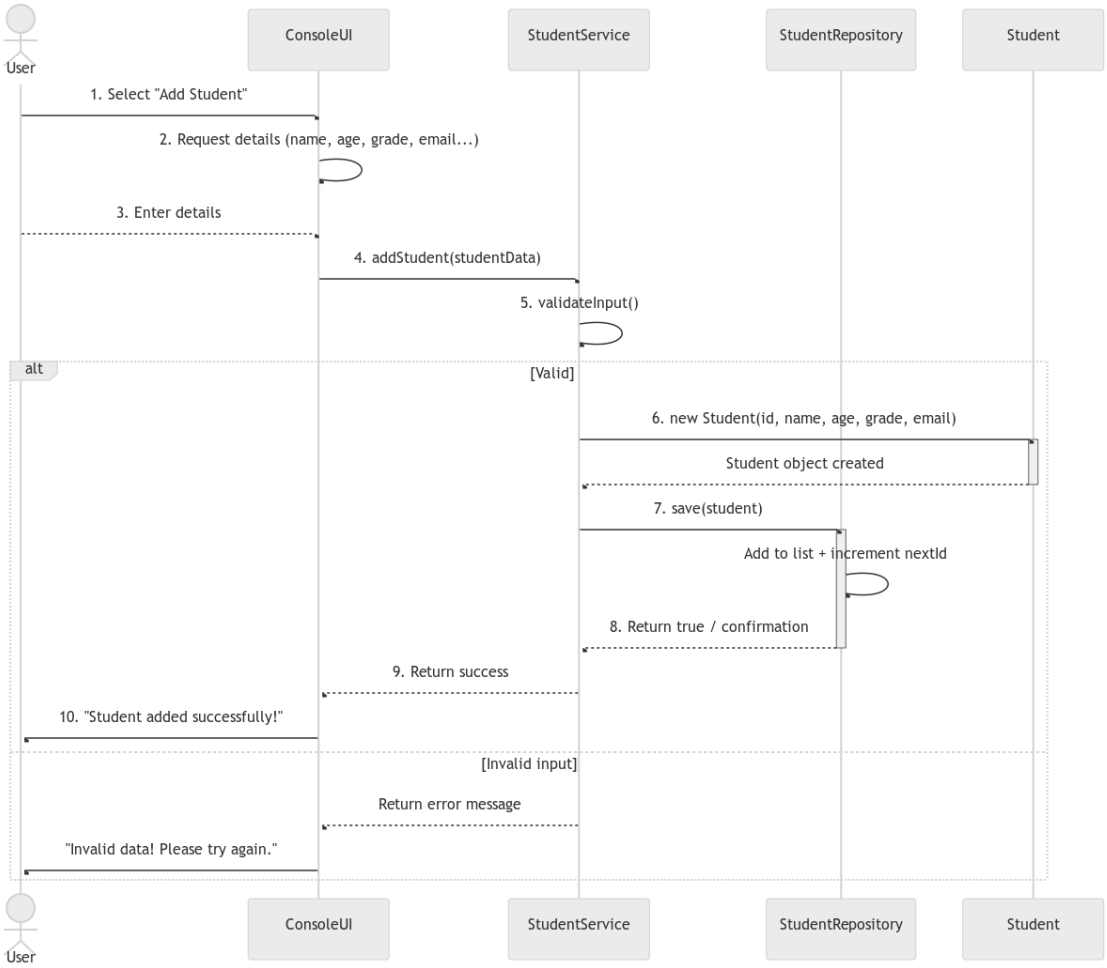

Fig 7.3 Sequence Diagram (Add Student – Representative Example)

The Sequence Diagram demonstrates the exact interaction and message flow between objects when the **Add Student** operation is performed.

Participants (left to right): **User** → **ConsoleUI** → **StudentService** → **StudentRepository** → **Student**.

Step-by-step flow:

1. User selects option 1.
2. ConsoleUI prompts for name, age, grade, and email and collects input.
3. ConsoleUI calls addStudent(...) on StudentService.
4. StudentService validates the data (non-empty name, positive age, valid email).
5. If validation passes, StudentService forwards the request to StudentRepository.

6. StudentRepository creates a new Student object with the next auto-incremented ID, adds it to the ArrayList, and returns the object.
7. Success confirmation flows back through Service → ConsoleUI → User.
8. If validation fails, an appropriate error message is displayed immediately without involving the repository.

This diagram proves the correct implementation of separation of concerns, input validation, and layered communication.

