WORKSHOP No. 4 – LAYERS ARCHITECTURE

Juan Sebastián Herrera Rodríguez

20242020032

Isabela Chica Becerra

20242020035

School of Engineering, Francisco Jose de Caldas District University

Object-Oriented Programing

Carlos Andres Sierra

Bogotá D.C

2025

PROJECT DEFINITION

PinBerry is a simplified and personalized web application inspired by Pinterest, designed with a strong focus on minimalism, clarity, and ease of use. Unlike PinBerry is a minimalist and user-focused web application inspired by Pinterest, designed to offer a simple and intuitive space for managing personal photos. Unlike traditional platforms overloaded with social and tagging features, PinBerry allows users to perform only the essential actions: upload, view, delete, and download images—without distractions or unnecessary complexity.

The project was driven by two main goals: to provide a clean, distraction-free photo management tool, and to implement core object-oriented programming (OOP) principles. The system is built around modular classes such as User, Photo, and Folder, each with clear responsibilities, making the codebase both maintainable and extensible. Though still under development, PinBerry serves as both a practical tool and a learning project, demonstrating how thoughtful design and OOP can produce clean, scalable, and user-centered applications.

PROJECT OBJECTIVES

- ***General objectives***
  - To develop a minimalist and user-centered web application that allows users to upload, view, delete, download, and organize images into folders within a clean, accessible, and distraction-free environment.

- ***Specific objectives***
  - Design and implement a responsive and intuitive user interface to ensure ease of use across desktop devices, by applying minimalistic design principles and optimizing layout responsiveness.
  - Develop a modular object-oriented architecture to support clean separation of concerns and future scalability, by defining well-structured classes such as User, Photo, and Folder, each with clear responsibilities.
  - Implement essential photo management functionalities to enable user registration, authentication, image upload, deletion, downloading, and folder creation, using structured and reusable methods.
  - Ensure long-term maintainability and usability of the system by adhering to clean code practices, incorporating internal documentation, and fostering a distraction-free experience focused on private image organization.

1. REQUIREMENTS DOCUMENTATION

Functional requirements:

- **User Account Management**

  – The system shall provide a registration interface that allows new users to create an account by submitting a unique username and a secure password.

  – The system shall authenticate users by validating the submitted credentials (username and password) against stored data during the login process.

  – The system shall restrict access to photo-related functionalities only to authenticated users.

- **Photo Upload and Download**

  – The system shall allow authenticated users to upload one or more photos from their local device through a file selection interface.

  – The system shall store uploaded photos in the user's dedicated General Folder and update the user interface accordingly.

  – The system shall allow users to download any photo from their collection, providing the image in its original resolution or a predefined system resolution.

- **Photo Deletion**

– The system shall allow users to permanently delete any photo from their General Folder, removing it from both the user interface and the storage/database.

- **Photo Organization**
  - The system shall automatically create a User Folder for each user upon registration, to serve as the primary container for all user's uploaded images.
  - The system shall allow users to create additional folders with custom names, which will reference selected images from the General Folder.
  - The system shall allow users to assign existing images from the General Folder into one or more custom folders

- **Photo Browsing**
  - The system shall provide an interface for users to browse their photo collection in a grid or gallery layout.
  - The system shall implement either infinite scrolling or a paginated mechanism to display photos, depending on the dataset size and performance considerations.

Non-functional requirements:

- **Usability and Design**
  - The system shall provide a graphical user interface (GUI) that adheres to minimalistic and visually appealing design principles, ensuring ease of navigation and interaction on both desktop and mobile devices.
  - The user interface hall complies with established usability heuristics (e.g., consistency, user feedback, and simplicity), enabling new users to complete core operations—photo upload, deletion, and download—within a maximum of 3 minutes without external guidance.

- **Reusability**
  - The system shall adopt a component-based architecture, where core functionalities (e.g., file uploading, input validation, user authentication) are implemented as independent and reusable modules.
  - All shared components shall be designed with **parameterization and abstraction**, allowing them to be reused in future versions or extended features without modification to their internal logic.

- **Flexibility and Maintainability**
  - The software shall be implemented using object-oriented programming (OOP) principles, including encapsulation, polymorphism, and abstraction, to support the future addition of new features (e.g., photo tagging, search filtering) with minimal disruption to the existing codebase.
  - The system shall be designed in accordance with the SOLID principles of object-oriented programming to ensure modularity,

scalability, and robustness. Specifically, each class shall adhere to the Single Responsibility Principle by encapsulating a single, well-defined functionality.

– The Open/Closed Principle shall be followed to allow classes to be extended without modifying existing code.

– The Liskov Substitution Principle shall ensure that derived classes maintain compatibility with their base types.

– The Interface Segregation Principle shall guide the creation of specific and minimal interfaces to avoid forcing classes to implement unused methods

– The Dependency Inversion Principle shall be applied to decouple high-level components from low-level implementations through the use of abstractions, promoting a flexible and maintainable system architecture.

– The source code shall conform to recognized coding standards (e.g., Java Code Conventions), including proper indentation, naming, commenting, and file organization to enhance readability and reduce technical debt.

– The system shall include internal documentation and inline comments covering at least 80% of the classes and methods, facilitating long-term maintenance and knowledge transfer.

2. USER STORIES

Title: View Photos	**Priority:** High	**Estimate:** 16-20 hours
User Story: As a User, I want to view the available images on the platform, So that in can explore interesting visual content.		
Acceptance criteria: **Given** that I am on the homepage **When** I navigate through the gallery, **Then** I should be able to see all the available images.		

Title: Upload Photos	**Priority**: High	**Estimate**: 20-24 hours
User Story: As a User, I want to upload images to the platform, So that I can share my memories and experiences.		
Acceptance criteria: **Given** that I am on the upload page, **When** I select an image and confirm the upload, **Then** the image should be stored on the platform.		

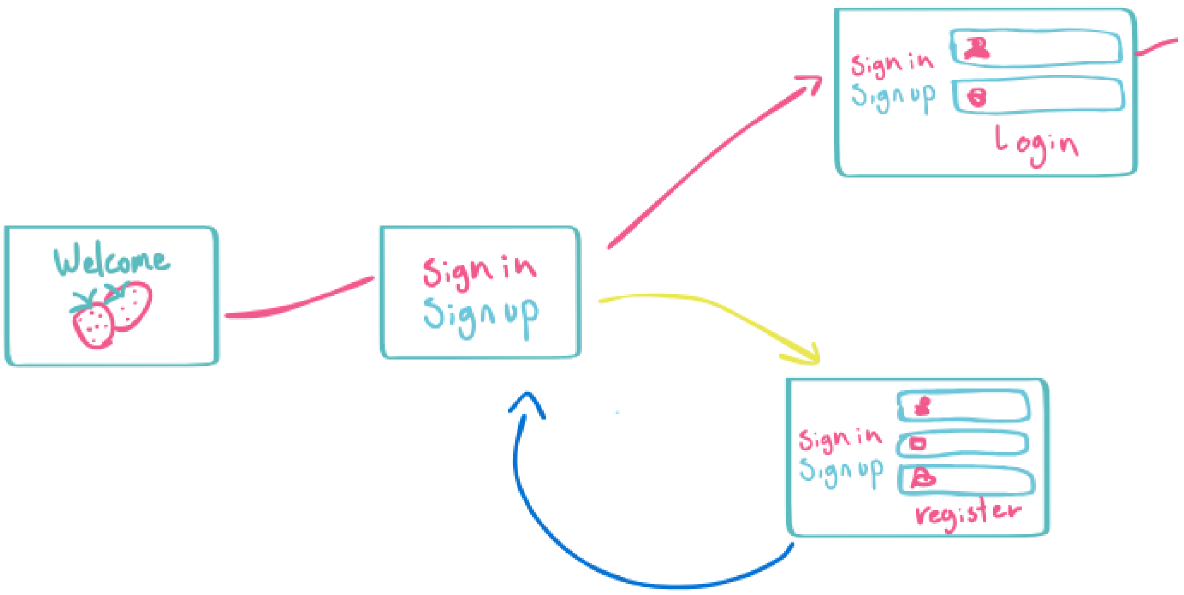
Title: Delete Photos	**Priority**: Medium	**Estimate**: 16-20 hours
User Story: **As** a User, **I** want to delete images that I have uploaded, So that I can remove content I no longer need to.		
Acceptance criteria: **Given** that I am on my gallery page, **When** I select an image and press the delete button, **Then** the image should be permanently removed from the app.		

Title: Save Photos	**Priority**: Medium	**Estimate**: 16-20 hours
User Story: **As a** User, **I** want to save images by downloading them, **So that I** can have the picture on my device.		
Acceptance criteria: **Given** that I am viewing an image, **When** I choose to download, **Then** the image should be saved on my device.		

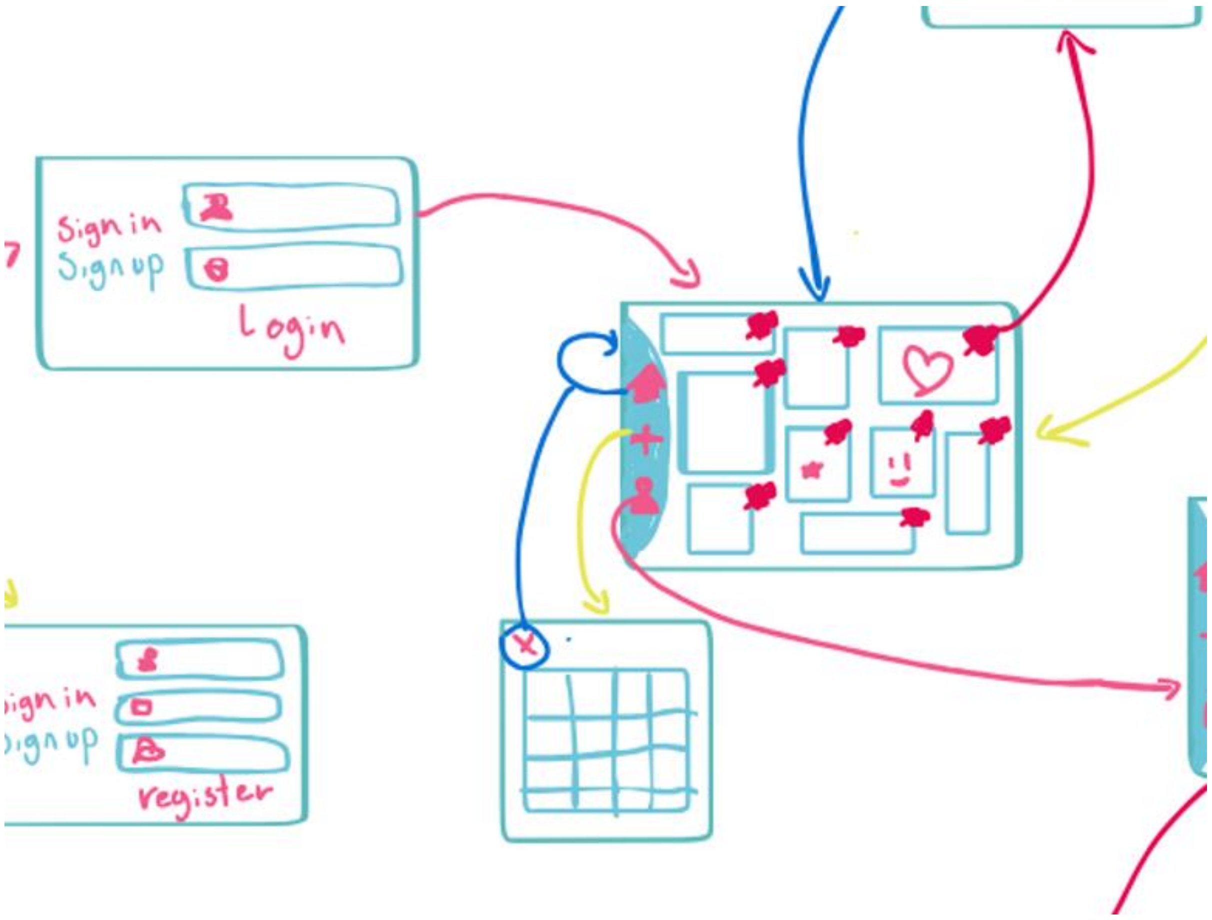
Title: Create Folder	**Priority**: High	**Estimate**: 20-24 hours
User Story: **As a** User, **I want** to create custom folders to organize selected photos, **So that I** can easily access and view specific images later without having to search through my entire collection.		
Acceptance criteria: **Given** that I am logged in and viewing my photo collection, **When** I create a new folder and assign specific photos to it, **Then** the system should store the folder with the selected images and allow me to access it from my folders list at any time.		

3. MOCKUPS

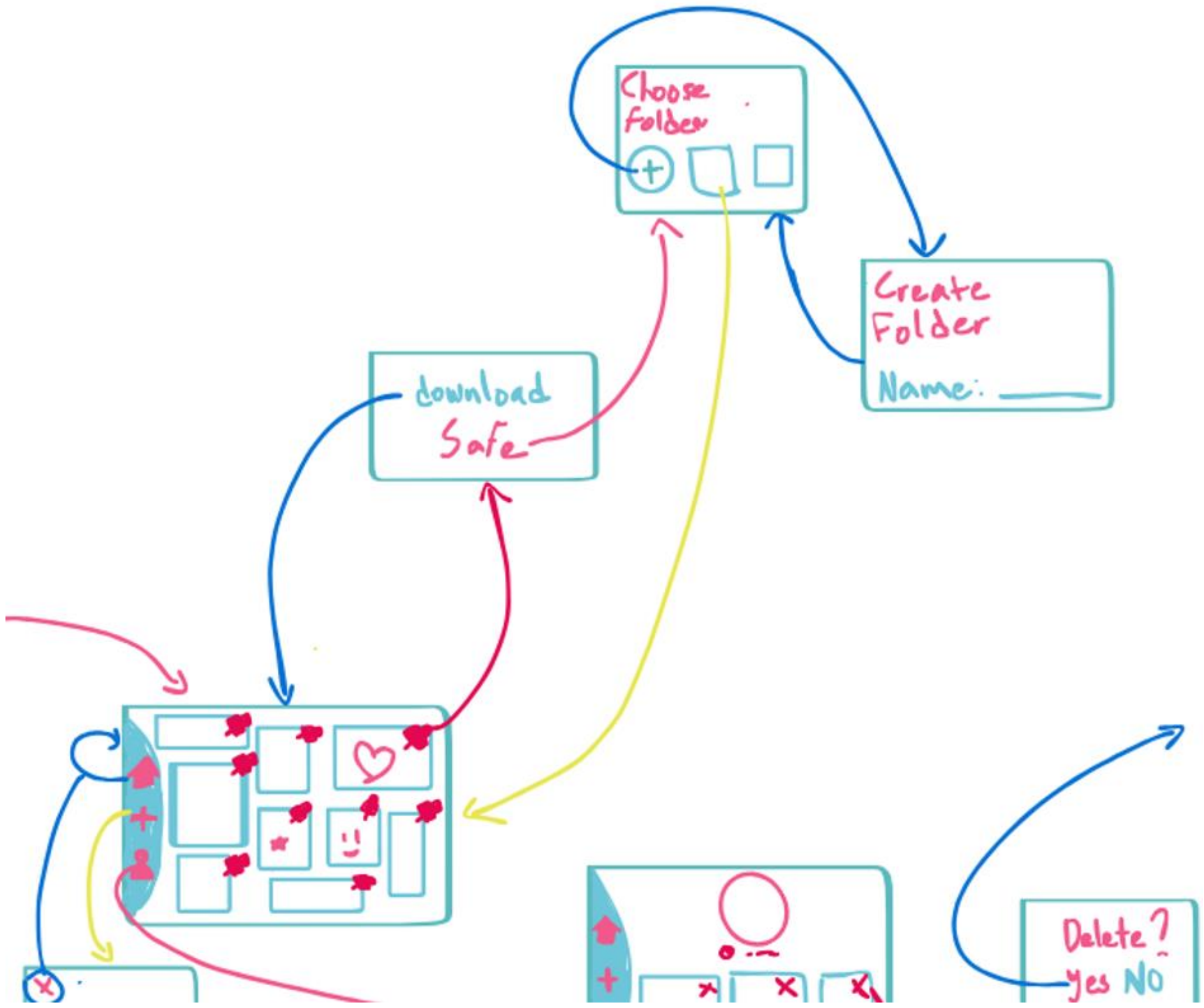
Mockups drawn diagram



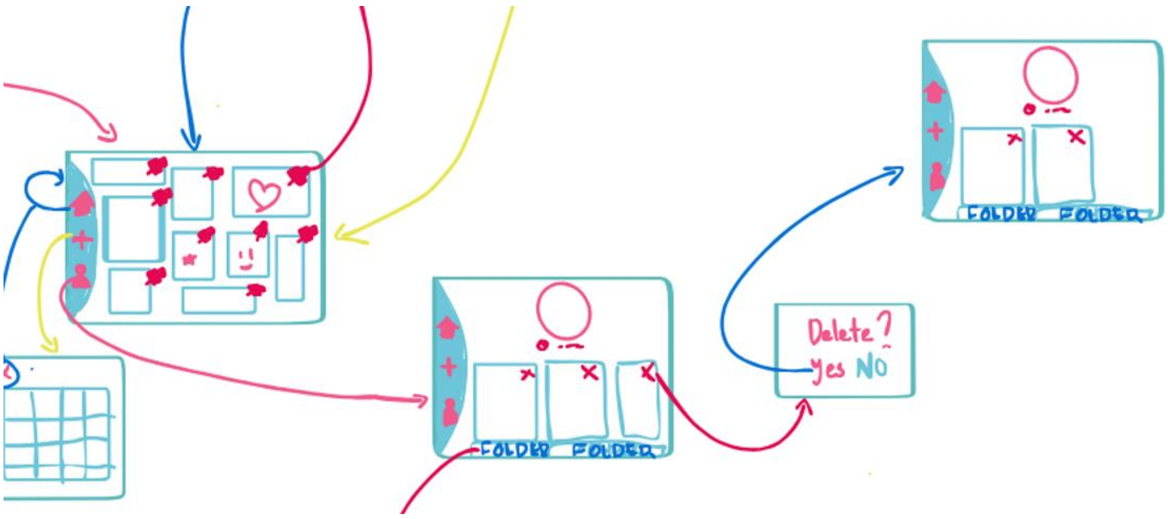
The mockup sequence begins with a loading screen, followed by two main options: **Sign In** and **Sign Up**. Selecting **Sign Up** leads the user through registration and then redirects back to the login screen.



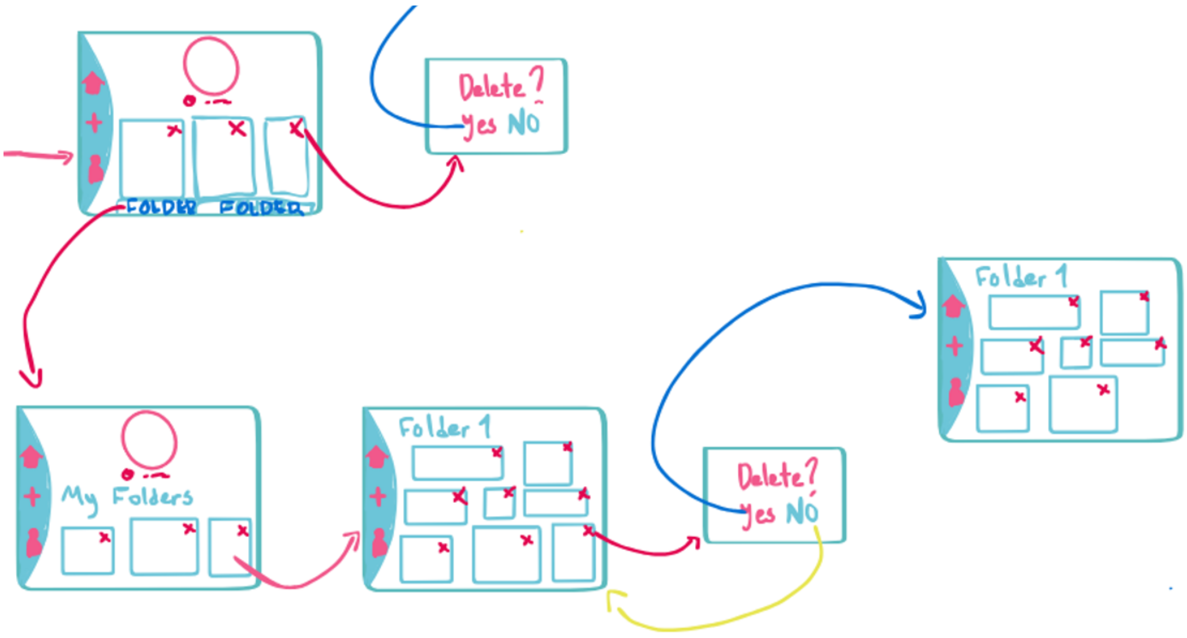
Upon successful login, the user lands on the **main page**, which displays various multimedia content (photos, videos, GIFs) with interactive pins overlaying each item. The interface offers three main actions: selecting the **plus (+)** button allows the user to upload new multimedia; the **cancel** button returns to the main page; and the **home icon** simply reloads the main feed.



Clicking on a pin opens a modal with options to **download** or **save** the item. Choosing **save** prompts the user to select a folder. A **plus (+)** icon on this view allows the user to create a new folder by assigning it a name. After creating or selecting a folder, the user is returned to the folder selection view or the main page.



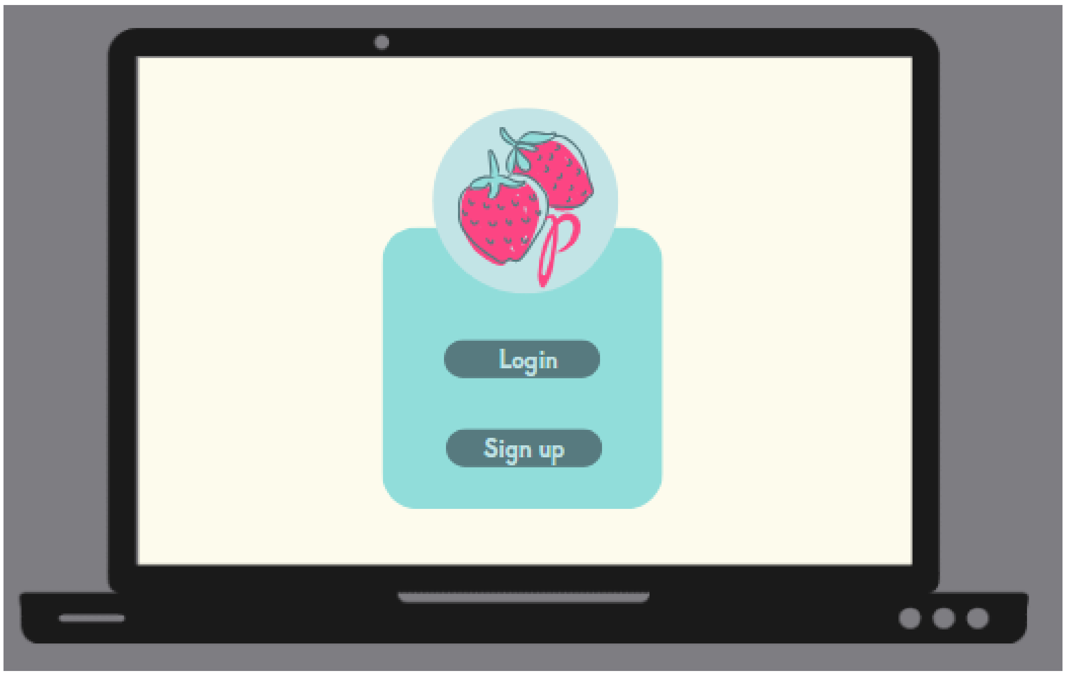
From the main interface, selecting the **profile icon** takes the user to their personal profile, where they can view all multimedia they've uploaded, each marked with an **X** to enable permanent deletion.



