1. Overall Design and Object-Oriented Principles

The Course Enrollment System leverages fundamental object-oriented principles to create a modular and scalable solution:

- **Encapsulation**: Classes such as `Authentication`, `Course`, and `Student` encapsulate related attributes and methods, ensuring data is accessible only through well-defined interfaces. For example, the `Student` class provides methods to retrieve and update a student's course enrollment without directly exposing internal data structures.
- **Inheritance**: The `User` class serves as a base class for different user types (e.g., `Student`). This approach promotes code reuse and simplifies the addition of new user types in the future.
- **Polymorphism**: Polymorphic behavior is utilized in the system's design through virtual methods in the base class `User`, allowing operations like authentication or course management to be performed in a context-specific manner.

2. Design Patterns Used

The system incorporates several key design patterns to enhance maintainability and scalability:

- **Factory Pattern**: Used in the `CourseManager` class to create course objects dynamically based on input parameters, abstracting the instantiation logic.
- **Singleton Pattern**: Applied in the `Authentication` and `DatabaseManager` classes to ensure a single, consistent point of user verification and database interaction throughout the application lifecycle.
- **Observer Pattern**: Implemented in the `MainMenuInterface` to update UI components dynamically when user actions trigger changes, such as course enrollment updates.
- **Data Access Object (DAO) Pattern**: Introduced in the `DatabaseManager` class to abstract and encapsulate all database-related operations, ensuring a clean separation of concerns and easy maintainability.

3. Quality Attributes

The system addresses key quality attributes as follows:

- **Scalability**: Modular design with distinct classes for different functionalities allows the system to scale easily. For instance, adding new features like faculty management requires minimal changes to existing code. The integration of `mydatabase.sqlite` provides persistent and scalable data handling.

- **Performance**: Efficient data structures are used for managing course and user data, ensuring quick retrieval and updates even with a large dataset. Database indexing and optimized SQL queries further improve performance.
- **Maintainability**: Adherence to object-oriented principles and the use of design patterns simplify debugging and future enhancements. Clear documentation and modular code further aid maintainability.

4. Reflection on Challenges

During development, several challenges were encountered:

- **The problem of time constraints**: the project was a bit large and we were not able to do it perfectly and exquisitely.
- **Handling Large Data Sets**: Managing large numbers of courses and students required optimizing data structures and algorithms for performance.
    - *Solution*: Implemented hash maps for quick lookup of course and student details. The `DatabaseManager` class was also introduced for persistent and efficient data handling.
- **Concurrency Management**: Ensuring thread-safe operations when multiple users access the system concurrently posed a challenge.
    - *Solution*: Used mutex locks to synchronize access to shared resources.
- **Integration of UI and Core Logic**: Bridging the gap between the user interface and backend logic required careful design.
    - *Solution*: Used the Observer pattern to decouple UI updates from backend operations.
- **Database Integration**: Implementing database functionality with SQLite required managing schema initialization and error handling.
    - *Solution*: The `DatabaseManager` class includes robust methods for database setup and execution of SQL scripts, with appropriate error checks.
- **Notification System**: Developing a notification interface to dynamically inform users of changes such as enrollment status or deadlines.
    - *Solution*: The `NotificationInterface` class was implemented to send real-time alerts to students and administrators through the GUI.

5. Test Cases or Testing Report

The system was validated through comprehensive testing:

- **Unit Testing**: Each class and method was tested independently to ensure correctness.
    1. Example: The `Authentication` class was tested with valid and invalid credentials to verify access control.
- **Integration Testing**: Interactions between components such as `CourseManager` and `Student` were tested to ensure seamless integration.

- **Performance Testing**: Simulated high user loads to verify system responsiveness and stability under stress. Database queries were profiled to ensure efficient execution.
- **Test Case Example**:
    1. *Test Name*: Valid Course Enrollment
        - Input: Student ID, Course ID
        - Expected Output: Enrollment success message, updated student schedule
    2. *Test Name*: Prerequisite Check Failure
        - Input: Student ID, Course ID without prerequisites met
        - Expected Output: Error message indicating prerequisite violation
    3. *Test Name*: Database Initialization
        - Input: Valid database path and SQL script
        - Expected Output: Database tables created successfully without errors
    4. *Test Name*: Notification Trigger
        - Input: Enrollment in a course with prerequisites
        - Expected Output: Real-time notification sent to the student's GUI
- **Testing Tools**: Used `Google Test` framework for automated unit testing and stress tests.

Conclusion

The Course Enrollment System successfully addresses the challenges of course management and enrollment through robust object-oriented design, effective use of design patterns, and careful attention to quality attributes. The integration of SQLite database functionality and the notification system further enhance the system's scalability, performance, and user experience. The final implementation demonstrates a scalable, maintainable, and user-friendly solution suitable for a wide range of environments.