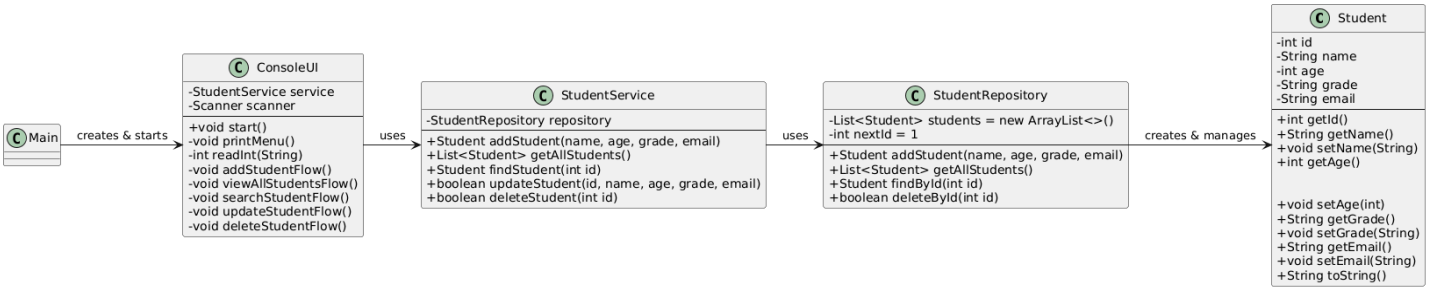

Fig 7.4 Class Diagram

The Class Diagram provides a complete static view of the system structure.

It contains five classes with their exact attributes, operations, and relationships:

- **Student** (Model): Private fields (id, name, age, grade, email) with public getters/setters and toString().
- **StudentRepository** (Data Access): Holds the ArrayList<Student> and nextId; provides add, retrieve, search, and delete methods.
- **StudentService** (Business Logic): Contains a reference to the repository and implements validation + all CRUD operations.
- **ConsoleUI** (Presentation): Holds a StudentService reference and a Scanner; implements the menu loop and user interaction flows.
- **Main**: Entry point that instantiates and connects all layers.

8. Design Decisions & Rationale

Several conscious design decisions were made to ensure the project is clean, professional, extensible, and fully aligned with object-oriented principles:

1. **Layered Architecture (Presentation → Service → Repository → Model)** Chosen over a single-class or monolithic approach to enforce separation of concerns. This makes the code easy to read, test, and maintain, and allows future replacement of any layer (e.g., swapping in-memory storage with a database) without touching other parts.
2. **Repository Pattern** All direct interaction with the ArrayList is encapsulated inside StudentRepository. This provides a clean abstraction over data storage and makes future migration to file-based or database persistence straightforward.
3. **Service Layer for Validation and Business Rules** Input validation (non-empty name, positive age, email containing '@') and business logic are centralized in StudentService instead of scattering them across the UI or repository. This keeps the UI simple and ensures rules are applied consistently.
4. **Auto-incrementing ID in Repository** A private nextId counter guarantees unique student IDs without requiring user input or external dependencies, preserving referential integrity even in memory.
5. **Partial Update Support** The update operation allows the user to leave fields blank to retain existing values. This user-friendly behavior is implemented cleanly using null/-1 checks in the Service layer.
6. **Console-based UI** A deliberate choice to focus entirely on core logic, architecture, and clean code rather than GUI complexity, as permitted and encouraged by the course requirements.
7. **Defensive Programming & Robust Input Handling** A dedicated readInt() method with a loop eliminates InputMismatchException. All user inputs are validated before processing, preventing crashes and ensuring reliability.
8. **Immutable ID** The id field has only a getter and no setter, ensuring the primary identifier can never be accidentally changed after creation.

These decisions collectively elevate the project from a basic console program to a well-structured, production-like application.

9. Implementation Details

The project is implemented using five focused, single-responsibility classes:

- Student.java – Plain Old Java Object (POJO) representing a student record with private fields (id, name, age, grade, email), appropriate getters and setters, and an overridden toString() method for readable output.
- StudentRepository.java – Manages in-memory storage using an ArrayList<Student>, maintains an auto-incrementing ID counter, and provides core CRUD operations (add, retrieve all, find by ID, delete by ID).
- StudentService.java – Contains the complete business logic and validation rules; receives the repository via constructor injection and coordinates all operations while ensuring data integrity.
- ConsoleUI.java – Implements the interactive console interface, displays the menu, handles user input, invokes appropriate service methods, and presents results and error messages in a user-friendly manner.
- Main.java – Serves as the application entry point; instantiates the repository, service, and UI layers, wires them together, and starts the program loop.

