

project:

This project will focus on creating a platform and social network where users can share their thoughts, news, updates and multimedia content in a fast and direct way. In this application, users will be able to participate in each other's conversations.

objectives:

Among the objectives we have for this application are:

- to provide a simple to understand interface
- basic user information should be able to be entered: name, age, profile picture, etc.
- Possibility to publish texts of up to 280 characters, links and images.
- Have a button with which users can react to each other's content.
- Have a main feed where you can see the content of other users.

System Requirements:

-Functional Requirements:

- The system must allow user registration using a decoupled authentication interface.
- Users must be able to create and publish content through modular and extensible content classes.
- A user can react to posts using a dedicated reaction module that implements interface-based communication.

-Non-Functional Requirements:

- The application must follow SOLID principles to ensure modularity, testability, and future extension without modifying existing code.
- All components must be loosely coupled and follow the single responsibility principle to simplify maintenance.
- The system must be extensible, allowing new post types or reaction types to be added without breaking existing functionality.
- Sensitive user data must be managed through secure encapsulation.
- The system must offer quick performance and scale easily, supported by well-structured and abstracted logic.

User stories:

Title:	Priority:	Estimate:
User register	HIGH	
User Story: As a user, I want to register easily using my email or social networks, so that the process is quick and convenient.		
<ul style="list-style-type: none">•The system provides a decoupled authentication interface that allows sign-up via email or social media.•The system validates user data and redirects to their profile upon successful registration.•Validation logic is separated from the authentication controller		

Title: User post	Priority: HIGH	Estimate:
----------------------------	--------------------------	------------------

User Story:

As a user

I want to be able to post a tweet with text, images, and links, So that my followers can see what I want to share and participate in the conversation.

- The user can write a tweet of up to 280 characters.
- Can attach images, links, or videos to the tweet.
- System displays a preview of the tweet before publishing it.
- Content is managed using a base class (Post) and specific subclasses to ensure extensibility (Open/Closed + Liskov Substitution Principles).

Title: Interacción with post	Priority: MEDIUM	Estimate:
--	----------------------------	------------------

User Story:

As a user

I want to be able to "like" other users' content, So that I can show my support or share interesting content with my followers.

- The user can click on a "like" icon to mark as a favorite.
- Changes in the number of "likes" are reflected in real time.
- Reactions are handled by a dedicated class implementing a Reaction interface (Interface Segregation + SRP).

Title: Follow other users	Priority: LOW	Estimate:
-------------------------------------	-------------------------	------------------

User Story:

As a user

I want to be able to follow other users to see their content in my feed, so I can keep up with their posts and participate in their conversations.

Acceptance Criteria:

- User can search and follow other users from their profile.
- Users can stop following others at any time.
- The user-follow relationship is managed by a specific class (FollowManager) to ensure modularity.

Title: Feed Viewing	Priority: High	Estimate:
-------------------------------	--------------------------	------------------

As a user,

I want to see a feed with posts from users I follow, so that I can stay updated and interact with relevant content.

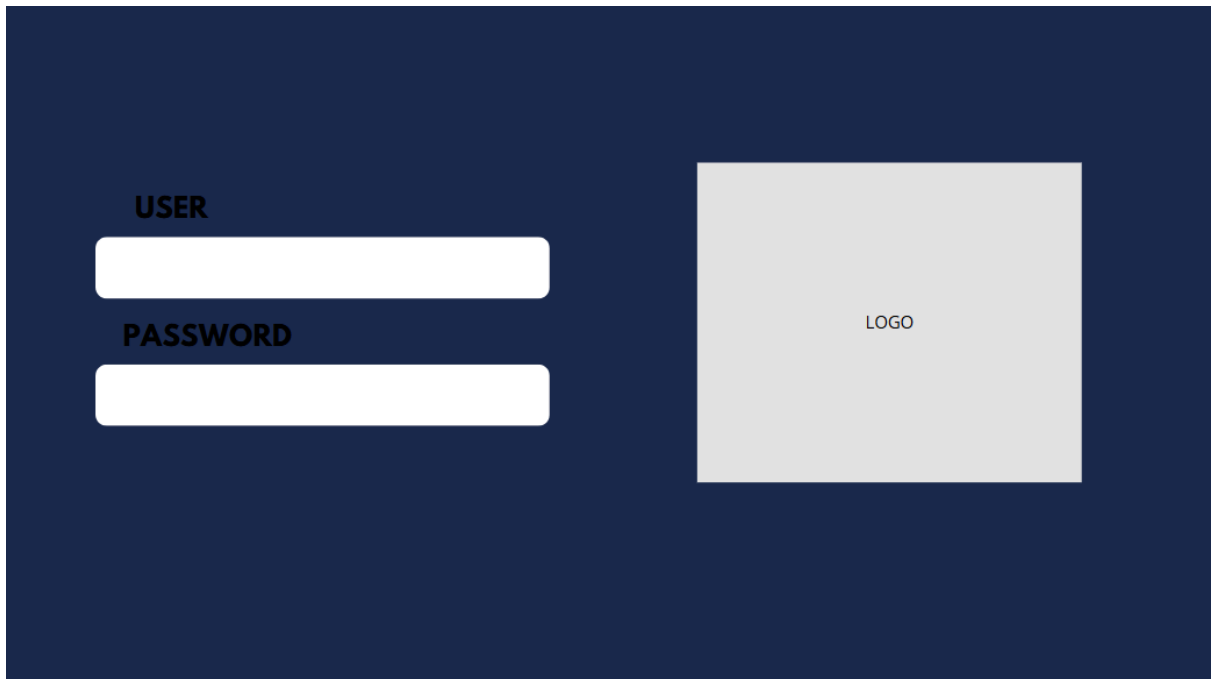
- The user can view a feed displaying posts from followed users.
- Posts are sorted by date or relevance using sorting strategies.
- The feed display logic is handled through an abstract interface allowing future customization.

