Object-Oriented Programming
Semester 2025-III

Workshop No. 1 — Object-Oriented Design (Updated Version)

Alejandro Escobar 20251020094
Jhon Gonzalez 20251020087
Sebastián Zambrano 20251020102

Computer Engineering Program
Universidad Distrital Francisco José de Caldas

1 Introduction

The **GetClasses** project is an online platform designed to connect students and tutors through a smooth, intuitive, and secure learning environment. The system provides a space where students can easily find qualified tutors by subject, pricing, or location, while tutors can promote their academic expertise and manage their teaching schedules efficiently.

The motivation for this project comes from the increasing demand for reliable and accessible remote learning solutions. The main objective of GetClasses is to streamline academic interaction, communication, and payment management within a unified digital ecosystem. The scope of the system includes multilingual support, layered system architecture, user management for students and tutors, and future integration of additional services.

2 Technical Design (Updated UML Diagrams)

The following UML diagram represents the updated technical structure of the **Get-Classes** system after applying refactorings, layered architecture adjustments, and SOLID principles. This version focuses on simplifying responsibilities, strengthening inheritance hierarchies, and improving modularity across the main components.

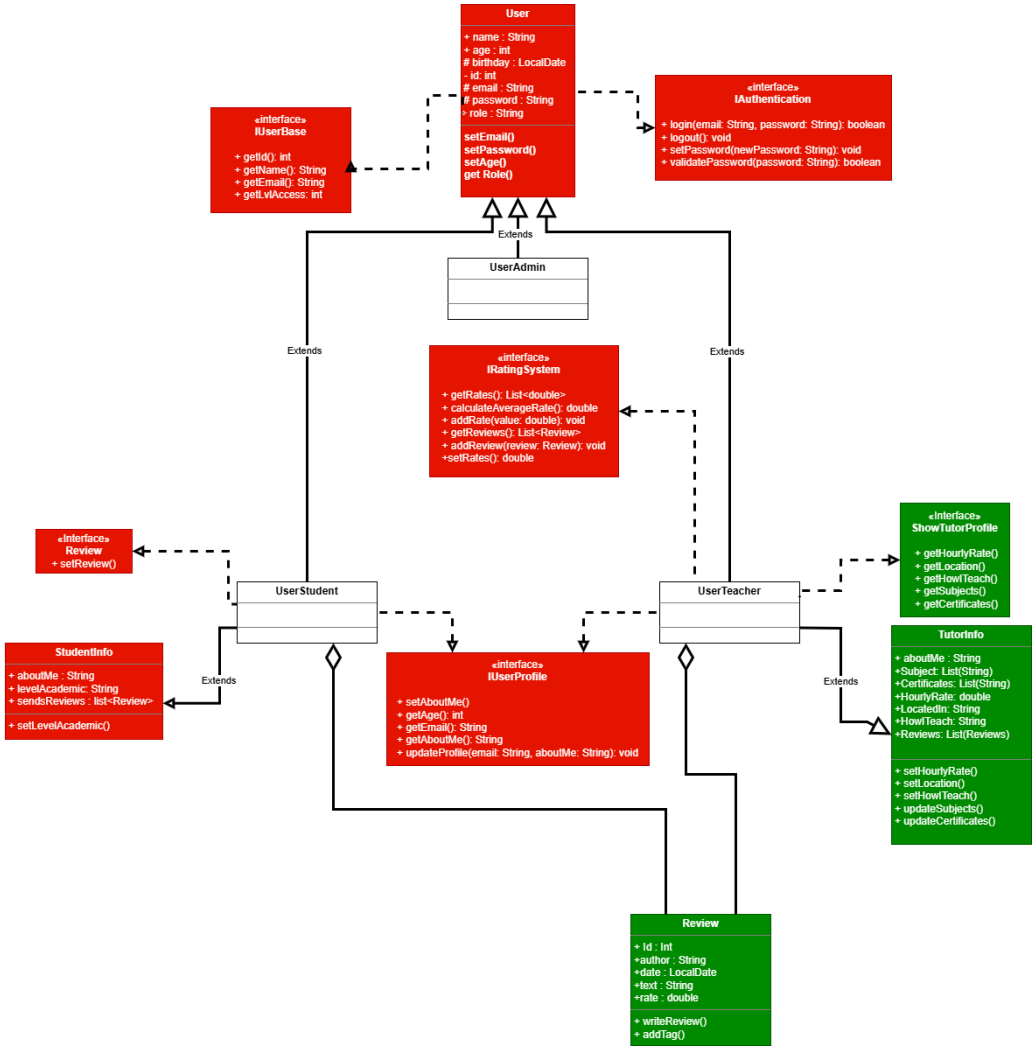


Figure 1: Updated UML class diagram applying SOLID principles, inheritance, and a clearer separation of responsibilities.

The updated diagram highlights:

- A refined class hierarchy where the abstract `User` class defines shared fields and behaviors.
- Specialized subclasses—`StudentUser`, `TutorUser`, and `AdminUser`—that extend and override functionalities according to their roles.
- Clearer associations with system components such as schedules, payments, reviews, and messaging.
- Encapsulation applied through private attributes and public methods (getters and setters).

