Universidad Distrital Francisco José de Caldas

School of engineering

Technical Report Digital wallet

Software modelling

Cristian Santiago López Cadena - 20222020027

Carlos Alberto Barriga Gámez - 20222020179

Bogotá, D.C.

1. Introduction

This project consists of developing a monolithic software for a digital wallet application with multiple backends which are connected through web services.

The objective of the development of this project is to implement the digital wallet using object-oriented programming, design patterns and SOLID principles.

Now, among the tools used to develop the software, it was decided to use python and java for the development of the backend. As a data management method, it was decided to use json files, due to their ease of handling.
For the tools for managing the services, it was decided to use FastApi to develop the services in the Python backend, while Spring Boot was used in the Java backend.

Regarding the hardware that will be used to make the software, it was decided that it will need 1 GB of RAM, 1 GB of hard drive and a tenth generation Intel Core i5 processor.
About stakeholders, we consider that our software can be functional for those banking companies that want to expand their capabilities and reach with the public. Likewise, those users who require a simple and efficient way to manage their money may be interested in the virtual wallet.

2. User stories

UH_1: As CTO, I want to keep a record of the users who enter the application. I want to know their first and last name, cell phone number, email address, and ID, to know who is using our service.
UH_2: As CTO I want to offer a simple way for users to access their digital wallet, so that everyone can access the service easily and immediately.
UH_3: As CTO I want to be able to send money between different wallets so that users can make fast and secure transfers within the ecosystem.
UH_4: As CTO I want users to be able to link a debit card to their wallet so they can make payments easily.
UH_5: As a user, I want to be able to withdraw money from my wallet at a physical location so that I can access my cash quickly and conveniently when I need it.
UH_6: As a user, I want to be able to load money into my account in different ways so that I can have flexibility and ease when adding funds.
UH_7: As a user, I want to be able to request money from a friend who also has the app so that I can receive the money quickly and easily without complications.

UH_8: As a user, I want to be able to see how much money I currently have and a history of all the transactions made so that I can keep track and monitor my finances.
UH_9: As CTO, I want to know the user's place of residence and their ID to ensure that all legal requirements are met.

3. Functional and no functional requirements

Functional requirements

- Allow quick and secure login for both admins and customers.
- Register a new customer or admin.
- Allow the creation of a single wallet per customer.
- Allow the customer to add funds to their account.
- Allow the customer to send funds to another account.
- Allow the customer to request funds from another account.
- Allow the customer to view the movements they have made with their wallet.
- Allow the customer to create specific pockets in their wallet.
- Allow the customer to add a credit card to their wallet.

No functional requirements

- Be secure for clients and admins.
- Be easy to manage for users.
- Be efficient in its operation.
- Provide ease of maintenance.
- Be scalable according to the needs of the system.
- Be flexible to changes and improvements.
- Continuous availability of the digital wallet.
- Backup of information in a database.

4. Technical decisions

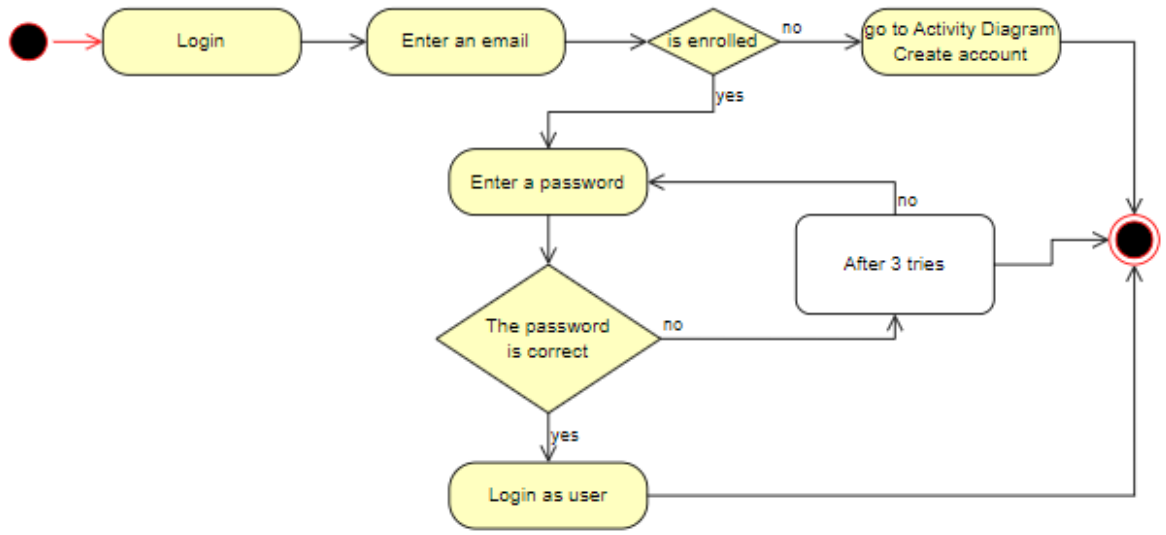
In terms of technical decisions, it was decided to structure the project under the idea of a layered architecture, in order to develop a scalable, maintainable and flexible system.

In this way, the application will be made up of a backend that will have all the logic for the operation of the software, as well as the data layer. The backend will be made in two programming languages, Python and Java.
On the other hand, it was decided that the frontend will be a command line interface, because the project aims to focus on the use of design patterns and good object-oriented programming practices.
Similarly, FastApi and Spring Boot were sought to easily unify the backends made with Java and Python, as well as to have a complete tool for managing the application's web services.
Finally, it was decided to use Docker to perform the software integration, as well as to automate the deployment of the application.

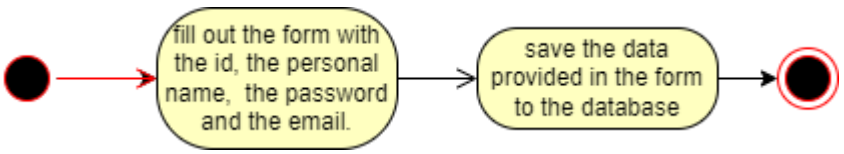
5. UML diagrams

Activity diagrams

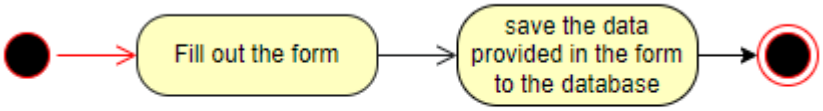
Login: This diagram shows how the user enters an email to log in. If the user is not registered, he is asked to create his account. On the other hand, if he is registered, he enters a password, and if it is correct, the user logs into the application.



