Functional Design

Number	Name	Priority	Remarks
F1	Sign Up/In page	M	With user feedback, with restriction on passwords and allowed user names
F2.1	User profiles	M	Bio, user profile image
F2.2	Editable User Profiles	C	Be able to edit username, password, bio, and remove pictures
F2.3	Private User profiles	C	The only user's able to see your profile are ones following you. You have to accept the follow request
F3.1	Image Posting	M	Post image on a separate screen with a caption
F3.2	Liking Image	C	Be able to like another user's post
F3.3	Editing Image	C	with certain filters, or cropping
F3.4	Posts sorted from newest to oldest	S	On home tab
F3.5	Explore tab	C	Find other posts from other users
F4	Following functionality	S	Go to other user's profile and follow that user
F5	Messaging	S	
F6	Notifications	S	Update users on new interactions
F7	Admin users	S	Moderate content, be able to remove inappropriate/harmful content
F7.1	Report button on each post	S	Data available to moderator
F7.2	Weekly challanges	C	In the explore page to drive up interaction
F8	Live streaming	W	
F9	In app shopping	W	Source of revenue
F10.1	Video posting	W	
F10.2	Video editing	W	
F11.1	Accessibility options (font sizing, contrast)	S	

F11.2	Photo alt text Addition when posting	C	
F11.3	Text to Speech Functionality	C	For the visually impaired
F12	User feedback	S	Through a contact form/ survey

Number	Name	Priority	Remarks
NF1	The app must be safe	S	
NF2	The app must be private	S	
NF3	The app gives clear, to the point error messages	S	
NF4	Inappropriate content should be fought	S	
NF5	The app gives no delay that disturbs the UX	S	
NF6	The app should be secure	S	
NF7.1	The app has a vibrant color palette that fits with dacs	S	
NF7.2	The app has harmonic modern fonts	S	Apple SF fonts could work here
NF7.3	The app is simple and intuitive in design	S	

Code refactor

First, the sign in and sign up pages were recognized to generally have the same structure and layout other than the fields and the buttons' labels (look at figure 1.2); therefore, a good adjustment could be the simplification of the authentication UI code in order to prevent duplication and enhance modifiability. The best option for that would be using inheritance, as

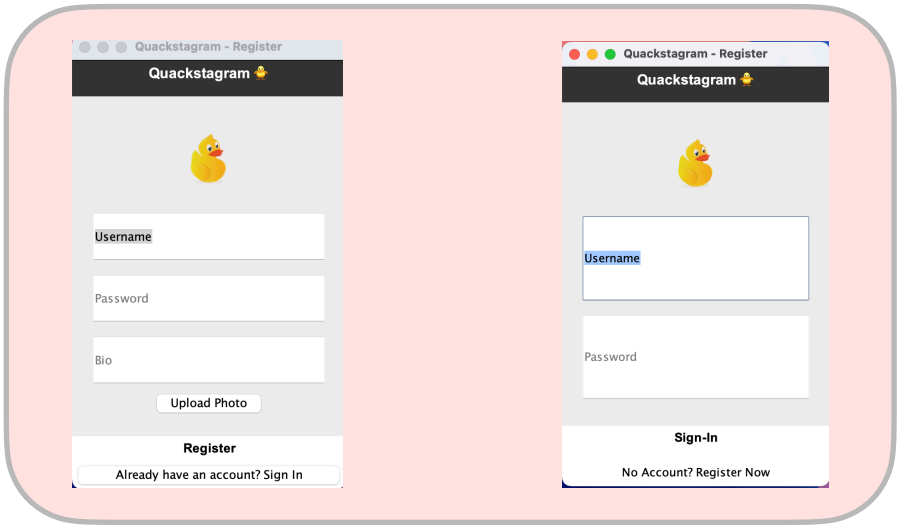

Figure 1.1, Sign In/Up views

the two views generally hold the same properties and most importantly layout but differ in details. Lets call that abstract super class, AuthenticationView.

```
public abstract class AuthenticationView extends JFrame {

//InitializeUi() creates and intitializes UI using the other three abstract functions to handle the details
    final private void initializeUI();
     abstract protected Component[] createInfoFormComponents();
     abstract protected JButton createActionButton();
     abstract protected JButton  createButtonToNavigateToAlternativeAuthenticationView();
}
```

Now that AuthenticationView is written, we can remove InitializeUI from SignInUI and also remove its constructer (that sets frame size and other duplicate code) and only call the parent's constructor AuthenticationView(); leaving the component layout logic all in

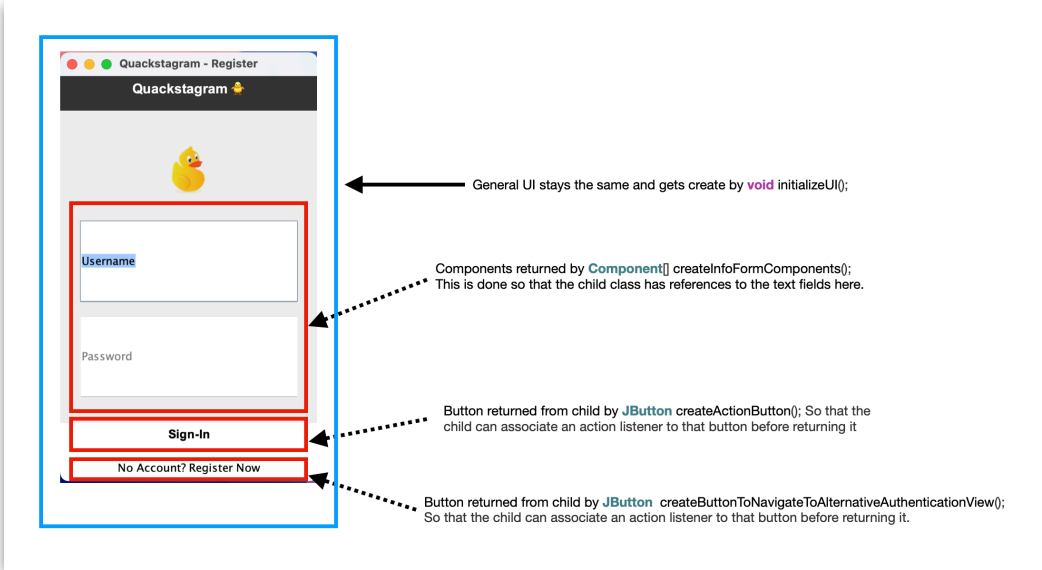

Figure 1.2, elements shared between SignInUI and SignUpUI is bordered by blue, implementation specific elements by red.

AuthenticationView. The code is tested and it was observed to be working, we can then apply the same changes to SignUpUI. Now that we refactored the authentication process, whilst keeping the code function generally the same, we can look at "the internal screens" [after login].

Looking at the screens, one can identify that all of them share a tab bar for navigation in between each other and most share a header, except for the "DACs profile" screen (look at Figure 1.3). This pattern is also realized in the code, which is very repetitive, causing code bloat.

