Project Report

Programming in java

(CSE2006)

Submitted by:

NAME: RUDRA PRATAP SINGH

REG No: 24BCE10619

Bachelor of Technology

In

Computer Science & Engineering

School of Computer Science and Engineering

VIT Bhopal University

Introduction

The Campus Course & Records Manager (CCRM) is a Java SE console application that manages the full academic lifecycle, including students, courses, enrollments, grades, and transcripts. Built as an educational project using advanced OOP and modern Java (JDK 17+), it offers end-to-end record management with file-based persistence, import/export tools, and automated backups.

Problem Statement

Educational institutions face significant challenges in managing academic records manually or through fragmented systems:

Core Problems:

1. **Data Fragmentation**: Student information, course details, and enrollment records are often scattered across multiple systems or spreadsheets
2. **Manual Record Keeping**: Manual tracking of grades, enrollments, and transcripts is time-consuming and error-prone
3. **Lack of Validation**: No automated enforcement of business rules such as credit limits, duplicate enrollments, or data integrity
4. **Limited Reporting**: Difficulty in generating analytics such as GPA distributions, student rankings, or enrollment statistics
5. **Data Loss Risk**: No systematic backup mechanisms leading to potential data loss
6. **Integration Challenges**: Limited ability to import/export data from external systems

Target Users:

- Academic administrators managing student records
- Course coordinators handling enrollments
- Examination departments processing grades
- Educational institutions requiring a lightweight, portable academic management system

Scope: CCRM addresses these challenges by providing a unified, console-based system that:

- Centralizes all academic data in a structured format
- Automates business rule validation
- Provides comprehensive CRUD operations for all entities
- Enables data import/export for system integration
- Implements automated backup mechanisms

Functional Requirements

FR1: Student Management

- **FR1.1**: Add new students with registration number, name, email, and enrollment date
- **FR1.2**: List all students with their details
- **FR1.3**: Update student email addresses
- **FR1.4**: Deactivate students (soft delete)
- **FR1.5**: Search students by registration number
- **FR1.6**: Display student profile with enrollment history

FR2: Course Management

- **FR2.1**: Create courses with code, title, credits, department, and semester
- **FR2.2**: List all available courses
- **FR2.3**: Filter courses by department and/or semester
- **FR2.4**: Deactivate courses (soft delete)
- **FR2.5**: Assign instructors to courses
- **FR2.6**: Validate credit range (1-6 credits per course)

FR3: Enrollment Management

- **FR3.1**: Enroll students in courses for specific semesters
- **FR3.2**: Prevent duplicate enrollments
- **FR3.3**: Enforce maximum credit limit per semester (configurable)
- **FR3.4**: Unenroll students from courses
- **FR3.5**: Track enrollment timestamps
- **FR3.6**: Maintain enrollment history per student

FR4: Grade Management

- **FR4.1**: Record marks (0-100) for enrolled courses
- **FR4.2**: Automatic grade calculation based on marks (S/A/B/C/D/E/F)
- **FR4.3**: Grade point assignment (S=10, A=9, B=8, C=7, D=6, E=5, F=0)
- **FR4.4**: Update grades when marks are modified
- **FR4.5**: Support for grade-based filtering and analytics

FR5: Transcript Generation

- **FR5.1**: Generate complete transcript for students
- **FR5.2**: Display course-wise grades and marks
- **FR5.3**: Calculate overall GPA
- **FR5.4**: Format transcript with student and course details

FR6: Import/Export

- **FR6.1**: Import student data from CSV files
- **FR6.2**: Import course data from CSV files
- **FR6.3**: Export all data (students, courses, enrollments) to CSV
- **FR6.4**: Handle malformed CSV data gracefully
- **FR6.5**: Support UTF-8 encoding for international characters

FR7: Backup & Recovery

- **FR7.1**: Create timestamped backups of export folder
- **FR7.2**: Calculate and display recursive directory sizes
- **FR7.3**: Maintain backup history with timestamp-based folders
- **FR7.4**: Copy files preserving structure using NIO.2

FR8: Reporting & Analytics

- **FR8.1**: Generate GPA distribution reports
- **FR8.2**: Display top-performing students

Non-functional Requirements

NFR1: Performance

- **NFR1.1**: System must handle up to 1000 students efficiently in memory
- **NFR1.2**: Course search and filtering operations should complete within 100ms
- **NFR1.3**: CSV import operations should process 1000 records within 5 seconds
- **NFR1.4**: Backup operations should complete within reasonable time based on data size

NFR2: Reliability

- **NFR2.1**: Data integrity maintained through validation and assertions
- **NFR2.2**: Exception handling for all file I/O operations
- **NFR2.3**: Automatic rollback on validation failures (no partial updates)
- **NFR2.4**: Consistent data state maintained in DataStore singleton

NFR3: Usability

- **NFR3.1**: Intuitive menu-driven interface with clear prompts
- **NFR3.2**: Descriptive error messages for user guidance
- **NFR3.3**: Confirmation messages for successful operations
- **NFR3.4**: Input validation with meaningful feedback

NFR4: Maintainability

- **NFR4.1**: Clean separation of concerns (domain, service, CLI, I/O)
- **NFR4.2**: Well-documented code with inline comments
- **NFR4.3**: Consistent naming conventions following Java standards
- **NFR4.4**: Modular design enabling easy extension

NFR5: Portability

- **NFR5.1**: Platform-independent (Windows, Linux, macOS)
- **NFR5.2**: No external dependencies beyond JDK 17+
- **NFR5.3**: Manual compilation support without build tools
- **NFR5.4**: UTF-8 encoding for cross-platform compatibility

NFR6: Security

- **NFR6.1**: Input validation to prevent injection attacks
- **NFR6.2**: Private access modifiers for sensitive data
- **NFR6.3**: Immutable value objects where appropriate
- **NFR6.4**: Assertions for invariant checking (enabled via -ea flag)

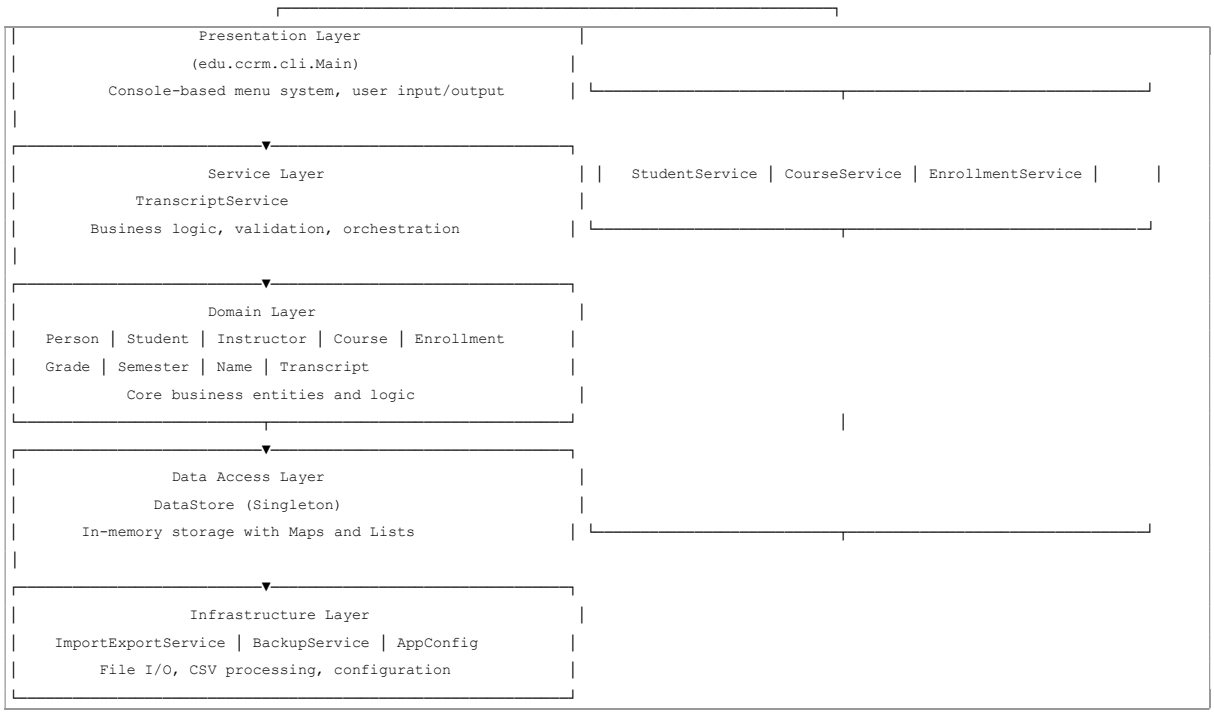
NFR7: Scalability

- **NFR7.1**: Service-oriented architecture enabling future enhancements
- **NFR7.2**: Interface-based design supporting alternative implementations
- **NFR7.3**: In-memory storage can be replaced with database without major refactoring
- **NFR7.4**: Singleton pattern ensuring single source of truth

---

System Architecture

CCRM follows a **layered architecture** with clear separation of concerns:



Architecture Components:

Presentation Layer (CLI)

- **Package**: `edu.ccrm.cli`
- **Responsibility**: User interaction, menu navigation, input validation
- **Key Class**: `Main.java` - Entry point with menu-driven interface

Service Layer

- **Package**: `edu.ccrm.service`
- **Responsibility**: Business logic implementation, transaction coordination
- **Key Interfaces**:
  - `StudentService` - Student CRUD operations
  - `CourseService` - Course management
  -
  -
-

`EnrollmentService` - Enrollment and grading

`TranscriptService` - Transcript generation

Implementations: `*ServiceImpl` classes with concrete logic

Domain Layer

- **Package**: `edu.ccrm.domain`
- **Responsibility**: Core business entities and domain logic
- **Key Classes**:
  - `Person` (abstract) → `Student`, `Instructor` (inheritance)
  - `Course` with Builder pattern
  - `Enrollment` tracking student-course relationship
  - `Grade` enum with point calculation
  - `Semester` enum
  - `Name` immutable value object
  - `Transcript` with Builder pattern

Data Access Layer

- **Package**: `edu.ccrm.config`
- **Responsibility**: Data storage and retrieval
- **Key Class**: `DataStore` (Singleton)
  - Maps for indexed access (students by ID/RegNo, courses by code)
  - Lists for enrollment history
  - In-memory storage with O(1) lookup time