The classes follow a clean dependency chain: Main → ConsoleUI → StudentService → StudentRepository → Student. All fields are private, dependencies are injected via constructors, and the code adheres strictly to encapsulation and single-responsibility principles, resulting in a modular, maintainable, and easily extensible implementation.

10. Screenshots/Results

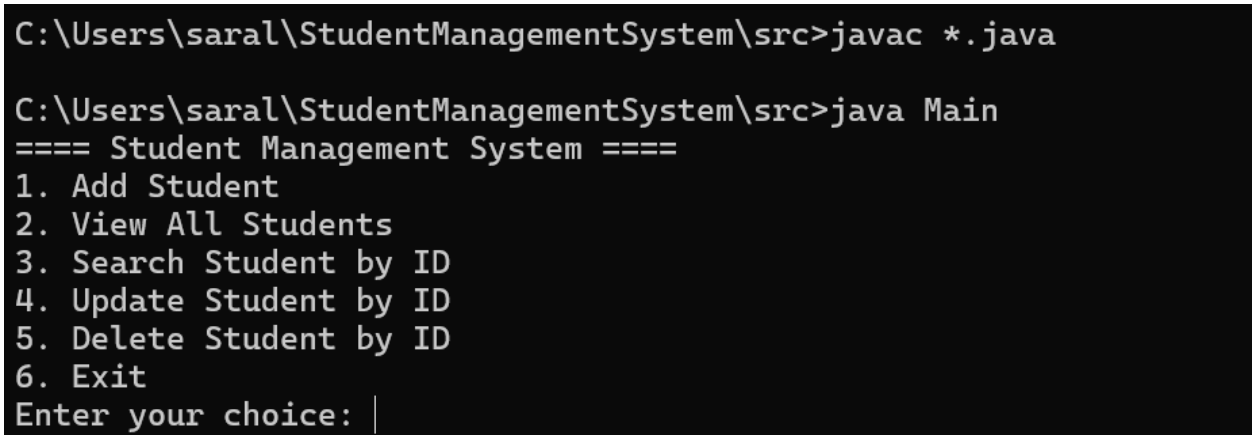

Fig 10.1 StudentManagementSystem Main Menu

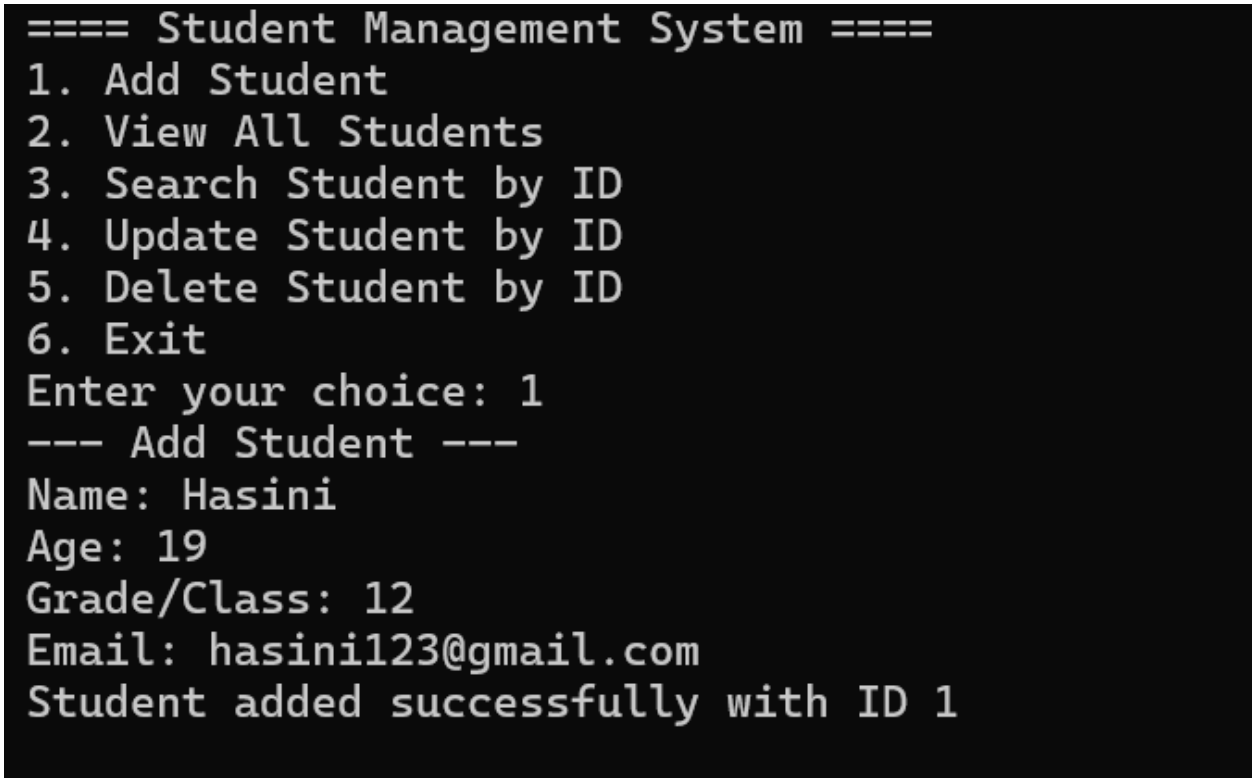

Fig 10.2 Adding a new student

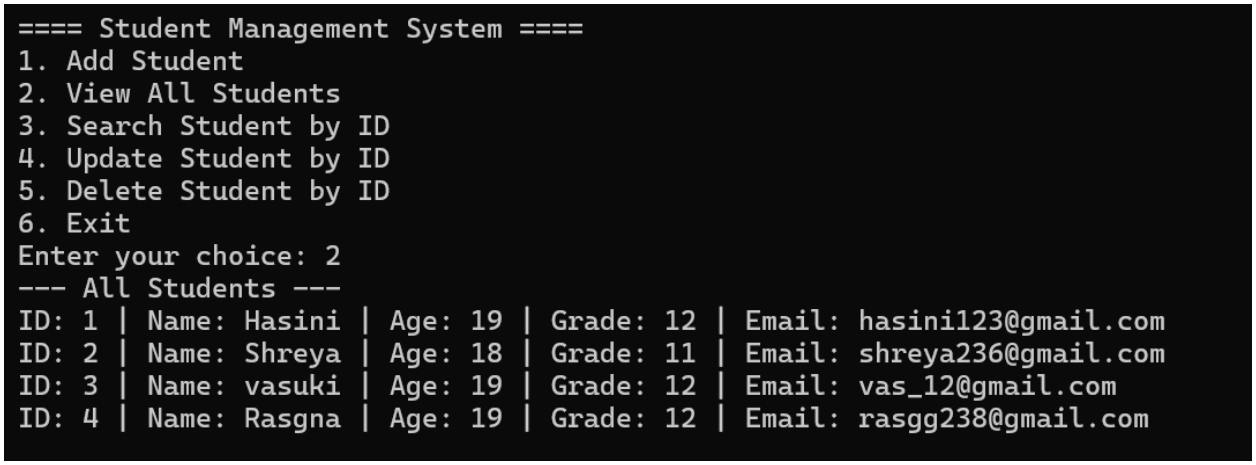

Fig 10.3 Viewing all students

```
==== Student Management System ====
1. Add Student
2. View All Students
3. Search Student by ID
4. Update Student by ID
5. Delete Student by ID
6. Exit
Enter your choice: 3
--- Search Student ---
Enter student ID: 3
Student details:
ID: 3 | Name: vasuki | Age: 19 | Grade: 12 | Email: vas_12@gmail.com
```

Fig 10.4 Searching student by ID

```
==== Student Management System ====
1. Add Student
2. View All Students
3. Search Student by ID
4. Update Student by ID
5. Delete Student by ID
6. Exit
Enter your choice: 4
--- Update Student ---
Enter student ID: 4
Leave field empty to keep old value.
New name (Rasgna):
New age (19): 20
New grade (12):
New email (rasgg238@gmail.com):
Student updated successfully.
```

Fig 10.5 Updating student details

```
==== Student Management System ====
1. Add Student
2. View All Students
3. Search Student by ID
4. Update Student by ID
5. Delete Student by ID
6. Exit
Enter your choice: 2
--- All Students ---
ID: 1 | Name: Hasini | Age: 19 | Grade: 12 | Email: hasini123@gmail.com
ID: 2 | Name: Shreya | Age: 18 | Grade: 11 | Email: shreya236@gmail.com
ID: 3 | Name: vasuki | Age: 19 | Grade: 12 | Email: vas_12@gmail.com
ID: 4 | Name: Rasgna | Age: 20 | Grade: 12 | Email: rasgg238@gmail.com
```