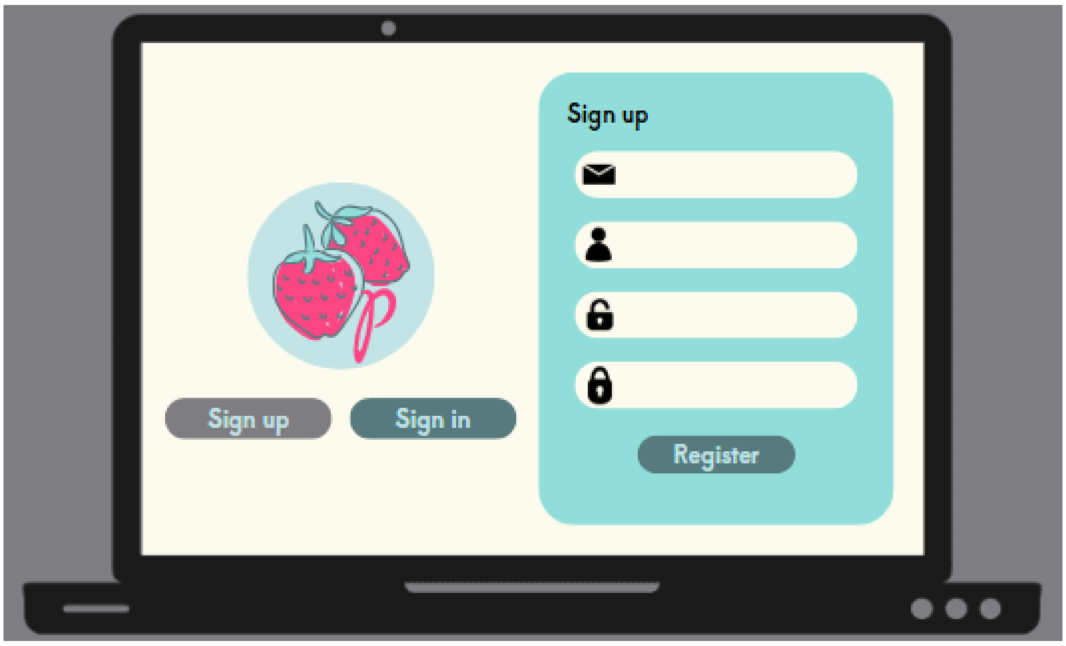
Below the multimedia section, folders are displayed. Selecting the **folders** tab transitions the view to show only the user's folders. Opening a folder displays all media stored inside it, each with an **X** for optional removal from that folder only (the item remains in the general folder). Returning to the profile view, if the user chooses to delete a file from their uploaded items using the **X**, that multimedia file is permanently deleted from the general folder.

INTERFACE

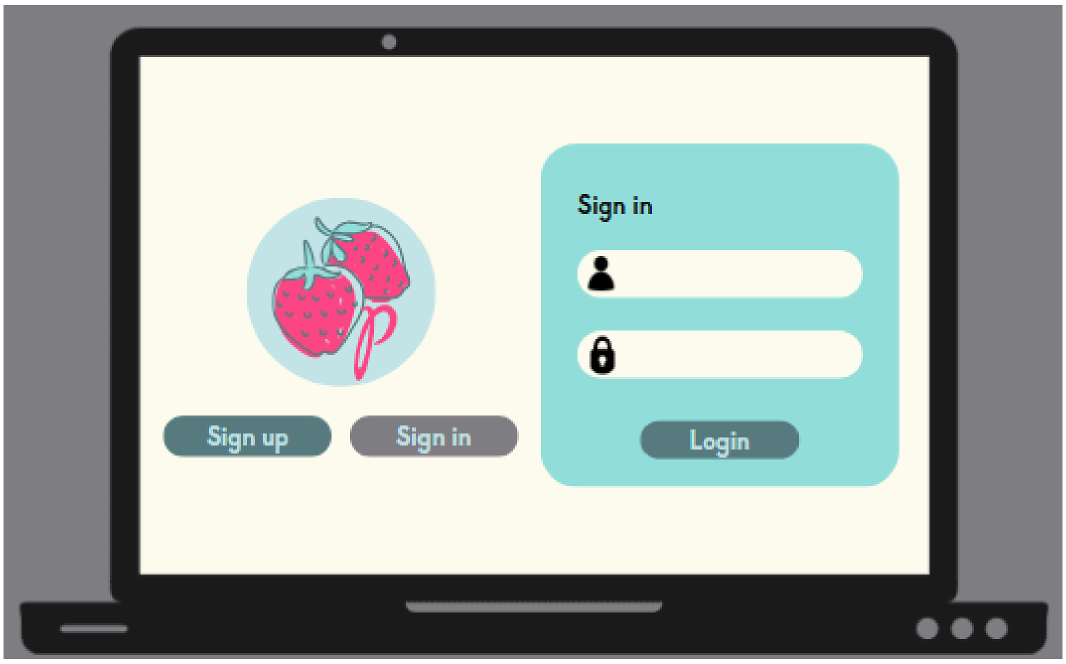
When the user enters the application, has the option to register or login if they already have an account



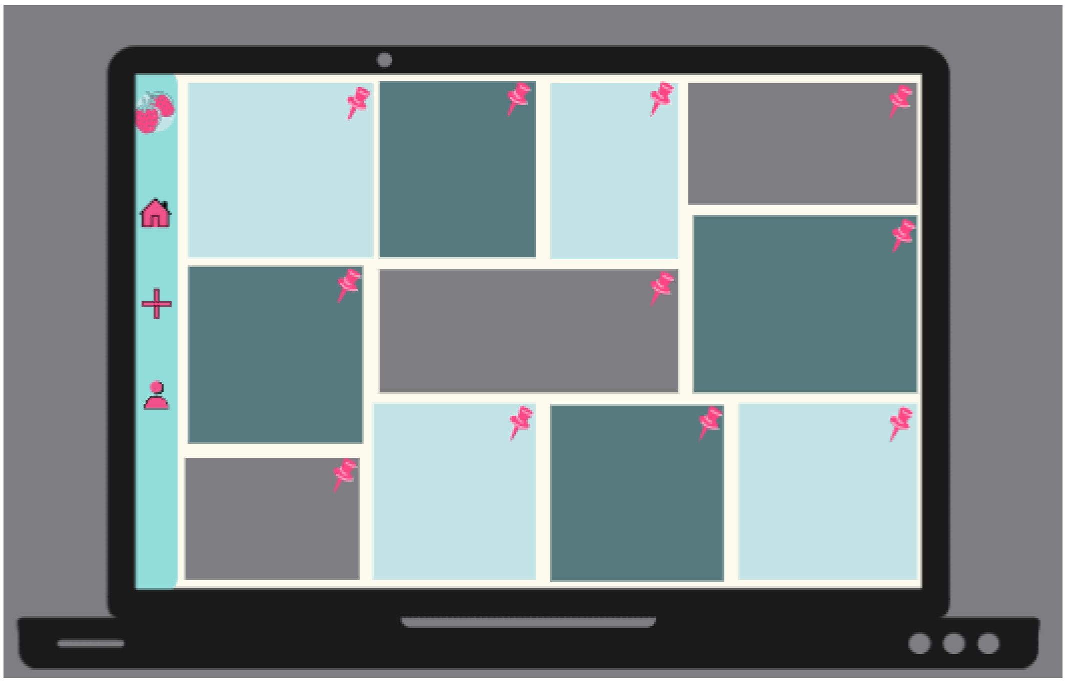
If the user decides to register, will be directed to the following interface:



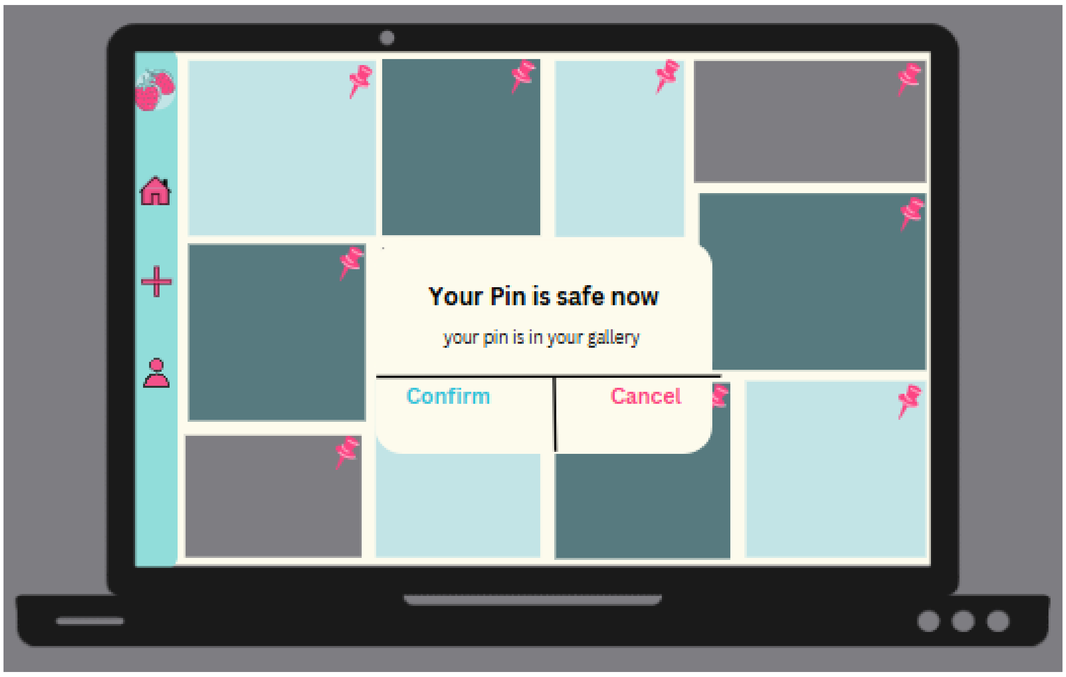
If, instead, decide to log in their account or they have already registered and are about to log in, they will be redirected to the following interface:



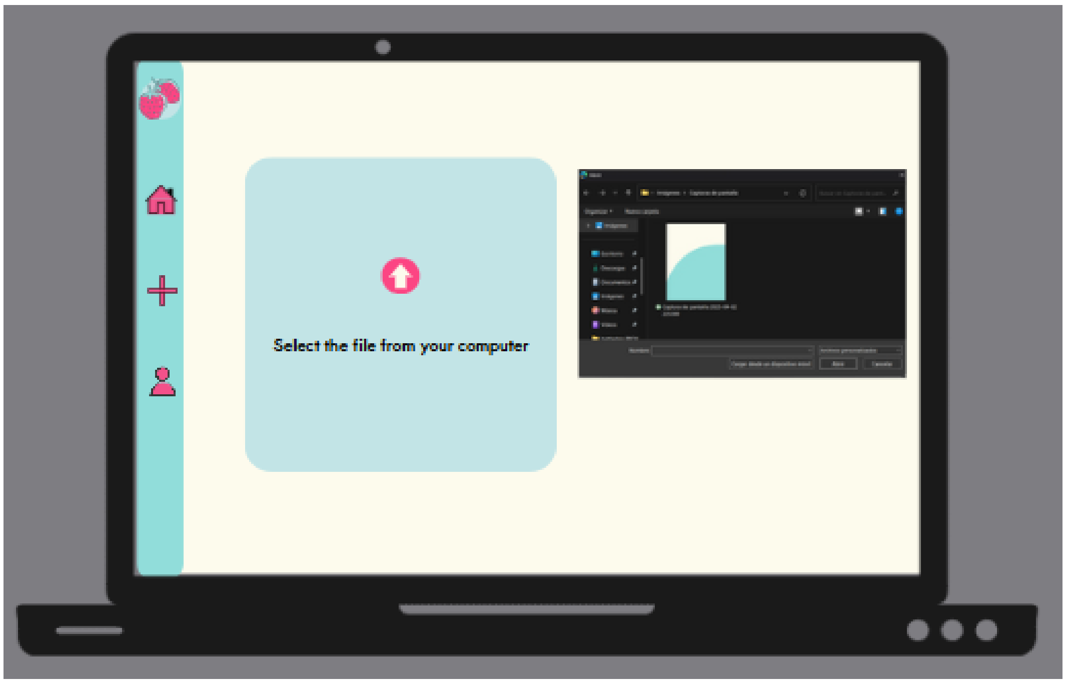
Once the user is logged in, they will be able to enter completely to the application, and it will be presented with the main interface.



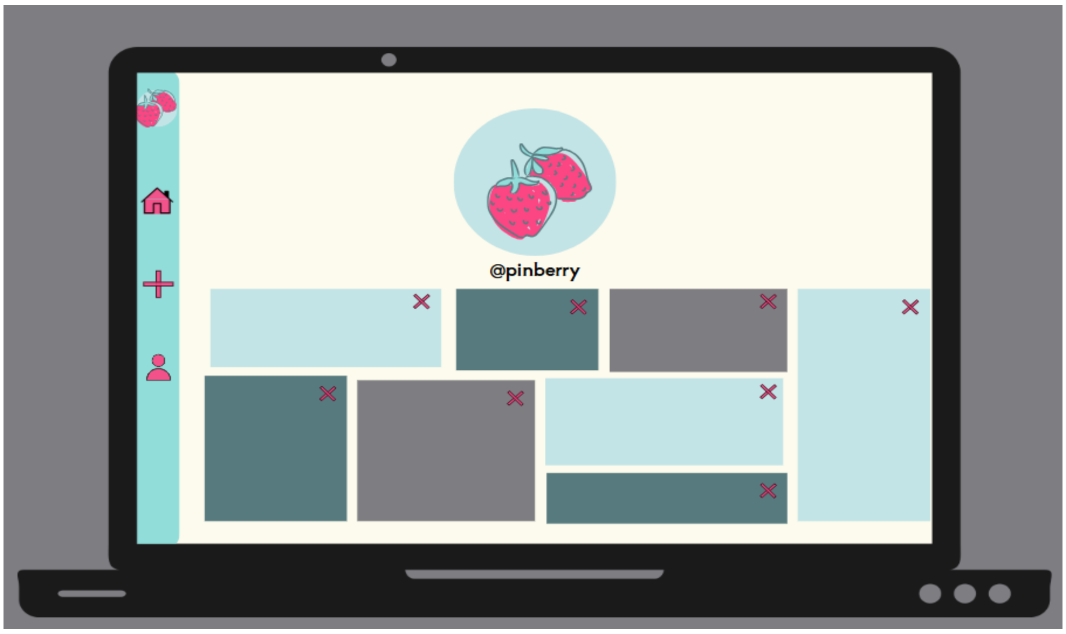
In this main interface, the user will be able to see the various photos, videos and gifs that have been uploaded by other users and if he/she wishes, will be able to save the multimedia of his/her choice, if they click the pink pin. Then it is going to appear a warning box showing that the pin has been saved.


Your Pin is safe now
your pin is in your gallery
Confirm
Cancel

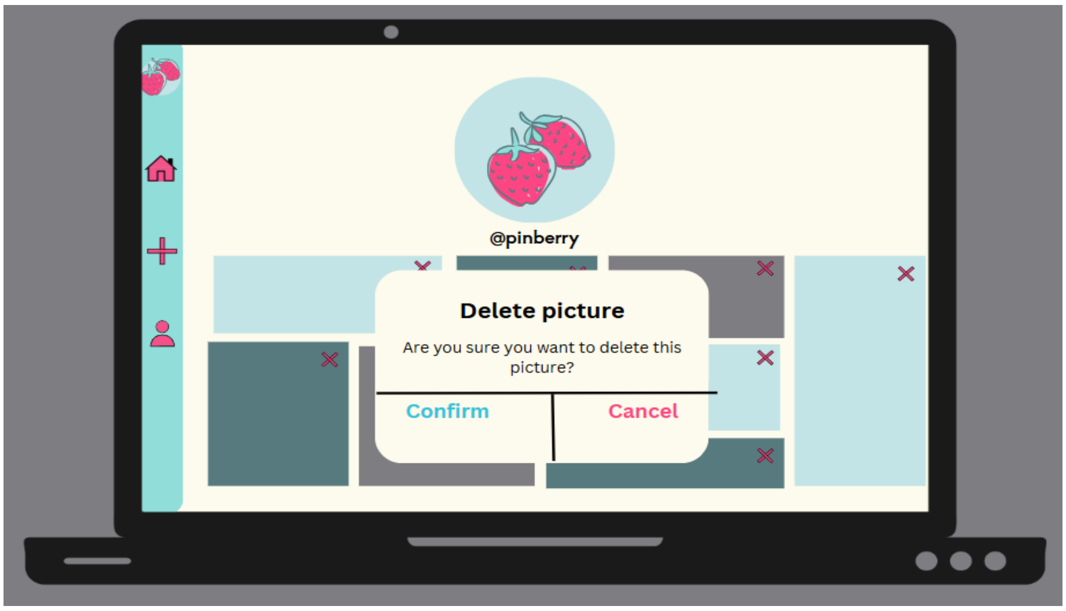
Also, appart from being able to save multimedia they want, the user can also upload multimedia to the application from their device. They just need to click on the plus in the middle of the screen.



In addition, if the user clicks on the person's icon at the bottom right, they can also review their own profile to review the multimedia that has been uploaded



Once they are in their own profile, they also have the option to delete any of the multimedia they have previously uploaded. Just click on the "X" found at the top right corner of the multimedia that they want to remove. Once they click on the "X" a warning box will appear to confirm if they really want to delete the photo



Delete picture

Are you sure you want to delete this picture?

4. CRC Cards:

Class: User	
Responsabilities:	**Collaborator:**
– Register with username, password	• Multimedia
– Log in	• Folder
– Upload multimedia	• App
– Delete multimedia	
– Download multimedia	

Interface: Multimedia	
Responsabilities:	**Collaborator:**
– Define common interface for media files.	• Photo
– Provide general methods: getRoute(), getName(), getSize(), getDate()	• Gif
	• Video
	• User
	• Folder

Class: Photo	
Responsabilities: – Store and retrieve image-specific metadata (name, size, path, date) – Inherit and implement methods from Multimedia	**Collaborator:** • Multimedia • User • Folder

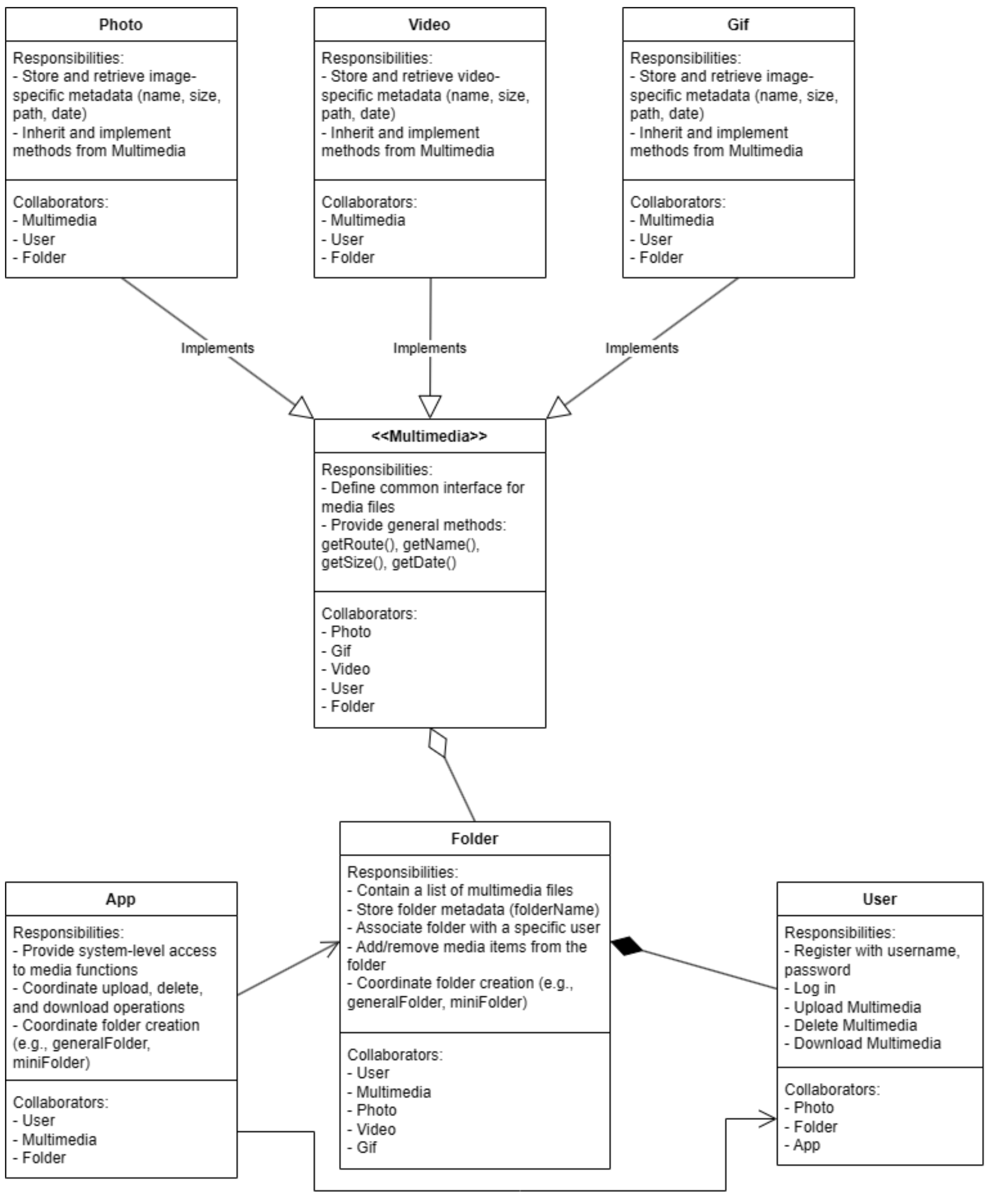
Class: Gif	
Responsabilities: – Store and retrieve gif-specific metadata (name, size, path, date) – Inherit and implement methods from Multimedia	**Collaborator:** • Multimedia • User • Folder

Class: Video

Responsabilities:	**Collaborator:**
– Store and retrieve video-specific metadata (name, size, path, date) – Inherit and implement methods from Multimedia	• Multimedia • User • Folder

Class: Folders	
Responsabilities: – Contain a list of multimedia files – Store folder metadata (folderName) – Associate folder with a specific user – Add/remove media items from the folder – Coordinate folder creation (e.g., generalFolder, miniFolder)	**Collaborator:** • User • Multimedia • Photo • Video • Gif

Class: App	
Responsabilities: – Provide system-level access to media functions – Coordinate upload, delete, and download operations – Coordinate folder creation (e.g., generalFolder, miniFolder)	**Collaborator:** • User • Multimedia • Folder



Photo

Responsibilities:
- Store and retrieve image-specific metadata (name, size, path, date)
- Inherit and implement methods from Multimedia

Collaborators:
- Multimedia
- User
- Folder

Video

Responsibilities:
- Store and retrieve video-specific metadata (name, size, path, date)
- Inherit and implement methods from Multimedia

Collaborators:
- Multimedia
- User
- Folder

Gif

Responsibilities:
- Store and retrieve image-specific metadata (name, size, path, date)
- Inherit and implement methods from Multimedia

Collaborators:
- Multimedia
- User
- Folder

<<Multimedia>>

Responsibilities:
- Define common interface for media files
- Provide general methods: getRoute(), getName(), getSize(), getDate()

Collaborators:
- Photo
- Gif
- Video
- User
- Folder

App

Responsibilities:
- Provide system-level access to media functions
- Coordinate upload, delete, and download operations
- Coordinate folder creation (e.g., generalFolder, miniFolder)

Collaborators:
- User
- Multimedia
- Folder

Folder

Responsibilities:
- Contain a list of multimedia files
- Store folder metadata (folderName)
- Associate folder with a specific user
- Add/remove media items from the folder
- Coordinate folder creation (e.g., generalFolder, miniFolder)

Collaborators:
- User
- Multimedia
- Photo
- Video
- Gif

User

Responsibilities:
- Register with username, password
- Log in
- Upload Multimedia
- Delete Multimedia
- Download Multimedia

Collaborators:
- Photo
- Folder
- App

Diagram showing the relationships between CRC Cards

1. UPDATES

Changes from the second workshop:

Removed the Folder class entirely

- The Folder class was deleted from the model to simplify the architecture and reduce unnecessary complexity. Since folder structures were not essential for meeting functional requirements, all associations, responsibilities, and collaborations related to this class were eliminated from the design.

