

Quackstagram report.

Object Oriented Modeling

Group 10

March 2025

Contents

1	Project Management	2
1.1	Team Members	2
1.2	Overview of who did what	2
2	Introduction	2
3	Requirements Analysis	2
3.1	Functional requirements	2
3.2	Non functional Requirements	4
4	Use Case Descriptions	4
4.1	User Registration	4
4.2	User Login	5
4.3	Users Post	5
4.4	Liking of Post	5
4.5	Error Handling and Edge Cases	6
4.6	Edit Profile	6
5	UML diagrams	8
5.1	Class Diagrams	8
5.1.1	Given source code	8
5.1.2	Refactored code	10
6	Documentation of Refactoring	11
6.1	Refactoring Dispensables	11
6.2	Refactoring of Sign In and Sign Up	17
6.3	Additional Refactoring	20
7	Design Pattern Implementation and New Functionalities	20
7.1	Design Pattern Implementation	20
7.2	Added Functionalities	20

1 Project Management

1.1 Team Members

Student Names	Student ID
Louis Nathan Vessah Njoya Tchuenté	i6371413
Catalin Baraboi	i6379754
Ashour Kariakous	i6385235
Abdul Moiz Akbar	i6375558

1.2 Overview of who did what

Student Names	Student ID	Task
Louis Nathan Vessah Njoya Tchuenté	i6371413	bug fixes, Implementation of direct messaging, password hashing, management of repository, final report
Catalin Baraboi	i6379754	Refactoring of SignIn/UP UI, bug fixes, final report
Ashour Kariakous	i6385235	Implementation of comments, direct Messaging, UI refactoring, final report
Abdul Moiz Akbar	i6375558	Refactoring of Duplicate code from all UI files, bug fixes, and signIN/UP UI design, final report

2 Introduction

The core of this app revolves around letting users set up and personalize their profiles. Users sign up, log in, and add a personal touch with photos and bios, just as they would on Instagram. Once that is done, users can start sharing snapshots of their lives by uploading images with captions.

3 Requirements Analysis

3.1 Functional requirements

- 1 Log-in / Sign-Up: Profile picture / Bio
- 2 Profile picture: Drawing the letter of username, display upload picture, or a default profile picture
- 3 Sharing snapshots with Captions: Hashtags
- 4 Discovery / Explore Page: Interesting tags, weekly photo challenges

- 5 Algorithm for Trending / Recommendations: In your year, course, faculty
- 6 Private/public mode: User must follow to see your content
- 8 Handle a variety of image formats
- 9 UI for exploring others' posts: Ability to Like them and Comment on them
- 10 Messaging system: Able to direct message a person
- 11 Commenting system: Ability to comment on someone's post
- 12 Real-Time notification system, about comments, likes and follow
- 14 Load Data from a txt file
- 15 Bio length of max of 150 characters
- 16 Reporting tool, multiple options like (trying to be someone else, posting abusive content etc)
- 17 Relationship Manager: Ability to Follow other users
- 18 Image sizes: Example cannot upload a image more than 5 MB for better system performance
- 19 Photo editing / Cropping Profile Pictures in a Circle to fit
- 20 User styling: Vibrant color palette
- 21 Font size should be similar to mobile view
- 22 Accessibility: handle visual and auditory(increase font sizes) impairments by implementing text-to-speech, color contrast
- 23 Live Streaming
- 24 Shopping functionality
- 25 Feature requests: submit potential extensions
- 26 Submit review: feedback form
- 27 Filtering option to filter swear words by replacing characters by hashtags

3.2 Non functional Requirements

- 1** Good privacy system, user data, posts, comments, likes are stored with care, ensuring integrity, HTTPS / Moderation Included
- 2** User password stored securely
- 3** Interface should be intuitive, easy to navigate
- 4** System should provide helpful error message
- 5** System should be modular and easily extendable
- 6** Performance: liking and following should be fast but limit i.e like Instagram
- 7** Encryption of user data and decryption: when login and reading credential files

4 Use Case Descriptions

4.1 User Registration

If a user has not made an account, he has the option to create an account, then system displays registration form, and user fills Username, Bio, Password. User also has an option to upload a profile picture, but its not mandatory and if user proceeds without uploading photo no profile picture would be shown in his profile. If a user enters a duplicate name which has already been saved as credentials, then an error message is shown to the user to change the username. See figure 1

