System Design Document (SDD)

Student Management System

Document Version: 1.0

Date: December 2025

---

1. Introduction

1.1 Purpose

This document describes the system architecture, design patterns, and component interactions for the Student Management System.

1.2 Scope

The design encompasses all modules, classes, and relationships within the console-based Student Management System.

---

2. System Architecture

2.1 Architecture Style

The system follows a **Layered Architecture** pattern with three primary layers:

```
┌─────────────────────────────────────┐
│     Presentation Layer (CLI)        │
│  StudentManagementSystemApp.java    │
├─────────────────────────────────────┤
│        Business Logic Layer         │
│    StudentManager.java (Service)    │
├─────────────────────────────────────┤
│          Data Model Layer           │
│   Student.java, Course.java         │
└─────────────────────────────────────┘
         Utilities (InputValidator)
```

2.2 Component Overview

Presentation Layer:

- Handles user interaction through console interface
- Menu display and navigation
- Input collection and output formatting

Business Logic Layer:

- Implements CRUD operations
- Business rules enforcement
- Data validation and processing

Data Model Layer:

- Encapsulated entity classes
- Data structures and relationships
- Domain logic

---

3. Class Design

3.1 Student Class

Package: `com.studentmanagement.model`

Purpose: Represents a student entity with encapsulation

Attributes:

```
- studentId: String (unique identifier)
- firstName: String
- lastName: String
- email: String
- age: int
- courses: List<Course>
- gpa: double (calculated)
```

Methods:

```
+ Student(studentId, firstName, lastName, email, age)
+ getStudentId(): String
+ getFirstName(): String
+ getLastName(): String
+ getEmail(): String
+ getAge(): int
+ getCourses(): List<Course>
+ getGpa(): double
+ setFirstName(String): void
+ setLastName(String): void
+ setEmail(String): void
+ setAge(int): void
+ addCourse(Course): void
+ removeCourse(String): void
+ calculateGPA(): void
+ toString(): String
```

Design Principles Applied:

- **Encapsulation**: All fields are private with public getters/setters
- **Data Protection**: getCourses() returns a copy to prevent external modification
- **Validation**: Setter methods validate input before assignment
- **Automatic Calculation**: GPA is recalculated when courses change

3.2 Course Class

Package: `com.studentmanagement.model`

Purpose: Represents a course with grade information

Attributes:

```
- courseCode: String (unique identifier)
- courseName: String
- credits: int
- grade: double (0-100 scale)
```

Methods:

```
+ Course(courseCode, courseName, credits, grade)
+ getCourseCode(): String
+ getCourseName(): String
+ getCredits(): int
+ getGrade(): double
+ setCourseName(String): void
+ setCredits(int): void
+ setGrade(double): void
+ getGradePoint(): double
+ getLetterGrade(): String
+ toString(): String
+ equals(Object): boolean
+ hashCode(): int
```

Design Principles Applied:

- **Value Object**: Courses are compared by courseCode
- **Business Logic**: Grade conversion logic encapsulated in the class
- **Validation**: Input validation in setters

3.3 StudentManager Class

Package: `com.studentmanagement.service`

Purpose: Manages CRUD operations and business logic for students

Attributes:

```
- students: Map<String, Student>
- nextId: int (for ID generation)
```

Methods:

```
+ StudentManager()
- generateUniqueId(): String
+ createStudent(firstName, lastName, email, age): String
+ getStudent(studentId): Student
+ updateStudent(studentId, field, value): void
+ deleteStudent(studentId): boolean
+ searchStudents(searchTerm): List<Student>
+ assignCourse(studentId, courseCode, courseName, credits, grade): void
+ removeCourse(studentId, courseCode): void
+ getAllStudents(): List<Student>
+ getTotalStudents(): int
+ studentExists(studentId): boolean
+ getStudentsByMinGPA(minGPA): List<Student>
```

Design Principles Applied:

- **Single Responsibility**: Focuses solely on student management
- **Factory Pattern**: Generates unique IDs for students
- **Collection Management**: Uses HashMap for efficient lookups
- **Stream API**: Uses Java 8 streams for filtering and searching

3.4 InputValidator Class

Package: `com.studentmanagement.util`

Purpose: Centralized input validation logic

Methods:

```
+ isValidEmail(String): boolean
+ isValidAge(int): boolean
+ isValidName(String): boolean
+ isValidCourseCode(String): boolean
+ isValidGrade(double): boolean
+ isValidCredits(int): boolean
+ isNotEmpty(String): boolean
```

Design Principles Applied:

- **Utility Class**: Static methods for validation
- **Regular Expressions**: Pattern matching for format validation
- **Separation of Concerns**: Validation logic separated from business logic

