Project Proposal:
College Course Enrollment System

Prepared by:

Shaotian Li, Xiaowei Feng

2024/10/22

Contents

1 Problem Statement

Description of the Problem

Enrolling in courses is a fundamental part of the college experience, yet many students face challenges navigating complex course catalogs, understanding prerequisites, and ensuring they meet graduation requirements. Traditional enrollment systems can be cumbersome, unintuitive, and may not provide real-time feedback on prerequisite fulfillment or course availability.

Importance and Relevance

An efficient and user-friendly course enrollment system is crucial for students to plan their academic journey effectively. With increasing course options and diverse program requirements, students need a platform that simplifies the enrollment process, reduces administrative bottlenecks, and minimizes the risk of enrollment errors that could delay graduation.

Specific Challenges Addressed

- **Complex Navigation:** Simplifying the browsing of extensive course catalogs.
- **Prerequisite Confusion:** Providing clear prerequisite checks before enrollment.
- **Inefficient Search:** Enhancing search and filter functions to find courses quickly.
- **User Authentication:** Securing student information through a reliable login system.
- **Information Overload:** Presenting course descriptions and requirements in an accessible manner.

2 System Features

Functional Requirements

1. **User Authentication and Profile Management**
    - Secure login and logout functionality.
    - Student profile page displaying personal information and enrolled courses.
2. **Course Catalog Browsing**
    - Display all available courses with essential details.
    - Organized categorization by departments, programs, and semesters.
3. **Advanced Search and Filter Functions**
    - Search courses by keywords, course codes, instructors, or schedules.
    - Filter courses based on credits, prerequisites, availability, and level.
4. **Course Details and Descriptions**
    - Detailed course pages with descriptions, objectives, syllabus, and instructor information.
    - Visibility of course schedules, locations, and seat availability.

5. **Prerequisite Verification**
    - Automated checks to verify if students meet course prerequisites.
    - Notifications for missing prerequisites with suggestions for fulfillment.
6. **Enrollment Management**
    - Add or drop courses from the schedule.
    - Real-time updates on enrollment status and waitlist positions.

Non-Functional Requirements

1. **Usability**
    - Intuitive user interface with easy navigation.
    - Accessible design accommodating various user needs.
2. **Scalability**
    - Ability to handle a growing number of users and courses without performance degradation.
3. **Security**
    - Encryption of sensitive data.
    - Compliance with data protection regulations.
4. **Performance**
    - Fast load times and responsive interactions.
    - Real-time data updates without significant latency.
5. **Reliability**
    - System availability with minimal downtime.
    - Robust error handling and user feedback mechanisms.

3 Design Approach

Object-Oriented Principles Application

Encapsulation: Each class will manage its own data and expose functionalities through public methods, hiding internal implementations.

Inheritance: Base classes will be created for shared attributes, allowing derived classes to inherit and extend functionalities.

Polymorphism: Interfaces and abstract classes will enable different objects to be treated uniformly based on shared behaviors.

System Structure and Key Classes

1. **User Class (Base Class)**
   - **Attributes:** (int)userID, (string)name, (string)email, (string)password.
   - **Methods:** login(name, email), logout(), viewProfile().
2. **Student Class (Derived from User)**
   - **Attributes:** (course[])enrolledCourses, (course[])completedCourses, (string)major
   - **Methods:** enrollCourse(course), dropCourse(course), checkPrerequisite(course)
3. **Course Class**
   - **Attributes:** (int)courseID, (string)courseName, (string)description, (float)credits, (course[])prerequisites, (student[])waitlist, (vector<string >)schedule, (string)instructor.
   - **Methods:** getDetails(), checkAvailability(course).
4. **Instructor Class (Derived from User)**
   - **Attributes:** (course[])coursesTaught, (string)department.
   - **Methods:** updateCourseInfo(courseID, courseName, description, prerequisites), viewEnrolledStudents().
5. **EnrollmentManager Class**
   - **Attributes:** (vector<string>)enrollmentRecords.
   - **Methods:** addEnrollment(studentID, courseID), removeEnrollment(studentID, courseID), verifyPrerequisites(studentID, courseID), getEnrollmentStatus(studentID, courseID)
6. **SearchEngine Class**
   - **Attributes:** (struct)searchCriteria, map <int, string>courseIndex
   - **Methods:** searchCourses(keywords), filterResults(filters, courseList), sortResults(sortOption, courseList)

   struct SearchCriteria string keyword vector <string>courseCodes vector<string>courseNames vector<string>programs
7. **PrerequisiteChecker Class**
   - **Attributes:** (map<string, string[]>)courseRequirements, studentRecord
   - **Methods:** validatePrerequisites(student, courseID)