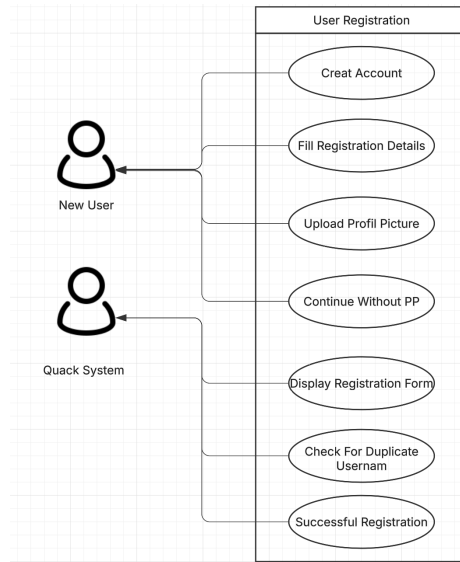


Figure 1: User Registration

4.2 User Login

If the user has an account and want to login again to the app, the system asks the user to enter the username and password. It then checks the credentials file and if the username and password is found it lets user login with all the previous saved interactions user had made before with that profile. In case credentials do not match an error is shown to the user.

4.3 Users Post

Users can upload a post by clicking on the image icon to upload. The user is asked for a caption and an upload image button that opens file chooser dialog. The uploaded image is now shown in the user's DACS profile and in Explore page. Each uploaded image is stored in a folder, and image data such as imageID and timestamp are stored in image details file.

4.4 Liking of Post

If a user has followed someone, he will be able to see uploaded post in Quack-stagram Home page and also be able to like the post and the like count is incremented.

4.5 Error Handling and Edge Cases

Initially whenever a user created a profile his credentials were not saved separately and he had to make a new profile every time to access Quackstagram. We fixed the writing method, and now after every registration, a new User is created and stored inside users.txt file. see figure 2.

4.6 Edit Profile

When a user navigates to Instagram profile, they can edit their profile by clicking on the edit profile button. The user can change their bio or change their profile picture by uploading a new one. Updating the profile picture is optional, meaning the user can proceed without changing it. Once the user finished making changes, they can decide to either submit or cancel.



Figure 2: users.txt

Although we faced a new problem that is, User "Lorin" was always displayed because we had no conditional to check for the logged In user inside the users.txt file before displaying its profile hence since Lorin is the User on the first line of the users.txt file, it was always read and displayed. see figure 3.

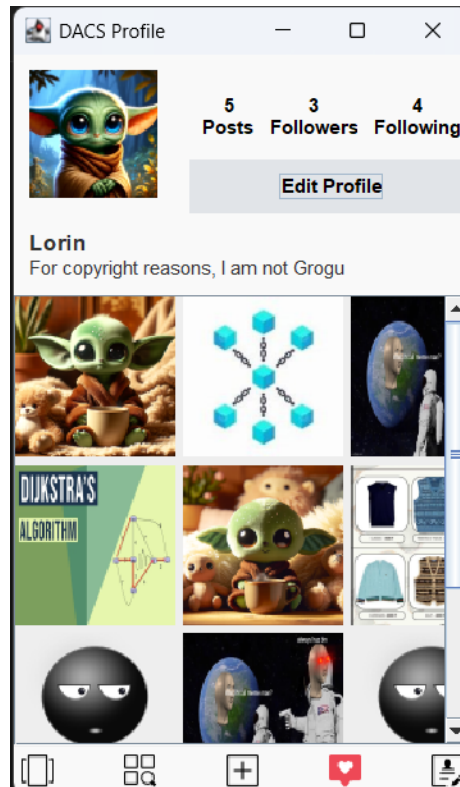


Figure 3: Lorin Profile

The last 3 posts was made from other profile but it is showing in lorin so we had to modify the code and add a loop that iterates the users.txt file and a conditional to verify the logged in user. For example; the following code(see figure 4) was updated to (see figure 5).



```
1 String currentUser = "";
2 try (BufferedReader reader = Files.newBufferedReader(Paths.get("data", "users.txt"))) {
3     String line = reader.readLine();
4     if (line != null) {
5         currentUser = line.split(":")[0].trim();
6     }
7 } catch (IOException e) {
8     e.printStackTrace();
9 }
```

Figure 4: Error codeBlock



```
1 String currentUser = RefactoredSignIn.getLoggedInUsername();
2 try (BufferedReader reader = Files.newBufferedReader(Paths.get("data", "users.txt"))) {
3     String line;
4
5     while ((line = reader.readLine()) != null) { // Iterate through each line
6         String[] parts = line.split(":");
7
8         if (parts.length > 0 && parts[0].trim().equalsIgnoreCase(currentUser)) {
9             currentUser = parts[0].trim();
10            break; // Stop searching once found
11        }
12    }
13 } catch (IOException e) {
14     e.printStackTrace();
15 }
16 }
```

