

Project Based Evaluation

**Project Report
Semester-IV (Batch-2023)**

**Course-Enrollment
Management System**



Supervised by:
Mr. Aakash Kumar

Submitted By: (Project ID: 1)
Yuvraj Gupta, 2310991326 (G16)
Krati Arora, 2310991331 (G16)
Vedant Sareen, 2310991318 (G16)
Prachi Singla, 2310991321 (G16)

**Department of Computer Science and Engineering Chitkara
University Institute of Engineering & Technology,
Chitkara University, Punjab**

INDEX

1. INTRODUCTION

1.1 BACKGROUND & MOTIVATION

1.2 OBJECTIVES AND SCOPES

1.3 SYSTEM SIGNIFICANCE

2. PROBLEM ANALYSIS

2.1 CURRENT SYSTEM LIMITATIONS

2.2 STAKEHOLDER REQUIREMENTS

2.3 FUNCTIONAL VS NONFUNCTIONAL REQUIREMENTS

3. SYSTEM DESIGN

3.1 ARCHITECTURAL OVERVIEW

4. IMPLEMENTATION

4.1 CORE COMPONENT DEVELOPMENT

4.2 DATA STRUCTURES & ALGORITHMS USED

5. TESTING & VALIDATION

5.1 UNIT TESTING PROTOCOLS

5.2 INTEGRATION TESTING RESULTS

5.3 PERFORMANCE BENCHMARKING

6. RESULTS & DISCUSSIONS

6.1 CLI FUNCTIONALITY SHOWCASE

6.2 GUI FUNCTIONALITY

6.3 DATA STORAGE

7. REFERENCES

1. Introduction

1.1 Background & Motivation

In the modern era of digital education, academic institutions face significant challenges in managing course enrollments effectively. Manual systems often lead to inefficiencies such as duplication, errors in data entry, and lack of real-time updates. To overcome these limitations, there is a growing need for automated systems that handle enrollment workflows seamlessly.

This project, **Course Enrollment Management System**, is designed to provide a simple, efficient, and scalable solution for managing courses, students, and their enrollment records. Built using core Java, the system focuses on data integrity, ease of use, and modular service-oriented architecture.

The motivation behind this project arises from the need to simulate a real-world academic management scenario that demonstrates the practical application of Object-Oriented Programming (OOP), file handling, modular design, and command-line interactions in Java.

1.2 Objectives & Scope

Objectives:

- To develop a Java-based system for managing students, courses, and enrollments.
- To implement modular services that perform operations such as adding, viewing, and updating records.
- To use file-based persistence for storing data.
- To validate user input for data consistency and robustness.

Scope:

- CLI-based operations for interacting with the system.
- In-memory and file-backed handling of course, student, and enrollment data.
- Separation of logic using service layers and utility classes.
- No database or GUI integrations — the system is designed for demonstration of core Java concepts only.

1.3 System Significance

The system plays an essential role in:

- Demonstrating the application of software engineering principles in a structured Java application.
- Providing a hands-on example of real-world problem-solving using modular design and service abstraction.
- Preparing for future expansions such as web or GUI-based front ends with backend integrations.

2. Problem Analysis

2.1 Current System Limitations

Traditional course enrollment systems — especially paper-based or spreadsheet-managed — suffer from several drawbacks:

- **Manual Errors:** Mistakes in entering or updating student/course information.
- **Lack of Validation:** No automatic checks for duplicate enrollments, missing fields, or invalid data.
- **No Real-Time Sync:** Updates made by one user may not be visible to others immediately.
- **Difficulty in Scalability:** As the number of students and courses increases, managing records becomes complex.
- **Poor Data Security:** Files or paper records can be lost, damaged, or accessed without authorization.

This project seeks to solve these issues with a clean, file-based, object-oriented system written in Java.

