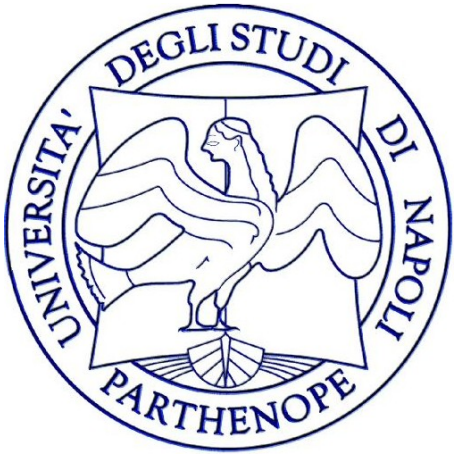
Università degli Studi di Napoli “Parthenope”

Facoltà di Scienze e Tecnologie

Corso di Laurea in Informatica



Relazione di Programmazione 3

Twitter

Candidato

Pierluigi Scotto

0124001277

Anno Accademico 2019/2020

Indice

Sommario

Introduzione....3
Requisiti....4
Amministratore....4
Utente....4
Diagramma UML....5
Design Pattern Mediator....7
Design Pattern Singleton....9
Codice....10
Design pattern Singleton....10
Design pattern Mediator....12
Twitter....15
Login....20
Registrazione....22
Admin....23

Introduzione

Il nome del progetto è **Twitter**.

Twitter è un servizio di social networking che fornisce agli utenti una pagina personale aggiornabile tramite messaggi di testo.

Ogni utente puo' avere un certo numero di ***follower*** che ricevono i suoi messaggi pubblicati.

I messaggi (chiamati ***tweet***) possono essere effettuati tramite il sito stesso, via SMS, con programmi di messaggistica istantanea e posta elettronica. Ogni messaggio puo' contenere un ***hashtag***. I messaggi contenenti lo stesso hashtag sono ***categorizzati insieme***.

I design pattern utilizzati in questo progetto sono il ***Singleton*** e il ***Mediator***.

Il design pattern ***Singleton*** permette di istanziare un'unica istanza del database.

Il design pattern ***Mediator*** permette di incapsulare il modo di interagire tra gli utenti.

Il linguaggio di programmazione utilizzato è ***Java*** con il ***JDK versione 8***.

Il database utilizzato è ***Sqlite***. Scaricare il driver sqlite-jdbc-"versione".jar e inserirlo nel progetto per un corretto utilizzo del database.

L'ambiente di sviluppo utilizzato per questo progetto è ***Intellij***.

Il software per la creazione dei diagramma UML delle classi è ***Intellij***.

Requisiti

Il sistema prevede l'accesso sia in modalità ***utente*** che in modalità ***amministratore***

Amministratore

L'amministratore effettua le seguenti operazioni

- mostrare l'elenco degli utenti in base al numero di messaggi ricevuti o inviati
- visualizzare i messaggi divisi in categorie in base agli hashtag
- data una parola, visualizzare tutti i messaggi dei diversi utenti che contengono quella parola

Utente

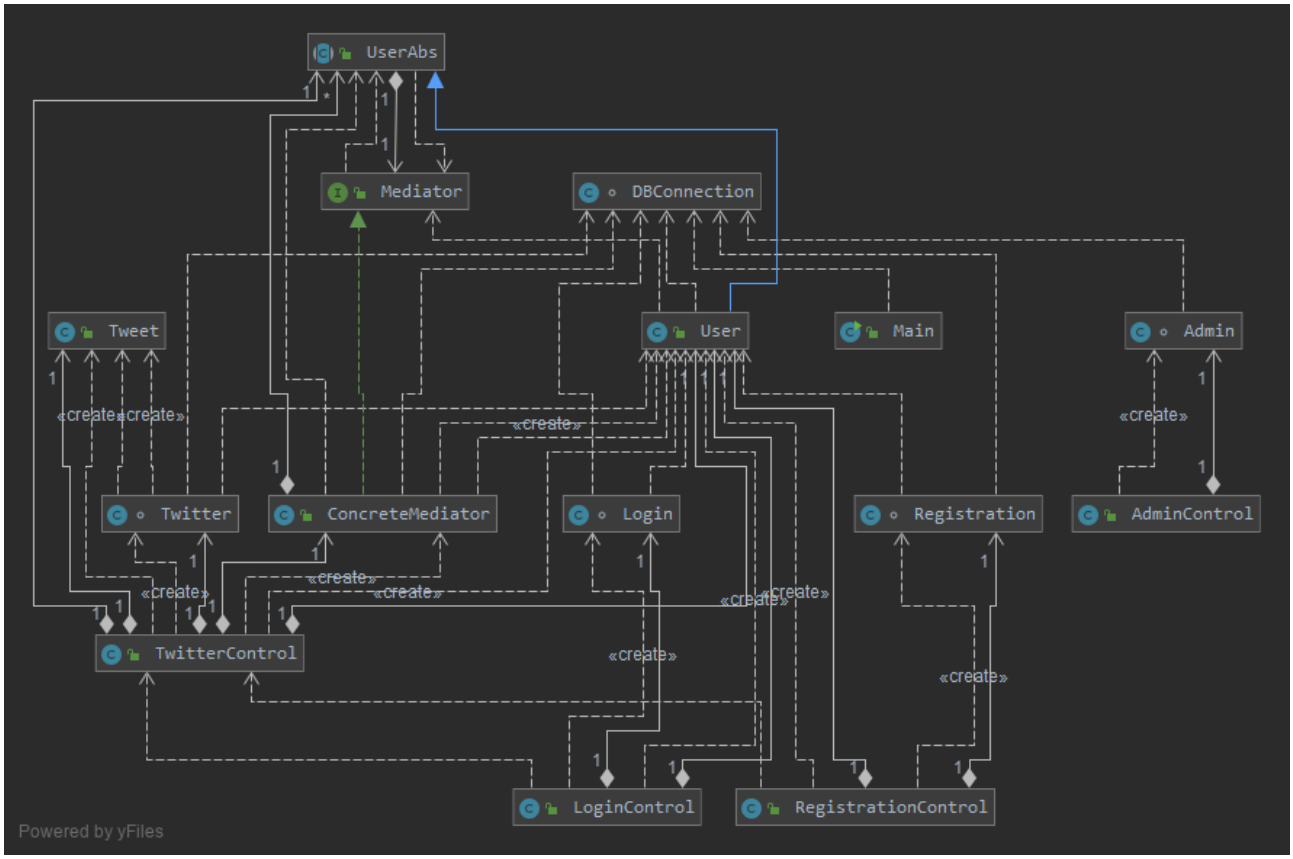
L'utente effettua le seguenti operazioni

