Slovak Technical University in Bratislava
Faculty of Informatics and Information Technology

OOP_B

Project

Music Voting

Lucas Daniel Espitia Corredor

Time practice: Wednesday 14:00-15:30

Ing. Ahmed Lofti Alqnatri

Music Voting

This document serves as the foundational framework for the development of the MusicVoting application, a JAVA-based endeavor forming part of the OOP_B curriculum at STU University's FIIT faculty during the summer semester.

MusicVoting is envisioned as an interactive software platform, offering users a highly intuitive and user-friendly experience. Its primary objective is to furnish users with a sophisticated software solution, allowing them to make selections and designate their top three preferred songs, organized by Latinoamerican genre and tailored to a specified audience.

The application's user interface will be marked by its ease of navigation and accessibility. It will encompass two distinct login portals: one designated for administrative personnel, enabling them to modify and manage various aspects of the program, and another exclusively for voters, facilitating user registration and participation in the voting process.

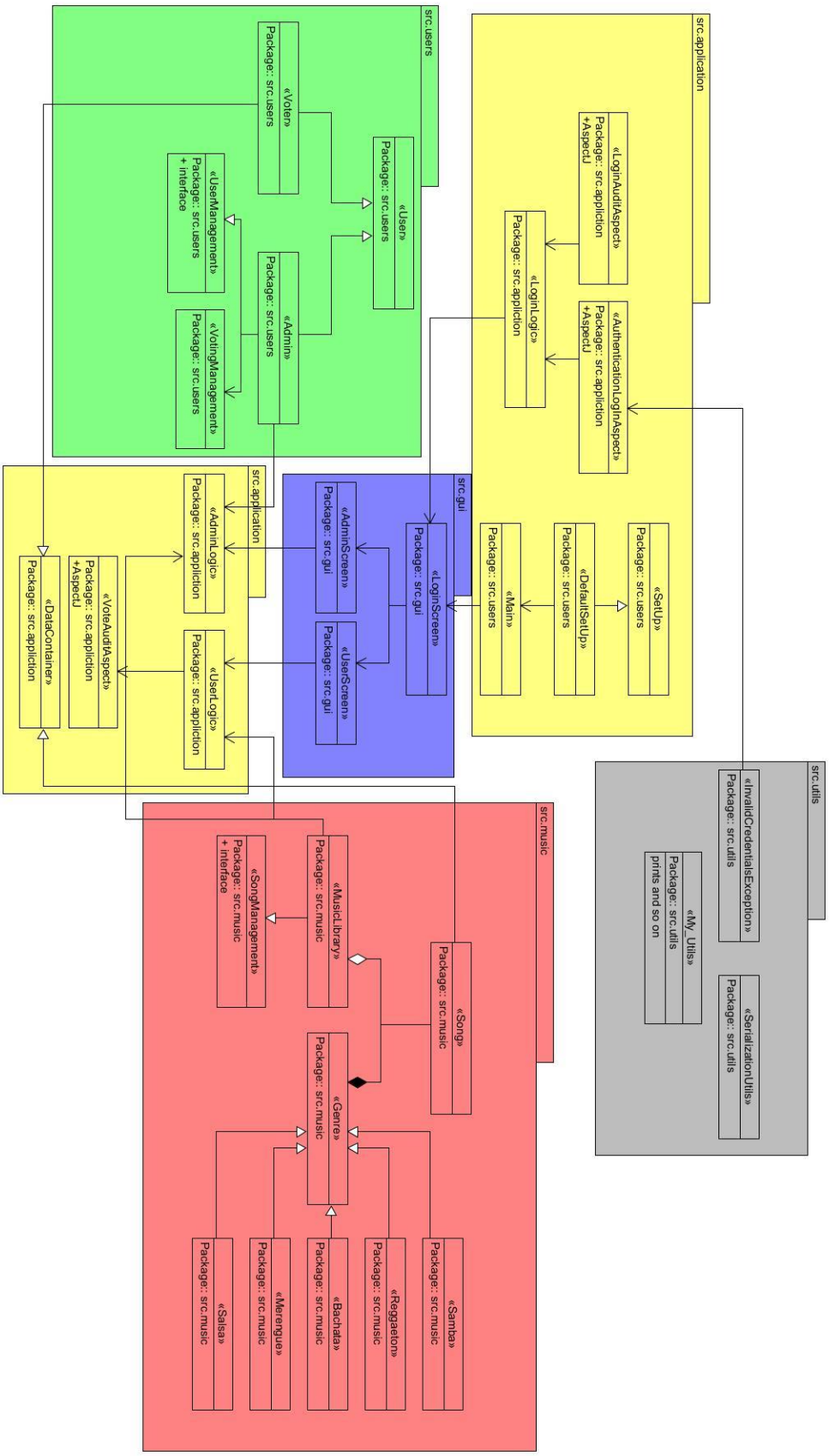
Users will be presented with an extensive array of genre options, each comprising a minimum of three songs. During each login session, users will have the opportunity to select a minimum of two songs, irrespective of genre.

Upon the conclusion of user logins, the program will meticulously tabulate the votes, ultimately revealing the top three most favored songs and their respective genres.