Refactored the Photo class

- Attributes were adjusted to reflect a more minimal and relevant data structure focused on image management Methods related to folder association and organization were removed.
- The next Methods were removed:
- download(): Allows the user to download the image to their local storage.
- delete(): Permanently removes an uploaded image from the system.
- upload(): Uploads a new image to the system and links it to the user.

Updated the User class

- The methods updatePassword() and updateProfilePhoto() were removed as they were no longer aligned with the current scope of the application.
- Two new responsibilities were added to the class CRC (Class-Responsibility-Collaborator) card:
- The user can now delete images (deletePicture()).

- The user can now download images (downloadPicture()).
- These actions reflect direct interaction with image content, reinforcing the shift in focus from profile management to image manipulation.

Introduced StorageService as a collaborator

- Given the user's new responsibilities related to file operations (uploading, downloading, and deleting images), StorageService was added as a collaborator to handle persistent storage operations. This ensures a clear separation of concerns, where storage logic is centralized and reusable across components.

Rewritten user stories to align with the revised class structure

- All user stories were updated to be consistent with the modified responsibilities of User and Photo. This ensures that each requirement (e.g., uploading an image, saving it, downloading it, deleting it) is traceable to concrete class methods and interactions in the system design.

Removed retrievePhotos() method

- This method was eliminated as part of simplifying the application logic. Image retrieval is now handled implicitly through user interactions such as viewing the gallery or initiating a download, removing the need for a separate retrieval function.

Changes from the third workshop:

System Design and Class Structure Updates in PinBerry

Following guidance from our instructor and the continued application of SOLID principles, several significant architectural updates have been implemented in the PinBerry project to improve cohesion, reduce complexity, and enhance clarity.

1. Class Removals and Justification

We removed three previously defined classes: AuthenticationService, Gallery, and StorageService. These components were deemed unnecessary, as their functionalities were either redundant or could be more effectively handled by other parts of the system. Authentication and storage operations were streamlined and integrated directly within core logic, eliminating overhead.

2. Reintroduction and Redefinition of Folders

Initially removed, the Folder class was reinstated based on updated design needs. It now serves as the core container for multimedia organization—essentially replacing the need for a separate Gallery class. Folders now encapsulate a collection of media items (Photo, Video, Gif) and act as the organizational backbone of the user interface. Three types of folders are defined:

- General Folder: Contains all uploaded content, serving as a global "For You" feed.
- Mini Folders: Created by users to save selected multimedia for future reference.
- User Folder: Automatically generated upon user registration, storing all uploads by that user. Only media within this folder can be permanently deleted by its owner.

3. Introduction of the Multimedia Interface and Class Diagram Design

A key architectural enhancement in the current version of the PinBerry system is the introduction of the Multimedia interface. This interface was designed to abstract the common behaviors and properties of different media types—namely Photo, Video, and Gif. All three classes implement this interface, thereby inheriting its contract while maintaining their distinct identities. Each subclass provides metadata access methods such as getRoute(), getName(), getSize(), and getDate()—centralizing shared logic while adhering to the Interface Segregation Principle and the Open/Closed Principle from the SOLID paradigm.

The decision to incorporate the Multimedia interface was not only driven by code organization but also reflected in the design of the class diagram itself. The structure promotes polymorphism, enabling the Folder class to maintain a homogeneous list of heterogeneous media objects through the Multimedia type. This abstraction allows Folder to treat all media types uniformly without depending on their specific implementations, respecting the Dependency Inversion Principle and significantly improving the flexibility and extensibility of the system. Moreover, the diagram models two levels of aggregation that clarify ownership and lifecycle constraints within the domain:

- **Folder → Multimedia (Aggregation):** Each Folder contains a list of Multimedia items. This represents an aggregation because media items belong to folders but can theoretically exist independently of them in terms of logic (i.e., moving items between folders without destroying them). However, in the actual business logic, every media item must belong to at least one folder to be accessible.
- **User → Folder (Aggregation):** Each User aggregates multiple Folder objects. This design captures the logical grouping of folders under a user identity. When a new user is registered, a default User Folder is automatically created to store all uploads. Although folders belong to users, they are not composed strictly (i.e., deleting a user may not immediately delete all associated folders, depending on persistence rules), thereby justifying the aggregation of relationship rather than composition.

This model exemplifies how SOLID principles have guided not just the implementation code but also the conceptual modeling and class diagram construction. The use of the Multimedia interface and the aggregation relationships between User, Folder, and Multimedia demonstrate a robust and scalable architecture. These decisions reduce coupling, increase reusability, and ensure that the system remains extensible as new media types or folder behaviors are introduced in the future.

4. Revised Folder Composition and Multimedia Aggregation

The Folder class now holds a list of Multimedia objects, as well as its own name. It does not depend on specific media types, aligning with the Dependency Inversion Principle. This design also reflects proper aggregation: a folder can exist without media, but media cannot exist outside of a folder.

5. Updated User and App Roles

The User class retains essential behaviors such as login, and interaction with uploaded content. However, attributes like profile images have been removed for simplicity. Uploading, deleting, and downloading operations are now coordinated by the App class, which acts as the main orchestrator. It also handles folder creation logic, including the instantiation of a User Folder upon registration. This centralized coordination allows the App to manage the full media lifecycle across all users and folders, ensuring that only authenticated users can remove content they originally uploaded.

6. Project Refinement and Architectural Improvements Based on SOLID Principles

The project has undergone substantial refinement aimed at increasing clarity, technical depth, and alignment with industry-standard object-oriented practices—particularly the SOLID principles. The introduction section was revised to provide a more structured and explicit overview of the application's purpose.

- The motivation for choosing the project was clearly stated, followed by well-differentiated general and specific objectives. These were reformatted for clarity and conciseness, reflecting the architectural shift that placed Folder at the core of the media organization's logic.
- Functional requirements were rewritten in a more technical and descriptive format, moving away from user story language. This adjustment improves their precision and ensures they are actionable from an engineering standpoint.

- Non-functional requirements were also refined to be more measurable and concrete, emphasizing usability, maintainability, and scalability—directly reinforcing the SOLID-based structure.

- In response to feedback and evolving project needs, user stories were cleaned of non-essential formatting (such as color coding) to improve readability and better reflect the functional scope. A significant decision was made to remove mobile mockups and focus exclusively on desktop design. This narrowing of scope was strategically chosen to improve UI fidelity and development focus.

- The CRC cards were completely reworked to align with the revised architecture. Legacy classes such as AuthenticationService, Gallery, and StorageService were removed, and Folder was reintroduced as a principal entity. Classes were updated with clearer responsibilities and collaborators, adhering closely to the Single Responsibility Principle (SRP) by isolating distinct roles across components.

- The class diagram was redesigned from the ground up, using a minimalist visual approach for clarity. Relationships between classes were revised to reflect accurate aggregations and dependencies, following the Dependency Inversion Principle and Liskov Substitution Principle where applicable. The introduction of the Multimedia interface stands as a cornerstone of this redesign. It enables polymorphic behavior among Photo, Video, and Gif classes, allowing folders to aggregate them through a common type—cleanly separating abstractions from implementations as SOLID recommends.

- We reordered documentation content to prioritize sequence diagrams before implementation discussions, ensuring a more natural and understandable narrative flow. Each process (e.g., uploading, downloading, deleting media) was explicitly modeled using activity and sequence diagrams, reinforcing a Single Responsibility mindset and making system behavior more predictable and testable.
- Finally, tabular formats were adopted to organize specifications and processes, enhancing document readability. All these revisions stem from a commitment to rigorous software engineering practices and a deep integration of SOLID principles into every phase of the system's design and documentation.

Changes from the 4 workshop:

- **Cover Page**

  The title has been updated to **"School of Engineering"** for greater institutional precision.

- **Specific Objectives**

  We refined and reduced the objectives to four items, each clearly stating *what* we aim to do and *how* we will achieve it, enhancing both clarity and coherence.

- **Class Diagram Terminology**

  The verb *"extend"* has been replaced with *"implement"* in the class diagram to better reflect the actual technical relationships.

- **Summary of Previous Project Updates**

  A new section highlights enhancements made since earlier iterations, showcasing the system's progression and design evolution.

- **Concise Explanations**

  Extended bullet lists have been replaced by cohesive paragraphs, improving readability while preserving all key information.

- **Activity & Sequence Diagrams**

  We restructured this section to first explain core object-oriented principles before presenting the diagrams, ensuring a clearer conceptual foundation.

- **Sequence Diagram Enhancements**
  - Refined the diagrams to better represent the *request-response* flow.
  - Added commentary explaining why we introduced a `HashMap` in the design—to efficiently manage mapping between user sessions and photo references.

- **Folder/File Structure**

  We moved away from the "one folder per class" model, opting instead for a more modular and maintainable structure that scales with future development.

2. Technical Design (UML Diagrams)

In the initial stages of PinBerry's development, the application architecture followed a composition-based design that avoided inheritance due to the absence of shared behaviors and the lack of polymorphic needs. Classes such as User, Photo, Gallery, AuthenticationService, and StorageService were structured to encapsulate distinct responsibilities, and at that stage, inheritance was deemed unnecessary and potentially detrimental to clarity and maintainability.

However, as the system matured, particularly with the integration of multiple media types—Photo, Video, and Gif—a clear opportunity for abstraction emerged. To support scalability, reduce redundancy, and enable polymorphic behavior across different types of media, the project introduced a new interface named Multimedia. This interface defines a common contract for all media entities, encapsulating shared attributes and behaviors such as getRoute(), getName(), getSize(), and getDate(). Each media class now implements this interface, enabling the application to treat all media types uniformly within collections (e.g., in folders), without requiring knowledge of their specific implementations.

Inheritance, in the form of interface implementation, is now applied precisely where it adds value—across structurally similar entities (Photo, Video, Gif) that share behavior and need to be treated interchangeably. This refined architecture reflects a balanced and thoughtful application of object-oriented principles. It avoids unnecessary complexity while enabling code reuse, flexibility, and adherence to SOLID, especially where inheritance is both justified and necessary—in contrast to the original design that relied solely on composition.

2.1 Class Diagram

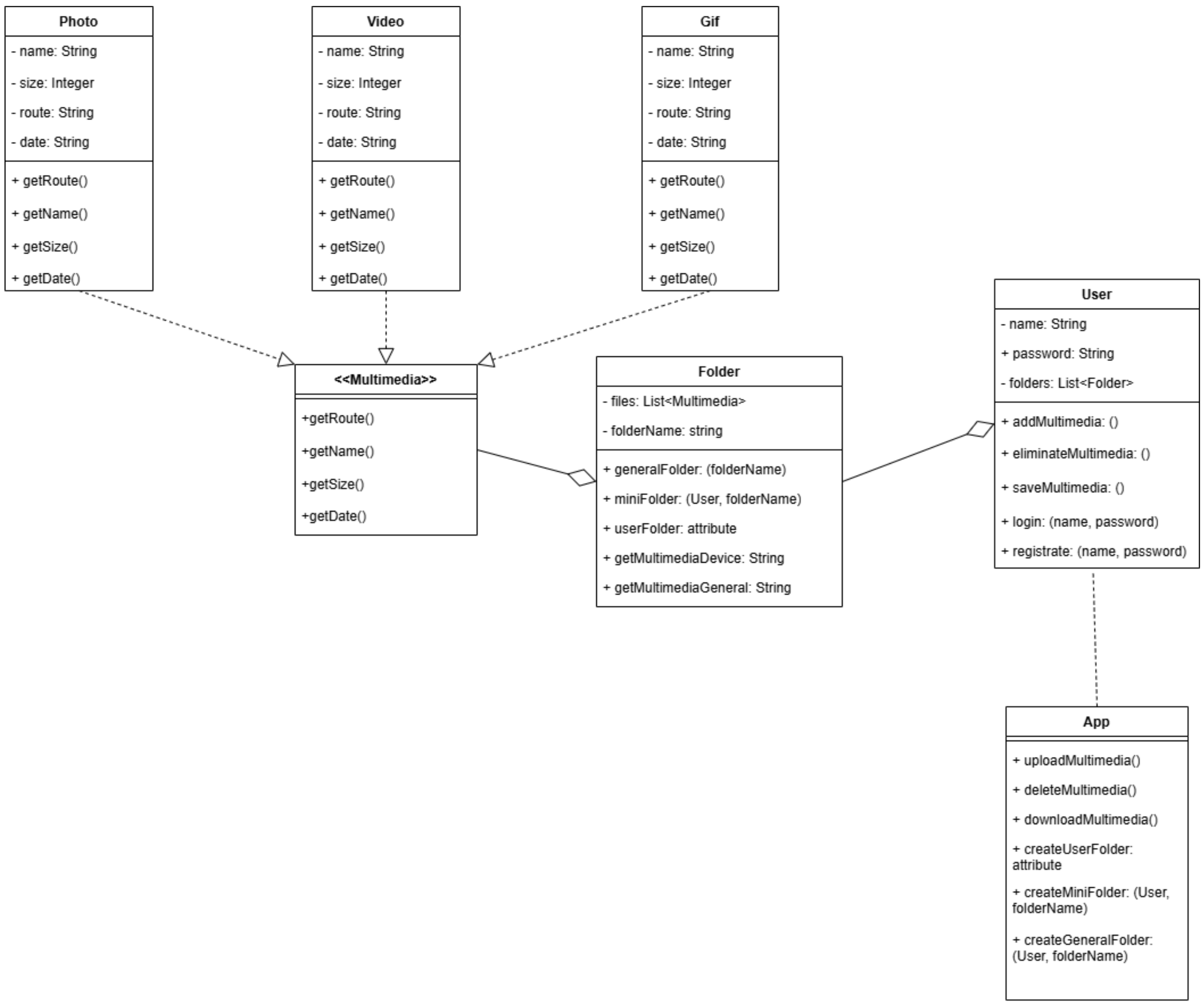
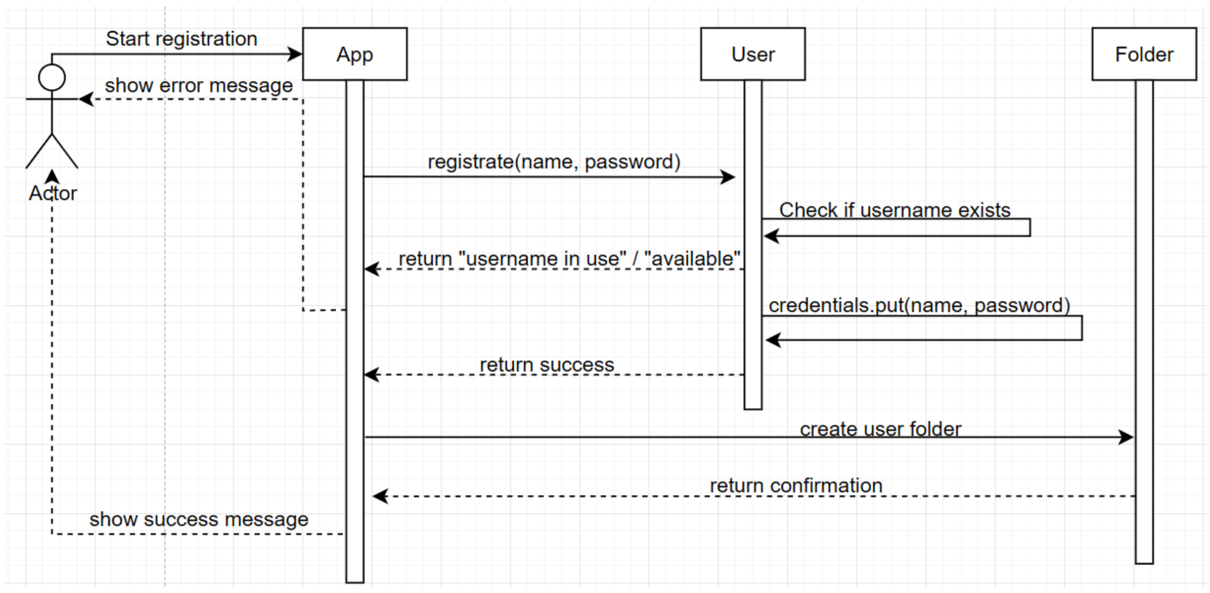


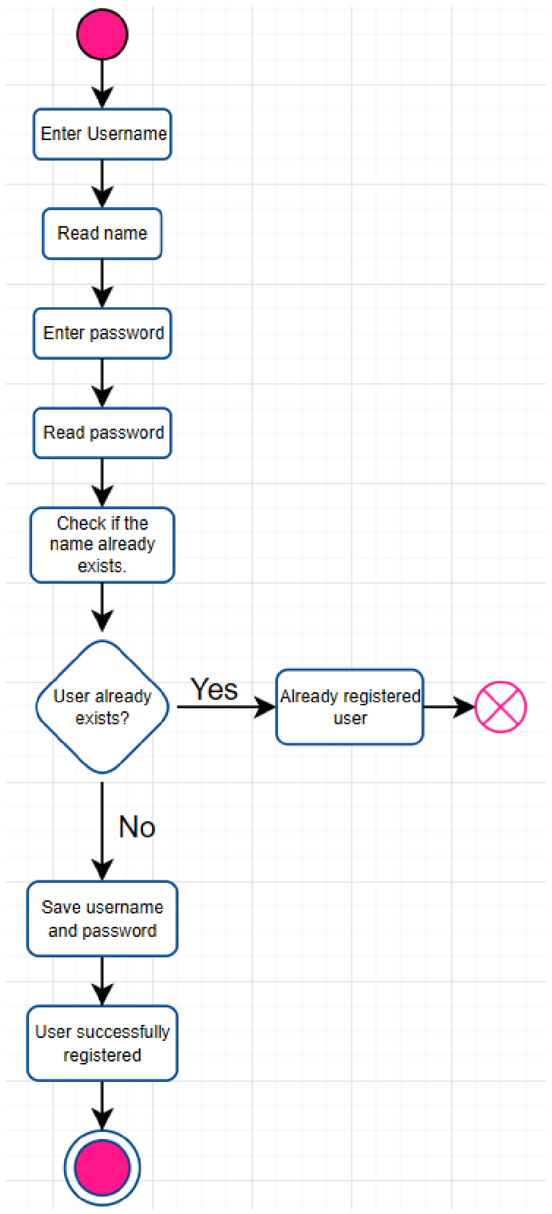
Photo
- name: String
- size: Integer
- route: String
- date: String
+ getRoute()
+ getName()
+ getSize()
+ getDate()
Video
- name: String
- size: Integer
- route: String
- date: String
+ getRoute()
+ getName()
+ getSize()
+ getDate()
Gif
- name: String
- size: Integer
- route: String
- date: String
+ getRoute()
+ getName()
+ getSize()
+ getDate()
<<Multimedia>>
+getRoute()
+getName()
+getSize()
+getDate()
Folder
- files: List<Multimedia>
- folderName: string
+ generalFolder: (folderName)
+ miniFolder: (User, folderName)
+ userFolder: attribute
+ getMultimediaDevice: String
+ getMultimediaGeneral: String
User
- name: String
+ password: String
- folders: List<Folder>
+ addMultimedia: ()
+ eliminateMultimedia: ()
+ saveMultimedia: ()
+ login: (name, password)
+ registrate: (name, password)
App
+ uploadMultimedia()
+ deleteMultimedia()
+ downloadMultimedia()
+ createUserFolder: attribute
+ createMiniFolder: (User, folderName)
+ createGeneralFolder: (User, folderName)

2.2 METHODS IMPLEMENTATION

2.2.1.1 Sequence Diagram (Register User)

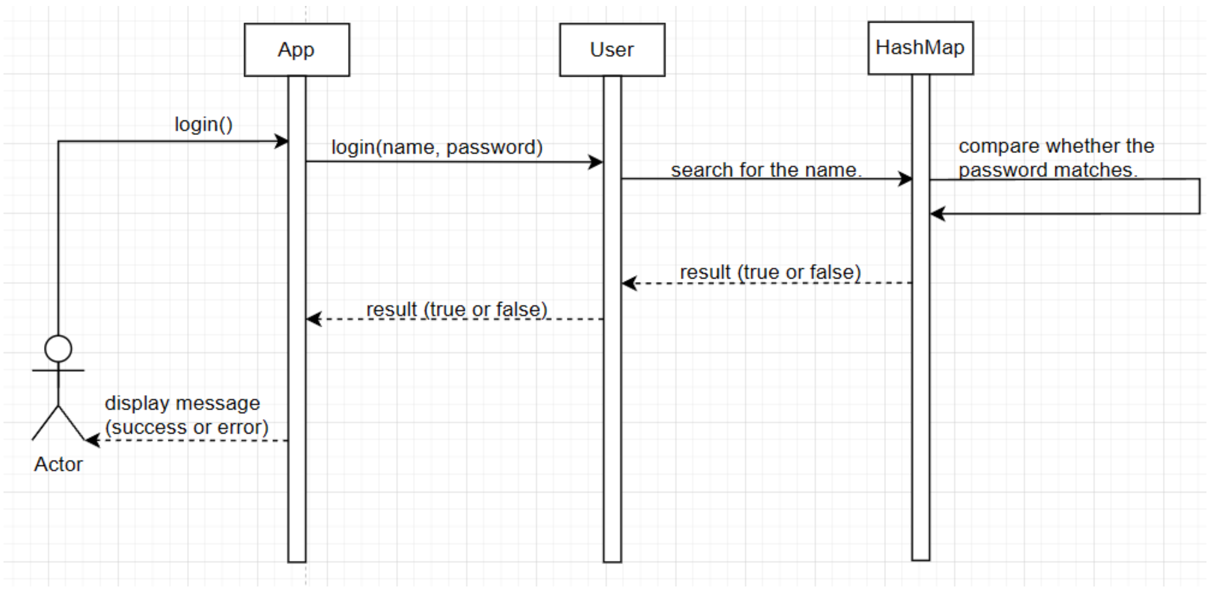


2.2.2.1 Activity diagrams (Register User)



- The App object calls the register() method from the User class, which requests user input through the console using Scanner. The system checks whether the username is already taken; if so, it notifies the user and stops the process. If the username is available, the user (name and password) is stored in an internal list, with the credentials protected through private attributes. Then, the general folder for that user is created (if it does not already exist), and the registration is completed. This method is independent, allowing it to evolve without affecting other parts of the system, and all logic remains encapsulated within the User class.

2.2.1.2 Sequence Diagram (Log in)



In our project, we use a dictionary to store all registered users. This dictionary connects each username to its corresponding user data (like the password or the User object). This is very useful during the login process.

The reason dictionaries are so important is because they let us find a user very quickly, just by using the username. Behind the scenes, dictionaries work using a system called HashMap, which means they don't have to go through every user one by one — they jump directly to the one they're looking for.