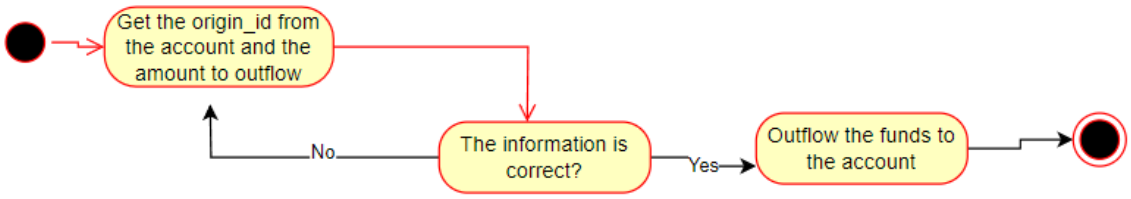
Create account: In this case, the user fills out the form with the ID, the personal name, the password and the email and this data is saved in the database.



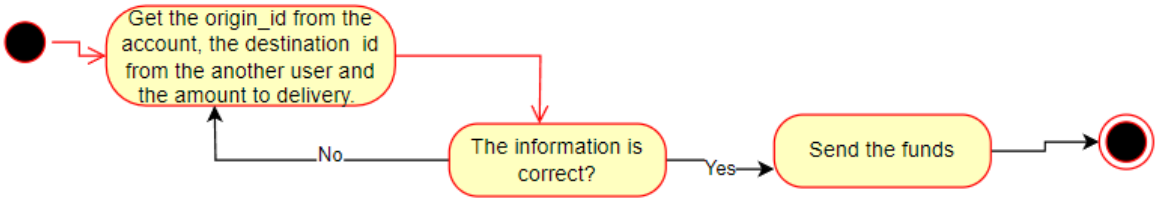
Add address: In this diagram the user fills out the form to add the address and this information is saved in the database



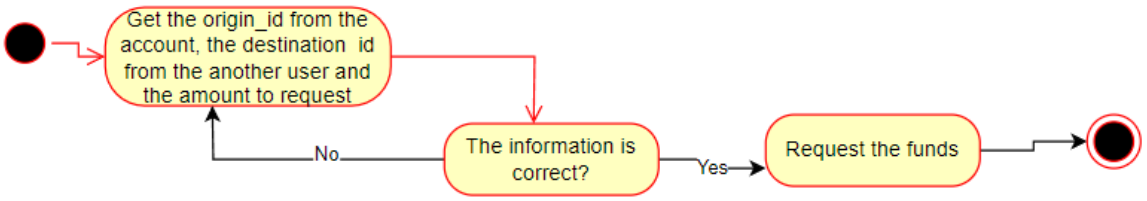
Outflow funds: In this diagram the system receives the account ID and the amount to be outflow, then the system requests verification of the data and when they are correct, the money is outflow from the account.



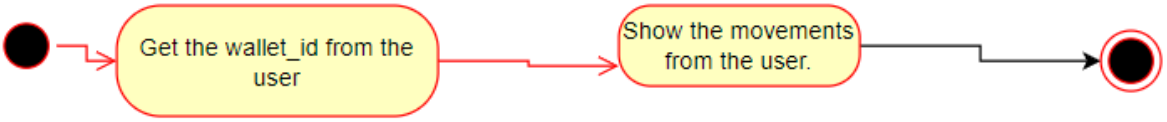
Send funds:
In this diagram the system receives the account ID, the amount to be send to another user and the id from the receiver user, then the system requests verification of the data and when they are correct, the money is sent to another account.



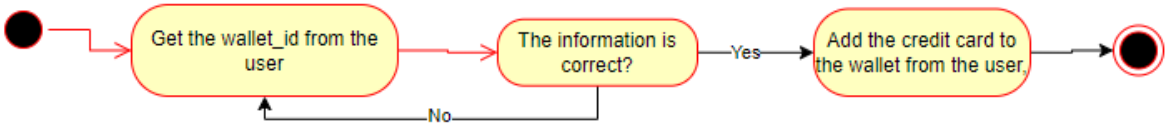
Request funds: In this diagram the system receives the ID of the account that sends the money, the amount to be requested from another user and the ID of the receiving user, then the system requests verification of the data and when they are correct the money is sent to another account.



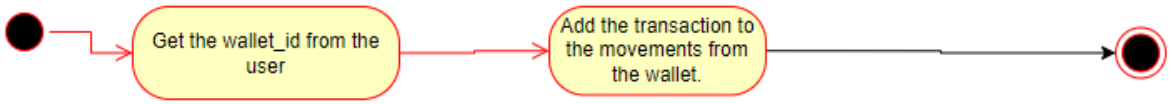
Show movements: In this diagram the wallet ID is received, and with it the movements made with the wallet are shown.



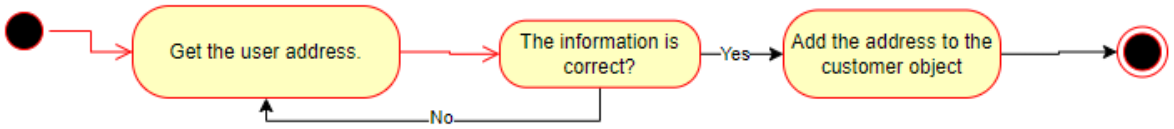
Add credit card: In this diagram the wallet id is received, and if the information is correct, the credit card is added to the user's wallet.



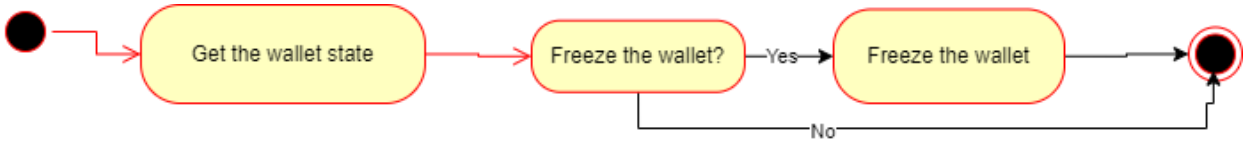
Add transactions to movements: In this diagram the wallet id is received, and if the information is correct, the transactions are sent to the wallet movements.