mockups:

home:



log in:



Sing up:

USER

PASSWORD

EMAIL

AGE

LOGO

Feed:

LOGO

CRC Cards:

Class USER	
Responsibility <ul style="list-style-type: none">•Store user data (name, email, age, profile picture)•Create new posts or replies•View personalized feed•Follow and unfollow other users•React to other users' posts•Maintain tweet history	Collaborator <ul style="list-style-type: none">•Tuit (interface for Post/Reply)•Feed•FollowManager•Reaction•App

Class POST	
Responsibility	Collaborator
<ul style="list-style-type: none"> •Represent a main post •Store text, media, links •Provide publication preview •Track publication date and author 	<ul style="list-style-type: none"> •User •Reaction

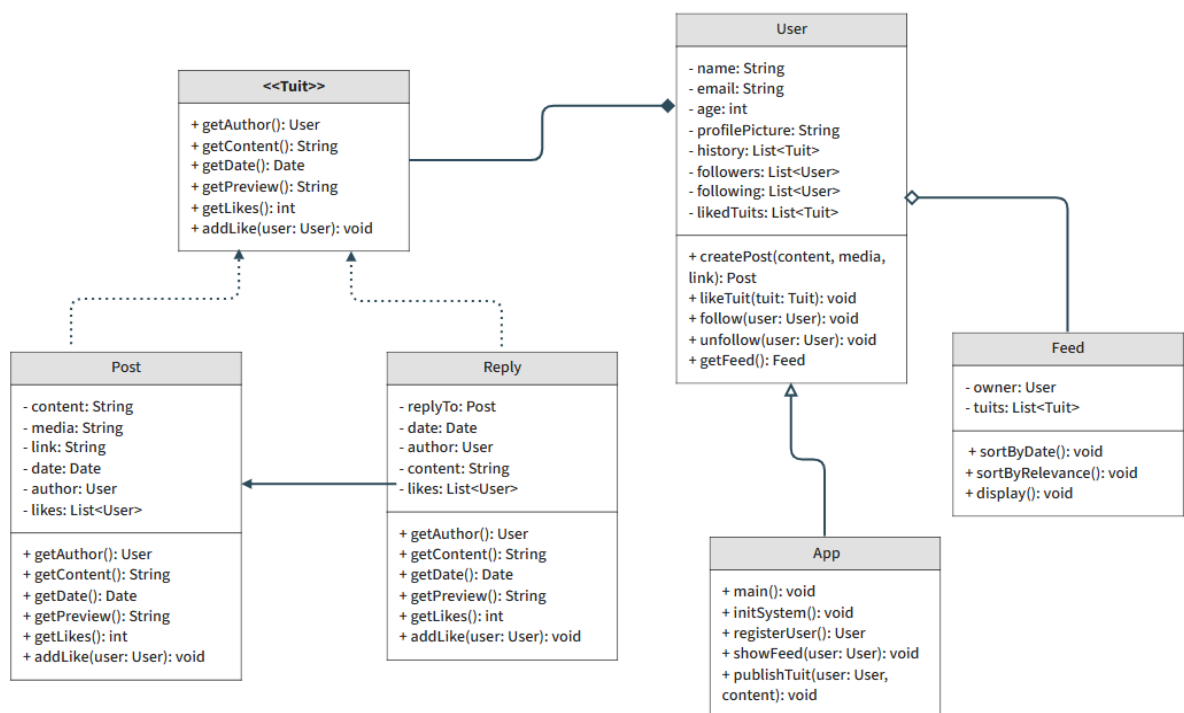
Interface: Tuit	
Responsibility	Collaborator
<ul style="list-style-type: none"> •Define shared methods for posts and replies •Provide abstract behavior: getAuthor(), getContent(), getDate(), getPreview() 	<ul style="list-style-type: none"> •User •Reaction