The solution is very similar to what we did with authentication, create an abstract class with an abstract function that returns a content panel that is unique to the concrete class. We can also do it in a different way where we pass a content panel object to the super constructer of a parent class; however, despite it being simpler from the point of view of a client using this API, we will stick with the former for better code semantics (an abstract skeleton view can't be constructed alone, making it fit more with our model) and for uniformity with the older AuthenticationView code. Furthermore, although the DACs profile technically has a header, it is considered as inherently part of the screen's content and isn't shared with any other screen.

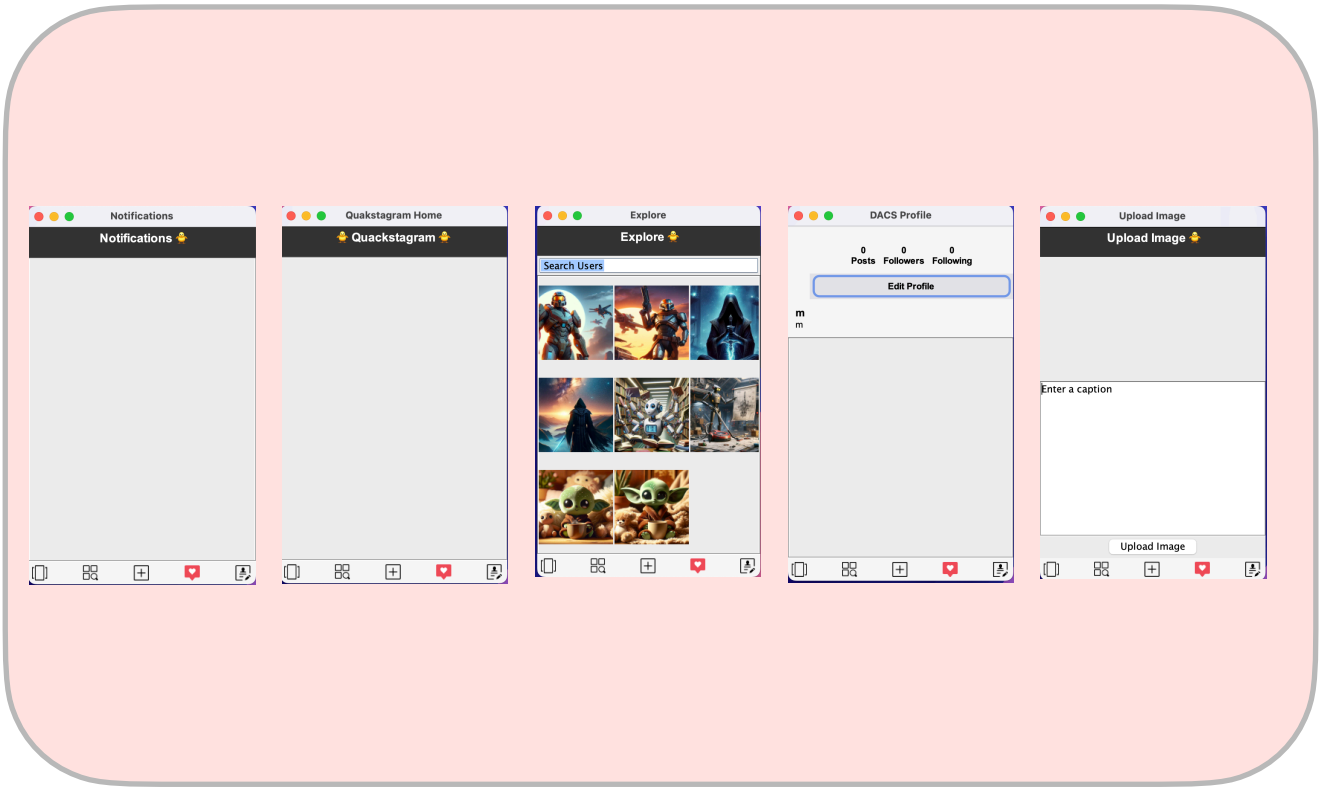

Figure 1.3, main screens. Notice the similarities between them.

Hence, two abstract classes were created: TabView, a screen that has a tab bar for navigation. TabViewWithHeader, a TabView that has a header that is the one shared by the four screens seen in Figure 1.3. This is so as to not force the user bio screen to have a header that looks like the other screens. Otherwise, the black header code would need to be repeated multiple times.

The work started on instagramProfileUI. The design pattern employed in AuthenticationView needed to be modified a bit, since calling overridden abstract functions of the super class did cause problems since the child class fields aren't initialized yet (the fields of the child class are initialized in the constructor of the child class, the overridden abstract function written in the child class uses the child class's fields, but the overridden function is called from the super class's constructor, which is called before the child class's field initialization statements). To prevent such error, components of the layout that needed the child class's fields were called by the child class when it is ready (delayed/lazy initialization).

```
public abstract class TabView extends JFrame {
    TabView(String windowTitle);

    // Call at the end of the child initializes… written here as to make sure the
child's fields are initialized
    protected void showContentPanel() {
        JPanel contentPanel = createContentPanel();
        add(contentPanel, BorderLayout.CENTER);
        revalidate();
        repaint();
    }

    protected void redrawCommonComponents()
    {
        JPanel navigationPanel = createNavigationPanel();
        add(navigationPanel, BorderLayout.SOUTH);
        revalidate();
```

```
        repaint();
    }

    protected void removeScreenContents()

    protected void addCustomComponentNorth(Component c);

    protected void addCustomComponentCenter(Component c);

    protected void addCustomComponentSouth(Component c);


    private JPanel createNavigationPanel();

    private JButton createIconButton(String iconPath, String buttonType);

    private void openProfileUI();

    private void notificationsUI();

    private void ImageUploadUI();

    private void openHomeUI();

    private void exploreUI();

    protected abstract JPanel createContentPanel();
}



public abstract class TabViewWithHeader extends TabView {

   String headerTitle;
   TabViewWithHeader(String headerTitle, String windowTitle);

   @Override
   protected void redrawCommonComponents()
   {
    super.redrawCommonComponents();
    JPanel headerPanel = createHeaderPanel(headerTitle);
    add(headerPanel, BorderLayout.NORTH);
    revalidate();
    repaint();
   }
   private JPanel createHeaderPanel(String headerTitle);
}
```

After restructuring all the other classes using the two classes above as frameworks, removing all unnecessary and redundant comments (which was all of them), and formatting the code we observe that despite hundreds of lines of repeated code being wiped out, there still lies the heavy interdependence between the UI and data modification and extraction··· causing bloat, this can cause problems if we wanted to say for example, move to a sql database, or launch on other devices with different persistent storage APIs.

To suppress this issue (classes breaking the single principle and DRY), we will use the following strategy: separate the classes into the categories, classes that model data, classes that create and customize views visually, controllers that interlink the data base abstraction/service layers (that housed the models) and the views.

