Advanced Programming Final Proyect Technical Report

Juan Nicolás Diaz Salamanca
20232020059
Mathew Zahav Rodriguez Clavijo
2023202050
Francisco Jose de Caldas University

CONTENTS

LIST OF FIGURES

List of Tables

Advanced Programming Final Proyect Technical Report

I. Introduction

In order to present and develop an application used and adapted to the real needs of a target costumers, we proposed the development of a cloud storage application, popularly known as Drive applications; the objective of this project is to develop an application of this type following the software qualities of scalability, Reliability, Manteability. Drive-type applications are applications in charge of providing cloud storage services, that is, providing the user with the ability to store files, organize them in folders and access them from any computer with an internet connection. The following diagram is the business model of a cloud service company:

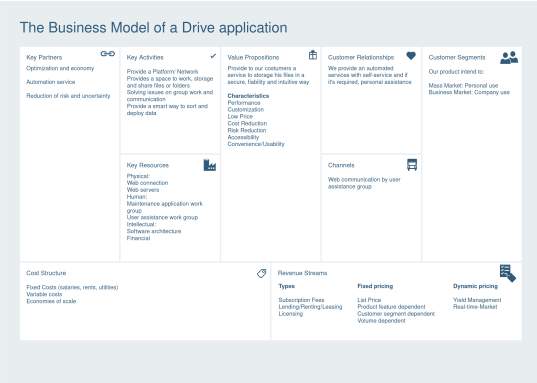

Fig. 1. Business Model of a Drive application

II. User stories

After a survey realized to different kinds of costumers, we recollect their needs in their own words and summary in the next user stories:

- I as a user I expect an intuitive software to upload and sort files.
- I as a user, I except to keep my files and folders secure of thirds persons.
- I as user, I except that my storage always be enough for my folders and files.
- I as user, I except a graphic and intuitive interface.
- I as user, I expect to open my files with third party applications.
- I as user, I expect to download files from my storage to my personal
- computer.
- I as user, I expect to see the modification date of my files and open it
- without required to download.
- I as user, I expect to sort my files by name and modification date.
- I as user, I expect an easy access and search through my storage.

III. Requirements

A. Functional

These are the functionalities which users interact with while the application is executing, the clients will expect to recognize their needs on these requirements. Summarizing and turning into technical specifics the user needs, we got the next functional requirements:

- Browser: The applications must include a search functionality to get files and folders that match with user input.
- Elements visualization: The application must show folders and its files with modification date, name, extension and size.
- Pre-visualization of files: The application must allow to open files without required the download in the user pc.
- Organize elements: The application must allow to sort folders and files by his name and modification date.
- Friendly GUI: The application must supply and friendly user interface for the user activities.

B. No functional

These are functionalities which users do not interact in the normal execution of the application, but it is required for the well-being of the application. Based on the user stories, the next no functional requirements are necessary to summit the costumer's expectation:

- Compatibility: The application must work with any application which can open and edit files inner it.
- Security: The application must not give unauthorized access to files and folders from a user
- Storage Consistency: The application must keep the integrity of the user files.

IV. UML Diagrams

Based on the functional requirements we propose the next activities diagrams to show which's are the normal applications flows in the user interacts: :