MusicVoting is poised to emerge as an engaging and dynamic application, ideally suited for communal usage among friends, within educational environments such as classrooms, or in any group setting, facilitating the collective selection of preferred musical songs.

** It should be noted that the application already comes with some default songs, but if the admin wants to remove or edit this, he can do it.

UML DIAGRAM


src.users
«Voter»
Package:: src.users
«User»
Package:: src.users
«UserManagement»
Package:: src.users
+ interface
«Admin»
Package:: src.users
«VotingManagement»
Package:: src.users
src.application
«LoginAuditAspect»
Package:: src.application
+AspectJ
«LoginLogic»
Package:: src.application
«AuthenticationLoginAspect»
Package:: src.application
+AspectJ
«SetUp»
Package:: src.users
«DefaultSetUp»
Package:: src.users
«Main»
Package:: src.users
src.gui
«LoginScreen»
Package:: src.gui
«AdminScreen»
Package:: src.gui
«UserScreen»
Package:: src.gui
src.application
«AdminLogic»
Package:: src.application
«UserLogic»
Package:: src.application
«VoteAuditAspect»
Package:: src.application
+AspectJ
«DataContainer»
Package:: src.application
src.utils
«InvalidCredentialsException»
Package:: src.utils
«My_Utils»
Package:: src.utils
prints and so on
«SerializationUtils»
Package:: src.utils
src.music
«SongManagement»
Package:: src.music
+ interface
«MusicLibrary»
Package:: src.music
«Song»
Package:: src.music
«Genre»
Package:: src.music
«Salsa»
Package:: src.music
«Merengue»
Package:: src.music
«Bachata»
Package:: src.music
«Reggaeton»
Package:: src.music
«Samba»
Package:: src.music

IMPORTANT INFORMATION

I have to clarify that the program is completely manual, therefore to be able to perform the simulation, the person in charge of reviewing the project will have to log in as a voter at least 3 times and can also enter the Admin login to finish the project.

In general, at this moment the program is running at 85% of the expected idea.

Unfortunately, it is necessary to finish some features, two to be exact. The first one, is the addition of users by file, and the second one is the option to delete the song of a respective genre.

On the other hand, the rest is functional and you can enjoy a very entertaining interface.

To do the adminLLogin, use the following credential

Username: L

Password: 1

Make sure that the capital letters are capitalized and that there are no spaces.

To log in as a voter, you will see the default users created in the console, however, here you can see the users, in this file, make sure you are using the correct password with the correct user.

Username: user1 Password: a12345

Username: user2 Password: b12345

Username: user3 Password: c12345

Username: user4 Password: d12345

Username: user5 Password: e12345

Also, I have added in the src folder, an example of files for the program to be able to add songs or users, if you want you can try the method of adding songs with file and use the file top3Songs (the user one is disabled for the moment).

Oop uses

Hierarchy

SUPERCLASS Users

```java
/**
 * The User class represents a user in the system.
 * It serves as the parent class for both voters and administrators.
 * This class provides encapsulated attributes for username and password,
 * along with getter and setter methods to access and modify these attributes.
 */
public abstract class User {
    // Attributes
    private String username;
    private String password;

    /**
     * Constructs a new User with the specified username and password.
     *
     * @param username The username of the user.
     * @param password The password of the user.
     */
    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    /**
     * Gets the username of the user.
     *
     * @return The username of the user.
     */
    public String getUsername() {
        return username;
    }

    /**
     * Sets the username of the user.
     *
     * @param username The new username to set.
     */
    public void setUsername(String username) {
        this.username = username;
    }

    /**
```

SUBCLASSES user

```java
/**
 * Public admin class
 * This class contains the logic and the attributes of the Admin
 * The principal function of the admin is add the people which is going to vote in the program
 * Add Songs or delete them from the program.
 * Use singletone Patron, because there's only one Admin
 */
public class Admin  extends User implements UserManagement{


    //For singleTone
    private static Admin instance;
    //Data
    private Map<String, Voter> users;
    private int numbersOfUsersvoted;

    /**
     * Constructor encapsulated
     * @param adminUsername
     * @param adminPassword
     */
    private Admin(String adminUsername, String adminPassword) {
        super(adminUsername, adminPassword);
        users = new HashMap<>();

    }

    /**
     * Singletone
     * @return
     */
    public static Admin getInstance() {
        if(instance == null) {
            instance = new Admin("Lucas", "123456789a");
        }
        return instance;

    }
    //getter
    public Voter getUser(String username) {
        return users.get(username);
    }
```

```java
/**
 * Public class Voter
 * This class is in charge of voting and containing the user instances in our application
 * It uses a Set to be able to know if the user has already voted for the respective song.
 */
public class Voter extends User {

    private Set<String> votedSongs;
    private boolean doVoteAlready;
    private int remainingVotes;
    private Admin admin = Admin.getInstance();

    /**
     * Constructor for Voter class.
     * @param username The username of the voter.
     * @param password The password of the voter.
     */
    public Voter(String username, String password) {
        super(username, password);
        setRemainingVotes(2);
        setDoVoteAlready(false);
        votedSongs = new HashSet<>();
    }

    /**
     * Getter for remaining votes.
     * @return The number of remaining votes.
     */
    public int getRemainingVotes() {
        return remainingVotes;
    }

    /**
     * Getter for whether the voter has already voted.
     * @return True if the voter has already voted, otherwise false.
     */
    public boolean getDoVoteAlready() {
        return doVoteAlready;
    }
```