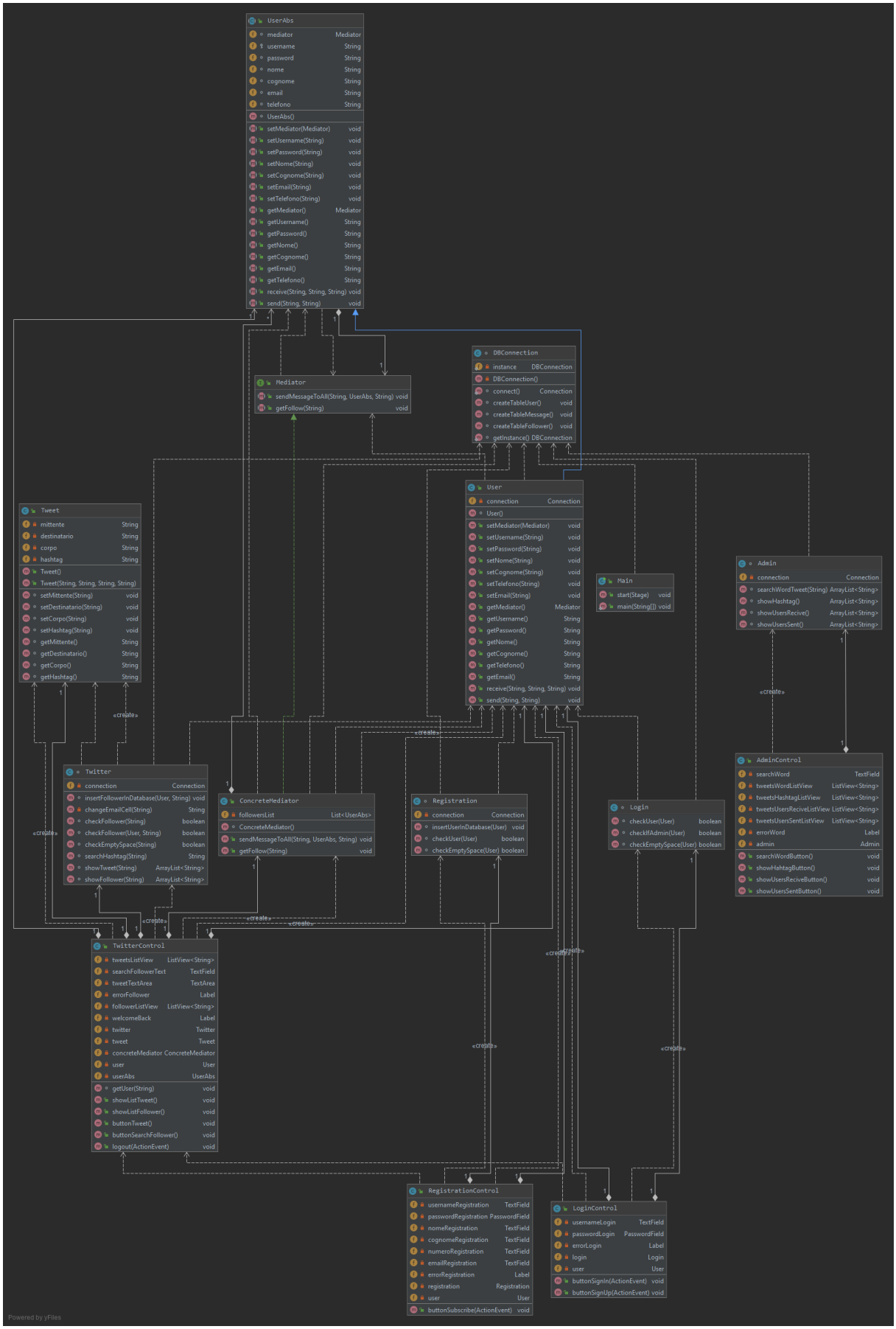
- registrarsi al servizio
- aggiungere un follower
- scrivere un messaggio (eventualmente contenente un hashtag). Il messaggio sarà visibile a tutti i follower
- visualizzare i follower (utente che segue altri utenti)
- visualizzare i following (utenti che stanno seguendo l'utente)

Diagramma UML

In questa sezione viene inserito il diagramma UML delle classi ed una piccola spiegazione di ogni classe.

- **UserAbs**: classe astratta dell'utente che possiede un riferimento al Mediator e definisce dei metodi di notifica ad esso.
- **Mediator**: definisce un'interfaccia di comunicazione da parte di UserAbs.
- **DBconnection**: stabilisce una connessione con il database SQLite e definisce dei metodi per la creazione delle tabelle
- **Tweet**: classe che contiene le informazioni del messaggio
- **User**: comunica gli eventi al Mediator.
- **Admin**: implementa le funzioni dell'admin
- **Twitter**: implementa le funzioni del sito Twitter
- **ConcreteMediator**: implementa il comportamento cooperativo da parte dei ***UserAbs***.
- **Login**: implementa le funzioni del login
- **Registration**: implementa le funzioni della registrazione
- **AdminControl**: implementa l'interfaccia grafica dell'admin
- **TwitterControl**: implementa l'interfaccia grafica di Twitter
- **LoginControl**: implementa l'interfaccia grafica del login
- **RegistrationControl**: implementa l'interfaccia grafica della registrazione




