Object-Oriented Programming
Semester 2025-III

GetClass - Final Delivery

Alejandro Escobar 20251020094
Jhon Gonzalez 20251020087
Sebastián Zambrano 20251020102

Computer Engineering Program
Universidad Distrital Francisco José de Caldas

December 2025

1 Introduction — Business Model Focus

GetClasses is a digital marketplace connecting students with qualified tutors. The core business hypothesis is that a centralized platform offering verified profiles, scheduling tools, and trust mechanisms increases match success, platform revenue, and user retention compared to offline alternatives.

Domain Problem and Target Market Students struggle to find qualified, available, and affordable tutors, and to compare them objectively. Tutors lack a wide audience and efficient schedule management tools. The target market includes high-school and university students and freelance tutors aiming to professionalize their services.

Value Proposition and Business Drivers GetClasses provides:

- **Efficient Matching:** Filters by subject, availability, rating, and cost.
- **Trust and Credibility:** Verified profiles, ratings, and reviews to reduce risk.
- **Operational Simplicity:** Easy management of schedules, sessions, and communication.
- **Key Metrics (KPIs):** Conversion Rate (Visitor → Session), Average Tutor Rating, and User Retention.

2 Literature Review

The development of robust software systems requires the integration of proven architectural patterns and design principles that ensure maintainability and scalability. Below are the theoretical foundations supporting the design of *GetClasses*.

2.1 Layered Software Architectures

Layered architecture is a fundamental pattern in software engineering that promotes Separation of Concerns. According to Sommerville (2015), this approach organizes the system into hierarchical strata where each layer only communicates with the one immediately below it. In the context of desktop applications with graphical interfaces (GUI), separating the presentation logic (View/Controller) from the business logic is critical to avoid excessive coupling, ensuring that interface changes do not affect business rules or data access.

2.2 Object-Oriented Design Principles (SOLID)

To manage code complexity, development is grounded in SOLID principles. These principles, popularized by Robert C. Martin, are essential to avoid software rigidity:

- **Single Responsibility Principle (SRP):** States that a class should have only one reason to change.
- **Open/Closed Principle (OCP):** Software entities should be open for extension but closed for modification, achieved through polymorphism and inheritance.

- **Interface Segregation (ISP) and Dependency Inversion (DIP):** Promote the use of abstractions over concrete implementations, facilitating modularity and unit testing.

2.3 Data Persistence and Transactionality (ACID)

The choice of persistence mechanism is vital for data integrity. Unlike flat file storage or JSON, which lack native concurrency controls, Relational Database Management Systems (RDBMS) offer ACID properties (Atomicity, Consistency, Isolation, and Durability). For commerce or reservation systems, where referential integrity between users and transactions is critical, the relational model remains the industrial standard compared to NoSQL solutions for this specific domain scope.

3 Methodology

The development of the *GetClasses* project followed an incremental and iterative methodology, structured into four main phases. This approach allowed for the refinement of requirements and architecture as the understanding of the problem domain evolved.

3.1 Phase 1: Analysis and Business Model Definition

In this initial stage, business objectives and system scope were defined. Key actors (Students, Tutors, and Administrators) were identified, and their needs were documented through **User Stories** using the *Gherkin* format (Given/When/Then). This established clear and verifiable acceptance criteria, prioritizing critical functionalities such as user registration, tutor search, and the rating system.

3.2 Phase 2: Architectural Design and Modeling

Before coding, extensive system modeling was conducted:

- **CRC Cards:** Used to preliminarily identify classes, their responsibilities, and collaborators.
- **UML Class Diagram:** The class hierarchy was structured applying inheritance (base class `User`) and polymorphism. The design was refined by applying **SOLID** principles, introducing interfaces (`IAuthentication`, `IRatingSystem`) to decouple modules.
- **System Architecture:** A strict layered architecture (Presentation, Domain, Data Access) was defined to organize the source code.

3.3 Phase 3: Technical Implementation

Software construction was carried out using Java, implementing specific design patterns to solve common problems:

- **DAO Pattern (Data Access Object):** Implemented to abstract database logic, allowing the business layer to operate with Java objects without knowing SQL query details.

- **DTO Pattern (Data Transfer Object):** Lightweight objects were designed to transfer data between persistence layers and the graphical interface.
- **Transaction Management:** Transactional control logic was implemented in critical operations (such as simultaneous registration of users and their profiles) to ensure atomic database consistency.