SUPERCLASS Genres

```java
/**
 * The Genre class represents a musical genre in the music library.
 * It serves as the abstract superclass for specific genres and provides
 * common functionality shared by all genres.
 */
public abstract class Genre {

    private String name;
    private int idGenre;
    private List<Song> songs;

    /**
     * Constructs a new Genre object with the specified genre ID.
     *
     * @param idGenre The unique identifier for the genre.
     */
    public Genre(int idGenre) {
        setName(name);
        setIdGenre(idGenre);
        this.songs = new ArrayList<Song>();
    }

    /**
     * Returns the unique identifier for the genre.
     *
     * @return The genre's unique identifier.
     */
    public int getIdGenre() {
        return idGenre;
    }

    /**
     * Returns the name of the genre.
     *
     * @return The name of the genre.
     */
    public String getName() {
```

SUBCLASSES genre

```java
/**
 * The Bachata class represents the bachata genre in the music library.
 * It extends the Genre class and follows the Singleton pattern to ensure
 * that only one instance of Bachata exists.
 */
public class Bachata extends Genre {

    private static Bachata instance;

    /**
     * Constructs a new Bachata object with the specified genre ID.
     *
     * @param idGenre The unique identifier for the Bachata genre.
     */
    private Bachata(int idGenre) {
        super(idGenre);
        setName("Bachata");
    }
```

```java
/**
 * The Reggaeton class represents the Reggaeton genre.
 * It follows the Singleton design pattern to ensure that only one instance of the Reggaeton genre exists.
 */
public class Reggaeton extends Genre {

    // Singleton instance of Reggaeton
    private static Reggaeton instance;

    /**
     * Private constructor to prevent external instantiation.
     * Initializes the Reggaeton genre with the specified genre ID and name.
     *
     * @param idGenre The unique identifier for the Reggaeton genre.
     */
    private Reggaeton(int idGenre) {
        super(idGenre);
        setName("Reggaeton");
    }
```

```java
/**
 * The Merengue class represents the Merengue genre in the music library.
 * It is a subclass of the Genre class and follows the Singleton design pa
 * ensuring that only one instance of the Merengue genre exists.
 */
public class Merengue extends Genre {

    // Singleton instance of Merengue
    private static Merengue instance;

    /**
     * Private constructor to prevent external instantiation.
     * Initializes the Merengue genre with the specified genre ID and name
     *
     * @param idGenre The unique identifier for the Merengue genre.
     */
    private Merengue(int idGenre) {
        super(idGenre);
        setName("Merengue");
    }
```

```java
/**
 * The Salsa class represents the Salsa genre.
 * It follows the Singleton design pattern to ensure that only one instance of the Salsa genre exists.
 */
public class Salsa extends Genre {

    // Singleton instance of Salsa
    private static Salsa instance;

    /**
     * Private constructor to prevent external instantiation.
     * Initializes the Salsa genre with the specified genre ID and name.
     *
     * @param idGenre The unique identifier for the Salsa genre.
     */
    private Salsa(int idGenre) {
        super(idGenre);
        setName("Salsa");
    }
```

```java
/**
 * The Samba class represents the Samba genre.
 * It follows the Singleton design pattern to ensure that only one instance of the Samba genre exists.
 */
public class Samba extends Genre {

    // Singleton instance of Samba
    private static Samba instance;

    /**
     * Private constructor to prevent external instantiation.
     * Initializes the Samba genre with the specified genre ID and name.
     *
     * @param idGenre The unique identifier for the Samba genre.
     */
    private Samba(int idGenre) {
        super(idGenre);
        setName("Samba");
    }
```

POLIMORFYSMUS

Most of my polymorphism is handled in my MusicLibrary class by using a HashMap of my genres, and storing the genre child classes. We get the genre and use a method for the respective genre we want to do something to, for example, add a song, add more than one song, delete songs, and get the list of songs of each genre.

```java
 */
public List<Song> getSongsForGenre(String genreName) {
    Genre genre = genres.get(genreName);
    return genre.getAllSongs();
}

/**
 * Retrieves a song with the specified title for the specified genre.
 *
 * @param genreName The name of the genre.
 * @param songTitle The title of the song.
 * @return The song object, or null if the song does not exist in the genre.
 */
public Song getSongForGenre(String genreName, String songTitle) {
    Genre genre = genres.get(genreName);
    return genre.getSongByTitle(songTitle);
}

/**
```

```java
@Override
public boolean addSong(String genreName, Song song) {
    Genre genre = genres.get(genreName);
    if (genre != null) {
        genre.addSong(song);
        allSongs.add(song);
        return true;
    }
    return false;
}

@Override
public void deleteSong(String genreName, Song song) {
    Genre genre = genres.get(genreName);
    if (genre != null) {
        genre.deleteSong(song);
    } else {
        System.out.println("The genre " + genreName + " does not exist in the music library.");
    }
}
```

```java
/**
 * Prints all songs stored in the music library, organized by genre.
 */
public void printAllSongs() {
    for (Map.Entry<String, Genre> entry : genres.entrySet()) {
        Genre genre = entry.getValue();
        System.out.println("Genre: " + genre.getName());
        for (Song song : genre.getAllSongs()) {
            System.out.println("Title: " + song.getTitle());
            System.out.println("Artist: " + song.getArtist());
            System.out.println("Album: " + song.getAlbum());
            System.out.println("---");
        }
    }
}
```

