Final Project

Developing A University Student Management System with Visual C++

Kelvin K. Ahiakpor & Emmanuel A. Acquaye

CS313: Intermediate Computer Programming

Dr. Jamal-Deen Abdulai

December 3, 2024

A University Student Management System using Visual C++

In this document, you will find release notes for our student management system. These may also be viewed in our GitHub repository. Fine release!

Part 1: Functional Requirements

FR 2.1.1: User Authentication

The system must authenticate users through a secure login process. For this, we use the SHA-256 hashing algorithm to securely store and compare user passwords. On the frontend, the system hides password characters as they are typed to enhance security. When the user submits their credentials, the password is hashed using SHA-256 before being sent to the server for verification, ensuring that the password is never transmitted in plain text.

FR 2.1.2: Role-based Access Control

The system enforces role-based access control (RBAC) for different user types, such as students, faculty, and administrators. Each user object is passed to the corresponding form which dynamically determines which controls should be visible, enabled, or disabled based on the user's role. This is implemented by checking the user's type at runtime during forms *Load* events. For example, administrative features are hidden from students, and faculty can only access course management and grading functionalities.

FR 2.1.3: Password Reset and Recovery

The system implements a password reset and recovery feature. If a user forgets their password, they can reset their password by answering a security question: "What is the name of your first pet?" and typing their new password. The new password is then hashed using SHA-256 and stored securely. Also, users with no security question, would be asked to input one before they can proceed to manage profile.

FR 2.2.1: Student Course Enrollment

Students can search and enroll in available courses. We have implemented a real-time search functionality in the course enrollment form. As the student types in the search textbox, the system dynamically queries the database using SQL queries like `SELECT columns FROM Courses WHERE courseName LIKE '%searchText%'`, where *searchText* is the input entered by the student. The backend uses prepared statements to prevent SQL injection. The results are displayed as the user types, allowing them to quickly find available courses. The system will then allow students to enroll in the selected courses by clicking a "Enroll" button, updating the database to reflect the student's course enrollment.

FR 2.2.2: Enrollment Verification by Faculty

Faculty members can verify student enrollments. When a student enrolls in a course, the enrollment status is initially set to "Pending." The faculty member can review the list of students enrolled in their courses and change the status of each student to "Enrolled" after verifying their eligibility. This is tracked in the database with the `Status` field in the `Enrollments` table, which can be updated using a simple SQL query, e.g., `UPDATE Enrollments SET Status = 'Enrolled' WHERE EnrollmentID = enrollmentId`.

FR 2.3.1: Faculty Grade Input and Editing

Faculty members can input, edit, and submit grades for students. The system provides an interface where faculty can select students from a course roster and assign grades. Grades are stored in the database in the `Enrollments` table, with each student's grade being linked to a specific course offering. Faculty can edit grades at any time before submitting them. Once finalized, grades are marked as "Submitted," and any further changes are restricted.

FR 2.3.2: Student Viewing of Grades and Transcripts

Students can view their grades and transcripts. Once logged in, the system retrieves the student's grades and displays them in a table format on the dashboard. The grades are pulled from the `Enrollments` table in the database, and the transcript is dynamically generated based on the student's course history. Although students can view their grades and transcripts, the system does not yet support printing these documents.

FR 2.3.3: Admin Report Generation and Printing

Admins can audit student academic performance and generate reports, such as transcripts, for individual students or for all students in a specific course. The system allows the admin to filter data based on various criteria (e.g., course, student, semester). Once a report is generated, it can be exported to PDF or printed. The report is built using data pulled from the `Enrollments` and `Transcripts` tables in the database.

FR 2.4.1: Faculty Course Material Management

Faculty members can create and update course materials. This includes uploading syllabi, assignments, and other resources. The system provides an interface where faculty can manage these materials, which are stored in the database as links to files or digital documents. These materials are made accessible to enrolled students through their course dashboard.

FR 2.4.2: Admin Course Management

Admins can manage curricula and prerequisites. Admins are able to add, edit, or remove courses from the curriculum. This feature allows admins to define course prerequisites and modify course details such as course name, credits, and instructor. Currently, course prerequisites are verified using simple string matching, e.g., comparing course names, but we plan to implement more sophisticated checks in the future to validate course prerequisites more effectively.

FR 2.4.3: Support for Online Course Materials and Digital Classrooms

The system supports the management of online course materials and digital classrooms. Faculty can upload materials such as lecture slides, videos, and Zoom links, which are made available to students enrolled in their courses. In the future, we plan to integrate external platforms like Zoom for live lectures and provide students with direct links to course materials in the system.

Part 2: Non-Functional Requirements

The non-functional requirements for the University Student Management System include performance, security, usability, and reliability enhancements, as outlined below:

Performance

- **Search Speeds:** To optimize search performance, we have indexed frequently accessed tables in the database. This indexing ensures that searches, for instance for student and course records, are fast and responsive, especially during peak use times like course registration.
- **Scalability:** We have designed the database with scalability in mind by indexing critical columns such as student IDs, course IDs, and grades. Our normalized database allows us to scale up in the future and partition tables effectively as the records in the system expand. In the envisioned **Version 2.0** of our management system, we would use database connection pooling to reduce the overhead associated with frequently opening and closing database connections to the front end. All these features combined allow our system to handle up to 10,000 concurrent users during peak times, like registration periods, without performance degradation.