First a new class called AuthenticationController was created to interface between the SignInUI and SignUpUI classes, then an abstract class named Database<Model> was created to be an abstraction between database implementation details and other code. The classes are shown below.

```
public abstract class Database<Model> {
    private List<Model> models;

    Database() {
        loadPath(); // make sure that a db path is there, if it isnt declared as a constant
        models = retrieveAll();
    }

    protected abstract List<Model> retrieveAll();

    abstract public boolean save(Model m);

    abstract public boolean update(Model m);

    public List<Model> getMatching(Predicate<Model> filter) {
        List<Model> matchingModels = new ArrayList<>();
        for (Model m : models) // checking models != null was done here on purpose
                               //to ensure functionallity at
                                // or close to compile time, check child implementations of
retreiveAll()
        {
            if (filter.test(m)) {
                matchingModels.add(m);
            }
        }
        return matchingModels;
    }

    protected abstract void loadPath();
}
```

```
public class AuthenticationController {

    private Database<User> credDB;
    private Database<User> userDB;
    private Database<ProfilePhotoData> profilePicDB;

    public class VerificationResult {
        private boolean success = false;
        private User newUser = null;

        VerificationResult(boolean success, User newUser);
        //getters and setters here
    }

    AuthenticationController() {
        credDB = new CredentialsDatabase();
        userDB = new UsersDatabase();
        profilePicDB = new ProfilePhotoDatabase();
    }

    public VerificationResult verifyCredentials(String username, String password);

    private void saveUserInformation(User user);
    public void saveProfilePicture(File file, String username);
    public void saveCredentials(User u);

    public boolean doesUsernameExist(User m);

    private boolean credentialsPredicate(User t, User other);

    private boolean userNamePredicate(User t, User other);
}
```

Note to examiner, I didn't cheat, I have previous experience with swiftData and swiftUI and therefore I am familiar with closures and query predicates.

Notice the use of Predicate<Model> in Database<Model>, this makes life much easier when working with a Database child class, since there doesn't need to be multiple variations of

List<Model> getMatching each time we need a different testing criterion, we just need to write the test on the fly and that is it.

Refactoring SignInUI and SignUpUI with these classes makes the former much cleaner and cohesive, the two classes now have one and only one mission, draw the UI. Data manipulation and extraction is handled in another class now, and changing that class won't have us change SignIn/SignUp UI class's implementation.

Notifications was changed next, a Notification class was introduced to contain all information needed about notifications, and similar to the approach used on the authentication screens, a NotificationController and NotificationDatabase were introduced too. All of the other classes were changed in a similar manner too, for brevity the changes will not be discussed in detail but at the end of this phase of the refactor we have the following:

Models:

```
class User
public class SimplePicture
class Picture extends SimplePicture
public class ProfilePhotoData
public class Notification
```

Views:

```
public abstract class TabViewWithHeader extends TabView
public abstract class TabView extends JFrame
public class SignUpUI extends AuthenticationView
public class SignInUI extends AuthenticationView
public class QuakstagramHomeUI extends TabViewWithHeader
public class NotificationsUI extends TabViewWithHeader
public class InstagramProfileUI extends TabView
public class ImageUploadUI extends TabViewWithHeader
public class ExploreUI extends TabViewWithHeader
public abstract class AuthenticationView extends JFrame
```

Controllers:

```
public class TabViewController
public class ProfileController extends TabViewController
public class NotificationsController extends TabViewController
public class ImageUploadController extends TabViewController
public class HomeController extends TabViewController
public class ExploreController extends TabViewController
public class AuthenticationController
```

Databases/Data handler abstractions:

```
public class UsersDatabase extends Database<User>
public class UserRelationshipManager
public class UploadedImagesDatabase extends Database<SimplePicture>
public class ProfilePhotoDatabase extends Database<ProfilePhotoData>
public class PicturesDatabase extends Database<Picture>
public class NotificationsDatabase extends Database<Notification>
public class CredentialsDatabase extends Database<User>
public abstract class Database<Model>
```

In TabView, there was the following function:

```
    private JButton createIconButton(String iconPath, String buttonType) {
        ImageIcon iconOriginal = new ImageIcon(iconPath);
        Image iconScaled = iconOriginal.getImage().getScaledInstance(NAV_ICON_SIZE, NAV_ICON_SIZE,
Image.SCALE_SMOOTH);
        JButton button = new JButton(new ImageIcon(iconScaled));
```

```
        button.setBorder(BorderFactory.createEmptyBorder());
        button.setContentAreaFilled(false);

        if ("home".equals(buttonType)) {
            button.addActionListener(_ -> openHomeUI());
        } else if ("profile".equals(buttonType)) {
            button.addActionListener(_ -> openProfileUI());
        } else if ("notification".equals(buttonType)) {
            button.addActionListener(_ -> notificationsUI());
        } else if ("explore".equals(buttonType)) {
            button.addActionListener(_ -> exploreUI());
        } else if ("add".equals(buttonType)) {
            button.addActionListener(_ -> ImageUploadUI());
        }
        return button;

    }
```

Along with the navigation bar creation in the tabview, make the addition of new actions to the tabview a bit harder and tabview more cluttered, hence a new class was created (`public class NavigationBar extends JPanel`) along with the following interface:

```
public interface INavButtonCreator {
    List<TabViewNavigationButton> createNavButtons();
}
```

Which tabview is implemented to return the needed buttons, making the navigation bar much more modular.

Looking at QuakstagramHomeUI class we can see the following in two functions (populateContentPanel and displayImage) as repeated code:

```
            BufferedImage originalImage = ImageIO.read(new File(p.getImagePath()));
            BufferedImage croppedImage = originalImage.getSubimage(0, 0,
                Math.min(originalImage.getWidth(), IMAGE_WIDTH),
                Math.min(originalImage.getHeight(), IMAGE_HEIGHT));
            ImageIcon imageIcon = new ImageIcon(croppedImage);
            imageLabel.setIcon(imageIcon);
```

The following helper function was extracted:

```
    private ImageIcon loadCroppedImage(String imagePath, int maxWidth, int maxHeight) {
        try {
            BufferedImage originalImage = ImageIO.read(new File(imagePath));
            int width = Math.min(originalImage.getWidth(), maxWidth);
            int height = Math.min(originalImage.getHeight(), maxHeight);
            BufferedImage croppedImage = originalImage.getSubimage(0, 0, width, height);
            return new ImageIcon(croppedImage);
        } catch (IOException ex) {
            return null;
        }
    }
```

Observe the code of Database<Model>, children of the class are forced to implement the abstract function update, despite no database except one (the picture database) use it, in accordance with the interface segrigation principle and to insure a coherent interface to our software, we seperate Database<Model> into two abstract classes:

```
public abstract class UpdatableDatabase<Model> extends Database<Model>
public abstract class Database<Model>
```

This concludes our refactor.

Understand that it is almost impossible to show all the code snippets (as requested); however, I hope I displayed some semblance of competence with the selected snippets and explanations above, and that that is satisfactory to you the examiner. Consult the original given code with the new refactored code to see all the changes.

Adding functionality

Despite function F3.5 being of a lower priority, one could argue that the only thing that isn't working according to the expectations of the client is the search bar, going against what we laid out in NF7.3.

We therefore chose the search bar as the next functionality to be added as part of the explore tab, because the explore page was already made, and now is part of the product.