A polymorphism has also been added in the Admin class, precisely with the interface as follows:

```java
/**
 * Method to import users from a file.
 * @param filePath The file path from which to import users.
 */
void addAllUsers(File filePath);

/**
 * Method to add all users from a 2D array.
 * @param usernameAndPassword The 2D array containing username
 */
void addAllUsers(String[][] usernameAndPassword);
/**
 * Add all users from the serialization
 * @param voters
 */
void addAllUsers(List<Voter> voters);
/**
 * Print the voter username
 * @param user
```

Aggregation

In my musicLibrary class we aggregate the genres in a HashMap and all the songs in a SongList.

```
 */
public class MusicLibrary implements SongManagement {

    private static MusicLibrary instance;
    private Map<String, Genre> genres;
    private List<Song> allSongs;

    /**
```

In my Admin class we add the voters in a hashmap, where the key will be the username

```
public class Admin  extends User implements UserManagement{


    //For singleTone
    private static Admin instance;
    //Data
    private Map<String, Voter> users;
    private int numbersOfUsersvoted;

    /**
```

Composition

The composition has been done and is that each genre is composed of a list of songs, strictly for the songs to exist, they must have a specific genre, therefore it is characterized by composition, if the genre is destroyed, the songs can not be because they are characteristic of the genre.

```
public abstract class Genre {

    private String name;
    private int idGenre;
    private List<Song> songs;

    /**
     * Constructs a new Genre object with the spec
     *
     * @param idGenre The unique identifier for th
     */
    public Genre(int idGenre) {
        setName(name);
        setIdGenre(idGenre);
        this.songs = new ArrayList<Song>();
    }
```

FINA CONTROL POINT

The project meets the basic requirements for the program solution. The finished product also fulfills the functions intended from the beginning, performs the required actions and complies with the OOP principles.

As noted above the software meets the criteria using the use of two hierarchies user and genre, the use of polymorphism in the musicLibrary class, the implementation of interfaces.

It also uses correctly the aggregation and composition in the previous classes, on the other hand, the use of encapsulation has been done in a clear way, where each class contains most of the methods and attributes in a private or protected way. Only the necessary methods are used in a public way and even only used in the two main classes. Admin and MusicLibrary, which control users and song-genres respectively.

Likewise, the code is organized in a clear way, where it is separated by packages, the application as such, is correctly separated from the logic and the interface.

To review polymorphism, inheritance and aggregation please review the points above.

ENCAPSULATION

The use of encapsulation has been efficient since most of the data is stored privately.

```java
public class Admin  extends User implements UserManagement{

    /**
     * A unique identifier for serialization purposes.
     */
    private static final long serialVersionUID = 1L;

    /**
     * The single instance of the Admin class (Singleton pattern).
     */
    private static Admin instance;

    /**
     * A map containing usernames as keys and corresponding Voter objects as values.
     */
    private Map<String, Voter> users;


    /**
     * Constructor encapsulated
     * @param adminUsername
     * @param adminPassword
     */
    private Admin(String adminUsername, String adminPassword) {
        super(adminUsername, adminPassword);
        users = new HashMap<>();

    }

    /**
     * Singletone, get the only instance of the admin class
     * @return
     */
    public static Admin getInstance() {
        if(instance == null) {
            instance = new Admin("L", "1");
        }
        return instance;

    }
    /**
     * get the User as an object
```

```java
    * A unique identifier for serialization purposes.
    */
   private static final long serialVersionUID = 1L;

   /**
    * The username of the user.
    */
   private String username;

   /**
    * The password of the user.
    */
   private String password;

   /**
    * Constructs a new User with the specified username and password.
    *
    * @param username The username of the user.
    * @param password The password of the user.
    */
   public User(String username, String password) {
       this.username = username;
       this.password = password;
   }
   /**
    * Constructor used for deserialization
    */
   public User() {

   }

   /**
    * Gets the username of the user.
    *
    * @return The username of the user.
    */
   public String getUsername() {
```

```java
/**
 * The singleton instance of the MusicLibrary class.
 */
private static MusicLibrary instance;

/**
 * A map containing genres as keys and corresponding Genre objects as values.
 */
private Map<String, Genre> genres;

/**
 * A list containing all songs in the music library.
 */
private List<Song> allSongs;

/**
 * Private constructor to prevent external instantiation.
 * Initializes the music library with empty collections for genres and songs.
 */
private MusicLibrary() {
    this.genres = new HashMap<>();
    this.allSongs = new ArrayList<>();
    initializeGenres();
}
```

```java
     */
    public Genre getGenre(String genreName) {
        return genres.get(genreName);
    }

    /**
     * Retrieves all songs for the specified genre.
     *
     * @param genreName The name of the genre.
     * @return A list of songs belonging to the genre.
     */
    public List<Song> getSongsForGenre(String genreName) {
        Genre genre = genres.get(genreName);
        return genre.getAllSongs();
    }

    /**
     * Retrieves a song with the specified title for the specified genre.
     *
     * @param genreName The name of the genre.
     * @param songTitle The title of the song.
     * @return The song object, or null if the song does not exist in the genre.
     */
    public Song getSongForGenre(String genreName, String songTitle) {
        Genre genre = genres.get(genreName);
        return genre.getSongByTitle(songTitle);
```

```java
private static final long serialVersionUID = 1L;

/**
 * A set containing the titles of songs that the user has voted for.
 */
private Set<String> votedSongs;

/**
 * A flag indicating whether the user has already voted.
 */
private boolean doVoteAlready;

/**
 * The number of remaining votes the user has.
 */
private int remainingVotes;

/**
 * The admin instance used for certain operations.
 */
private Admin admin = Admin.getInstance();
```

```java
public int getRemainingVotes() {
    return remainingVotes;
}

/**
 * Getter for whether the voter has already voted.
 * @return True if the voter has already voted, otherwise false.
 */
public boolean getDoVoteAlready() {
    return doVoteAlready;
}

/**
 * Setter for remaining votes.
 * @param remainingVotes The number of remaining votes to set.
 */
public void setRemainingVotes(int remainingVotes) {
    this.remainingVotes = remainingVotes;
}

/**
 * Setter for whether the voter has already voted.
 * @param doVoteAlready True if the voter has already voted, otherwise false.
 */
public void setDoVoteAlready(boolean doVoteAlready) {
    this.doVoteAlready = doVoteAlready;
}
```