- Polymorphism implemented by method overriding and the use of interfaces when necessary.

This revised UML supports a layered architecture and prepares the system for a clean implementation within a monolithic JavaFX application.

SOLID application

The final UML model was thoroughly refined to ensure a rigorous application of the SOLID principles, resulting in a clearer, more modular, and extensible design. Each class was redefined with a unique and well-delimited responsibility, reducing coupling and increasing the overall cohesion of the system. The User class was simplified to represent only shared attributes and behaviors, functioning as the base entity from which Student and Teacher inherit. This guarantees compliance with the **Single Responsibility Principle (SRP)** and the **Liskov Substitution Principle (LSP)**, allowing subclasses to replace User without altering the system's behavior.

The Chat class was redesigned to handle interactions between two instances of User instead of using text strings for names, improving object relationships and future extensibility under the **Open/Closed Principle (OCP)**. Message was refined to encapsulate only the content, date, and time of the message, maintaining a single reason for change and facilitating future extensions such as message types or attachments. Similarly, Review was redefined with precise attributes (id, author, date, text, rate) and focused methods such as writeReview() and addTag(), ensuring extensibility without modifying its core structure. In addition, Reviews was introduced as an aggregator class responsible for managing multiple instances of Review, offering operations for querying and analyzing ratings in accordance with **SRP** and the Dependency **Inversion Principle (DIP)**.

On the other hand, the classes Request and Answer were introduced, from which ClassRequest inherits. Request manages information associated with class requests (dates, schedules, participants, subject, modality, price, and status), while Answer encapsulates the response (attribute accept and method setAnswer()). As a result, ClassRequest now contains only the sendRequest() method, adhering to **SRP** and reinforcing the hierarchy under **LSP** and **OCP**.

Finally, the Schedule class was simplified to handle only availability through setAvailable(), while two new classes, ChangeClasses and ClassesInfo, were added to distribute responsibilities for modification and query operations. ChangeClasses manages the addition, deletion, or clearing of classes, and ClassesInfo handles queries such as retrieving schedules or counting total events. This refactoring promotes a highly modular, clear, and extensible structure, exemplifying the application of **SRP**, **OCP**, and **DIP**, and achieving a clean, flexible, and easily maintainable architecture.

In addition, a set of interfaces was incorporated to reinforce the application of the **Interface Segregation Principle (ISP)** and the **Dependency Inversion Principle (DIP)**. Interfaces such as IShowRates, IRate, IShowProfile, IAccessLevel, IAuthentication, ISchedule, IShowTutorProfile, and IReview ensure that each class implements only the behaviors it truly requires, avoiding unnecessary dependencies and improving system flexibility. For instance, User implements interfaces related to profile visualization, ratings, and access control, while TutorInfo uses its own interface to display tutor-specific information. This strategic use of interfaces guarantees low coupling, promotes code reusability, and allows new functionalities to be added or modified without affecting existing classes, consolidating a scalable, coherent model aligned with the core principles of object-oriented design.

3 OOP Principles in Action

Inheritance

The `User` class defines shared attributes such as `id`, `name`, `email`, and `password`, as well as methods like `login()` and `logout()`. Subclasses such as `StudentUser`, `TutorUser`, and `AdminUser` inherit these and add their own properties and behaviors:

- **StudentUser:** Adds `level`, `bookClass()`, and `sendReview()`.
- **TutorUser:** Adds `specialty`, `availability`, and `createClass()`.
- **AdminUser:** Includes `manageUsers()` and `reviewReports()`.

Polymorphism

Polymorphism allows functions to treat subclass instances as generic `User` objects. For example, `showProfile(User u)` can call subclass-specific implementations depending on the object type.

Encapsulation

Private and protected attributes ensure controlled access through getters and setters, maintaining security and consistency across the system.

System Overview

The **GetClasses** system is composed of several entities that interact to support its core functionalities. These entities include users (students and tutors), administrators, and subsystems for scheduling, payments, messaging, and reviews.

This modular design allows flexibility in maintaining and scaling the platform, ensuring that each module can evolve independently while maintaining integration with the overall architecture.

System Behavior and Data Flow

In **GetClasses**, the main operations are driven by user interactions. Students can create accounts, search for tutors, book classes, and complete payments through integrated modules. Tutors manage their profiles, set availability, and interact with students through chat and virtual sessions. The administrator ensures compliance and resolves disputes.

The data flow of the platform is summarized as follows: users input their data through web forms, which are processed and validated by the backend. Once authenticated, the system stores user information, manages session scheduling, and records transactions. Notifications are automatically triggered to keep users informed of bookings, payments, and reviews.

4 Requirements Documentation

Functional Requirements

1. **User Registration and Authentication:** The system must allow both students and tutors to register using an email address or third-party authentication (e.g., Google). It should securely validate credentials and provide password recovery options.

2. **Profile management:** Users can manage their personal profiles, including photo, biography, experience, academic degrees, hourly rate, availability schedule, and preferred teaching subjects. Tutors must be able to verify credentials through documentation uploads.