Let us specify exactly what we mean by the search bar, the following excerpt from lab 4 maual states that: "Search Functionality: Add a search bar that lets users find profiles, hashtags, or posts. Powerful, but challenging!"

That statement is vague, by the use of "or" the search criteria could be any combination of the stated criteria (profiles, hashtags, or posts); furthermore, how are we searching for each post (by what they contain? By who posted them? By their location?) Or profile (by profile name, or do we create a tag system and generate tags from the bio of the profile).

We therefore assert that our implementation will be using only the usernames of the posters of posts to give back posts of a certain poster (this also fufils the case where a person wants to search for a profile, they press on a image posted, and go into the profile; ofcourse, there is the edge case of a person not posting anything, which we address by displaying a button navigating to the profile).

This functionality can be expanded to have the other criteria (by searching the caption of a post, and by creating a new component that displays info about profiles and acts as a button/navigation link to the profiles respectively) but as F3.5 is of low priority, we reserve that "polish" for later, we just want our app to be functional.

Our implementation of Database<Model> pays dividends here and it makes it very easy for us to simply collect all the posts and filter by username using our predicate.

This is the main content creation function of our exploreUI class:

```
    private JPanel createMainContentPanel() {
        JPanel searchPanel = new JPanel(new BorderLayout());
        JTextField searchField = new JTextField(" Search Users");

        searchPanel.add(searchField, BorderLayout.CENTER);
        searchPanel.setMaximumSize(new Dimension(Integer.MAX_VALUE, searchField.getPreferredSize().height));

        JPanel imageGridPanel = new JPanel(new GridLayout(0, 3, 2, 2));

        // TODO:get content of search field

        List<SimplePicture> images = controller.getAllUploadedImages(""); //notice the string "", this is where
our username goes as a filter
        for (SimplePicture image : images) {
            ImageIcon icon = new ImageIcon(new
ImageIcon(image.getImagePath()).getImage().getScaledInstance(IMAGE_SIZE,
                    IMAGE_SIZE, Image.SCALE_SMOOTH));
            JLabel imageLabel = new JLabel(icon);
            imageLabel.addMouseListener(new MouseAdapter() {
                @Override
                public void mouseClicked(MouseEvent e) {
                    displayImage(image.getImageID());
                }
            });
            imageGridPanel.add(imageLabel);
        }

        JScrollPane scrollPane = new JScrollPane(
                imageGridPanel);
        scrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
        scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);

        JPanel mainContentPanel = new JPanel();
        mainContentPanel.setLayout(new BoxLayout(mainContentPanel, BoxLayout.Y_AXIS));
        mainContentPanel.add(searchPanel);
        mainContentPanel.add(scrollPane);
```

```
        return mainContentPanel;
    }
```

This is the code for fetching the uploaded images in explore controller is the following:

```
    public List<SimplePicture> getAllUploadedImages(String username) {
        List<SimplePicture> images = uploadedImageDataBase.getMatching(username.isBlank() ?
                                                                       (_) -> true // return all
images if the search field is empty
                                                                       : (t) ->
t.getImagePosterName().equals(username));
        return images;
    }
```

〔The reason the explore controller is so perfectly set up for this is that I looked at the lab 4 before completing the refactor, I am doing all of this the day before the deadline, I know… but I can explain〕

We should find a way to contact our controller when the search field changes as… this post: https://stackoverflow.com/questions/3953208/value-change-listener-to-jtextfield states that I should add a document listener to my searchfield's underlying document.

Segragating the creation of the imageGrid we get:

```
    private void updateImageGrid(String usernameFilter, JPanel imageGridPanel) {
        imageGridPanel.removeAll();

        List<SimplePicture> images = controller.getAllUploadedImages(usernameFilter.trim());

        for (SimplePicture image : images) {
            ImageIcon icon = new ImageIcon(image.getImagePath());
            Image scaled = icon.getImage().getScaledInstance(IMAGE_SIZE, IMAGE_SIZE,
Image.SCALE_SMOOTH);
            JLabel imageLabel = new JLabel(new ImageIcon(scaled));

            imageLabel.addMouseListener(new MouseAdapter() {
                @Override
                public void mouseClicked(MouseEvent e) {
                    displayImage(image.getImageID());
                }
            });

            imageGridPanel.add(imageLabel);
        }

        imageGridPanel.revalidate();
        imageGridPanel.repaint();
    }
}
```

Notice that our posts filter only works when the username in the search bar is case correct, also, we would like to have the posts be shown even if the username is partially complete. We update our predicate to this.

```
    private boolean isUsernamePartiallyMatching(String searchFieldText, String other)
    {
        searchFieldText = searchFieldText.toLowerCase();
        other = other.toLowerCase();
        return other.length() >= searchFieldText.length() && other.substring(0,
searchFieldText.length()).equals(searchFieldText);
    }
    public List<SimplePicture> getAllUploadedImages(String username) {
        List<SimplePicture> images = uploadedImageDataBase.getMatching(username.isBlank() ?
                                                                       (_) -> true // return all images if the
search field is empty
                                                                       : (t) ->
isUsernamePartiallyMatching( username, t.getImagePosterName()));
        return images;
    }
```

```
  private JPanel createMainContentPanel() {

    JTextField searchField = new JTextField("");
    JPanel searchPanel = new JPanel(new BorderLayout());
    searchPanel.add(searchField, BorderLayout.CENTER);
    searchPanel.setMaximumSize(new Dimension(Integer.MAX_VALUE, searchField.getPreferredSize().height));

    JPanel imageGridPanel = new JPanel(new GridLayout(0, 3, 2, 2));
    JScrollPane scrollPane = new JScrollPane(imageGridPanel);

    scrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

    searchField.getDocument().addDocumentListener(new DocumentListener() {
        @Override
        public void insertUpdate(DocumentEvent e) {
            updateImageGrid(searchField.getText(), imageGridPanel);
        }

        @Override
        public void removeUpdate(DocumentEvent e) {
            updateImageGrid(searchField.getText(), imageGridPanel);
        }

        @Override
        public void changedUpdate(DocumentEvent e) {
            updateImageGrid(searchField.getText(), imageGridPanel);
        }
    });

    updateImageGrid("", imageGridPanel); // initially show all images

    JPanel mainContentPanel = new JPanel();
    mainContentPanel.setLayout(new BoxLayout(mainContentPanel, BoxLayout.Y_AXIS));
    mainContentPanel.add(searchPanel);
    mainContentPanel.add(scrollPane);

    return mainContentPanel;
}
```