Add user address: In this diagram the wallet id is received, and if the information is correct, the transactions are sent to the wallet movements.

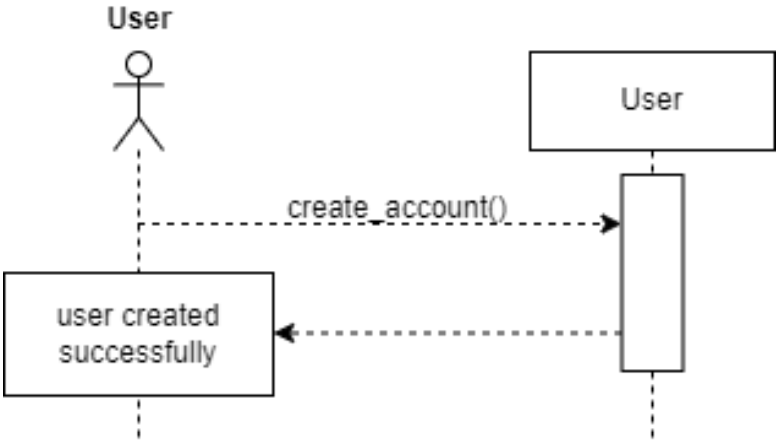


Freeze account: In this diagram the admin receives the wallet state and if it's necessary, the admin freezes the account

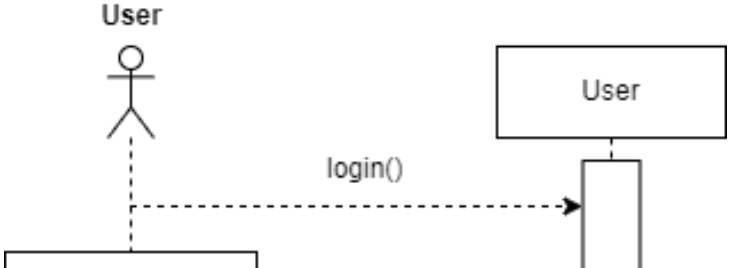


Sequence Diagrams

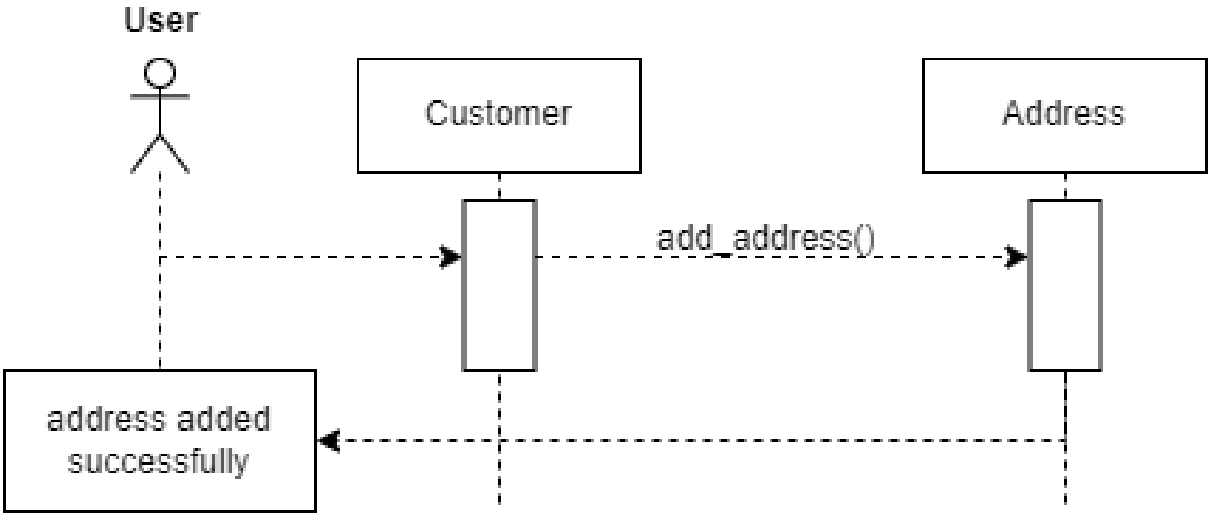
Create Account: This diagram shows how the customer tries to create an account, and the system returns a registration message if he has entered the expected data.



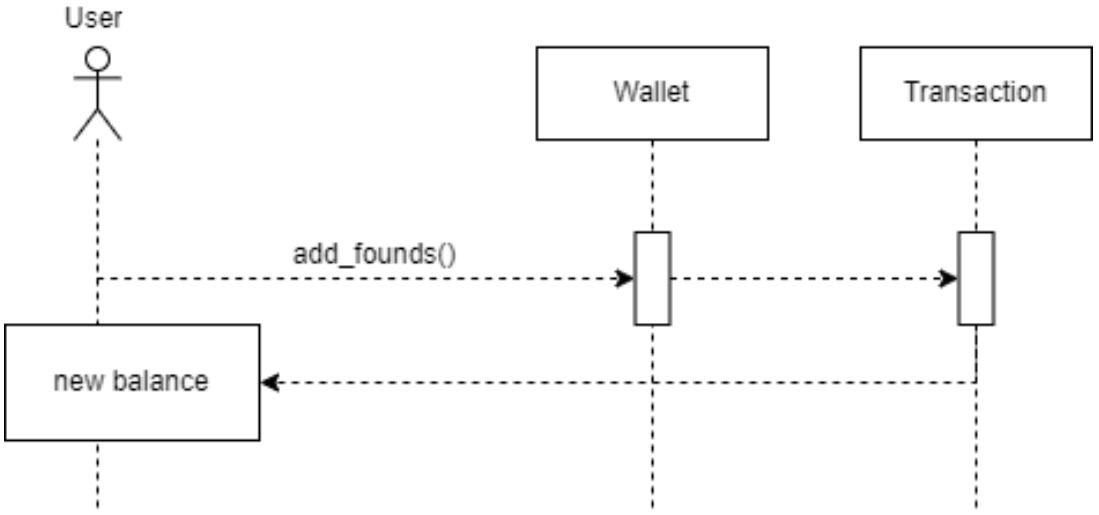
Login: This diagram shows how the user tries to log in to the system, and the system returns a message of validity or invalidity depending on the data entered.