Fig 10.6 viewing the updated student details

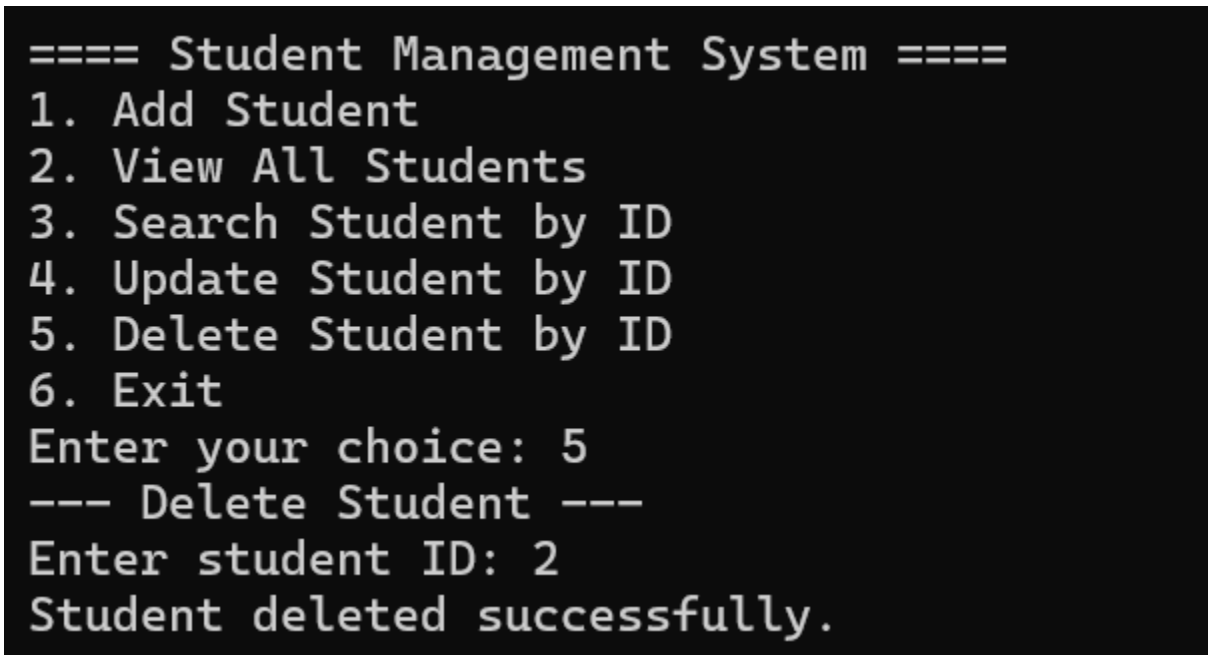

Fig 10.7 Deleting a student

```
==== Student Management System ====
1. Add Student
2. View All Students
3. Search Student by ID
4. Update Student by ID
5. Delete Student by ID
6. Exit
Enter your choice: 2
--- All Students ---
ID: 1 | Name: Hasini | Age: 19 | Grade: 12 | Email: hasini123@gmail.com
ID: 3 | Name: vasuki | Age: 19 | Grade: 12 | Email: vas_12@gmail.com
ID: 4 | Name: Rasgna | Age: 20 | Grade: 12 | Email: rasgg238@gmail.com
```

Fig 10.8 viewing the student details after deletion

11. Testing Approach

Manual testing was performed with the following test cases:

Test Case	Input	Expected Result	Status
Add valid student	Name=Rasgna, Age=19, Grade=12, valid email	Success + ID generated	Pass
Add with empty name	Name=""	Error message	Pass
Add with invalid email	email="ras"	Error message	Pass
Search non-existing ID	ID=999	"Student not found"	Pass
Update only email	Leave other fields blank	Only email changes	Pass
Delete non-existing ID	ID=999	"Student not found"	Pass

12. Challenges Faced

- Setting up the Java JDK environment and PATH configuration.
- Implementing robust partial update logic

- Designing clean separation between layers initially
- Creating accurate UML diagrams for the first time
- Structuring multiple Java classes into a working modular application.

13. Learnings & Key Takeaways

- Deep understanding of OOP principles and their real-world application.
- Working with ArrayList and object storage.
- Importance of layered architecture for maintainability.
- Value of input validation and defensive programming.
- Practical experience with Git & GitHub workflow.
- Documentation and diagramming are critical for professional projects.

14. Future Enhancements

- Replace in-memory storage with persistent database (MySQL/PostgreSQL) or file storage (JSON/CSV).
- Add user authentication and role-based access (Admin vs Viewer).
- Develop a graphical user interface using JavaFX or Swing.
- Implement additional modules: attendance tracking, grade management, fee records.
- Add export functionality to PDF/Excel reports.
- Include search by name and sorting/filtering capabilities.
- Write automated unit tests using JUnit for all service and repository methods.

15. References

1. Oracle Java Documentation – https://docs.oracle.com/en/java/
2. Online Java tutorials
3. GeeksforGeeks Java Collections and Scanner articles
4. vityarthi course lecture notes/slides