Interactions Between Classes

- The **Student** interacts with the **Course** through the **EnrollmentManager**, which handles enrollment processes and prerequisite verification using the **PrerequisiteChecker**.
- The **SearchEngine** allows **Student** objects to search and filter **Course** objects based on various criteria.
- **Instructor** can update course information, which reflects in the **Course** class instances.
- The **EnrollmentManager** maintains records of enrollments and provides real-time updates to the **Student**'s profile.

Logical and Effective Structure Through OOP

By leveraging object-oriented concepts, the system's structure promotes code reusability, scalability, and maintainability. Encapsulation ensures that each class manages its responsibilities, reducing interdependencies. Inheritance allows for a hierarchical organization of user types, and polymorphism provides flexibility in handling different objects through common interfaces.

This design approach sets a solid foundation for the system, allowing for future enhancements and integration of additional features without significant restructuring.

4 Diagrams

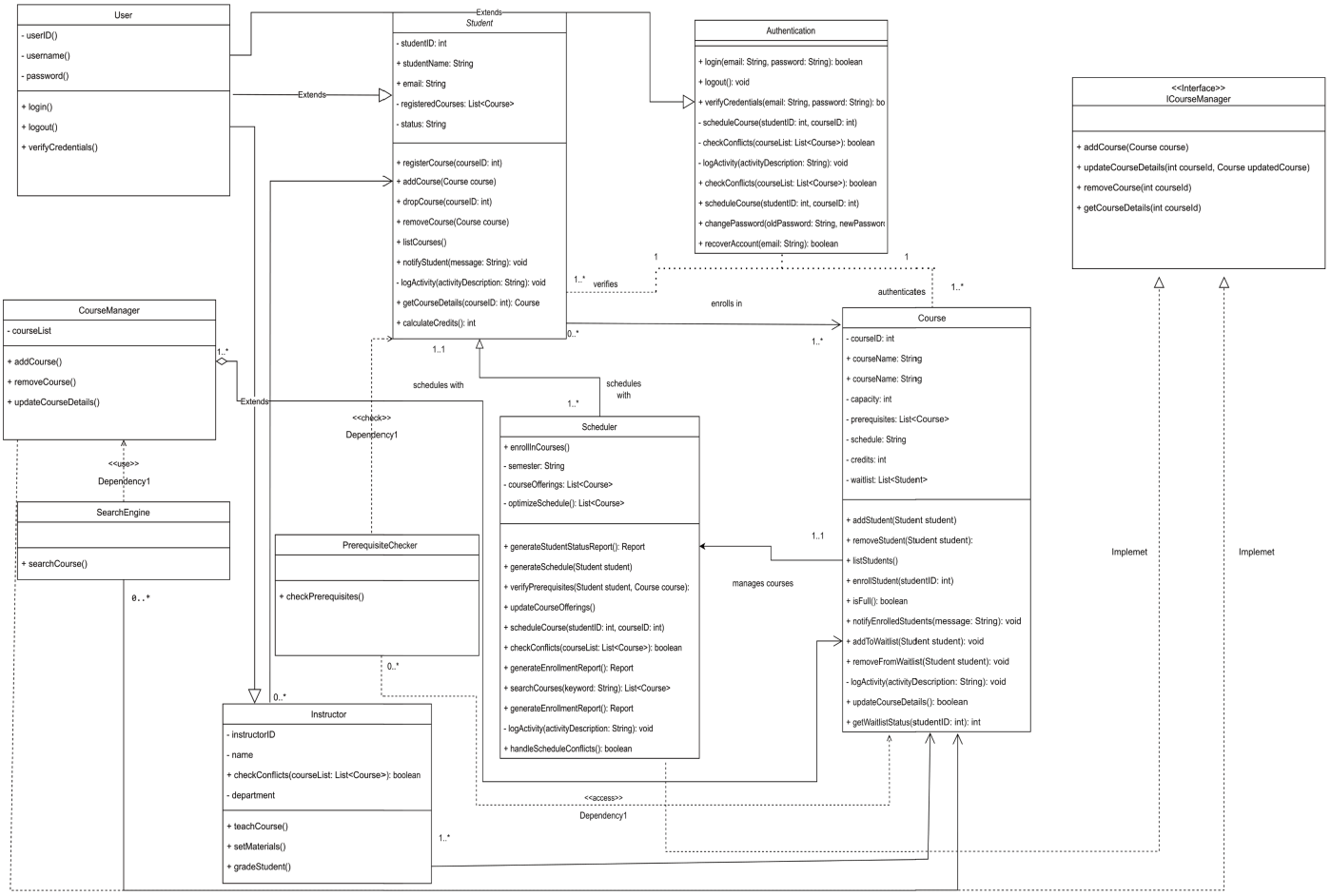

Figure 1: Class Diagram

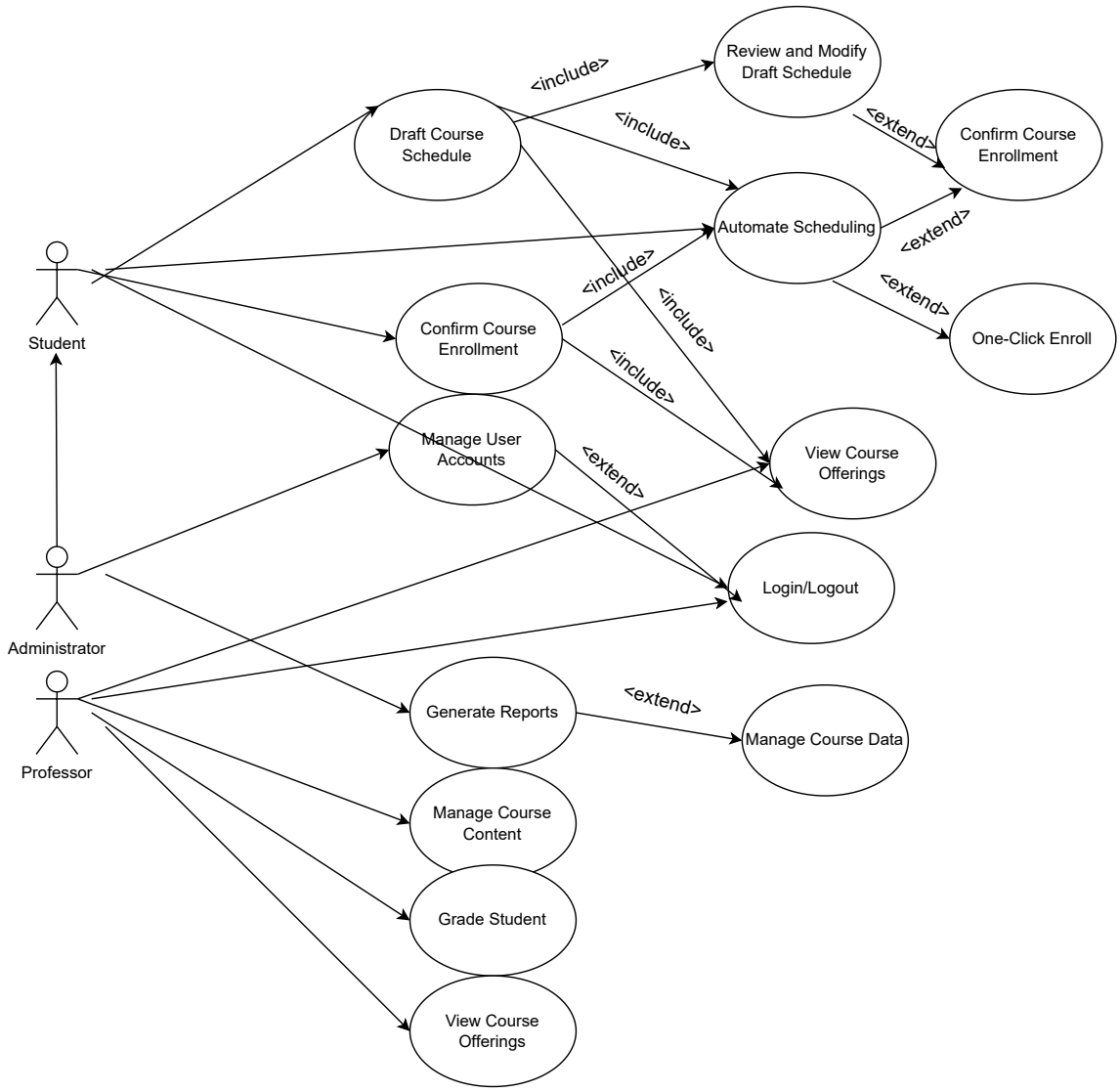

Figure 2: User Case Diagram

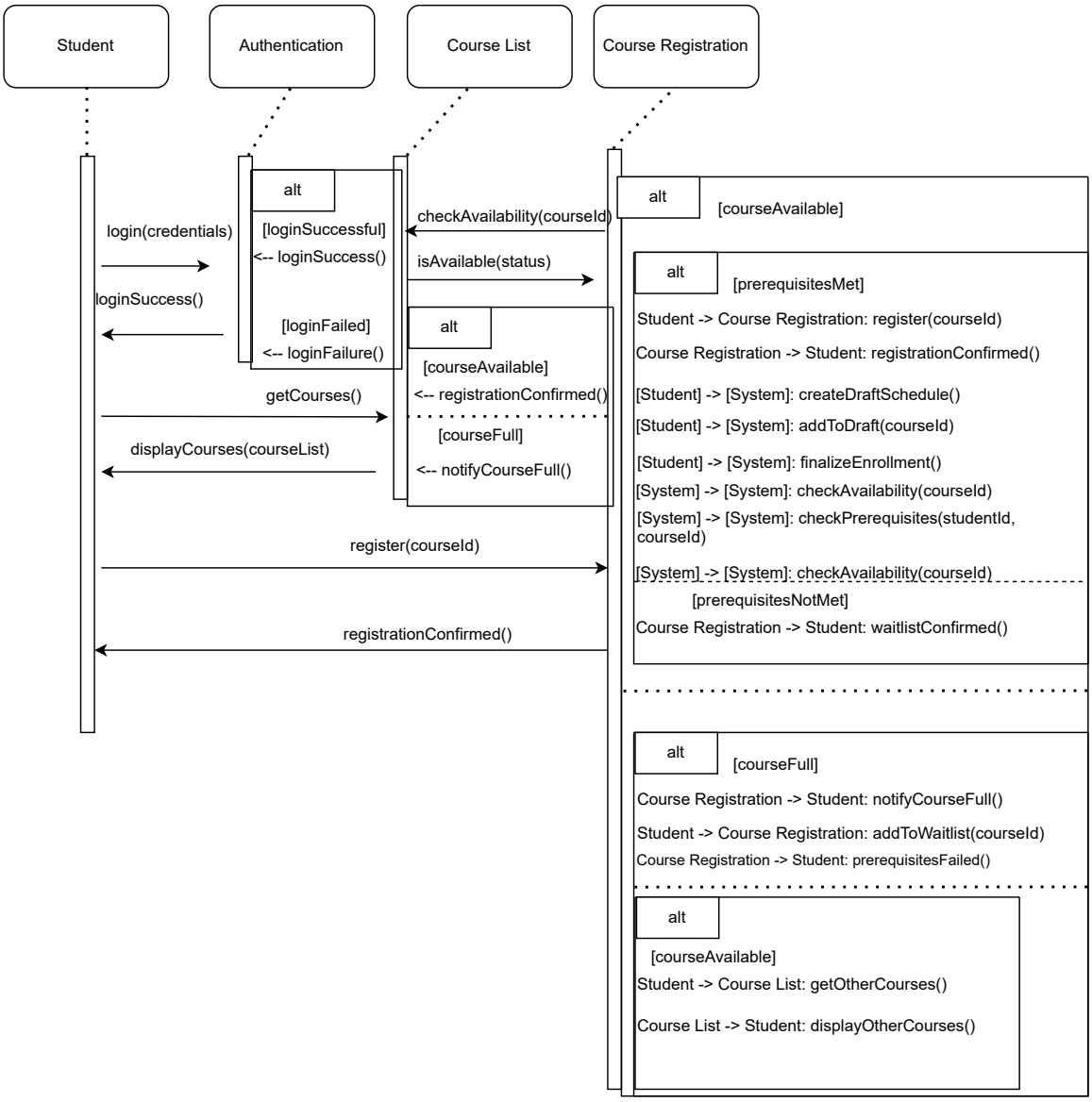

Figure 3: Sequence Diagram