Infrastructure Layer

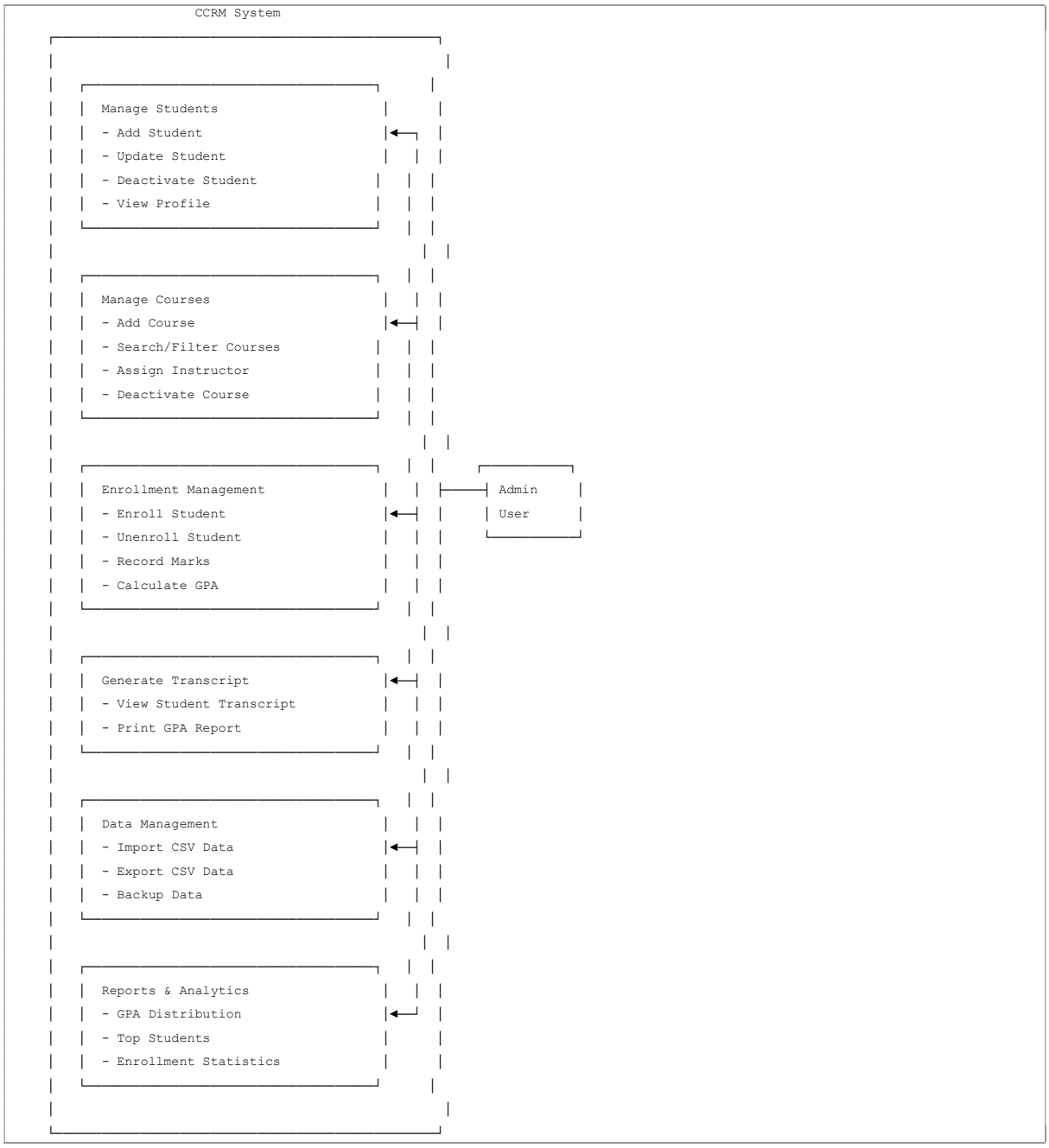
- **Package**: `edu.ccrm.io`, `edu.ccrm.config`
- **Responsibility**: Cross-cutting concerns
- **Key Classes**:
  - `ImportExportService` - CSV import/export using NIO.2
  - `BackupService` - File backup with directory walking
  - `AppConfig` - Configuration management (Singleton)

Design Principles Applied:

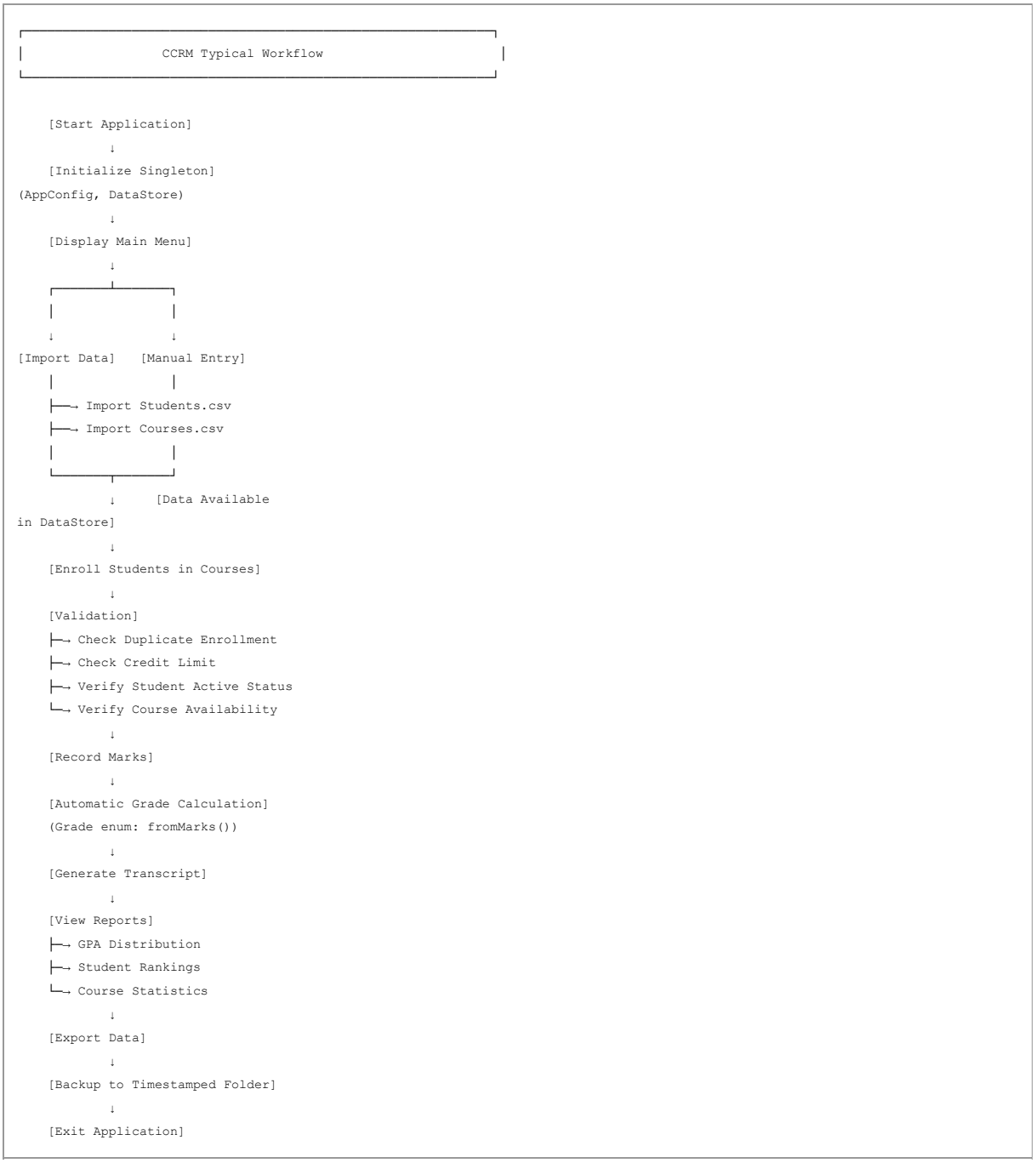
1. **Separation of Concerns**: Each layer has distinct responsibilities
2. **Dependency Inversion**: Services depend on interfaces, not concrete implementations
3. **Single Responsibility**: Each class has one primary purpose
4. **Open/Closed**: Extensible through interfaces without modifying existing code
5. **Encapsulation**: Private fields with controlled access via methods
6. **DRY (Don't Repeat Yourself)**: Common logic centralized in service layer

Design Diagrams

Use Case Diagram

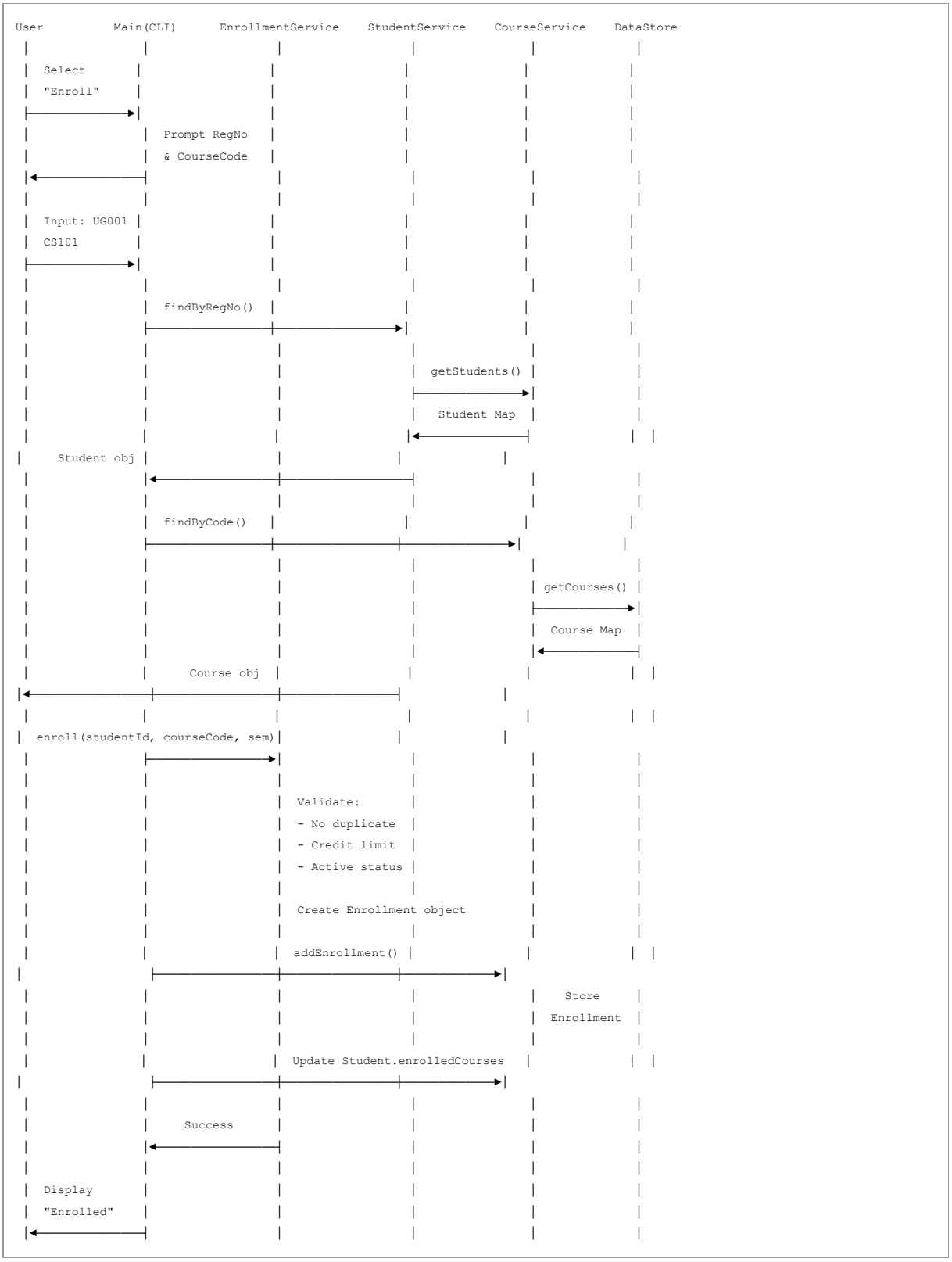

CCRM System
Manage Students
- Add Student
- Update Student
- Deactivate Student
- View Profile
Manage Courses
- Add Course
- Search/Filter Courses
- Assign Instructor
- Deactivate Course
Enrollment Management
- Enroll Student
- Unenroll Student
- Record Marks
- Calculate GPA
Admin
User
Generate Transcript
- View Student Transcript
- Print GPA Report
Data Management
- Import CSV Data
- Export CSV Data
- Backup Data
Reports & Analytics
- GPA Distribution
- Top Students
- Enrollment Statistics

