# Music streaming platform “Spotify”

Laura Daniela Muñoz Ipus - 20221020022
Juan Diego Alvarez Cristancho - 20221020076
Distrital University
Software modeling

# Objective

Currently with the growth of the 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.

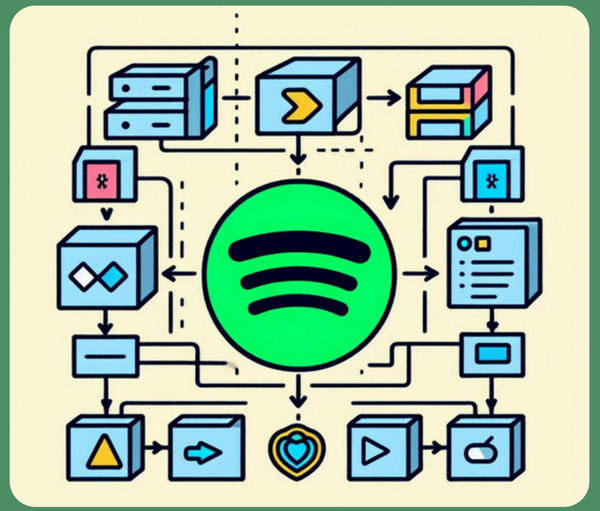
# Business model

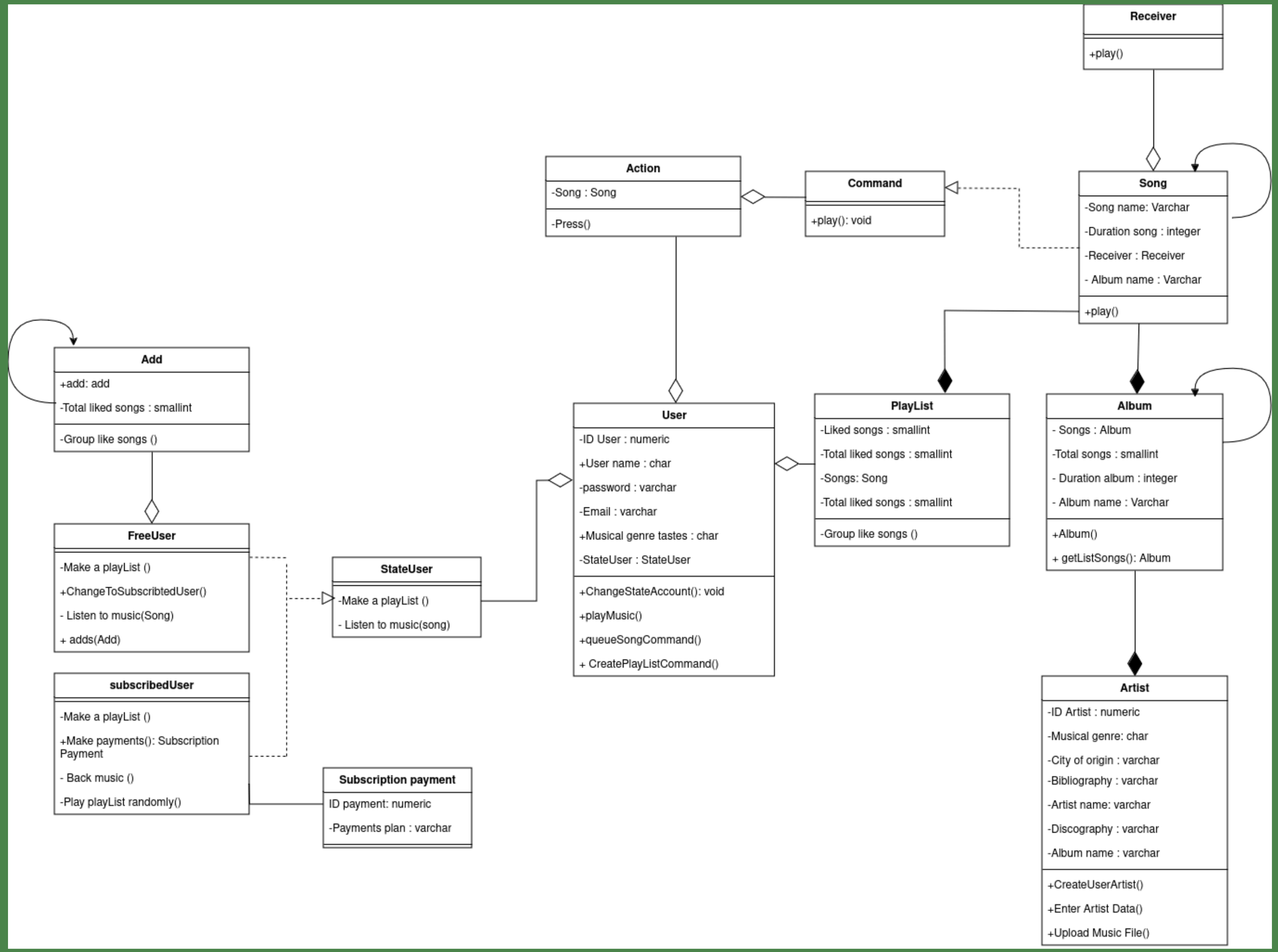
In order to realize a music player you want to take into account business rules, requirements and roles.