As you can see, the use of setters and getters has been used very efficiently, all attributes are in private and the only way to access them is through setters and getters so the encapsulation is correct.

Other criteria

singletone pattern

```
private static Admin instance;
```

```
/**
 * Singletone, get the only instance of the admin class
 * @return
 */
public static Admin getInstance() {
    if(instance == null) {
        instance = new Admin("L", "1");
    }
    return instance;

}
```

```
 */
private static MusicLibrary instance;
```

```
public static MusicLibrary getInstance() {
    if (instance == null) {
        instance = new MusicLibrary();
    }
    return instance;
}
```

pattern set Up

```
/**
 * The SetUp class is responsible for initializing the application setup.
 * It provides methods to add users and songs to the system.
 */
public abstract class SetUp {

    // Instance of the Admin class
    protected Admin admin = Admin.getInstance();

    // Instance of the MusicLibrary class
    protected MusicLibrary musicLibrary = MusicLibrary.getInstance();

    /**
     * Adds multiple users to the system based on the provided data.
     * @param usersAndPasswords A 2D array containing usernames and passwords for each user.
     */
    protected void addAllUsers(String[][] usersAndPasswords) {
        admin.addAllUsers(usersAndPasswords);
    }

    /**
     * Adds multiple songs to the music library based on the provided data.
     * @param songs A 2D array containing song data including genre, title, artist, and album.
     */
    protected void addAllSongs(String[][] songs) {
        for (String[] songData : songs) {
            String genre = songData[0];
            String title = songData[1];
            String artist = songData[2];
            String album = songData[3];

            Song song = new Song(genre, title, artist, album);
            musicLibrary.addSong(genre, song);
        }
    }
}
```

```java
/**
 * Constructs a new DefaultSetUp object.
 * Initializes the default users and songs arrays.
 */
public DefaultSetUp() {
    // Initialization of default users and songs
    usersAndPasswords = new String[][] {
        {"user1", "a12345"},
        {"user2", "b12345"},
        {"user3", "c12345"},
        {"user4","d12345"},
        {"user5", "e12345"}
    };
    songs = new String[][] {
        // Genre: Salsa
        {"Salsa", "Vivir mi vida", "Marc Anthony", "3.0"},
        {"Salsa", "La vida es un carnaval", "Celia cruzova", "My life is a song"},
        {"Salsa", "Detalles", "Oscar D Leon", "Unknown"},

        // Genre: Bachata
        {"Bachata", "Eres mia", "Romeo Santos", "Formula Vol1"},
        {"Bachata", "Propuesta indecente", "Romeo Santos", "Formula Vol2"},
        {"Bachata", "Darte un beso", "Prince Royce", "I am the same"},

        // Genre: Reggaeton
        {"Reggaeton", "Despacito", "Luis Fonsi", "Life"},
        {"Reggaeton", "Gasolina", "Daddy Yankee", "Hood"},
        {"Reggaeton", "Moscow Mule", "Bad Bunny", "A summer without you"},

        // Genre: Merengue
        {"Merengue", "Suavemente", "Elvis Crespo", "Unknown"},
        {"Merengue", "Abusadora", "Wilfrido Vargas", "Unknown"},
        {"Merengue", "Guallando", "Fulanito", "The Most Famous Man On Earth"},

        // Género: Samba
        {"Samba", "Samba", "Gloria Estefan", "Brazil305"},
        {"Samba", "Maricaipinha", "Carlinhos Brown", "Unknown"},
        {"Samba", "Aquiarela Brasileira", "Martinho de Vila", "Aquiarela Brasileira 4"}
    };
    addAllSongs(songs);
    addAllUsers(usersAndPasswords);
    admin.setVoteCount(0);
}
```

```java
/**
 * Constructor of the EmptySetUp
 */
public EmptySetUp() {

    //We need to add all elements manually
    admin.setVoteCount(0);
}
```