3.5 StudentManagementSystemApp Class

Package: `com.studentmanagement`

Purpose: Main application class with CLI interface

Attributes:

```
- ADMIN_USERNAME: String (constant)
- ADMIN_PASSWORD: String (constant)
- MAX_LOGIN_ATTEMPTS: int (constant)
- studentManager: StudentManager
- scanner: Scanner
```

Methods:

```
+ main(String[]): void
+ run(): void
- printWelcomeBanner(): void
- performLogin(): boolean
- showMainMenu(): void
- printMainMenu(): void
- createStudent(): void
- viewAllStudents(): void
- viewStudent(): void
- updateStudent(): void
- deleteStudent(): void
- searchStudents(): void
- assignCourse(): void
- removeCourse(): void
- viewStatistics(): void
- getIntInput(String): int
- getDoubleInput(String): double
- pressEnterToContinue(): void
```

Design Principles Applied:

- **Controller Pattern**: Coordinates between user and business logic
- **Menu-Driven Interface**: Clear navigation structure
- **Error Handling**: Try-catch blocks throughout
- **Input Validation**: Validates before processing

---

4. Class Relationships

4.1 UML Class Diagram (Text Representation)

```
┌──────────────────────────────────┐
│  StudentManagementSystemApp      │
├──────────────────────────────────┤
│ - studentManager: StudentManager │
│ - scanner: Scanner               │
├──────────────────────────────────┤
│ + main(String[]): void           │
│ + run(): void                    │
│ - showMainMenu(): void           │
└────────────┬─────────────────────┘
             │ uses
             ▼
┌──────────────────────────────────┐
│      StudentManager              │
├──────────────────────────────────┤
│ - students: Map<String, Student> │
│ - nextId: int                    │
├──────────────────────────────────┤
│ + createStudent(...): String     │
│ + getStudent(String): Student    │
│ + updateStudent(...): void       │
│ + deleteStudent(String): boolean │
│ + searchStudents(String): List   │
└────────────┬─────────────────────┘
             │ manages
             ▼
┌──────────────────────────────────┐
│         Student                  │
├──────────────────────────────────┤
│ - studentId: String              │
│ - firstName: String              │
│ - lastName: String               │
│ - email: String                  │
│ - age: int                       │
│ - courses: List<Course>          │
│ - gpa: double                    │
├──────────────────────────────────┤
│ + addCourse(Course): void        │
│ + removeCourse(String): void     │
│ + calculateGPA(): void           │
└────────────┬─────────────────────┘
             │ has many
             ▼
┌──────────────────────────────────┐
│         Course                   │
├──────────────────────────────────┤
│ - courseCode: String             │
│ - courseName: String             │
│ - credits: int                   │
│ - grade: double                  │
├──────────────────────────────────┤
│ + getGradePoint(): double        │
│ + getLetterGrade(): String       │
```

```
└───────────────────────────────────┘


┌───────────────────────────────────┐
│      InputValidator               │
├───────────────────────────────────┤
│ + isValidEmail(String): boolean   │
│ + isValidAge(int): boolean        │
│ + isValidName(String): boolean    │
│ + isValidCourseCode(String): bool│
└───────────────────────────────────┘

        ▲
        │ uses
        └───── StudentManagementSystemApp
```

4.2 Relationships

- **StudentManagementSystemApp** → **StudentManager**: Composition (1:1)
- **StudentManagementSystemApp** → **InputValidator**: Dependency
- **StudentManager** → **Student**: Aggregation (1:Many)
- **Student** → **Course**: Composition (1:Many)

---

5. Design Patterns

5.1 Factory Pattern

Location: StudentManager.generateUniqueId() **Purpose**: Centralized ID generation ensures uniqueness **Implementation**: Sequential ID generation with collision checking

5.2 Data Access Object (DAO) Pattern

Location: StudentManager **Purpose**: Abstracts data access and manipulation **Implementation**: Provides CRUD operations interface

5.3 Model-View-Controller (MVC) Variant

Model: Student, Course classes **View**: Console output formatting in main app **Controller**: StudentManagementSystemApp, StudentManager

5.4 Singleton Consideration

Note: While not implemented, Scanner could be singleton **Reason**: Current design is sufficient for single-threaded application

---

6. Data Structures

6.1 Student Storage

Structure: `HashMap<String, Student>` **Reason**: O(1) average lookup time by student ID **Key**: Student ID (unique) **Value**: Student object

6.2 Course Storage

Structure: `ArrayList<Course>` (within Student) **Reason**: Ordered list, allows duplicates (same course retaken) **Operations**: Add, remove, iterate

---

7. Algorithms

7.1 GPA Calculation

```
Algorithm: calculateGPA()
Input: List of courses with grades and credits
Output: GPA on 4.0 scale

1. Initialize totalGradePoints = 0
2. Initialize totalCredits = 0
3. For each course in courses:
   a. Convert percentage grade to grade point (4.0 scale)
   b. totalGradePoints += gradePoint × credits
   c. totalCredits += credits
4. If totalCredits > 0:
   GPA = totalGradePoints / totalCredits
5. Else:
   GPA = 0.0
6. Return GPA
```