Data Integrity

- **Enums for Grades:** To ensure consistent data input and prevent errors, we have used enumerated types (`Enums`) for grade values (`A+, A, B+, B, C+, C, D+, D, E, F`). This ensures that all grades entered into the system adhere to a predefined and validated set of values, reducing the risk of data corruption.
- **Enums for Credits:** We also used enumerated types (`Enums`) for course credit values (`0.5, 1.0`).

Caching

- **Profile Management Caching:** To improve page load speeds and reduce redundant database calls, we implemented a caching mechanism for the Profile Management Form. The profile data is stored within the student object and only fetched from the database when the profile is updated, significantly reducing the load time for the profile page. This caching functionality helps us keep our average page load time under 3 seconds as per the requirements.

Memory Management

- **Pass-by-Reference:** In order to minimize memory overhead, we utilized pass-by-reference when passing objects between classes – mostly forms. This prevents unnecessary copying of large objects,

thus improving memory usage efficiency.

- **Object Cleanup:** The system uses destructors to nullify objects upon user logout. This ensures that resources are properly released and avoids memory leaks, ensuring better performance during long-term use of the system.

Security

- **Data Encryption:** All sensitive data, including user passwords and personal information, is encrypted both in transit and at rest. This ensures compliance with data protection regulations such as GDPR and FERPA.

- **Authentication and Authorization:** The system enforces role-based access control (RBAC) for different types of users (students, faculty, and administrators). Only authorized users are granted access to the appropriate sections of the system, ensuring security and confidentiality of user data. We implemented this functionality by passing *User* objects to form constructors when they are called. Based on the dynamically cast user type, certain controls are restricted on each form's *Load* event.

Usability and Accessibility

- **UI Responsiveness:** The user interface is designed to be responsive, ensuring that the system performs well across a variety of desktop devices.

- **Accessibility Features:** The system does not yet comply with WCAG 2.1 standards. In our envisioned **Version 2.0** release, we will ensure that the application is more accessible to users with disabilities.

Reliability and Availability

- **Uptime:** The system is designed to achieve 99.9% uptime during critical academic periods, ensuring minimal disruption to service. See the Performance section on Page 5.

- **Data Backups:** We possess 2 redundant, regularly backed-up databases; however, we plan to implement fail-safe backup subsystems in our envisioned **Version 2.0**. These recovery systems will help prevent data loss and allow for system recovery within an hour of failure.

Maintainability

- **Object-Oriented Architecture:** Aside from the obvious classes created for forms and user role types in our system, we implemented more modular structures for maintainability. For example, our `DatabaseManager.h` header file is used to manage all database objects and provide methods for connecting to a database, closing connections, or just getting connections that can be opened later on. We also utilized the `ProjectResources` feature in Visual Studio to manage our application's file dependencies, such as the Ashesi Logo Icon. With this, we can simply select our desired resources from the project resource rather than always cycling through folders to find them.
- **Reusable Functions:** We created various reusable functions such as `OpenChildForm(formTypeID, User)` to help us open forms by passing *User* objects.
- **Soft Deletes:** For a university management system, it is imperative that we keep as much data as we can in the system for historical and sometimes legal purposes. To keep data in our database long-term even though they may not be needed day-to-day, we implemented soft deletes to virtually delete records from our databases. Rather than delete a whole record, we simply use an update query and set *isDeleted* to 1. The next time we query that table, it will be excluded by slicing only rows with *isDeleted* as 0.

Development Roles

Emmanuel Acquaye

■ **Database development**

Designed a normalized database for the data layer of our management system

■ **Rapid functional development**

Rapidly built SQL queries and event handlers to integrate required backend functionalities.

■ **Functional requirements**

Integrated most of the functional requirements, namely the student enrollment and registration, academic records and grades management & course and curriculum management.

■ **Object Oriented Design**

Implemented classes such as the `DatabaseManger.h`, `User.h`, `Faculty.h`,`Student.h`,`Admin.h` & `Course.h`

■ **Testing**

Performed tests to ensure functionality worked as expected especially after teammate's code optimization and refactoring.

Kelvin Ahiakpor

- **Front-end development**

  Designed windows forms to interact with backend.

- **User-centric iterative development**

  Optimized the User Experience with caching, and data grid refreshing to improve speed and flow. For example, profiles are loaded from the user object rather than database unless updated and searching for courses is done in realtime using a collection of retrieved cached courses.

- **Code profiling**

  Optimized code by implementing helper functions in forms performing Create, Update, Read, Delete (CRUD) operations.

- **Backend optimization**

  Optimized database with indices to improve search speeds.

- **Non functional requirement integration**

  Integrated most of the non-functional requirements. That is data encryption, password recovery with `PasswordManager.h`, data caching, backup databases, documentation and modular architecture.