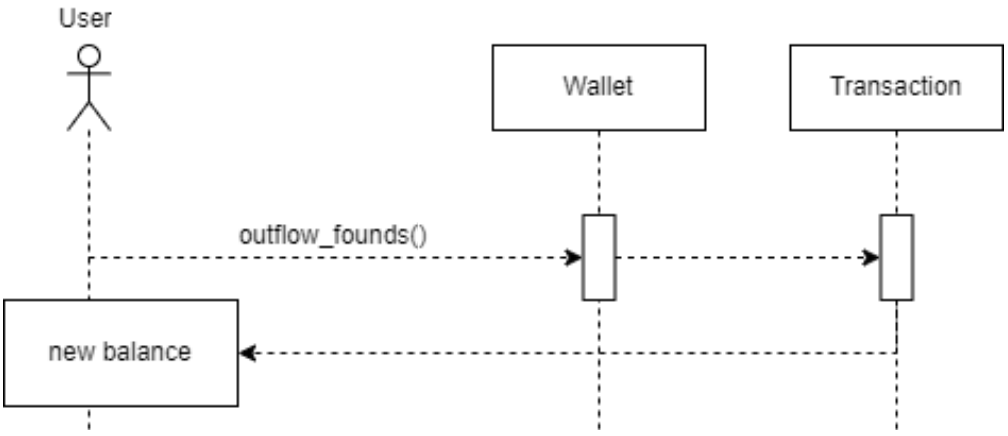
Add Address: This sequence diagram shows how the user add an address and the system response with a confirmation message.



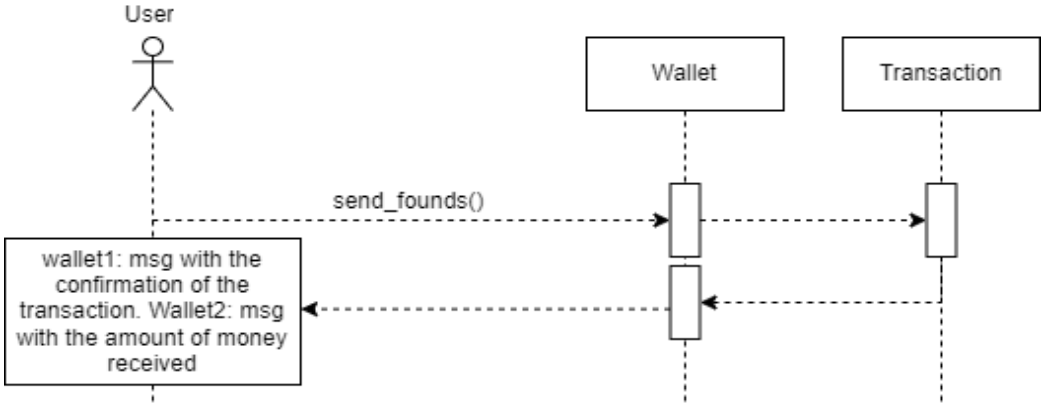
Add Founds: This sequence diagram shows the process of adding funds to the wallet, and the system response is the new wallet balance.



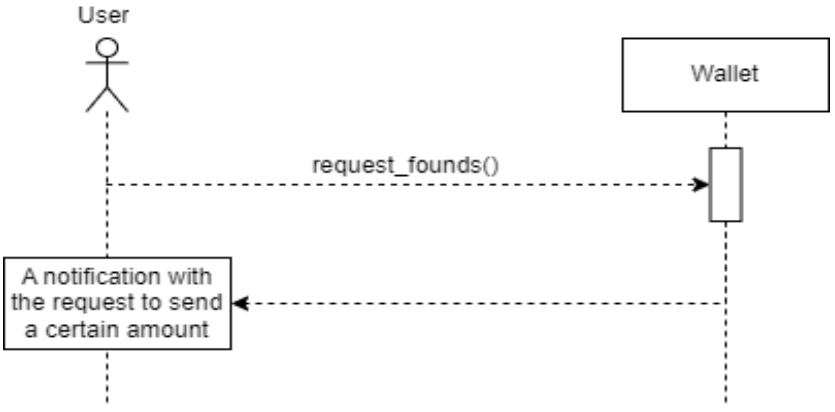
Outflow Founds: This sequence diagram shows the process of outflow Funds from the wallet and the response of the system that in this case is show the new balance



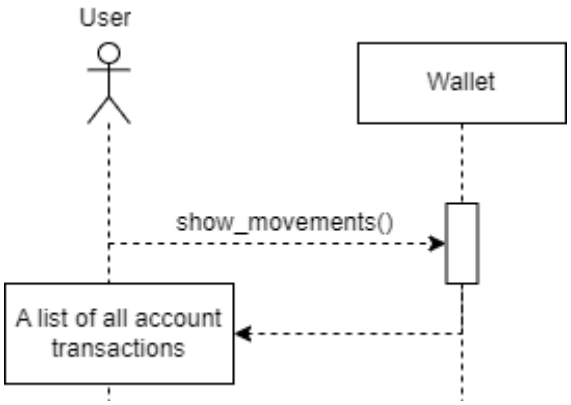
Send Founds: This sequence diagram shows the process of sending money between wallets and the system response consisting of a message with the status of the transaction for the person sending the money and a message with the amount of money received for the person to whom the transaction was directed.