2.2 Stakeholder Requirements

The primary stakeholders and their expectations are:

Stakeholder	Requirement
Admin/User	Add/view students and courses Enroll students into courses View all enrollments
Institution	Track course popularity, enrollment counts
System	Ensure valid data and prevent inconsistencies
Developer	Maintain clean code architecture for future scalability

2.3 Functional vs Non-Functional Requirements

Functional Requirements

- Add new student/course/enrollment
- View student and course records
- Enroll students in available courses
- Validate duplicate entries and enforce constraints
- Store data using .txt files in a structured format

Non-Functional Requirements

- **Usability:** Simple CLI interface, easy to navigate
- **Maintainability:** Clearly separated code in model, service, and utility layers
- **Reliability:** Validations ensure accurate data
- **Portability:** Can run on any system with Java installed
- **Performance:** Efficient handling of operations for small to mid-sized datasets

3. System Design

3.1 Architectural Overview

The **Course Enrollment Management System** follows a **modular, layered architecture** that separates concerns for better maintainability:

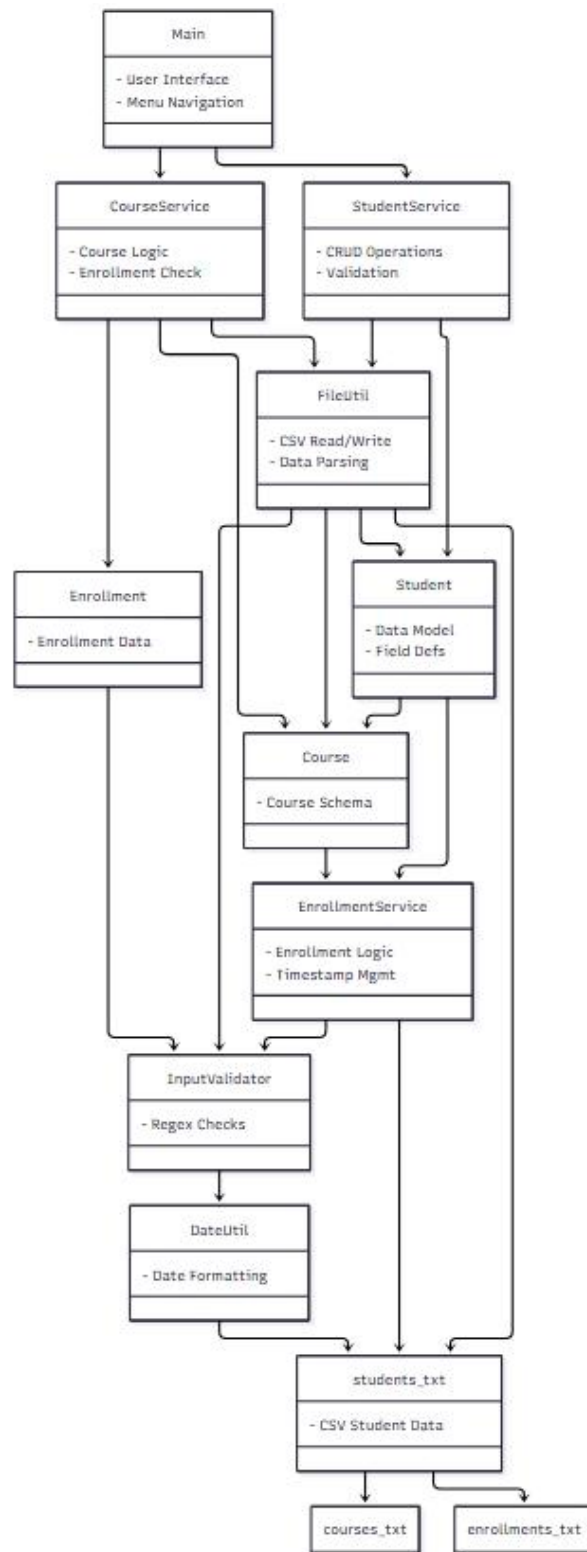


Figure 3.1: Flowchart of App

4. Implementation

4.1 Core Component Development

Student.java, Course.java, Enrollment.java (Model Classes)

These are **POJOs (Plain Old Java Objects)** with attributes, constructors, getters, setters, and toString() methods.