UserAbs
mediator Mediator
username String
password String
nome String
cognome String
email String
telefono String
UserAbs()
setMediator(Mediator) void
setUsername(String) void
setPassword(String) void
setNome(String) void
setCognome(String) void
setEmail(String) void
setTelefono(String) void
getMediator() Mediator
getUsername() String
getPassword() String
getNome() String
getCognome() String
getEmail() String
getTelefono() String
receive(String, String, String) void
send(String, String) void
Mediator
sendMessageToAll(String, UserAbs, String) void
getFollow(String) void
DBConnection
instance DBConnection
DBConnection()
connect() Connection
createTableUser() void
createTableMessage() void
createTableFollower() void
getInstance() DBConnection
User
connection Connection
User()
setMediator(Mediator) void
setUsername(String) void
setPassword(String) void
setNome(String) void
setCognome(String) void
setTelefono(String) void
setEmail(String) void
getMediator() Mediator
getUsername() String
getPassword() String
getNome() String
getCognome() String
getTelefono() String
getEmail() String
receive(String, String, String) void
send(String, String) void
Tweet
mittente String
destinatario String
corpo String
hashtag String
Tweet()
Tweet(String, String, String, String)
setMittente(String) void
setDestinatario(String) void
setCorpo(String) void
setHashtag(String) void
getMittente() String
getDestinatario() String
getCorpo() String
getHashtag() String
Main
start(Stage) void
main(String[]) void
Admin
connection Connection
searchWordTweet(String) ArrayList<String>
showHashtag() ArrayList<String>
showUsersRecive() ArrayList<String>
showUsersSent() ArrayList<String>
Twitter
connection Connection
insertFollowerInDatabase(User, String) void
changeEmailCell(String) String
checkFollower(String) boolean
checkFollower(User, String) boolean
checkEmptySpace(String) boolean
searchHashtag(String) String
showTweet(String) ArrayList<String>
showFollower(String) ArrayList<String>
ConcreteMediator
followersList List<UserAbs>
ConcreteMediator()
sendMessageToAll(String, UserAbs, String) void
getFollow(String) void
Registration
connection Connection
insertUserInDatabase(User) void
checkUser(User) boolean
checkEmptySpace(User) boolean
Login
checkUser(User) boolean
checkIfAdmin(User) boolean
checkEmptySpace(User) boolean
AdminControl
searchWord TextField
tweetsWordListView ListView<String>
tweetsHashtagListView ListView<String>
tweetsUsersReciveListView ListView<String>
tweetsUsersSentListView ListView<String>
errorWord Label
admin Admin
searchWordButton() void
showHahtagButton() void
showUsersReciveButton() void
showUsersSentButton() void
TwitterControl
tweetsListView ListView<String>
searchFollowerText TextField
tweetTextArea TextArea
errorFollower Label
followerListView ListView<String>
welcomeBack Label
twitter Twitter
tweet Tweet
concreteMediator ConcreteMediator
user User
userAbs UserAbs
getUser(String) void
showListTweet() void
showListFollower() void
buttonTweet() void
buttonSearchFollower() void
logout(ActionEvent) void
RegistrationControl
usernameRegistration TextField
passwordRegistration PasswordField
nomeRegistration TextField
cognomeRegistration TextField
numeroRegistration TextField
emailRegistration TextField
errorRegistration Label
registration Registration
user User
buttonSubscribe(ActionEvent) void
LoginControl
usernameLogin TextField
passwordLogin PasswordField
errorLogin Label
login Login
user User
buttonSignIn(ActionEvent) void
buttonSignUp(ActionEvent) void
«create»
Powered by yFiles

Design Pattern Mediator

In questa sezione vediamo l'UML delle classi del design pattern ***Mediator***.

I partecipanti sono:

- **UserAbs**: classe astratta dell'utente che possiede un riferimento al ***Mediator*** e definisce dei metodi di notifica ad esso.
- **Mediator**: definisce un'interfaccia di comunicazione da parte di ***UserAbs***.
- **ConcreteMediator**: implementa il comportamento cooperativo da parte dei ***UserAbs*** e possiede riferimenti verso gli ***User***.
- **User**: comunica gli eventi al ***Mediator***.

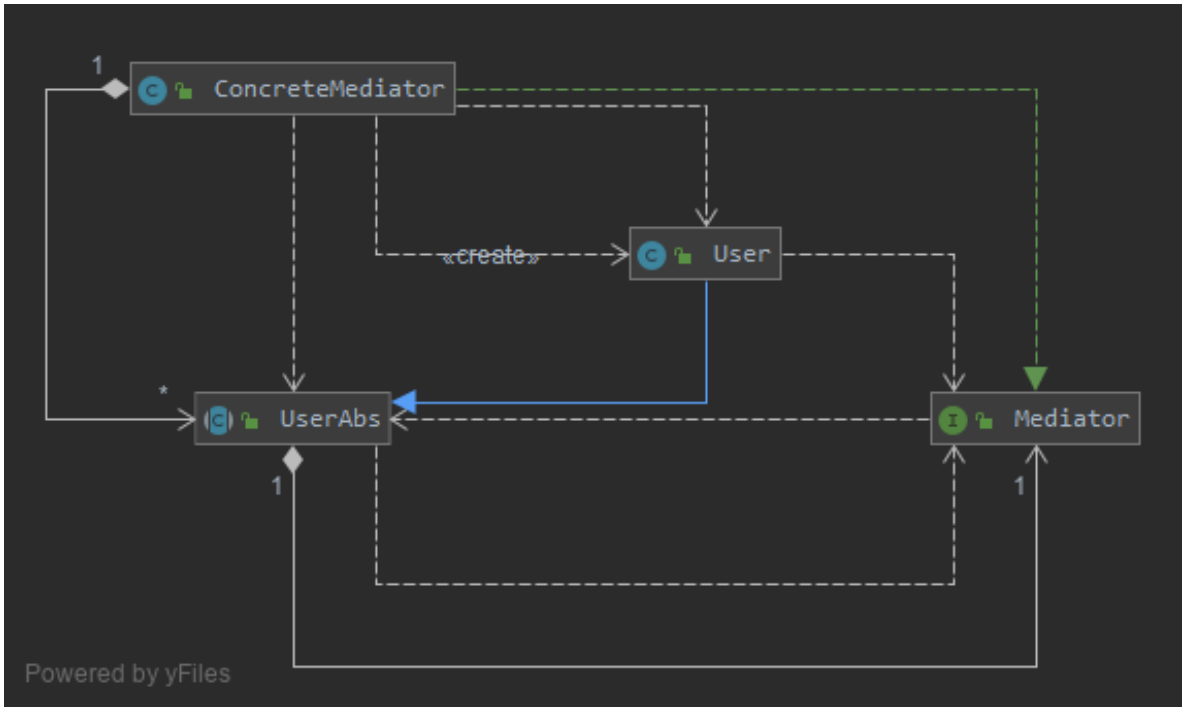
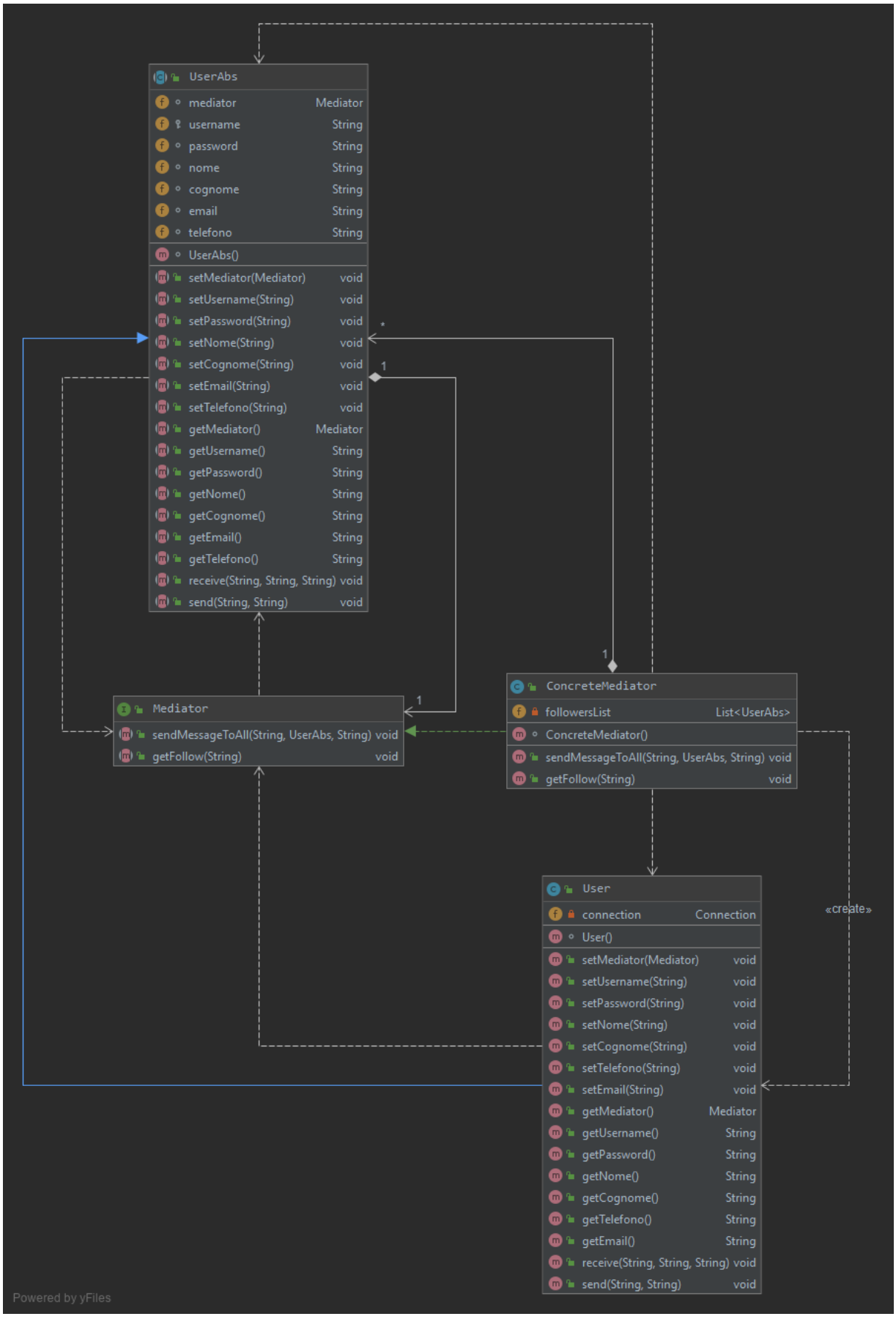