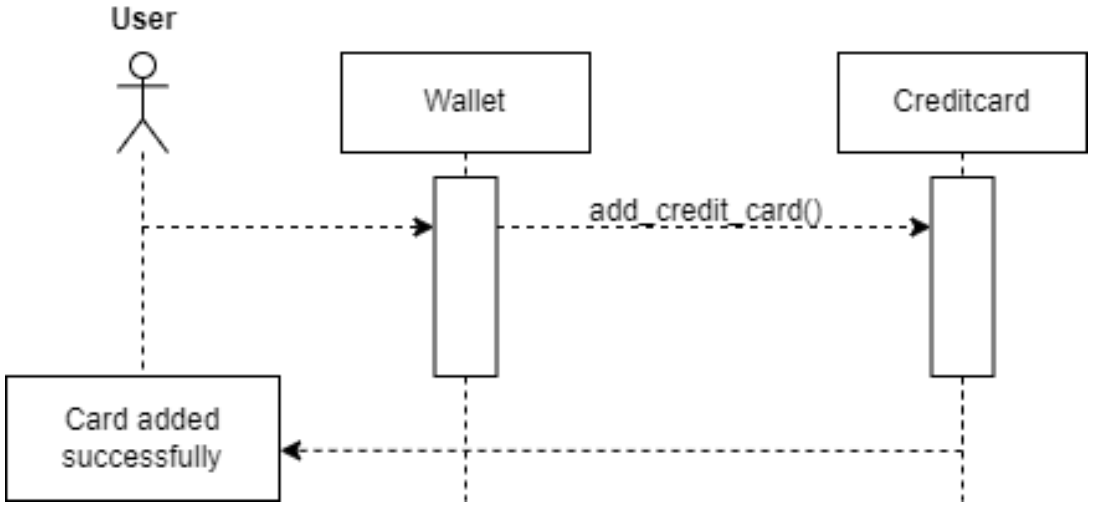
Request Founds: This sequence diagram shows the process of request money between user of the application.



Show movements: This sequence diagram shows the process to watch the history of movements from the account.

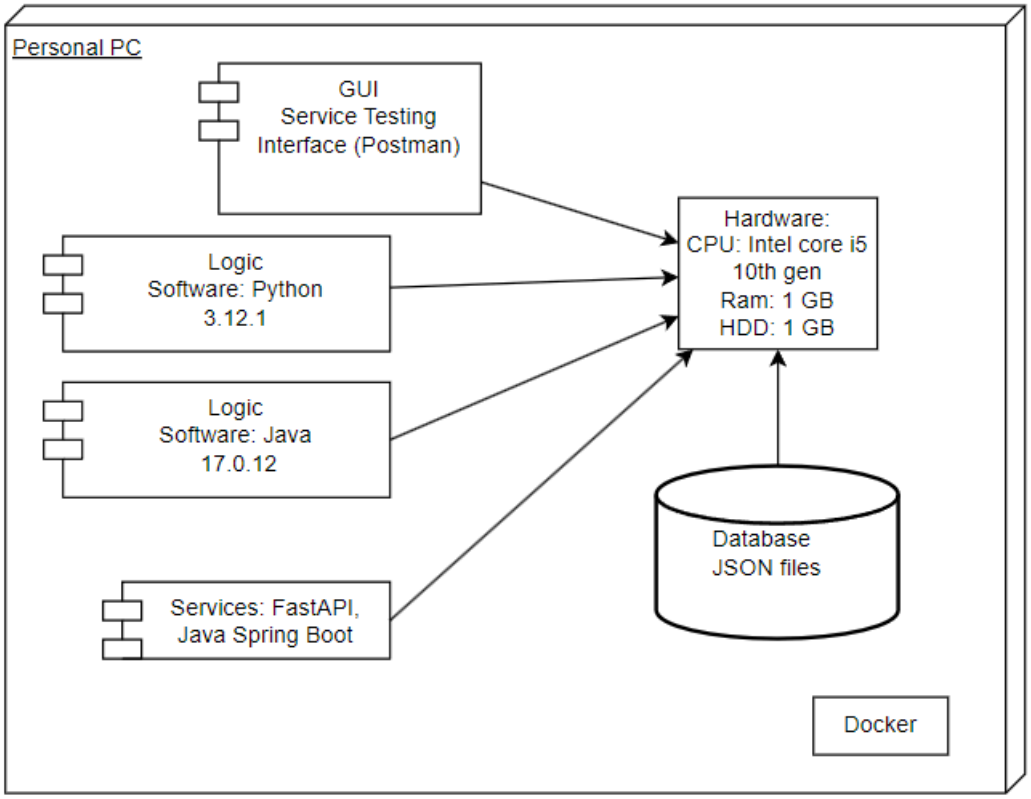


Add Credit Card: This sequence diagram shows the process of adding a new credit card linked to the wallet.

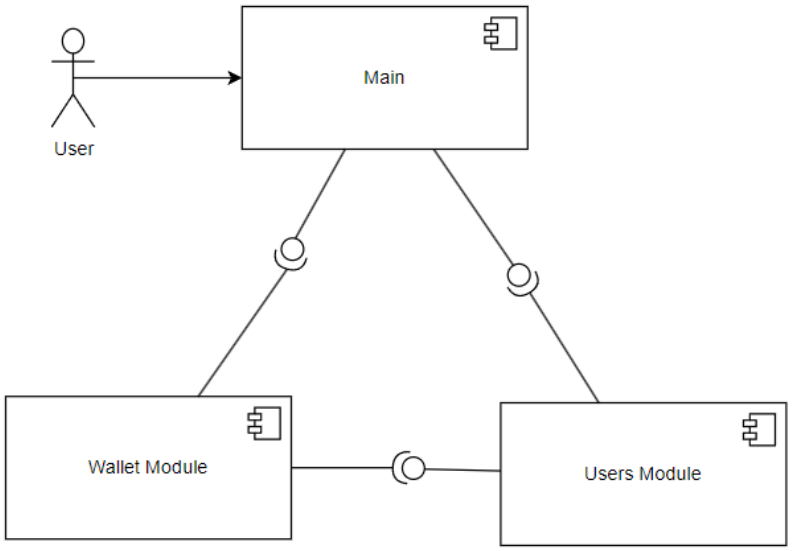


6. **Architecture**

Deployment Diagrams: The application has two backends (Java 17.0.12 and Python 3.12.1). FastAPi and Java Spring Boot were also used to expose the services.
On the other hand, the data storage method used was json files.
The graphical interface used is the postman service test interface.
On the other hand, the tool selected for automated deployment was docker.

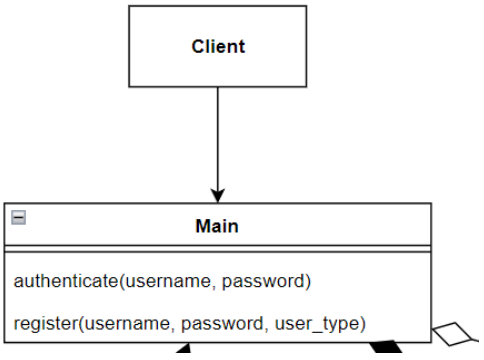


Components diagram: This component diagram is composed of 3 main components, the Main entry point and the Wallet and Users modules, connected through web services.