One more thing, in case a username exists but they don't have any posts, we'd like to show some button that leads to the user's profile, if it doesn't exist we tell the user that there is nothing to be shown.

```
private void updateImageGrid(String username, JPanel imageGridPanel) {
    imageGridPanel.removeAll();

    List<SimplePicture> images = controller.getAllUploadedImages(username.trim());

    if (images.isEmpty()) {
        imageGridPanel.setLayout(new BorderLayout());
        User user = controller.doesUsernameExist(username.trim());
        if (user != null) {
            JButton goToUserProfileButton = new JButton("View " + user.getUsername() + "'s profile");
            goToUserProfileButton.setPreferredSize(new Dimension(200, 50));
            goToUserProfileButton.addActionListener(e -> {
                InstagramProfileUI profileUI = new InstagramProfileUI(user);
                profileUI.setVisible(true);
                dispose();
            });
            JPanel buttonPanel = new JPanel();
            buttonPanel.add(goToUserProfileButton);
            imageGridPanel.add(buttonPanel, BorderLayout.CENTER);
        } else {
            imageGridPanel.add(new JLabel("No results found."), BorderLayout.CENTER);
        }
    } else {
        imageGridPanel.setLayout(new GridLayout(0, 3, 2, 2));
        for (SimplePicture image : images) {
            ImageIcon icon = new ImageIcon(image.getImagePath());
            Image scaled = icon.getImage().getScaledInstance(IMAGE_SIZE, IMAGE_SIZE, Image.SCALE_SMOOTH);
            JLabel imageLabel = new JLabel(new ImageIcon(scaled));

            imageLabel.addMouseListener(new MouseAdapter() {
                @Override
                public void mouseClicked(MouseEvent e) {
                    displayImage(image.getImageID());
                }
            });
            imageGridPanel.add(imageLabel);
        }
    }
    imageGridPanel.revalidate();
```

```
        imageGridPanel.repaint();
    }
```

It seems that the search bar isn't working anymore, the key insertion is off. After some research, we find that the revalidate and repaint calls are happening too fast (at each key insertion/removal), we therefore have to delay the updates by having a timer and resetting the timer each time a key is pressed.

```
        int delay = 300;
        Timer timer = new Timer(delay, e -> updateImageGrid(searchField.getText(), imageGridPanel));
        timer.setRepeats(false);

        searchField.getDocument().addDocumentListener(new DocumentListener() {
            @Override
            public void insertUpdate(DocumentEvent e) {
                timer.restart();
            }
            @Override
            public void removeUpdate(DocumentEvent e) {
                timer.restart();
            }
            @Override
            public void changedUpdate(DocumentEvent e) {
                timer.restart();
            }
        });
```

The search bar is now functional.

Design Patterns

Taking a look at our controllers, we see the following characteristic shared between them:

```
    Database<SimplePicture> uploadedImageDataBase;
    Database<Picture> detailedImagDatabase;
    Database<User> credDB;
    ExploreController() {
        super();
        uploadedImageDataBase = new UploadedImagesDatabase();
        detailedImagDatabase = new PicturesDatabase();
        credDB = new CredentialsDatabase();
    }
```

The dependencies are all created in the constructor, making testing harder, and the abstraction of Database<Model> useless, what we need is Dependency injection, to decouple the controllers from the data access objects.

```
    public ExploreUI() {
        super("Explore 🐣", "Explore");
        controller = new ExploreController();
        showContentPanel();
    }

    private Database<SimplePicture> uploadedImageDataBase;
    private Database<Picture> detailedImagDatabase;
    private Database<User> credDB;
    ExploreController(Database<User> credDb, Database<SimplePicture> uploadedImageDb, Database<Picture>
detailedImageDb, Database<User> usersDb) {
        super(usersDb);
        uploadedImageDataBase = uploadedImageDb;
        detailedImagDatabase = detailedImageDb;
        credDB = credDb;
    }

    ExploreController()
    {
        super(new UsersDatabase());
        uploadedImageDataBase = new UploadedImagesDatabase();
        detailedImagDatabase = new PicturesDatabase();
        credDB = new CredentialsDatabase();
    }
```

Notice how the interface to the UI didn't change, as the UI shouldn't be concerned with the initialization of the Databases. This change was done to all controllers.

One more thing, the authentication is handled in the Authentication controller, suppose you wanted to have a cloud service handle your authentication (as usual), like Google Cloud's Firebase, that would require you to edit the AuthenticationController, violating the open and closed principle. What we need to do, is have a service layer be responsible for authentication, a proxy. We call this proxy, IAuthenticationService

```
public interface IAuthenticationService {
    void registerUser(String username, String bio, String password);
    VerificationResult loginUser(String username, String password);
}
```

We create a concretion of this service, AuthenticationService

```
import java.util.List;

class AuthenticationService implements IAuthenticationService {
    private Database<User> credDB;
    private Database<User> userDB;

    AuthenticationService(Database<User> credDB, Database<User> userDB) {
        this.credDB = credDB;
        this.userDB = userDB;
    }

    @Override
    public void registerUser(String username, String bio, String password) {
        User u = new User(username, password, bio);
        credDB.save(u);
    }

    @Override
    public VerificationResult loginUser(String username, String password) {
        List<User> matches;
        matches = credDB.getMatching(
                (t) -> credentialsPredicate(t, username, password));

        if (!(matches.isEmpty())) {
            User u = matches.getFirst(); // getting first since only one user is going to match the
predicate
            saveUserInformation(u);
            return new VerificationResult(true, u);
        }
        return new VerificationResult(false, null);
    }

    private void saveUserInformation(User user) {
        userDB.save(user);
    }

    private boolean credentialsPredicate(User t, String username, String password) {
        return (t.getUsername().equals(username)
                && t.getPassword().equals(password));
    }
}
```

The controller is now much cleaner. One can say that a service layer could be made for all of the Database<Model> and have controllers interact with even a more abstract object, taking the data manipulation out of the controllers and into some proxy. 〔However, currently I view Database<Model> to be abstract enough, although if it was my own personal project, multiple service layer creation is exactly the thing I would do next.〕

Appendix A:

Use cases:

Use Case 1: User Registration

- Use Case Name: Register User
- Primary Actor: New User
- Goal: To create a new account on the app
- Preconditions:
  - The user has not registered an account before
  - The user has access to the Quackstagram application.
- Postconditions:
  - A new user (with a unique username) account is created in the system
  - The user can log in to app with the new credential
- Main Success Scenario:
  - BF.1 The new user accesses the registration page
  - BF.2 The new user enters a unique username
  - BF.3 The new user enters a password
  - BF.4 The new user enters a bio (optional; however, "No bio" gets put in)
  - BF.5 The new user may choose to upload a profile picture (optional)
  - BF.6 The new user submits the registration form
  - BF.7 The app validates the input (unique username) {AF.1, AF.2}
  - BF.8 The app creates a new user account with the provided info.
  - BF.9 The system stores the user credentials and profile data. {EF.1}
  - BF.10 The app switches to the SignIn page
- Alt flows:
  - AF.1 The username is already taken.
    1. The system informs the user that the username is not available.
  - AF.2 The user does not provide all the required information.
    1. The system prompts the user to fill all the required fields.
- Exception flows:
  - EF.1 Registration fails due to database fault.