# Metodology

In order 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.

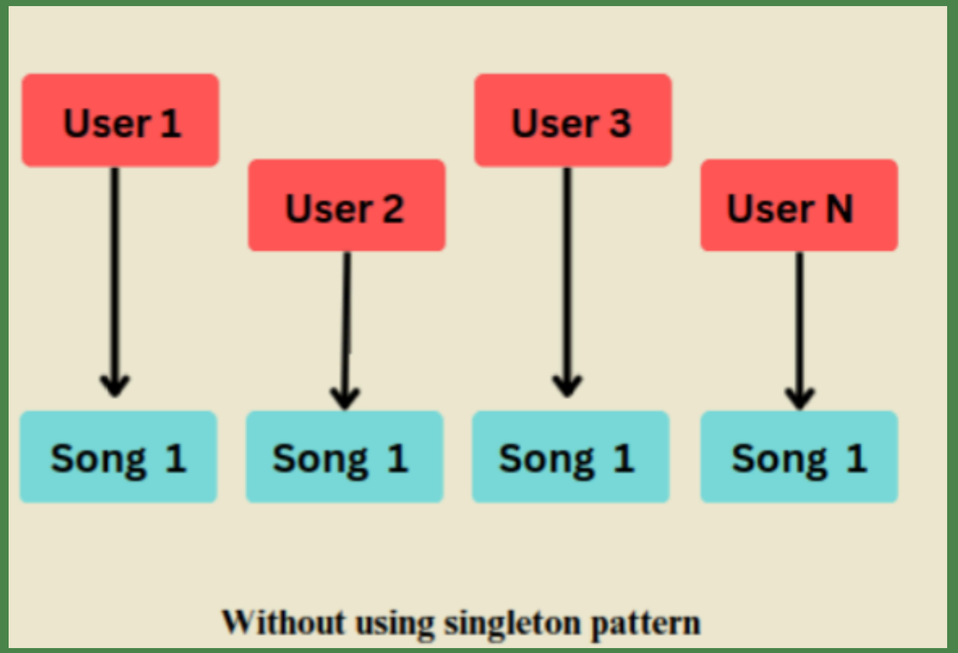
To this proyect we use creational and behavioral patterns: Singleton, State and command