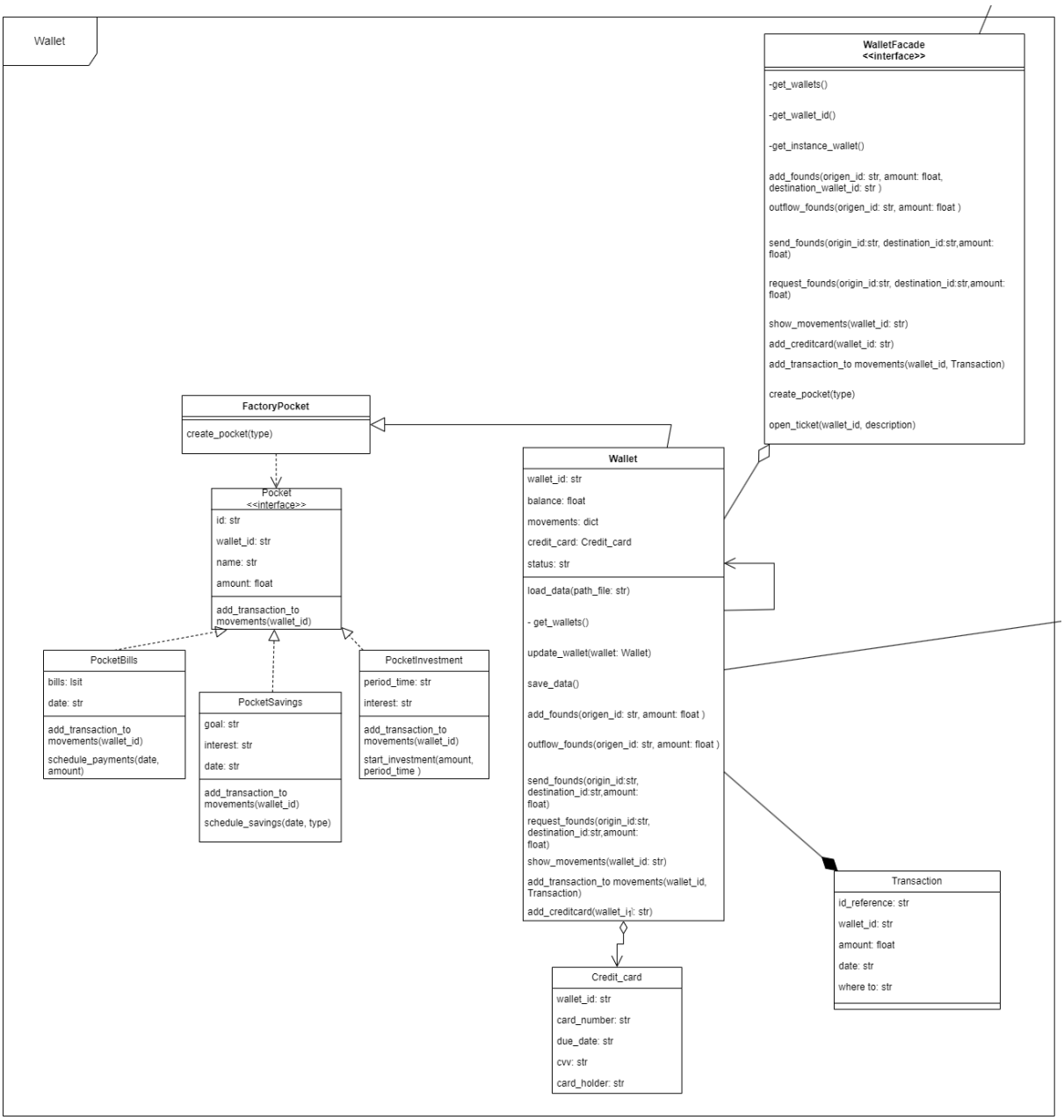


Class diagram:

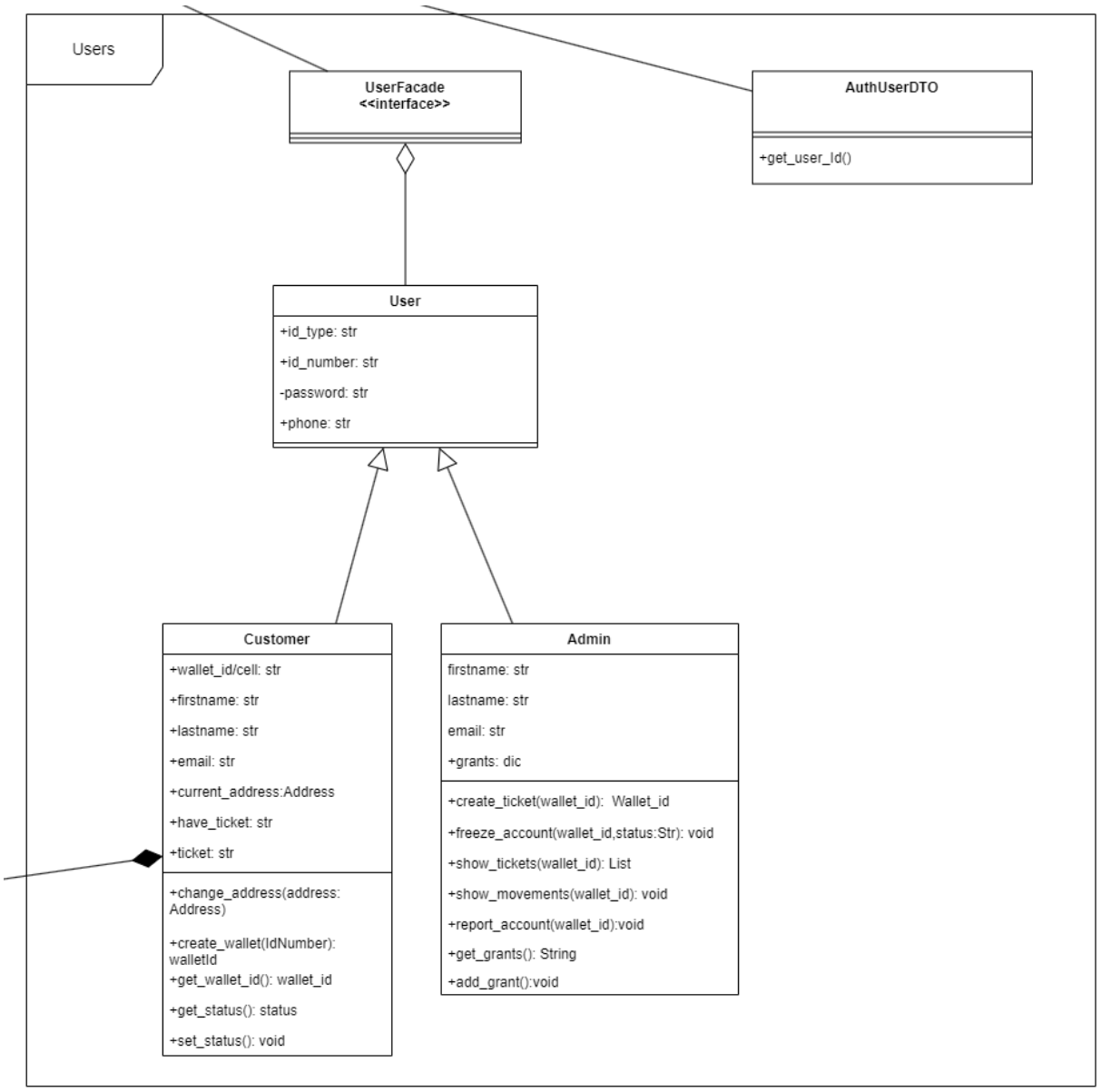
The class diagram starts with a Main class to which the user connects to access the lower-level functionalities. This class has authentication and registration methods.



