Universidad Distrital Francisco Jose de Caldas

Department of Computer Engineering

Technical report: Messager app project development

Andres Camilo Ramos - 20242020005

Supervisor: Carlos Andres Sierra

A report submitted in partial fulfilment of the requirements of
the Universidad Distrital Francisco Jose de Caldas for the degree of
faculty of Engineering in *Systems Engineering*

July 11, 2025

Declaration

I, Andres Camilo Ramos, of the Department of Computer Engineering, Universidad Francisco Jose de Caldas, confirm that this is my own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

I give consent to a copy of my report being shared with future students as an exemplar.

I give consent for my work to be made available more widely to members of UoR and public with interest in teaching, learning and research.

Andres Camilo Ramos
July 11, 2025

Abstract

This project aims to develop a solution for secure and private digital communication. In today's increasingly connected world, access to the internet has made digital communication more accessible and essential for everyday life. However, the ease of communication often comes with concerns about privacy and security, especially when relying on public or commercial platforms. To address this, the initiative focuses on creating a dedicated private communication application designed to facilitate safe and reliable interactions between individuals across different locations. The application is intended to empower users by giving them control over their communication channels, reducing dependence on third-party services, and ensuring that conversations remain confidential. Through thoughtful design and a focus on user needs, the project seeks to bridge distances and foster secure connections in an increasingly digital world..

Keywords: Project guidelines, engineering education
Contents, list of figures, list of tables

Contents

1 Introduction **1**
- 1.1 Context . . . 1
  - 1.1.1 Historical context . . . 1
  - 1.1.2 What is meant to be solved . . . 2
  - 1.1.3 Project context . . . 2
- 1.2 Objectives . . . 2

2 Literature Review **3**
- 2.1 Object-oriented programming benefits . . . 3
  - 2.1.1 Modularity . . . 3
  - 2.1.2 Inheritance . . . 3
  - 2.1.3 Polymorphism . . . 3
- 2.2 Design . . . 3
  - 2.2.1 SOLID . . . 4

3 Objectives **6**

4 Scope **7**
- 4.1 Boundaries of the project . . . 7
  - 4.1.1 What it's included . . . 7
  - 4.1.2 What it's not included . . . 7

5 Assumptions **8**
- 5.1 What you need to know . . . 8
  - 5.1.1 About OOP . . . 8

6 Limitations **9**
- 6.1 Database Limitations . . . 9
- 6.2 Interface Limitations . . . 9
- 6.3 Data Limitations . . . 9

7 Methodology **10**

8 Results **12**

9 Discussion and Analysis **13**

10 Conclusions and Future Work **14**
- 10.1 Conclusions . . . 14
- 10.2 Future work . . . 14

References **15**

Chapter 1

Introduction

1.1 Context

This is a object oriented project that pretends to solve communication

1.1.1 Historical context

This project aims to provide a practical and modern solution to persistent communication challenges by focusing on digital connectivity. In today's interconnected world, the ability to communicate remotely through the internet is essential for various sectors such as academic research, politics, business, and everyday social interactions. As the global demand for secure and efficient communication increases, it becomes crucial to develop systems that can meet these evolving needs. Assuming internet access is available, this project proposes a tailored digital communication platform to bridge the gap between individuals in different locations.

Historically, communication was limited to local networks where users could only exchange information within the same system or infrastructure. One of the earliest milestones in digital messaging was the first message sent through the Compatible Time-Sharing System (CTSS) (Wikipedia contributors, 2024) at the Massachusetts Institute of Technology. While groundbreaking at the time, it did not constitute a fully functional email system as we understand it today, primarily due to its inability to connect over the internet and other technological limitations of the era.

Over time, several solutions have emerged to facilitate digital communication. Applications like Microsoft Outlook (Microsoft Learn, 2025) introduced the concept of email addresses, enabling users to connect with colleagues, friends, and even strangers for work or research purposes. Outlook also offers domain services, allowing businesses to personalize their email platforms, which brings additional benefits in branding and security. However, while powerful, such platforms are not always optimized for real-time, chat-based interactions. On the other hand, modern messaging apps like Line (LINE Corporation, 2025) and Telegram (Telegram, 2025) more closely resemble the communication style this project aims to implement. These applications are built for instant messaging, providing users with account-based access using secure login credentials such as usernames and passwords

.