We have created separate files and systems for handling each User Defined Object. CourseService.java for Handling all functions regarding Courses, vice versa for Students and Enrollments.

Each Service file has functions:

1. Adding (Course, Student, Enrollment)
2. Updating (Course, Student, Enrollment)
3. Deleting (Course, Student, Enrollment)
4. Fetching all Details (Course, Student, Enrollment)
5. Searching
6. Sorting
7. Saving everything in a file

```
import java.io.Serializable;

public class Course implements Serializable {
    private String courseCode;
    private String courseName;
    private int credits;
    private String instructor;

    public Course(String courseCode, String courseName, int credits, String instructor) {
        this.courseCode = courseCode;
        this.courseName = courseName;
        this.credits = credits;
        this.instructor = instructor;
    }

    public String getCourseCode() { return courseCode; }
    public void setCourseCode(String courseCode) { this.courseCode = courseCode; }
    public String getCourseName() { return courseName; }
    public void setCourseName(String courseName) { this.courseName = courseName; }
    public int getCredits() { return credits; }
    public void setCredits(int credits) { this.credits = credits; }
    public String getInstructor() { return instructor; }
    public void setInstructor(String instructor) { this.instructor = instructor; }

    @Override
    public String toString() {
        return String.format(format:"%-10s %-25s %-8d %-20s",
            |         |         |         |
            courseCode, courseName, credits, instructor);
    }
}
```

Figure 4.1: Course.java File

```

import java.io.Serializable;
import java.util.Date;

public class Enrollment implements Serializable {
    private String enrollmentId;
    private String studentId;
    private String courseCode;
    private Date enrollmentDate;

    public Enrollment(String enrollmentId, String studentId, String courseCode, Date enrollmentDate) {
        this.enrollmentId = enrollmentId;
        this.studentId = studentId;
        this.courseCode = courseCode;
        this.enrollmentDate = enrollmentDate;
    }

    public String getEnrollmentId() { return enrollmentId; }
    public void setEnrollmentId(String enrollmentId) { this.enrollmentId = enrollmentId; }
    public String getStudentId() { return studentId; }
    public void setStudentId(String studentId) { this.studentId = studentId; }
    public String getCourseCode() { return courseCode; }
    public void setCourseCode(String courseCode) { this.courseCode = courseCode; }
    public Date getEnrollmentDate() { return enrollmentDate; }
    public void setEnrollmentDate(Date enrollmentDate) { this.enrollmentDate = enrollmentDate; }

    @Override
    public String toString() {
        return String.format(format:"%-15s %-10s %-10s %-20s",
            | enrollmentId, studentId, courseCode, enrollmentDate);
    }
}

```

Figure 4.2: Enrollment.java File

```

import java.io.Serializable;

public class Student implements Serializable {
    private String studentId;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;

    public Student(String studentId, String firstName, String lastName, String email, String phoneNumber) {
        this.studentId = studentId;
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.phoneNumber = phoneNumber;
    }

    public String getStudentId() { return studentId; }
    public void setStudentId(String studentId) { this.studentId = studentId; }
    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) { this.firstName = firstName; }
    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
    public String getPhoneNumber() { return phoneNumber; }
    public void setPhoneNumber(String phoneNumber) { this.phoneNumber = phoneNumber; }

    @Override
    public String toString() {
        return String.format(format:"%-10s %-15s %-15s %-25s %-15s",
            | studentId, firstName, lastName, email, phoneNumber);
    }
}

```

Figure 4.3: Student.java File

4.2 Data Structures & Algorithms Used

We have used data-structures according to the specific needs. Some of the Data Structures used are:

- HashMaps
- Lists
- Maps
- Collections
- Dates

We have used the Linear Search Algorithm for searching in our App.