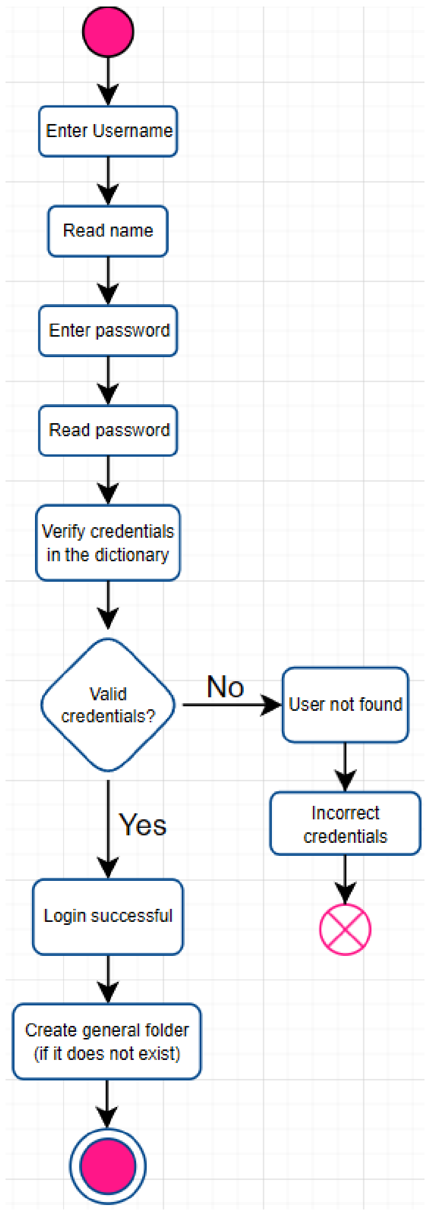
So, when a user tries to log in:

- The program checks if the username exists in the dictionary.
- If it does, it looks up the saved password and compares it with the one the user typed.
- If both match, the user is allowed in.

If we didn't use a dictionary and used something like a list instead, the program would have to check each user one by one, which would be slower and messier, especially if we have many users.

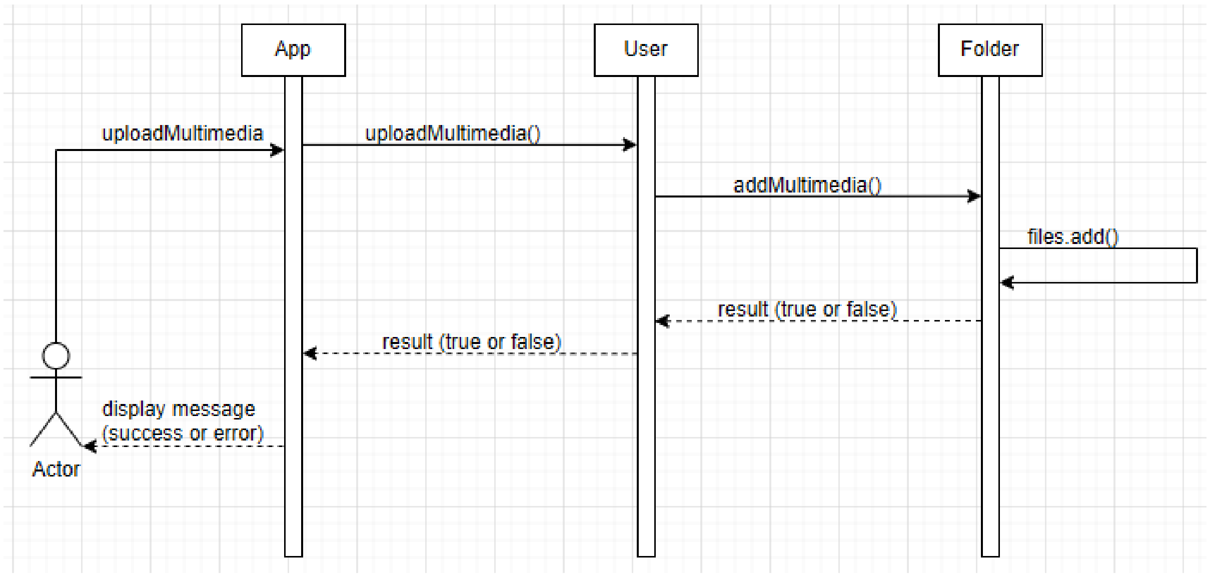
In summary, using a dictionary (based on the idea of a HashMap) makes our login system faster, easier to program, and more organized, because we can find and check users directly by their username without extra steps.

2.2.2.2 Activity diagrams (Login)

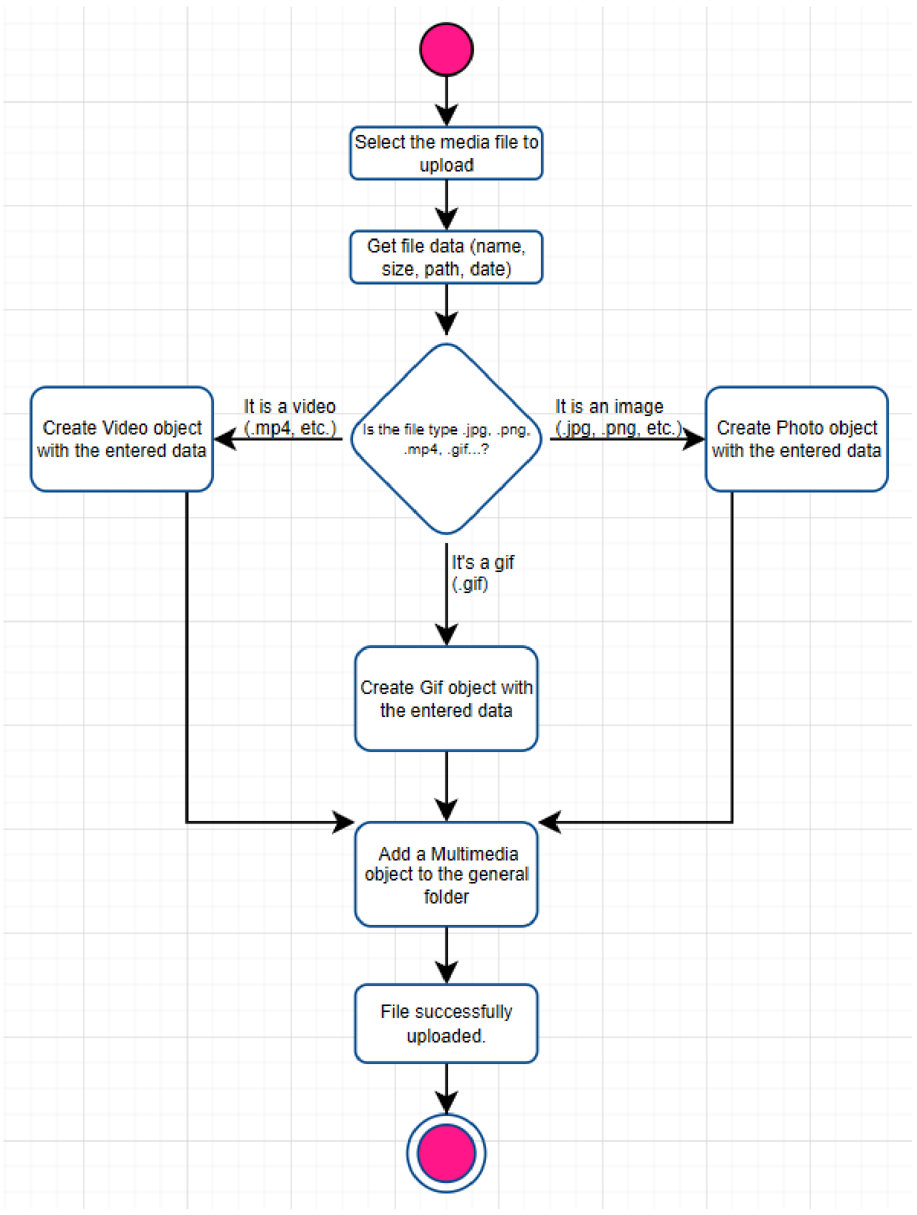


- The App object calls the login() method from the User class, which requests the username and password via the console using Scanner. The method scans the list of registered users and compares the entered credentials. If they match, the corresponding authenticated User object is returned; otherwise, an error message is displayed and the process is terminated. The credentials remain protected within the User class, which is solely responsible for handling access validation.

2.2.1.3 Sequence Diagram (Upload Multimedia)



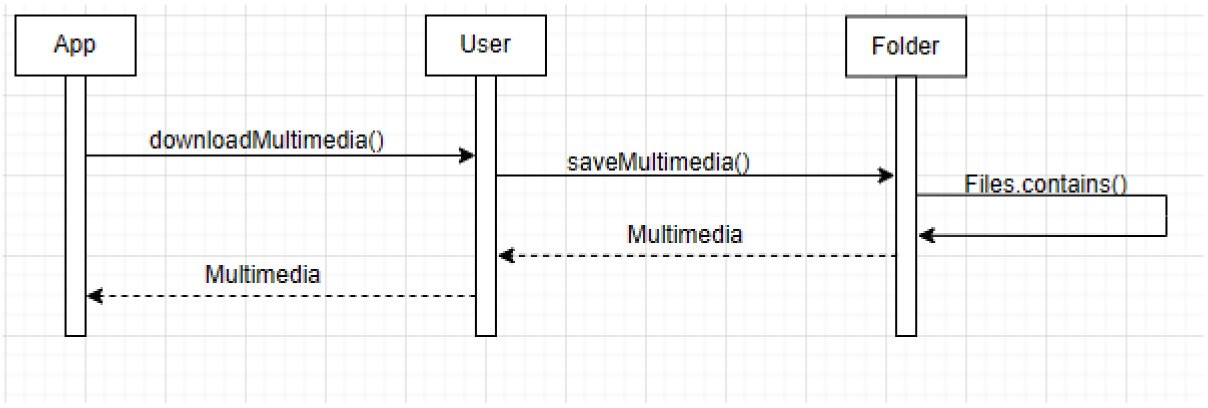
2.2.2.3 Activity diagrams (Upload Multimedia)



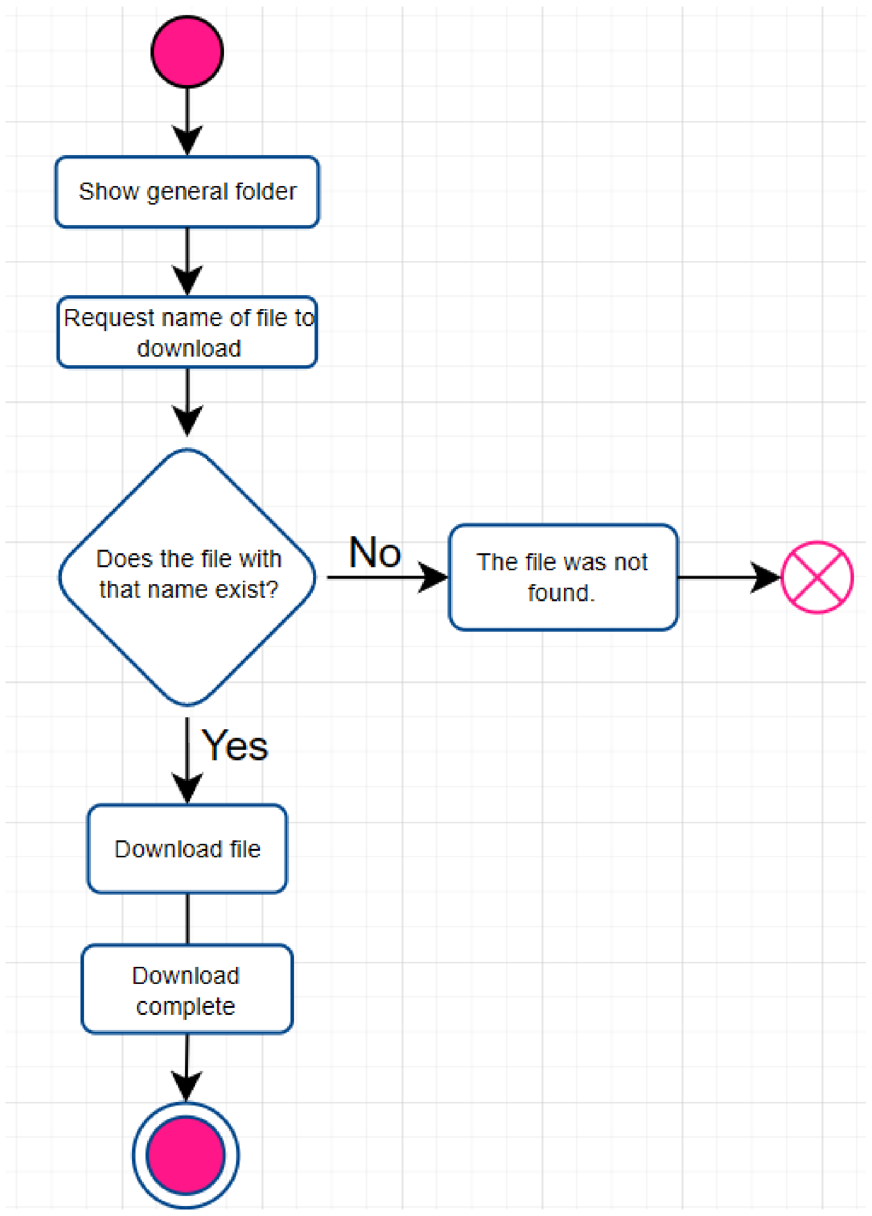
Select the media file to upload
Get file data (name, size, path, date)
Is the file type .jpg, .png, .mp4, .gif...?
It is a video (.mp4, etc.)
Create Video object with the entered data
It is an image (.jpg, .png, etc.)
Create Photo object with the entered data
It's a gif (.gif)
Create Gif object with the entered data
Add a Multimedia object to the general folder
File successfully uploaded.

- The App object calls its own uploadMultimedia() method, which receives a Multimedia object—whether a Photo, Video, or Gif, thanks to the use of polymorphism. Inside this method, the application accesses the corresponding User and locates the user's general folder, which was previously created when the application was initialized. Once found, the method addMultimedia(Multimedia) from the Folder class is called to store the file. The multimedia file is stored exclusively in the general folder, and each file maintains its internal data (name, path, size, date) as private attributes. The App.uploadMultimedia() method does not need to distinguish between file types, since all are handled through the Multimedia abstraction.

2.2.1.4 Sequence Diagram (Download Multimedia)

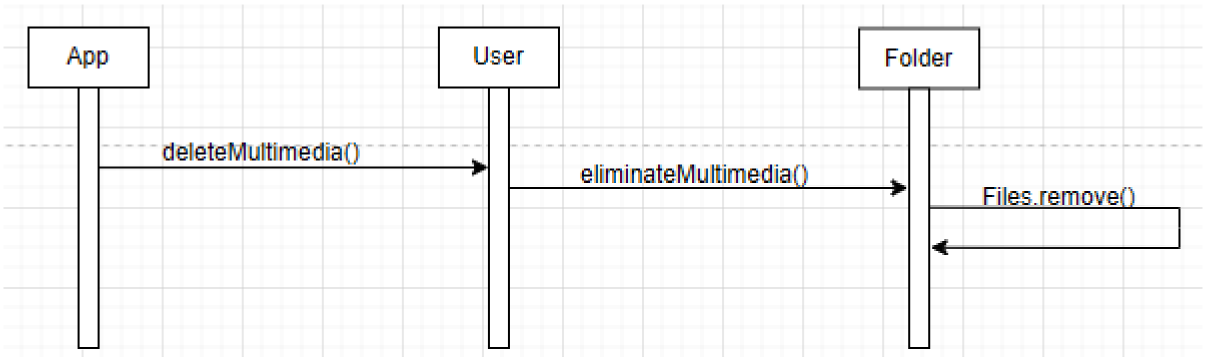


2.2.2.4 Activity diagrams (Download Multimedia)

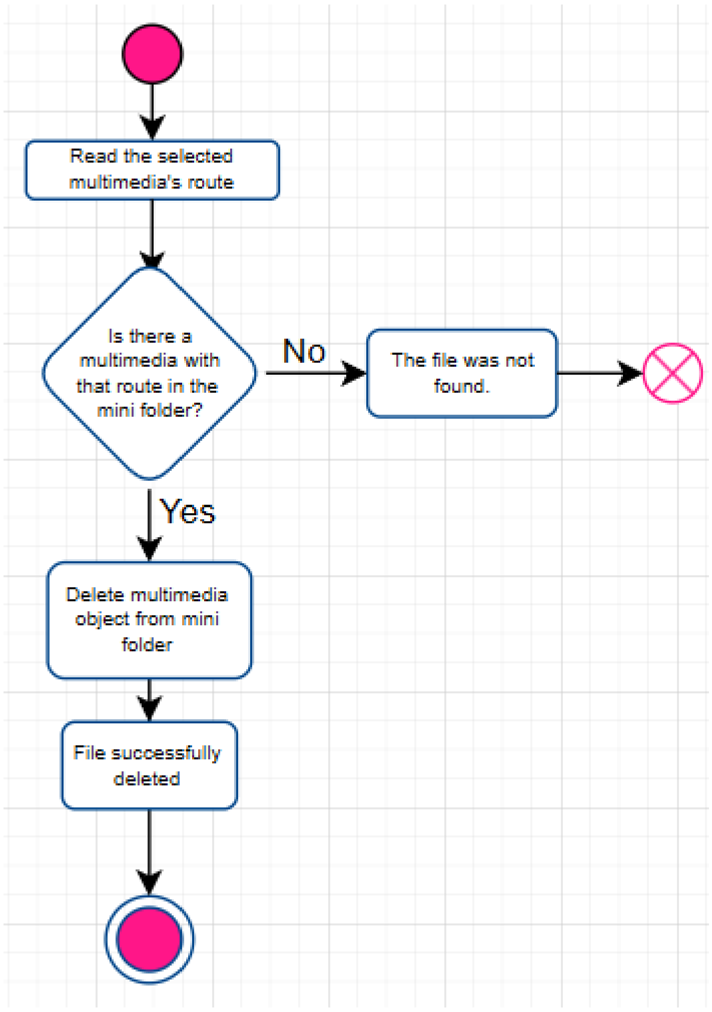


- The user initiates a request to download a file, prompting the app to call the downloadMultimedia(filename) method. The app directly accesses the generalFolder, which is a shared instance not tied to any specific user, and searches for a file matching the requested name. Inside the general folder, the list of Multimedia objects is scanned, comparing each object's name with the parameter. If a match is found, the download is simulated—either by printing to the console or adding the file to a “downloads” list. The search and access are performed uniformly across all Multimedia types, regardless of whether the file is a Photo, GIF, or Video, since all comply with the Multimedia interface.

2.2.1.5 Sequence Diagram (Delete Photo)

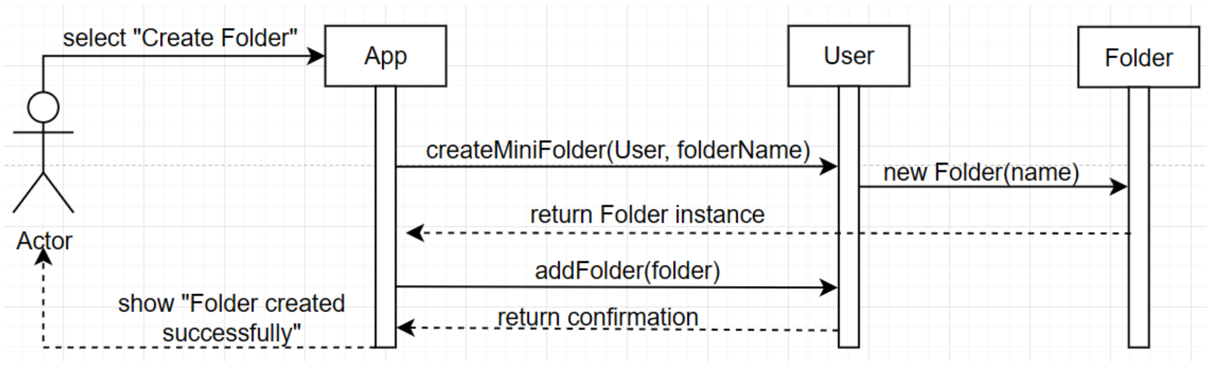


2.2.2.5 Activity diagrams (Delete Photo)

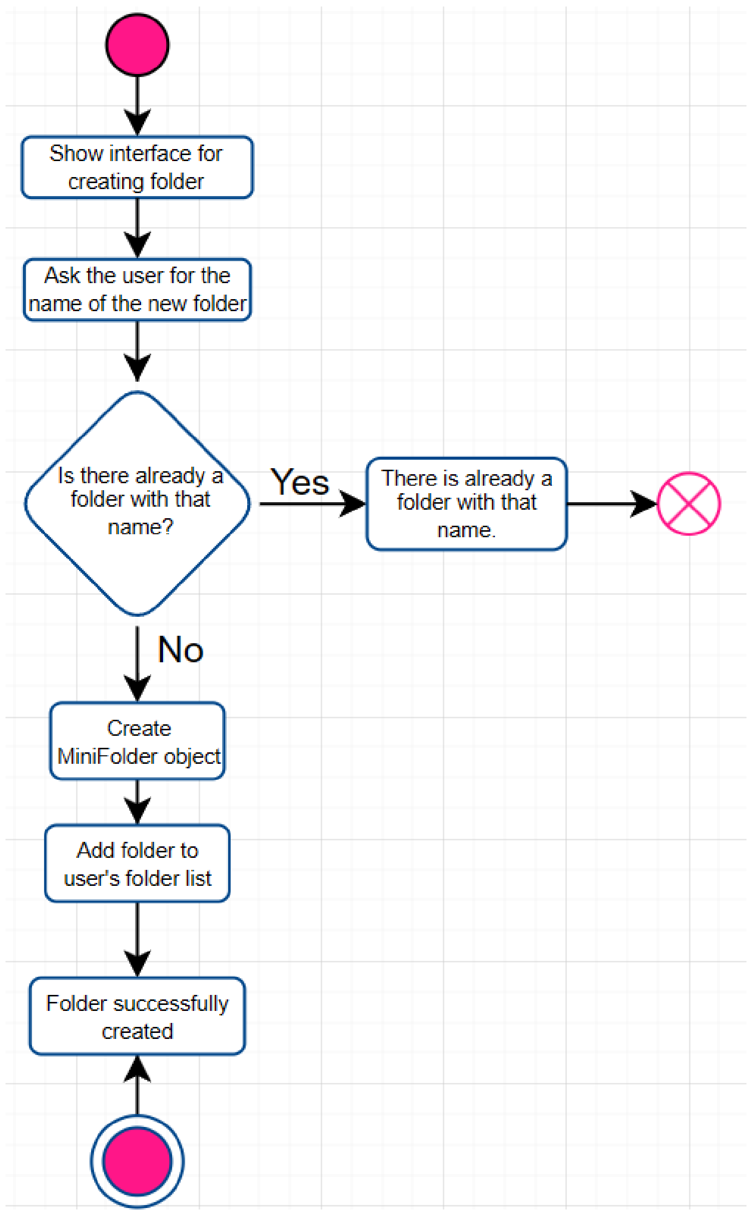

Read the selected multimedia's route
Is there a multimedia with that route in the mini folder?
No
The file was not found.
Yes
Delete multimedia object from mini folder
File successfully deleted

- The user requests to delete a file from the general folder, typically by providing the file's name. The App class invokes the deleteMultimedia(fileName) method, which accesses the shared generalFolder (not linked to any specific user). Inside the method, the list of files in the folder is traversed, and the name of each Multimedia object is compared to the input parameter. If a match is found, the corresponding object is removed from the list. A confirmation message is displayed if the deletion is successful; otherwise, an error message is shown. The general folder may contain different types of Multimedia (e.g., Photo, Video, Gif), but they are all treated the same way during the search and deletion, since they all implement the getName() method defined in the Multimedia interface.