1.1.2 What is meant to be solved

This project directly addresses the problem of ineffective communication caused by cluttered digital environments. In both professional and personal contexts, there is a clear need for a focused communication channel that is not subject to the constant influx of unsolicited content. While voice calls offer an alternative, they are limited by the mutual availability of participants and lack a persistent, searchable history of the conversation, which is often crucial for accountability and reference. Furthermore, popular messaging applications like WhatsApp, while ubiquitous, often blur the lines between personal and professional life, and many users are uncomfortable using a single platform for all interactions. Therefore, we propose the development of a private messaging application designed specifically for targeted communication, allowing users to create a distinct space for work or other specific purposes, thereby avoiding the inconvenience of content mixing and digital noise.

1.1.3 Project context

This project is being developed using Object-Oriented Programming (OOP) (Meyer, 1997) principles, which emphasize modularity, reusability, and organized code architecture (Onu, 2015). The design phase began with the creation of a theoretical prototype, including class diagrams, attribute definitions, and interface mockups. This process allowed for a clearer understanding of the system's structure and functionalities. The practical development started by identifying the core components necessary for the application. These include a User class to represent individual accounts, a Chat class to manage conversations, and a Message class to handle individual messages. By structuring the application this way, the project avoids the immediate need for complex database management while maintaining a clear organization of data and processes. As development progresses, the focus remains on translating this theoretical model into a functional, user-friendly platform that ensures reliable and private communication.

1.2 Objectives

- To obtain a functional messaging application that meets the proposed requirements.
- Understand how object-oriented programming can solve certain issues

Chapter 2

Literature Review

In this section you will find context to understand how the program was re-implemented; additionally, you will find information regarding previous solutions that are considered not to solve the issue.

2.1 Object-oriented programming benefits

There are many projects that use object-oriented programming, even today it is most common for companies to use it due to the advantages it offers through its design, including the ability to see a scalable and modular system from the beginning of its development, thus allowing developers to have the possibility to make improvements to the systems if they become necessary.

2.1.1 Modularity

Among the benefits of object-oriented programming is modularity, which seeks to facilitate problem-solving by separating them into more manageable parts, which also helps in the distribution of work in some development companies.

2.1.2 Inheritance

Inheritance allows for code reuse as well as facilitating the application of modularity, enabling us to divide our system into smaller parts (TOP-DOWN) to make it easier to manage.

2.1.3 Polymorphism

Polymorphism, also described as a flexibility in data types, allows the programmer to call a class in a method, provided that this class has derived classes, which will allow that method to accept an object generated by that derived class, thus facilitating development and maximizing code reuse.

2.2 Design

There are several types of design in object-oriented programming, in this case the SOLID design is sought for its benefits in understanding concepts related to OOP,

2.2.1 SOLID

SOLID principles are essential guidelines for software design, which guarantee the maintainability and scalability of the code, additionally considered a good practice in the environment due to the benefits it brings when it comes to development, maintenance and longevity of the project in which it is applied.

Single Responsibility Principle

The Single Responsibility Principle states that a class should do one thing and therefore it should have only a single reason to change.

To state this principle more technically: Only one potential change (database logic, logging logic, and so on.) in the software's specification should be able to affect the specification of the class.

This means that if a class is a data container, like a Book class or a Student class, and it has some fields regarding that entity, it should change only when we change the data model.

Following the Single Responsibility Principle is important. First of all, because many different teams can work on the same project and edit the same class for different reasons, this could lead to incompatible modules.

Second, it makes version control easier. For example, say we have a persistence class that handles database operations, and we see a change in that file in the GitHub commits. By following the SRP, we will know that it is related to storage or database-related stuff.

Merge conflicts are another example. They appear when different teams change the same file. But if the SRP is followed, fewer conflicts will appear – files will have a single reason to change, and conflicts that do exist will be easier to resolve.

Open | Closed Principle

The Open-Closed Principle requires that classes should be open for extension and closed to modification.

Modification means changing the code of an existing class, and extension means adding new functionality.

So what this principle wants to say is: We should be able to add new functionality without touching the existing code for the class. This is because whenever we modify the existing code, we are taking the risk of creating potential bugs. So we should avoid touching the tested and reliable (mostly) production code if possible.

But how are we going to add new functionality without touching the class, you may ask. It is usually done with the help of interfaces and abstract classes.

Liskov Substitution Principle

The Liskov Substitution Principle states that subclasses should be substitutable for their base classes.

This means that, given that class B is a subclass of class A, we should be able to pass an object of class B to any method that expects an object of class A and the method should not give any weird output in that case.