Figure 5: Corrected CodeBlock

5 UML diagrams

5.1 Class Diagrams

5.1.1 Given source code

see figure 6

5.1.2 Refactored code

see figure 7

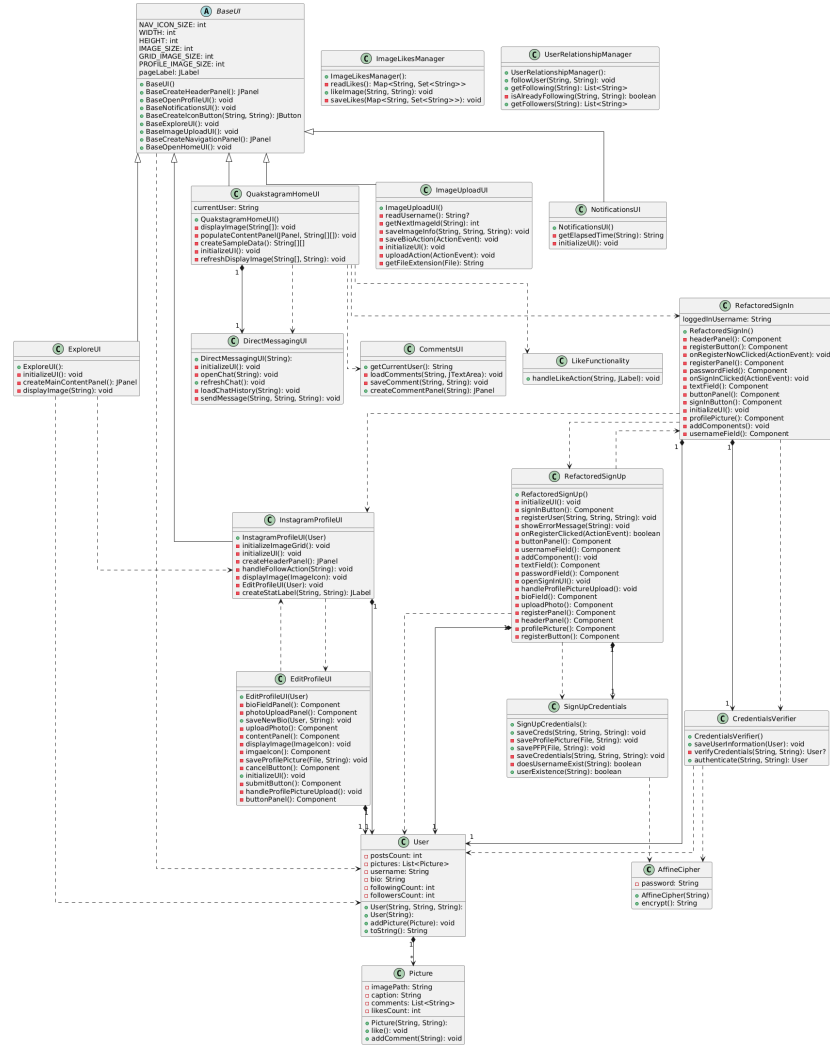


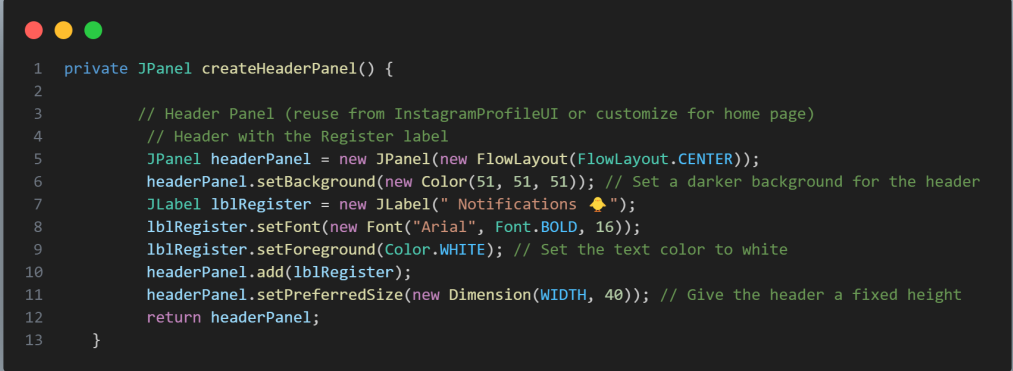
Figure 7: Refactored Code UML diagram

6 Documentation of Refactoring

6.1 Refactoring Dispensables

There was a total of 8 dispensables present in 5 classes precisely UI related class files so we made an abstract class "BaseUI class" that contains all of the dispensables. If needed to add new features we would simply modify the BaseUI class for that. These dispensables are reused by all subclasses, promoting DRY principles. Making BaseUI abstract prevents it from being instantiated directly, as it is meant only to provide base functionality. The dispensables are:

- 1 Create Header Panel: This methods was repeated in every UI page and its function was to create a header with some properties. See figure 8



```
1 private JPanel createHeaderPanel() {
2
3     // Header Panel (reuse from InstagramProfileUI or customize for home page)
4     // Header with the Register label
5     JPanel headerPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
6     headerPanel.setBackground(new Color(51, 51, 51)); // Set a darker background for the header
7     JLabel lblRegister = new JLabel(" Notifications 🔔");
8     lblRegister.setFont(new Font("Arial", Font.BOLD, 16));
9     lblRegister.setForeground(Color.WHITE); // Set the text color to white
10    headerPanel.add(lblRegister);
11    headerPanel.setPreferredSize(new Dimension(WIDTH, 40)); // Give the header a fixed height
12    return headerPanel;
13 }
```

Figure 8: Creation of Header Panel

- 2 Create Icon Buttons: This method was used to add action listeners to every button prevent on the navigation panel. See figure 9



Figure 9: Adding action listeners to Icon buttons

- 3 Create Navigation Panel: This method was used to add the icon buttons such as home button, explore page button etc to a panel. See figure 10



```
1
2 private JPanel createNavigationPanel() {
3     // Create and return the navigation panel
4     // Navigation Bar
5     JPanel navigationPanel = new JPanel();
6     navigationPanel.setBackground(new Color(249, 249, 249));
7     navigationPanel.setLayout(new BorderLayout(navigationPanel, BorderLayout.X_AXIS));
8     navigationPanel.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
9
10    navigationPanel.add(createIconButton("img/icons/home.png", "home"));
11    navigationPanel.add(Box.createHorizontalGlue());
12    navigationPanel.add(createIconButton("img/icons/search.png", "explore"));
13    navigationPanel.add(Box.createHorizontalGlue());
14    navigationPanel.add(createIconButton("img/icons/add.png", " "));
15    navigationPanel.add(Box.createHorizontalGlue());
16    navigationPanel.add(createIconButton("img/icons/heart.png", "notification"));
17    navigationPanel.add(Box.createHorizontalGlue());
18    navigationPanel.add(createIconButton("img/icons/profile.png", "profile"));
19
20    return navigationPanel;
21 }
```

Figure 10: Navigation Panel

4 Action Listeners: These 5 methods are the actual action listeners for each icon button. See figure 11

```
1 private void openProfileUI() {
2     // Open InstagramProfileUI frame
3     this.dispose();
4     String loggedInUsername = "";
5
6     // Read the logged-in user's username from users.txt
7     try (BufferedReader reader = Files.newBufferedReader(Paths.get("data", "users.txt"))) {
8         String line = reader.readLine();
9         if (line != null) {
10             loggedInUsername = line.split(":")[0].trim();
11         }
12     } catch (IOException e) {
13         e.printStackTrace();
14     }
15     User user = new User(loggedInUsername);
16     InstagramProfileUI profileUI = new InstagramProfileUI(user);
17     profileUI.setVisible(true);
18 }
19
20 private void notificationsUI() {
21     // Open InstagramProfileUI frame
22     this.dispose();
23     NotificationsUI notificationsUI = new NotificationsUI();
24     notificationsUI.setVisible(true);
25 }
26
27 private void ImageUploadUI() {
28     // Open InstagramProfileUI frame
29     this.dispose();
30     ImageUploadUI upload = new ImageUploadUI();
31     upload.setVisible(true);
32 }
33
34 private void openHomeUI() {
35     // Open InstagramProfileUI frame
36     this.dispose();
37     QuakstagramHomeUI homeUI = new QuakstagramHomeUI();
38     homeUI.setVisible(true);
39 }
40
41 private void exploreUI() {
42     // Open InstagramProfileUI frame
43     this.dispose();
44     ExploreUI explore = new ExploreUI();
45     explore.setVisible(true);
46 }
```

Figure 11: Action Listeners

After refactoring, we had the following methods which were used in the different UI classes by inheritance of the BaseUI class.