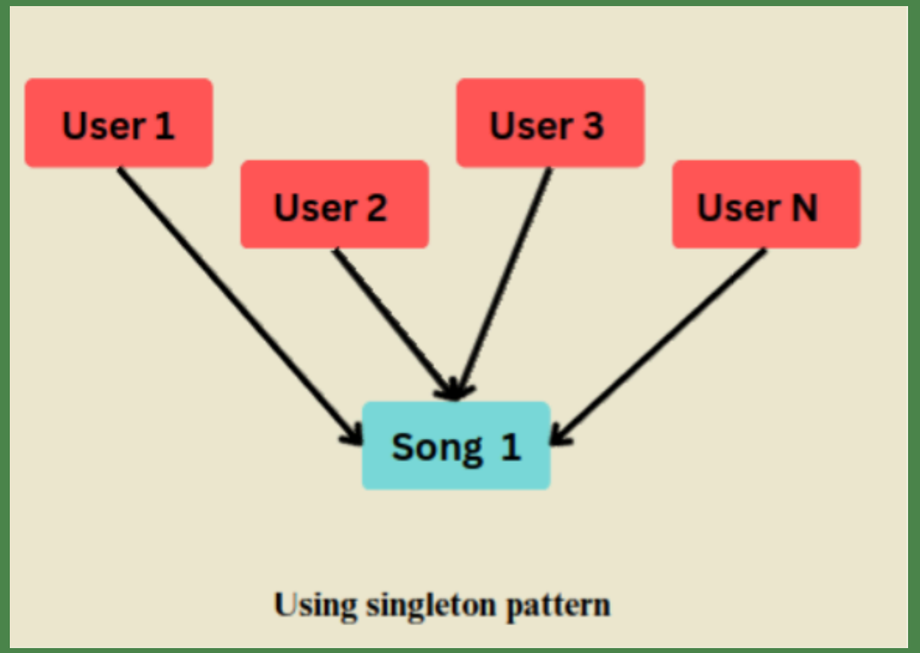



Receiver
+play()
Action
-Song : Song
-Press()
Command
+play(): void
Song
-Song name: Varchar
-Duration song : integer
-Receiver : Receiver
- Album name : Varchar
+play()
Add
+add: add
-Total liked songs : smallint
-Group like songs ()
FreeUser
-Make a playList ()
+ChangeToSubscribtedUser()
- Listen to music(Song)
+ adds(Add)
StateUser
-Make a playList ()
- Listen to music(song)
User
-ID User : numeric
+User name : char
-password : varchar
-Email : varchar
+Musical genre tastes : char
-StateUser : StateUser
+ChangeStateAccount(): void
+playMusic()
+queueSongCommand()
+ CreatePlayListCommand()
PlayList
-Liked songs : smallint
-Total liked songs : smallint
-Songs: Song
-Total liked songs : smallint
-Group like songs ()
Album