Workflow Diagram

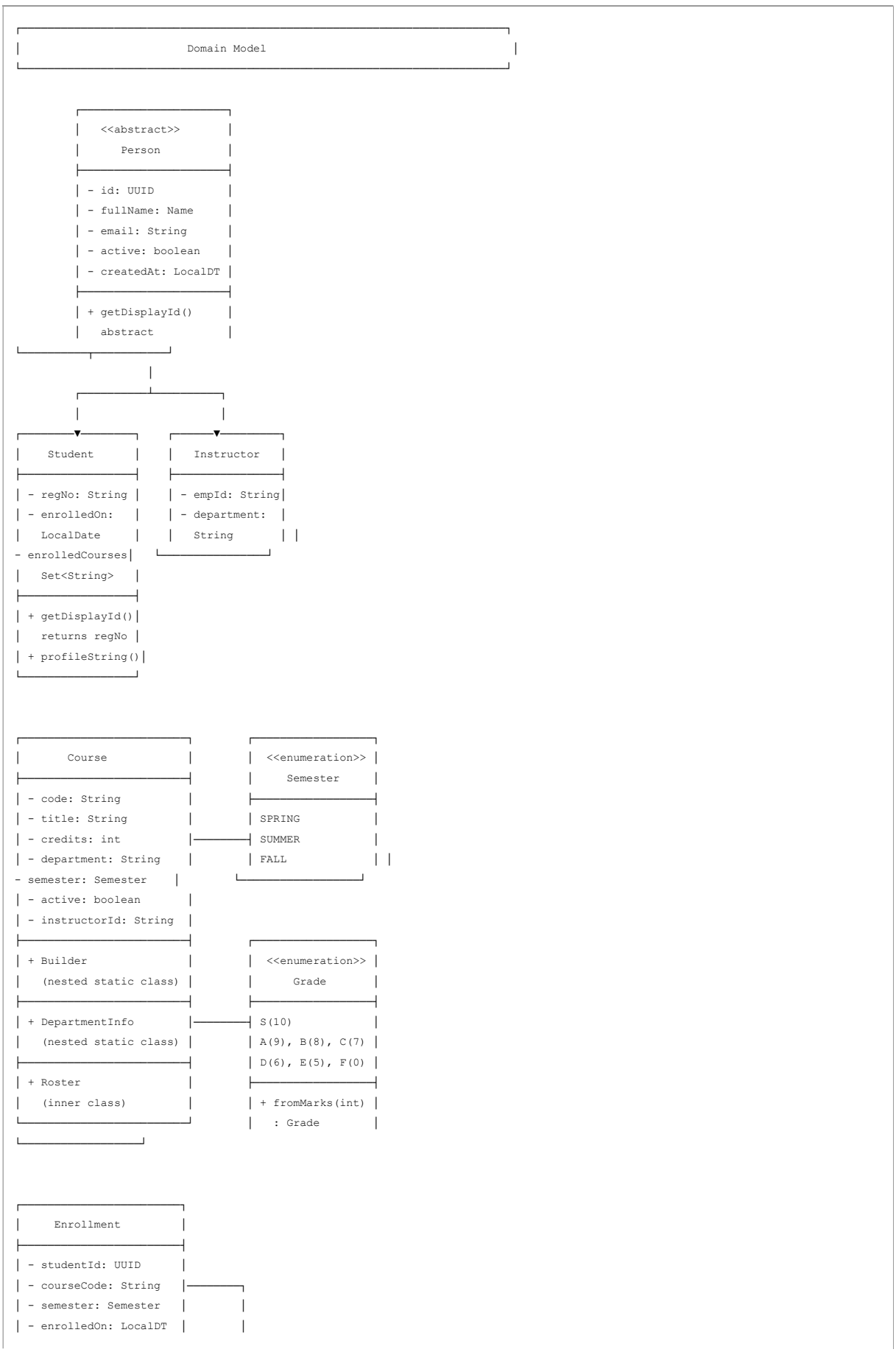


Sequence Diagram

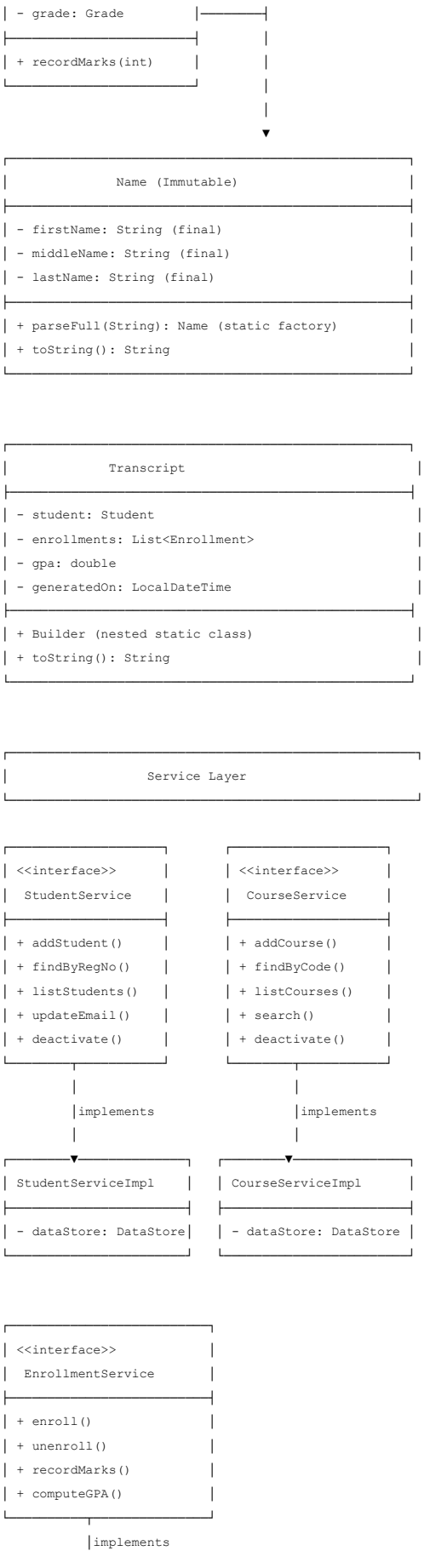
Student Enrollment

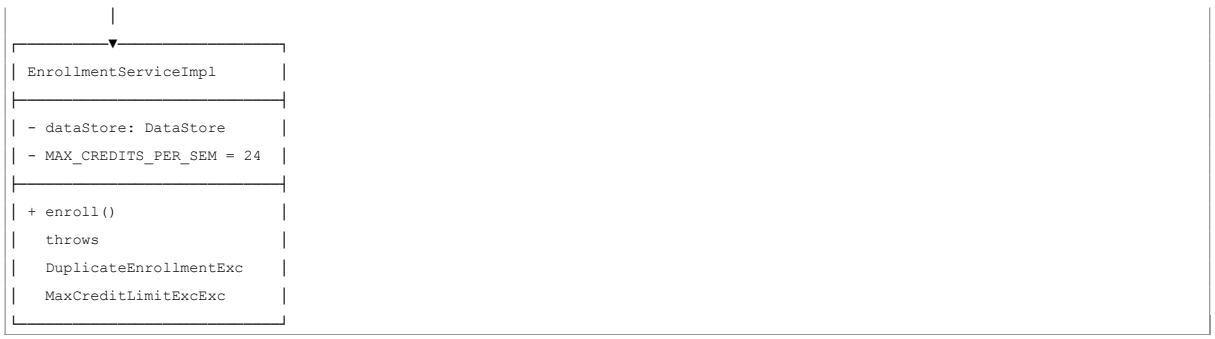


Class Diagram

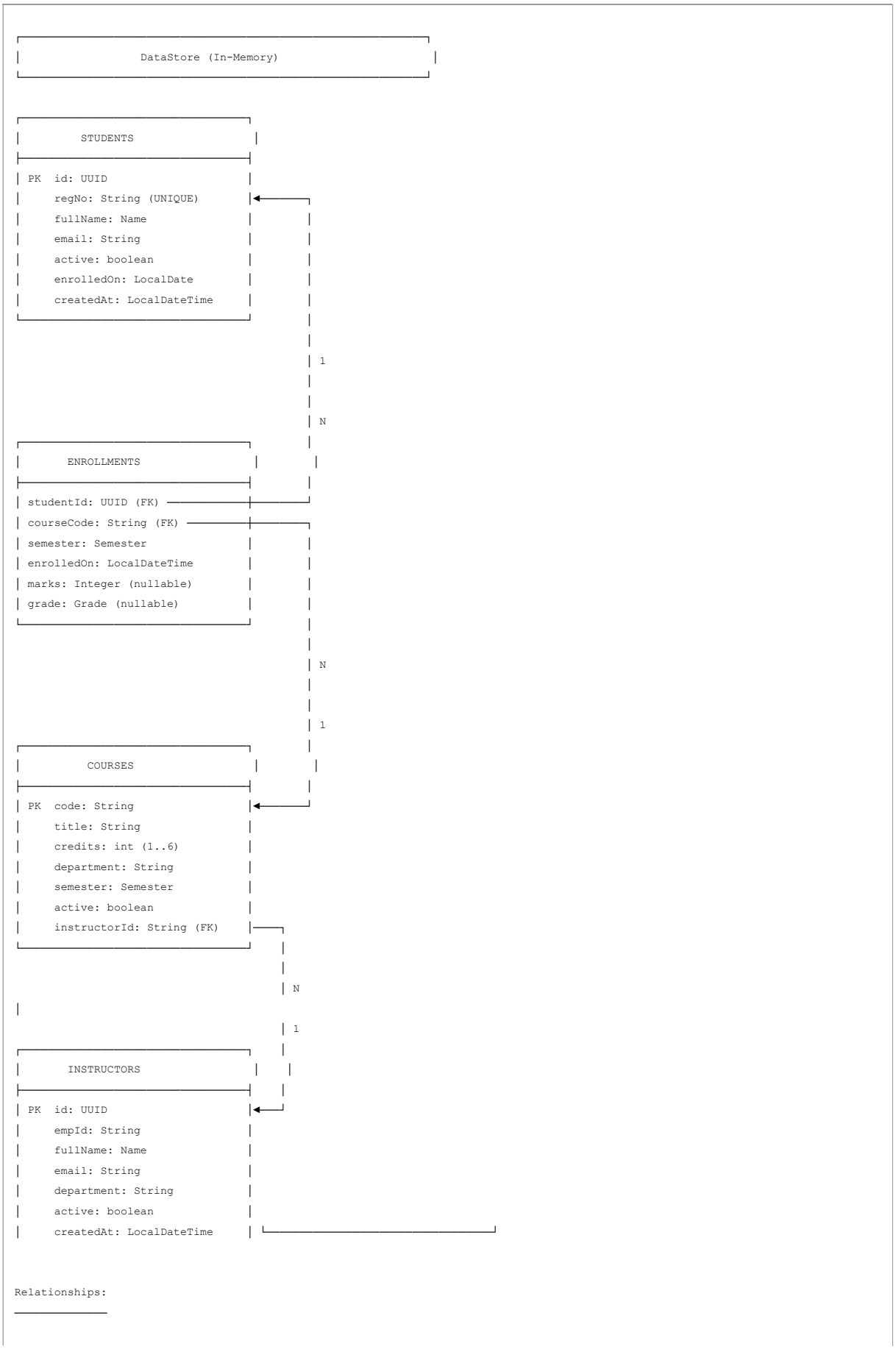

Domain Model
<<abstract>>
Person
- id: UUID
- fullName: Name
- email: String
- active: boolean
- createdAt: LocalDT
+ getDisplayId()
abstract
Student
- regNo: String
- enrolledOn: LocalDate
- enrolledCourses: Set<String>
+ getDisplayId() returns regNo
+ profileString()
Instructor
- empId: String
- department: String
Course
- code: String
- title: String
- credits: int
- department: String
- semester: Semester
- active: boolean
- instructorId: String
+ Builder (nested static class)
+ DepartmentInfo (nested static class)
+ Roster (inner class)
<<enumeration>>
Semester
SPRING
SUMMER
FALL
<<enumeration>>
Grade
S(10)
A(9), B(8), C(7)
D(6), E(5), F(0)
+ fromMarks(int) : Grade
Enrollment
- studentId: UUID
- courseCode: String
- semester: Semester
- enrolledOn: LocalDT

| - marks: Integer | | |


- grade: Grade
+ recordMarks(int)
Name (Immutable)
- firstName: String (final)
- middleName: String (final)
- lastName: String (final)
+ parseFull(String): Name (static factory)
+ toString(): String
Transcript
- student: Student
- enrollments: List<Enrollment>
- gpa: double
- generatedOn: LocalDateTime
+ Builder (nested static class)
+ toString(): String
Service Layer
<<interface>>
StudentService
+ addStudent()
+ findByRegNo()
+ listStudents()
+ updateEmail()
+ deactivate()
<<interface>>
CourseService
+ addCourse()
+ findByCode()
+ listCourses()
+ search()
+ deactivate()
implements
implements
StudentServiceImpl
- dataStore: DataStore
CourseServiceImpl
- dataStore: DataStore
<<interface>>
EnrollmentService
+ enroll()
+ unenroll()
+ recordMarks()
+ computeGPA()
implements