This is the expected behavior, because when we use inheritance we assume that the child class inherits everything that the superclass has. The child class extends the behavior but never narrows it down.

Therefore, when a class does not obey this principle, it leads to some nasty bugs that are hard to detect.

Interface Segregation Principle

Segregation means keeping things separated, and the Interface Segregation Principle is about separating the interfaces.

The principle states that many client-specific interfaces are better than one general-purpose interface. Clients should not be forced to implement a function they do no need.

Dependency Inversion Principle

The Dependency Inversion principle states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions.

We want our classes to be open to extension, so we have reorganized our dependencies to depend on interfaces instead of concrete classes. Our PersistenceManager class depends on InvoicePersistence instead of the classes that implement that interface.

Chapter 3

Objectives

This research attempts to clarify whether a private mail type messaging system is useful, mainly as a solution to the problem posed, remembering that many people feel overwhelmed by the amount of emails that arrive in their inboxes, it is hoped that with the information provided about the development of the project, a clear idea will be obtained as to whether the solution is useful or not, in addition to presenting the knowledge acquired throughout the Object Oriented Programming (OOP) course.

- Is it useful for users in the business sector to have a private communication application?
- Is it useful for natural users to have a private communication application?
- Is it helpful to stop receiving emails from websites you register or subscribe to?

Chapter 4

Scope

In this section, the limits encountered for the project will be defined, including also those aspects that will be considered during its implementation.

4.1 Boundaries of the project

Starting with the main limits, we have that the project will be object-oriented and will consist of 2 main classes. The email and the user class

4.1.1 What it's included

- All the main functions evidenced in the design were included as responsibilities, among which the user is allowed to fully use the messaging service, thus enabling them to communicate with other registered people in the system, whom the user can identify.
- The design of interfaces is planned to provide a pleasant and intuitive interaction for the user, thus facilitating the use of the application, primarily thinking about ensuring that people who do not fully understand new technologies are not excluded from the possibility of using the system.

4.1.2 What it's not included

- The decision was made to exclude a database manager of the SQL or Oracle type during the design, due to a lack of knowledge in this area and the possibility that such an implementation could complicate the project excessively, so more traditional database.
- Some update and deletion functions for messages were discarded when discarding a database manager due to the belief that performing an update or deletion in a traditional database would be much more complicated, potentially causing serious damage to the integrity of the stored data, such as messages in this example.

In general, you will find a system that is expected to be developed in its entirety with basic tools and using object-oriented programming as the main approach, additionally learning from it and understanding how its concepts apply in a real project.

Chapter 5

Assumptions

5.1 What you need to know

For this technical report, we hope and assume that as a reader you will have some idea, however vague, of what object-oriented programming entails. Similarly, the methodology will describe some logical processes to arrive at the proposed solutions and achieve the objectives. It is expected that you have basic concepts about programming, in addition to the ability to understand some diagrams that will be included to provide a better explanation, although the diagrams will be high-level, so no additional complications are expected for them to be understandable at a general level.

5.1.1 About OOP

We hope that the explanation of object-oriented programming has been conclusive in recalling certain concepts of it, as well as highlighting what it implies and how it limits in a certain way. This is general because it is expected that the basics are understood, so that later when we specifically discuss why decision-making or the project flow was carried out in this way, it will be comprehensible for the reader.

Chapter 6

Limitations

6.1 Database Limitations

As the first limitation, we found that not using a database manager was an impediment for some ideas that were in the initial design. Without having a database available, we found that some operations for the records would be complicated, which is why certain methods and actions that are complex using the available tools were eliminated.

6.2 Interface Limitations

To create the interface, Java Swift was used. This constitutes a limitation since Swift is a fairly obsolete application and therefore the interfaces are very simple.

6.3 Data Limitations

Limitations were found with the data, to perform a comparative analysis between the practicality of the applications, satisfaction data would be needed from users who have used a regular email and the developed one for a considered time, to have an idea of which seemed better, in this case the software was not implemented in a real environment and with numerous users, which prevents a true satisfaction rate from being achieved with respect to both applications, additionally the limitation of the internet connection was found, the developed service is a local application so it is not possible to distribute it to do tests in a real environment.