- Songs : Album
-Total songs : smallint
- Duration album : integer
- Album name : Varchar
+Album()
+ getListSongs(): Album
subscribedUser
-Make a playList ()
+Make payments(): Subscription Payment
- Back music ()
-Play playList randomly()
Subscription payment
ID payment: numeric
-Payments plan : varchar
Artist
-ID Artist : numeric
-Musical genre: char
-City of origin : varchar
-Bibliography : varchar
-Artist name: varchar
-Discography : varchar
-Album name : varchar
+CreateUserArtist()
+Enter Artist Data()
+Upload Music File()

# Singleton

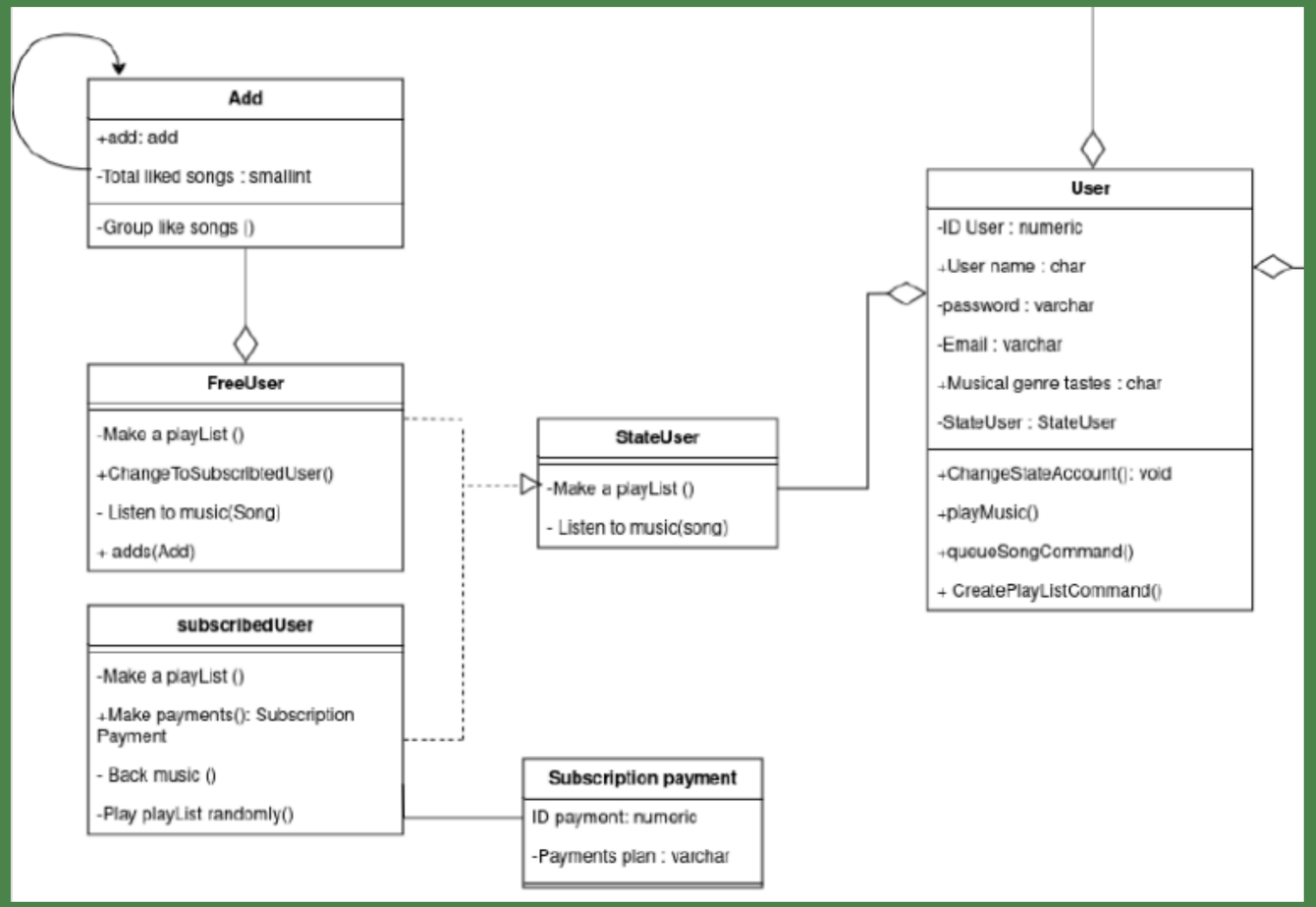
To guarantee the maintainability and flexibility of the service, we chose to use the singleton pattern, which allows a concrete class, which does not change its state, to be instantiated only once.