3.4 Phase 4: Persistence and User Interface

Finally, the **SQLite** database engine was integrated due to its portability and full SQL support. Concurrently, the graphical user interface (GUI) was developed focusing on usability, ensuring that navigation flows defined in user stories were reflected in the final screens (Login, Search, Profile).

4 Technical Design — Updated UML and SOLID Application

4.1 Updated UML Class Diagram

The following UML diagram represents the core structure. Note the emphasis on clear responsibilities and interface use, addressing previous feedback regarding SOLID adherence.

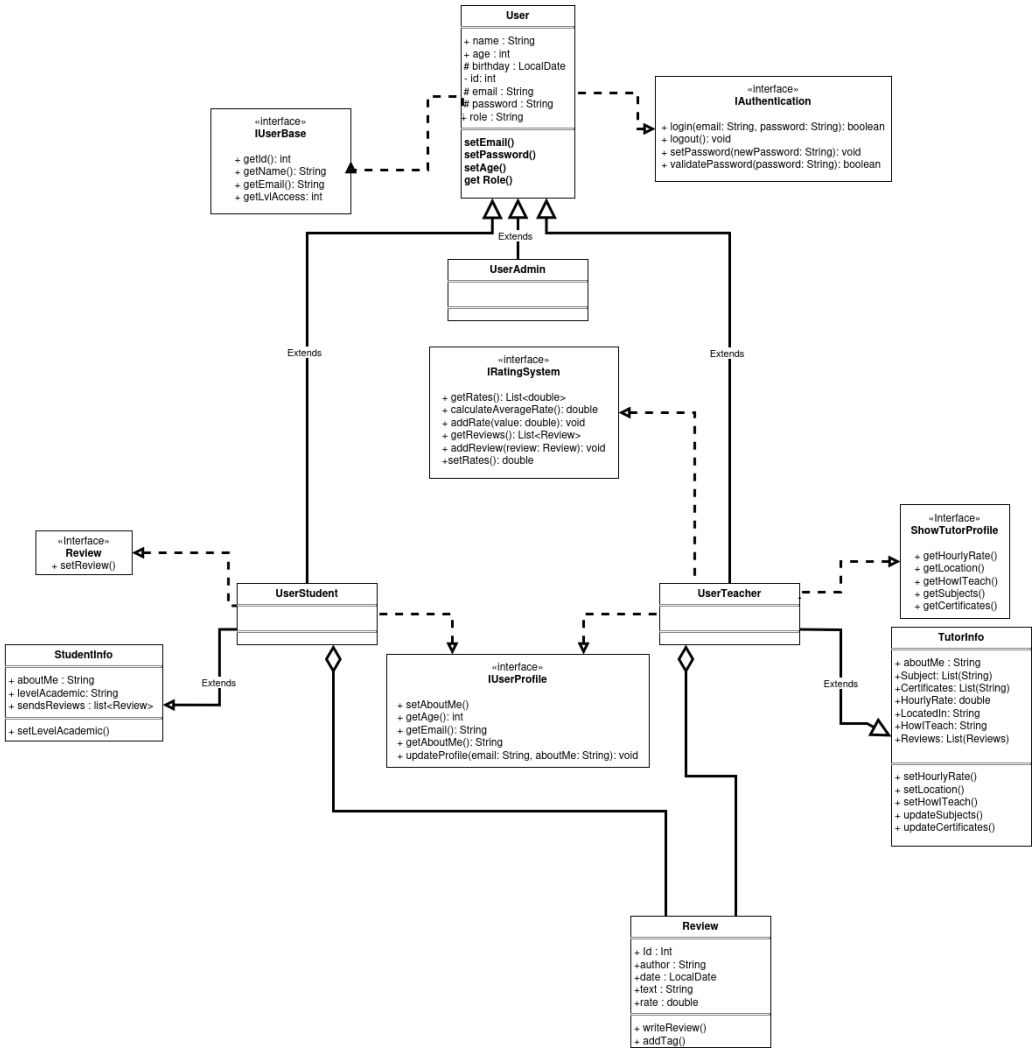

Figure 1: Updated UML class diagram. The diagram's visual clarity (colors/style) is an external issue, but the structural corrections (inheritance/interfaces) reflect SOLID.

