TinderApp: Simplified Solution for Connecting Users

J. Carvajal Garnica
Systems engineering, school of Engineering
Universidad Distrital Francisco José de Caldas
jcarvajalg@udistrital.edu.co
Bogotá, Colombia

A.M Cepeda Villanueva
Systems engineering, school of Engineering
Universidad Distrital Francisco José de Caldas
amcepedav@udistrital.edu.co
Bogotá, Colombia

***Abstract*—(i) In the context of today's world, having daily interaction with technological tools, there is the feasibility of creating an application that facilitates communication between Internet users and, in turn, creates affective relationships between them. (ii) As a solution, we propose the creation of a dating app, taking the Tinder dating app as a reference, using an object-oriented organization and Java as the main tool. Feature a profile creation, interaction system, and a notification tool.**

***Index Terms*—Dating apps, object-oriented programming, user interaction, Java development, user experience, online safety**

I. Introduction

Online dating apps have transformed the dynamics of social interaction, satisfying the human need for emotional relationships. Platforms such as Grindr (founded in 2009), Bumble (founded in 2014), and Tinder (founded in 2012) exemplify this success. With 3.34 million monthly downloads in the Americas [1], Tinder is considered the most popular dating app in the world. However, key challenges remain: 60% of users abandon these social platforms due to a lack of control over their interactions [2] and intuitive designs that hinder the user experience, as says the book Alone Together [3], highlighting this paradox of digital connection versus real isolation.

Users constantly require systems that foster more authentic and natural interactions. Tinder, the world's most popular dating app, is constantly innovating its servers. It is designed to connect people with similar interests. It works with a swipe system, where the user can swipe right if they are interested in someone or left if they are not. If two people swipe right on each other, a match is created, allowing them to start a conversation. Tinder prioritizes location, common interests, and recent activity to suggest potential matches. However, according to Statista [1], on Tinder, women are more likely than men to receive sexually explicit messages or images without their consent, as well as insults, which represents a security alert. Bumble proposed a solution to this problem of safe interaction between users. In this app, only women can initiate conversations after a match [4]. This seeks to encourage more respectful interactions and reduce these unwanted messages for women by introducing a safe system in its functionalities for the app's users.

We envisioned creating a lightweight version of a dating app like Tinder, built using Java's object-oriented principles. The main goal is not only to replicate the key features of these platforms, such as creating profiles, chatting, and receiving instant notifications, but also to build the application using ideas for system quality, through SOLID principles.

Previous solutions in the dating app space often rely on complex back-end services or frameworks that can obscure the fundamental object-oriented architecture. However, our project focuses on educational clarity and technical rigor, building the system from the ground up using essential Java components, Swing for the graphical interface, and UML diagrams for documentation. Although some concepts like inheritance and interfaces were intentionally excluded in this iteration for simplicity, the architecture remains extensible and well-structured for future development.

This article discusses our journey, how we built it, the essentials of doing so, and what we discovered with our test version. We discuss turning user stories into system actions, how models become a visual interface, and how we manage data and files. The result is a modular prototype that demonstrates the concept and serves as a solid example of object-oriented design.

II. Methods and Materials

A. General Design

The system was designed following a modular architecture rooted in Object-Oriented Programming (OOP) principles. Each component is responsible for a single function, which promotes the readability, maintainability, and scalability of the code. The core components include User, App, Notification, and Photo, each implemented as a separate class.

This structure allows for clear delegation of tasks: users can register, edit profiles, and interact with others, while the application orchestrates the logic. Notifications manage feedback, and the photo system allows image association with profiles.

In the updated version of the project, the system also integrates a graphical user interface (GUI) using Java Swing, improving usability and aligning with the mock-ups created during the design phase. The interface includes screens for login, registration, profile completion, and interaction, with

components such as JFrame, JPanel, JLabel, JTextField, JButton, and JPasswordField.

Each module is loosely coupled and connected through method calls, allowing for future extensions such as database integration, admin profiles, or chat features without refactoring the current logic.

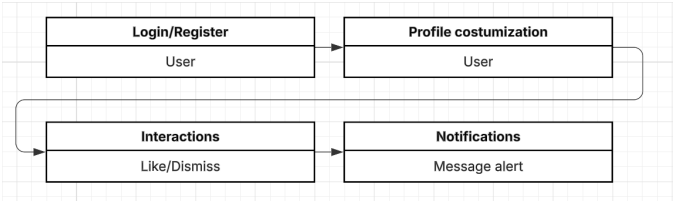

Fig. 1. Modular organization diagram

B. Design process

With the main modules defined, the next step was to establish the system requirements, classifying them as functional and nonfunctional. This distinction provided us with a clear development guide that serves as a rubric that guides both the design and validation of the final product.