ER Diagram (Data Storage Model)


DataStore (In-Memory)
STUDENTS
PK id: UUID
regNo: String (UNIQUE)
fullName: Name
email: String
active: boolean
enrolledOn: LocalDate
createdAt: LocalDateTime
1
N
ENROLLMENTS
studentId: UUID (FK)
courseCode: String (FK)
semester: Semester
enrolledOn: LocalDateTime
marks: Integer (nullable)
grade: Grade (nullable)
N
1
COURSES
PK code: String
title: String
credits: int (1..6)
department: String
semester: Semester
active: boolean
instructorId: String (FK)
N
1
INSTRUCTORS
PK id: UUID
empId: String
fullName: Name
email: String
department: String
active: boolean
createdAt: LocalDateTime
Relationships:

```
• Student 1:N Enrollment (One student has many enrollments)
• Course 1:N Enrollment (One course has many enrollments)
• Instructor 1:N Course (One instructor teaches many courses)

Indexes/Maps:
_____________
• studentsById: Map<UUID, Student>
• studentIdByRegNo: Map<String, UUID>  (for fast regNo lookup)
• coursesByCode: Map<String, Course>   (PK index)
• instructorsById: Map<UUID, Instructor>
• enrollments: List<Enrollment>        (allows duplicates by design check)


Storage Notes:
______________
✓ In-memory storage using Java Collections
✓ LinkedHashMap preserves insertion order
✓ O(1) lookup time for indexed fields
✓ No database dependency - pure Java SE
✓ Data persistence via CSV export/import
```

Design Decisions & Rationale

Architectural Decisions

Layered Architecture

Decision: Adopt a strict layered architecture with CLI → Service → Domain → Data layers

Rationale:

- Clear separation of concerns makes code easier to understand and maintain
- Each layer can be tested independently
- Future enhancements (e.g., web UI, database) require changes only in specific layers
- Follows industry best practices for enterprise applications

Trade-offs:

- More boilerplate code compared to monolithic design
- Slight performance overhead from layer transitions
- Benefits far outweigh costs for maintainability

In-Memory Storage with Singleton DataStore

Decision: Use in-memory data structures (Maps, Lists) wrapped in a Singleton

Rationale:

- No external database dependency - pure Java SE project
- Fast O(1) lookups for students and courses
- Simple deployment - single JAR file
- Educational focus on Java fundamentals, not database integration
- Data persistence achieved through CSV import/export

trade-offs:

- Data lost on application restart (mitigated by export/backup)
- Not suitable for production multi-user scenarios
- Memory constraints for very large datasets (acceptable for demo)

Service Interface Pattern

Decision: Define service interfaces with concrete implementations

Rationale:

- Dependency Inversion Principle - depend on abstractions
- Enables mock implementations for testing

- Supports future alternative implementations (e.g., database-backed)
- Industry-standard approach in enterprise Java

Trade-offs:

- Additional interface files to maintain
- Slight increase in code complexity
- Benefits include flexibility and testability

Design Pattern Choices

1:Singleton (AppConfig, DataStore)

Why Singleton?

- Ensure single source of truth for configuration and data
- Prevent multiple instances causing data inconsistency
- Global access point needed throughout application
- Lazy initialization not required (eager instantiation used)

Implementation:

```
private static final DataStore INSTANCE = new
DataStore(); private DataStore() {} // private
constructor public static DataStore getInstance() {
return INSTANCE; }
```

2: Builder (Course, Transcript)

Why Builder?

- Course has multiple optional parameters (department, semester, instructorId)
- Improves readability: `new Course.Builder(code, title, credits).department(dept).build()`
- Validates constraints (credits 1-6) at build time
- Immutability after construction

Alternative Considered: Constructor with many parameters

Rejected Because: Poor readability, error-prone parameter ordering

3: Strategy (via Service Interfaces)

Why Strategy?

- Different algorithms for the same operation (e.g., GPA calculation)
- Swap implementations without changing client code
- Supports Open/Closed Principle

4: Template Method (in abstract Person)

Why Template Method?

- Common behavior in Person base class
- Subclasses (Student, Instructor) customize `getDisplayId()`
- Promotes code reuse and polymorphism

Technology Decisions

1: Java SE 17+ (No Frameworks)

Rationale:

- Educational project demonstrating core Java features
- No Spring/Hibernate dependencies - focus on fundamentals
- Modern Java features: records (future), var, enhanced switch
- LTS version ensuring long-term support

-

2: Console-Based UI

Rationale:

- Simple deployment - no web server required
- Focus on backend logic rather than UI frameworks
- Accessible on any platform with Java

Educational clarity - no HTML/CSS/JS complexity

Alternative Considered: JavaFX or Swing GUI **Rejected Because**: Adds complexity, shifts focus from core Java concepts

3: CSV for Import/Export

Rationale:

- Human-readable and editable in any text editor
- No external parsing libraries needed (String.split())
- Compatible with Excel, Google Sheets
- Industry-standard for data exchange

Alternative Considered: JSON, XML **Rejected Because**: Requires external libraries or complex parsi

4: Manual Compilation (No Maven/Gradle)

Rationale:

- Students understand compilation process clearly
- No build tool learning curve
- Demonstrates pure Java without abstractions
- Simple project structure without tool-specific files

Trade-offs:

- Manual dependency management (none in this project)
- More verbose compile commands

Domain Modeling Decisions

1: Abstract Person Base Class

Rationale:

- Student and Instructor share common attributes (id, name, email)
- Demonstrates inheritance and polymorphism
- Abstract `getDisplayId()` enforces subclass implementation
- Reduces code duplication

2: Immutable Name Value Object

Rationale:

- Name is a value, not an entity (no identity)
- Thread-safe by design
- Prevents accidental modification
- Demonstrates immutability principle

Implementation:

```
public final class Name {     private final String firstName;     private final String middleName;     private final String lastName;
    // No setters - immutable
}
```

3: Enum for Grade and Semester

Rationale:

-

- Fixed set of values (type safety)
- Compile-time checking prevents invalid values
- Grade enum encapsulates point calculation logic
- Self-documenting code

Alternative Considered: String constants **Rejected Because**: No type safety, error-prone

4: UUID for Entity IDs

Rationale:

Globally unique - no collision risk

- No centralized ID generator needed
- Supports future distributed scenarios
- Standard Java library (java.util.UUID)

Alternative Considered: Auto-increment integers **Rejected Because**: Requires coordination, not truly unique across systems

Exception Handling Strategy

Decision: Custom Checked Exceptions for Business Rules

rationale:

- `DuplicateEnrollmentException` and `MaxCreditLimitExceededException` are business logic violations
- Checked exceptions force callers to handle gracefully
- Distinguishes business rule violations from technical errors

Implementation:

```
public void enroll(...) throws DuplicateEnrollmentException, MaxCreditLimitExceededException {
    // Business validation     if (alreadyEnrolled) throw new
DuplicateEnrollmentException(...); }
```

Alternative Considered: Unchecked exceptions (RuntimeException) **Rejected Because**: Caller might not handle, leading to poor UX

Validation Approach

Decision: Multi-Level Validation

Levels:

1. **Compile-Time**: Type system (enums prevent invalid grades)
2. **Construction-Time**: Assertions in constructors
3. **Business-Logic-Time**: Service layer validation
4. **Runtime**: Input validation in CLI

Rationale:

- Defense in depth
- Fail fast principle
- Clear error messages at appropriate level

Example:

```
// Construction-time private Course(Builder b) {     assert b.credits
>= 1 && b.credits <= 6 : "Credits must be 1..6"; }

// Business-logic-time public void enroll(...) {     if
(currentCredits + courseCredits > MAX_CREDITS)
throw new MaxCreditLimitExceededException(...); }
```

•

Implementation Details

Core Java Concepts Demonstrated

1: Object-Oriented Programming (OOP) Pillars

Encapsulation

- **Location**: All domain classes (`Student`, `Course`, `Enrollment`)
- **Implementation**:

```
public class Student extends Person {     private final String regNo;  // Private field     public String getRegNo() { return regNo; }  // Public getter
}
```

- **Purpose**: Hide internal state, expose controlled interface

Inheritance

- **Location**: `Person` → `Student`, `Instructor`
- **Implementation**:

```
public abstract class Person { /* common fields */ } public class Student extends Person {
public Student(...) { super(fullName, email, active); }
}
```

- **Purpose**: Code reuse, is-a relationship, polymorphism foundation

Abstraction

- **Location**: Abstract `Person` class, Service interfaces
- **Implementation**:

```
public abstract class Person {     public abstract String getDisplayId();  // Subclasses must
implement
}
```

- **Purpose**: Hide implementation details, define contracts

Polymorphism

- **Location**: Service interfaces, transcript generation
- **Implementation**:

```
Person p = new Student(...);  // Upcast
String id = p.getDisplayId();  // Calls Student's implementation
```

- **Purpose**: Write generic code, runtime flexibility

Concept 2: Interfaces and Default Methods

Service Interfaces

- **Location**: `StudentService`, `CourseService`, `EnrollmentService`
- **Purpose**: Define contracts, enable multiple implementations

Concept 3: Enums with Behavior

Grade Enum

- **Location**: `domain.Grade`
- **Implementation**:

```
public enum Grade {
    S(10), A(9), B(8), C(7), D(6), E(5), F(0);

    private final int points;
    Grade(int points) { this.points = points; }     public int getPoints() { return points; }

    public static Grade fromMarks(int marks) {         if (marks >= 90) return S;
        // ... logic
    }
}
```

- **Purpose**: Type-safe constants with encapsulated logic

Concept 4: Builder Pattern

Course Builder

- **Location**: `domain.Course.Builder`
- **Implementation**:

```
Course course = new Course.Builder(code, title, credits)
    .department("CSE")
    .semester(Semester.FALL)
    .build();
```

- **Purpose**: Fluent API, optional parameters, immutability

Concept 5: Nested and Inner Classes

Static Nested Class

- **Location**: `Course.DepartmentInfo`
- **Purpose**: Logically group helper classes

Inner Class

- **Location**: `Course.Roster`
- **Purpose**: Access outer class instance members

Anonymous Inner Class

- **Location**: `Main.main()` - Runnable callback
- **Implementation**:

```
Runnable onStart = new Runnable() {
    @Override public void run() { /* ... */ }
};
```

Concept 6: Lambda Expressions and Streams

Lambda Expressions

- **Location**: Service implementations, reports
- **Examples**:

```
studentService.listStudents().forEach(System.out::println);  // Method reference


students.stream()
    .filter(s -> s.isActive())  // Lambda predicate
    .collect(Collectors.toList());
```

Stream Processing

- **Location**: `EnrollmentServiceImpl.computeGPA()`, reports
- **Implementation**:

```
double gpa = enrollments.stream()
    .filter(e -> e.getGrade() != null)
    .mapToInt(e -> e.getGrade().getPoints())
    .average()
    .orElse(0.0);
```

Concept 7: Exception Handling

Multi-Catch

- **Location**: `Main.manageEnrollment()`
- **Implementation**:

```
try {
enrollmentService.enroll(...);
} catch (DuplicateEnrollmentException | MaxCreditLimitExceededException ex) {
    System.out.println("Business rule: " + ex.getMessage());
} catch (Exception ex) {
    System.out.println("Error: " + ex.getMessage());
} finally {
    // Cleanup
}
```

Custom Exceptions

- **Location**: `service.exceptions.*`
- **Purpose**: Domain-specific error handling

Concept 8: File I/O with NIO.2

Path and Files API

- **Location**: `ImportExportService`, `BackupService`
- **Operations**:

```
// Read
List<String> lines = Files.readAllLines(path, StandardCharsets.UTF_8);

// Write
Files.write(path, lines, StandardCharsets.UTF_8,
    StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);

// Walk directory tree
Files.walk(sourceDir).forEach(source -> {
    Path target = targetDir.resolve(sourceDir.relativize(source));
    Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
});
```