Class FEED	
Responsibility	Collaborator
<ul style="list-style-type: none"> •Display list of posts or replies from followed users •Sort by relevance or date using strategy •Filter content by interests or hashtags 	<ul style="list-style-type: none"> •User •Tuit •FeedStrategy

Class Reply	
Responsibility	Collaborator
<ul style="list-style-type: none"> •Represent a reply to a post •Maintain link to original Post •Inherit shared post behavior •Override methods like getPreview() if needed 	<ul style="list-style-type: none"> •User •Post

Class App	
Responsibility	Collaborator
<ul style="list-style-type: none"> •Initialize and coordinate system •Manage access to users, feed, post creation •Interface with controllers and views 	<ul style="list-style-type: none"> •User •Post, Reply •Feed

UML Diagrams:



-User

Encapsulation: All user attributes such as name, email, age, and profilePicture are declared as private, ensuring that they cannot be modified directly from outside the class. Interactions such as following other users or liking content are handled through dedicated public methods like `follow(User user)` or `likeTuit(Tuit tuit)`, which control the internal state safely.

Abstraction: The logic of operations like following users or getting a personalized feed is hidden behind methods like `getFeed()` and `follow(User user)`. The internal handling of the followers, following, and history lists is abstracted away, exposing only high-level interactions.

Polymorphism: While **User** is not extended by other classes in the current model, it interacts polymorphically with objects of type **Tuit**, which could be either **Post** or **Reply**.

SOLID Applied:

Single Responsibility: The class focuses solely on managing user identity and user

actions.

Dependency Inversion: The user depends on the Tuit abstraction, allowing flexibility in interacting with both Post and Reply.

-Post

Encapsulation: Attributes like author, date, likes, and content are private and accessed or modified only through methods such as `getAuthor()` and `addLike(User user)`, which maintain control over how likes are processed.

Abstraction: The method `getPreview()` abstracts the formatting or rendering logic of the post, allowing the user to see a preview without knowing how it's generated.

Inheritance: Post implements the Tuit interface, making it one of the possible forms of a tuit. This allows flexibility in the system to interact with general tuits without needing to distinguish between posts and replies.

Polymorphism: Through the Tuit interface, a Post can be treated as a generic tuit. This supports substitutability when a list of Tuit is processed by the Feed.

SOLID Applied:

Open/Closed: The class is open to extension through the Reply class but closed to modification of existing logic.

Liskov Substitution: Post can be used wherever a Tuit is expected without breaking the system.

-Reply

Encapsulation: The `replyTo` attribute is private and only used internally to link to a Post. The method `addLike()` ensures that user reactions are controlled and consistent.

Abstraction: Like Post, the method `getPreview()` can be overridden to customize how a reply is rendered, without exposing internal details.

Inheritance / Interface Implementation: Reply also implements the Tuit interface, enabling polymorphic behavior when being managed by User, Feed, or App.

Polymorphism: When a User likes or views a Tuit, it can be either a Post or Reply, and the logic works the same due to polymorphism.

SOLID Applied:

Liskov Substitution: A Reply behaves correctly as a Tuit.

Single Responsibility: Reply focuses only on handling responses to posts.

-Feed

Encapsulation: The internal list of Tuit objects (tuits) is private. External classes interact with the feed through methods like `sortByDate()` or `display()`, without accessing or modifying the list directly.

Abstraction: The user sees a simplified view of the feed through high-level methods like `sortByRelevance()` and `display()`, which abstract the complexity of sorting and filtering.

Polymorphism: The feed works with the Tuit interface, allowing it to treat both Post and Reply objects the same. This supports flexible and extensible feed behavior.

SOLID Applied:

Interface Segregation: The feed only uses methods from the Tuit interface that are relevant for displaying and sorting content.

Open/Closed: Future sorting strategies can be added (e.g., by topic or popularity) without modifying the core logic.