V. Activity Diagrams

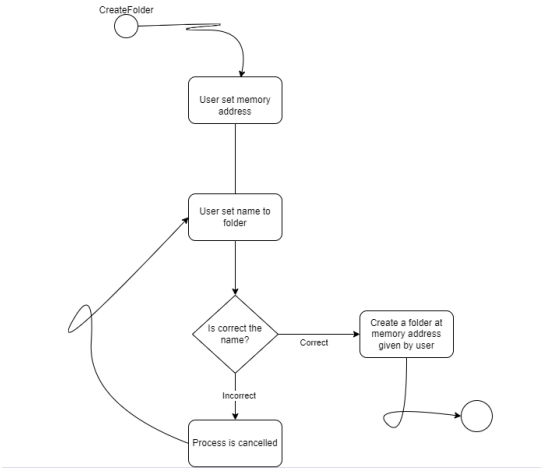

Fig. 2. Activity of create a folder

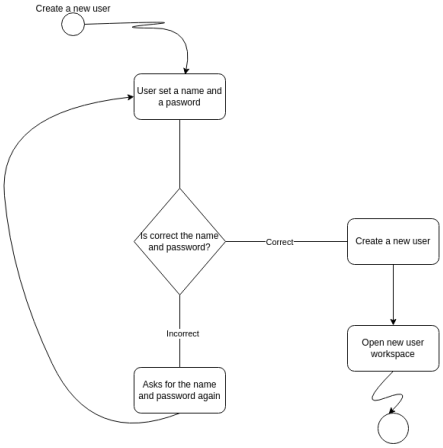

Fig. 3. Activity of create a user

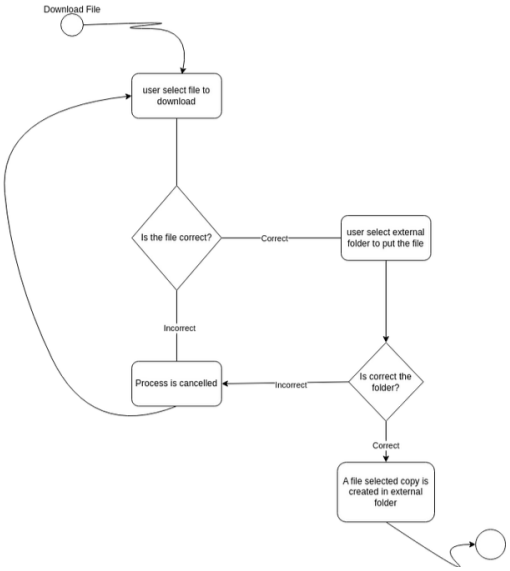

Fig. 4. Activity of download file

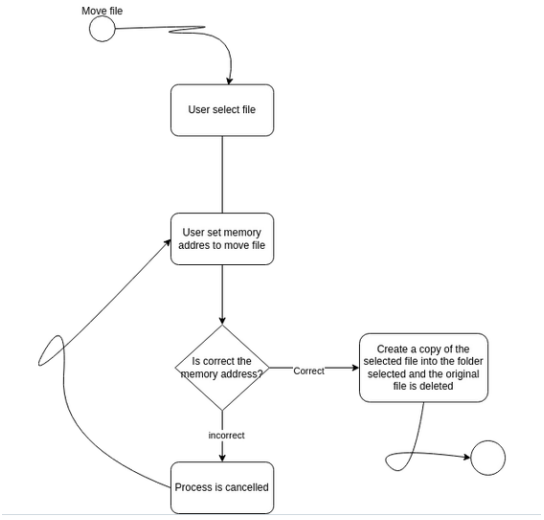

Fig. 5. Activity of move file

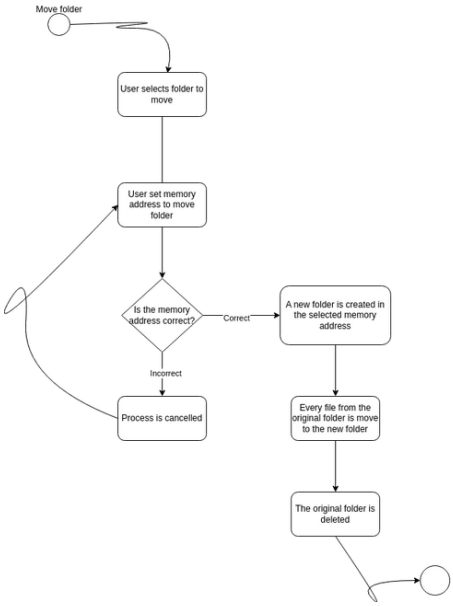

Fig. 6. Activity of move folder