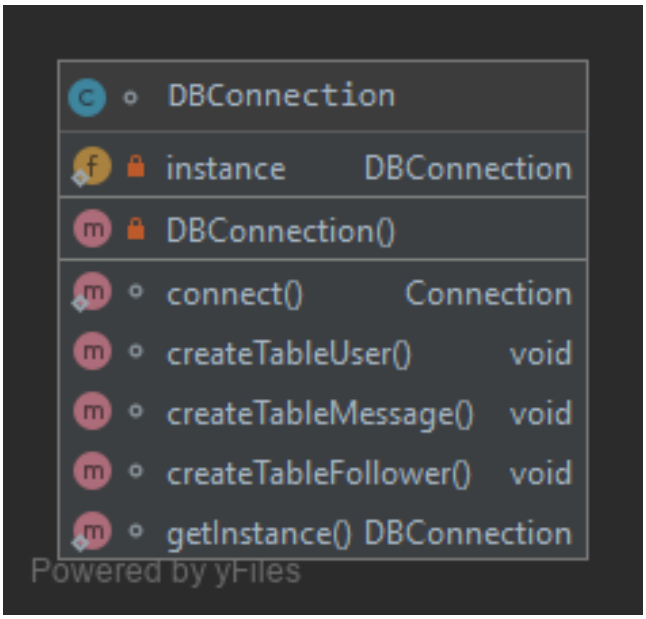
Diagramma UML delle classi nel dettaglio



Design Pattern Singleton

In questa sezione vediamo l'UML delle classi del design pattern ***Singleton***.

L'unico partecipante è ***DBConnection***. La classe permette di creare e ritornare un'unica istanza della classe con il suo metodo.



Codice

In questa sezione vengono inserite le parti più rilevanti del codice sviluppato.

Viene illustrato il codice del:

- design pattern Singleton (connessione al database e la creazione delle tabelle)
- design pattern Mediator (invio e ricezione dei messaggi da parte degli utenti)
- funzionamento di Twitter (sito in cui si possono inviare e ricevere i tweet)
- funzionamento del login
- funzionamento della registrazione
- funzionamento dell'admin