-App

Encapsulation: Central application logic is wrapped in public methods like `main()`, `initSystem()` or `publishTuit()`, ensuring structured flow and controlled interactions between users, posts, and feeds.

Abstraction: Acts as the facade for initializing and coordinating components, without exposing implementation details of User, Feed, or Tuit.

SOLID Applied:

Dependency Inversion: The app works with the Tuit abstraction and delegates actions to other classes, without depending on specific implementations.

Relations:

User to Tuit: Each user can create tuits, including posts and replies. These tuits are stored in the user's history list. This is a composition relationship, meaning the tuits depend on the user's existence. It aligns with the Single Responsibility Principle by allowing the user class to manage its own content and supports the Open/Closed Principle by enabling extension without modifying existing structures.

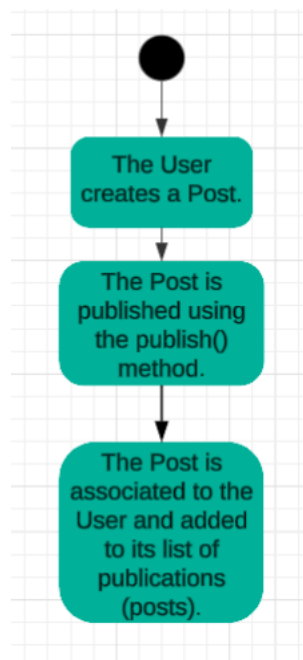
Reply to Post: A reply references the original post through a simple association. The reply depends on the post logically but does not own it. This allows replies to remain valid even if the original post is removed from view. It respects the Single Responsibility Principle by giving the reply class a focused purpose.

Feed to Tuit: The feed aggregates a collection of tuits (posts and replies) for display. This is an aggregation, meaning the feed organizes the tuits but does not own them. The feed uses methods like `sortByDate` and `sortByRelevance`, applying abstraction and interface segregation by interacting only with the relevant methods of the Tuit interface.

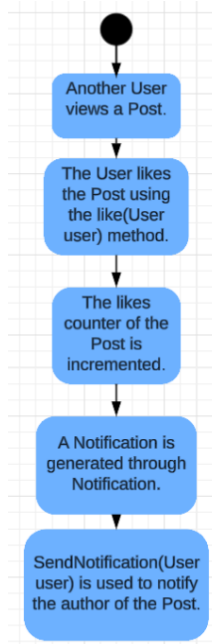
Feed to User: Each feed belongs to a specific user, indicated by the owner attribute. This aggregation ensures content personalization. It helps isolate feed behavior while maintaining a clear connection to the user, following Dependency Inversion by depending on abstractions.

App to User: The App class coordinates the system, using associations to User, Tuit, and Feed. It manages user registration, content publishing, and feed display. These are usage relationships that support Dependency Inversion by allowing App to interact with abstractions rather than concrete implementations.

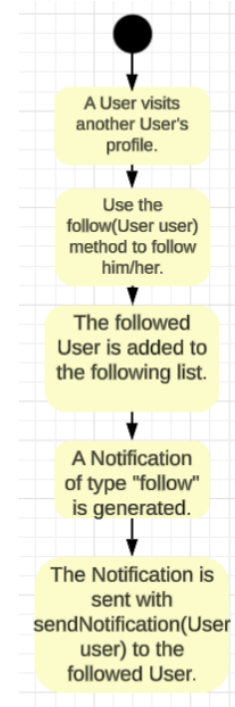
post publication:



Interaction with "Like" on a Post:



Follow between Users:



Implementation Plan for OOP Concepts:

-Encapsulation will be implemented using access modifiers, along with getters and setters:

Sensitive attributes such as password, email, and likes will be private or protected. Public methods (getEmail(), setProfilePicture(), getLikes()) will be provided to access/control their modification.

-Inheritance will be used to create hierarchies between general classes and their variants:

Post will be a base class for subclasses such as ImagePost, VideoPost. Notification will be a superclass for LikeNotification, FollowNotification and others.

-Abstraction:

Methods such as publish() or sendNotification() hide the details of how content is stored or how a notification is sent. The user or programmer only needs to know what it does, not how it does it. Classes can be defined as abstract (Post or Notification) if they are not to be instantiated directly, but only through their subclasses (ImagePost, LikeNotification, etc.).