Exception

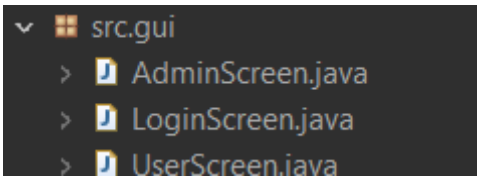
```java
/**
 * Exception thrown to indicate that the provided credentials are invalid.
 */
public class InvalidCredentialsException extends RuntimeException {
    /**
     * Default serial version ID.
     */
    private static final long serialVersionUID = 1L;

    /**
     * Constructs a new InvalidCredentialsException with the specified detail message.
     *
     * @param message The detail message (which is saved for later retrieval by the getMessage() method).
     */
    public InvalidCredentialsException(String message) {
        super(message);
    }
}
```

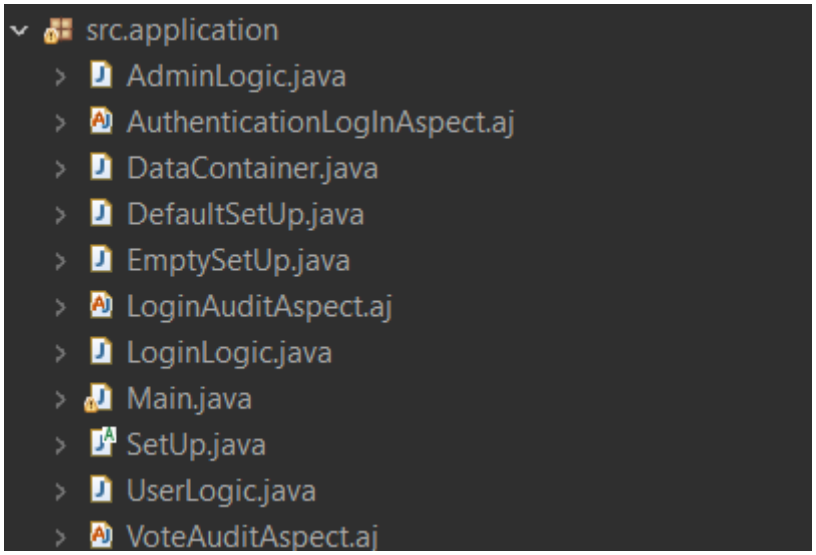
We also create more exceptions, e.g. users cannot vote more than twice, etc.

```java
public void doVote(Song song) {
    if (remainingVotes > 0) {
        if (!votedSongs.contains(song.getTitle())) {
            //My_Utils.print("Test1");
            song.addVote();
            remainingVotes -= 1;
            votedSongs.add(song.getTitle());
            My_Utils.print("Remaining votes: " + getRemainingVotes());
            if(remainingVotes == 0 ) {
                //My_Utils.print("Test2");
                VotingManager.vote();
                doVoteAlready = true;
                My_Utils.print("Voted: " + admin.getVoteCount());
            }
        } else {
            My_Utils.printErrorText("Error: You have already voted for this song.");
            My_Utils.print("You have already voted for this song.");
        }
    } else {
        My_Utils.print("No remaining votes left.");
    }
}
```

An error box is printed

Interface separated from the logic





Nested classes

```java
public class AdminLogic {
    /**
     * Instance of Admin
     */
    private Admin admin = Admin.getInstance();
    /**
     * Instance of musicLibrary
     */
    private MusicLibrary musicLibrary = MusicLibrary.getInstance();

       public class UserLogic {
           private Admin admin = Admin.getInstance();
           /**
            * Confirm add Users to the program from the selected file
            *
            * @param selectedFile
            */
           public void confirmaddUserByFile(File selectedFile) {
                  admin.addAllUsers(selectedFile);
           }
```

In the administrator logic class, the logic of users and songs has been separated.

```java
// Lógica relacionada con canciones
public  class SongLogic {
    private MusicLibrary musicLibrary = MusicLibrary.getInstance();

     /**
     * Confirms the addition of songs from the selected file.
     *
     * @param selectedFile2 The selected file containing songs to add.
     */
    public void confirmFileAddSongs(File selectedFile2) {
           musicLibrary.addSongsFromFile(selectedFile2);
    }
    /**
```

Lambda Expressions

```
/**
 *Lambda expression that allows me to print my users more efficiently
 * @param action
 */
public void forEachUser(Consumer<Voter> action) {
        users.values().forEach(action);
}
```

```
@Override
public void addAllUsers(List<Voter> voters) {
    voters.forEach(voter -> addUser(voter));
```

```
@Override

public void addSongFromSerialization(List<Song> songs) {
    songs.forEach(song -> addSong(song.getGenre(), song));
}
```