Design pattern Singleton

```
/**
 * La classe si occupa di
 * 1. stabilire una connesione con il database Sqlite
 * 2. creare le tabelle
 * 3. popolare le tabelle
 * Utilizza il pattern Singleton per creare un unica istanza della classe
DBConnection
 */
class DBConnection {
    //Variabili
    private static DBConnection instance;

    //Costruttore default Database
     private DBConnection(){ }

    /**
     * Metodo che consente la creazione e la connesione al database
     * Il database utilizzato è l'sqlite
     */
    static Connection connect() throws SQLException {
        String url = "jdbc:sqlite:lib/DB";
        return DriverManager.getConnection(url);
    }

    /**
     * Metodo che consente la creazione della tabella USER
     * Consente di tenere traccia di tutti gli utenti che si registrano al
servizio twitter
     * I campi obbligatori da inserire sono USERNAME, PASSWORD, NUMERO DI
TELEFONO e EMAIL
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    void createTableUser() throws SQLException {
        //Connesione database
        Connection connection=null;
        // Oggetto per interrogare il database
        Statement statement=null;
        try{
            // Query che crea la tabella user
            String query = "CREATE TABLE IF NOT EXISTS users(" +
                    "id integer PRIMARY KEY,"+
```

```
                    "username varchar(50) NOT NULL UNIQUE,"+
                    "password varchar(20) NOT NULL," +
                    "nome varchar(20)," +
                    "cognome varchar(20)," +
                    "numero_telefono varchar(10) NOT NULL UNIQUE," +
                    "email varchar(50) NOT NULL UNIQUE)";
            // Connesione al db
            connection = connect();
            // Creazione dell'oggeto per interrogare il db
            statement = connection.createStatement();
            // Esecuzione della query
            statement.execute(query);
        }catch(SQLException e){
            e.printStackTrace();
        }finally {
            // Chiusura oggetti
            if(statement!=null) statement.close();
            if(connection!=null) connection.close();
        }
    }

    /** Metodo che consente la creazione della tabella TWEET
     * Consente di tenere traccia di tutti i tweet che vengono scritti dagli
utenti
     * I campi obbligatori da inserire sono MITTENTE, DESTINATARIO, CORPO
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    void createTableMessage() throws SQLException {
        // Oggetto per interrogare il database
        Statement statement=null;
        // Connesione al database
        Connection connection=null;
        try{
            // Query che crea la tabella tweet
            String query = "CREATE TABLE IF NOT EXISTS tweet(" +
                    "id integer PRIMARY KEY," +
                    "mittente varchar(20) NOT NULL," +
                    "destinatario varchar(20) NOT NULL," +
                    "corpo varchar(140) NOT NULL," +
                    "hashtag varchar(20))";
            // Connesione al db
            connection=connect();
            // Creazione dell'oggetto per interrogare il db
            statement = connection.createStatement();
            // Esecuzione della query
            statement.execute(query);
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            // Chiusura oggetti
            if(statement!=null) statement.close();
            if(connection!=null) connection.close();
        }
    }

    /** Metodo che consente la creazione della tabella FOLLOWER
     * Consente di tenere traccia degli utenti che vogliono seguire i
"follower"
     * I campi obbligatori da inserire sono USERNAME e USERNAMEFOLLOWER
```

```
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    void createTableFollower() throws SQLException {
        // Oggetto per interrogare il database
        Statement statement=null;
        // Connesione database
        Connection connection=null;
        try{
            String query = "CREATE TABLE IF NOT EXISTS follower(" +
                    "id integer PRIMARY KEY," +
                    "username varchar(20) NOT NULL," +
                    "usernameFollower varchar(20) NOT NULL)";
            // Connesione al db
            connection=connect();
            // Creazione dell'oggetto per interrogare il db
            statement = connection.createStatement();
            // Esecuzione query
            statement.execute(query);
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            // Chiusura oggetti
            if(statement!=null) statement.close();
            if(connection!=null) connection.close();
        }
    }

    /**
     * Metodo che crea un unica istanza del database
     * PATTERN SINGLETON
     * @return istanza del database
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    static DBConnection getInstance() throws SQLException{
        if(instance == null)
            instance = new DBConnection();
        else if(connect().isClosed())
            instance = new DBConnection();

        return instance;
    }
}
```

Design pattern Mediator

Classe UserAbs

```
/**
 * La classe possiede un riferimento al MEDIATOR
 * Implementa un metodo di notifica di eventi al MEDIATOR
 * Utilizzato nel design patter MEDIATOR
 */

public abstract class UserAbs {
    Mediator mediator;

    /**
```

```
     * Metodo che consente di riceve i messaggi dal mediator
     * @param username mediator
     * @param messaggio inviato dal mediator
     * @param hashtag parola che contiene un hastag
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    public abstract void receive(String username, String messaggio, String
hashtag) throws SQLException;

    /**
     * Metodo che consente di inviare i messaggi dal mediator ai follower
     * @param messaggio scritto dal mediator
     * @param hashtag parola che contiene un hastag
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    public abstract void send(String messaggio, String hashtag) throws
SQLException;
}
```

Classe User

```
/**
 * Classe USER che si occupa di
 * 1. memorizzare le informazioni dell'utente
 * 2. comunicare gli eventi al MEDIATOR
 * Utilizzato nel design patter MEDIATOR
 */
public class User extends UserAbs {
    /**
     * Metodo che consente di riceve i messaggi dal mediator.
     * L'utente destinatario salverà il messaggio nel database
     * @param username mediator
     * @param messaggio inviato dal mediator
     * @param hashtag parola che contiene un hastag
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    @Override
    public void receive(String username, String messaggio, String hashtag)
throws SQLException {
        // Query che inserisce il messaggio nel db
        String query = "INSERT INTO tweet(mittente,destinatario,corpo,hashtag)
VALUES (?,?,?,?)";
        // Oggetti per interrogare il db
        PreparedStatement preparedStatement=null;

        try{
            connection = DBConnection.connect();
            preparedStatement = connection.prepareStatement(query);
            preparedStatement.setString(1,username);
            preparedStatement.setString(2,this.getUsername());
            preparedStatement.setString(3,messaggio);
            preparedStatement.setString(4,hashtag);
            preparedStatement.executeUpdate();
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
```

```
            if (preparedStatement!=null) preparedStatement.close();
            if (connection!=null) connection.close();
        }
    }


    /**
     * Metodo che consente di inviare i messaggi al destinatario.
     * @param messaggio scritto dal mediator
     * @param hashtag parola che contiene un hastag
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    @Override
    public void send(String messaggio, String hashtag) throws SQLException {
        this.mediator.sendMessageToAll(messaggio,this,hashtag);
    }
}
```

Classe Mediator

```
/**
 * Interfaccia del mediator
 * Specifica l'interfaccia per la comunicazione da parte di USERABS
 * Design patter Mediator
 */

public interface Mediator {
    /**
     * Metodo che consente di inviare il messaggio e l'hastag a tutti i
partecipanti della comunicazione
     * @param messaggio scritto dall'utente
     * @param username follower che segue l'utente
     * @param hashtag parola che contiene un hashtag
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    void sendMessageToAll(String messaggio, UserAbs username, String hashtag)
throws SQLException;

    /**
     * Metodo che consente di ricavare il follower dal database
     * @param username follower dell'utente
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    void getFollow(String username) throws SQLException;
}
```