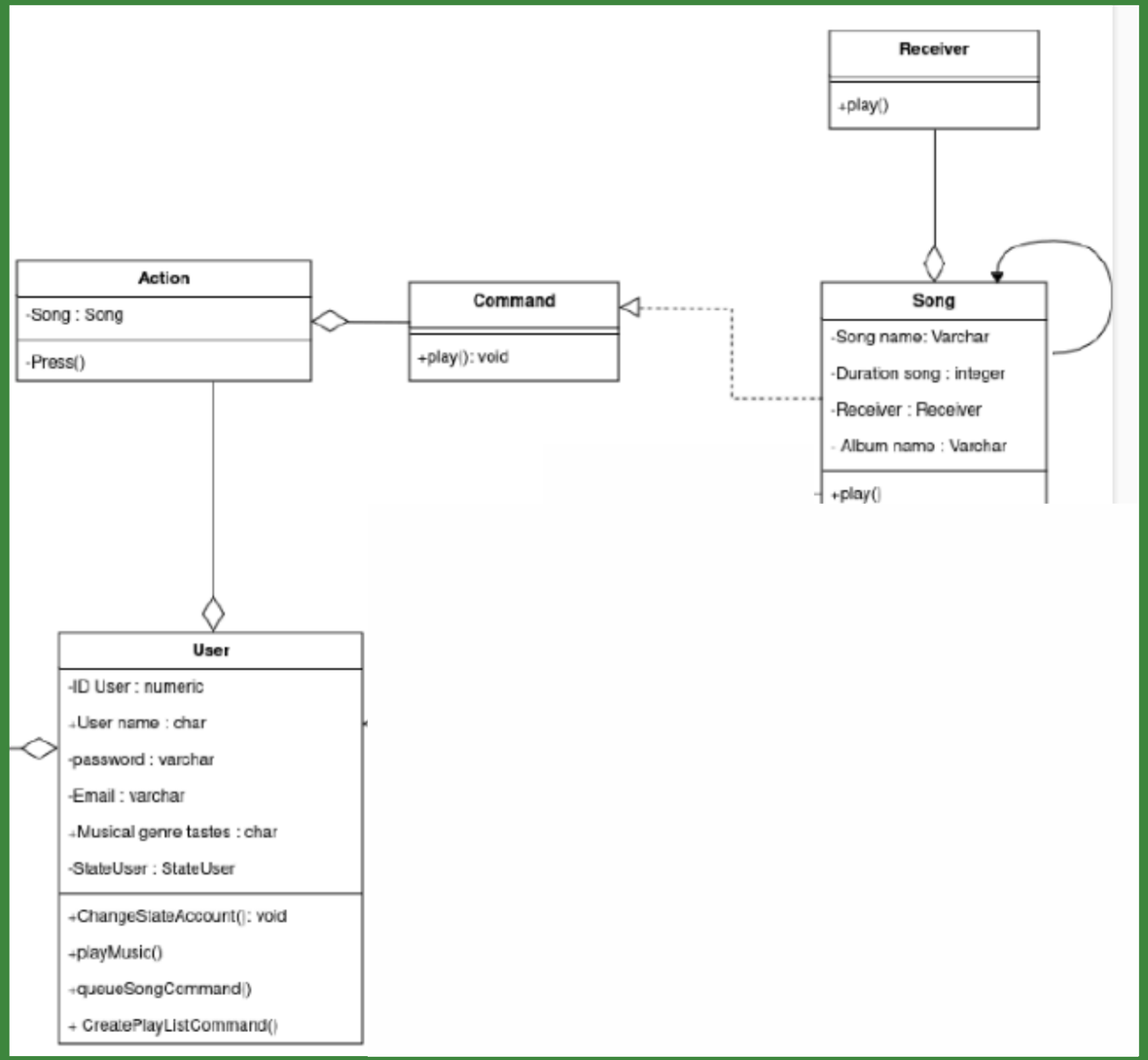
# State

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".



# Command

The command pattern is a behavior pattern whose functionality is to encapsulate a request for some operation under a method.
What you want to do implementing this design pattern is that a user chooses a song to play and when listening to a song continues adding songs to listen later, so it would be adding objects with internal requests in a queue of requests.



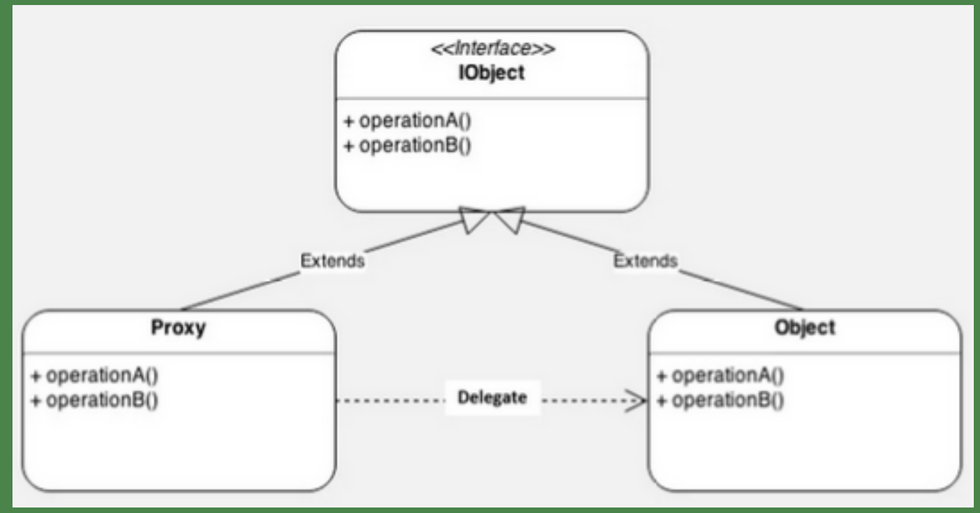
# implementation of antipatterns

A design antipattern is a design pattern that invariably leads to a bad solution to a problem.