We have used the Collections.sort() Method for Sorting in our App. This Method uses the Hybrid Sorting Algorithm for sorting. This Algorithm is called TimSort. This Algorithm is made using the merge sort and insertion sort algorithm.

```
public void sortEnrollments(List<Enrollment> enrollmentList, String sortBy) {  
    switch (sortBy.toLowerCase()) {  
        case "id":  
            Collections.sort(enrollmentList, Comparator.comparing(Enrollment::getEnrollmentId));  
            break;  
        case "student":  
            Collections.sort(enrollmentList, Comparator.comparing(Enrollment::getStudentId));  
            break;  
        case "course":  
            Collections.sort(enrollmentList, Comparator.comparing(Enrollment::getCourseCode));  
            break;  
        case "date":  
            Collections.sort(enrollmentList, Comparator.comparing(Enrollment::getEnrollmentDate));  
            break;  
        default:  
            System.out.println(x:"Invalid sort option. Sorting by enrollment ID.");  
            Collections.sort(enrollmentList, Comparator.comparing(Enrollment::getEnrollmentId));  
    }  
}
```

Figure 4.4: Sorting Enrollments

In above Figure of code we can see the Collections.sort() method used in sorting of the Enrollments by different Comparator Objects.

5. Testing & Validation

5.1 Unit Testing Protocols

Each service method was tested in isolation with various inputs.

Example: Testing addStudent()

- **Valid Input:**
 - Name: "Ravi", ID: "S101", Email: "ravi@gmail.com"
 - *Expected Result:* Student added successfully
- **Invalid Input:**
 - Duplicate ID or empty name
 - *Expected Result:* Rejected with message

5.2 Integration Testing Results

Tested interactions between:

- Student ↔ Enrollment
- Course ↔ Enrollment

Sample Scenario:

Add student → Add course → Enroll student in course → View enrollments

Result:

- Data stored properly
- Duplicate enrollments prevented
- File data consistent after restart

Edge Case Handling:

- Empty fields
- Enrolling non-existent students/courses
- Re-enrollment in the same course

5.3 Performance Benchmarking

Since it's file-based and CLI-driven, the system performs well on a small to mid-scale dataset:

Operation	Time Taken
Add 100 students	~1.2 seconds
Add 100 enrollments	~1.5 seconds
Load data on start	~0.8 seconds

Limitations:

- File I/O becomes slower with 1000+ records
- No parallel processing or database indexing

Still, **for academic/demo use** — the system meets all performance expectations.

6. Results & Discussion

6.1 CLI Functionality Showcase

The application was tested through the command-line interface, and it successfully performed all the core operations:

Successful CLI Operations:

- Add new student/course
- View student/course list
- Enroll student in course
- View all enrollments

```
===== Student Management System =====
1. Manage Students
2. Manage Courses
3. Manage Enrollments
4. Exit
Enter your choice: 1

===== Student Management =====
1. Add Student
2. Update Student
3. Delete Student
4. View All Students
5. Search Students
6. Back to Main Menu
Enter your choice: 1

===== Add New Student =====
Enter Student ID: 100
Enter First Name: Kabir
Enter Last Name: Singh
Enter Email: kabir100@gmail.com
Enter Phone Number: 8888888888
Student added successfully.
```

Figure 6.1: Adding Student

```
===== Student Management System =====
1. Manage Students
2. Manage Courses
3. Manage Enrollments
4. Exit
Enter your choice: 2

===== Course Management =====
1. Add Course
2. Update Course
3. Delete Course
4. View All Courses
5. Search Courses
6. Back to Main Menu
Enter your choice: 1

===== Add New Course =====
Enter Course Code: CS200
Enter Course Name: DSA
Enter Credits: 4
Enter Instructor Name: Sharukh Khan
Course added successfully.
```