Classe ConcreteMediator

```
/**
 * La classe si occupa di implementare l'interfaccia Mediator
 * Implementa l'invio e la ricezione dei messaggi
 * Possiede riferimenti verso USERABS
 * Utilizzato nel design patter MEDIATOR
 */
```

```java
public class ConcreteMediator implements Mediator {
    // Lista utenti
    private List<UserAbs> followersList;

    // Costruttore
    ConcreteMediator(){this.followersList = new ArrayList<>(); }

    /**
     * Metodo che consente di inviare un messaggio agli utenti che sono
associati al mediator
     * @param messaggio scritto dall'utente
     * @param username utente
     * @param hashtag se il messaggio contiene un hashtag
     */
    @Override
    public void sendMessageToAll(String messaggio, UserAbs username, String
hashtag) throws SQLException {
        for(UserAbs u:this.followersList)
            u.receive(username.getUsername(),messaggio,hashtag);
    }

    /**
     * Metodo che consnte di riceve dal database il follower
     * @param username utente follower
     */
    @Override
    public void getFollow(String username) throws SQLException {
        // Query che seleziona gli username dell'utente
        String query = "SELECT username FROM follower WHERE usernameFollower=?";
        // Oggetti per interrogare il db
        Connection connection = DBConnection.connect();
        PreparedStatement preparedStatement=null;
        ResultSet resultSet = null;

        try{
            preparedStatement=connection.prepareStatement(query);
            preparedStatement.setString(1,username);
            resultSet = preparedStatement.executeQuery();

            while(resultSet.next()){
                User user = new User();
                user.setUsername(resultSet.getString("username"));
                followersList.add(user);
            }
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            if (preparedStatement!=null) preparedStatement.close();
            if (resultSet!=null) preparedStatement.close();
            if (connection!=null) connection.close();
        }
    }
}
```

Twitter

```java
/**
 * Classe che si occupa di
 * 1. inserire un follower nel database
```

```
 * 2. cambiare email o cellulare in username, nel caso in cui l'utente
inserisca uno dei due campi
 * 3. controllare se esiste il follower nella tabella FOLLOWER
 * 4. controllare gli spazi vuoti
 * 5. prendere l'hashtag dal messaggio
 * 6. creare una lista di tweet per l'interfaccia utente
 * 7. creare una lista di follower per l'interfaccia utente
 */
class Twitter {
    private Connection connection;

    /**
     * Metodo che consente di inserire nel database un follower
     * @param user classe UTENTE in cui sono definite le informazioni
dell'utente, utente che ha effettuato l'accesso
     * @param follower utente che inserisce nella barra di ricerca il FOLLOWER
da seguire
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    void insertFollowerInDatabase(User user, String follower) throws
SQLException {
        // Query per inserire un dato nel db
        String query = "INSERT INTO follower(username,usernameFollower)
VALUES(?,?)";
        // Creazione oggetto per interrogare il db
        PreparedStatement preparedStatement = null;
        //Se trova l'email o il cellulare, lo cambia in username
        String username = changeEmailCell(follower);
        connection = DBConnection.connect();

        try{
            preparedStatement = connection.prepareStatement(query);
            preparedStatement.setString(1, user.getUsername());
            preparedStatement.setString(2, username);
            preparedStatement.executeUpdate();
        }catch (SQLException e) {
            e.printStackTrace();
        }finally {
            if(preparedStatement!=null) preparedStatement.close();
            if (connection!=null) connection.close();
        }
    }

    /**
     * Metodo che consente di controllare, nel caso in cui l'utente inserisce
come follower l'EMAIL o il CELLULARE, di
     * cambiare uno dei due campi in USERNAME
     * @param email_cell EMAIL o CELLULARE dell'utente follower da cercare
     * @return l'USERNAME dell'utente FOLLOWER
     * @throws SQLException  lancia un eccezione nel caso si verifichino degli
errori
     */
    private String changeEmailCell(String email_cell) throws SQLException {
        //Query che seleziona l'username
        String query = "SELECT username FROM users where username=? OR email=?
OR numero_telefono=?";
        //Oggetti per interrogare il database
        PreparedStatement preparedStatement=null;
        ResultSet resultSet=null;
```

```
        connection = DBConnection.connect();
        //Variabile di ritorno
        String username=null;

        try{
            preparedStatement = connection.prepareStatement(query);
            preparedStatement.setString(1,email_cell);
            preparedStatement.setString(2,email_cell);
            preparedStatement.setString(3,email_cell);
            resultSet = preparedStatement.executeQuery();

            while(resultSet.next()){
                username = resultSet.getString("username");
            }

            return username;
        } catch (SQLException e) {
            e.printStackTrace();
        }finally {
            //Chiusura oggetti
            if (resultSet!=null) resultSet.close();
            if(preparedStatement!=null) preparedStatement.close();
            if (connection!=null) connection.close();
        }
        return null;
    }

    /**
     * Metodo che consente di controllare se l'utente esiste già nel database
     * @param user classe UTENTE in cui sono definite le informazioni
dell'utente FOLLOWER
     * @return true se esiste, false viceversa
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    boolean checkFollower(String user) throws SQLException {
        //Query per cercare l'utente
        String query = "SELECT * FROM users WHERE username=? OR email=? OR
numero_telefono=?";
        //Oggetti per interrogare il db
        PreparedStatement preparedStatement = null;
        ResultSet resultSet = null;
        connection = DBConnection.connect();

        try {
            preparedStatement = connection.prepareStatement(query);
            preparedStatement.setString(1, user);
            preparedStatement.setString(2, user);
            preparedStatement.setString(3, user);
            resultSet = preparedStatement.executeQuery();

            return resultSet.next();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if (resultSet!=null) resultSet.close();
            if(preparedStatement!=null) preparedStatement.close();
            if (connection!=null) connection.close();
        }
        return false;
```

```
    }

    /**
     * Metodo che consente di controllare se l'utente sta già seguendo il
FOLLOWER che ha inserito
     * @param user classe UTENTE in cui sono definite le informazioni
dell'utente che ha effettuato l'accesso
     * @param follower utente da seguire
     * @return true se esite, false viceversa
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    boolean checkFollower(User user, String follower) throws SQLException {
        //Query che cerca l'utente
        String query = "SELECT * FROM follower WHERE username=? AND
usernameFollower=?";
        PreparedStatement preparedStatement=null;
        ResultSet resultSet=null;
        connection = DBConnection.connect();

        try{
            preparedStatement = connection.prepareStatement(query);
            preparedStatement.setString(1,user.getUsername());
            preparedStatement.setString(2,follower);
            resultSet = preparedStatement.executeQuery();

            return resultSet.next();
        } catch (SQLException e) {
            e.printStackTrace();
        }finally {
            if (resultSet!=null) resultSet.close();
            if(preparedStatement!=null) preparedStatement.close();
            if (connection!=null) connection.close();
        }
        return false;
    }

    /**
     * Metodo che consente di controllare se ci sono spazi vuoti
     * @param user utente follower
     * @return true se ci soono spazi vuoti, false viceversa
     */
    boolean checkEmptySpace(String user){ return user.equals(""); }

    /**
     * Metodo che consente di cercare l'hashtag nel TWEET
     * @param tweet tweet dell'utente
     * @return parola che contiene tutto l'hashtag, viceversa una stringa vuota
     */
    String searchHashtag(String tweet){
        for(String word:tweet.split("\\s+"))
            if(word.contains("#"))
                return word;
        return  " ";
    }

    /**
     * Metodo che consente di ritornare una lista di TWEET
     * @param username utente che ha effettuato l'accesso
     * @return lista di tweet
```

```
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    ArrayList<String> showTweet(String username) throws SQLException {
        //Query che prende i messaggi del destinatario
        String query="SELECT * FROM tweet WHERE destinatario=?";
        //Oggetti per interrogare il db
        PreparedStatement preparedStatement=null;
        connection=DBConnection.connect();
        ResultSet resultSet=null;
        //Lista messaggi
        ArrayList<String> tweetsList = new ArrayList<>();

        try {
            preparedStatement=connection.prepareStatement(query);
            preparedStatement.setString(1,username);
            resultSet = preparedStatement.executeQuery();

            while(resultSet.next()){
                Tweet tweet = new Tweet();
                String mittente = resultSet.getString("mittente");
                String destinatario = resultSet.getString("destinatario");
                String corpo = resultSet.getString("corpo");
                String hashtag = resultSet.getString("hashtag");

                tweet.setMittente(mittente);
                tweet.setDestinatario(destinatario);
                tweet.setCorpo(corpo);
                tweet.setHashtag(hashtag);

                tweetsList.add(tweet.getMittente()+": "+tweet.getCorpo());
            }

            return tweetsList;
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            if (connection!=null) connection.close();
            if (preparedStatement!=null) preparedStatement.close();
            if(resultSet!=null) resultSet.close();
        }
        return null;
    }

    /**
     * Metodo che consente di ritornare una lista di FOLLOWER
     * @param username utente che ha effettuato l'accesso
     * @return lista di follower dell'utente che ha effettuato l'accesso
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    ArrayList<String> showFollower(String username) throws SQLException {
        //Query che prende i messaggi del destinatario
        String query="SELECT * FROM follower WHERE username=?";
        //Oggetti per interrogare il db
        PreparedStatement preparedStatement=null;
        connection=DBConnection.connect();
        ResultSet resultSet=null;
        //Lista messaggi
        ArrayList<String> followersList = new ArrayList<>();
```

```
        try {
            preparedStatement=connection.prepareStatement(query);
            preparedStatement.setString(1,username);
            resultSet = preparedStatement.executeQuery();

            while(resultSet.next()){
                String follower = resultSet.getString("usernameFollower");
                followersList.add(follower);
            }

            return followersList;
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            if (connection!=null) connection.close();
            if (preparedStatement!=null) preparedStatement.close();
            if(resultSet!=null) resultSet.close();
        }
        return null;
    }

    /**
     * Metodo che consente di ritornare una lista di FOLLOWING
     * @param username utente che ha effettuato l'accesso
     * @return lista di follower dell'utente che ha effettuato l'accesso
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    ArrayList<String> showFollowing(String username) throws SQLException {
        //Query che prende i messaggi del destinatario
        String query="SELECT * FROM follower WHERE usernameFollower=?";
        //Oggetti per interrogare il db
        PreparedStatement preparedStatement=null;
        connection=DBConnection.connect();
        ResultSet resultSet=null;
        //Lista messaggi
        ArrayList<String> followingList = new ArrayList<>();

        try {
            preparedStatement=connection.prepareStatement(query);
            preparedStatement.setString(1,username);
            resultSet = preparedStatement.executeQuery();

            while(resultSet.next()){
                String follower = resultSet.getString("username");
                followingList.add(follower);
            }

            return followingList;
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            if (connection!=null) connection.close();
            if (preparedStatement!=null) preparedStatement.close();
            if(resultSet!=null) resultSet.close();
        }
        return null;
    }
}
```