4.1.1 Addressing Previous UML Feedback

The abstract `User` class defines shared fields. Specialized subclasses (`UserStudent`, `UserTeacher`, `UserAdmin`) extend core behaviors. The use of interfaces like `IAuthentication` and `IRatingSystem` ensures **Interface Segregation (ISP)** and **Dependency Inversion (DIP)**. The composition relationship between `UserTeacher` and `TutorInfo` isolates tutor-specific data, reinforcing **Single Responsibility (SRP)**.

4.2 SOLID Principles in Practice

The model ensures a rigorous application of SOLID principles for modularity and extensibility.

Single Responsibility Principle (SRP) Classes like `Review` were refined to handle only the review attributes and state. Aggregation logic (calculating averages, storing) is delegated to `IRatingSystem`.

Listing 1: Class Review (Example of SRP)

```
public class Review {
    // Only handles review attributes and state
    private int id;
    private int rate;
    private String text;
    private LocalDate date;

    public Review(int tutorId, int studentId, int rate, String
        comment){
        // Constructor logic
    }

    public void writeReview() { /* Persistence logic moved to DAO
        /Service */ }
    public void addTag() { /* Only responsible for modifying self
        state */ }
}
```

Open/Closed Principle (OCP) The `User` hierarchy is open for extension. New user types (e.g., `EnterpriseUser`) can be added by extending `User` and implementing interfaces without modifying the core system or base class.

Liskov Substitution Principle (LSP) `UserStudent`, `UserTeacher`, and `UserAdmin` can replace `User` in any context that expects the base class (e.g., in a `showProfile(User u)` method) without breaking functionality. This is enabled by clear inheritance and method overriding.

Interface Segregation Principle (ISP) Multiple, role-specific interfaces (`IAuthentication`, `IShowTutorProfile`, `IUserBase`) are used instead of one large interface. For example, `UserTeacher` implements `IAuthentication`, but only `TutorInfo` would implement `IShowTutorProfile`.

Dependency Inversion Principle (DIP) High-level modules (e.g., a `ReviewService`) depend on abstractions (interfaces like `IRatingSystem` or `IDataAccessObject`), not on concrete implementations (e.g., `SQLiteReviewDAO`).

5 OOP Principles in Action

5.1 Inheritance

The abstract `User` class provides a foundation for attributes like `id`, `name`, and `password`. Subclasses extend this foundation with unique responsibilities:

- **UserStudent:** Adds `level`, `bookClass()`, and `sendReview()`.
- **UserTeacher:** Adds `specialty`, `availability`, and `createClass()`.
- **UserAdmin:** Includes `manageUsers()` and `reviewReports()`.

5.2 Polymorphism

Polymorphism is implemented through method overriding (e.g., subclass-specific profile viewing) and the use of interfaces, allowing the system to treat all user subclasses as the generic `User` object in common operations.

5.3 Encapsulation

All attributes are declared as `private` or `protected` to ensure controlled access. Data consistency and security are maintained by forcing modifications through public accessor methods (`getters` and `setters`).

6 System Architecture Diagram (Layered Design)

A clear layered architecture is used to adhere to best practices, reduce coupling, and enable future scalability. This design ensures that components only interact with the layer immediately below them.

7 Requirements Documentation

7.1 Functional Requirements

1. **User Registration and Authentication:** Allow both students and tutors to register and securely validate credentials.
2. **Profile Management:** Users can manage personal profiles, rates, schedules, and subjects; tutors can upload credential documentation.
3. **Tutor Search and Filtering:** Students can search and filter tutors by subject, hourly rate, language, and rating.
4. **Scheduling and Notifications:** Students can request sessions; both users receive real-time notifications for confirmations/cancellations.
5. **Payment Processing:** Support secure online payments through integrated modules (simulated in the prototype).
6. **Messaging System:** Private chat feature for pre- and post-session communication with secure history storage.
7. **Reviews and Ratings:** Students can rate tutors after each session, with averaged ratings publicly displayed.
8. **Dispute Resolution:** Administrator tools to manage user complaints and disputes.

7.2 Non-Functional Requirements — Quality Attributes

The following requirements are oriented toward essential quality attributes for a robust application, backed by technical justifications.