3. **Tutor Search and Filtering:** Students must be able to search for tutors using filters such as subject, hourly rate, spoken language, country, and rating. The system should display ranked results with basic tutor information.

4. **Scheduling and Notifications:** Students can request sessions with tutors based on their availability. Both users will receive real-time notifications (email and in-app) about session confirmations, cancellations, or rescheduling.

5. **Payment Processing:** The platform must support secure online payments through credit card and e-wallets, ensuring transaction integrity and confirmation receipts.

6. **Messaging System:** Students and tutors should have access to a private chat feature for pre- and post-session communication, with message history stored securely.

7. **Reviews and Ratings:** After each completed session, students can rate their tutors and leave feedback. Ratings will be averaged and displayed publicly on tutor profiles.

8. **Dispute Resolution:** An administrator must have tools to manage user complaints and disputes, including viewing session logs and communication records for fair resolution.

Non-Functional Requirements

- **Performance:** The application must start and load its main interface in under 5 seconds on a standard computer. Core operations such as searching tutors or loading schedules should complete in under 3 seconds. It should function smoothly on systems with at least 4 GB of RAM.
- **Usability:** The interface must be clear, intuitive, and consistent, allowing users to navigate key features (registration, search, scheduling) without prior training. The design should follow standard desktop UI conventions (menus, dialogs, toolbars). Language support in English is required; Spanish may be added optionally.
- **Security:** User credentials must be encrypted or hashed before being stored locally. Communication features (e.g., messaging or class scheduling) should not expose private data. Any payment simulation or sensitive operation should be represented through mock interfaces for safety.
- **Availability:** The system should operate without crashes or data loss during typical use. Session and profile data must be saved persistently in a local database or file system. A manual backup option should be available to prevent accidental data loss.
- **Flexibility:** The architecture should allow gradual integration of advanced features (e.g., online payments, video calls, or multi-language support) in future versions, while keeping the current version lightweight and functional for local use.

5 User Stories

ID: 1	**Tutor Registration**
Priority:	High
Effort (hrs):	6
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:	The tutor account is successfully created after entering valid information.

Table 1: User Story 1 – Tutor Registration

ID: 2	**Tutor Profile Picture**
Priority:	Medium
Effort (hrs):	3
Description:	As a tutor, I want to upload a profile picture so that students can identify me easily.
Acceptance Criteria:	The uploaded image appears correctly on the tutor's profile page.

Table 2: User Story 2 – Tutor Profile Picture

ID: 3	**Tutor Search**
Priority:	High
Effort (hrs):	8
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:	Tutors matching the selected filters are displayed correctly in the search results.

Table 3: User Story 3 – Tutor Search

ID: 4	**Tutor Listing**
Priority:	High
Effort (hrs):	5
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:	The system displays a complete list of tutors with names, subjects, and ratings.

Table 4: User Story 4 – Tutor Listing

ID: 5	**Chat with Tutor**
Priority:	High
Effort (hrs):	6
Description:	As a student, I want to chat in real time with tutors so that I can clarify doubts before booking a session.
Acceptance Criteria:	The chat allows real-time message exchange between tutor and student.

Table 5: User Story 5 – Chat with Tutor

ID: 6	**Tutor Rates Student**
Priority:	Medium
Effort (hrs):	3
Description:	As a tutor, I want to rate students after a session to maintain platform quality and accountability.
Acceptance Criteria:	The tutor can submit a rating and review after completing a class.

Table 6: User Story 6 – Tutor Rates Student

ID: 7	**Student Rates Tutor**
Priority:	Medium
Effort (hrs):	3
Description:	As a student, I want to rate tutors after a session so that other users can make informed decisions.
Acceptance Criteria:	The student can rate the tutor and leave a comment after class.

Table 7: User Story 7 – Student Rates Tutor

ID: 8	**View Ratings**
Priority:	Medium
Effort (hrs):	3
Description:	As a user, I want to view tutor and student ratings so that I can evaluate trust and performance.
Acceptance Criteria:	Ratings and feedback are visible and linked to corresponding user profiles.

Table 8: User Story 8 – View Ratings

ID: 9	**Tutor Auto-Responder**
Priority:	Low
Effort (hrs):	2
Description:	As a tutor, I want to enable an auto-responder when I am unavailable so that students receive immediate feedback.
Acceptance Criteria:	The system automatically sends a pre-defined message when the tutor is offline.

Table 9: User Story 9 – Tutor Auto-Responder

ID: 10	**Contact Support/Admin**
Priority:	High
Effort (hrs):	4
Description:	As a user, I want to contact platform support or administrators to report issues or request assistance.
Acceptance Criteria:	Support requests are sent successfully and confirmation is received.

