Music Streaming platform - ”Spotify”

L.D Muñoz, Student member, J.D Álvarez, Student member,

***Abstract*—Currently with the growth of music industries the demand for music platforms has also grown with great boom, so this paper is based on the creation of a music playback website with great similarities to one of the most popular platforms globally, which is Spotify. This paper explains the methodologies and tools used that were taken into account which lead us to good software quality metrics, in addition to this, it ends with the performance of unit tests and testing to show the quality of the website and the development that this had.**

***Index Terms*—keywords, temperature, xxxx equation, etc.**

I. Introduction

IN these moments of the present time the music distribution has increased a lot, for the same reason the market that offers these services has grown a lot, with all this to create a music platform can be very risky if it is not carried out with the due care, for that reason a music playback website was developed with some functionalities and some similarities to one of the most known platforms worldwide, Spotify.

To achieve this and to have a good realization and structure of software, patterns, anti-patterns, code smells, solid principles and crud operations were taken into account, so that in this way a good quality of software can be achieved. We also made use of databases and to demonstrate that the software complies with the requirements we performed unit tests and tests in virtual environments.

II. Specification of requirements

- We want a person who wants to enter the music platform to have two options, a log in or a sign up.
- If the person indicates that he/she wants to log in because he/she already has a registered account, the system verifies that the account has been registered before.
- If the person indicates to register, the system saves the data provided by the user.
- There are two types of users, a free user and a subscribed user. The user becomes a subscriber when he/she makes a monthly payment to the platform, which provides certain benefits, such as listening to music without ads, returning songs and playing songs from a playlist randomly.
- When a user wants to be an artist he/she goes to a menu of options in which he/she indicates his/her artist data and has to upload an album with songs.
- A user (regardless of the state he/she is in) can make a playList with the songs he/she likes.

III. Methodology and materials

Taking into account the requirements, a software design and analysis phase follows. This part is crucial to visualize the structure and behavior of the system, improving communication and development efficiency. In turn, by implementing them, concrete classes, interfaces and relationships between classes can be identified.
UML implementation diagrams are as follows:

- **Activity diagram:** The activity diagram is a diagram used to model the workflow or behavior of a system or business process. These diagrams are useful for representing complex processes, procedures and algorithms in a visual and easy to understand manner.

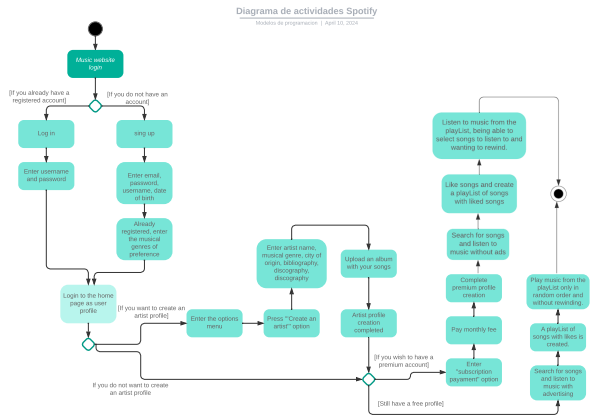

Fig. 1. Activity Diagram.

- **entity-relationship model:** It is s a graphical representation that illustrates how “entities” (such as people, objects, or concepts) relate to each other within a people, objects or concepts) relate to each other within a system. system. These diagrams are used to design or debug relational databases in fields such as software in fields such as software engineering, business information systems, education and information systems, education and research.

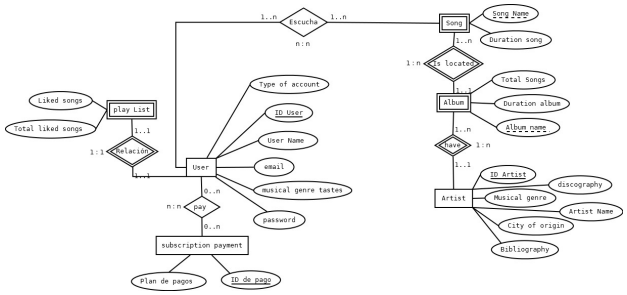

Fig. 2. entity-relationship model.

- **Class diagram:**The class diagram is a fundamental part in the implementation of a software development. This is the one that outlines the software structure, determining the interacting classes, their attributes and methods. In addition to this, it is possible to visualize the design patterns which are being implemented.