- **Performance (Response Time):** The application must start in under 5 seconds. Core operations (search, schedule loading) must complete in under 3 seconds on a standard computer.
- **Security (Data Integrity):** User credentials must be hashed (e.g., BCrypt). Sensitive data (e.g., payment simulations) must be protected; communication features must not expose private data.
- **Reliability (Availability):** The system must operate without crashes. Session and profile data must be saved persistently using *SQLite transactions (ACID)* to ensure atomic writes.
- **Maintainability (Modularity):** The layered architecture must enforce low coupling, allowing each component (UI, Service, DAO) to be unit-tested and modified independently.
- **Scalability (Future Path):** The DAO abstraction layer allows for future migration from SQLite to a full relational server (e.g., MySQL/PostgreSQL) with minimal impact on business logic.

- **Usability (Intuitiveness):** The interface must be clear, consistent, and allow core tasks (registration, search, scheduling) to be completed without prior training, following standard desktop UI conventions.

8 User Stories (Gherkin Format)

The acceptance criteria are refined to follow a detailed *Given / When / Then* structure (Gherkin format) for clear and testable requirements.

ID: 1	**Tutor Registration**
Description:	As a tutor, I want to register on the platform by entering my personal and professional information so that I can offer tutoring sessions.
Acceptance Criteria:	**Given** I am on the registration page and enter all required personal and professional information correctly. **When** I click the 'Register' button. **Then** The tutor account is successfully created and I am logged into the platform.

Table 1: User Story 1 – Tutor Registration

ID: 2	**Tutor Profile Picture**
Description:	As a tutor, I want to upload a profile picture so that students can identify me easily.
Acceptance Criteria:	**Given** I am logged in as a tutor and on my profile editing page. **When** I select an image file and click 'Upload'. **Then** The uploaded image appears correctly on my profile page for students to see.

Table 2: User Story 2 – Tutor Profile Picture

ID: 3	**Tutor Search**
Description:	As a student, I want to search for tutors by subject, rate, or language so that I can find the best match for my learning needs.
Acceptance Criteria:	**Given** I am on the tutor search page and I apply filters (e.g., subject, rate, or language). **When** I execute the search. **Then** Only tutors matching the selected filters are displayed correctly in the search results list.

Table 3: User Story 3 – Tutor Search

ID: 4	**Tutor Listing**
Description:	As a student, I want to view a list of available tutors so that I can compare their profiles and select one.
Acceptance Criteria:	**Given** I am on the main tutor listing page. **When** The page loads. **Then** The system displays a complete list of tutors, and for each one, their name, subjects taught, and average rating are visible.

Table 4: User Story 4 – Tutor Listing

ID: 5	**Tutor Rates Student**
Description:	As a tutor, I want to rate students after a session to maintain platform quality and accountability.
Acceptance Criteria:	**Given** I have completed a tutoring session with a student. **When** I access the post-class rating form and submit a score and optional review. **Then** My rating and review are successfully recorded and associated with the student's profile.

Table 5: User Story 5 – Tutor Rates Student

ID: 6	**Student Rates Tutor**
Description:	As a student, I want to rate tutors after a session so that other users can make informed decisions.
Acceptance Criteria:	**Given** I have completed a tutoring session with a tutor. **When** I access the post-class rating form and submit a score and optional comment. **Then** My rating and comment are successfully recorded, the tutor's average rating is updated, and the comment is visible on their profile.

Table 6: User Story 6 – Student Rates Tutor

ID: 7	**View Ratings**
Description:	As a user, I want to view tutor and student ratings so that I can evaluate trust and performance.
Acceptance Criteria:	**Given** I am viewing a tutor's or student's profile. **When** I navigate to the ratings section. **Then** Ratings and feedback are visible, clearly displaying the associated rating score and the reviewer's profile.

Table 7: User Story 7 – View Ratings

9 CRC Cards (Detailed Responsibilities)

Responsibilities are detailed and explicit to improve the clarity of the design model.

Class: User	
Responsibility	**Collaborators**
Representing the general data and behaviors of a user within the system (e.g., ID, Email, Authentication).	UserAdmin UserStudent UserTeacher IAuthentication IDataBase

Table 8: User CRC

Class: TutorInfo	
Responsibility	**Collaborators**
Manage and store the specific data that only the tutor should have (e.g., Hourly Rate, Certificates, Subject/Areas).	UserTeacher IBillingSystem IShowTutorProfile

Table 9: TutorInfo CRC

Class: StudentInfo	
Responsibility	**Collaborators**
Manage and store the specific data that only the student should have (e.g., Academic Level, List of Reviews sent).	UserStudent Review

Table 10: StudentInfo CRC

Class: Review	
Responsibility	**Collaborators**
Represent a specific rating instance; store the review details (author, date, text, rate).	IBillingSystem UserTeacher UserStudent

Table 11: Review CRC

10 Persistence Justification: SQLite vs JSON

For persistent storage involving structured, relational data and user-generated content, **SQLite was selected over JSON**. The use of a relational engine provides more robustness, data integrity, and long-term maintainability, aligning with the complexity of a tutoring platform.