From these, the following core features were implemented:

- User Registration and Login: Email and password-based registration. Duplicates are checked using a validation function to ensure unique emails.
- Image Upload: Users can upload a profile picture, stored via the Photo class, and referenced from their user object.
- Interaction system: Users browse other profiles using indexed navigation (excluding themselves) and perform 'like' or 'dismiss' actions.
- Notifications: Notifications are stored and linked to users when a like is given.
- Data Persistence: All user data are saved and loaded using Java file I/O classes (FileWriter, BufferedReader). The format is CSV-like, allowing for easy parsing and object reconstruction.
- Swing GUI: Each action flows through buttons and input fields that trigger logical methods from the core classes using ActionListener. User stories were essential for defining the expected behavior of each module from the user's perspective. From them, acceptance criteria were determined, which define the minimum requirements for each action to be considered successful within the application's user flow.

Once the user stories were developed, we began the technical design applying the object-oriented programming paradigm. We used a top-down approach based on functional analysis, defining the necessary classes, their responsibilities, attributes, and methods. Since the project is intended to be implemented in Java, the class structure was crucial to organizing the system in a coherent and extensible manner.

As part of the object-oriented design, guided by information obtained from Object-Oriented Analysis and Design [5] and Object-Oriented Software Construction, Second Edition [6], we applied encapsulation by declaring class attributes as private and controlling access through public getter and setter methods. This ensured data integrity and protected internal logic, such as password validation and profile visibility, from being accessed or modified directly by other classes. In contrast to earlier drafts, inheritance and polymorphism were not used, as there was no need for hierarchical behavior. The team prioritized simplicity and clarity over abstract generalizations. However, the current design supports future implementation of features such as admin roles, which could use inheritance if needed.

Drawing on requirements, user stories, and CRC cards, we visually modeled the interaction between objects using UML diagrams, specifically class, sequence diagrams, and activity diagrams. This made it possible to clearly represent attributes, methods, and relationships between entities, as well as the application of principles such as encapsulation and inheritance.

As seen in Figure 2, the class diagram represents a simplified version of the conceptual model, where the abstract account class gives rise to user and administrator through inheritance. Multiplicities were used to indicate one-to-many relationships, as well as dependencies between classes that only exist if others are active. For example, Photo makes no sense without User.

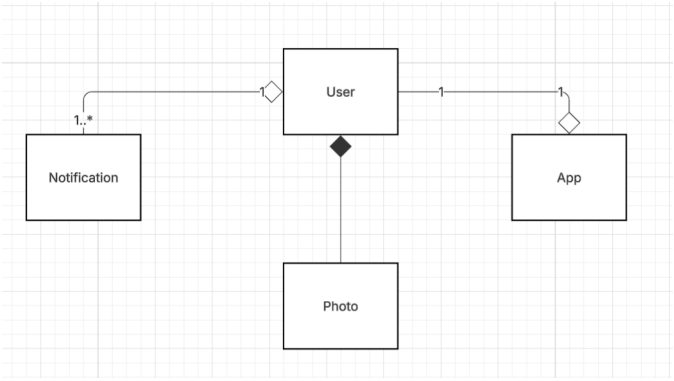

Fig. 2. Simplified UML class diagram

Finally, sequence diagrams made it possible to visualize how classes exchange messages and respond to each other, which served as a basis for initiating the structure of the code in its preliminary version ("work in progress").

Visual models such as UML class diagrams, sequence diagrams, and activity diagrams were created to document object responsibilities, message flow, and system behavior. These models guide both development and future maintenance.

C. Code structure

The project is implemented entirely in Java, using packages and separate class files to organize responsibilities. Naming conventions follow CamelCase, and the code is documented with inline comments to support collaborative development.

Each core class performs specific duties:

- User: Stores personal data and preferences. Includes methods like register(), login(), giveLike(), and editProfile().
- App: Acts as the central coordinator. It loads users from the file, handles navigation, and controls GUI logic transitions.
- Notification: Manages messages related to likes, matches, and other relevant actions. Accessible through addNotification() and showAll().
- Photo: Handles photo upload and management. Only safe methods are exposed to access or replace the image reference.

The main class (Main) launches the application and shows the welcome screen GUI (showWelcomeScreen()), which links to login or registration. Every action in the GUI is directly tied to backend logic, ensuring consistency with user expectations.

Java Swing is used to build the GUI. Layouts like BoxLayout and GridLayout are used to maintain visual consistency. Navigation is triggered by ActionListener methods, which call corresponding logic functions (e.g., registering a new user, displaying a profile, showing matches).