7.2 Search Algorithm

```
Algorithm: searchStudents(searchTerm)
Input: Search term (string)
Output: List of matching students

1. Convert searchTerm to lowercase
2. Create empty result list
3. For each student in students:
   a. Check if searchTerm matches:
      - Student ID (case-insensitive)
      - First name (case-insensitive)
      - Last name (case-insensitive)
      - Email (case-insensitive)
   b. If match found, add to result list
4. Return result list
```

7.3 Unique ID Generation

```
Algorithm: generateUniqueId()
Output: Unique student ID
```

```
1. Generate ID = "STU" + nextId
2. Increment nextId
3. While ID exists in students map:
   a. Generate ID = "STU" + nextId
   b. Increment nextId
4. Return ID
```

8. Error Handling Strategy

8.1 Exception Hierarchy

```
Exception (Java built-in)
├── IllegalArgumentException (validation errors)
├── NumberFormatException (input parsing errors)
└── General Exception (catch-all)
```

8.2 Error Handling Levels

Layer 1: Input Validation

- Validate before processing
- Use InputValidator utility
- Prompt user to retry

Layer 2: Business Logic

- Throw IllegalArgumentException for business rule violations
- Validate state before operations
- Provide descriptive error messages

Layer 3: Presentation

- Catch all exceptions in CLI methods
- Display user-friendly error messages
- Allow operation retry

8.3 Try-Catch Usage

```java
try {
    // Operation code
    // Input parsing
    // Business logic call
} catch (NumberFormatException e) {
    // Handle numeric input errors
    System.out.println("✗ Invalid number format");
} catch (IllegalArgumentException e) {
    // Handle validation errors
    System.out.println("✗ " + e.getMessage());
} catch (Exception e) {
    // Handle unexpected errors
    System.out.println("✗ Error: " + e.getMessage());
}
```

---

9. Security Considerations

9.1 Authentication

- Simple username/password check
- Limited login attempts (3)
- Plain text storage (educational purposes only)

9.2 Input Validation

- All inputs validated before processing
- Regex patterns for format validation
- Range checks for numeric inputs

9.3 Data Protection

- Encapsulation prevents direct field access
- Defensive copying for collections
- Immutable student ID

---

10. Performance Considerations

10.1 Time Complexity

- Student lookup by ID: O(1) - HashMap
- Search students: O(n) - Linear scan
- Add/Remove course: O(1) - ArrayList operations
- Calculate GPA: O(n) - Where n is number of courses

10.2 Space Complexity

- Student storage: O(n) - Where n is number of students
- Course storage per student: O(m) - Where m is courses per student
- Overall: O(n × m)

10.3 Optimization Opportunities

- Index students by name for faster search
- Cache GPA calculation results
- Implement pagination for large student lists

---

11. Scalability Considerations

11.1 Current Limitations

- In-memory storage (data lost on exit)
- Single user access
- No concurrent operations

11.2 Future Enhancements

- Database integration (MySQL, PostgreSQL)
- Multi-user support with threading
- RESTful API for web/mobile clients
- Caching layer (Redis)

---

12. Testing Strategy

12.1 Unit Testing

- Test each class method independently
- Mock dependencies (e.g., Scanner)
- Validate edge cases

12.2 Integration Testing

- Test class interactions
- Validate data flow between layers
- Test CRUD operation sequences

12.3 System Testing

- End-to-end user scenarios
- Menu navigation testing
- Error handling validation

---

13. Deployment Architecture

```
┌─────────────────────────────────┐
│   User's Computer/Server        │
│                                 │
│  ┌──────────────────────────┐   │
│  │  Java Runtime (JRE 8+)   │   │
│  │                          │   │
│  │  ┌─────────────────────┐ │   │
│  │  │ Student Management  │ │   │
│  │  │ System Application  │ │   │
│  │  └─────────────────────┘ │   │
│  │                          │   │
│  └──────────────────────────┘   │
│                 ▲                │
│                 │                │
│         Console I/O              │
└──────────────────────────────────┘
```

13.1 Deployment Requirements

- Java Development Kit (JDK) 8 or higher
- Console/Terminal access
- No external dependencies or libraries

13.2 Deployment Steps

1. Copy source files to target system
2. Compile Java files: `javac *.java`
3. Run application: `java StudentManagementSystemApp`

---

14. Maintenance Guidelines

14.1 Code Organization

- Package structure clearly defined
- One class per file
- Meaningful naming conventions

14.2 Documentation Standards

- Javadoc comments for all public methods
- Inline comments for complex logic
- README with usage instructions

14.3 Version Control

- Use Git for source control
- Feature branches for new features
- Meaningful commit messages

Document Prepared By: Development Team
Approved By: Technical Lead
Review Date: December 2025