- 1** Create Header Panel: This method was added to the Base UI class and inherited by all UI pages. Implementation remained unchanged.
- 2** Create Icon Buttons: This method was added to the Base UI class and inherited by all UI pages. Implementation remained unchanged.
- 3** Create Navigation Panel: This method was added to the Base UI class and inherited by all UI pages. Implementation remained unchanged.
- 4** Action Listeners: These methods were added to the Base UI class and to ensure the header panels were correctly updated, upon calling of the method it changes the header panel's label while other UI pages had their own UI initialization (see an example in figure). See figure 12

```

1  public void BaseImageUploadUI() {
2      // Open InstagramProfileUI frame
3      this.dispose();
4      pageLabel = new JLabel("Image Upload 📷");
5      ImageUploadUI upload = new ImageUploadUI();
6      upload.setVisible(true);
7  }
8
9  public void BaseOpenProfileUI() {
10     this.dispose();
11
12     // Access the logged-in username correctly
13     String loggedInUsername = RefactoredSignIn.getLoggedInUsername(); // Fix: Use the getter method
14
15     if (loggedInUsername == null || loggedInUsername.isEmpty()) {
16         System.out.println("Error: No user is logged in!");
17         return;
18     }
19
20     // Open profile with the correct user
21     User user = new User(loggedInUsername);
22     InstagramProfileUI profileUI = new InstagramProfileUI(user);
23     profileUI.setVisible(true);
24 }
25
26 public void BaseNotificationsUI() {
27     // Open InstagramProfileUI frame
28     this.dispose();
29     pageLabel = new JLabel("Notification 📢");
30     NotificationsUI notificationsUI = new NotificationsUI();
31     notificationsUI.setVisible(true);
32 }
33
34 public void BaseOpenHomeUI() {
35     // Open InstagramProfileUI frame
36     this.dispose();
37     QuakstagramHomeUI homeUI = new QuakstagramHomeUI();
38     homeUI.setVisible(true);
39 }
40
41 public void BaseExploreUI() {
42     // Open InstagramProfileUI frame
43     this.dispose();
44     pageLabel = new JLabel("Explore 📖");
45     ExploreUI explore = new ExploreUI();
46     explore.setVisible(true);
47 }

```

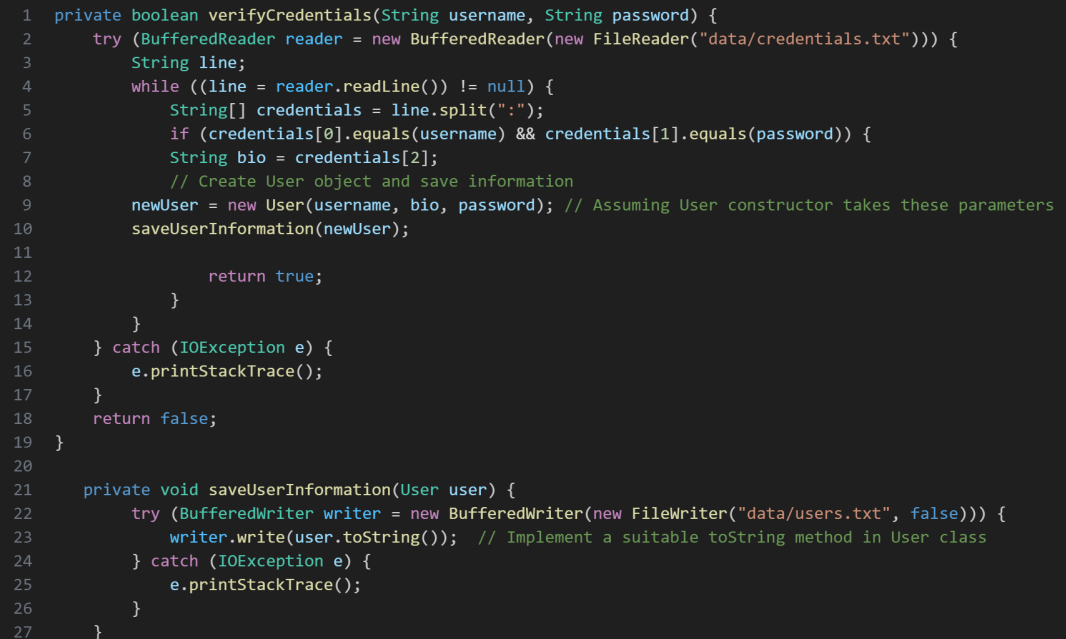
Figure 12: refactored Listeners

6.2 Refactoring of Sign In and Sign Up

The original implementation of SignInUI and SignUpUI contained multiple flaws, all of which heavily complicated the code, making it difficult to comprehend and maintain.