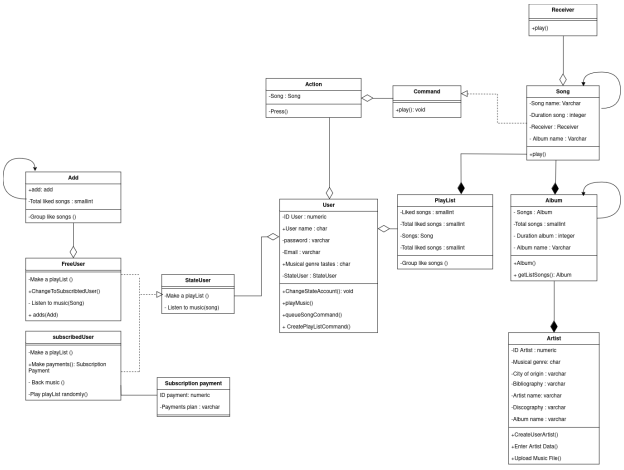

Fig. 3. ClassDiagram.

IV. Design patterns

Design patterns are models or templates that guide developers in finding solutions to common problems in software development and other areas of interaction or interface design. These patterns provide standard and reusable solutions to complex problems regardless of the programming language used.
Design patterns are divided into three:

- **Creational patterns:**solve problems related to the creation of object instances.
- **Structural patterns:**focus on the composition of classes and objects to form larger, more flexible structures.
- **Behavioral patterns:**focus on how the objects interact and distribute responsibilities.

- **Creational patterns used:**

  - **Singleton:**
    To guarantee the maintainability and flexibility of the service, we chose to use the singleton pattern, which allows a concrete class, which does not change its state, to be instantiated only once.

    This pattern adapts perfectly to the “Song” class, which has the attributes of the song; to the “Album” class, which contains certain songs of an artist and the “Add” class which is responsible for adding ads to the Free User class.

    If we detail the class “song”, the objects that are in this class are static, so it would not make sense to give each user that plays a certain song a different memory space for the same song. It is convenient to use a singleton pattern in which you have a single object that is instantiated only once and for each user who needs the information of this object that has been instantiated by another user (artist) obtains the same information.This helps in memory consumption performance by not having to instantiate so many objects with the same information.

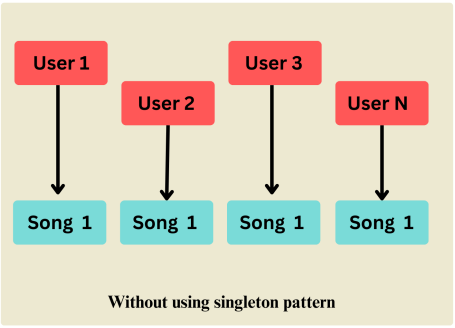

Fig. 4. Without using singleton pattern: Example diagram of how several classes trying to access different instances of objects with the same characteristics behave.

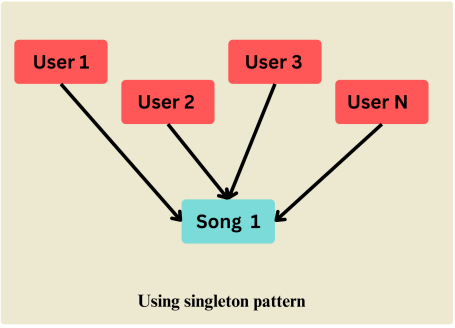

Fig. 5. Using singleton pattern: Example diagram in which several classes access the instance of a single object .

- **Behavioral patterns used:**
  - **State:**
    The State pattern allows an object to change its behavior according to its current state, so the object will appear to change its class. This is achieved by polymorphism and obeys the solid principles of open closed and single responsability.

    This pattern is implemented since a user can take two forms of state: “FreeUser” and “SubscribedUser”.

    A free user does not allow you to have certain benefits that a subscribed user would have, so when a user makes a monthly payment to the platform it automatically adopts other methods that the free user would not have.

- **Command:**
  The command pattern is a behavior pattern whose functionality is to encapsulate a request for some operation under a method, which converts a request into an independent object containing all the information about the request. This transformation allows you to pass requests as an argument to a method, delay or queue the execution of a request, and support operations that can be undone. focus on the composition of classes and objects to form larger, more flexible structures.
  What you want to do by implementing this design pattern is that a user chooses a song to play and when listening to a song continues adding songs to listen to later, so it would be adding objects with internal requests in a queue of requests.
  When the user selects to play or queue a song this request becomes a separate object containing all the information about the request.

V. Antipatterns

A design antipattern is a design pattern that invariably leads to a bad solution to a problem. Unlike design patterns, which offer good solutions, anti-patterns instead of solving a problem quickly and optimally add complexity to the code without the need for it.

For the realization of this web service we wanted to implement many more patterns, however it was analyzed that implementing them became counterproductive. This respects the principle of "Smokes and Mirrors" what its use is to show how a functionality will be before it is implemented.

Besides this, containing many patterns in the same code does not mean that the code is well implemented, on the contrary, it runs the risk of being overloaded, this because it lacks more knowledge in implementing certain "Bad Management" patterns.

Some anti-patterns that allowed us to take successful paths are the following:

- **Observer Pattern:**
  The observer is an example of antipatterns, since we must not force a pattern to the program implementation. it is used when you need to notify state changes to an object. It is composed of three classes, the subject (object under observation), the observer and the client (who receives the notifications).