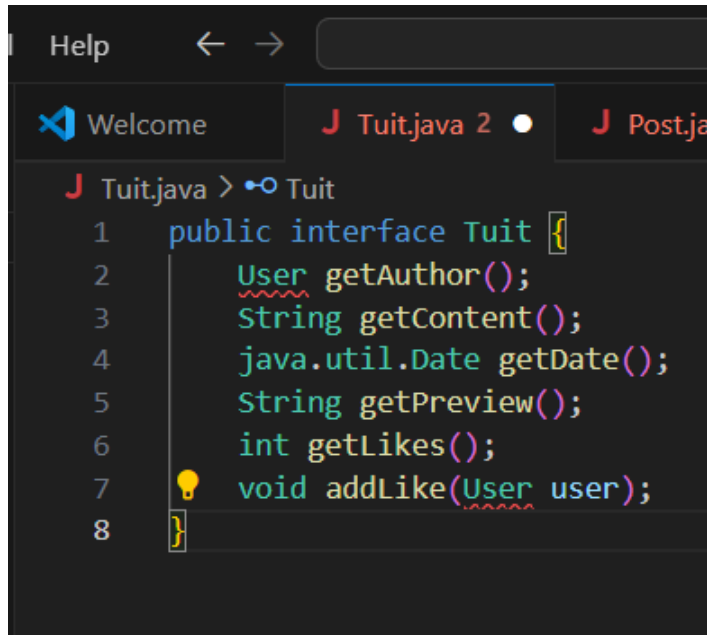
-polyformism:

the app has different types of notifications, "like" "new follower", each one can have its own way of sending or displaying the notification. The system simply uses the

sendNotification() method on any notification type, and each subclass executes it in its own way.

preliminary directory structure:

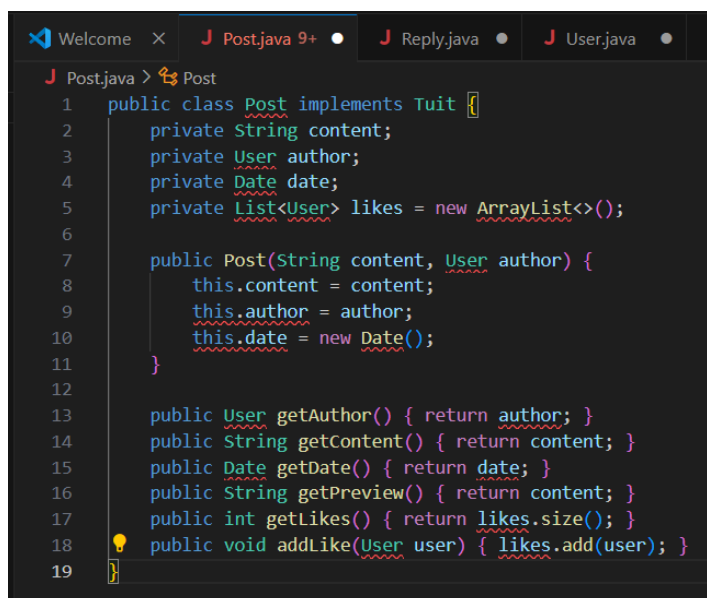
Progress Code and SOLID change:



```
Help  <  >
Welcome  J Tuit.java 2  J Post.java

J Tuit.java > Tuit
1  public interface Tuit {
2      User getAuthor();
3      String getContent();
4      java.util.Date getDate();
5      String getPreview();
6      int getLikes();
7      void addLike(User user);
8  }
```

The interface Tuit defines a common contract for all types of content, such as posts and replies. It includes methods like getAuthor(), getContent(), getDate(), getPreview(), getLikes(), and addLike(User user). This promotes the Interface Segregation Principle by exposing only the essential behaviors needed for content objects. It also supports the Liskov Substitution Principle, since any class implementing Tuit can be used interchangeably.



```
Welcome  X  J Post.java 9+  J Reply.java  J User.java

J Post.java > Post
1  public class Post implements Tuit {
2      private String content;
3      private User author;
4      private Date date;
5      private List<User> likes = new ArrayList<>();
6
7      public Post(String content, User author) {
8          this.content = content;
9          this.author = author;
10         this.date = new Date();
11     }
12
13     public User getAuthor() { return author; }
14     public String getContent() { return content; }
15     public Date getDate() { return date; }
16     public String getPreview() { return content; }
17     public int getLikes() { return likes.size(); }
18     public void addLike(User user) { likes.add(user); }
19 }
```