2.2.1.6 Sequence Diagram (Create Folder)

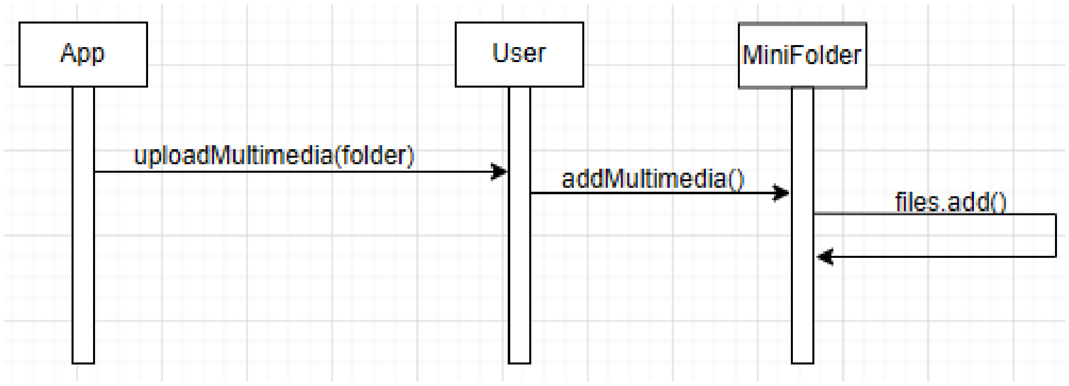


2.2.2.6 Activity diagrams (Create Folder)

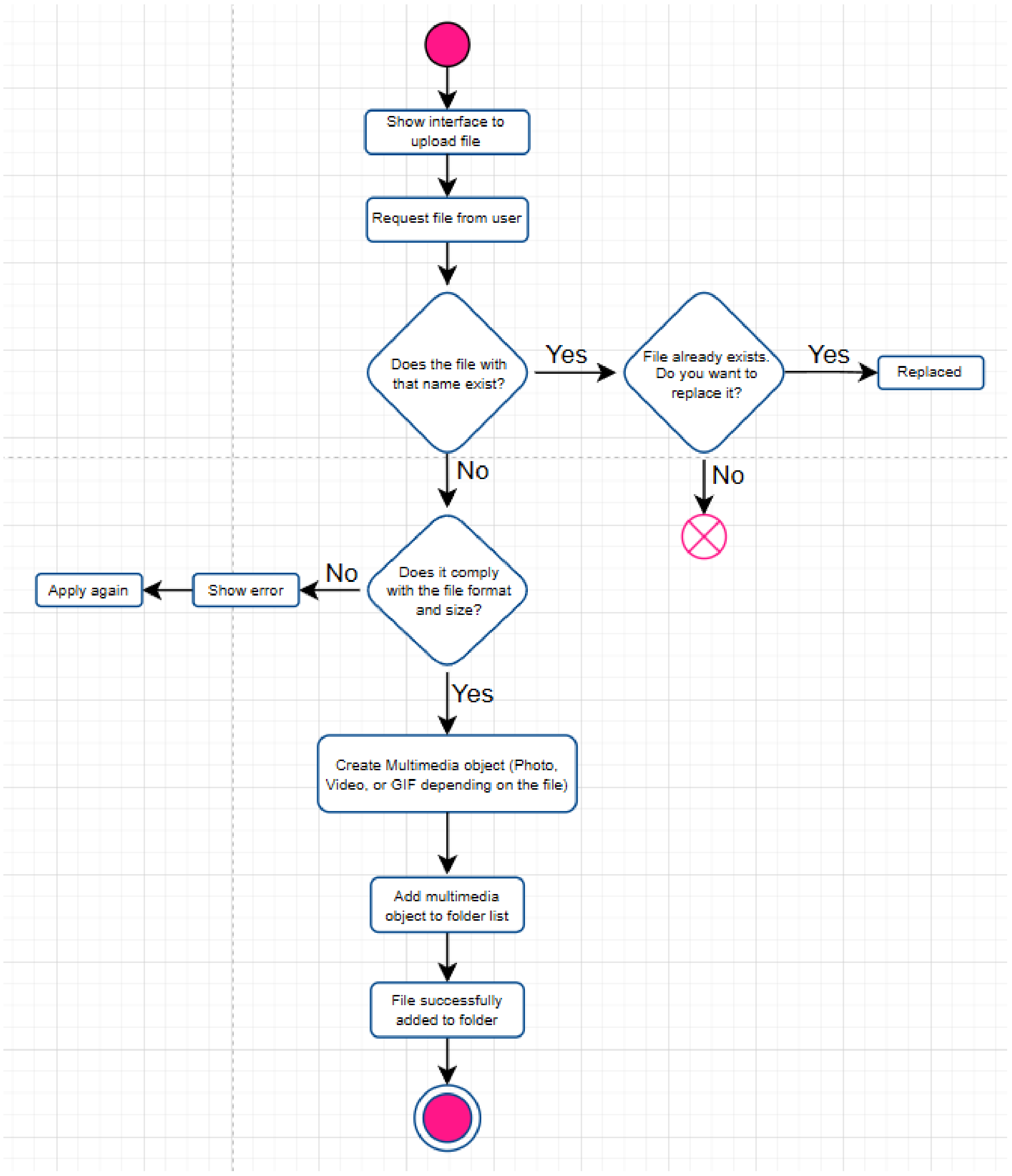

Show interface for creating folder
Ask the user for the name of the new folder
Is there already a folder with that name?
Yes
There is already a folder with that name.
No
Create MiniFolder object
Add folder to user's folder list
Folder successfully created

- After logging in, the user selects the option to create a new folder. The system, through the App class, calls the createMiniFolder(user, folderName) method. A new Folder object is created using the provided name, and it is added to the user’s list of folders. A confirmation message is shown to indicate that the folder was successfully created. The folder is instantiated within the App method, without exposing the internal details of how the Folder is initialized. Only the name is passed as a parameter, and the folder handles internally its association with the user.

2.2.1.7 Sequence Diagram (Upload Multimedia to the MiniFolder)

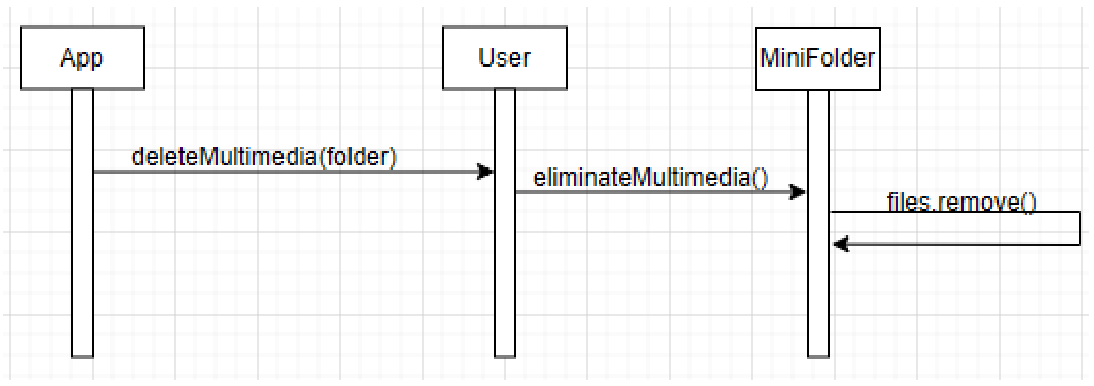


2.2.2.7 Activity diagrams (Upload Multimedia to the MiniFolder)

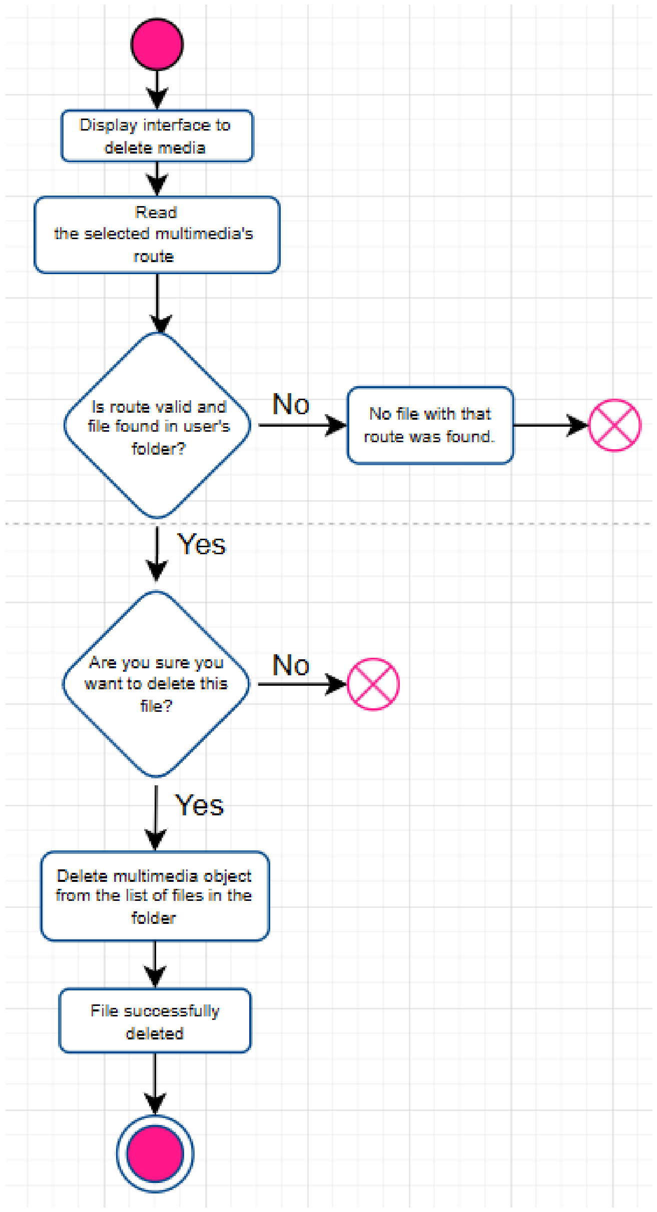

Show interface to upload file
Request file from user
Does the file with that name exist?
Yes
File already exists. Do you want to replace it?
Yes
Replaced
No
No
Does it comply with the file format and size?
No
Show error
Apply again
Yes
Create Multimedia object (Photo, Video, or GIF depending on the file)
Add multimedia object to folder list
File successfully added to folder

- The process begins when the upload interface is displayed, prompting the user to select a media file from their device. Once the file is received, the application checks if a file with the same name already exists in the corresponding MiniFolder. If it does, the user is asked whether they want to replace it. If the user declines, the process ends without changes; if they agree, the existing file is removed to proceed. If the file does not exist or replacement is authorized, the system validates the file's format and size. If the file is invalid, an error message is shown and a new file is requested. If it meets the requirements, a Multimedia object (Photo, Video, or GIF) is created using the file data and stored in the appropriate MiniFolder. The addition is done via the addMultimedia() method, which accepts any object that implements the Multimedia interface, ensuring all media types are handled uniformly. A confirmation message is displayed upon successful upload.

2.2.1.8 Sequence Diagram (Delete Multimedia from MiniFolder)

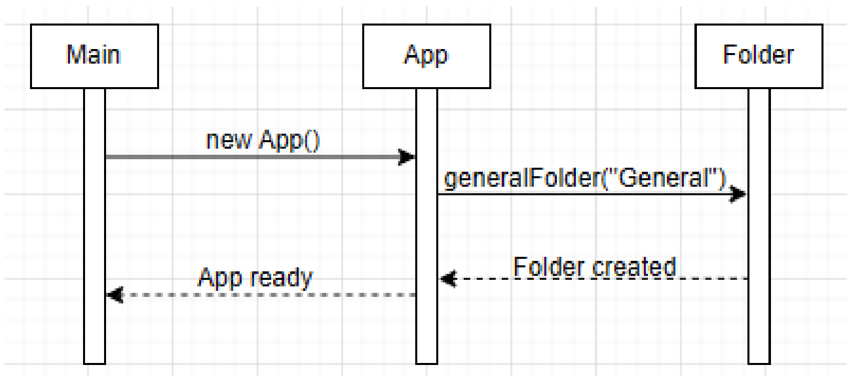


2.2.2.8 Activity diagrams (Delete Multimedia from MiniFolder)

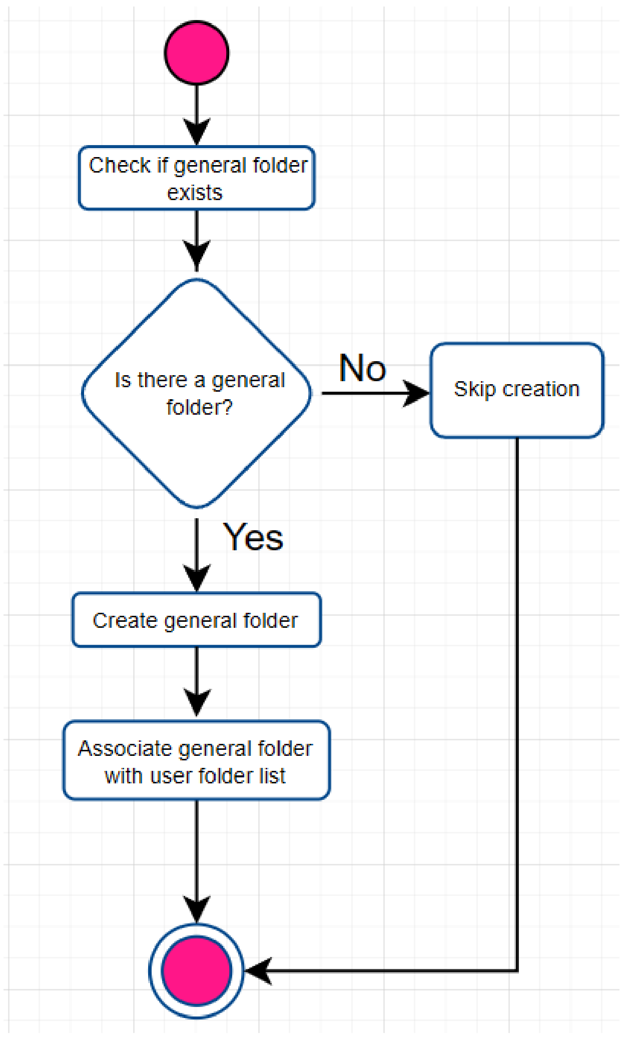

Display interface to delete media
Read the selected multimedia's route
Is route valid and file found in user's folder?
No
No file with that route was found.
Yes
Are you sure you want to delete this file?
No
Yes
Delete multimedia object from the list of files in the folder
File successfully deleted

- The process begins by displaying the file deletion interface to the user, who selects the file they wish to delete. The system reads the path of the selected file and searches the corresponding MiniFolder to check whether a file with that path exists. If no such file is found, an error message is displayed and the process ends. If the file is found, the user is asked to confirm the deletion. If the user declines, no changes are made. If the user confirms, the file is removed from the folder's file list using the removeMultimedia() method. A confirmation message is displayed to indicate successful deletion. The file list in the Folder class is private and can only be modified through this method. Since all file types (Photo, Gif, Video) implement the Multimedia interface, they can be deleted in the same way. If new media types are added in the future, deletion will continue to work seamlessly as long as they implement the Multimedia interface.

2.2.1.9 Sequence Diagram (Create general folder)



2.2.2.9 Activity diagrams (Create general folder)


Check if general folder exists
Is there a general folder?
No
Skip creation
Yes
Create general folder
Associate general folder with user folder list

- When the program starts (before any user logs in), an instance of the Folder class is created with the name "General" and initialized with an empty list of media files. This folder is stored as a shared attribute within the App class, making it accessible to all users for uploading or viewing files. A new instance of "General" is not created per user; it is a single, unique folder shared across the entire application. If additional special folders (such as "Favorites") are needed in the future, they can be added without modifying the existing implementation. The GeneralFolder is treated like any other Folder, ensuring compatibility with the system's existing logic.

2.4.1 User class:

OOP application:

- Encapsulates credentials as private attributes.
- Related to Folder, App, and Multimedia types
- Complies with SRP (Single Responsibility Principle): only handles user data and their actions.

Responsibilities: register, login, upload, delete and download multimedia.

Implemented methods:

- register(name, password)
- login(username, password)
- addMultimedia()
- eliminateMultimedia()
- saveMultimedia()

2.4.2 Photo Class

OOP implementation:

- Manage your data (name, size, route, date) in a controlled manner through public methods.
- Implements the methods defined by the Multimedia interface.
- It can be treated as a Multimedia type, allowing it to be handled alongside Video and GIF.

Responsibilities: store, retrieve and manage photo metadata

Implemented methods:

- getRoute()
- getName()
- getSize()
- getDate()

2.4.3 Video class

OOP application:

- Manage your data (name, size, route, date) in a controlled manner through public methods.

- Implements the methods defined by the Multimedia interface.
- It can be treated as a Multimedia type, allowing it to be handled alongside Photo and GIF.

Responsibilities: store, retrieve and manage video metadata.

Implemented methods:

- getRoute()
- getName()
- getSize()
- getDate()

2.4.4 Gif class

Responsibilities: store, retrieve and manage gif metadata.

Implemented methods:

- getRoute()
- getName()
- getSize()
- getDate()

2.4.5 Multimedia interface

OOP implementation:

- Manage your data (name, size, route, date) in a controlled manner through public methods.
- Implements the methods defined by the Multimedia interface.
- It can be treated as a Multimedia type, allowing it to be handled alongside Photo and Video.

Responsibilities: define a common interface and provide general methods for media files.

Implemented methods:

- getRoute()
- getName()
- getSize()
- getDate()

2.4.6 Folder class

OOP implementation:

- Internally manages your list of multimedia files and their names (folderName), controlling access through methods.
- It is related to the User class (each folder belongs to a user).
- It is exclusively responsible for managing folders and the files within them.

Responsibilities: store metadata, manage multimedia content, and coordinate folder creation for a specific user.

Implemented methods:

- generalFolder: (folderName)
- miniFolder: (User, folderName)
- userFolder: attribute
- getMultimedia Device: String
- getMultimediaGeneral: String

2.4.7 App class

OOP implementation:

- Manage system operations internally (upload, delete, download multimedia, and create folders).
- Communicates with User, Folder, and Multimedia to coordinate actions on files and folders.
- Its sole responsibility is to coordinate system functions and serve as a general access point.

Responsibilities: coordinate media operations and manage system-level folder creation.

Implemented methods:

- uploadMultimedia()
- deleteMultimedia()
- downloadMultimedia()
- createUserFolder attribute
- createMiniFolder: (User, folderName)
- create GeneralFolder: (User, folderName)

Element	Responsibility	Interaction with other classes	Key Methods
App	Manages general functions such as uploading, deleting, or downloading files.	Interacts with User, Folder, Photo, Video, Gif.	uploadPhoto(), deletePhoto(), downloadPhoto(), createGeneralFolder( )
User	Represents each user, stores personal information and folders.	Uses a list of Folder objects, works with Photo, Video, Gif.	addPhoto(), eliminatePhoto(), savePhoto(), login(), register()
Folder (Mini/General)	Contains multimedia files. Manages storage logic within a specific folder.	Related to Multimedia objects.	miniFolder(), generalFolder(), userFolder(), files
Multimedia (Superclass)	Base class for all file types. Defines general attributes and shared methods.	Extended by Photo, Video, Gif.	getRoute(), getName(), getSize(), getDate()
Photo / Video / Gif	Inherit from Multimedia and represent specific file types.	Are added to Folder file lists.	Inherit Multimedia methods

Step	UploadMultimedia (MiniFolder)	DeleteMultimedia (MiniFolder)
Start	The user selects a multimedia file (photo, gif, or video).	The system displays the delete file interface.
Folder selection	The system asks the user to select a previously created MiniFolder.	The user selects a file from the folder's file list.
Folder or file search	The app looks for the selected folder name in the user's folder list.	The system reads the selected file's route and looks for it in the folder's file list.
Validation	If the folder is found, the app calls the addMultimedia() method with the file object.	If the file is not found or the route is invalid, an error message is shown and the process ends.
Main action	The file is added to the internal files list of the MiniFolder.	If the file is found, the user is asked to confirm the deletion.
User confirmation	Not applicable (upload proceeds immediately).	If confirmed, the file is removed from the folder’s files list.
Final result	A success message is shown.	A success message is shown if deleted; otherwise, an error or cancel message appears.

3. Implementation Plan for OOP Concepts

3.1

- **Encapsulation**

In PinBerry, encapsulation is applied as a foundational principle to ensure data protection, logic integrity, and rule-driven interactions. Rather than relying on superficial encapsulation—where private fields are merely exposed through public getters and setters—the application enforces deep encapsulation, where all internal attributes and operations are accessed only through validated methods that reflect business logic.

For instance, the User class includes sensitive attributes such as username, password, and a reference to the user's personal folder. These fields are never exposed directly. Instead of allowing external access to raw credentials through insecure methods like getPassword(), operations like updatePassword(String oldPassword, String newPassword) are designed to verify the current password before applying changes. This ensures secure, controlled access and eliminates the possibility of unauthorized manipulation.

Likewise, the Folder class encapsulates a list of Multimedia elements (which may include Photo, Video, or Gif objects), along with metadata like the folder name. These elements cannot be directly accessed or modified. Instead, functions like addMedia(User user, Multimedia item) or removeMedia(User user, Multimedia item) ensure that only authorized users can modify content, enforcing ownership and access policies. Folders are categorized into three types: general folders, user folders (automatically created per user at registration), and custom mini-folders. This encapsulation of structural logic within the folder class ensures tight control over file management workflows.

The App class, which orchestrates all upload, download, delete, and folder creation operations, does not expose internal mappings or services directly. All state changes occur through explicit high-level functions that coordinate access between users, folders, and multimedia entities, enforcing validations such as token validity, media ownership, and user identity.

Through this rigorous application of encapsulation, the system maintains a secure boundary between state and behavior, eliminating unauthorized access paths, promoting clear contracts for object interaction, and ensuring that every modification is validated, traceable, and maintainable.

- **Polymorphism**

Polymorphism is meaningfully applied in the PinBerry architecture through the introduction of the Multimedia interface. This interface defines a unified contract for all media-related classes (Photo, Video, and Gif), each of which implements shared behaviors such as getRoute(), getName(), getSize(), and getDate(). This design allows the system to operate polymorphically when dealing with media items, particularly within the Folder class, which maintains an aggregated list of objects typed as Multimedia.

This use of polymorphism offers multiple advantages. Firstly, it enables uniform treatment of diverse media types, allowing the application to process, display, store, or delete any kind of media using the same high-level logic. Secondly, it allows the system to scale seamlessly—new types of media (e.g., Audio, LivePhoto) can be added by simply implementing the Multimedia interface, without modifying existing folder or app logic. This follows the Open/Closed Principle, ensuring the system is open for extension but closed for modification.

Moreover, this polymorphic design is key to enabling interface-driven aggregation. The Folder class does not need to know the concrete type of media it contains; it only interacts with them through the Multimedia interface. This decoupling simplifies testing, enhances flexibility, and supports future growth. The aggregation also reflects strong architectural clarity: media cannot exist outside a folder, and folders are intrinsically tied to users—this is reinforced through a well-defined aggregation hierarchy (User → Folder → Multimedia), which aligns with domain expectations.

In earlier iterations, polymorphism was avoided due to the limited scope of media types. However, with the evolving needs of PinBerry, particularly its support for rich media handling and user-customized content organization, polymorphism has been selectively and strategically introduced where it adds real value, without compromising system simplicity or clarity.

- **Abstraction**

Abstraction in PinBerry is implemented to reduce complexity and expose only the essential features required by each component. Rather than overloading classes with unnecessary technical details or exposing irrelevant internal mechanisms, abstraction ensures that every object interacts through clearly defined interfaces or minimal, task-specific methods. This reduces coupling and improves maintainability across the system.

The most evident use of abstraction is the introduction of the Multimedia interface. This abstraction captures the shared attributes and behaviors of different media types—such as Photo, Video, and Gif—without tying the system to any specific implementation. The interface defines methods like getRoute(), getName(), and getDate(), which are implemented uniformly across all concrete classes. This design allows client classes, such as Folder or

App, to work with any Multimedia object without needing to know its exact type or internal logic.