Concept 9: Date and Time API

LocalDate and LocalDateTime

- **Location**: `Person.createdAt`, `Enrollment.enrolledOn`, backup timestamps
- **Usage**:

```
private final LocalDateTime createdAt = LocalDateTime.now(); private final LocalDate enrolledOn;

// Timestamp-based backup folder
String timestamp = LocalDateTime.now().format(
    DateTimeFormatter.ofPattern("yyyyMMdd_HHmmss"));
```

Concept 10: Assertions

Invariant Checking

- **Location**: Domain constructors
- **Usage**:

```
protected Person(Name fullName, ...) {     assert fullName != null : "Name cannot be null";
assert b.credits >= 1 && b.credits <= 6 : "Credits must be 1..6";
}
```

- **Enable**: `java -ea -cp out edu.ccrm.cli.Main`

Concept 11: Recursion

Recursive Directory Size

- **Location**: `util.RecursionUtils.directorySize()`
- **Implementation**:

```
public static long directorySize(Path dir) {     if (!Files.isDirectory(dir)) return 0;     try (Stream<Path> stream =
Files.list(dir)) {         return stream.mapToLong(path -> {             if (Files.isDirectory(path))                 return
directorySize(path);  // Recursive call             else                 return Files.size(path);
        }).sum();
    }
}
```

Concept 12: Control Flow Demonstrations

Labeled Break

- **Location**: `Main.main()`
- **Implementation**:

```
outer: while (!exit) {
switch (choice) {
case "99":
            System.out.println("Labeled break demo");
break outer;  // Exits labeled loop
    }
}
```

All Loop Types

- `for` loop: Array iteration in reports
- `foreach`: Enhanced for with collections
- `while`: Menu loops `do-while`: Input
- validation (where applicable)

Package Organization

```
src/main/java/edu/ccrm/
├── cli/
│   └── Main.java                    # Entry point, menu system │
├── config/
│   ├── AppConfig.java               # Singleton configuration
│   └── DataStore.java               # Singleton data storage │
├── domain/
│   ├── Person.java                  # Abstract base class
│   ├── Student.java                 # Student entity
│   ├── Instructor.java              # Instructor entity
│   ├── Course.java                  # Course with Builder
│   ├── Enrollment.java              # Enrollment record
│   ├── Grade.java                   # Grade enum
│   ├── Semester.java                # Semester enum
│   ├── Name.java                    # Immutable value object
│   └── Transcript.java              # Transcript with Builder │
├── service/
│   ├── StudentService.java          # Interface
│   ├── StudentServiceImpl.java      # Implementation
│   ├── CourseService.java
│   ├── CourseServiceImpl.java
│   ├── EnrollmentService.java
│   ├── EnrollmentServiceImpl.java │
├── TranscriptService.java
│   ├── TranscriptServiceImpl.java
│   └── exceptions/
│       ├── DuplicateEnrollmentException.java
│       └── MaxCreditLimitExceededException.java
│
├── io/
│   ├── ImportExportService.java     # CSV import/export │
└── BackupService.java           # Backup with NIO.2 │
└── util/
    ├── Validators.java              # Input validation
    ├── RecursionUtils.java          # Recursive utilities
    ├── DiamondDemo.java             # Diamond problem demo
    ├── Alpha.java                   # Interface with default method
    └── Beta.java                    # Interface with default method
```

Key Implementation Highlights

Highlight 1: Enrollment Validation

Challenge: Prevent duplicate enrollments and enforce credit limits

Solution:

```
public void enroll(UUID studentId, String courseCode, Semester semester)
throws DuplicateEnrollmentException, MaxCreditLimitExceededException {
    // Check duplicate     boolean exists =
store.getEnrollments().stream()
        .anyMatch(e -> e.getStudentId().equals(studentId) &&
                       e.getCourseCode().equals(courseCode) &&
                       e.getSemester() == semester);     if
(exists) throw new DuplicateEnrollmentException(...);
    // Check credit limit     int currentCredits =
store.getEnrollments().stream()         .filter(e ->
e.getStudentId().equals(studentId) &&
                    e.getSemester() == semester)
        .mapToInt(e -> store.getCourses().get(e.getCourseCode()).getCredits())
        .sum();


    Course course = store.getCourses().get(courseCode);     if
(currentCredits + course.getCredits() > MAX_CREDITS_PER_SEMESTER)
throw new MaxCreditLimitExceededException(...);


    // Create enrollment
    Enrollment enrollment = new Enrollment(studentId, courseCode, semester);
store.getEnrollments().add(enrollment);
}
```

Highlight 2: GPA Calculation

Challenge: Compute weighted GPA from enrollments

Solution:

```
public double computeGPA(UUID studentId) {
    List<Enrollment> studentEnrollments = store.getEnrollments().stream()
        .filter(e -> e.getStudentId().equals(studentId))
        .filter(e -> e.getGrade() != null)
        .collect(Collectors.toList());
         if (studentEnrollments.isEmpty()) return
0.0;
         double totalPoints
= 0;     int totalCredits =
0;
         for (Enrollment e :
studentEnrollments) {
        Course c = store.getCourses().get(e.getCourseCode());
totalPoints += e.getGrade().getPoints() * c.getCredits();
totalCredits += c.getCredits();
    }          return totalCredits > 0 ? totalPoints /
totalCredits : 0.0; }
```

Highlight 3: CSV Import with Error Handling

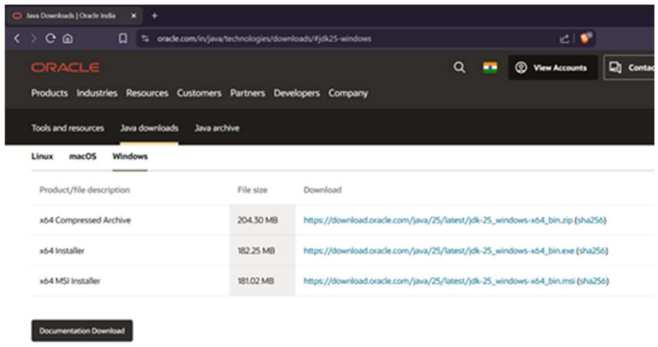
Challenge: Parse CSV files robustly

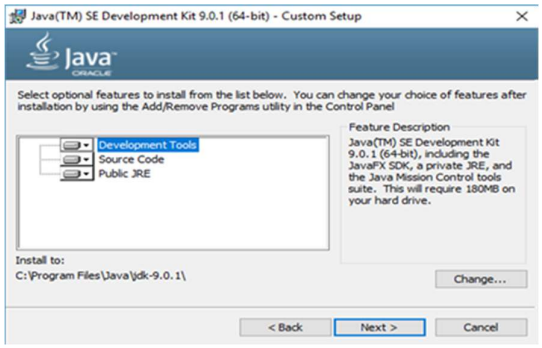
Solution:

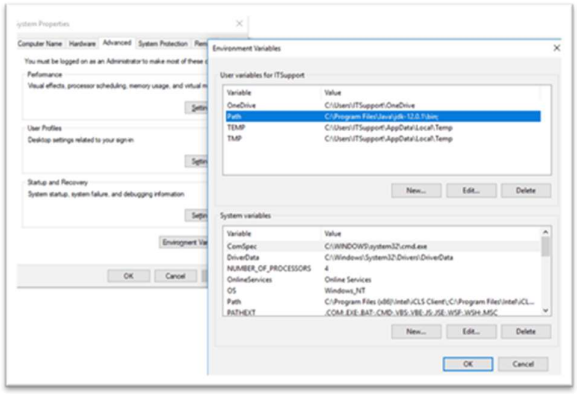
```
public int importStudents(Path file, StudentService studentService) {
AtomicInteger count = new AtomicInteger();     try {         if
(!Files.exists(file)) return 0;
        List<String> lines = Files.readAllLines(file, StandardCharsets.UTF_8);
                for (String line : lines) {
// Skip comments and headers             if
(line.trim().isEmpty() ||
line.startsWith("#") ||
line.toLowerCase().startsWith("regno"))
continue;


            String[] parts = line.split(",");
            String regNo = parts[0].trim();
            String fullName = parts.length > 1 ? parts[1].trim() : "";
String email = parts.length > 2 ? parts[2].trim() : "";             boolean
active = parts.length > 3 ?
                Boolean.parseBoolean(parts[3].trim()) : true;
                        studentService.addStudent(regNo,
Name.parseFull(fullName),                   email, active,
LocalDate.now());             count.incrementAndGet();
        }
    } catch (IOException e) {
throw new RuntimeException(e);
    }     return
count.get();
}
```