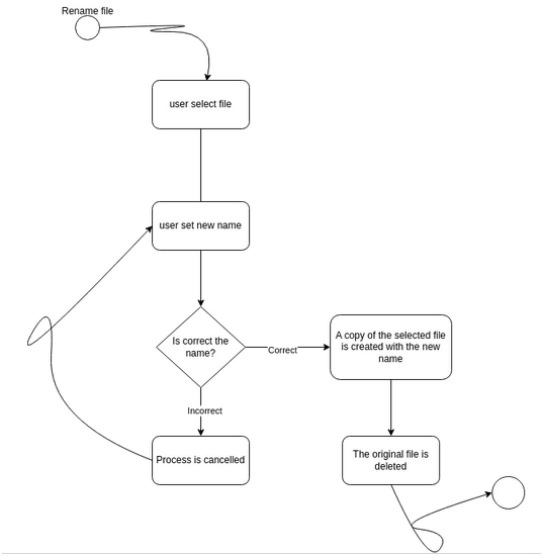

Fig. 7. Activity rename file

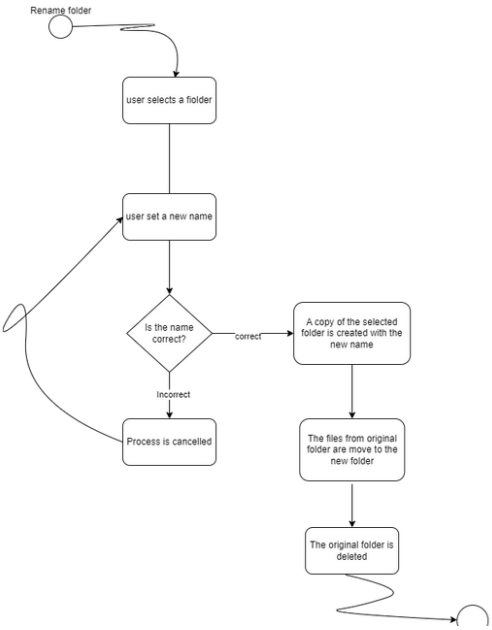

Fig. 8. Activity rename folder

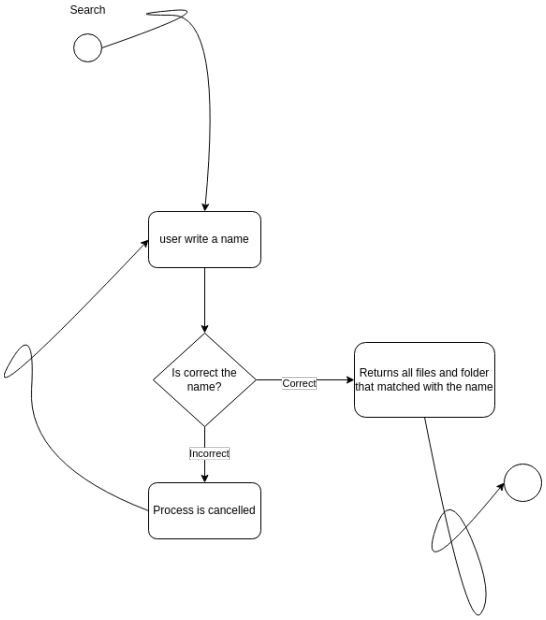

Fig. 9. Activity search

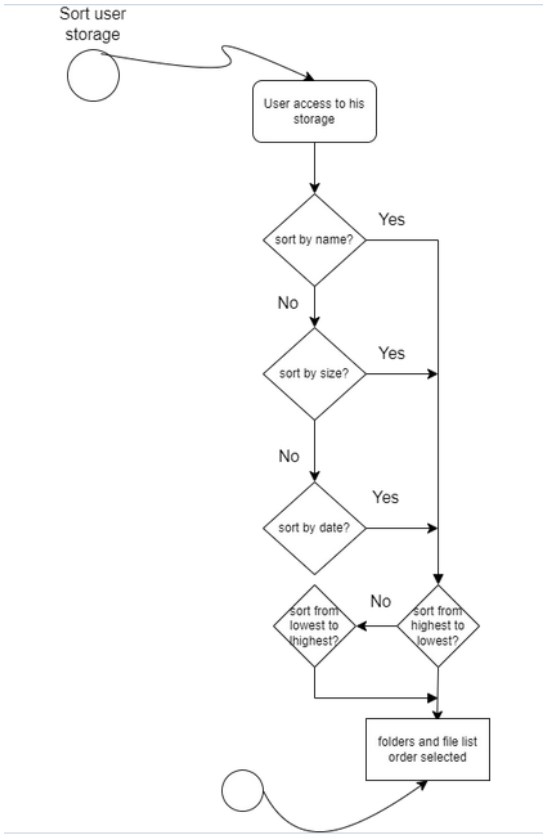

Fig. 10. Activity sort elements

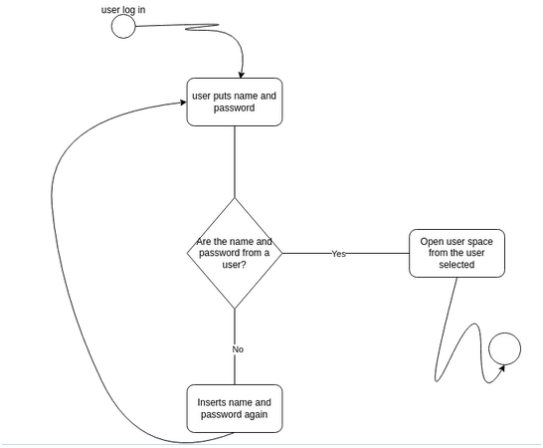

Fig. 11. Activity log in

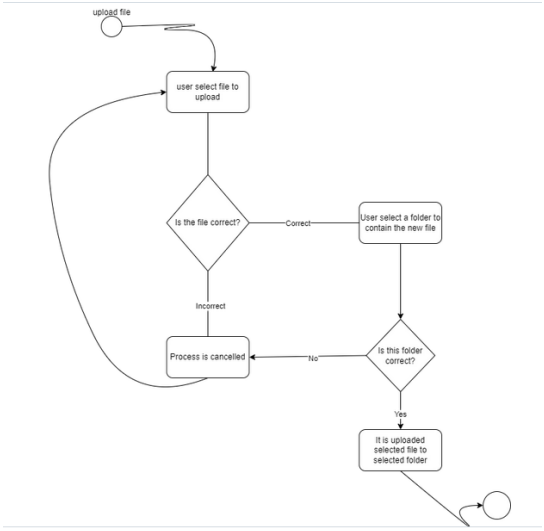