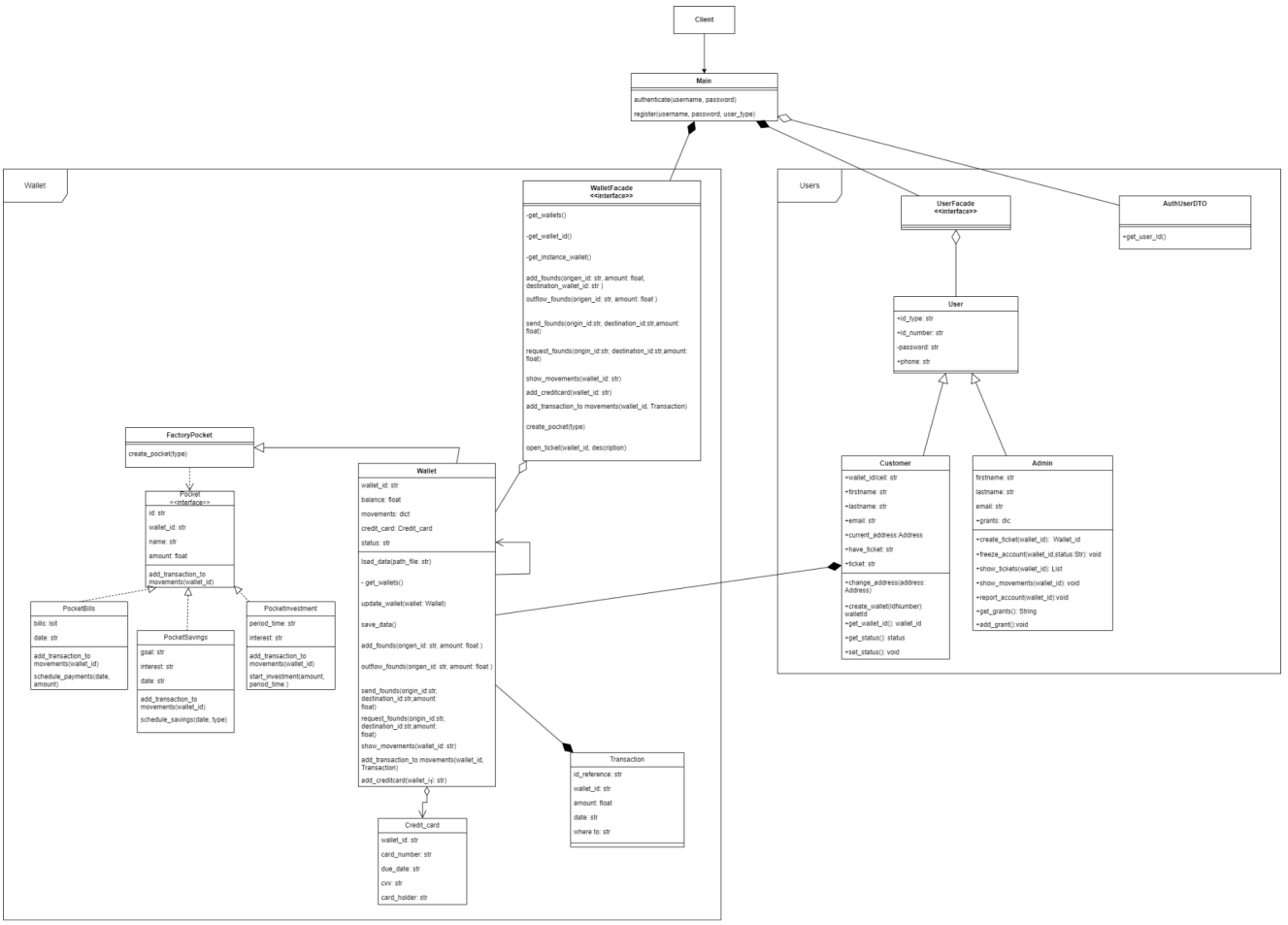
There is also a wallet module that connects to the Main through Wallet Facade. This module also has the classes Wallet, FactoryPocket, Pocket Bills, PocketSavings, PocketInvestment, Credit Card and Transaction.



Finally, we have the Users module, which connects to the Main class through UserFacade. This module also includes the Customer, Admin and AuthUserDTO classes.



The complete class diagram is shown below:



7. Analysis of SOLID principles in architecture

In our project, we have applied the SOLID principles through the use of design patterns and good practices in class structuring. Below we explain how each principle was implemented:

Single Responsibility Principle (S): Each class was designed with a single specific responsibility, avoiding the accumulation of multiple functionalities in a single entity. In addition, interfaces were used to decouple functionalities, reinforcing this principle.

Open/Closed Principle (O): This principle was implemented through the use of abstractions and interfaces, allowing the expansion of functionalities without modifying the source code of existing classes. Different types of pockets (Pocket) were added to the project. The pockets are created using a factory pattern which allows us to incorporate new objects without modifying the code of the creation logic.