It violated multiple SOLID principles as follows:

- 1 Single Responsibility Principle: The codes had to build the user interface, read and write user data to a file, and handle security like verifying user credentials, and saving them if needed. This complicated the code by making it prone to more bugs upon changing something, making it difficult to maintain or upgrade. See figure 13



```
1 private boolean verifyCredentials(String username, String password) {
2     try (BufferedReader reader = new BufferedReader(new FileReader("data/credentials.txt"))) {
3         String line;
4         while ((line = reader.readLine()) != null) {
5             String[] credentials = line.split(":");
6             if (credentials[0].equals(username) && credentials[1].equals(password)) {
7                 String bio = credentials[2];
8                 // Create User object and save information
9                 newUser = new User(username, bio, password); // Assuming User constructor takes these parameters
10                saveUserInformation(newUser);
11            }
12            return true;
13        }
14    }
15 } catch (IOException e) {
16     e.printStackTrace();
17 }
18 return false;
19 }
20
21 private void saveUserInformation(User user) {
22     try (BufferedWriter writer = new BufferedWriter(new FileWriter("data/users.txt", false))) {
23         writer.write(user.toString()); // Implement a suitable toString method in User class
24     } catch (IOException e) {
25         e.printStackTrace();
26     }
27 }
```

Figure 13: Violation of Single Responsibility Principle in SignUI class

- 2 The Open/Closed principle: Given that the code was not prone to easy changes due to its cluttered design, which was also difficult to navigate through, so extending the code would be difficult if not impossible.
- 3 The Liskov Substitution Principle: It was violated through the code's high coupling, having to handle drawing the UI and verifying user data at the same time, all that while reading data from a file as well will lead to errors if a subclass is implemented to substitute the UI.
- 4 The Interface Segregation Principle: Both classes, SignInUI and SignUpUI had to handle too many tasks without using any interfaces, which caused them to have too much unnecessary code, like verifying credentials.
- 5 The Dependency Inversion Principle: Through the direct access to I/O files for both SignInUI and SignUpUI, making them dependent on lower level modules. See figure 13

As for the refactored side of the codes, we made sure they respect the SOLID principles, whilst also being cohesive, well-structured, and easily maintainable.

- 1 RefactoredSignIn and RefactoredSignUp now only handle the UI logic. A different file named CredentialsVerifier has been made in order to handle the credential verification and file operations for RefactoredSignIn. RefactoredSignUp is using a similar file named SignUpCredentials, which handles the credential verification and saving, similar to CredentialsVerifier for RefactoredSignIn. By doing these we respect the single Responsibility principle
- 2 The Open/Closed principle is respected by allowing easy authentication strategies by modifying just the credential codes, without having to touch SignIn or SignUp.
- 3 The Interface Segregation Principle is respected due to the credential codes acting as abstract interfaces, simply being called by SignIn and SignUp.
- 4 Dependency Inversion is respected through the codes' dependency on abstract classes (CredentialsVerifier, SignUpCredentials), SignIn and SignUp now call a simple method, instead of containing an entire block of code for authentication logic.
- 5 As for the Liskov Substitution Principle, whilst not being directly addressed, CredentialsVerifier and SignUpCredentials act like abstractions by separating authentication logic from the UI. Furthermore, you can now extend those 2 classes or implement an Authenticator interface without changing how RefactoredSignIn/RefactoredSignUp work. See figure 14, where the SOLID principles are respected

```

1  public class SignUpCredentials{
2
3      private final String profilePhotoStoragePath = "img/storage/profile/";
4      private final String credentialsFilePath = "data/credentials.txt";
5
6      private void saveProfilePicture(File file, String username) {
7          try {
8              BufferedImage image = ImageIO.read(file);
9              File outputFile = new File(profilePhotoStoragePath + username + ".png");
10             ImageIO.write(image, "png", outputFile);
11         } catch (IOException e) {
12             e.printStackTrace();
13         }
14     }
15
16     private void saveCredentials(String username, String password, String bio) {
17         try (BufferedWriter writer = new BufferedWriter(new FileWriter("data/credentials.txt", true))) {
18             AffineCipher passwordHasher = new AffineCipher(password);
19             writer.write(username + ":" + passwordHasher.encrypt() + ":" + bio);
20             writer.newLine();
21         } catch (IOException e) {
22             e.printStackTrace();
23         }
24     }
25
26     private boolean doesUsernameExist(String username) {
27         try (BufferedReader reader = new BufferedReader(new FileReader(credentialsFilePath))) {
28             String line;
29             while ((line = reader.readLine()) != null) {
30                 if (line.startsWith(username + ":")) {
31                     return true;
32                 }
33             }
34         } catch (IOException e) {
35             e.printStackTrace();
36         }
37         return false;
38     }
39
40     public void savePFP(File file, String username){
41         saveProfilePicture(file, username);
42     }
43     public void saveCreds(String username, String password, String bio){
44         saveCredentials(username, password, bio);
45     }
46     public boolean userExistence(String username){
47         return doesUsernameExist(username);
48     }

```

Figure 14: Creation of a separate class use to save information upon signUp

In conclusion, refactoring SignInUI and SignUpUI leads to significant improvements in many areas of the codes. The codes are now much cleaner, much more comprehensible, and easily maintainable. Isolating the authentication logic from the UI logic improved the structure by removing unnecessary dependencies and adding dependencies on abstract classes. Additionally, the codes can now be easily extendable if needed, in case you might even want to switch to a database type of storage.