Login

```
/**
 * La classe si occupa di
 * 1. implementare i metodi che vengono usati dall'interfaccia utente (classe
LoginControl)
 * 2. controllare se esiste l'utente
 * 3. controllare se l'utente è un amministratore
 */
class Login {
    /**
     * Metodo che consente di controllare, tramite accesso database, se
l'utente che inserisce
     * le credenziali sono corrette per entrare nel sito
     * @param user classe UTENTE in cui sono definite le informazioni
dell'utente, necessarie per l'accesso
     *             Troviamo i campi obbligatori USERNAME, EMAIL, TELEFONO e
PASSWORD
     * @return true se l'utente è nel database, false viceversa
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    boolean checkUser(User user) throws SQLException {
        // Query che controlla se esistono le informazioni inserite dall'utente
        String query = "SELECT * FROM users WHERE (username=? OR email=? OR
numero_telefono=?)AND password=?";
        // Oggetti per interrogare il database
        ResultSet resultSet = null;
        PreparedStatement preparedStatement = null;
        // Connesione al database
        Connection connection = DBConnection.connect();

        try {
            // Controlla che non ci siano spazi vuoti
            if(!checkEmptySpace(user)){
                preparedStatement = connection.prepareStatement(query);
                preparedStatement.setString(1,user.getUsername());
                preparedStatement.setString(2,user.getEmail());
                preparedStatement.setString(3,user.getTelefono());
                preparedStatement.setString(4,user.getPassword());

                resultSet = preparedStatement.executeQuery();

                return resultSet.next();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }finally {
            if (resultSet!=null) resultSet.close();
            if (preparedStatement!=null) preparedStatement.close();
            if (connection !=null) connection.close();
        }
        return false;
    }

    /**
     * Metodo che consente di controllare se l'utente è un admin
     * @param user classe UTENTE in cui sono definite le informazioni
```

```
dell'utente necessarie per l'accesso, troviamo i
     *             campi obbligatori USERNAME, EMAIL, TELEFONO e PASSWORD
     * @return true se l'utente è un admin, viceversa false
     */
    // Controlla se l'utente è un admin
    boolean checkIfAdmin(User user){
        return (user.getUsername().equals("admin")
                || user.getEmail().equals("admin@test.it")
                || user.getTelefono().equals("0000000000"))
                && user.getPassword().equals("admin");
    }

    /**
     * Metodo che consente di controllare se l'utente ha inserito dei campi
vuoti
     * @param user classe UTENTE, troviamo i campi USERNAME e la PASSWORD
     * @return true se ci sono spazi vuoti, viceversa false
     */
     boolean checkEmptySpace(User user){ return user.getUsername().equals("") &&
user.getPassword().equals(""); }
}
```