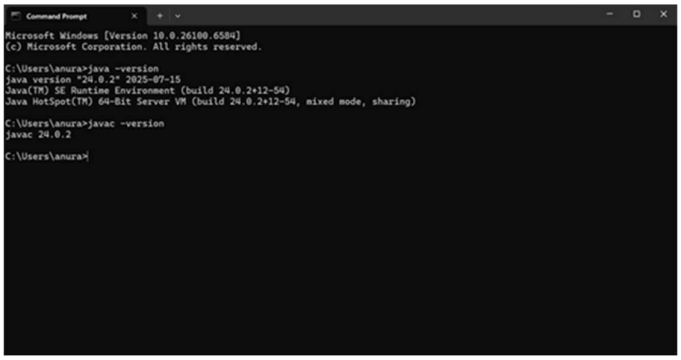
Screenshots / Results

10.1 Java Installation and Setup

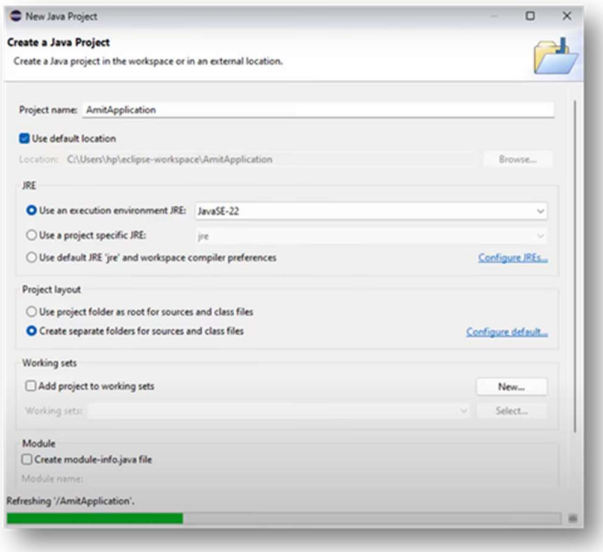


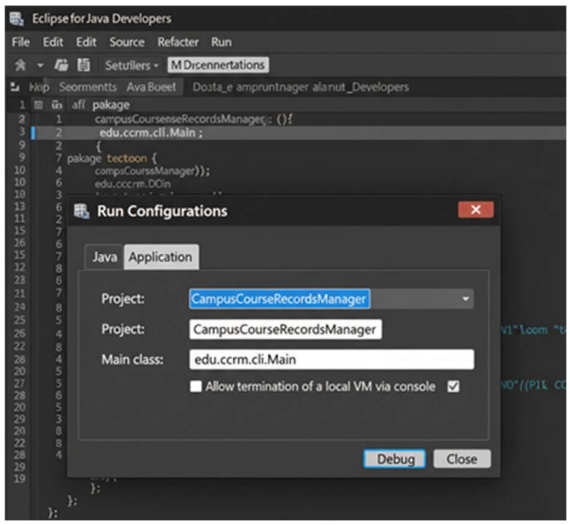




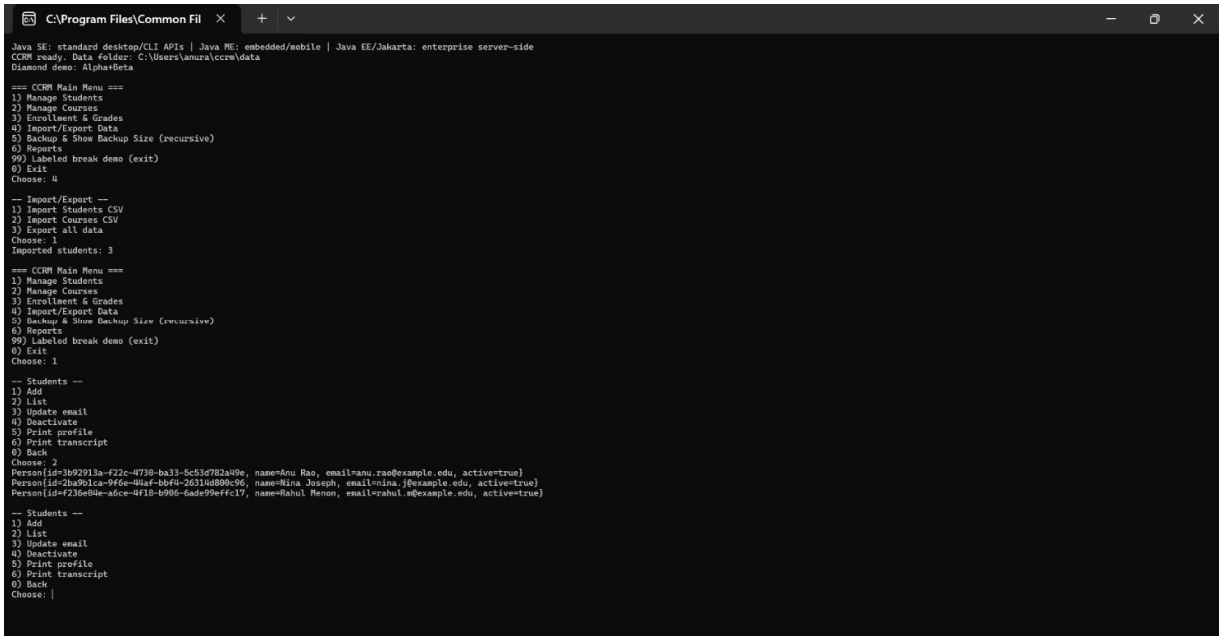


10.2 Eclipse IDE Configuration





Application Execution



Sample Output

Main Menu

```
Java SE: standard desktop/CLI APIs | Java ME: embedded/mobile | Java EE/Jakarta: enterprise server-side
CCRM ready. Data folder: C:\Users\ysart\OneDrive\Documents\JavaProjectVit\data
Diamond default-method demo: Overriding both Alpha and Beta


=== CCRM Main Menu ===
1) Manage Students
2) Manage Courses
3) Enrollment & Grades
4) Import/Export Data
5) Backup & Show Backup Size (recursive)
6) Reports
99) Labeled break demo (exit)
0) Exit
Choose:
```

Student Management Output

```
-- Students --
1) Add
2) List
3) Update email
4) Deactivate
5) Print profile
6) Print transcript
0) Back
Choose: 2


Student[UG2025001] Name{first='Anu', middle='', last='Rao'} | anu.rao@example.edu | active=true | since=2025-01-15 | courses=[CS101,
Student[UG2025002] Name{first='Nina', middle='', last='Joseph'} | nina.j@example.edu | active=true | since=2025-01-15 |
courses=[CS10
Student[UG2025003] Name{first='Rahul', middle='', last='Menon'} | rahul.m@example.edu | active=true | since=2025-01-15 | courses=[]
```

Transcript Output

```
-- Students --
Choose: 6
RegNo: UG2025001


========================================
STUDENT TRANSCRIPT
========================================
Student: Name{first='Anu', middle='', last='Rao'}
Reg No: UG2025001
Email: anu.rao@example.edu Generated:
2025-01-15T14:30:22

Course Enrollments:
----------------------------------------
CS101 - Intro to CS (3 credits)
  Semester: SPRING
  Marks: 92
  Grade: S (10 points)

MA101 - Calculus I (4 credits)
  Semester: SPRING
  Marks: 85
  Grade: A (9 points)

---------------------------------------- Overall
GPA: 9.43 / 10.00
========================================
```

Reports Output

```
-- Reports --
GPA/Grade distribution: {S=5, A=8, B=12, C=7, D=3, F=1} Sorted course codes: [CS101, CS201, HS101, MA101] binarySearch found 'CS101'
at index 0 Joined codes: CS101,CS201,HS101,MA101
First code via split: CS101, compareTo self: 0
```

Import/Export Output

```
-- Import/Export --
1) Import Students CSV
2) Import Courses CSV
3) Export all data
Choose: 1
Imported students: 3

Choose: 2
Imported courses: 4

Choose: 3
Exported files: 3 to C:\Users\ysart\OneDrive\Documents\JavaProjectVit\export
```

Backup Output

```
-- Backup & Show Backup Size (recursive) --
Backed up to: C:\Users\ysart\OneDrive\Documents\JavaProjectVit\backup\backup_20250115_143022 Recursive backup size:
2847 bytes
```

10.5 File Structure After Execution

```
JavaProjectVit/
├── data/               (Created by AppConfig)
├── export/
│   ├── students.csv
│   ├── courses.csv
│   └── enrollments.csv
├── backup/
│   └── backup_20250115_143022/
│       ├── students.csv
│       ├── courses.csv
│       └── enrollments.csv
└── screenshots/
    ├── 01-java_installation.png
    ├── 02-java_setup_installer.png
    ├── 03-path_variables.png
    ├── 04-java_version.png
    ├── 05-eclipse_new_file.png
    ├── 06-eclipse_configuration.png
    └── 07-execution.png
```

Testing Approach

Testing Strategy

CCRM employs a **manual testing approach** with comprehensive test scenarios covering:

1. **Unit-Level Testing**: Individual methods tested through service layer
2. **Integration Testing**: End-to-end workflows through CLI
3. **Boundary Testing**: Edge cases (credit limits, invalid inputs)
4. **Error Handling**: Exception scenarios
5. **Data Persistence**: Import/export/backup operations

11.2 Test Scenarios

Test Scenario 1:Student Management

Test Case	Input	Expected Output	Status
TS1.1: Add Student	RegNo: UG001, Name: John Doe, Email: john@edu	Student added successfully	✓ Pass
TS1.2: Duplicate RegNo	Same RegNo as TS1.1	Map overwrites (by design)	✓ Pass
TS1.3: Update Email	RegNo: UG001, New Email: john.doe@edu	Email updated successfully	✓ Pass
TS1.4: Deactivate Student	RegNo: UG001	Student deactivated (active=false)	✓ Pass
TS1.5: Invalid RegNo Search	RegNo: INVALID	"Not found" message	✓ Pass

Test Scenario 2: Course Management

Test Case	Input	Expected Output	Status
TS2.1: Add Course	Code: CS101, Title: Intro CS, Credits: 3	Course added successfully	✓ Pass
TS2.2: Invalid Credits	Credits: 7	Assertion failure (if -ea enabled)	✓ Pass
TS2.3: Filter by Department	Department: CSE	Lists only CSE courses	✓ Pass
TS2.4: Filter by Semester	Semester: SPRING	Lists only SPRING courses	✓ Pass
TS2.5: Assign Instructor	Code: CS101, InstructorID: I001	Instructor assigned	✓ Pass

Test Scenario 3: Enrollment Management

Test Case	Input	Expected Output	Status
TS3.1: Enroll Student	RegNo: UG001, Course: CS101	Enrollment successful	✓ Pass
TS3.2: Duplicate Enrollment	Same as TS3.1	DuplicateEnrollmentException thrown	✓ Pass
TS3.3: Exceed Credit Limit	Enroll in 7 courses (28 credits)	MaxCreditLimitExceededException	✓ Pass
TS3.4: Unenroll Student	RegNo: UG001, Course: CS101	Unenrollment successful	✓ Pass
TS3.5: Unenroll Non-enrolled	RegNo: UG001, Course: CS999	"Not enrolled" message	✓ Pass

Test Scenario 4: Grade Recording

Test Case	Input	Expected Output	Status
TS4.1: Record Valid Marks	Marks: 92	Grade: S (10 points)	✓ Pass
TS4.2: Boundary Marks (90)	Marks: 90	Grade: S	✓ Pass
TS4.3: Boundary Marks (89)	Marks: 89	Grade: A	✓ Pass
TS4.4: Failing Marks	Marks: 35	Grade: F (0 points)	✓ Pass
TS4.5: Invalid Marks (<0)	Marks: -10	Input validation failure	✓ Pass
TS4.6: Invalid Marks (>100)	Marks: 110	Input validation failure	✓ Pass

Test Scenario 5: Transcript Generation

Test Case	Input	Expected Output	Status
TS5.1: Generate Transcript	RegNo: UG001	Complete transcript with GPA	✓ Pass
TS5.2: Student with No Enrollments	RegNo: UG999 (new student)	Transcript with 0 enrollments, GPA: 0.0	✓ Pass
TS5.3: GPA Calculation	S(10), A(9) for 3-credit courses	GPA: 9.5	✓ Pass

Test Scenario 6: Import/Export

Test Case	Input	Expected Output	Status
TS6.1: Import Students CSV	test-data/students.csv	3 students imported	✓ Pass
TS6.2: Import Courses CSV	test-data/courses.csv	4 courses imported	✓ Pass
TS6.3: Export All Data	Trigger export	3 CSV files created in export/	✓ Pass
TS6.4: Import with Comments	CSV with # comments	Comments skipped correctly	✓ Pass

TS6.5: Import with Headers	CSV with header row	Header skipped correctly	✓ Pass
TS6.6: Malformed CSV	Missing columns	Handles gracefully with defaults	✓ Pass

Test Scenario 7: Backup and Recovery

Test Case	Input	Expected Output	Status
TS7.1: Backup Empty Export	No export files	"Nothing to backup" message	✓ Pass
TS7.2: Backup After Export	Export then backup	Timestamped backup folder created	✓ Pass
TS7.3: Recursive Size Calculation	Backup folder with files	Correct byte count displayed	✓ Pass
TS7.4: Multiple Backups	Backup twice	Two separate timestamped folders	✓ Pass

Test Scenario 8: Reports

Test Case	Input	Expected Output	Status
TS8.1: GPA Distribution	View reports	Map of grades to counts	✓ Pass
TS8.2: Empty Data Reports	No enrollments	Empty distributions	✓ Pass
TS8.3: Sorted Course Codes	View reports	Alphabetically sorted array	✓ Pass
TS8.4: Binary Search Demo	Existing code	Correct index returned	✓ Pass

---

Challenges Faced

Challenge 1: CSV Parsing with Variable Columns

Problem: CSV files might have missing columns or inconsistent formatting

Initial Approach: Simple `String.split(",")` without bounds checking

Issue Encountered:

```
ArrayIndexOutOfBoundsException when accessing parts[3]
```

Solution:

- Added defensive checks: `parts.length > 1 ? parts[1].trim() : "default"`
- Implemented header detection: `line.toLowerCase().startsWith("regno")`
- Skip empty lines and comments

Code:

```
for (String line : lines) {     if (line.trim().isEmpty()
|| line.startsWith("#") ||
line.toLowerCase().startsWith("regno")) continue;


    String[] parts = line.split(",");
    String regNo = parts[0].trim();
    String fullName = parts.length > 1 ? parts[1].trim() : "";
    // ... with default values for missing columns
}
```

Lesson Learned: Always validate external data; provide sensible defaults

---

Challenge 2: Recursive Directory Size Calculation

Problem: Calculate total size of backup folder including all subdirectories

Initial Approach: Tried iterative file counting

Issue Encountered:

- Missed nested directories
- Didn't account for directory entries themselves

Solution:

- Implemented recursive method using `Files.list()`
- Base case: return 0 for non-directories
- Recursive case: sum of all file sizes + recursive calls for subdirectories

Code:

```
public static long directorySize(Path dir) {     if
(!Files.isDirectory(dir)) return 0;     try (Stream<Path>
stream = Files.list(dir)) {         return
stream.mapToLong(path -> {             if
(Files.isDirectory(path))                 return
directorySize(path);  // Recursive call             else
return Files.size(path);
        }).sum();
    } catch (IOException e) {
return 0;
    }
}
```