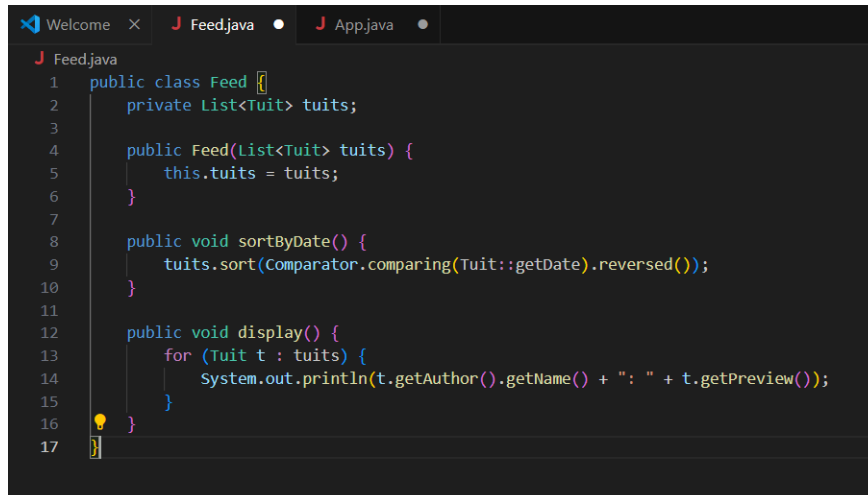
The Post class implements Tuit and represents a standard publication. It encapsulates its internal state using private attributes like content, media, link, date, author, and a list of users who liked the post. Public methods allow controlled access to this data and enforce consistent logic, such as limiting how likes are added. This follows the Single Responsibility Principle by focusing only on post behavior and the Open/Closed Principle by allowing future extensions without modifying the class.

```
1 public class Reply implements Tuit {
2     private Post replyTo;
3     private String content;
4     private User author;
5     private Date date;
6     private List<User> likes = new ArrayList<>();
7
8     public Reply(Post replyTo, String content, User author) {
9         this.replyTo = replyTo;
10        this.content = content;
11        this.author = author;
12        this.date = new Date();
13    }
14
15    public User getAuthor() { return author; }
16    public String getContent() { return content; }
17    public Date getDate() { return date; }
18    public String getPreview() { return "👁️ " + content; }
19    public int getLikes() { return likes.size(); }
20    public void addLike(User user) { likes.add(user); }
21 }
```

The Reply class also implements Tuit and represents a response to a post. It contains a reference to the post being replied to (replyTo), along with its own content, author, and like list. It overrides methods like getPreview() to customize the display for replies. This class complies with the Single Responsibility Principle by focusing solely on reply logic, and with the Open/Closed and Liskov Substitution Principles by extending the system through inheritance and behaving as a valid Tuit.

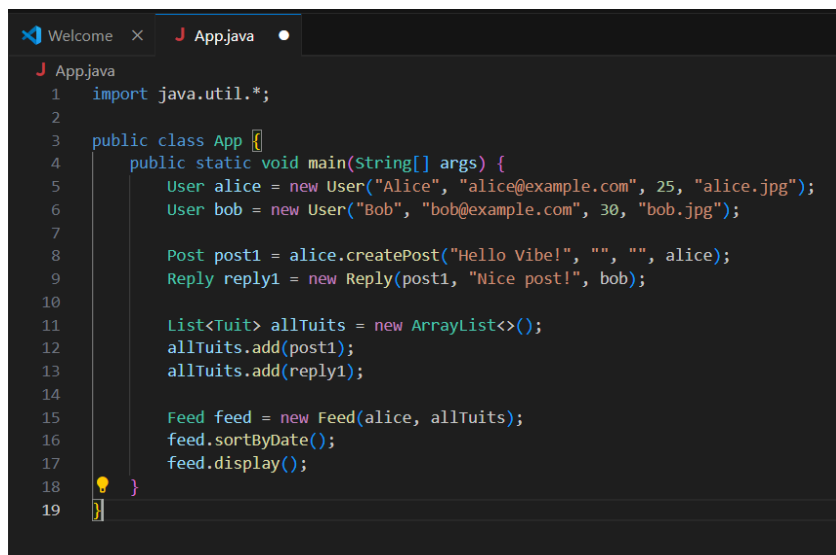
```
1 import java.util.*;
2
3 public class User {
4     private String name;
5     private String email;
6     private int age;
7     private String profilePicture;
8     private List<Tuit> history = new ArrayList<>();
9     private List<User> followers = new ArrayList<>();
10    private List<User> following = new ArrayList<>();
11
12    public User(String name, String email, int age, String profilePicture) {
13        this.name = name;
14        this.email = email;
15        this.age = age;
16        this.profilePicture = profilePicture;
17    }
18
19    public Post createPost(String content, String media, String link) {
20        Post post = new Post(content, media, link, this);
21        history.add(post);
22        return post;
23    }
24
25    public void likeTuit(Tuit tuit) {
26        tuit.addLike(this);
27    }
28
29    public void follow(User other) {
30        if (!following.contains(other)) {
31            following.add(other);
32            other.followers.add(this);
33        }
34    }
35
36    public void unfollow(User other) {
37        following.remove(other);
38    }
39 }
```