Use Case 2: User Login

- Use Case Name: Log In
- Primary Actor: Registered User
- Goal: To access the user's account.
- Preconditions:
  - The user has a registered account.
- Postconditions:
  - The user is logged into their quackstagram account.
  - The user can access their profile, the home feed, etc.
- Main Success Scenario:
  - BF.1 The user accesses the login page
  - BF.2 The user enters their username
  - BF.3 The user enters their password
  - BF.4 The user submits the login form {AF.1}
  - BF.5 The system verifies the credentials against stored user data {EF.1}
  - BF.6 The system grants access to the user's account
  - BF.7 The user is redirected to the home feed
- Alternative Flows:
  - AF.1. The username does not exist or password is incorrect or both
    1. The user is informed that "username or password doesn't exist".
- Exception flows:
  - EF.1 Login fails due to database fault.

Use Case 3: View Home Feed

- Use Case Name: View Home Feed
- Primary Actor: Logged-in User

- Goal: To view the posts from the users they follow.
- Preconditions:
  - The user is logged into their account.
  - The user follows at least one other user.
  - Other users have uploaded content.
- Postconditions:
  - The user sees a list of posts from the users they follow.
- Main Success Scenario:
  - BF.1 The user navigates to the home feed.
  - BF.2 The system retrieves the list of users the current user follows.
  - BF.3 The system retrieves the posts from the followed users.
  - BF.4 The system displays the posts in a feed format, showing the user who posted and the content. {AF.1}
  - BF.5 The user can interact with the posts (like).
- Alternative flows:
  - AF.1. The followed users posted nothing.
    1. The user is showed nothing.

Use Case 4: Upload Image

- Use Case Name: Upload Image
- Primary Actor: Logged-in User
- Goal: To share a new image with their followers.
- Preconditions:
  - The user is logged into their account.
  - The user has an image to upload.
- Postconditions:
  - The image is uploaded and saved.
  - The image is displayed in the user's profile.
  - The image is displayed to the users who follow the user.
- Main Success Scenario:
  - BF.1 The Logged-in User navigates to the image upload page.
  - BF.2 The user selects an image file from their device.
  - BF.3 The User may add a description to the image (optional, but "no caption" will be added instead).
  - BF.4 The user confirms the upload. {AF.1}
  - BF.5 The system uploads the image. {EF.1}
  - BF.6 The system saves the image data (location, caption, timestamp, etc.)
  - BF.7 The system notifies the user that the upload was successful.
- Alternative flows:
  - AF.1. The user cancels the upload.
    1. The system discards the upload attempt.
- Exception flows:
  - EF.1 Upload fails due to database fault.

Use Case 5: View User Profile

- Use Case Name: View User Profile
- Primary Actor: Logged-in User
- Goal: To view another user's profile information and posts.
- Preconditions:
  - The user is logged into their account.
  - The user knows the username of the profile they want to view.
- Postconditions:
  - The other user's profile information (username, bio, posts, followers/following count) is displayed.
  - The user may choose to follow the other user.
- Main Success Scenario:

    - BF.1 The user searches for another user by username on the explore page
    - BF.2 Posts by the other user are shown. {AF.1, AF.2}
    - BF.3 User presses on post.
    - BF.4 User presses on other user's username.
    - BF.5 User gets shown other user's profile.
- Alternative flows:
    - AF.1. The other user doesn't have posts.
        1. The user gets shown a singular button navigating to the other user's profile.
    - AF.2. The other user doesn't exist.
        1. The app informs the user that it has no results from the search.

Use Case 6: Follow

- Use Case Name: Follow
- Primary Actor: Logged-in User
- Goal: To follow another user on Quackstagram.
- Preconditions:
    - The user is logged into their account.
    - The user is viewing another user's profile.
- Postconditions:
    - The user is added to the list of followers of the followed user.
    - The user may now see the post of the user in their home feed.
    - The other user has an updated number of followers.
- Main Success Scenario:
    - BF.1 The Logged-in User is viewing another user's profile.
    - BF.2 The user clicks the "Follow" button.
    - BF.3 The system updates the follower/following relationships.
    - BF.4 The system updates the number of followers of the user.
    - BF.5 The system updates the home feed of the user if necessary.
    - BF.5 The system displays the updated status (following).