Implicit use in interfaces

```
* Interface for user management.
*/
public interface UserManagement {

    /**
     * Method to add a user.
     * @param user The user to add.
     * @return True if the user was added successfully, otherwise false.
     */
    boolean addUser(Voter user);

    /**
     * Method to delete a user.
     * @param userID The ID of the user to delete.
     */
    void deleteUser(String userID);

    /**
     * Method to import users from a file.
     * @param filePath The file path from which to import users.
     */
    void addAllUsers(File filePath);

    /**
     * Method to add all users from a 2D array.
     * @param usernameAndPassword The 2D array containing usernames and passwords.
     */
    void addAllUsers(String[][] usernameAndPassword);
    /**
     * Add all users from the serialization
     * @param voters
     */
    void addAllUsers(List<Voter> voters);
    /**
     * Print the voter username
     * @param user
     */
    default void printUser(Voter user) {
        My_Utils.print(user.getUsername());
    }
}
```

AspectJ

```java
/**
 * AuthenticationLogInAspect is an aspect-oriented class responsible for validating user credentials
 * before executing the login logic.
 */
@Aspect
public class AuthenticationLogInAspect {

    private Admin admin = Admin.getInstance();

    /**
     * Validates user credentials before executing the login logic.
     * Throws InvalidCredentialsException if the username or password is invalid.
     *
     * @param username The username provided for login.
     * @param password The password provided for login.
     */
    @Before("execution(* src.application.LoginLogic.validateUserCredentials(..)) " +
            "&& args(username, password)")
    public void validateUserCredentials(String username, String password)  {
        if (!isValid(username, password)) {
            throw new InvalidCredentialsException("Invalid username or password");
        }
    }

    /**
     * Checks if the provided username and password are valid.
     *
     * @param username The username provided for validation.
     * @param password The password provided for validation.
     * @return true if the username and password are valid, false otherwise.
     */
    private boolean isValid(String username, String password) {
         User user = admin.getUser(username);
         if (user != null) {
             if (user.getPassword().equals(password)) {
                 return true;
             } else {
                 return false;
             }
         } else {
             return false;
         }
     }
```

```java
**
* LoginAuditAspect is an aspect class that performs audit logging for login attempts.
* It logs login attempts for both administrators and users.
*/
ublic aspect LoginAuditAspect {
    /**
     * Pointcut for admin login attempt.
     * It captures the execution of the validateAdminCredentials method in LoginLogic.
     */
    pointcut adminLoginAttempt(String adminUsername, String adminPassword):
        execution(boolean LoginLogic.validateAdminCredentials(String, String)) &&
        args(adminUsername, adminPassword);

    /**
     * Pointcut for user login attempt.
     * It captures the execution of the validateUserCredentials method in LoginLogic.
     */
    pointcut userLoginAttempt(String username, String password):
        execution(String LoginLogic.validateUserCredentials(String, String)) &&
        args(username, password);

    /**
     * Advice executed before an admin login attempt.
     * Logs the timestamp and admin username for the login attempt.
     */
    before(String adminUsername, String adminPassword): adminLoginAttempt(adminUsername, adminPassword) {
        LocalDateTime dateTime = LocalDateTime.now();
        System.out.println("Admin login attempt at " + dateTime + " by admin: " + adminUsername);
    }

    /**
     * Advice executed before a user login attempt.
     * Logs the timestamp and username for the login attempt.
     */
    before(String username, String password): userLoginAttempt(username, password) {
        LocalDateTime dateTime = LocalDateTime.now();
        System.out.println("User login attempt at " + dateTime + " by user: " + username);
    }
```

```java
/**
 * VoteAuditAspect is an aspect class that performs audit logging for user votes.
 * It logs the details of user votes, including the voter username, genre, and selected song.
 */
public aspect VoteAuditAspect {

    /**
     * Pointcut for user vote.
     * It captures the execution of the confirmVote method in UserLogic.
     */
    pointcut userVote(Voter user, String genre, String songSelected):
        execution(void UserLogic.confirmVote(Voter, String, String)) &&
        args(user, genre, songSelected);

    /**
     * Advice executed before a user vote.
     * Logs the timestamp, voter username, genre, and selected song for the vote.
     */
    before(Voter user, String genre, String songSelected): userVote(user, genre, songSelected) {
        LocalDateTime dateTime = LocalDateTime.now();
        System.out.println("User " + user.getUsername() + " voted for song " + songSelected + " in genre " + genre + " at " + dateTime);
    }
}
```

Serialization

```java
*/
public class DataContainer implements Serializable {
    /**
     * Serial version UID for serialization.
     */
    private static final long serialVersionUID = 1L;

    /**
     * List of users stored in the data container.
     */
    private List<Voter> users;

    /**
     * List of songs stored in the data container.
     */
    private List<Song> songs;

    /**
     * Instance of the voting Manager
     */

    private int voteCount;


    /**
     *  Constructs a new DataContainer with the specified lists of users and songs.
     * @param users
     * @param songs
     * @param admin
     * @param loadedFromFile
     */
    public DataContainer(List<Voter> users, List<Song> songs, int votingCount) {
        this.users = users;
        this.songs = songs;
        this.voteCount = votingCount;
    }
```

```java
 */
public class Song implements Serializable {
        /**
         * The serial version UID for serialization.
         */
        private static final long serialVersionUID = 1L;

        /**
```