Fig. 12. Activity upload file

VI. Business Diagram

Based on the business model and user needs, we propose the next business diagram:

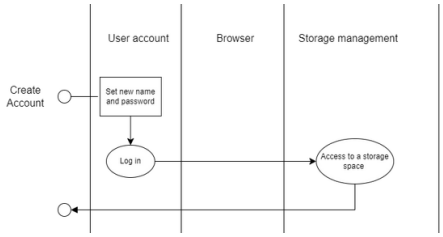

Fig. 13.

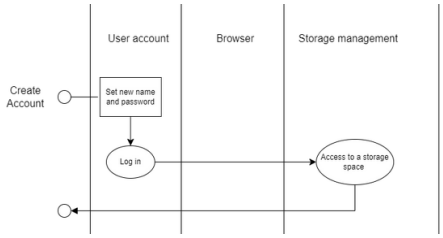

Fig. 14.

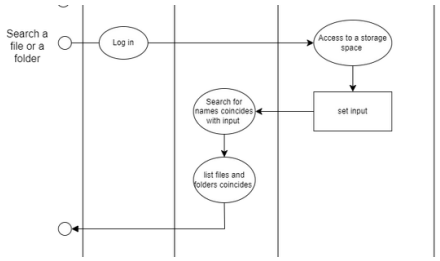

Fig. 15.

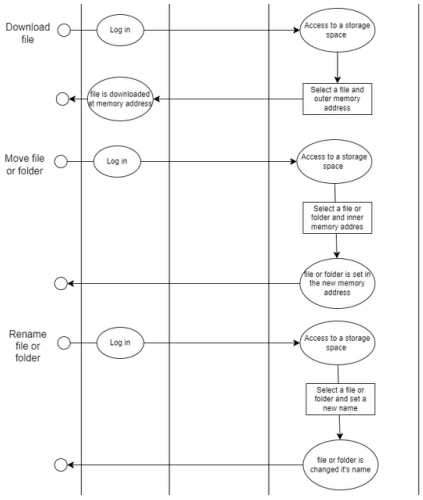

Fig. 16.

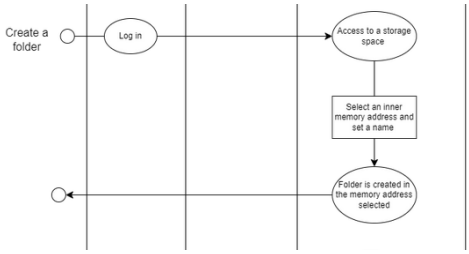

Fig. 17.