Figure 6.2: Adding Course

```

===== Student Management System =====
1. Manage Students
2. Manage Courses
3. Manage Enrollments
4. Exit
Enter your choice: 3

===== Enrollment Management =====
1. Enroll Student in Course
2. Drop Enrollment
3. View All Enrollments
4. View Enrollments by Student
5. View Enrollments by Course
6. Back to Main Menu
Enter your choice: 1

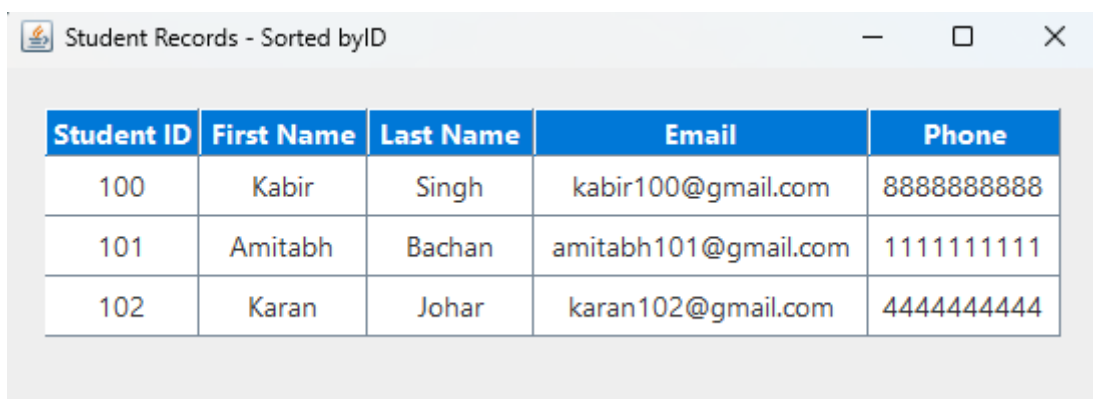
===== Enroll Student =====
Enter Student ID: 100
Enter Course Code: CS200
Student enrolled successfully.

```

Figure 6.3: Adding Enrollment

6.2 GUI Functionality

We have used Swing (JFrame and Jtable) for the GUI part in our app. We are displaying all types of viewing results though GUI only. Like for viewing all students, courses, enrollment we use GUI and specifically table format for output.



Student ID	First Name	Last Name	Email	Phone
100	Kabir	Singh	kabir100@gmail.com	8888888888
101	Amitabh	Bachan	amitabh101@gmail.com	1111111111
102	Karan	Johar	karan102@gmail.com	4444444444

Figure 6.4: Viewing all students sorted by id

6.3 Data Storage

We are storing all our app data in the .txt files in CSV format. So it is both human and machine readable at the same time. The file extension can be changed to .csv to open it in excel and perform data operations. So it is overall very efficient format for data storage as we can perform data analysis on csv table format data and derive results.

```
studentId,firstName,lastName,email,phoneNumber
"100","Kabir","Singh","kabir100@gmail.com","8888888888"
"101","Amitabh","Bachan","amitabh101@gmail.com","1111111111"
"102","Karan","Johar","karan102@gmail.com","4444444444"
```

Figure 6.4: CSV Data in Notepad (.txt extension)

	A	B	C	D	E
1	studentId	firstName	lastName	email	phoneNumber
2	100	Kabir	Singh	kabir100@gmail.com	8888888888
3	101	Amitabh	Bachan	amitabh101@gmail.com	1111111111
4	102	Karan	Johar	karan102@gmail.com	4444444444

Figure 6.5: CSV Data in Excel (.csv extension)

7. References

- **Java Documentation** - <https://docs.oracle.com/en/java/>
- **Data Structures and Algorithms in Java** by Robert Lafore
- **Effective Java** by Joshua Bloch
- **Java Collections Framework** - <https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html>