Registrazione

```
/**
 * La classe si occupa di
 * 1. implementare i metodi che vengono usati dall'interfaccia utente (classe
RegistrationControl)
 * 2. inserire l'utente nel database
 * 3. controllare se l'utente non è già registrato
 */
class Registration {
    private Connection connection;

    /** Metodo che consente di inserire i dati dell'utente nel database
     * @param user classe UTENTE in cui sono definite le informazioni
dell'utente, necessarie per la registrazione
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    void insertUserInDatabase(User user) throws SQLException {
        PreparedStatement preparedStatement=null;
        try{
            // Query per inserire un dato nel db
            String query = "INSERT INTO
users(username,password,nome,cognome,numero_telefono,email)
VALUES(?,?,?,?,?,?)";
            // Connesione al db
            connection = DBConnection.connect();
            // Creazione oggetto per interrogare il db
            preparedStatement = connection.prepareStatement(query);
            // Inserimento stringa
            preparedStatement.setString(1,user.getUsername());
            preparedStatement.setString(2,user.getPassword());
            preparedStatement.setString(3,user.getNome());
            preparedStatement.setString(4,user.getCognome());
            preparedStatement.setString(5,user.getTelefono());
            preparedStatement.setString(6,user.getEmail());
```

```
            // Esecuzione query
            preparedStatement.executeUpdate();
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            // Chiusura oggetti
            if(connection!=null) connection.close();
            if(preparedStatement!=null) preparedStatement.close();
        }
    }

    /**
     * Metodo che consente di controllare, se i campi inseriti dall'utente,
sono già inserite nel database
     * @param user classe UTENTE in cui sono definite le informazioni
dell'utente, necessarie per la registrazione
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
     boolean checkUser(User user) throws SQLException {
         // Query che controlla se le informazioni dell'utente non sono già
state inserite
        String query = "SELECT * FROM users WHERE username=? OR
numero_telefono=? OR email=?";
        // Connesione al db
        connection = DBConnection.connect();
        // Oggetti per interrogare il db
        ResultSet resultSet=null;
        PreparedStatement preparedStatement=null;

        try{
            preparedStatement = connection.prepareStatement(query);
            preparedStatement.setString(1, user.getUsername());
            preparedStatement.setString(2, user.getTelefono());
            preparedStatement.setString(3, user.getEmail());

            resultSet = preparedStatement.executeQuery();

            return resultSet.next();

        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            //Chiusura oggetti
            if(preparedStatement!=null) preparedStatement.close();
            if(resultSet!=null) resultSet.close();
        }

        return false;
    }
}
```

Admin

```
/**
 * La classe si occupa di
 * 1. trovare un TWEET data una parola
 * 2. trovare TWEET in base all'hashtag
 * 3. trovare numero di UTENTI in base al mumero di TWEET ricevuti o inviati
```

```
 */
class Admin {
    private Connection connection;

    /**
     * Metodo che consente di cercare i TWEET in base alla parola inserita
     * @param word parola da cercare
     * @return lista di TWEET
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    ArrayList<String> searchWordTweet(String word) throws SQLException {
        //Query che seleziona il mittente e il corpo data una parola
        String query="SELECT DISTINCT mittente,corpo FROM tweet WHERE corpo LIKE
'%"+word+"%'";
        //Oggetti per interrogare il db
        Statement statement=null;
        connection=DBConnection.connect();
        ResultSet resultSet=null;
        //Lista messaggi
        ArrayList<String> tweetsList = new ArrayList<>();

        try {
            statement=connection.createStatement();
            resultSet = statement.executeQuery(query);
            while(resultSet.next()){
                String mittente = resultSet.getString("mittente");
                String corpo = resultSet.getString("corpo");
                tweetsList.add(mittente+": "+corpo);
            }

            return tweetsList;
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            if (connection!=null) connection.close();
            if (statement!=null) statement.close();
            if(resultSet!=null) resultSet.close();
        }
        return null;
    }

    /**
     * Metodo che consente di cercare i messaggi in base agli HASHTAG
     * @return una lista di TWEET contenenti gli hashatag
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    ArrayList<String> showHashtag() throws SQLException {
        String query = "SELECT DISTINCT COUNT(id),hashtag,corpo FROM tweet GROUP
BY id,hashtag,corpo";
        //Oggetti per interrogare il db
        Statement statement=null;
        connection=DBConnection.connect();
        ResultSet resultSet=null;
        //Lista messaggi
        ArrayList<String> hashtagList = new ArrayList<>();

        try {
            statement=connection.createStatement();
```

```
            resultSet = statement.executeQuery(query);
            while(resultSet.next()){
                String corpo = resultSet.getString("corpo");
                hashtagList.add(corpo);
            }

            return hashtagList;
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            if (connection!=null) connection.close();
            if (statement!=null) statement.close();
            if(resultSet!=null) resultSet.close();
        }
        return null;
    }

    /**
     * Metodo che contente di cercare gli UTENTI in base ai messaggi ricevuti
     * @return lista di utenti che hanno ricevuto un messaggio
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    ArrayList<String> showUsersRecive() throws SQLException {
        String queryMittente = "SELECT DISTINCT username FROM users JOIN tweet
on users.username = tweet.destinatario GROUP BY (username)";
        //Oggetti per interrogare il db
        Statement statement=null;
        connection=DBConnection.connect();
        ResultSet resultSet=null;
        //Lista messaggi
        ArrayList<String> usersList = new ArrayList<>();

        try {
            statement=connection.createStatement();
            resultSet = statement.executeQuery(queryMittente);
            while(resultSet.next()){
                String username = resultSet.getString("username");
                usersList.add(username);
            }

            return usersList;
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            if (connection!=null) connection.close();
            if (statement!=null) statement.close();
            if(resultSet!=null) resultSet.close();
        }
        return null;
    }

    /**
     * Metodo che contente di cercare gli UTENTI in base ai messaggi inviati
     * @return lista di utenti che hanno inviato un messaggio
     * @throws SQLException lancia un eccezione nel caso si verifichino degli
errori
     */
    ArrayList<String> showUsersSent() throws SQLException {
        String queryMittente = "SELECT DISTINCT username " +
```

```
                "FROM users JOIN tweet on users.username = tweet.mittente" +
                " GROUP BY (username)";
        //Oggetti per interrogare il db
        Statement statement=null;
        connection=DBConnection.connect();
        ResultSet resultSet=null;
        //Lista messaggi
        ArrayList<String> usersList = new ArrayList<>();

        try {
            statement=connection.createStatement();
            resultSet = statement.executeQuery(queryMittente);
            while(resultSet.next()){
                String username = resultSet.getString("username");
                usersList.add(username);
            }

            return usersList;
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            if (connection!=null) connection.close();
            if (statement!=null) statement.close();
            if(resultSet!=null) resultSet.close();
        }
        return null;
    }
}
```