## Proxy

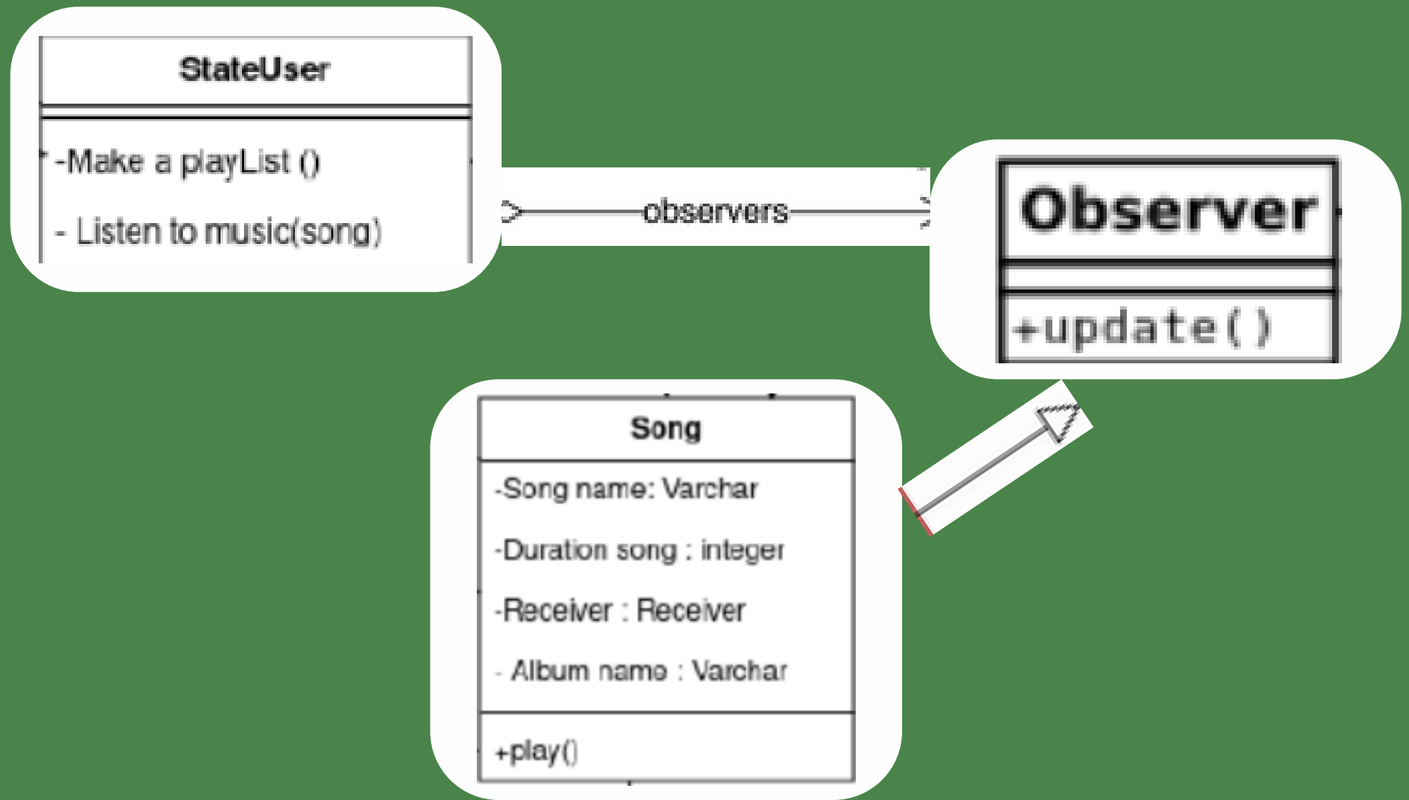
There are several types of proxy, including the protection proxy and the intelligent reference proxy pattern, both of which allow additional maintenance tasks to be performed when an object is accessed.

It gave more difficulties than solutions since "song" and "proxy" would have to inherit from the "user" class, which by structure is wrong.



# Observer

It is composed of three classes, the subject (object under observation), the observer and the client (who receives the notifications).




# Facade

To simplify interactions with complex subsystems such as media playback controls or user account management systems by providing a simplified interface to these subsystems.

# Code Smells

Code Smells are traces in the code that indicate a deeper problem in the application or code base. To solve this problems its necesary:


- refactoring, splitting up functions that were too long and simplifying complex structures
- tried to handle variables and class names in a way that is intuitive and easy to understand
- avoided making "spaghetti code" and added comments necessary to describe certain functionalities.
-Its necesary to use pylint

# principle SOLID

Design patterns together 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 and The Open/Closed Principle (OCP)  is respected

The Single Responsibility and Dependency Inversion Principle is respected when applying the command pattern.

Code smells like "Inappropriate inheritance" (what was seen when using the proxy pattern). Fixing this implements Liskov Substitution Principle

# Results

In order for the music platform service to work and store information, a database was connected to a docker, using the Postgress database engine.

As a result, running a virtual environment for python "pyenv", when performing unit tests of the small parts that make up the system, it can be seen that the software works as expected.

PostgreSQL