The User class represents platform users and manages personal data and actions like creating posts, following other users, and liking content. Attributes such as name, email, and profile picture are private and accessed only through methods. The user interacts with Tuit objects without knowing their specific implementation, which supports Dependency Inversion. The class adheres to the Single Responsibility Principle by handling only user-specific logic.



```
1 public class Feed {
2     private List<Tuit> tuits;
3
4     public Feed(List<Tuit> tuits) {
5         this.tuits = tuits;
6     }
7
8     public void sortByDate() {
9         tuits.sort(Comparator.comparing(Tuit::getDate).reversed());
10    }
11
12    public void display() {
13        for (Tuit t : tuits) {
14            System.out.println(t.getAuthor().getName() + ": " + t.getPreview());
15        }
16    }
17 }
```

The Feed class displays content to a user. It receives a list of Tuit objects and provides methods like sortByDate() and sortByRelevance() to organize them. Feed interacts with the Tuit interface, allowing it to treat posts and replies equally. This use of abstraction and interface-based design follows the Interface Segregation and Dependency Inversion Principles, and the structure supports the Open/Closed Principle by allowing new sorting strategies without altering existing code.



```
1 import java.util.*;
2
3 public class App {
4     public static void main(String[] args) {
5         User alice = new User("Alice", "alice@example.com", 25, "alice.jpg");
6         User bob = new User("Bob", "bob@example.com", 30, "bob.jpg");
7
8         Post post1 = alice.createPost("Hello Vibe!", "", "", alice);
9         Reply reply1 = new Reply(post1, "Nice post!", bob);
10
11         List<Tuit> allTuits = new ArrayList<>();
12         allTuits.add(post1);
13         allTuits.add(reply1);
14
15         Feed feed = new Feed(alice, allTuits);
16         feed.sortByDate();
17         feed.display();
18     }
19 }
```

The App class coordinates the system. It acts as the entry point with a main() method and uses the other classes to simulate user interactions. It does not depend on concrete implementations like Post or Reply, but works with the Tuit abstraction, fulfilling the Dependency Inversion Principle.

Layer and design review:

In developing the Vibe system, we used an MVC (Model–View–Controller) structure to clearly separate responsibilities and avoid making the parts of the system too dependent on each other.

The Model layer includes classes like *User*, *Post*, *Reply*, *Feed*, and the *Tuit* interface. These are in charge of handling data and the core rules of how the system works.

The View layer is handled by the *ConsoleView* class, which is responsible for showing information to the user (through the console in this case) without changing how the system works behind the scenes.

The Controller layer, represented by the *App* class, acts as a middleman between the model and the view. It takes care of creating objects, managing interactions between parts of the system, and showing results.

We reviewed the class diagrams and the way the code is organized to make sure each class focuses on one main task, following the Single Responsibility Principle from SOLID. We also checked that the classes aren't too tightly connected, using interfaces like *Tuit* when needed and avoiding unnecessary direct links. The design is considered solid and in line with clean architecture principles.

Swing-based graphical user interface (GUI) prototype:

We developed a functional prototype using the Java Swing library to visually represent the basic features of the Vibe system.

The graphical interface was designed with a focus on simplicity and usability, allowing the user to interact with the system clearly through organized tabs.

The prototype is divided into two main tabs, implemented using the *JTabbedPane* component:

Tab 1 – "Post": Allows the user to write a tweet of up to 280 characters and publish it.

Tab 2 – "Feed": Displays all previously published tweets by the user, simulating a feed similar to the original app.