Chapter 7

Methodology

Analysis and Architectural Foundation

From its inception, the project was architecturally grounded in the object-oriented programming (OOP) paradigm. This was a strategic decision, not merely a choice of style, but a commitment to building a system that was inherently modular, scalable, and maintainable over the long term. The core tenets of OOP provided direct benefits. Encapsulation, for example, allowed us to bundle data (attributes) with the functions (methods) that operate on them, creating robust, self-contained classes that protected their internal state from outside interference. This structure is instrumental in simplifying development and debugging, as it effectively isolates functionalities, reduces complex interdependencies, and minimizes the ripple effect of any future changes.

Following this paradigm selection, we initiated a critical abstraction process to distill the application's essential purpose. The primary goal was to filter out superfluous features and concentrate exclusively on the core functionality required to solve the problem of cluttered communication. This involved a rigorous inquiry into the system's absolute necessities, posing fundamental questions like, "What is the minimum set of attributes needed to uniquely identify a user and their state?" and "What elemental components constitute a single, complete message?" This disciplined analysis led us to the conclusion that the entire system could be effectively and efficiently modeled with just two primary classes: a User class, responsible for managing the registration, authentication, and identification of each individual with system access, and an Email class, designed to encapsulate and store the content, metadata, and state of messages exchanged between users. In accordance with OOP design principles, each class was meticulously conceptualized with its own distinct attributes and behaviors, ensuring the correct and independent functionality of each component. With this lean foundational architecture established, we proceeded to define the project's functional and non-functional requirements. This step was crucial for creating a clear and unambiguous roadmap, outlining specific deliverables, which provided a clear mandate for all subsequent development phases.

System Design and Modeling

Moving into the design phase, our focus shifted to translating the abstract requirements from the analysis into a concrete, detailed technical blueprint. For this, we relied heavily on the industry-standard Unified Modeling Language (UML) to visually represent the system's architecture and behavior. This formal design process began with the creation of Class-Responsibility-Collaboration (CRC) cards for our proposed User and Email classes. This hands-on exercise proved invaluable for validating our initial class structure and for clearly delineating the responsibilities of each entity. For instance, it solidified that the User class was solely responsible for managing credentials and its associated message collections, while the Email class was responsible only for holding the data of a single message.

With these responsibilities clarified, we developed a detailed class diagram that served as the static architectural guide for the entire development process. This diagram meticulously detailed the specific methods for each class. Critically, it also specified the relationships between them—a one-to-many relationship where a single User could possess collections of Email objects. To complement this static view, we designed sequence and activity diagrams to map out the system's dynamic behavior and user flow. These UML diagrams illustrated the precise, step-by-step interactions expected within the system, modeling use cases like a user logging in, composing a message, and successfully sending it. This practice was instrumental in anticipating and mitigating risks, allowing us to cover all expected pathways and minimize the potential for unexpected runtime errors. The final design solidified the two-class model, where the Email class contained standard attributes like subject, content, and sender/receiver information, and the User class held basic user data, with the username serving as the key identifier for message routing. The resulting diagrams outlined a straightforward interaction flow, simplifying the system by focusing exclusively on text-based exchange and preventing overly complex user actions that could lead to system failures.

Implementation, Refactoring, and Testing

The coding phase represented the most tangible stage of progress, where our theoretical designs were translated into a functional application. This phase proved to be extensive and highly iterative, primarily because the process of implementation revealed unforeseen complexities and flaws in our initial design. We discovered that some of the classes originally proposed in the early UML diagrams, such as separate Inbox and Outbox managers, were superfluous and unnecessarily complicated the application's logic. Consequently, a significant refactoring effort was undertaken to streamline the architecture down to the two essential User and Email classes. This involved revising their methods and responsibilities and introducing a main controller class, implemented using a Singleton design pattern, to manage the in-memory storage of all User and Email objects. This controller acted as a centralized, temporary data store, preventing data loss during a runtime session and simplifying object retrieval across the application. These critical adjustments were made in close consultation with the project supervisor, ensuring the final product was both functional and architecturally sound.

Once the core logic was deemed robust and complete, our focus shifted to enhancing the user interface. The initial console-based interaction, while functional for testing, was not intuitive for an end-user. Therefore, we developed a simple graphical interface designed to lower the user's cognitive load and guide them through each step of the messaging process, thereby minimizing the potential for input errors. The development phase concluded with a multi-tiered testing strategy. This included unit tests to verify the correctness of individual methods within each class, integration tests to ensure that the end-to-end flow of sending and receiving messages worked seamlessly, and informal user acceptance testing to gather feedback on the usability and clarity of the new graphical interface. This rigorous process ensured the final application was stable, reliable, and met its core objectives.