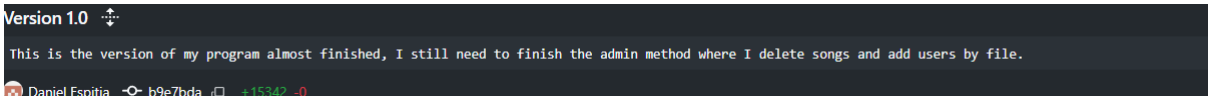
```java
 */
public abstract class User implements Serializable {

    /**
     * A unique identifier for serialization purposes.
     */
    private static final long serialVersionUID = 1L;
```

```java
public class Voter extends User implements Serializable{
    /**
     * A unique identifier for serialization purposes.
     */
    private static final long serialVersionUID = 1L;
```

```java
import java.io.*;
public class SerializationUtils {
    /**
     * Method to save the Data of the program in the file
     *
     * @param object
     * @param fileName
     */
    public static void saveToFile(Object object, String fileName) {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fileName))) {
            oos.writeObject(object);
           My_Utils.print("Object saved successfully.");
        } catch (FileNotFoundException e) {
            My_Utils.print("File not found: " + fileName);
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Method to Load the data from the file to the program.
     * @param fileName
     * @return
     */
    public static Object loadFromFile(String fileName) {
        Object object = null;
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(fileName))) {
            object = ois.readObject();
            My_Utils.print("Object loaded successfully.");
        } catch (FileNotFoundException e) {
            My_Utils.print("File not found: " + fileName);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
        return object;
    }
}
```

GITHUB HISTORIAL

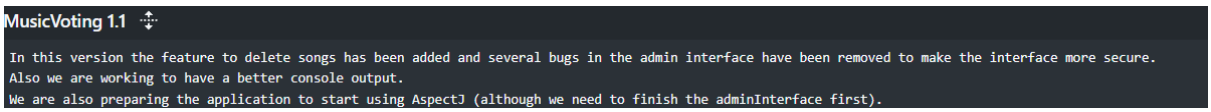
VERSION 1.0



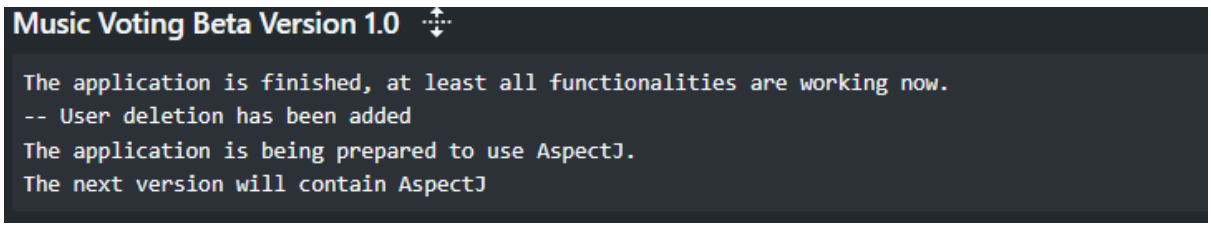
It was the first version of the program, the program and its functionality had not yet been completed.

In this version the OOP principles, and other things, have been used, please check the documentation carefully to see what has been added.

VERSION 1.1

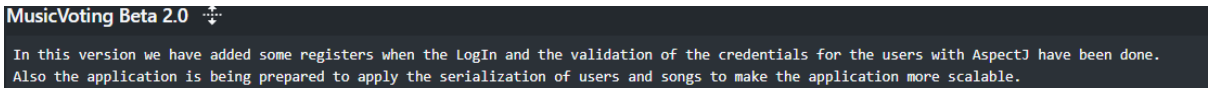


VERSION BETA 1.0



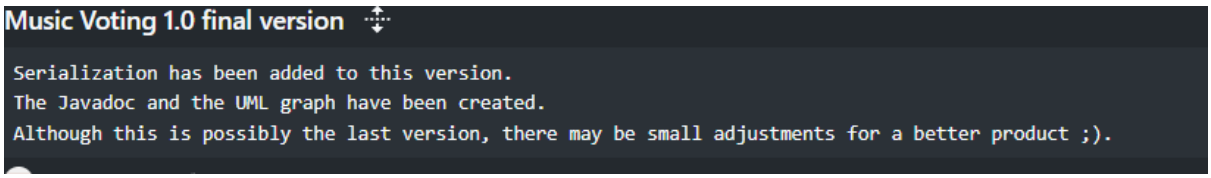
In this version all the functionality of the program is completed.

VERSION BETA 2.0



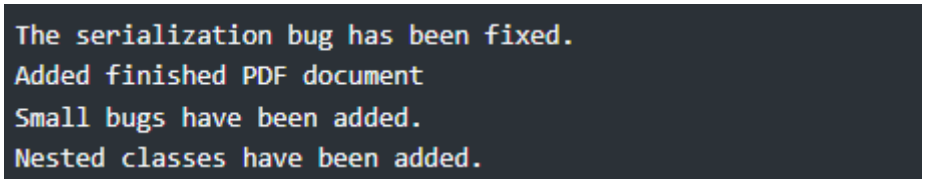
Aspects were added and the interface logic was successfully separated.

VERSION FINAL VERSION 1.0



Serialization was added and some of the concepts of the other criteria began to be applied.

FINAL VERSION



The serialization was fixed since it had a bug, the nested classes were added and another child was added in the SetUp pattern.

This is the final version of the program.