10.1 Why SQLite?

SQLite is an embedded relational database that requires no server and naturally fits the domain's need for data relations (Users, Tutors, Reviews).

- **ACID Transactions:** Guarantees consistency when storing reviews, updating ratings, or editing profiles, which is critical for a commerce platform.
- **Relational Integrity:** Foreign keys enforce correct linking between students, tutors, and reviews, preventing orphaned data.
- **Efficient Queries:** Complex searches (e.g., filtering tutors by subject, location, and rating simultaneously) are efficient.
- **Maintainability:** Using the DAO pattern with SQL (via JDBC) is standard practice and easier to maintain when the object graph evolves.

10.2 Why not JSON?

JSON storage is suitable for simple data or configuration files, but it presents challenges as a primary persistence mechanism for structured data:

- **Lack of Transaction Safety:** Concurrent access can easily lead to data corruption without complex manual locking mechanisms.
- **Poor Querying:** Searching for data (e.g., finding all tutors in a location) requires manual parsing and iteration of the entire file, which is inefficient.
- **Weak Integrity:** No built-in support for foreign keys or referential constraints, making data integrity management difficult.

Listing 2: SQLite Connection Code Example (DAO Layer)

```
public class ConnectionDB {
    // JDBC URL for SQLite database file
    private static final String URL = "jdbc:sqlite:src/main/
       resources/GetclassDB.db";

    public static Connection getConnection() {
        Connection conn = null;
        try {
            // Establishes connection to the local DB file
            conn = DriverManager.getConnection(URL);
            System.out.println("Conexi n establecida con  xito "
               );
        } catch (SQLException e) {
            System.out.println("Error al conectar: " + e.
               getMessage());
        }
        return conn;
    }
}
```

11 GUI Mockups (Standardized Views)

The UI focuses on clarity and standard desktop conventions. The sizing of elements is consistent to improve usability and optimize screen space.

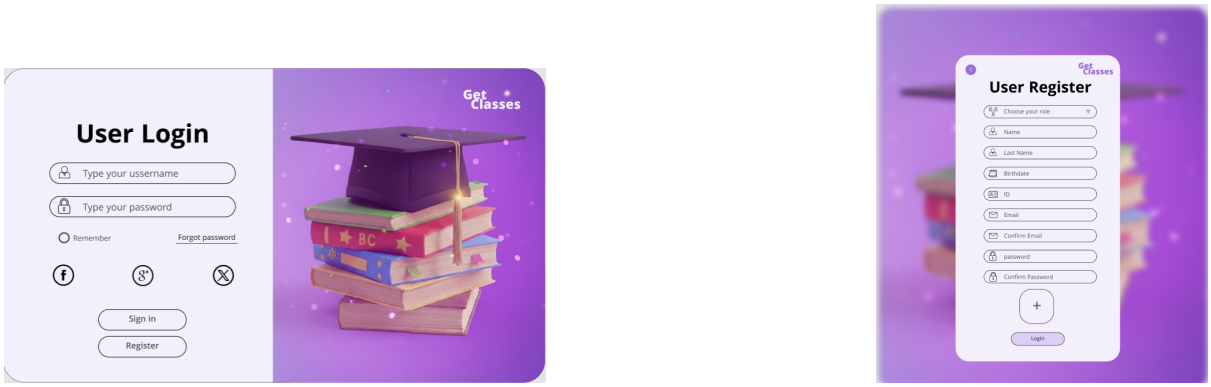

Figure 2: Login (Left) and Register (Right) Pages. Standardized sizing and clear form fields.