Plant uml code of class diagram after refactoring and application of design patterns:

```
@startuml
class AuthenticationController {
    - credDB : Database<User>
    - userDB : Database<User>
    - profilePicDB : Database<ProfilePhotoData>
    - authService : IAuthenticationService
    + AuthenticationController(credDB:Database<User>, userDB:Database<User>,
profilePicDB:Database<ProfilePhotoData>, authService:IAuthenticationService)
    + AuthenticationController()
    + verifyCredentials(username:String, password:String) : VerificationResult
    + saveProfilePicture(file:File, username:String) : void
    + saveCredentials(username:String, password:String, bio:String) : void
    + doesUsernameExist(username:String) : boolean
    - userNamePredicate(t:User, username:String) : boolean
}
AuthenticationController -- IAuthenticationService
AuthenticationController -- Database
AuthenticationController --> User : uses
AuthenticationController --> ProfilePhotoData : uses
AuthenticationController --> VerificationResult : passes on

class AuthenticationService implements IAuthenticationService{
    - credDB : Database<User>
    - userDB : Database<User>
    + AuthenticationService(credDB:Database<User>, userDB:Database<User>)
    + registerUser(username:String, bio:String, password:String) : void
    + loginUser(username:String, password:String) : VerificationResult
    - saveUserInformation(user:User) : void
    - credentialsPredicate(t:User, username:String, password:String) : boolean
}
```

```
AuthenticationService -- Database
AuthenticationService --> User : uses
AuthenticationService --> VerificationResult : generates

class AuthenticationView extends JFrame{
    - WIDTH : int
    - HEIGHT : int
    # controller : AuthenticationController
    + AuthenticationView()
    - initializeUI() : void
    # {abstract} createInfoFormComponents() : Component[]
    # {abstract} createActionButton() : JButton
    # {abstract} createButtonToNavigateToAlternativeAuthenticationView() : JButton
}
AuthenticationView -- AuthenticationController : delegates to
abstract class Database<Model> {
    - models : List<Model>
    + Database()
    # {abstract} retrieveAll() : List<Model>
    + {abstract} save(m:Model) : boolean
    + getMatching(filter:Predicate<Model>) : List<Model>
    # {abstract} loadPath() : void
}

class CredentialsDatabase extends Database{
    - credentialsFilePath : String
    + CredentialsDatabase()
    + retrieveAll() : List<User>
    + save(u:User) : boolean
    # loadPath() : void
}

CredentialsDatabase --> User : generates

class ExploreController {
    - uploadedImageDataBase : Database<SimplePicture>
    - detailedImagDatabase : Database<Picture>
    - credDB : Database<User>
    + ExploreController(credDb:Database<User>, uploadedImageDb:Database<SimplePicture>,
detailedImageDb:Database<Picture>, usersDb:Database<User>)
    + ExploreController()
    + getAllUploadedImages(username:String) : List<SimplePicture>
    + getPictureWithId(imageId:String) : Picture
    + doesUsernameExist(username:String) : User
    - picIDMatchingPredicate(t:Picture, imageId:String) : boolean
    - isUsernamePartiallyMatching(searchFieldText:String, other:String) : boolean
    - userNameMatchingPredicate(t:User, other:String) : boolean
}
ExploreController -- Database
ExploreController --> SimplePicture : uses
ExploreController --> Picture : uses
ExploreController --> User : uses

class ExploreUI extends TabViewWithHeader{
    - IMAGE_SIZE : int
    - controller : ExploreController
    + ExploreUI()
    - createMainContentPanel() : JPanel
    - displayImage(imageID:String) : void
    # createContentPanel() : JPanel
    - updateImageGrid(username:String, imageGridPanel:JPanel) : void
}
ExploreUI -- ExploreController : delegates to
ExploreUI --> SimplePicture : uses
ExploreUI --> Picture : uses
ExploreUI --> User : uses

class HomeController {
    - relationshipManager : UserRelationshipManager
    - picDatabase : UpdatableDatabase<Picture>
    - notificationsDB : Database<Notification>
    + HomeController(picDatabase:UpdatableDatabase<Picture>,
relationshipManager:UserRelationshipManager, notificationsDB:Database<Notification>,
usersDB:Database<User>)
    + HomeController()
    + getFollowing(username:String) : List<String>
    + createSampleData() : List<Picture>
    + getPictureWithId(imageId:String) : Picture
    + handleLikeAction(imageId:String, likesLabel:JLabel) : void
    - picPosterMatchingPredicate(t:Picture, poster:String) : boolean
```

```
    - picIDMatchingPredicate(t:Picture, imageId:String) : boolean
}
HomeController -- UserRelationshipManager
HomeController -- UpdatableDatabase
HomeController -- Database
HomeController --> Picture : uses
HomeController --> Notification : uses
HomeController --> User : uses

interface IAuthenticationService {
    + registerUser(username:String, bio:String, password:String) : void
    + loginUser(username:String, password:String) : VerificationResult
}

class ImageUploadController {
    - picDatabase : Database<Picture>
    - uploadedDatabase : Database<SimplePicture>
    + ImageUploadController(picDatabase:Database<Picture>, uploadedDatabase:Database<SimplePicture>,
usersDB:Database<User>)
    + ImageUploadController()
    + uploadAction(bioTextAreaText:String) : ImageIcon
    - saveImageInfo(imageId:String, username:String, bio:String) : void
}
ImageUploadController -- Database
ImageUploadController --> Picture : uses
ImageUploadController --> SimplePicture : uses
ImageUploadController --> User : uses

class ImageUploadUI extends TabViewWithHeader {
    - imagePreviewLabel : JLabel
    - bioTextArea : JTextArea
    - uploadButton : JButton
    - saveButton : JButton
    - controller : ImageUploadController
    + ImageUploadUI()
    + uploadAction(e:ActionEvent) : void
    - saveBioAction(event:ActionEvent) : void
    # createContentPanel() : JPanel
}
ImageUploadUI -- ImageUploadController : delegates to

interface INavButtonCreator{
    + {abstract} createNavButtons : List<TabViewNavigationButton>
}

class InstagramProfileUI extends TabView {
    - PROFILE_IMAGE_SIZE : int
    - GRID_IMAGE_SIZE : int
    - imagePanel : JPanel
    - upperPanel : JPanel
    - controller : ProfileController
    - followersCountLabel : JLabel
    + InstagramProfileUI(user:User)
    + InstagramProfileUI()
    - createHeaderPanel() : JPanel
    - initializeImageGrid() : void
    - displayImage(imageIcon:ImageIcon) : void
    - createStatLabel(number:String, text:String) : JLabel
    - createStatLabelText(number:String, text:String) : String
    # createContentPanel() : JPanel
}
InstagramProfileUI -- ProfileController : delegates to
InstagramProfileUI --> User : uses


class NavigationBar {
    - creator : INavButtonCreator
    + NavigationBar(creator:INavButtonCreator)
    + displayAllActions() : void
}
NavigationBar -- INavButtonCreator : delegates button creation to
NavigationBar --> TabViewNavigationButton : displays

class Notification {
    - recieverName : String
    - transmitterName : String
    - timeStamp : String
    - imageID : String
    + Notification(recieverName:String, transmitterName:String, timeStamp:String)
    + Notification(recieverName:String, transmitterName:String, timeStamp:String, imageID:String)
    + getRecieverName() : String
```

```
    + getImageID() : String
    + getTimeStamp() : String
    + getTransmitterName() : String
    + setRecieverName(recieverName:String) : void
    + setTimeStamp(timeStamp:String) : void
    + setTransmitterName(transmitterName:String) : void
}

class NotificationsController extends TabViewController {
    - notificationDB : Database<Notification>
    + NotificationsController(notificationDB:Database<Notification>, usersDB:UsersDatabase)
    + NotificationsController()
    + getElapsedTime(timestamp:String) : String
    + getAllNotificationsMessagesForCurrentUser() : List<String>
    - getAllNotificationsForReciever(recieverUserName:String) : List<Notification>
    - recieverNameMatchPredicate(t:Notification, recieverUserName:String) : boolean
}
NotificationsController -- Database
NotificationsController --> Notification : uses

class NotificationsDatabase extends Database{
    + NotificationsDatabase()
    + retrieveAll() : List<Notification>
    + save(m: Notification) : boolean
    # loadPath() : void
}
NotificationsDatabase --> Notification : generates

class NotificationsUI extends TabViewWithHeader {
    - controller : NotificationsController