Table 10: User Story 10 – Contact Support/Admin

6 CRC Cards (Tabular Format)

Class: ProfileUpdate	
Responsibility	**Collaborators**
Register the basic profile data	User

Table 11: ProfileUpdate CRC

Class: User	
Responsibility	**Collaborators**
Representing the general data and behaviors of a user within the system	ProfileUpdate Teacher Student

Table 12: User CRC

Class: TutorInfo	
Responsibility	**Collaborators**
Manage the data that only the tutor should have	Teacher

Table 13: TutorInfo CRC

Class: StudentInfo	
Responsibility	**Collaborators**
Manage the data that only the student should have	Student

Table 14: StudentInfo CRC

Class: Teacher	
Responsibility	**Collaborators**
Base of the tutor user; inherits attributes and methods from User	User TutorInfo Reviews ClassRequest Schedule PaymentSystem

Table 15: Teacher CRC

Class: Student	
Responsibility	**Collaborators**
Base of the student user; inherits attributes and methods from User	User StudentInfo Reviews ClassRequest Schedule PaymentSystem

Table 16: Student CRC

Class: PaymentSystem	
Responsibility	**Collaborators**
Handle payments	Teacher Student

Table 17: PaymentSystem CRC

Class: Reviews	
Responsibility	**Collaborators**
Add, delete or edit tutor reviews	Review Student Teacher

Table 18: Reviews CRC

Class: Review	
Responsibility	**Collaborators**
Create the review and assign a rating	Reviews

Table 19: Review CRC

Class: Chat	
Responsibility	**Collaborators**
Store and represent chat messages	Student Teacher Message

Table 20: Chat CRC

Class: Message	
Responsibility	**Collaborators**
Create and edit the message before sending	Chat

Table 21: Message CRC

Class: Request	
Responsibility	**Collaborators**
Create the class request	ClassRequest

Table 22: Request CRC

Class: Answer	
Responsibility	**Collaborators**
Establish the tutor's response to the class request	ClassRequest

Table 23: Answer CRC

Class: ClassRequest	
Responsibility	**Collaborators**
Send and handle class requests	Student Teacher Answer Request

Table 24: ClassRequest CRC

Class: ChangeClasses	
Responsibility	**Collaborators**
Add and remove classes from the schedule	Schedule

Table 25: ChangeClasses CRC

Class: ClassesInfo	
Responsibility	**Collaborators**
Show the information of assigned classes	Schedule

Table 26: ClassesInfo CRC

Class: Schedule	
Responsibility	**Collaborators**
Set availability and manage the base schedule	Teacher Student ChangeClasses ClassesInfo