Chapter 8

Results

The primary outcome of this project is a functional application prototype that successfully realizes the initial design goals. The resultant codebase establishes a self-contained, closed-loop messaging environment where registered users can securely and directly exchange messages. Comprehensive internal testing has validated the core functionalities of the system. These tests confirm that message transmission between different registered users is reliable and operates as intended, allowing for a focused and efficient communication channel.

This working prototype serves as a successful validation of the project's central concept: that by eliminating external digital noise, communication can become more effective. The system inherently promotes simplicity because users only receive messages pertinent to their direct interactions, from contacts they have personally approved within the network. This stands in stark contrast to conventional email, where inboxes are saturated with a mixture of crucial and non-essential content.

At the current stage, a direct quantitative comparison with established solutions like Gmail or Outlook is not feasible, as the application has not been deployed in a real-world business or organizational setting. Therefore, there are no empirical figures to definitively measure the reduction in message volume a user would experience. However, a substantial difference is logically projected. The application's architecture, by design, prevents the mass sending of information via distribution lists, third-party subscriptions, or promotional blasts. The main constraint of the system—its closed nature—is thus also its greatest strength. To enhance usability and provide greater clarity for end-users, the functional backend is complemented by a clean and intuitive graphical user interface developed in Swift, ensuring a straightforward and error-free user experience.

Chapter 9

Discussion and Analysis

Taking into account the limits6 presented we determine that the program has a very high possibility of being useful for some people, additionally it is taken as a reference that private messaging applications are very useful in the daily life of each person, as previously mentioned WhatsApp is a very good alternative but some people do not feel in agreement in using theirs, in this case we can say that there are companies that give specific numbers to their workers, and through these they communicate, unfortunately not everyone has the availability of having two numbers on their mobile device, and some companies do not have the economic resources to equip their workers in this way; With this clear it is considered that the application is very similar to one of these, therefore it is said that it is useful for employees to have an alternative of this type when communicating with their colleagues, even in the personal sphere it can be useful, but currently people use social networks more for their personal communication

Chapter 10

Conclusions and Future Work

10.1 Conclusions

The "Messager App" project was initiated to address the significant problem of information overload and distraction inherent in modern email communication. We developed a private, closed-loop messaging application using the object-oriented programming paradigm, focusing on creating a dedicated and secure channel for communication. The project progressed from conceptual analysis and UML design to the development of a functional prototype with an intuitive user interface. The primary result is a working application that successfully enables users to send and receive messages within a controlled environment, free from the advertisements, subscriptions, and spam that clutter conventional email inboxes.

The project is considered a success because it fully realized our initial goal: to create a viable alternative for focused communication. By building a functional prototype, we validated our core concept and demonstrated that a closed messaging system can effectively solve the problem of digital distraction. Our work directly helps users by providing them with a tool to separate important, targeted conversations from the noise of their primary email accounts. This allows for more efficient and private interactions, ensuring that critical messages are not overlooked and that users can maintain distinct communication channels for different aspects of their lives, such as professional and personal matters.

10.2 Future work

While working on the project, it was found that there are many possible extensions to the project, for future research or even for real-life applications, an Internet connection and a more user-friendly interface could be developed so that users can give the application a real use, as well as a critique and further develop how good the alternative is in different fields.

References

LINE Corporation (2025), 'LINE official website'. (accessed May 16, 2025).
URL: *https://www.line.me/en/*

Meyer, B. (1997), *Object-Oriented Software Construction*, Prentice Hall. Available at: https://archive.org/details/objectorientedso00meye_0 (accessed May 16, 2025).

Microsoft Learn (2025), 'Outlook documentation'. (accessed May 16, 2025).
URL: *https://learn.microsoft.com/es-es/outlook/*

Onu, F. (2015), 'Impacts of object oriented programming on web application development'. (accessed May 16, 2025).
URL: *https://www.researchgate.net/profile/Fergus-Onu/publication/283202529*$_I mpacts_o f_O bject_O riented_{Pr}$
$of - Object - Oriented - Programming - on - Web - Application - Development.pdf$

Telegram (2025), 'Obtaining API ID'. (accessed May 16, 2025).
URL: *https://core.telegram.org/api/obtaining*$_a pi_i d$

Wikipedia contributors (2024), 'Compatible Time-Sharing System'. (accessed May 16, 2025).
URL: *https://es.wikipedia.org/wiki/Compatible*$_T ime - Sharing_S ystem$