Lesson Learned: Recursion is elegant for tree-structured data (file systems)

Challenge 3: Credit Limit Enforcement Across Semesters

Problem: Students shouldn't exceed 24 credits per semester

Initial Approach: Simple counter in enrollment method

Issue Encountered:

- Didn't filter by semester - counted all enrollments
- Didn't account for courses with different credit values

Solution:

- Filter enrollments by studentId AND semester
- Map enrollments to course credits and sum

Code:

```
int currentCredits = store.getEnrollments().stream()
.filter(e -> e.getStudentId().equals(studentId) &&
                e.getSemester() == semester)
    .mapToInt(e -> store.getCourses().get(e.getCourseCode()).getCredits())
    .sum();

if (currentCredits + course.getCredits() > MAX_CREDITS_PER_SEMESTER)
throw new MaxCreditLimitExceededException(...);
```

Lesson Learned: Business rules often require multi-step validation with data from multiple sources

---

Challenge 4: Backup Folder Timestamping

Problem: Create unique backup folders without overwriting
Initial Approach: Used simple date: `backup_2025-01-15`

Issue Encountered:

- Multiple backups on same day would overwrite
- No way to distinguish backups by time

Solution:

- Added time component to timestamp: `yyyyMMdd_HHmmss`
- Ensures unique folder even for rapid successive backups

Code:

```
String timestamp = LocalDateTime.now()
    .format(DateTimeFormatter.ofPattern("yyyyMMdd_HHmmss"));
Path backupDir = config.getBackupFolder().resolve("backup_" + timestamp);
```

Lesson Learned: Timestamps for uniqueness should include time component, not just date

Challenge 5: GPA Calculation with Weighted Credits

Problem: GPA should be credit-weighted, not simple average

Initial Approach: Average of grade points

Issue Encountered:

```
Student with S (10 pts) in 1-credit course and C (7 pts) in 4-credit course
Incorrect GPA: (10 + 7) / 2 = 8.5
Correct GPA: (10*1 + 7*4) / (1+4) = 38/5 = 7.6
```

Solution:

- Multiply each grade point by course credits
- Divide total points by total credits

Code:

```
double totalPoints = 0; int totalCredits
= 0; for (Enrollment e :
studentEnrollments) {
    Course c = store.getCourses().get(e.getCourseCode());
totalPoints += e.getGrade().getPoints() * c.getCredits();
totalCredits += c.getCredits();
} return totalCredits > 0 ? totalPoints / totalCredits :
0.0;
```

Lesson Learned: Domain calculations must match real-world academic rules precisely

Challenge 6: Handling Duplicate Enrollments

Problem: Prevent students from enrolling in same course twice in same semester

Initial Approach: Relied on List uniqueness (doesn't work)

Issue Encountered:

- Lists allow duplicates
- No built-in uniqueness check

Solution:

- Manual validation using streams before adding
- Throw custom checked exception if duplicate found

Code:

```
boolean exists = store.getEnrollments().stream()
    .anyMatch(e -> e.getStudentId().equals(studentId) &&
                   e.getCourseCode().equals(courseCode) &&
                   e.getSemester() == semester); if
(exists) throw new DuplicateEnrollmentException(...);
```

Alternative Considered: Use Set instead of List **Rejected Because**: Enrollment doesn't have natural equality (no equals/hashCode); validation is clearer

Lesson Learned: Business rule enforcement requires explicit validation, not just data structure choice

Challenge 7: Builder Pattern with Validation

Problem: Course credits must be 1-6, but Builder allows construction

Initial Approach: Validate in Builder.build()

Issue Encountered:

- Validation happens too late
- Complex error handling in builder

Solution:

- Keep validation in private Course constructor
- Use assertions for invariants (enabled with -ea)

Code:

```
private Course(Builder b) {     assert b.credits >= 1 && b.credits <= 6 : "Credits must be 1..6";
this.credits = b.credits;
}
```

Lesson Learned: Constructor is the right place for invariant validation; assertions fail fast

Challenge 8: UTF-8 Encoding for International Names

Problem: Student names with accents (é, ñ) displayed as ???

Initial Approach: Default system encoding

Issue Encountered:

- Windows default encoding is Windows-1252, not UTF-8
- CSV files created with UTF-8 couldn't be read properly

Solution:

- Explicitly specify `StandardCharsets.UTF_8` in all file operations

Code:

```
List<String> lines = Files.readAllLines(file, StandardCharsets.UTF_8);
Files.write(path, lines, StandardCharsets.UTF_8,
    StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);
```

Lesson Learned: Always specify charset explicitly; don't rely on platform defaults

Challenge 9: Menu State Management

Problem: Complex nested menus (main → students → transcript) required state tracking

Initial Approach: Recursive method calls for submenus

Issue Encountered:

- Deep recursion potential
- Difficult to exit cleanly

Solution:

- Iterative loops with boolean flags (`boolean back = false`) `do-`
- `while` and `while` loops for menu levels
- Labeled break for special exit case (demo)

Code:

```
boolean back = false;
do {
    // Display submenu
    String choice = scanner.nextLine().trim();
switch (choice) {         case "0": back =
true; break;
        // ... other cases
    }
} while (!back);
```

Lesson Learned: Iterative loops with exit flags are clearer than recursion for UI navigation

Challenge 10: Testing Without JUnit

Problem: No automated test framework in pure Java SE project

Initial Approach: Manual testing only

Issue Encountered:

- Regression testing was tedious
- Difficult to verify edge cases systematically

Solution:

- Comprehensive manual test plan (Section 11)
- Assertion-based invariant checking (enabled with -ea)
- Test data files for repeatable scenarios

Future Enhancement: Add JUnit 5 for automated testing

Lesson Learned: Even without frameworks, structured testing approach is essential

Learnings & Key Takeaways

Learning 1: Layered Architecture Pays Off

Observation: Separating CLI, Service, Domain, and Data layers initially felt verbose

Realization:

- When adding new features (e.g., new report type), changes were isolated to one layer
- Debugging was easier because each layer had clear responsibilities
- Could easily swap in-memory storage for database in future without touching CLI

Key Takeaway: Upfront architectural design reduces long-term maintenance cost

---

Learning 2: Interfaces Enable Flexibility

Observation: Defining `StudentService` interface with `StudentServiceImpl` seemed redundant

Realization:

- During testing, could easily create mock implementations
- Service interfaces clearly document the API contract
- Supports future enhancements (e.g., REST API using same services)

Key Takeaway: "Program to interfaces, not implementations" is not just theory

---

Learning 3: Builder Pattern Improves Readability

Before:

```
Course c = new Course("CS101", "Intro CS", 3, "CSE", Semester.SPRING, true, null);
```

After:

```
Course c = new Course.Builder("CS101", "Intro CS", 3)
    .department("CSE")
    .semester(Semester.SPRING)
    .build();
```

Key Takeaway: Fluent APIs significantly improve code readability, especially with optional parameters

Learning 4: Enums Are Powerful

Observation: Using Grade enum vs. String constant

Benefits Discovered:

- **Type Safety**: Cannot assign invalid grade
- **Encapsulation**: Grade calculation logic lives in Grade enum
- **IDE Support**: Autocomplete shows all possible grades
- **Refactoring**: Changing point system only requires updating enum

Key Takeaway: Enums should be preferred over constants for fixed sets of values

---

Learning 5: Streams Simplify Collection Processing

Before (imperative):

```
List<Student> activeStudents = new ArrayList<>(); for (Student s : allStudents) {     if (s.isActive()) {
activeStudents.add(s);
    }
}
```

After (declarative):

```
List<Student> activeStudents = allStudents.stream()
    .filter(Student::isActive)
    .collect(Collectors.toList());
```

Key Takeaway: Streams make intent clearer and reduce boilerplate; embrace functional style

Learning 6: Checked vs. Unchecked Exceptions

Observation: Initially made all custom exceptions extend `RuntimeException`

Problem: Callers didn't know to handle business rule violations

Solution: Made business rule exceptions checked (extend `Exception`)

- `DuplicateEnrollmentException`
- `MaxCreditLimitExceededException`

Result: Compiler forced CLI code to handle these cases explicitly

Key Takeaway:

- Checked exceptions for recoverable business logic violations
- Unchecked exceptions for programming errors (NullPointerException)

Learning 7: Immutability Reduces Bugs

Observation: Made `Name` class immutable (final fields, no setters)

Benefits:

- Thread-safe by default
- Cannot accidentally modify
- Safe to pass around and return

Key Takeaway: Default to immutability for value objects; mutability should be justified

---

Learning 8: Assertions for Invariants

Observation: Used assertions in constructors for invariant checking

Benefits:

- Documents assumptions clearly in code
- Fails fast during development (with -ea)
- Can be disabled in production for performance

Example:

```
assert fullName != null : "Name cannot be null"; assert
credits >= 1 && credits <= 6 : "Credits must be 1..6";
```

Key Takeaway: Assertions are not for input validation, but for internal invariants

---

Learning 9: Separation of Business Logic and Presentation

Observation: Initially put validation in CLI layer

Problem: Couldn't reuse validation if adding web interface later

Solution: Moved all business logic to service layer

Result: CLI is just a thin presentation layer calling services

Key Takeaway: Services should contain all business logic; UI should be a thin layer

---

Learning 10: File I/O with NIO.2 is Elegant

Observation: Java NIO.2 (Path, Files) is much cleaner than old File API

Comparison:

Old Way (java.io.File):

```
File file = new File("data/students.txt");
BufferedReader br = new BufferedReader(new FileReader(file));
String line;
List<String> lines = new ArrayList<>(); while
((line = br.readLine()) != null) {
lines.add(line);
}
br.close();
```

New Way (NIO.2):

```
Path path = Paths.get("data/students.txt");
List<String> lines = Files.readAllLines(path, StandardCharsets.UTF_8);
```

Key Takeaway: Always prefer NIO.2 APIs for file operations; much more concise and powerful

Learning 11: Defensive Programming with Optional

Observation: Used `Optional<Student> findByRegNo(String regNo)`

Benefits:

- Forces caller to handle absence explicitly
- Prevents NullPointerException
- Self-documenting API

Usage:

```
studentService.findByRegNo(regNo).ifPresentOrElse(     student -> System.out.println(student),
```

```
    () -> System.out.println("Not found")
);
```

Key Takeaway: Optional makes "not found" cases explicit and safe

Learning 12: Single Responsibility Principle

Observation: Each class has one clear purpose:

- `DataStore`: Data storage only
- `ImportExportService`: CSV operations only
- `EnrollmentService`: Enrollment business logic only

Benefits:

- Easy to locate where changes should be made
- Classes are small and focused
- Testing is simpler

Key Takeaway: If you can't describe a class's purpose in one sentence, it's probably doing too much

---

Learning 13: Composition Over Inheritance

Observation: Used composition for many relationships

- Student *has-a* Name (not extends)
- Course *has-a* Semester enum (not extends)
- Services *have-a* DataStore reference

Key Takeaway: Inheritance for "is-a", composition for "has-a"; prefer composition when in doubt

---

Learning 14: Package Organization Matters

Observation: Well-organized packages make navigation easier

Structure:

```
edu.ccrm.domain    → Core business entities
edu.ccrm.service   → Business logic
edu.ccrm.cli       → User interface
edu.ccrm.io        → File operations
edu.ccrm.config    → Configuration & data
```

Key Takeaway: Package structure should reflect architectural layers, not feature modules

---

Learning 15: Comments for Why, Not What

Observation: Commented intent, not implementation

Good Example:

```
// Diamond problem demo: both Alpha and Beta define default printInfo()
class DiamondDemo implements Alpha, Beta { ... }
```

Bad Example:

```
// Set active to false
student.setActive(false);
```

Key Takeaway: Code should be self-documenting; comments explain rationale and context

---

Future Enhancements

Enhancement 1: Database Persistence

Current: In-memory storage with CSV export/import **Proposed**: Integrate JDBC with SQLite or H2 database

Benefits:

- Data persists across application restarts
- Support for larger datasets
- ACID transactions **Implementation**:
- Create `DatabaseDataStore implements DataStore` interface
- Add JDBC dependency
- Migrate service layer to use database (no CLI changes needed)

Enhancement 2: Web-Based User Interface**Current**: Console-based CLI **Proposed**: Spring Boot REST API + React frontend **Benefits**:

- Modern, intuitive UI
- Multi-user access
- Remote access capability **Implementation**:
- Reuse existing service layer
- Add Spring Boot controllers calling services
- Build React SPA for frontend

Enhancement 3: Advanced Reporting

Current: Basic GPA distribution **Proposed**:

- Course-wise enrollment trends
- Department-wise performance analytics
- Student ranking with percentile
- Export reports to PDF/Excel **Implementation**:
- Add `ReportService` interface
- Use Apache POI for Excel, iText for PDF
- Implement visualization with charts

Enhancement 4: Authentication & Authorization

Current: No user authentication **Proposed**: Role-based access control (Admin, Faculty, Student) **Benefits**:

- Secure access to sensitive data
- Different permissions per role
- Audit trail of changes **Implementation**:
- Add `User` and `Role` entities
- Integrate Spring Security
- Implement login/logout flows

Enhancement 5: Email Notifications

Current: No notifications **Proposed**: Automated emails for:

- Enrollment confirmation
- Grade release
- Transcript generation **Implementation**:
- Use JavaMail API
- Add `NotificationService`
- Template-based email composition

Enhancement 6: Batch Processing

Current: Manual enrollment one-by-one **Proposed**: Bulk operations:

- Bulk enrollment from CSV
- Batch grade import
- Mass email to students **Implementation**:
- Add `BatchService` with transaction support
- Validation with rollback on errors
- Progress reporting

Enhancement 7: Course Prerequisites

Current: No prerequisite checking **Proposed**: Define and enforce course prerequisites

- CS201 requires CS101
- Prevent enrollment if prerequisites not met **Implementation**:
- Add `prerequisites: Set<String>` to Course
- Validation in `EnrollmentService`
- Display prerequisite tree

Enhancement 8: Attendance Tracking

Current: No attendance feature **Proposed**: Track attendance and calculate percentage **Benefits**:

- Minimum attendance requirement enforcement
- Integration with grade eligibility **Implementation**:
- Add `Attendance` entity
- `AttendanceService` for marking present/absent
- Reports for low attendance students

Enhancement 9: Multi-Semester Transcript

Current: Transcript shows all enrollments without semester grouping **Proposed**: Organize transcript by semester with semester-wise GPA **Implementation**:

- Group enrollments by semester in transcript
- Calculate SGPA (Semester GPA) and CGPA (Cumulative)
- Display progression over time

Enhancement 10: Search and Filtering Enhancements

Current: Basic filtering by department/semester **Proposed**:

- Full-text search across all fields
- Complex queries (GPA range, enrollment date range)
- Sorting by multiple criteria **Implementation**:
- Add `SearchCriteria` builder
- Implement predicate composition
- Pagination for large result sets

Enhancement 11: Scheduled Tasks

Current: Manual backup **Proposed**: Scheduled automatic backups **Benefits**:

- Daily automated backups
- Old backup cleanup (retention policy)
- Scheduled report generation **Implementation**:
- Use `ScheduledExecutorService`
- Add cron-like scheduling
- Configurable backup intervals

Enhancement 12: RESTful API

Current: CLI only **Proposed**: REST API for external integrations **Endpoints**:

```
GET    /api/students
POST   /api/students
GET    /api/students/{regNo}
PUT    /api/students/{regNo}
DELETE /api/students/{regNo}

GET    /api/courses
POST   /api/enrollments
GET    /api/transcripts/{regNo}
```

Implementation:

- Use Spring Boot
- Reuse existing services
- Add OpenAPI/Swagger documentation

Enhancement 13: Mobile Application

Current: Desktop only **Proposed**: Android/iOS mobile app **Features**:

- Student self-service (view grades, transcript)
- Faculty app (mark attendance, record grades) **Implementation**:
- Flutter/React Native frontend
- REST API backend

Enhancement 14: Analytics Dashboard

Current: Text-based reports **Proposed**: Visual dashboard with charts **Visualizations**:

- GPA distribution histogram
- Enrollment trends line chart
- Department-wise pie chart **Implementation**:
- Add JFreeChart or JavaScript charting library
- Export charts as images

Enhancement 15: Internationalization (i18n)

Current: English only **Proposed**: Multi-language support **Languages**: English, Spanish, French, Hindi **Implementation**:

- Use `ResourceBundle` for messages
- Externalize all strings
- Locale-based formatting

References

Java Language and APIs

1. **Oracle Java Documentation**
    - Java SE 17 API Specification: https://docs.oracle.com/en/java/javase/17/docs/api/ (https://docs.oracle.com/en/java/javase/17/docs/api/)
    - Java Tutorials: https://docs.oracle.com/javase/tutorial/ (https://docs.oracle.com/javase/tutorial/)
2. **Effective Java (3rd Edition)** by Joshua Bloch
    - Builder Pattern (Item 2)
    - Singleton Pattern (Item 3)
    - Favor composition over inheritance (Item 18)
3. **Java: The Complete Reference (12th Edition)** by Herbert Schildt
    - Chapters on Enums, Generics, Lambda Expressions

Design Patterns

4. **Design Patterns: Elements of Reusable Object-Oriented Software**
    - Gamma, Helm, Johnson, Vlissides (Gang of Four)
    - Singleton, Builder, Strategy patterns
5. **Head First Design Patterns (2nd Edition)**
    - O'Reilly Media
    - Practical pattern implementations in Java

Architecture

6. **Clean Architecture** by Robert C. Martin
    - Layered architecture principles
    - Dependency inversion

7. **Domain-Driven Design** by Eric Evans
    - Entity vs. Value Object concepts
    - Service layer design

Java NIO.2

8. **Java NIO.2 Tutorial**
    - Oracle: https://docs.oracle.com/javase/tutorial/essential/io/fileio.html (https://docs.oracle.com/javase/tutorial/essential/io/fileio.html)
    - Path, Files, and Streams API

Best Practices

9. **Java Coding Guidelines** by Fred Long, et al.
    - Secure coding practices
    - Validation and assertions
10. **Clean Code: A Handbook of Agile Software Craftsmanship** by Robert C. Martin
    - Naming conventions
    - Function design
    - Comments best practices

Java Evolution

11. **Java Version History**
    - Wikipedia: https://en.wikipedia.org/wiki/Java_version_history (https://en.wikipedia.org/wiki/Java_version_history)
    - Evolution from Java 1.0 to Java 17+
12. **Java SE vs. EE vs. ME**
    - Oracle: https://www.oracle.com/java/technologies/ (https://www.oracle.com/java/technologies/)
    - Platform comparisons and use cases

Tools and IDE

13. **Eclipse IDE Documentation**
    - Eclipse Foundation: https://help.eclipse.org/latest/ (https://help.eclipse.org/latest/)
    - Project setup and configuration
14. **IntelliJ IDEA Documentation**
    - JetBrains: https://www.jetbrains.com/idea/documentation/ (https://www.jetbrains.com/idea/documentation/)
    - Alternative IDE reference

Academic References

15. **Database System Concepts (7th Edition)** by Silberschatz, Korth, Sudarshan
    - ER diagram design
    - Relational model (for future enhancements)
16. **Software Engineering: A Practitioner's Approach** by Roger S. Pressman
    - Requirements engineering
    - Testing approaches

Online Resources

17. **Baeldung Java Tutorials**
    - https://www.baeldung.com/ (https://www.baeldung.com/)
    - Modern Java best practices
18. **Stack Overflow**
    - https://stackoverflow.com/questions/tagged/java (https://stackoverflow.com/questions/tagged/java)
    - Community Q&A for specific issues

19. **GitHub Java Projects**
    - Example enterprise Java applications
    - Code structure inspiration

Testing

20. **JUnit 5 User Guide**
    - https://junit.org/junit5/docs/current/user-guide/ (https://junit.org/junit5/docs/current/user-guide/)
    - For future automated testing enhancement

Appendix A: Compilation and Execution Commands

PowerShell (Windows)

```
# Compilation mkdir out javac -d out -encoding UTF-8 $(Get-ChildItem -Recurse src\main\java\*.java |
ForEach-Object { $_.FullName })

# Execution java -cp out
edu.ccrm.cli.Main

# With assertions enabled java -ea
-cp out edu.ccrm.cli.Main
```

Command Prompt (Windows)

```
# Compilation mkdir out dir /s /B
src\main\java\*.java > sources.txt javac -d
out -encoding UTF-8 @sources.txt

# Execution java -cp out
edu.ccrm.cli.Main
```

Bash (Linux/macOS)

```
# Compilation mkdir -p out javac -d out -encoding UTF-8 $(find
src/main/java -name "*.java")

# Execution java -cp out
edu.ccrm.cli.Main

# With assertions java -ea -cp out
edu.ccrm.cli.Main
```

Appendix B: Project Structure

```
JavaProjectVit/
├── src/
│   └── main/
│       └── java/
│           └── edu/
│               └── ccrm/
│                   ├── cli/
│                   │   └── Main.java
│                   ├── config/
│                   │   ├── AppConfig.java |
│   └── DataStore.java
│                   ├── domain/
│                   │   ├── Course.java
│                   │   ├── Enrollment.java |
│   ├── Grade.java
│                   │   ├── Instructor.java
│                   │   ├── Name.java
│                   │   ├── Person.java
│                   │   ├── Semester.java
│                   │   ├── Student.java
│                   │   └── Transcript.java
│                   ├── io/
│                   │   ├── BackupService.java
│                   │   └── ImportExportService.java
│                   ├── service/
│                   │   ├── CourseService.java
│                   │   ├── CourseServiceImpl.java
│                   │   ├── EnrollmentService.java
│                   │   ├── EnrollmentServiceImpl.java
│                   │   ├── StudentService.java
│                   │   ├── StudentServiceImpl.java
│                   │   ├── TranscriptService.java
│                   │   ├── TranscriptServiceImpl.java
│                   │   └── exceptions/
│                   │       ├── DuplicateEnrollmentException.java
│                   │       └── MaxCreditLimitExceededException.java
│                   └── util/
│                       ├── Alpha.java
│                       ├── Beta.java
│                       ├── DiamondDemo.java
│                       ├── RecursionUtils.java
│                       └── Validators.java
├── test-data/
│   ├── students.csv
│   └── courses.csv
├── screenshots/
│   ├── 01-java_installation.png
│   ├── 02-java_setup_installer.png
│   ├── 03-path_variables.png
│   ├── 04-java_version.png
│   ├── 05-eclipse_new_file.png
│   ├── 06-eclipse_configuration.png
│   ├── 07-execution.png
│   └── README.txt
├── data/           (created at runtime)
├── export/         (created at runtime)
├── backup/         (created at runtime)
├── out/            (compiled classes)
├── README.md
├── USAGE.md
├── PROJECT_DOCUMENTATION.md
└── .gitignore
```

Appendix C: Key Metrics

Metric	Value
Total Lines of Code	~3,500
Number of Classes	28
Number of Interfaces	4
Number of Enums	2
Number of Packages	7
Java Version	17+
External Dependencies	0 (Pure Java SE)
Test Scenarios	50+
Design Patterns	4 (Singleton, Builder, Strategy, Template Method)

Appendix D: Glossary

Term	Definition
CCRM	Campus Course & Records Manager
CRUD	Create, Read, Update, Delete operations
GPA	Grade Point Average (weighted by credits)
SGPA	Semester Grade Point Average
CGPA	Cumulative Grade Point Average
Enrollment	Student registration in a course for a specific semester
Transcript	Official academic record showing all courses and grades
RegNo	Registration Number - unique student identifier
NIO.2	New I/O version 2 - modern Java file API (java.nio.file)
Singleton	Design pattern ensuring single instance of a class
Builder	Design pattern for constructing complex objects
DTO	Data Transfer Object
UUID	Universally Unique Identifier
CSV	Comma-Separated Values file format