Table 27: Schedule CRC

7 Mockups

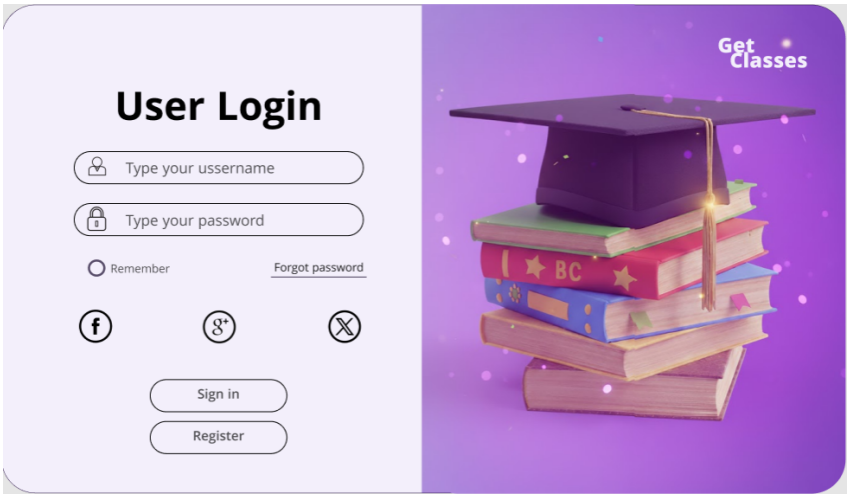


Figure 2: Login Page

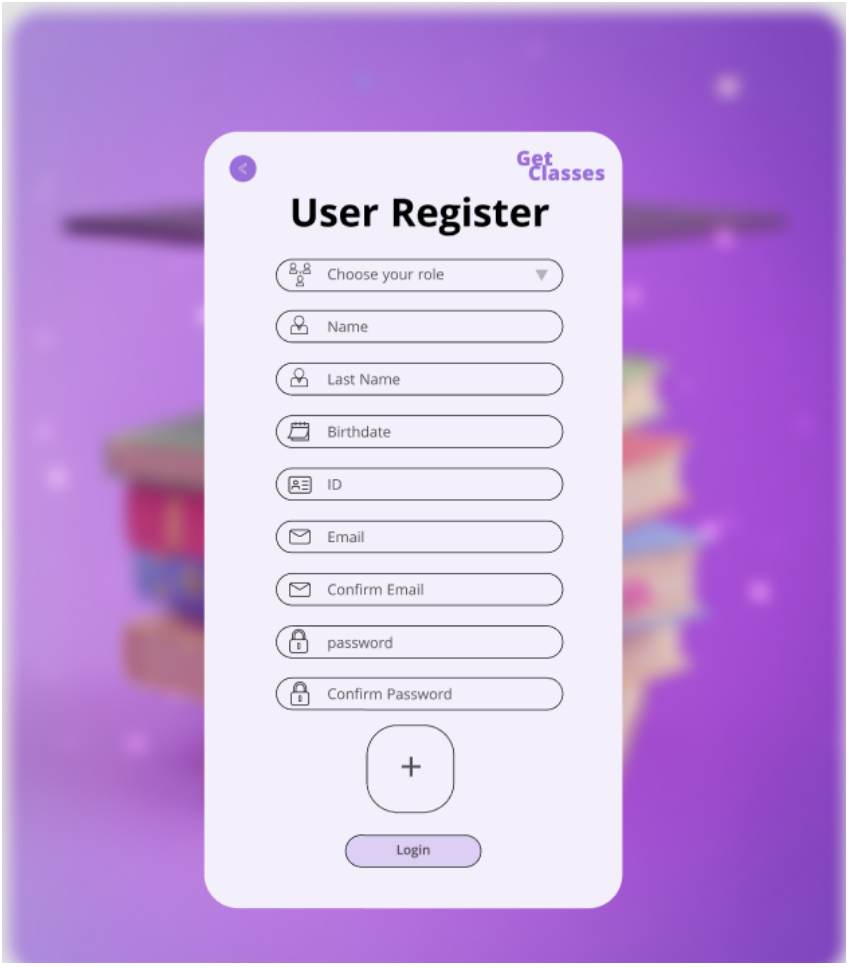


Figure 3: Register Page

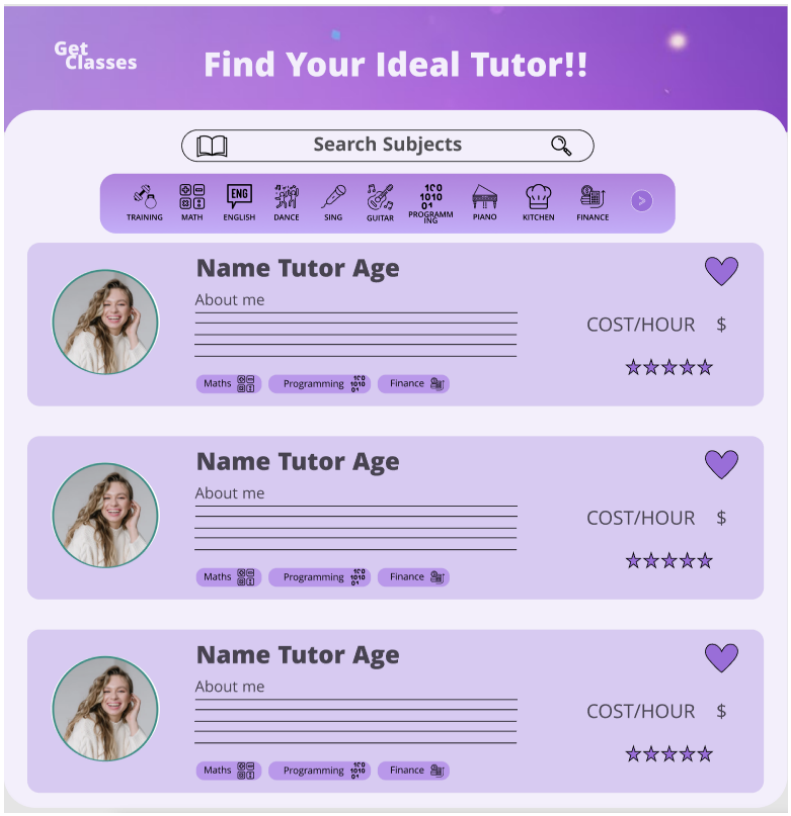


Figure 4: Main Page

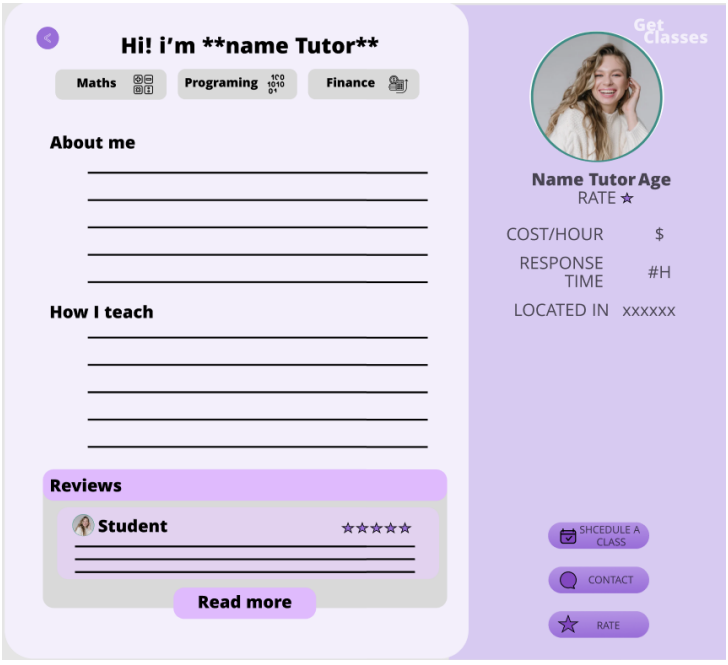


Figure 5: Tutor Profile

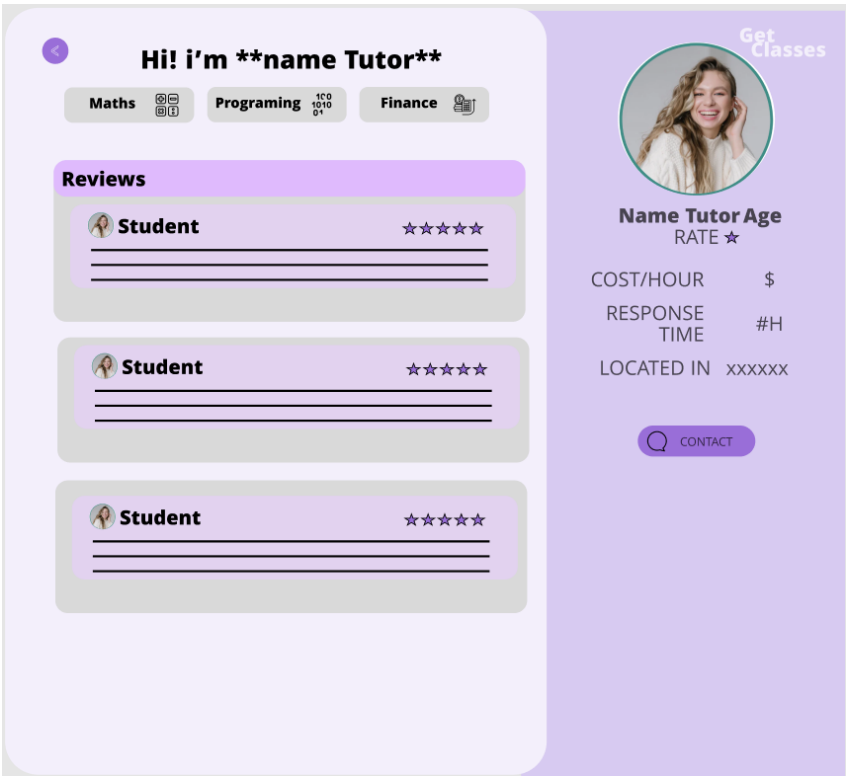


Figure 6: Reviews

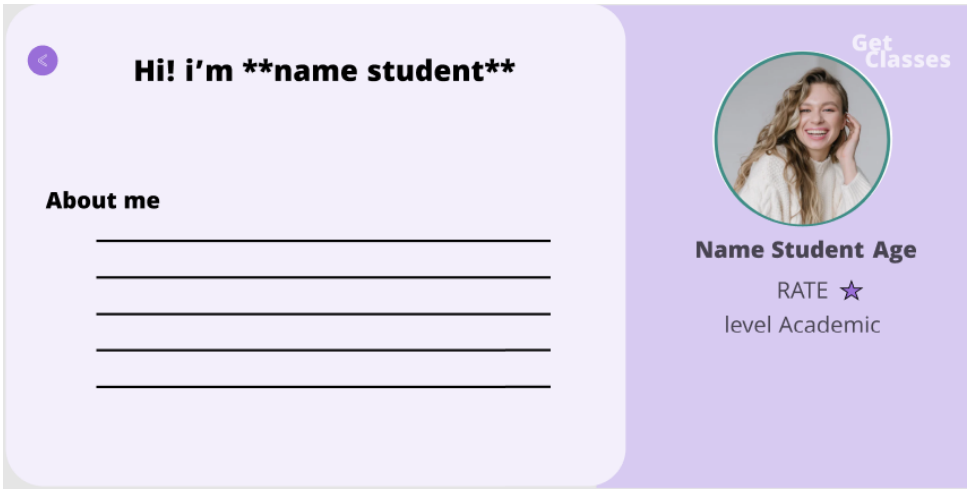


Figure 7: Student Profile

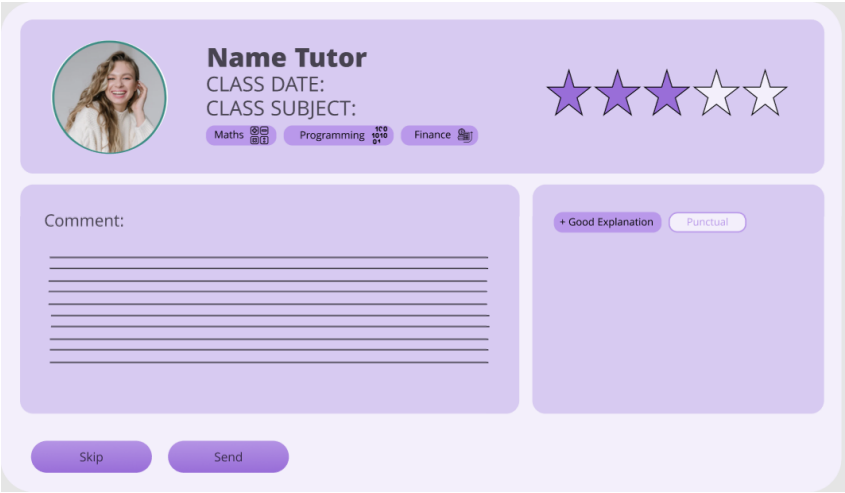


Figure 8: Make Review

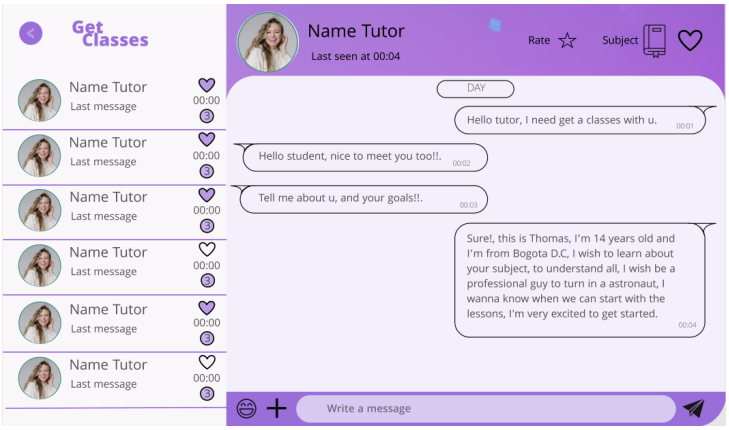


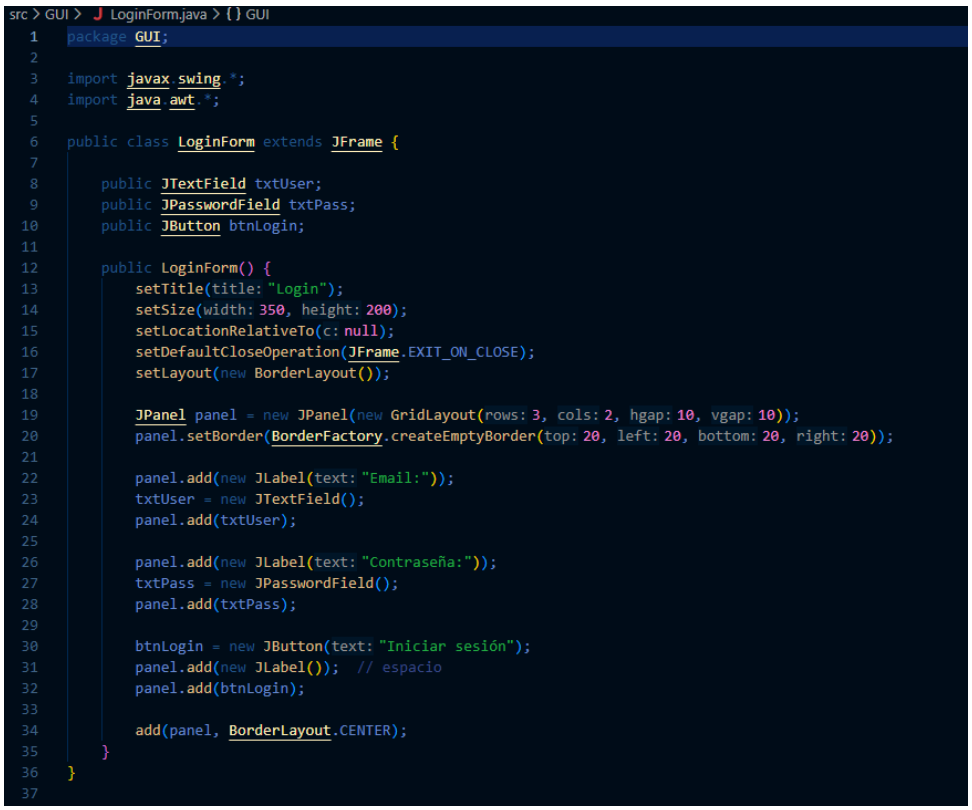
Figure 9: Chat Page

8 Code Snippets

Figure 10: Class User Code

```java
package DataBase.DTO;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
import Classes.Review;
import Classes.UserStudent;
import Classes.UserTeacher;

public class StudentInfoDTO {

    public int id;
    public String aboutMe;
    public String academicLevel;
    public List<Review> sendedReviews;


    public StudentInfoDTO(){
        this.sendedReviews = new ArrayList<>();

    }

    //setters
    public void setAboutMe(String aboutMe) {
        this.aboutMe = aboutMe;
    }

    public void setAcademicLevel(String academicLevel) {
        this.academicLevel = academicLevel;
    }

    public void addSendedReviews(int id, int tutorId, int studentId, int score, String comment, LocalDate date) {
        Review review = new Review(id, tutorId, studentId, score, comment, date);
        sendedReviews.add(review);
    }

    //getters
    public String getAboutMe() {
        return aboutMe;
    }

    public String getAcademicLevel() {
        return academicLevel;
    }

    public List<Review> getSendedReviews() {
        return sendedReviews;
    }
}
```