The main components used in the prototype are:

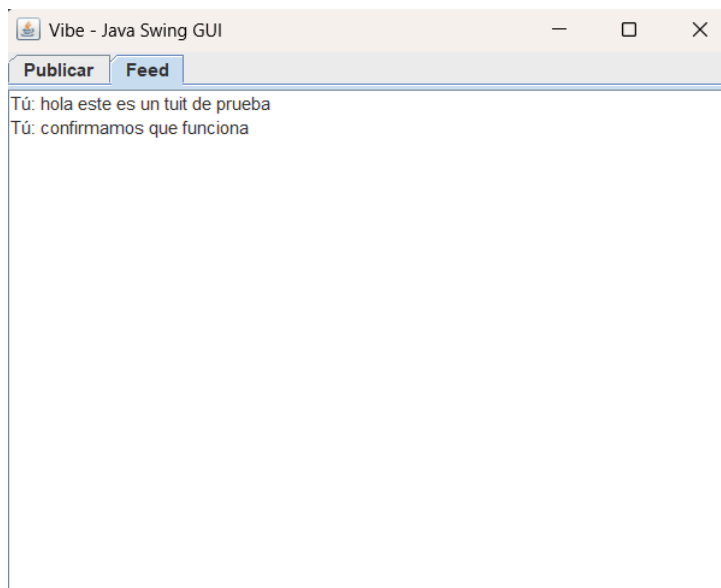
JTabbedPane was used to switch between the "Post" and "Feed" sections, making navigation simple and intuitive for the user.

TextArea served both as the input field where the user writes tweets and as the display area where published tweets appear in the feed.

Button allowed the user to publish a tweet once it was written.

JOptionPane was used to show popup messages for input validation, such as when a tweet was too long or left empty.

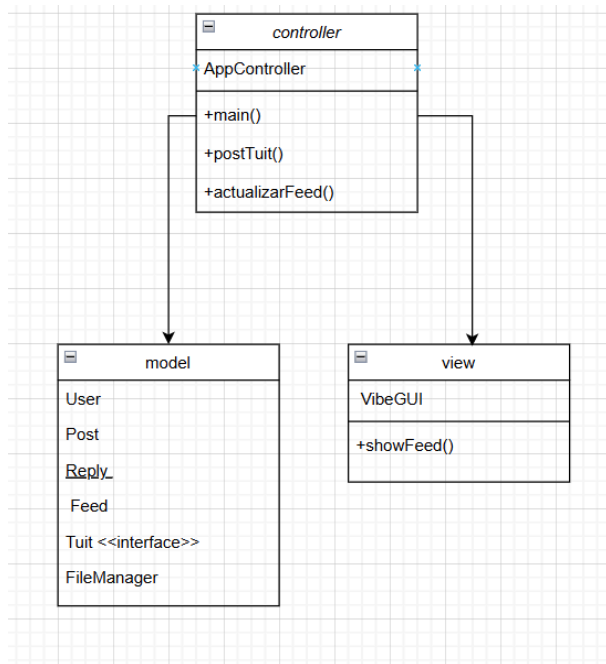
JScrollPane added scrollbars to the text areas, allowing users to scroll through longer content comfortably.



To store files, we used Java's serialization technique, which allows an object to be converted into a sequence of bytes that can be written to a file. Later, that file can be read to reconstruct the original object.

All core classes (User, Post, Reply, Feed) were modified to implement the Serializable interface, enabling their persistence. A utility class called FileManager was also created, containing two methods: guardar() and cargar(), which make it easy to save and retrieve any object.

UML diagram representing the relationship between layers (model, view, and controller):



Code examples GUI:

This code snippet is responsible for validating and posting a tweet in the application. Its main purpose is to ensure that the message written by the user is not empty and does not exceed the 280-character limit. If the content is valid, it adds the tweet to the list of posts, updates the feed to display the new tweet, and clears the text area to allow writing a new one. It also shows warning messages if the text does not meet the required conditions.

```
private void postTuit() {
    String content = inputArea.getText().trim();
    if (content.isEmpty()) {
        JOptionPane.showMessageDialog(this, message:"El tuit no puede estar vacío.");
        return;
    }
    if (content.length() > 280) {
        JOptionPane.showMessageDialog(this, message:"El tuit excede los 280 caracteres.");
        return;
    }

    posts.add(content);
    updateFeed();
    inputArea.setText("");
}
```

This code snippet is responsible for displaying the published tweets in the feed area and launching the application's graphical interface. It organizes the previously written messages so they appear on screen, and ensures that the interface loads correctly when the program runs.

```
private void updateFeed() {  
    StringBuilder feedContent = new StringBuilder();  
    for (String post : posts) {  
        feedContent.append(str: "Tú: ").append(post).append(str: "\n");  
    }  
    feedArea.setText(feedContent.toString());  
}
```

Run | Debug

```
public static void main(String[] args) {  
    SwingUtilities.invokeLater(() -> new VibeGUI());  
}
```

}