Another example of abstraction is present in the App class, which serves as the orchestration layer for high-level operations like uploading or deleting media. Rather than exposing the internal mechanisms for authentication, file access, or storage management, the App class delegates these operations to underlying encapsulated logic, acting as an abstract coordinator. This abstraction centralizes the application's use cases while decoupling domain logic from lower-level technical concerns.

Abstraction also supports security and clarity. For example, when a user deletes a photo, they interact only with a high-level method—deleteMedia(User user, Multimedia item)—which abstracts the internal verification processes (e.g., checking if the user owns the item, whether it exists in the folder, and if it can be removed from shared contexts like the general folder).

- **Decomposition with Inheritance**

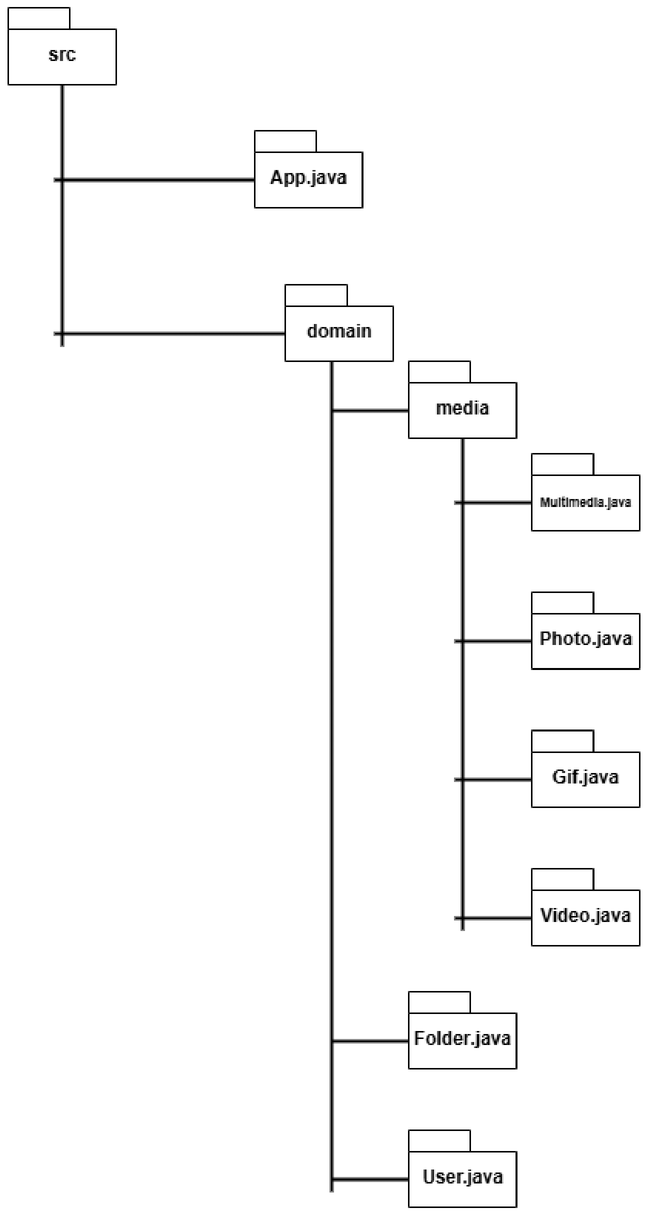
Inheritance in PinBerry is used in a focused and well-justified manner through the implementation of the Multimedia interface and its three concrete subclasses: Photo, Video, and Gif. Rather than introducing inheritance across unrelated service or domain layers—which would add unnecessary hierarchy and complexity—the application uses inheritance as a tool for decomposing a family of similar objects while preserving shared structure and behavior.

This design follows the Liskov Substitution Principle (LSP), a key component of SOLID. Any object of type Photo, Video, or Gif can be substituted wherever a Multimedia object is expected, without breaking the logic or contracts of the system. All these subclasses respect the same interface and return consistent information structures. This enables polymorphic behavior and provides a clear semantic contract: all media types share a route, name, size, and creation date, and are treated uniformly within folders and the app controller.

Decomposition through inheritance avoids code duplication and improves consistency. Without it, each media type would require redundant implementations of similar logic. Instead, shared behavior (e.g., returning metadata) can be centralized at the interface level and specialized only when necessary. If in the future certain media types require distinct functionality (e.g., Video with a play() method or Gif with animation controls), inheritance allows these extensions without disrupting other classes.

Additionally, this decomposition enhances clarity in architectural diagrams. The UML class diagram clearly illustrates a single parent interface (Multimedia) and three logically grouped child classes. This reflects the domain accurately and supports future extensibility: adding a new media format would not impact existing classes or require changes to Folder or App, as long as it implements Multimedia.

3.2 Estructure



SOLID-FOCUSED IMPLEMENTATION

The implementation of PinBerry is deeply rooted in the SOLID principles of object-oriented design, which serve as a foundational guide for building robust, maintainable, and scalable software. As the system evolved, each architectural and structural decision—from the definition of core entities to their interrelationships—was carefully evaluated to ensure adherence to these principles.

By integrating SOLID, the project avoids common pitfalls such as tight coupling, fragile dependencies, and code duplication. Instead, it fosters a modular ecosystem in which each class and interface fulfills a clearly defined responsibility, remains open to extension but closed to arbitrary modification, and respects substitution and separation boundaries. This approach not only improves code clarity and reusability but also ensures long-term maintainability and ease of future extension.

This section outlines how each of the SOLID principles—Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion—has been thoughtfully applied in the PinBerry application. Concrete examples, such as the introduction of the Multimedia interface and the redesign of folder-user-media relationships, demonstrate how SOLID informed both the theoretical design and the practical implementation of the system.

Together, these principles form the architectural backbone of PinBerry, ensuring that the platform is not only functional but also architecturally sound, scalable, and aligned with professional software development standards.

- **Justification of the Single Responsibility Principle (SRP) in PinBerry**

For instance, the User class is solely responsible for user account management, including operations such as registration, login, and the association with the multimedia objects owned by the user. This class does not handle image processing or contain any logic related to storage or display. Complementing this, the Photo class—along with Video and Gif—is solely focused on representing the essential attributes of a media file, such as its name, size, path, and date, and providing access to that information. Each media-related class is therefore responsible only for maintaining and communicating its internal state.

Similarly, the Folder class is responsible exclusively for organizing multimedia files. Its purpose is limited to containing Multimedia objects, managing their inclusion or removal, and maintaining a clear reference to their owner. It does not deal with authentication, rendering, or physical storage, ensuring that changes to storage policies, for example, do not impact this class.

Adhering to SRP in PinBerry brings significant advantages in terms of maintainability, testability, and scalability. Each module can be modified, replaced, or extended independently without risking unintended impacts on other parts of the system. Additionally, by reducing the complexity of individual classes, system understanding improves—not only during initial development but also for future integrations and expansions.

- **Justification of the Open/Closed Principle (OCP) in PinBerry**

In PinBerry, all media types—Photo, Video, and Gif—implement the Multimedia interface. This abstraction allows the system to interact with media content in a polymorphic and extensible way. The Multimedia interface defines the essential contract that all media classes must fulfill, including methods like getRoute(), getName(), getSize(), and getDate(). This design allows new media types to be introduced in the future—such as Audio, PDF, or

3DModel—by simply implementing this interface, without altering the existing logic that relies on Multimedia. Thus, the system remains closed to modification but open to new functionality.

For instance, if a new media type called Slideshow is added, the developer needs only to implement the Multimedia interface in the new class. All system components that operate on objects of type Multimedia—such as folders, galleries, or viewing modules—will handle the new media type seamlessly, without requiring code modifications in those components.

By adhering to OCP, PinBerry ensures that the core system is future-proof and adaptable. It encourages developers to build upon a stable foundation, promoting scalability without risking the integrity of the existing codebase. This architectural approach results in reduced regression bugs, improved maintainability, and enhanced capacity for growth—essential attributes for any long-term software project

- **Justification of the Liskov Substitution Principle (LSP) in PinBerry**.

In PinBerry, this principle is fully respected through the design of the Multimedia interface and its implementing classes: Photo, Video, and Gif. These classes all represent specific forms of media that share common attributes and behaviors, such as retrieving their file name, path, size, and date. By ensuring that each of these classes adheres to the interface's contract—without violating expectations in any way—the application guarantees full substitutability.

For example, when a Folder class contains a collection of Multimedia objects, it does not need to distinguish whether a particular instance is a Photo, a Gif, or a Video. It interacts with each media item strictly through the Multimedia interface, invoking methods such as getRoute() or getSize() confidently, without checking the object's specific type. This enables clean, polymorphic interaction and avoids the need for conditional logic or casting, which would otherwise violate LSP and introduce fragility into the codebase.

Moreover, the adherence to LSP ensures the application remains extensible. If in the future a new media type such as Slideshow or AudioClip is added and implements the Multimedia interface correctly, the rest of the system will continue to operate flawlessly. No change is required in classes like Folder, App, or the UI components that rely on Multimedia, as long as the new types fulfill the interface's behavioral contract..

- **Justification of the Interface Segregation Principle (ISP) in PinBerry**

In PinBerry, this principle is applied with precision through the introduction and careful design of the Multimedia interface. This interface defines only the essential behaviors required for media content within the application: retrieving the media's route, name, size, and date (getRoute(), getName(), getSize(), getDate()). These methods are minimal, relevant, and precisely aligned with the shared functionality expected from any type of media within the system—whether it be a Photo, Video, or Gif.

This design ensures that each implementing class is not burdened with behaviors it does not require. For example, Photo does not need to handle video-specific metadata or playback methods; it simply implements the interface in terms of the four necessary operations. The same applies to Gif and Video. Each concrete media class delivers exactly what the interface promises, without being forced to stub out or ignore irrelevant functionality. This keeps each class clean, focused, and highly cohesive.

Furthermore, from the client perspective—such as classes like Folder, App, or components of the user interface—interactions with media are streamlined. These clients only depend on the specific behaviors exposed by Multimedia, not on broader or unrelated operations that might exist in specialized subclasses. This modularity ensures that changes in one media class do not cascade across the system, improving maintainability and reducing the risk of regressions.

By adhering to ISP, the PinBerry application remains modular, extensible, and easy to understand. It avoids the trap of forcing classes or clients to handle unnecessary logic, leading to a cleaner, more adaptable architecture. This not only improves the developer experience but also prepares the system to evolve gracefully as new media types or services are introduced.

- **Justification of the Dependency Inversion Principle (DIP) in PinBerry**

In the context of the PinBerry application, DIP is effectively applied through the strategic use of interfaces and service abstraction, particularly in how high-level components (such as the App, Folder, and user interface logic) interact with low-level components.

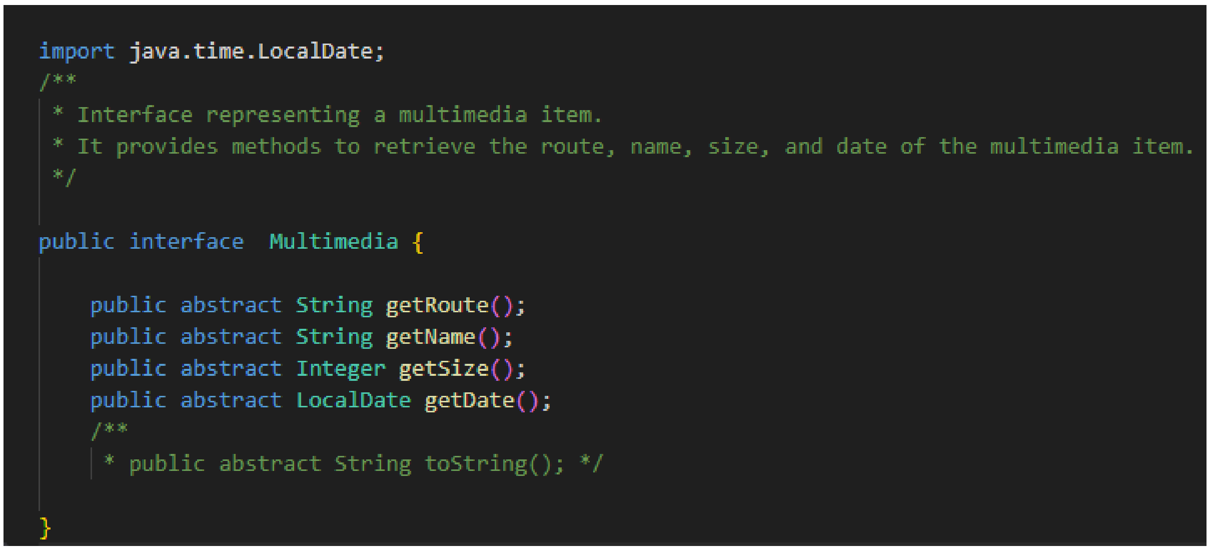
For example, consider the use of the Multimedia interface. High-level components like Folder and App work with Multimedia objects without knowing whether the object is a Photo, Gif, or Video. These classes depend only on the abstract interface Multimedia, and not on concrete implementations. The Folder class doesn't manage the construction, file handling, or type-specific behavior of the media; it simply trusts that each Multimedia object adheres to its defined contract. This inversion of dependency improves modularity, testability, and system resilience.

Moreover, by following DIP, PinBerry achieves a clear separation of concerns. High-level modules concentrate on user-facing logic, while the low-level details (data access, file encoding, storage mechanisms) are encapsulated and interchangeable. This abstraction not only allows for easier debugging and extension, but also significantly reduces the cognitive load when reasoning about or modifying specific parts of the system.

In conclusion, DIP is not just a theoretical rule in PinBerry—it is applied in a practical, strategic manner that enhances the scalability, flexibility, and long-term maintainability of the project. The deliberate design choices to depend on abstractions and isolate low-level operations ensure that the system can evolve without entangling components, making it robust and professionally sound.

4. Work in Progress Code

Interface Multimedia



```java
import java.time.LocalDate;
/**
 * Interface representing a multimedia item.
 * It provides methods to retrieve the route, name, size, and date of the multimedia item.
 */

public interface  Multimedia {

    public abstract String getRoute();
    public abstract String getName();
    public abstract Integer getSize();
    public abstract LocalDate getDate();
    /**
     * public abstract String toString(); */

}
```

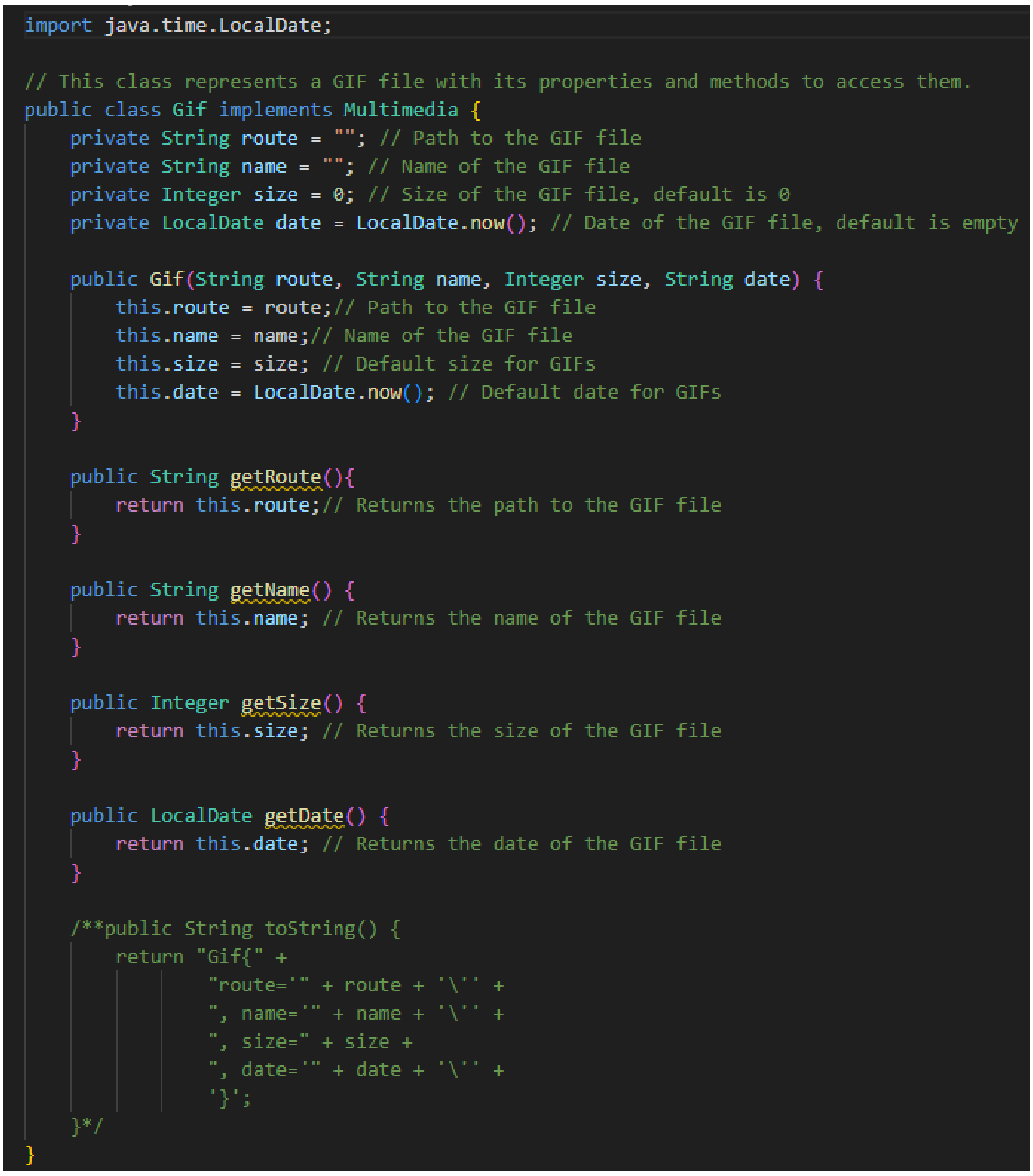
Improvement:

- Previously, each class (Photo, Gif, etc.) had its own logic, and if you wanted to treat them in a unified way (for example, displaying all photos, videos, and gifs in a single folder), you couldn't do so without duplicating code.
- Now, with the interface, any file type must implement the methods: getRoute(), getName(), getSize(), and getDate().

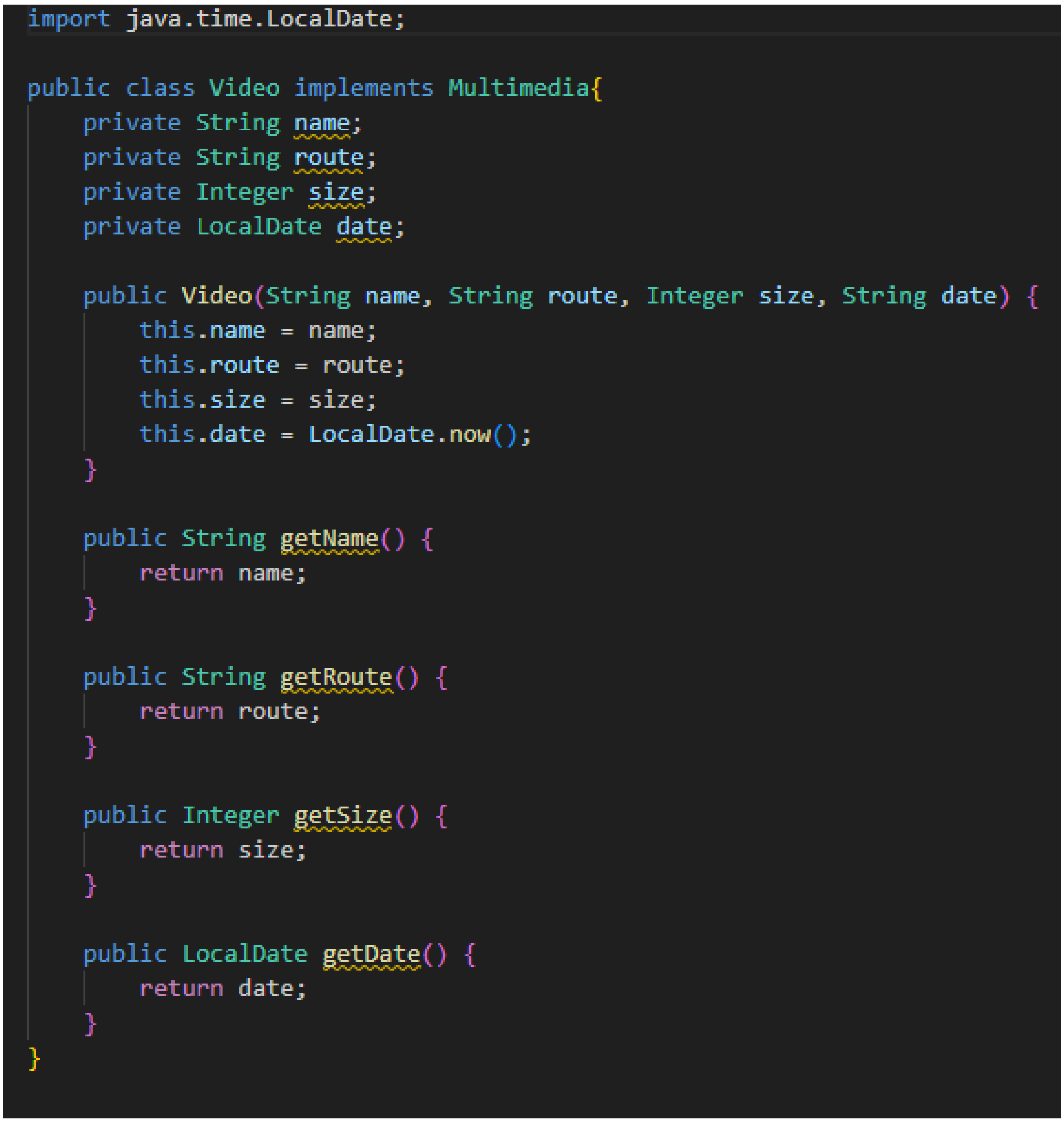
2. SOLID principles applied:

- I - Interface Segregation Principle (ISP): The interface only requires the implementation of essential methods, no more and no less. It does not overload the child classes.
- L - Liskov Substitution Principle (LSP): Any class that implements Multimedia can replace another without altering the expected behavior of the system.