Figure 11: Student Info DTO Code



```
src > GUI > J LoginForm.java > {} GUI
```

```java
package GUI;

import javax.swing.*;
import java.awt.*;

public class LoginForm extends JFrame {

    public JTextField txtUser;
    public JPasswordField txtPass;
    public JButton btnLogin;

    public LoginForm() {
        setTitle(title: "Login");
        setSize(width: 350, height: 200);
        setLocationRelativeTo(c: null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout());

        JPanel panel = new JPanel(new GridLayout(rows: 3, cols: 2, hgap: 10, vgap: 10));
        panel.setBorder(BorderFactory.createEmptyBorder(top: 20, left: 20, bottom: 20, right: 20));

        panel.add(new JLabel(text: "Email:"));
        txtUser = new JTextField();
        panel.add(txtUser);

        panel.add(new JLabel(text: "Contraseña:"));
        txtPass = new JPasswordField();
        panel.add(txtPass);

        btnLogin = new JButton(text: "Iniciar sesión");
        panel.add(new JLabel());  // espacio
        panel.add(btnLogin);

        add(panel, BorderLayout.CENTER);
    }
}
```

Figure 12: Login Form Code

9 Layer Review and System Redesign

As part of the refinement process, the system architecture proposed in earlier workshops was reevaluated. This review revealed the need to simplify responsibilities and reduce unnecessary complexity.

The project now focuses on addressing the requirements of a layered design that emphasizes **simplicity, effectiveness, and low coupling**. To achieve this:

- Design parameters were adjusted to better support modularity.
- Several structural components were remodeled for clarity.
- Responsibilities were reorganized to ensure functional separation.
- Redundant or overly complex relations were removed or refactored.

This redesign allowed the system to become more coherent, stable, and easier to implement in subsequent stages.

10 JavaFX Monolithic Architecture

After evaluating architectural alternatives, the final implementation will be developed as a **JavaFX monolithic application**. The main reasons supporting this decision include:

- It provides a simple, unified structure easy for new developers to understand.
- Internal modularity is maintained without the complexity of microservices.
- It enables a clear demonstration of OOP concepts (inheritance, polymorphism, encapsulation).
- It is easy to run, test, package, and evaluate in an academic setting.

The monolith integrates the GUI, business logic, and data access layers into a single executable, while still following best practices to prevent excessive coupling.

11 UML Updates and SOLID Principles

As part of the documentation update, UML diagrams were revised to reflect the architectural corrections and the explicit adoption of **SOLID principles**. These principles supported:

- Clearer assignment of responsibilities within classes.
- Improved extensibility without modifying existing, stable code.
- More robust and flexible dependency structures.
- Better alignment between class behavior and system requirements.

The updated UML diagrams include more coherent hierarchies, clearer relationships, and reorganized methods according to object-oriented best practices.

12 Methodology and Deliverables

1. Layer Review and Design Validation

- Class diagrams and design documents from previous workshops were validated against the final layered architecture.
- Adjustments and refactoring were applied to reinforce low coupling and clear responsibilities.
- The architecture continues to build incrementally on prior deliverables.

2. GUI Prototype using JavaFX

- Initial forms and windows were implemented for core actions such as creating transactions and listing elements.
- The GUI prioritizes simplicity and functionality over aesthetics.
- Windows follow basic design patterns to enhance maintainability.

3. File-Based Storage

- Methods for serializing and saving business objects to files were implemented.
- The application can reload persistent data when restarted.
- Basic validations and conflict handling were added to prevent corrupted data.

4. Documentation and Artifact Submission

- Updated UML diagrams (class and sequence diagrams) were generated, showing communication between layers in the final solution.
- Code snippets from the GUI and data access logic were included for traceability.
- Clear documentation was added describing how this workshop builds upon previous ones and what refinements were introduced.

Notes and Reflection

Conceptual and Technical Updates

This version integrates all revisions made after the final review of layered design and system architecture. The following updates summarize the adjustments applied:

- **Layer Alignment:** The conceptual design was refined to better match the layered architecture, ensuring low coupling and clear distribution of responsibilities across components.
- **Design Adjustments:** Several structures were remodeled to simplify interactions, remove redundancies, and improve coherence between business logic and data management.
- **UML Enhancements:** Class diagrams were updated to incorporate inheritance, encapsulation, and method overriding. SOLID principles were applied to reorganize responsibilities and strengthen system extensibility.
- **JavaFX Monolith Decision:** The implementation strategy was adjusted to adopt a simple JavaFX monolithic structure, allowing easier understanding, faster prototyping, and more controlled integration of GUI, logic, and data persistence.

Additional Notes

- Documentation remains in English to maintain consistency throughout all workshops and deliverables.
- External references should be included when UML notations, CRC methods, or JavaFX design decisions rely on academic or technical sources.
- This updated version consolidates previous work and prepares the foundation for the implementation phase.

Final Reflection

Throughout this workshop, the primary goal was to refine the system's structure while preserving clarity and simplicity in design. The adjustments implemented helped align the conceptual model with the technical architecture, ensuring that inheritance, polymorphism, and encapsulation are consistently applied across the system. These improvements also support a smoother transition toward the JavaFX prototype, updated UML models, and file-based storage implementation. Future iterations will focus on completing the GUI, validating the interaction flows, and reinforcing the architectural coherence of the final solution.

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)*.