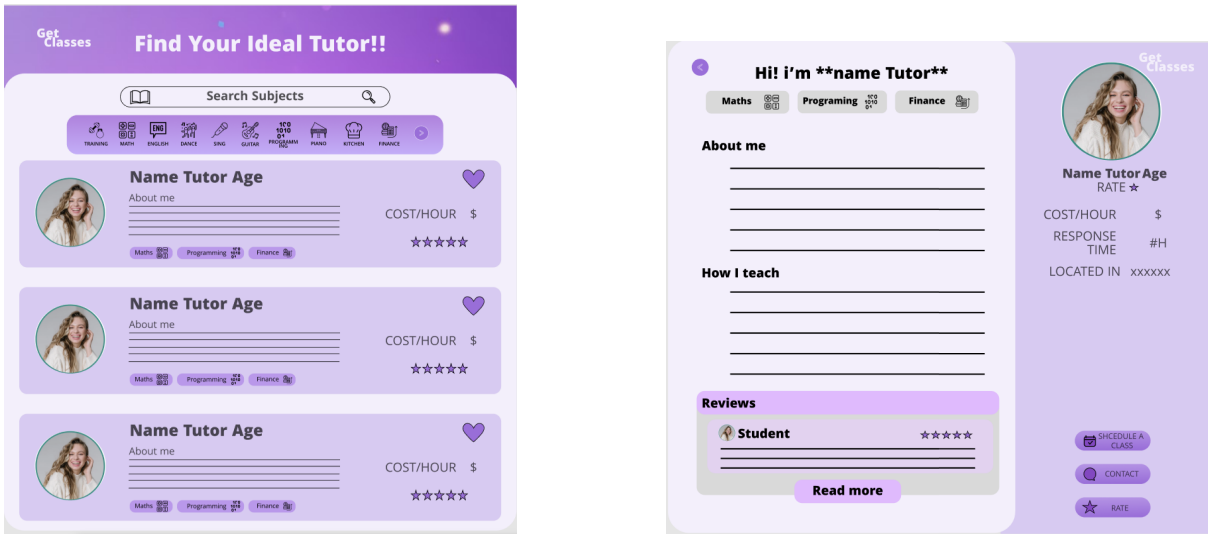

Figure 3: Main Page (Tutor Search List) and Tutor Profile Page. The list view utilizes modular `TutorCard` components.

12 Code Snippets and Best Practices

The following code snippets demonstrate the application of Object-Oriented Programming principles, robust error handling, and architectural patterns used in the **GetClasses** system.

12.1 ACID Transactions in DAO Layer

To ensure data integrity, especially when inserting data across multiple related tables (e.g., creating a User, then their TutorInfo, then their Subjects), we utilize JDBC transactions. The setAutoCommit(false) method allow us to treat multiple SQL statements as a single atomic unit.

Listing 3: Transaction Management in UserTeacherDAO

```
public static boolean save(Connection conn, UserTeacher teacher) {
    try {
        // 1. Disable auto-commit to start transaction
        conn.setAutoCommit(false);

        // 2. Insert into USER table
        stmtUser = conn.prepareStatement(INSERT_USER_SQL,
            Statement.RETURN_GENERATED_KEYS);
        // ... (setting parameters)
        stmtUser.executeUpdate();

        // 3. Insert into TUTOR_INFO table using the generated
            User ID
        stmtInfo = conn.prepareStatement(INSERT_TUTOR_INFO_SQL);
        // ... (setting parameters)
        stmtInfo.executeUpdate();

        // 4. Commit if all steps succeed
        conn.commit();
        return true;

    } catch (SQLException e) {
        // 5. Rollback to previous state if any error occurs
        try { conn.rollback(); } catch (SQLException ex) {}
        e.printStackTrace();
        return false;
    } finally {
        // 6. Restore default commit behavior and close resources
        try { conn.setAutoCommit(true); } catch (SQLException ex)
            {}
    }
}
```

12.2 Inheritance and DTO Pattern

The system uses *Data Transfer Objects (DTOs)* to move data between the database and the GUI without exposing the database structure directly. We apply *Inheritance* to avoid code duplication; UserDTO holds the common attributes (name, email), while UserStudentDTO extends it to add specific fields.

Listing 4: Inheritance in DTOs

```
// Base Class
public class UserDTO {
    public String name;
    public String email;
    protected String password;
    public String role;

    public UserDTO(int id, String name, String lastName, /.../) {
        this.id = id;
        this.name = name;
        // ...
    }
}

// Subclass
public class UserStudentDTO extends UserDTO {
    public StudentInfoDTO studentInfo;

    public UserStudentDTO(int id, String name, /.../) {
        // Reuse parent constructor
        super(id, name, lastName, birthDate, email, age, password
            , role);
    }

    public void setStudentInfo(StudentInfoDTO studentInfo){
        this.studentInfo = studentInfo;
    }
}
```

12.3 Resource Management (Preventing Memory Leaks)

Database resources (ResultSet, PreparedStatement) must be closed strictly to prevent memory leaks and connection pool exhaustion. We implement this using try-finally blocks in every DAO method.