Liskov Substitution Principle (L): This principle is respected by ensuring that concrete classes can replace their abstractions without affecting the behavior of the system, which is evidenced in the hierarchy of the User class, where Customer and Admin extend User and can be used interchangeably in any part of the code where a user is required. It is also observed in the different types of Pockets which can replace their base class without altering the expected functionality.

Interface Segregation Principle (I): To avoid unnecessarily large interfaces or those with unused methods, specialized interfaces were implemented according to the system's functionality:

- WalletFacade groups the wallet-specific functionalities.
- UserFacade encapsulates the operations related to users.

This guarantees that each class only implements the interfaces it really needs, avoiding unnecessary dependencies.

Dependency Inversion Principle (D): This principle was applied using the Facade pattern, ensuring that high-level classes depend on abstractions and not on concrete implementations. WalletFacade and UserFacade encapsulate the logic of the subsystems, preventing clients from directly interacting with internal classes and ensuring flexibility in future modifications.

8. Analysis and detail of the implementation of design patterns

Pattern	**Implemented**	**why?**
Builder	No	We did not implement the builder pattern because we did not identify the need to create an object with multiple attributes that required specific configuration.
Factory	Yes	We implement the Factory pattern because it allows us to dynamically create pockets based on their type. In our project, we offer three types of pockets, each with unique attributes and functionalities, as well as shared features. Thanks to the Factory pattern, we can instantiate these pockets without coupling the code to concrete classes, making it

		easier to expand the system. In addition, if new types of pockets are added in the future, implementation will be simpler, since it will only be necessary to extend the factory without modifying the existing logic.
Abstract Factory	NO	We do not implement Abstract Factory because our system does not require the creation of families of related objects or the recursive generation of predefined instances. Thanks to persistence in JSON files, we can retrieve and manage all the data of our objects without the need to constantly instantiate them, making the additional complexity of this pattern unnecessary.
Singleton	Yes	We implemented the Singleton pattern because in our application each client must access a single digital wallet and not multiple wallets instances simultaneously. This pattern allows us to ensure that only one instance of the wallet exists per client, avoiding unnecessary duplication of objects and ensuring data consistency.
Prototype	No	We did not implement the Prototype pattern because our project does not require cloning objects. We consider that duplicating wallets, pockets or any other application object could compromise data integrity and generate inconsistencies, and JSON files persistence allows us to manage information without the need for additional copies.
Bridge	No	We did not implement the Bridge pattern because we did not identify large classes that required separation. Instead, we used concrete classes with well-defined functionality and specific interfaces to organize the subsystems without adding unnecessary complexity.
Composite	NO	We do not implement the Composite pattern because our project does not require tree-like hierarchical structures or

		the composition of objects in a partial inheritance relationship.
Proxy	NO	
Flyweight	No	We did not implement the Flyweight pattern because our project does not handle a large number of object instances at the same time. Since we do not face high memory consumption issues due to mass object creation, reuse of immutable parts is not necessary.
Decorator	NO	We do not implement the Decorator pattern because our project currently does not require the dynamic addition of functionalities, since currently the system functionalities are well defined and do not need frequent modifications.
Adapter	No	We did not implement the Adapter pattern because our project was designed from scratch, ensuring compatibility between its interfaces and since we do not integrate third-party libraries or face incompatibility issues, this pattern was not necessary.
Facade	Yes	We implemented the Facade pattern to provide a single access interface to the functionalities of the wallets and pockets, simplifying the interaction with the system. In addition, we designed an interface to manage the users' functionalities, ensuring structured and efficient access. That was done through microservices.
Iterator	NO	We did not implement the Iterator pattern because our project does not handle complex data structures that require a specialized mechanism to iterate through their elements.
Memento	No	We did not implement the Memento pattern because our project does not require restoring objects to previous states. Allowing this could compromise data integrity and lead to inconsistencies.

Strategy	No	We did not implement the Strategy pattern because our project does not require dynamically switching between multiple algorithms. All functionalities follow a specific logic.
Template	No	We did not implement the Template pattern because our project does not require defining a base algorithm with customizable steps. All operations have well-defined steps that do not change dynamically between classes.
Chain of Responsibility	No	We do not implement the Chain of Responsibility pattern because our project does not require a request to go through multiple handlers before being processed, each request has a single handler.
State	No	We do not implement the State pattern because our project does not require objects to dynamically change their behavior based on their internal state.
Mediator	No	We do not implement the Mediator pattern because our project does not require managing complex interactions between multiple objects and communication between objects is done through interfaces, maintaining a modular and decoupled design.
Command	NO	We do not implement the Command pattern because our project does not require encapsulating requests in objects for parameterization or deferred execution. All actions are handled directly, they cannot be undoable and therefore there is no need to represent requests as objects.
Observer	No	We did not implement the Observer pattern because our project does not require establishing a subscription relationship where one object automatically notifies others when its state changes, nor did we identify scenarios where multiple objects need to

		be updated in real time in response to changes in other objects.

9. Analysis of the implementation of good programming practices and anti-patterns

To avoid code smells in our project, we implemented tools such as pylint and Black Formatter, which help us maintain clean and well-structured code. These tools allowed us to identify and correct problems such as Long Parameter List, Excessive Comments, among other code smells.

In addition, as you can read in the previous point, we performed a detailed analysis of the design patterns we implemented, making sure to implement only those that really added value to the project. This allowed us to avoid anti-patterns that could add unnecessary complexity. Regarding the patterns applied, we prioritized low coupling and the use of interfaces, which allowed us to develop decoupled, scalable, flexible, and easy-to-maintain code.

10. Conclusions

In conclusion, the implementation of SOLID principles and design patterns in the project has allowed for a flexible, maintainable and scalable architecture. A high degree of decoupling was achieved by employing interfaces and patterns such as Factory and Facade, which facilitates the extension of the system without modifying its base structure. The correct application of interface segregation and dependency inversion contributed to a modular design that is adaptable to future changes. Furthermore, the use of the Liskov substitution principle ensures that derived classes can replace their abstractions without affecting the behavior of the system. Overall, the approach adopted in the development of the project has resulted in a well-structured design, which favors code reuse and separation of responsibilities, facilitating future improvements and maintenance of the system.