Gif class



```java
import java.time.LocalDate;

// This class represents a GIF file with its properties and methods to access them.
public class Gif implements Multimedia {
    private String route = ""; // Path to the GIF file
    private String name = ""; // Name of the GIF file
    private Integer size = 0; // Size of the GIF file, default is 0
    private LocalDate date = LocalDate.now(); // Date of the GIF file, default is empty

    public Gif(String route, String name, Integer size, String date) {
        this.route = route;// Path to the GIF file
        this.name = name;// Name of the GIF file
        this.size = size; // Default size for GIFs
        this.date = LocalDate.now(); // Default date for GIFs
    }

    public String getRoute(){
        return this.route;// Returns the path to the GIF file
    }

    public String getName() {
        return this.name; // Returns the name of the GIF file
    }

    public Integer getSize() {
        return this.size; // Returns the size of the GIF file
    }

    public LocalDate getDate() {
        return this.date; // Returns the date of the GIF file
    }

    /**public String toString() {
        return "Gif{" +
                "route='" + route + '\'' +
                ", name='" + name + '\'' +
                ", size=" + size +
                ", date='" + date + '\'' +
                '}';
    }*/
}
```



```java
import java.time.LocalDate;

public class Video implements Multimedia{
    private String name;
    private String route;
    private Integer size;
    private LocalDate date;

    public Video(String name, String route, Integer size, String date) {
        this.name = name;
        this.route = route;
        this.size = size;
        this.date = LocalDate.now();
    }

    public String getName() {
        return name;
    }

    public String getRoute() {
        return route;
    }

    public Integer getSize() {
        return size;
    }

    public LocalDate getDate() {
        return date;
    }
}
```

1. Improvement:

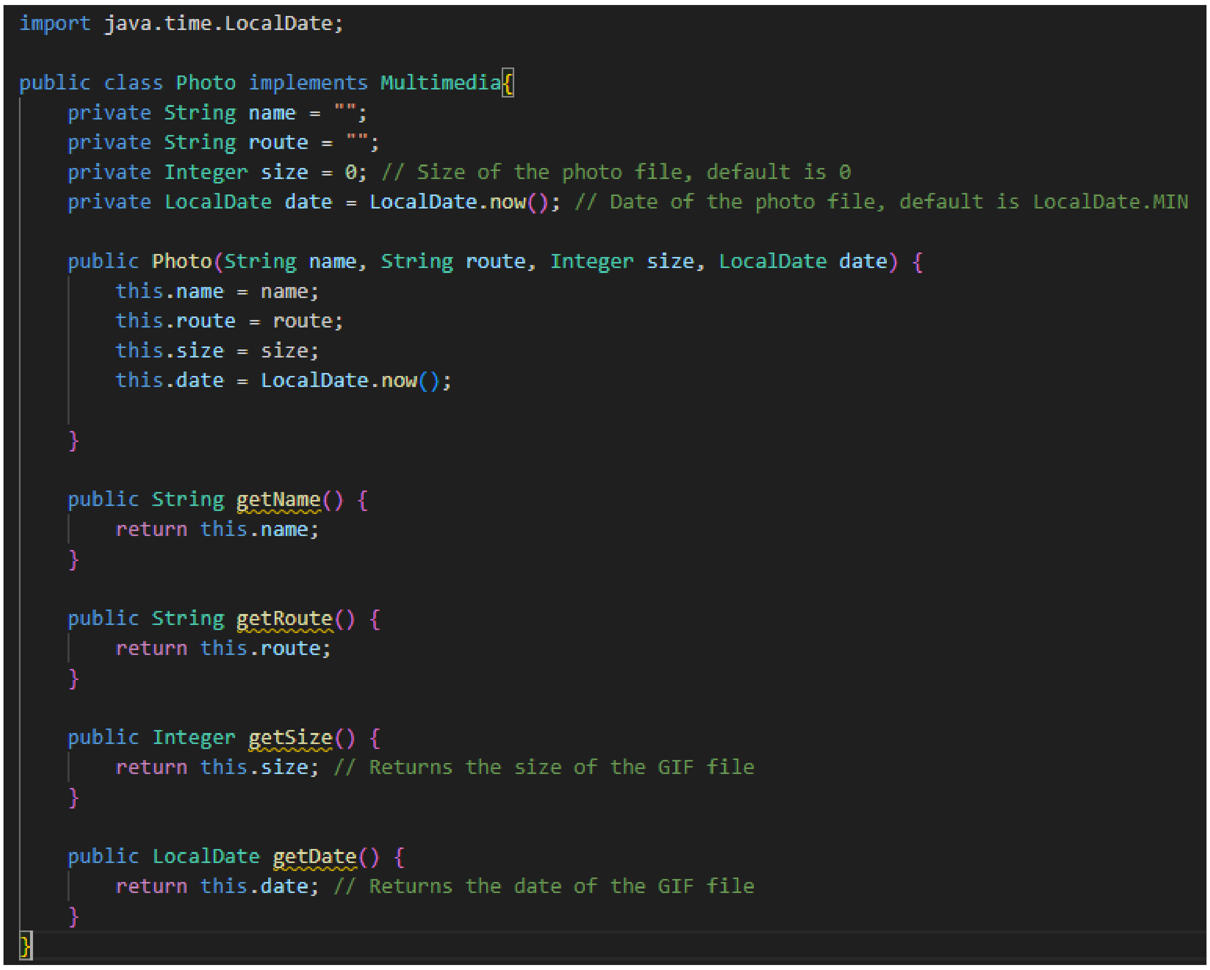
- These classes did not exist before. Now they all implement the Multimedia interface with a uniform structure.

2. Relationship with SOLID:

- S - SRP: Each class exclusively represents a type of multimedia with its basic properties.

- L - LSP / I - ISP: All can be used as Multimedia, fulfilling substitution and interface segregation.
- D - DIP: Programming is done against the interface, not the specific implementation.

Photo Class



```java
import java.time.LocalDate;

public class Photo implements Multimedia{
    private String name = "";
    private String route = "";
    private Integer size = 0; // Size of the photo file, default is 0
    private LocalDate date = LocalDate.now(); // Date of the photo file, default is LocalDate.MIN

    public Photo(String name, String route, Integer size, LocalDate date) {
        this.name = name;
        this.route = route;
        this.size = size;
        this.date = LocalDate.now();

    }

    public String getName() {
        return this.name;
    }

    public String getRoute() {
        return this.route;
    }

    public Integer getSize() {
        return this.size; // Returns the size of the GIF file
    }

    public LocalDate getDate() {
        return this.date; // Returns the date of the GIF file
    }
}
```

1. Improvement:

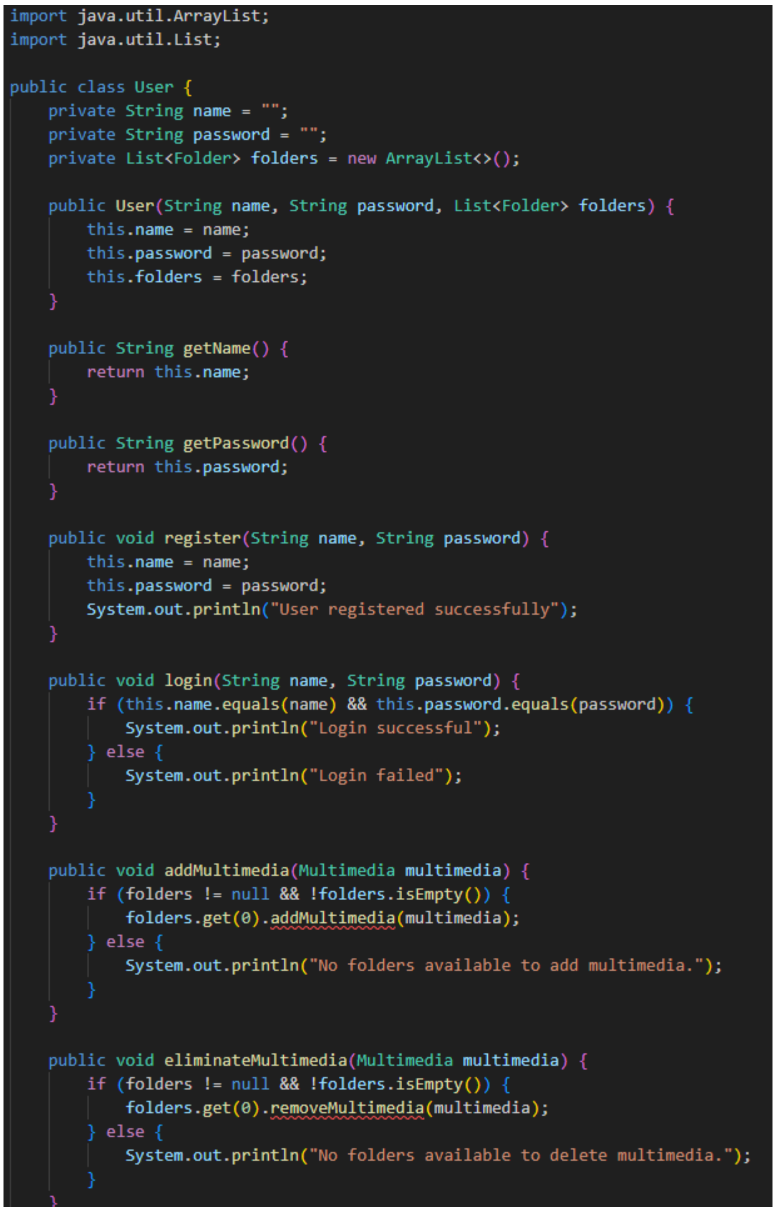
- In the new version, Photo implements an interface (Multimedia), and specific attributes such as isDeleted, height, and width are removed.
- It is standardized with other file types (Video, Gif) using the same methods (getName(), getRoute(), etc.).
- Use of LocalDate to handle dates more robustly.

2. Relationship with SOLID:

- S - Single Responsibility Principle (SRP): The class now only represents the basic data of a multimedia image, without state responsibilities such as logical deletion.
- L - Liskov Substitution Principle (LSP): By implementing Multimedia, it is guaranteed that a Photo, Video, or Gif can be used where a multimedia object is expected, without breaking functionality.
- I - Interface Segregation Principle (ISP): This applies when clearly defining what any type of multimedia file must have through the interface.

3. Evidence:

- Before: upload(), delete(), isDeleted() → mixed responsibilities.
- Now: Only simple getters and a clean constructor.



```java
import java.util.ArrayList;
import java.util.List;

public class User {
    private String name = "";
    private String password = "";
    private List<Folder> folders = new ArrayList<>();

    public User(String name, String password, List<Folder> folders) {
        this.name = name;
        this.password = password;
        this.folders = folders;
    }

    public String getName() {
        return this.name;
    }

    public String getPassword() {
        return this.password;
    }

    public void register(String name, String password) {
        this.name = name;
        this.password = password;
        System.out.println("User registered successfully");
    }

    public void login(String name, String password) {
        if (this.name.equals(name) && this.password.equals(password)) {
            System.out.println("Login successful");
        } else {
            System.out.println("Login failed");
        }
    }

    public void addMultimedia(Multimedia multimedia) {
        if (folders != null && !folders.isEmpty()) {
            folders.get(0).addMultimedia(multimedia);
        } else {
            System.out.println("No folders available to add multimedia.");
        }
    }

    public void eliminateMultimedia(Multimedia multimedia) {
        if (folders != null && !folders.isEmpty()) {
            folders.get(0).removeMultimedia(multimedia);
        } else {
            System.out.println("No folders available to delete multimedia.");
        }
```

1. Improvement:

- In the new version, the user has separate methods for register() and login().
- Unnecessary dependencies such as Photo lists are removed; it now handles Folder and Multimedia in general.

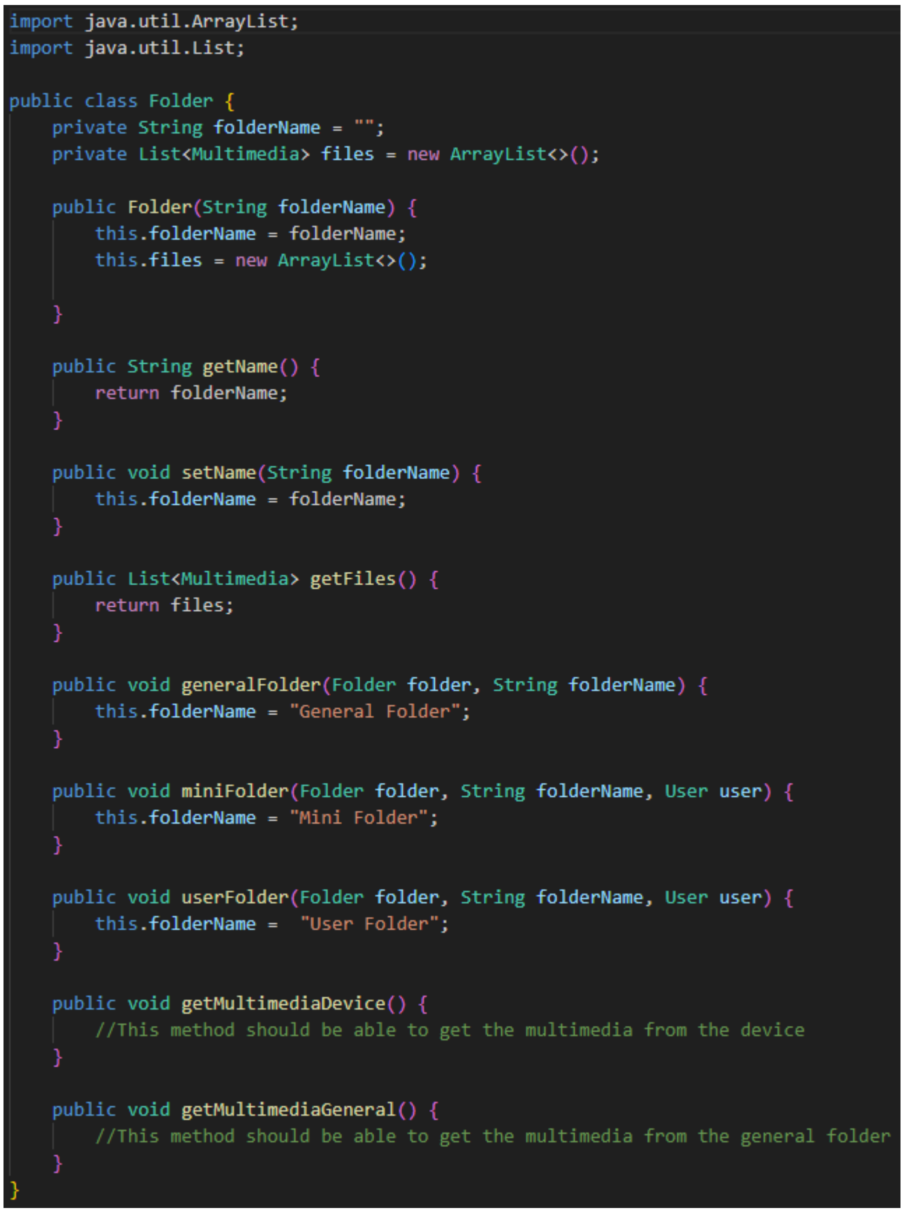
2. Relationship with SOLID:

S - SRP: The user is now only responsible for registering, authenticating, and managing their files, not for session control or specific list management.

- O - Open/Closed Principle (OCP): Uses a Folder list, which allows functionality to be extended without modifying the code base (for example, folders of different types can be added).
- D - Dependency Inversion Principle (DIP): Uses Multimedia abstraction, not concrete classes (such as Photo), promoting flexibility.

3. Evidence:

- Before: login(password) in User was used by AuthenticationService
- Now: User.login() is only responsible for showing whether it was successful



```java
import java.util.ArrayList;
import java.util.List;

public class Folder {
    private String folderName = "";
    private List<Multimedia> files = new ArrayList<>();

    public Folder(String folderName) {
        this.folderName = folderName;
        this.files = new ArrayList<>();

    }

    public String getName() {
        return folderName;
    }

    public void setName(String folderName) {
        this.folderName = folderName;
    }

    public List<Multimedia> getFiles() {
        return files;
    }

    public void generalFolder(Folder folder, String folderName) {
        this.folderName = "General Folder";
    }

    public void miniFolder(Folder folder, String folderName, User user) {
        this.folderName = "Mini Folder";
    }

    public void userFolder(Folder folder, String folderName, User user) {
        this.folderName =  "User Folder";
    }

    public void getMultimediaDevice() {
        //This method should be able to get the multimedia from the device
    }

    public void getMultimediaGeneral() {
        //This method should be able to get the multimedia from the general folder
    }
}
```

1. Improvement:

- In the old version, Folder had an unclear structure and some methods duplicated functions that were already in User.
- In the new version, Folder directly manages a list of Multimedia objects, which allows you to store Photo, Video, or Gif without changing the code.

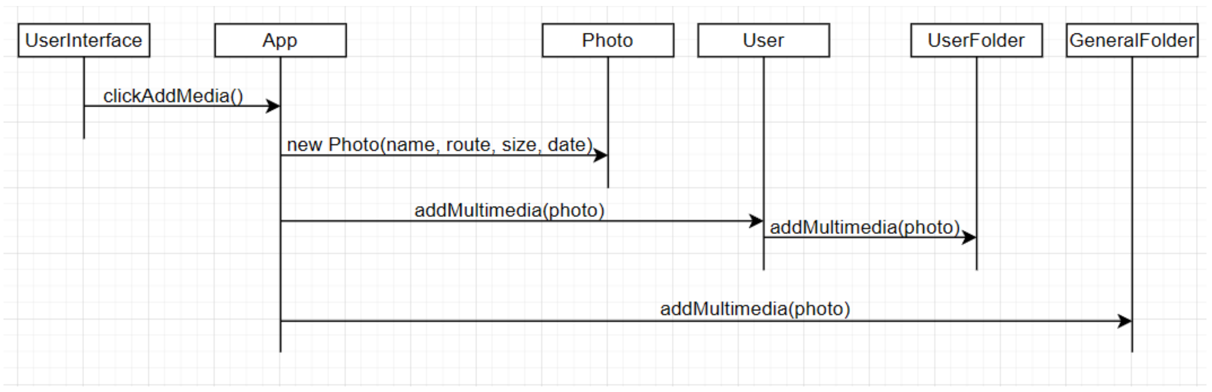
- Consistent operations are applied: addMultimedia(), removeMultimedia(), saveMultimedia(), and their respective getters and setters.

2. SOLID principles applied:

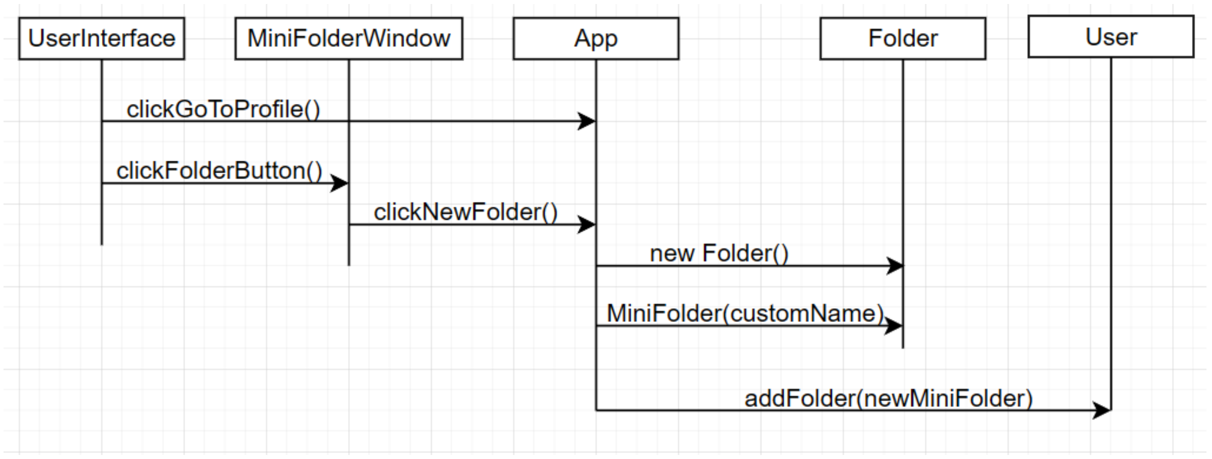
- S - Single Responsibility Principle: The Folder class is now only responsible for storing, deleting, and retrieving multimedia files. It does not mix functions such as login, validations, or console printing.
- O - Open/Closed Principle: The class is open for extension (you can add new methods or multimedia types), but closed for modification (you don't need to change the base code).
- D - Dependency Inversion Principle: It works with the Multimedia interface, not with concrete classes, which improves flexibility and decoupling.**5.**

DOCUMENTATION AND ARTIFACT SUBMISSIONS

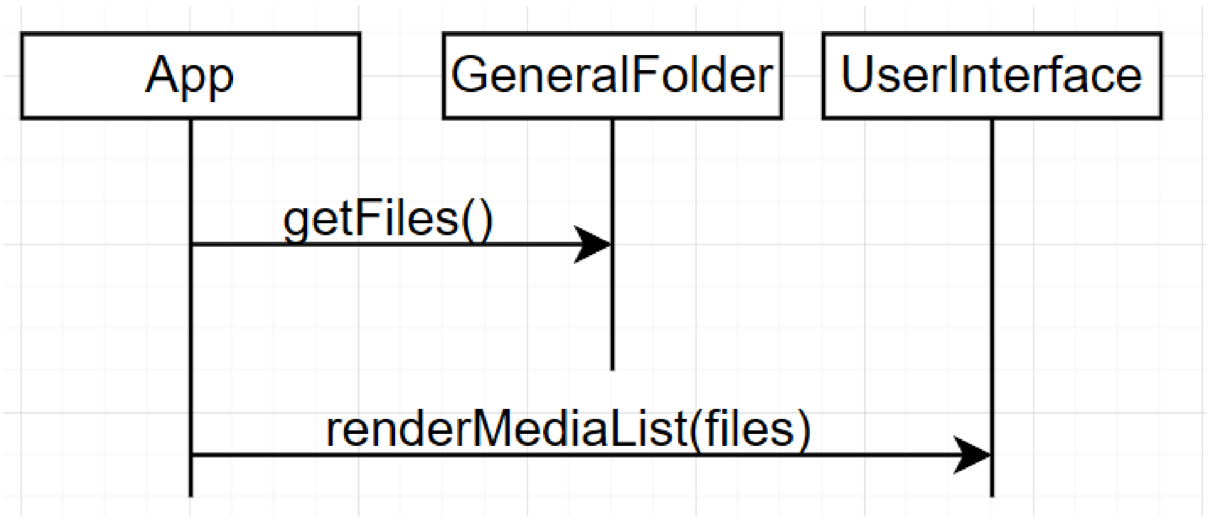
- UML Diagram: Communication Between Layers in the Final Solution
  - **Upload Photo**