Listing 5: Safe Resource Closing in UserDAO

```
public static UserDTO Login(Connection Conn, String email, String
    password) {
    PreparedStatement stmt = null;
    ResultSet rs = null;

    try {
        stmt = Conn.prepareStatement(LOGIN_QUERY);
        stmt.setString(1, email);
        stmt.setString(2, password);
        rs = stmt.executeQuery();

        if (rs.next()) {
            return new UserDTO(/* data from result set */);
        }
        return null;

    } catch (SQLException e) {
        e.printStackTrace();
        return null;
    } finally {
        // Best Practice: Always close resources in reverse order
            of opening
        try { if (rs != null) rs.close(); } catch (SQLException e
           ) {}
        try { if (stmt != null) stmt.close(); } catch (
           SQLException e) {}
    }
}
```

12.4 Separation of Concerns (DAO vs Controller)

The architecture strictly separates responsibilities. The *DAO* (Data Access Object) handles purely SQL operations, while the *Controller* manages the flow. The code below shows how the DAO constructs complex objects (like a Student with their Reviews) from the raw database rows.

Listing 6: Object Construction in UserStudentDAO

```
// Inside UserStudentDAO.getById()
while (rsReview.next()) {
    // The DAO is responsible for mapping SQL rows to Java
       Objects
    info.addSendedReviews(
        rsReview.getInt("review_id"),
        rsReview.getInt("tutor_user_id"),
        rsReview.getInt("student_user_id"),
        rsReview.getInt("score"),
        rsReview.getString("comment"),
        LocalDate.parse(rsReview.getString("review_date"))
    );
}
student.setStudentInfo(info);
return student;
```

13 Final Reflection and Next Steps

This workshop successfully implemented architectural corrections and solidified the domain model according to SOLID principles, with a strong focus on the business justification and quality attributes.

13.1 Summary of Key Improvements

- *Business Model Integration:* Introduction explicitly details the business problem, value proposition, and KPIs.
- *Architecture Clarity:* Introduction of a clear Layered Architecture Diagram and justification for the JavaFX monolithic structure.
- *SOLID Implementation:* Detailed narrative and code examples illustrate the adherence to all five SOLID principles.
- *Persistence Decision:* Detailed justification for choosing SQLite over JSON, ensuring data integrity and maintainability.
- *Testable Requirements:* User Stories were updated to use the **Given/When/Then** format for clearer and testable acceptance criteria.
- *Detailed CRC Cards:* Responsibilities are now explicit and detailed, reflecting true system behavior.

13.2 Immediate Next Steps (Addressing Remaining Gaps)

- *Code Quality:* Implement robust source code documentation (JavaDoc) and adhere to standard Java file and class naming conventions.
- *Test Automation:* Begin implementing unit tests to validate the business logic layer, especially for core functionality like rating calculations and class scheduling.
- *UI Polish:* Finalize the aesthetic design of the JavaFX UI, ensuring responsiveness and an intuitive user experience.

References

- Overleaf. (2024). *LaTeX tutorial: Learn LaTeX step by step*. Retrieved from https://www.overleaf.com/learn
- Lamport, L. (1994). *LaTeX: A Document Preparation System*. Addison-Wesley.
- Lucidchart. (2023). *UML Diagram Tutorial*. Retrieved from https://www.lucidchart.com/pages/uml-diagram
- Ambler, S. W. (2023). *Agile Modeling: UML and Class Design*. Retrieved from http://www.agilemodeling.com/artifacts/classDiagram.htm
- Beck, K., & Fowler, M. (2000). *Planning Extreme Programming*. Addison-Wesley.
- Mountain Goat Software. (2022). *User Stories*. Retrieved from https://www.mountaingoatsoftware.com/agile/user-stories
- Coad, P., & Yourdon, E. (1991). *Object-Oriented Design*. Prentice Hall.
- Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson.
- Visual Paradigm. (2023). *CRC Cards Tutorial*. Retrieved from https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-crc-card/
- IEEE. (2023). *Guide to Software Design Documentation (IEEE 1016-2020)*.