6.3 Additional Refactoring

After refactoring the code base and adding some new functionalities, we encounter a bloater, precisely the "QuackstagramHomeUI" class. To solve this issue, we decided to move some of its functionalities to separate files precisely the following:

- 1 Comments UI: This newly created class handles the loading of previous comments, and saving of comments
- 2 LikeFunctionality: This class handles the like action.

7 Design Pattern Implementation and New Functionalities

7.1 Design Pattern Implementation

- 1 Factory Pattern: We implemented this pattern to facilitate the creation of new Users open registration. This is done in the SignUp class where upon registration we call the registerUser method that calls the credentials saver and saves the new user's information into a txt file
- 2 Decorator: This was implemented in UI pages to add new functionalities such as direct messaging in the home UI
- 3 Strategy: In the abstract class BaseUI, a Strategy pattern is used to implement the various action listeners.

7.2 Added Functionalities

- 1 Password hashing was implemented to ensure security. This is to prevent unwanted access to the users account whose passwords are stored in the credentials/users text files.
To hash the passwords, an Affine Cipher was implemented, where the cipher applies a mathematical function to each letter's numeric value. The encryption function receives a string and places them into a character Array. The function then loops through the array and if the character is in lowercase, it applies a mathematical function which converts the lowercase letters into other values based on their unicodes else remains unchanged.

See figure 15. After each iteration the new character is added to a newly

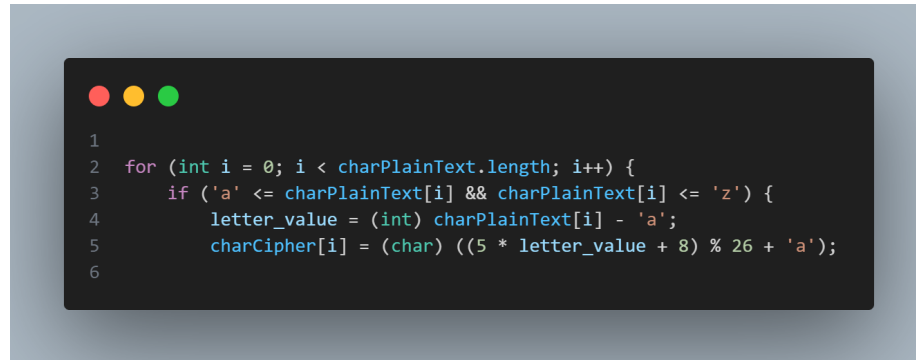
A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in a syntax-highlighted language, likely Java, and shows a loop that iterates over the length of a string 'charPlainText'. Inside the loop, it checks if the current character is a lowercase letter. If so, it calculates its value relative to 'a', applies the Affine Cipher formula $(5 * \text{letter_value} + 8) \% 26$, and then adds 'a' back to get the encrypted character, which is stored in 'charCipher'.

Figure 15: Applied Mathematical Function

created array, and stored inside the credentials and users txt files. Here is a code snippet that demonstrates how the cipher is used to encrypt the entered password upon signUp. See figure 16. See figure 17 for an overview of how encryption is performed and stored

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in a syntax-highlighted language, likely Java, and shows a method 'saveCredentials' that takes 'username', 'password', and 'bio' as parameters. It uses a 'try-catch' block to handle file writing. Inside the try block, it creates a 'BufferedWriter' for 'data/credentials.txt', creates an 'AffineCipher' instance with the password, and writes the username, encrypted password, and bio to the file, separated by colons. It also includes a catch block for 'IOException' that prints the stack trace.

Figure 16: Use of Encryption upon Registration

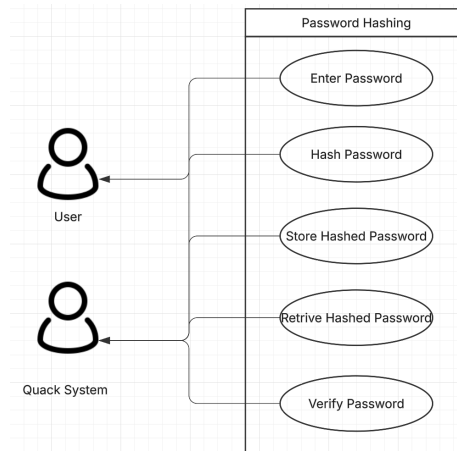


Figure 17: Password Encryption

- 2 Fixed Like Functionality: We updated `handleLikeAction` in `LikeFunctionality` class so that user can only like a post once and not spam. To accomplish this, we added new variables such as `likesTrackingPath`, `alreadyLiked` to keep a track of likes. See figure 18

```

1 static void handleLikeAction(String imageId, JLabel likesLabel) {
2     Path detailsPath = Paths.get("img", "image_details.txt");
3     Path likesTrackingPath = Paths.get("data", "likes_tracking.txt");
4     StringBuilder newContent = new StringBuilder();
5     boolean updated = false;
6     boolean alreadyLiked = false;
7     String currentUser = RefactoredSignIn.getLoggedInUsername();
8     String imageOwner = "";
9     String timestamp = LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
  