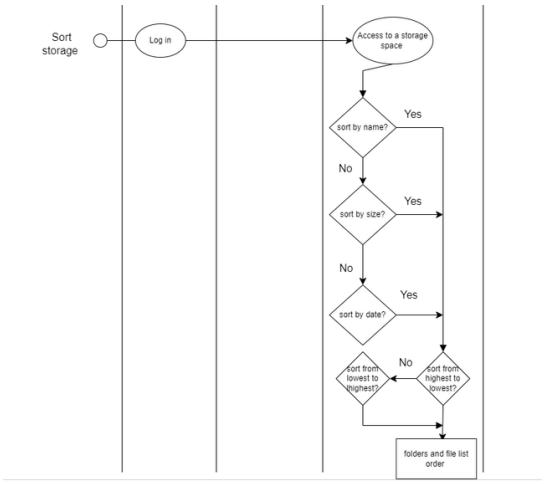
Sort storage
Log in
Access to a storage space
sort by name?
Yes
No
sort by size?
Yes
No
sort by date?
Yes
No
sort from lowest to highest?
sort from highest to lowest?
folders and file list order

Fig. 18.

VII. Deploy Diagram

To deploy successfully the application, we use the next deploy diagram:

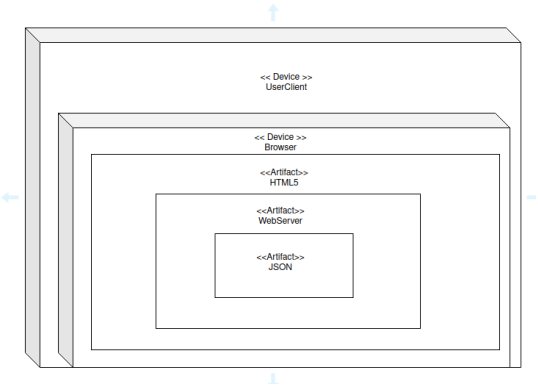

Fig. 19. Deploy Diagram

VIII. Concept Design

Based on the functional and non-functional requirements, the following entities are proposed and summarized by means of CRC cards:

A. CRC Cards

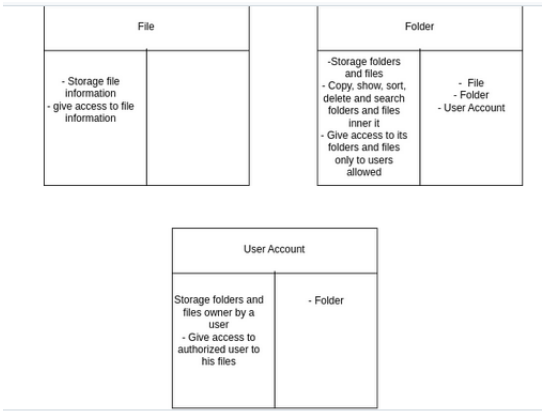

Fig. 20. CRC Cards

B. Mockup

And following the requirement of an intuitive GUI for the user, the following mockup was created:

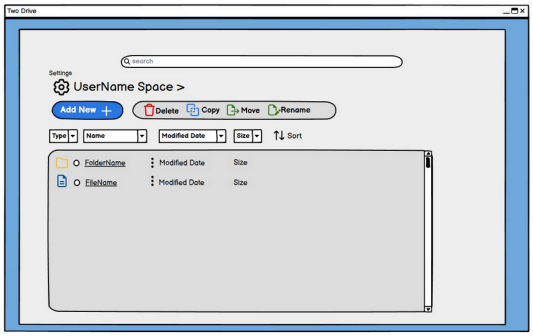

Fig. 21. Mockup for the user main page

IX. Architecture

The proyect bases on a microservices architecture, dividing by frontend, the graphic part which user interacts with and backend, the server side which makes all the logic works. the project is componend by two backends, one for the management of files and folders, and other one for authentifcation users.