  For the project we wanted to implement it in such a way that the user would be the observed class, since a user can change its state from free user to susbcritedUser. As the user is the class which is changing the observer notifies the album class so that it could in this way change its state as well, however, this would be somewhat counterproductive, the ideal is to leave the album class in a concrete form without it changing its state. The user class depending on its state changes its methods and applies them to the static album class, which is achieved with the pattern State.

- **Proxy pattern:**
  There are several types of proxy, including the protection proxy, which controls access to the original object, and the intelligent reference proxy pattern, which is a substitute for a simple pointer that performs additional operations when an object is accessed, both of which allow additional maintenance tasks to be performed when an object is accessed.

  To handle the access control we wanted to use the proxy pattern that works as an intermediary between the "Stateuser" class and the "song" class. When a user tries to access the class "song" what the proxy pattern does is to control the access to the concrete class "song" and obtain the information, performing additional operations depending on the state of the user without damaging the logic of the class that is being accessed. Therefore, if a free user wants to listen to music, what the proxy pattern does is to verify the state of the user and as it is a free user, it performs the function of adding advertisements to the song that the user is going to listen to without affecting the functionalities or the internal logic of the class "song".

  Implementing the proxy pattern gave more difficulties than solutions since "song" and "proxy" would have to inherit from the "user" class, which by structure is wrong. The solution given to this anti-pattern is to create an "Add" class which directly adds advertisements to the "FreeUser" class.

VI. Code Smells

Code Smells are traces in the code that indicate a deeper problem in the application or code base. They are not bugs or errors, but tangible and observable violations of the fundamentals of code design/development that could ultimately lead to poor code quality and technical debt.

To fix the code smells that existed at the time of developing the software, we had to refactor, splitting functions that were very long and simplifying complex structures, as when we wanted to implement the proxy, since this was a complex structure that could be solved in a better way by adding a class that added additional operations to one of the User states.

To make the code readable and understandable by anyone other than those who were making the software, we tried to handle variables and class names that are intuitive and easy to understand. In addition to this we tried to avoid making "spaghetti code" and added comments necessary to describe

certain features that the code had. At the time of making comments should be only to what is necessary, since to abuse this can become a code Smell.

In addition to this, to maintain code quality, a Python linter called “PyLint” was used, which helped in verifying errors and imposing coding standards.

Finally, to ensure code optimization, unit tests were performed, which when executed in isolation are designed to verify that the code executes as expected, according to the logic that must respect the requirements.

VII. Solid Principles

In addition to the identification of code smells for clean code, there are the SOLID principles, which are defined as a set of rules and concepts that help developers write clean, scalable and maintainable code.

Design patterns along with the application of code smells are closely related to the implementation of SOLID principles, significantly improving service design and maintainability.

When implementing the State pattern the Single Responsibility Principle is respected, since each state can have its own class, which means that each class has a single responsibility, which is to handle the behavior of that specific state.

The Open/Closed Principle (OCP) is also obeyed, as this principle indicates that a class or functionality must be open for extension, but closed for modification. With the State pattern, new states can be added without modifying existing classes. When applying the command pattern, the Single Responsibility Principle is also respected, since each command has a single responsibility, which is to execute a specific action. In addition to this, the Dependency Inversion Principle (DIP) is respected, since high-level modules can depend on abstractions (command interfaces) instead of concrete details.

Code smells like “Inappropriate inheritance” (which was seen when using the proxy pattern), may indicate that derived classes cannot replace their base classes correctly. Correcting this implements Liskov Substitution Principle

VIII. Results

For the music platform service to work and store information, a database was connected to a docker, using the Postgres database engine. In this database is stored all the information of the users, whether they are subscribed or free, the information of the artists, the albums and the songs in audio format is stored.

As a result, running a virtual environment for python “pyenv”, when performing unit tests of the small parts that make up the system can be seen that the software works as expected and follows a good linearity with the requirements and user stories.

IX. Conclusions

For the development of the music player web page, it can be concluded that a good software quality was maintained, thus meeting the established standards and expectations.
It is demonstrated that the code is functional, since with the unit tests it is verified that the functions are executed correctly and meets the expected requirements.
It is also seen the quality of performance since using the singleton pattern allows us to make a better use of system memory. In addition to this, the maintainability of the software is seen, since by implementing code smells and a good documentation it is possible to maintain and modify the code.

X. References

[1] C. A. Sierra (2024, March 19). Software modeling [Online]. Available: https://github.com/EngAndres/ud-public/tree/main/courses/software-modeling

[2]Refactoring Guru - Design Patterns: Refactoring Guru. Design Patterns, Inc web https://refactoring.guru/es/design-patterns

[3]SourceMaking - Design Patterns: SourceMaking. Design Patterns, Inc web https://sourcemaking.com/design-patterns