```

Figure 18: Additional variables

We also created a new file to maintain this functionality, and modified the code to add conditional that check if a post has already been liked by the logged in user and updates the newly created "likestracking" txt file. See figure 19



```
1 try (BufferedReader likesReader = Files.newBufferedReader(likesTrackingPath)) {
2     String line;
3     while ((line = likesReader.readLine()) != null) {
4         if (line.equals(currentUser + ";" + imageId)) {
5             alreadyLiked = true;
6             break;
7         }
8     }
9 } catch (IOException e) {
10     e.printStackTrace();
11 }
12
13 if (alreadyLiked) {
14     JOptionPane.showMessageDialog(null, "You have already liked this post!", "Like Failed", JOptionPane.ERROR_MESSAGE);
15     return;
16 }
17 // Read and update image_details.txt
18 try (BufferedReader reader = Files.newBufferedReader(detailsPath)) {
19     String line;
20     while ((line = reader.readLine()) != null) {
21         if (line.contains("imageId: " + imageId)) {
22             String[] parts = line.split(";");
23             imageOwner = parts[1].split(": ")[1];
24             int likes = Integer.parseInt(parts[4].split(": ")[1]);
25             likes++; // Increment the likes count
26             parts[4] = "Likes: " + likes;
27             line = String.join(";", parts);
28
29             // Update the UI
30             likesLabel.setText("Likes: " + likes);
31             updated = true;
32         }
33         newContent.append(line).append("\n");
34     }
35 } catch (IOException e) {
36     e.printStackTrace();
37 }
```

Figure 19: Like Tracking and updates

- 3 Direct Messaging:** A functional requirement of quackstagram is to be able to communicate with other users by direct messaging. The user clicks on the messaging icon, selects a user, compose a message, and send it; the system delivers the message to the selected user. These messages are stored in a txt file, and everytime a chat is open the system loads the messaging history and displays it. See figure 20

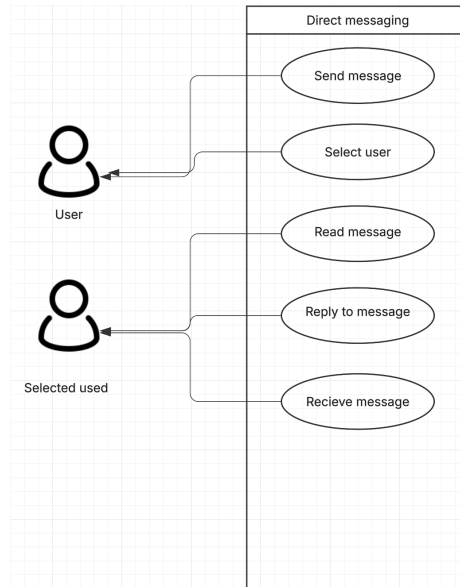


Figure 20: Diagram demonstrating Direct Messaging

- 4 Implmenting Comments:** Another functional requirement was the ability of users to comment under posts. To do so, we implemented a comment section under the HomeUI, where users can type comments and the comments are saved to a txt file. When a comment section is opened, the systems loads the comment history and displays it. See figure 21

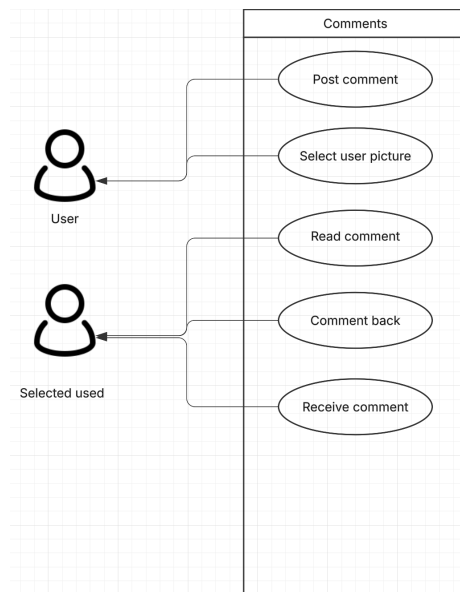


Figure 21: Diagram demonstratinng Commenting