A. Class diagram to management files and folders

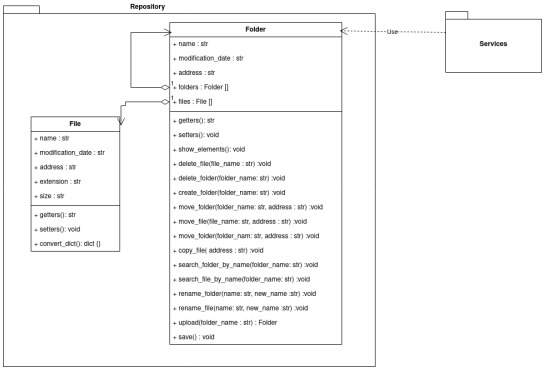

Fig. 22. Class Diagram for the backend to manage folders and files

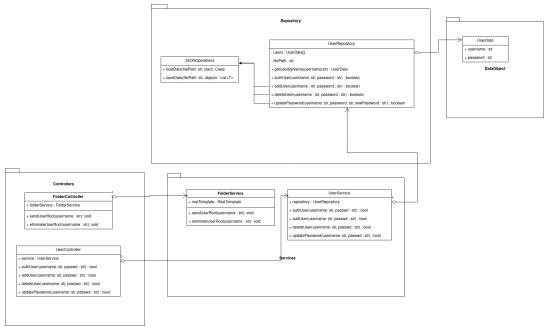

Fig. 23. Class Diagram for the backend to manage folders and files

B. Class diagram to management user authorization

X. Web services definition

Name	Endpoint	HTTP Method	Input	Output	Documentation
Update Password	/update_password	POST	username : str password : str	result : bool	This method update the password of the user given
Delete User	/add_user	POST	username : str password : str	result : bool	This method delete the user given
Upload data	/{folder_name}	GET	Folder_name : str	Folder : ditc {}	This method upload a folder to the database Args: folder_nam: str Returns:
Search File	/search_file/{file_name}	GET	File_name : str	File : dict {}	This method search files which match with the name given Args: file_name: str
Search Folder	/search_folder/{folder_name}	GET	Folder_name : str	Folder : dict {}	This method search a folder which match with the name given Args: folder_name: str Returns:
Show Elements	/show_elements	GET	None	Folder : ditc {}	This method show the elements in the folder root Returns: str: message
Show folder element	/show_folder_elemen{folder_name}	GET	Folder_name : str	Folder : dict {}	This method show the elements in a folder Args: folder_name: str
Create folder	/create_folder/{folder_name}	POST	Folder_name : str	Folder : dict {}	This method create a folder Args: folder_name: str Returns: new_folder: Folder object
Delete Folder	/delete_folder/{folder_name}	DELETE	Folder_name : str	Folder: dict{}	This method delete a folder Args: folder_name: str Returns: Folder_root : Folder object
Rename Folder	/rename_folder/{new_folder_name}	POST	Folder_name: str New_folder_name : str	Folder : dict {}	This method rename a folder Args: folder_name: str new_folder_name: str Returns:
Rename File	/rename_file/{new_file_name}	POST	File_name: str New_file_name : str	File : dict {}	This method rename a file Args: file_name: str new_file_name: str Returns:
Copy File	copy_file/{file}	POST	File : str Base_dir : str	File : dict {}	This method copy the file at the adress given As the argument is the file name and it's extension, the method
Delete File	/delete_file/{file}	DELETE	file : str	Folder : dict {}	This method delete a file inner the folder root Args: file_name: str Returns:
Move File	/move_file/{file}	POST	file_name :str Folder_name : str	message : str	This method move a file to a folder Args: file_name: str Returns: str: message
Move Folder	/move_folder/{folder_name}	GET	folder_move :str folder_reach : str	message : str	This method move a folder to a folder Args: folder_name: str
Save	/save/	POST		message: str	This method save the folder root modifications Args: None Returns:
AuthUser	/auth	POST	username : str password : str	result : bool	This method authentificate user
AddUser	/add_user	POST	username : str password : str	result : bool	This method add user
Get User by Name	/get_user/{name}	GET	username : str	result : bool	This method return a user if equals with the name given

Fig. 24. Class Diagram for the backend to manage folders and files

XI. SOLID Principles implementation

- Single responsibility: This principle was applied from the development of the conceptual design to the definition of the functionalities of each class. From this principle, for example, the responsibilities of the folder class and the file class were separated, since although both are objects that store data such as name, date modified, size. However, because the Folder class also stores folders or files, they cannot be generated from the same class.
- Open to extension, Out to modification: This principle is applied by making all classes inherit from an abstract interface or class.
- Liskov substitution: This principle was applied by making classes communicate through their interfaces or abstract classes, in this way, the child class can always replace the mother class.
- Interface Segregation: Applies by making every entity inherit from its own abstract class or interface
- Dependency inversion: Applied by making every class dependent on abstract classes or interfaces

XII. Conclusion

It is then concluded that the process carried out to develop the software, starting from the needs of the user, said in their own words, transforming their needs into functional and non-functional requirements, for the planning of the software by means of CRC cards and UML diagrams and through these generate the code; It proved to be an optimal process for application development as it avoids repeating planning and coding steps and facilitates the programming process.