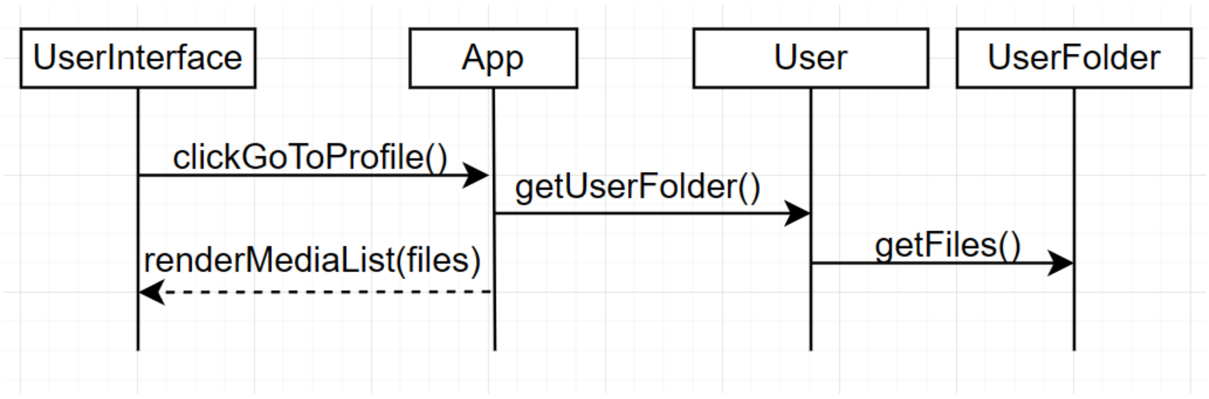
  - **Create Minifolder**



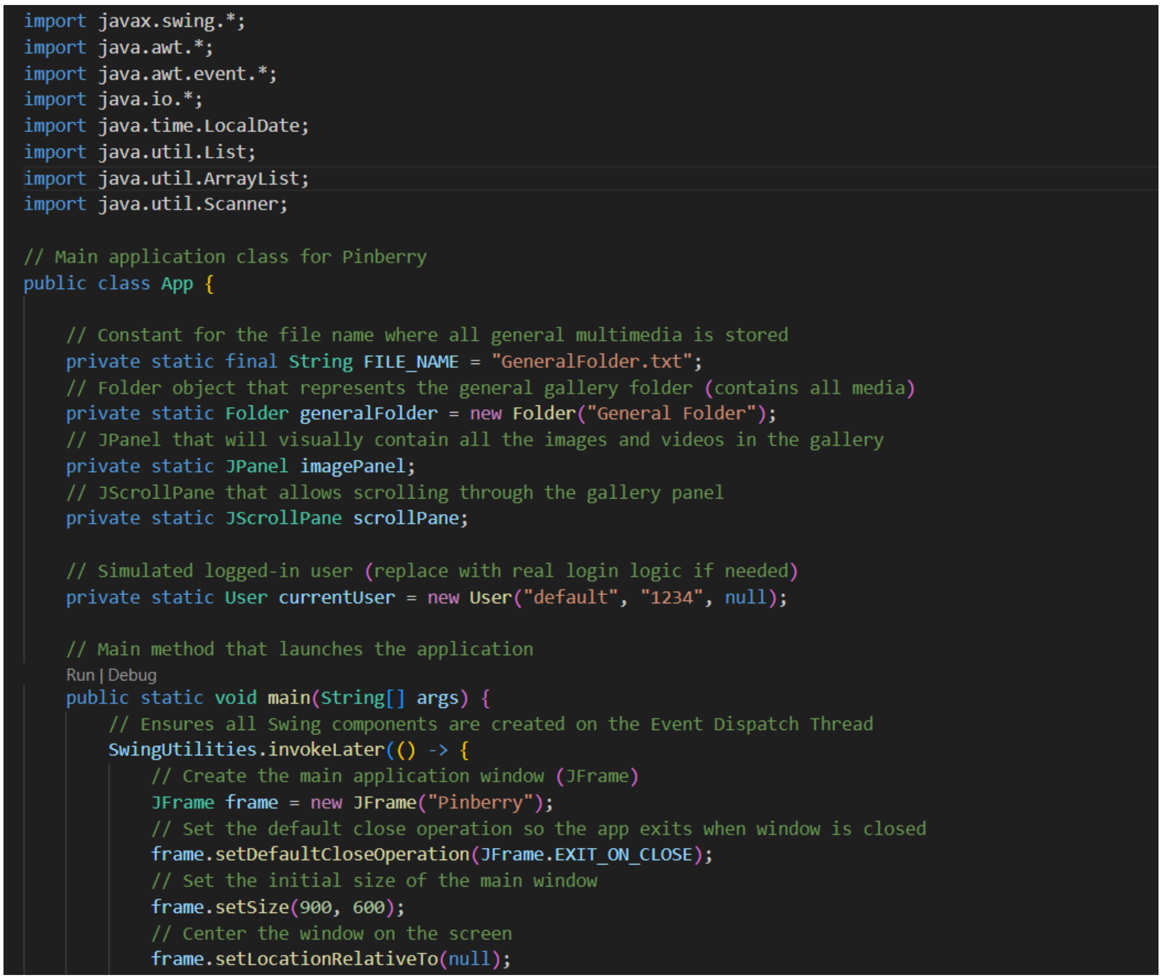
  - **Show General Folder**



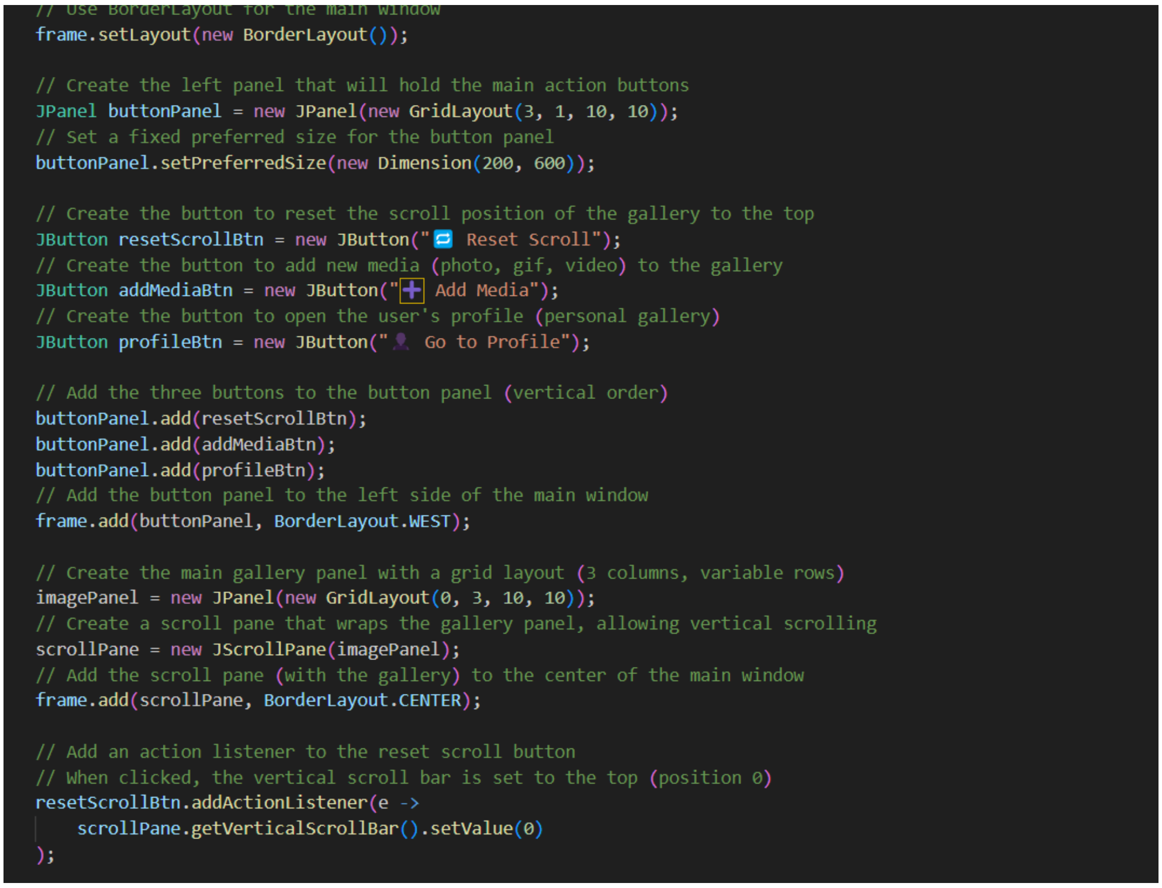
- **Go to Profile**



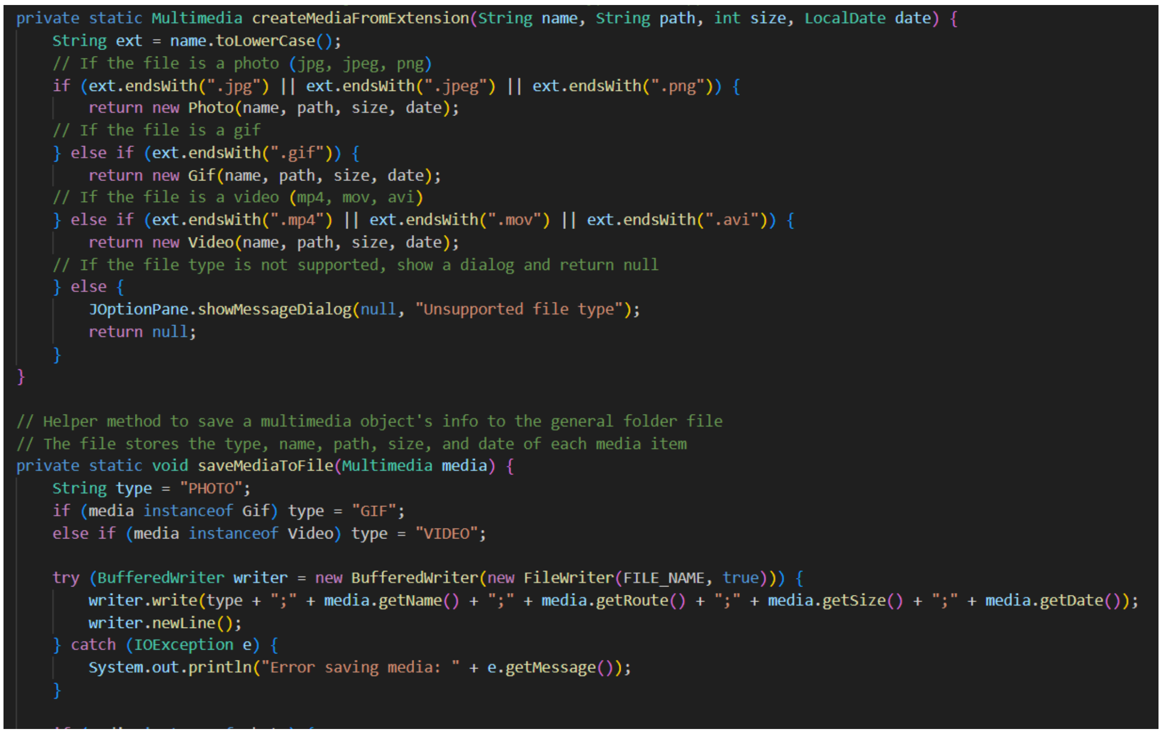
Code Samples



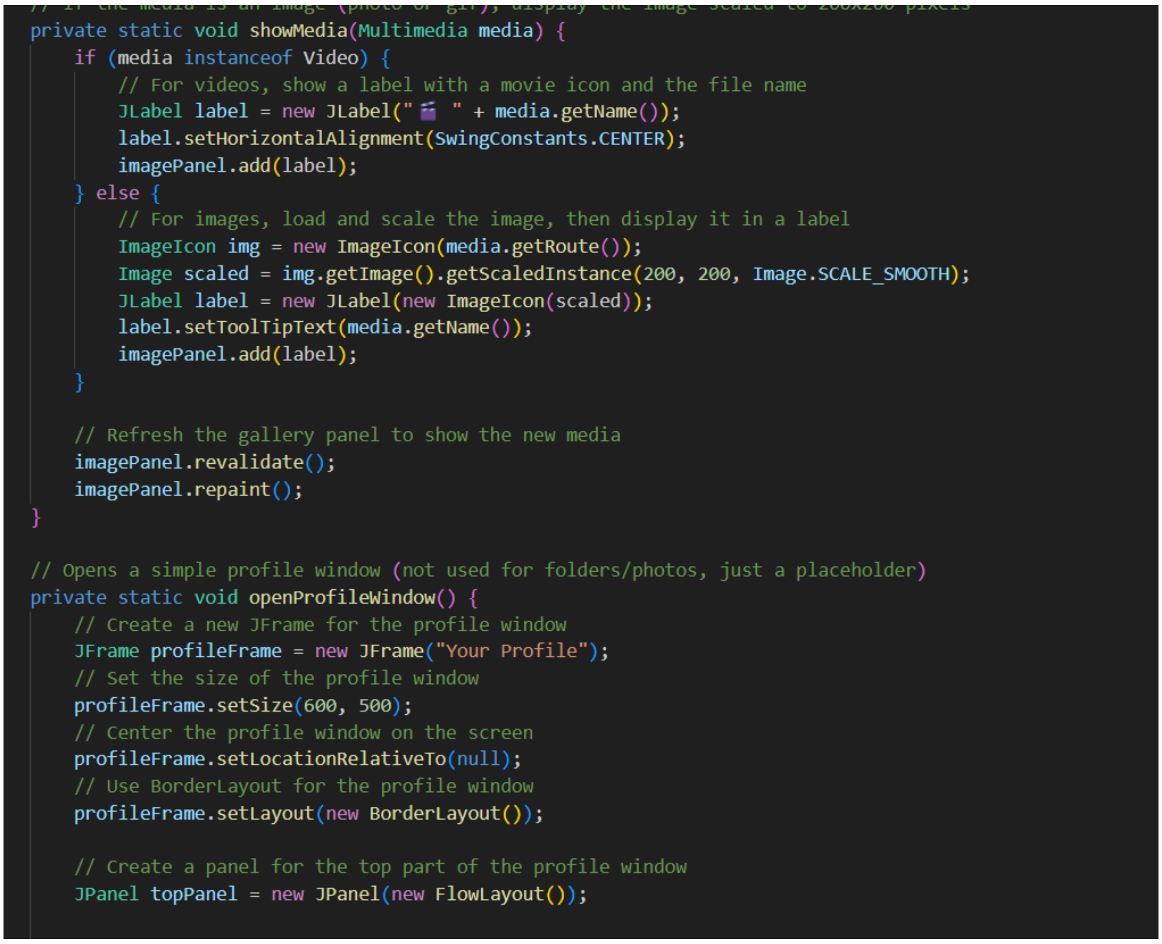
```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.time.LocalDate;
import java.util.List;
import java.util.ArrayList;
import java.util.Scanner;

// Main application class for Pinberry
public class App {

    // Constant for the file name where all general multimedia is stored
    private static final String FILE_NAME = "GeneralFolder.txt";
    // Folder object that represents the general gallery folder (contains all media)
    private static Folder generalFolder = new Folder("General Folder");
    // JPanel that will visually contain all the images and videos in the gallery
    private static JPanel imagePanel;
    // JScrollPane that allows scrolling through the gallery panel
    private static JScrollPane scrollPane;

    // Simulated logged-in user (replace with real login logic if needed)
    private static User currentUser = new User("default", "1234", null);

    // Main method that launches the application
    Run | Debug
    public static void main(String[] args) {
        // Ensures all Swing components are created on the Event Dispatch Thread
        SwingUtilities.invokeLater(() -> {
            // Create the main application window (JFrame)
            JFrame frame = new JFrame("Pinberry");
            // Set the default close operation so the app exits when window is closed
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            // Set the initial size of the main window
            frame.setSize(900, 600);
            // Center the window on the screen
            frame.setLocationRelativeTo(null);
```



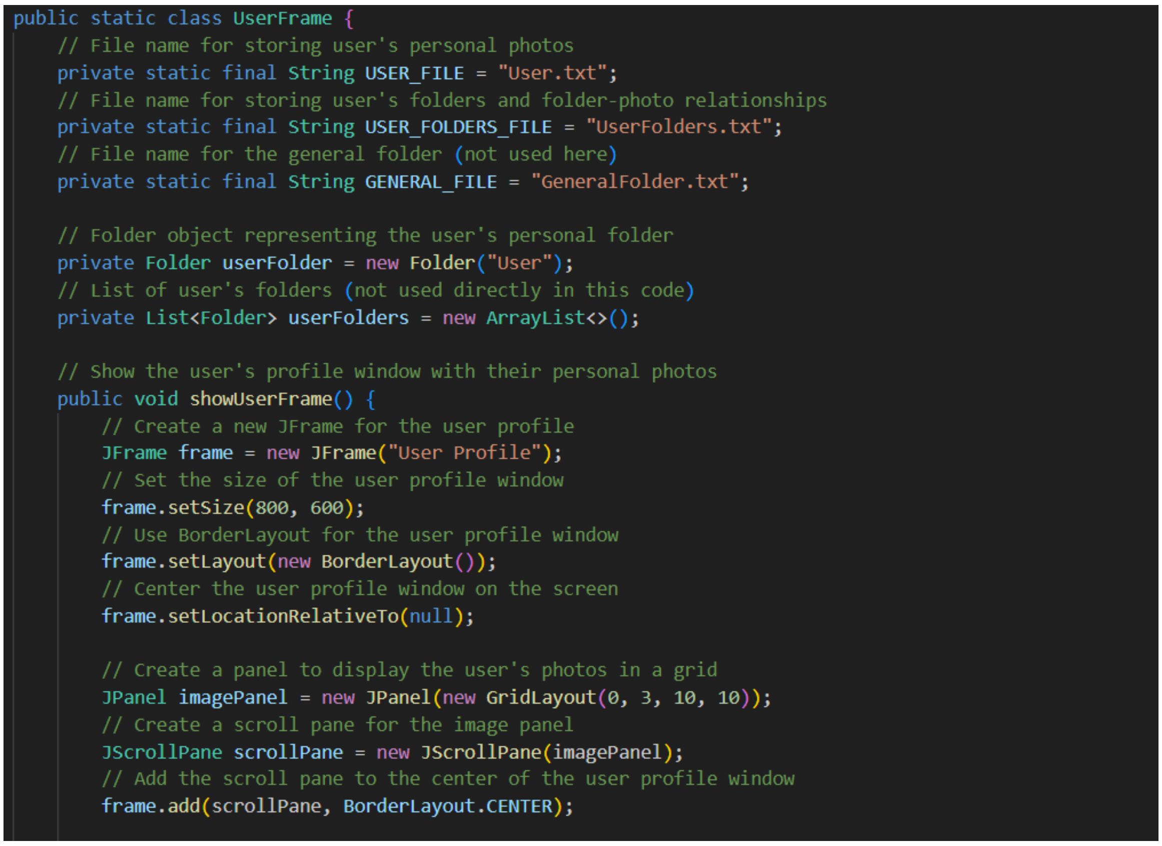
```java
    // Use BorderLayout for the main window
    frame.setLayout(new BorderLayout());

    // Create the left panel that will hold the main action buttons
    JPanel buttonPanel = new JPanel(new GridLayout(3, 1, 10, 10));
    // Set a fixed preferred size for the button panel
    buttonPanel.setPreferredSize(new Dimension(200, 600));

    // Create the button to reset the scroll position of the gallery to the top
    JButton resetScrollBtn = new JButton("🔄 Reset Scroll");
    // Create the button to add new media (photo, gif, video) to the gallery
    JButton addMediaBtn = new JButton("➕ Add Media");
    // Create the button to open the user's profile (personal gallery)
    JButton profileBtn = new JButton("👤 Go to Profile");

    // Add the three buttons to the button panel (vertical order)
    buttonPanel.add(resetScrollBtn);
    buttonPanel.add(addMediaBtn);
    buttonPanel.add(profileBtn);
    // Add the button panel to the left side of the main window
    frame.add(buttonPanel, BorderLayout.WEST);

    // Create the main gallery panel with a grid layout (3 columns, variable rows)
    imagePanel = new JPanel(new GridLayout(0, 3, 10, 10));
    // Create a scroll pane that wraps the gallery panel, allowing vertical scrolling
    scrollPane = new JScrollPane(imagePanel);
    // Add the scroll pane (with the gallery) to the center of the main window
    frame.add(scrollPane, BorderLayout.CENTER);

    // Add an action listener to the reset scroll button
    // When clicked, the vertical scroll bar is set to the top (position 0)
    resetScrollBtn.addActionListener(e ->
        scrollPane.getVerticalScrollBar().setValue(0)
    );
```



```java
private static Multimedia createMediaFromExtension(String name, String path, int size, LocalDate date) {
    String ext = name.toLowerCase();
    // If the file is a photo (jpg, jpeg, png)
    if (ext.endsWith(".jpg") || ext.endsWith(".jpeg") || ext.endsWith(".png")) {
        return new Photo(name, path, size, date);
    // If the file is a gif
    } else if (ext.endsWith(".gif")) {
        return new Gif(name, path, size, date);
    // If the file is a video (mp4, mov, avi)
    } else if (ext.endsWith(".mp4") || ext.endsWith(".mov") || ext.endsWith(".avi")) {
        return new Video(name, path, size, date);
    // If the file type is not supported, show a dialog and return null
    } else {
        JOptionPane.showMessageDialog(null, "Unsupported file type");
        return null;
    }
}

// Helper method to save a multimedia object's info to the general folder file
// The file stores the type, name, path, size, and date of each media item
private static void saveMediaToFile(Multimedia media) {
    String type = "PHOTO";
    if (media instanceof Gif) type = "GIF";
    else if (media instanceof Video) type = "VIDEO";

    try (BufferedWriter writer = new BufferedWriter(new FileWriter(FILE_NAME, true))) {
        writer.write(type + ";" + media.getName() + ";" + media.getRoute() + ";" + media.getSize() + ";" + media.getDate());
        writer.newLine();
    } catch (IOException e) {
        System.out.println("Error saving media: " + e.getMessage());
    }
```



```java
// If the media is an image (photo or gif), display the image scaled to 200x200 pixels
private static void showMedia(Multimedia media) {
    if (media instanceof Video) {
        // For videos, show a label with a movie icon and the file name
        JLabel label = new JLabel("🎬 " + media.getName());
        label.setHorizontalAlignment(SwingConstants.CENTER);
        imagePanel.add(label);
    } else {
        // For images, load and scale the image, then display it in a label
        ImageIcon img = new ImageIcon(media.getRoute());
        Image scaled = img.getImage().getScaledInstance(200, 200, Image.SCALE_SMOOTH);
        JLabel label = new JLabel(new ImageIcon(scaled));
        label.setToolTipText(media.getName());
        imagePanel.add(label);
    }

    // Refresh the gallery panel to show the new media
    imagePanel.revalidate();
    imagePanel.repaint();
}

// Opens a simple profile window (not used for folders/photos, just a placeholder)
private static void openProfileWindow() {
    // Create a new JFrame for the profile window
    JFrame profileFrame = new JFrame("Your Profile");
    // Set the size of the profile window
    profileFrame.setSize(600, 500);
    // Center the profile window on the screen
    profileFrame.setLocationRelativeTo(null);
    // Use BorderLayout for the profile window
    profileFrame.setLayout(new BorderLayout());

    // Create a panel for the top part of the profile window
    JPanel topPanel = new JPanel(new FlowLayout());
```



```java
public static class UserFrame {
    // File name for storing user's personal photos
    private static final String USER_FILE = "User.txt";
    // File name for storing user's folders and folder-photo relationships
    private static final String USER_FOLDERS_FILE = "UserFolders.txt";
    // File name for the general folder (not used here)
    private static final String GENERAL_FILE = "GeneralFolder.txt";

    // Folder object representing the user's personal folder
    private Folder userFolder = new Folder("User");
    // List of user's folders (not used directly in this code)
    private List<Folder> userFolders = new ArrayList<>();

    // Show the user's profile window with their personal photos
    public void showUserFrame() {
        // Create a new JFrame for the user profile
        JFrame frame = new JFrame("User Profile");
        // Set the size of the user profile window
        frame.setSize(800, 600);
        // Use BorderLayout for the user profile window
        frame.setLayout(new BorderLayout());
        // Center the user profile window on the screen
        frame.setLocationRelativeTo(null);

        // Create a panel to display the user's photos in a grid
        JPanel imagePanel = new JPanel(new GridLayout(0, 3, 10, 10));
        // Create a scroll pane for the image panel
        JScrollPane scrollPane = new JScrollPane(imagePanel);
        // Add the scroll pane to the center of the user profile window
        frame.add(scrollPane, BorderLayout.CENTER);
```

BIBLIOGRAFÍA

- ElJacobo. (2024, mayo 30). *CÓMO Documentar con Markdown USANDO VSCode | Windows 10* [Video]. YouTube. https://youtu.be/GW-PGABYMP8?si=T3L3Ah31MDNWnU3f

- MoureDev by Brais Moure. (2023, febrero 16). *Curso de GIT y GITHUB desde CERO para PRINCIPIANTES* [Video]. YouTube. https://youtu.be/3GymExBkKjE?si=yXa4Eypzq5998qH1

- ChatGPT. (2025). *Asistencia sobre la vinculación entre Git y GitHub* [Respuesta generada por inteligencia artificial]. OpenAI. https://chat.openai.com/

- Lucid Software Español. (2019, febrero 4). Tutorial - Diagrama de Clases UML[Video]. YouTube. https://youtu.be/Z0yLerU0g-Q?si=XAJgsszGMItbSM0N
- S/f). Stackoverflow.com. Recuperado el 16 de junio de 2025, de https://es.stackoverflow.com/questions/588967/c%C3%B3mo-hacer-render-recursivo-para-crear-una-estructura-de-carpetas
- https://www.youtube.com/watch?v=5G2XM1nlX5Q&pp=ygUdamF2YSBzd2luZyB2aXN1YWwgc3R1ZGlvIGNvZGU%3D
- https://www.youtube.com/watch?v=1vVJPzVzaK8&list=PL3bGLnkkGnuV699lP_f9DvxyK5lMFpq6U

•