Data is stored persistently across sessions. Upon startup, the application reads the users.txt file to reconstruct registered users. Before termination or after registration, the system writes the updated user list back to the file. If the file is missing or corrupted, appropriate error messages are displayed using dialog boxes.

D. Application of SOLID principles

To improve maintainability and design clarity, several SOLID principles were applied in the implementation:

TABLE I
SOLID PRINCIPLES IN THE PROJECT

Principle	Applied	Where in the Project
SRP	Yes	Each class has one responsibility (`User`, `App`, etc.)
OCP	Yes	New features can be added without modifying existing code
LSP	Partial	Inheritance not used yet, but design allows future extension
ISP	Yes	Only relevant methods are exposed per class
DIP	Basic	`App` depends on methods, not internal implementations

III. RESULTS

To validate the correctness and reliability of the system, we designed a testing plan that included unit tests, integration tests, and acceptance evaluations. The primary objective was to ensure that all functional requirements and user stories were fulfilled, and that the user experience was consistent with the intended design.

A. Experiment Setup

Testing was performed on a local Java development environment using simulated user inputs. No external libraries or frameworks were used. All components were tested in isolation and then integrated through the graphical interface. We focused on verifying input validation, behavior consistency, and data persistence.

B. Unit Testing

Each core method was tested individually using representative inputs. The goal was to confirm that methods returned correct outputs, handled edge cases (e.g., invalid emails, empty fields), and did not produce runtime errors.

C. Integration and Acceptance Testing

After verifying the functionality of each method, we ran full system simulations through the Swing GUI. These tests involved complete user flows:

- **User Flow 1:** Register → Complete Profile → Upload Image → Like another user → Receive Notification.
- **User Flow 2:** Log in with invalid credentials → Handle error → Log in successfully

Both flows were executed successfully, and expected screen transitions occurred without failures. Data was persisted between runs, and user interactions behaved as designed.

Acceptance testing was conducted based on the defined user stories. Each story was verified against its acceptance criteria. All tests were marked as "Passed."

TABLE II
ACCEPTANCE TEST RESULTS

User Story	Result
Register and log in	Passed
Complete and edit profile	Passed
Upload image	Passed
Like/dismiss profiles	Passed
Receive notification on match	Passed
Log out	Passed

D. Comparison and Limitations

Unlike production-level dating apps like Tinder or Bumble, our system is built as a prototype. It does not include real-time back-end services or databases. However, for its scope, it successfully replicates key functionalities using file storage and offers a responsive GUI with user feedback.

Performance remains stable with dozens of users. For larger-scale deployment, a transition to a relational database and concurrency control would be required.

IV. CONCLUSIONS

The development of the *TinderApp* prototype allowed us to demonstrate the correct application of Object-Oriented Programming principles in a real-world inspired scenario. Through the implementation of features such as user registration, profile customization, mutual interaction, and notification handling—along with a functional GUI built with Java Swing—the project met all defined requirements.

The use of encapsulation and abstraction helped structure a maintainable and modular codebase. While inheritance and

polymorphism were intentionally excluded for simplicity, the architecture remains extensible for future roles or behavioral variation. The adoption of SOLID principles further ensured that the system could evolve without compromising existing functionality.

The test plan, which included unit, integration, and acceptance tests, confirmed the functional validity of the core features. File-based persistence proved sufficient for the current scale, allowing data continuity between sessions. The GUI facilitated a smooth user experience aligned with the planned mockups.

Overall, the project not only fulfilled its technical goals but also served as an effective academic exercise in software design, architecture, and documentation. Future iterations could integrate advanced features such as matchmaking algorithms, admin roles, or database-backed persistence, building upon the solid foundation established here.

REFERENCES

[1] S.J. Dixon,"Tinder - statistics & facts," Statista. 2024, [Online]. Available: https://www.statista.com/topics/10082/tinder/

[2] Pew Research, Pew Research. 2023, [Online]. Available:https://www.pewresearch.org/

[3] S. Turkle, *Alone Together: Why We Expect More from Technology and Less from Each Other*. New York, USA: Basic Books, 2011.

[4] U. Peñaloza, "10 apps de citas gratuitas alternativas a Tinder," UrielPeñaloza. 2025, [Online]. Available: https://urielpenaloza.com/blog/10-apps-de-citas-gratuitas-alternativas-a-tinder/

[5] G. Booch, *Object-Oriented Analysis and Design*. USA: Benjamin Cummings Publishing Co, 1993.

[6] B. Meyer, *Object-Oriented Software Construction, Second edition*. España: Prentice Hall, 1997.