    + NotificationsUI()
    # createContentPanel() : JPanel
}
NotificationsUI -- NotificationsController : delegates to
NotificationsUI --> Notification : uses

class Picture extends SimplePicture{
    - caption : String
    - timeStamp : String
    - likesCount : int
    - comments : List<String>
    + Picture(imageID:String, imagePosterName:String, imagePath:String, caption:String,
timeStamp:String, likesCount:int)
    + addComment(comment:String) : void
    + like() : void
    + getCaption() : String
    + getLikesCount() : int
    + getComments() : List<String>
    + getLikeLabel() : String
    + setLikesCount(likesCount:int) : void
    + getDataString() : String
    + getTimeStamp() : String
}

class PicturesDatabase extends UpdatableDatabase{
    - databasePath : Path
    + PicturesDatabase()
    + retrieveAll() : List<Picture>
    + update(m:Picture) : boolean
    # loadPath() : void
    + save(m:Picture) : boolean
}
PicturesDatabase --> Picture : generates

class ProfileController {
    - relationshipManager : UserRelationshipManager
    - credDB : Database<User>
    - picDatabase : Database<Picture>
    - currentUser : User
    - uploadedPicsDB : Database<SimplePicture>
    - profilePicDB : Database<ProfilePhotoData>
    + ProfileController(user:User)
    + ProfileController(user:User, relationshipManager:UserRelationshipManager,
credDB:Database<User>, userDatabase:Database<User>, picDatabase:Database<Picture>,
uploadedPicsDB:Database<SimplePicture>, profilePicDB:Database<ProfilePhotoData>)
    + getUserWithCompleteData(user:User) : User
    + getCurrentlyViewedUser() : User
    + isCurrentUserLoggedInUser() : boolean
    + isCurrentUserFollowed() : boolean
    + handleFollowAction() : void
    + getProfileImages(user:User) : List<ImageIcon>
```

```
    + getCurrentlyViewedUserUsername() : String
    + getCurrentlyViewedUserProfilePicture() : ImageIcon
    - picPosterMatchingPredicate(t:Picture, posterName:String) : boolean
    - userNameMatchingPredicate(t:User, other:User) : boolean
}
ProfileController -- UserRelationshipManager
ProfileController -- Database
ProfileController --> User : uses
ProfileController --> Picture : uses
ProfileController --> SimplePicture : uses
ProfileController --> ProfilePhotoData : uses

class ProfilePhotoData {
    - file : File
    - username : String
    + ProfilePhotoData(f:File, username:String)
    + getFile() : File
    + getUsername() : String
    + setFile(file:File) : void
    + setUsername(username:String) : void
    + getProfilePicturePath() : String
}


class ProfilePhotoDatabase extends Database{
    - profilePhotoStoragePath : String
    + ProfilePhotoDatabase()
    + retrieveAll() : List<ProfilePhotoData>
    + save(d:ProfilePhotoData) : boolean
    # loadPath() : void
}
ProfilePhotoDatabase --> ProfilePhotoData : generates

class QuakstagramHomeUI extends TabViewWithHeader{
    - IMAGE_WIDTH : int
    - IMAGE_HEIGHT : int
    - LIKE_BUTTON_COLOR : Color
    - cardLayout : CardLayout
    - cardPanel : JPanel
    - homePanel : JPanel
    - imageViewPanel : JPanel
    - controller : HomeController
    + QuakstagramHomeUI()
    - populateContentPanel(panel:JPanel, pictures:List<Picture>) : void
    - displayImage(p:Picture) : void
    - refreshDisplayImage(imageId:String) : void
    # createContentPanel() : JPanel
    - createLikeButton(imageId:String, likesLabel:JLabel, isDetailView:boolean) : JButton
    - loadCroppedImageIcon(imagePath:String, maxWidth:int, maxHeight:int) : ImageIcon
}
QuakstagramHomeUI -- HomeController : delegates to
QuakstagramHomeUI --> Picture : uses

class SignInUI extends AuthenticationView{
    - txtUsername : JTextField
    - txtPassword : JTextField
    - btnSignIn : JButton
    - btnRegisterNow : JButton
    + SignInUI()
    - onSignInClicked(event:ActionEvent) : void
    - onRegisterNowClicked(event:ActionEvent) : void
    + main(args:String[]) : void
    # createActionButton() : JButton
    # createButtonToNavigateToAlternativeAuthenticationView() : JButton
    # createInfoFormComponents() : Component[]
}

class SignUpUI extends AuthenticationView{
    - txtUsername : JTextField
    - txtPassword : JTextField
    - txtBio : JTextField
    - btnRegister : JButton
    - btnUploadPhoto : JButton
    - btnSignIn : JButton
    + SignUpUI()
    - onRegisterClicked(event:ActionEvent) : void
    - handleProfilePictureUpload() : void
    - openSignInUI() : void
    + createInfoFormComponents() : Component[]
    + createActionButton() : JButton
    + createButtonToNavigateToAlternativeAuthenticationView() : JButton
```

```
}

class SimplePicture {
    - imagePosterName: String
    - imagePath: String
    - imageID: String
    - originPath: String
    + SimplePicture(poster: String, path: String, imageID: String)
    + getImagePath(): String
    + getImagePosterName(): String
    + setImagePath(imagePath: String): void
    + setImagePosterName(imagePosterName: String): void
    + getImageID(): String
    + setImageID(imageID: String): void
    + setOriginPath(originPath: String): void
    + getOriginPath(): String
}

class TabView extends JFrame implements INavButtonCreator{
    - WIDTH : int
    - HEIGHT : int
    - controller : TabViewController
    + TabView(windowTitle : String)
    + showContentPanel() : void
    + redrawCommonComponents() : void
    + removeScreenContents() : void
    + addCustomComponentNorth(c : Component) : void
    + addCustomComponentCenter(c : Component) : void
    + addCustomComponentSouth(c : Component) : void
    - executeProfileUIAction() : void
    - executeNotificationsUIAction() : void
    - executeImageUploadUIAction() : void
    - executeHomeUIAction() : void
    - executeExploreUIAction() : void
    + createNavButtons() : List<TabViewNavigationButton>
    + createContentPanel() : JPanel {abstract}
}
TabView -- TabViewController : delegates to
TabView --> TabViewNavigationButton : generates

class TabViewController {
    - userDatabase : Database<User>
    + TabViewController(userDatabase:Database<User>)
    + getLoggedInUserName() : String
    + getLoggedInUser() : User
}
TabViewController -- Database
TabViewController --> User : uses

class TabViewNavigationButton extends JButton{
    - final {static} NAV_ICON_SIZE : int
    + TabViewNavigationButton(iconPath : String, action : Runnable)
}

class TabViewWithHeader extends TabView{
    - headerTitle : String
    + TabViewWithHeader(headerTitle:String, windowTitle:String)
    - createHeaderPanel(headerTitle:String) : JPanel
    + redrawCommonComponents() : void
}

class UpdatableDatabase<Model> extends Database {
    + UpdatableDatabase()
    + {abstract} update(model:Model) : boolean
}

class UploadedImagesDatabase extends Database{
    - imageDir : Path
    + retrieveAll() : List<SimplePicture>
    + save(m:SimplePicture) : boolean
    - getFileExtension(path:String) : String
    # loadPath() : void
    - getNextImageId(username:String) : int
}
UploadedImagesDatabase --> SimplePicture : generates

class User {
    - username : String
    - bio : String
    - password : String
    - postsCount : int
```

```
    - followersCount : int
    - followingCount : int
    - pictures : List<Picture>
    + User(username:String, bio:String, password:String)
    + User(username:String)
    + addPicture(picture:Picture) : void
    + getUsername() : String
    + getBio() : String
    + setBio(bio:String) : void
    + getPostsCount() : int
    + getFollowersCount() : int
    + getFollowingCount() : int
    + getPictures() : List<Picture>
    + setFollowersCount(followersCount:int) : void
    + setFollowingCount(followingCount:int) : void
    + setPostCount(postCount:int) : void
    + toString() : String
    + getPassword() : String
}

class UserRelationshipManager {
    - followersFilePath : String
    + followUser(follower:String, followed:String) : void
    + isAlreadyFollowing(follower:String, followed:String) : boolean
    + getFollowers(username:String) : List<String>
    + getFollowing(username:String) : List<String>
}

class UsersDatabase extends Database{
    - usersFilePath : String
    + retrieveAll() : List<User>
    + save(u:User) : boolean
    + loadPath() : void
}

UsersDatabase --> User : generates

class VerificationResult {
    - success : boolean
    - newUser : User
    + VerificationResult(success:boolean, newUser:User)
    + getNewUser() : User
    + getSuccess() : boolean
    + setNewUser(newUser:User) : void
    + setSuccess(success:boolean) : void
}


@enduml
```