Fatima Jinnah Women University

Department of Software Engineering

PROJECT

Course Title

Software Design and Architecture

Submitted To

Dr. Mehreen Sirshar

Submitted By

Mariam Fatima

Section: A (4TH Semester)

Registration No: 2021-BSE-020

Date of Submission

June 20, 2023

SPOTIFY

Spotify is a popular digital music streaming platform that provides users with access to a vast collection of songs, albums, podcasts, and other audio content from various genres and artists. Launched in 2008, Spotify has grown to become one of the leading music streaming services worldwide, with millions of active users.

With Spotify, users can listen to music on-demand, create personalized playlists, discover new artists and songs through curated recommendations, and follow their favorite musicians to stay updated with their latest releases. The platform offers both free and premium subscription options, with the latter providing additional features and an ad-free experience.

DESIGN PATTERNS

A design pattern is a reusable solution to a common problem that occurs in software design and development. It is a proven approach to solving a specific design or architectural issue in a consistent and efficient manner. Design patterns capture best practices and provide a way to communicate and share effective solutions among developers

Following are the design patterns

- Singleton design pattern
- Facade design pattern
- Builder design pattern
- Iterator design pattern

SINGLETON DESIGN PATTERN

The Singleton pattern is a design pattern that ensures the existence of only one instance of a class throughout the application. It provides a global point of access to this instance, allowing other parts of the application to easily access it without the need to create multiple instances.

In the case of Spotify, the Singleton pattern could be applied to a class responsible for managing the user's playback session. This class would handle tasks such as playing, pausing, skipping tracks, and managing the user's current state, including the active playlist and playback settings.

CLASSES

Spotify: The Spotify class is the base class that implements the Singleton pattern. It includes the special new_() Spotify responsible for controlling the creation Spotify instance.

Spotify Service: The Spotify Service class represents a service within the Spotify application. It has a private property token to store the authentication token. The public methods setToken (), play (), and pause () allow setting the token, playing music, and pausing the playback, respectively.

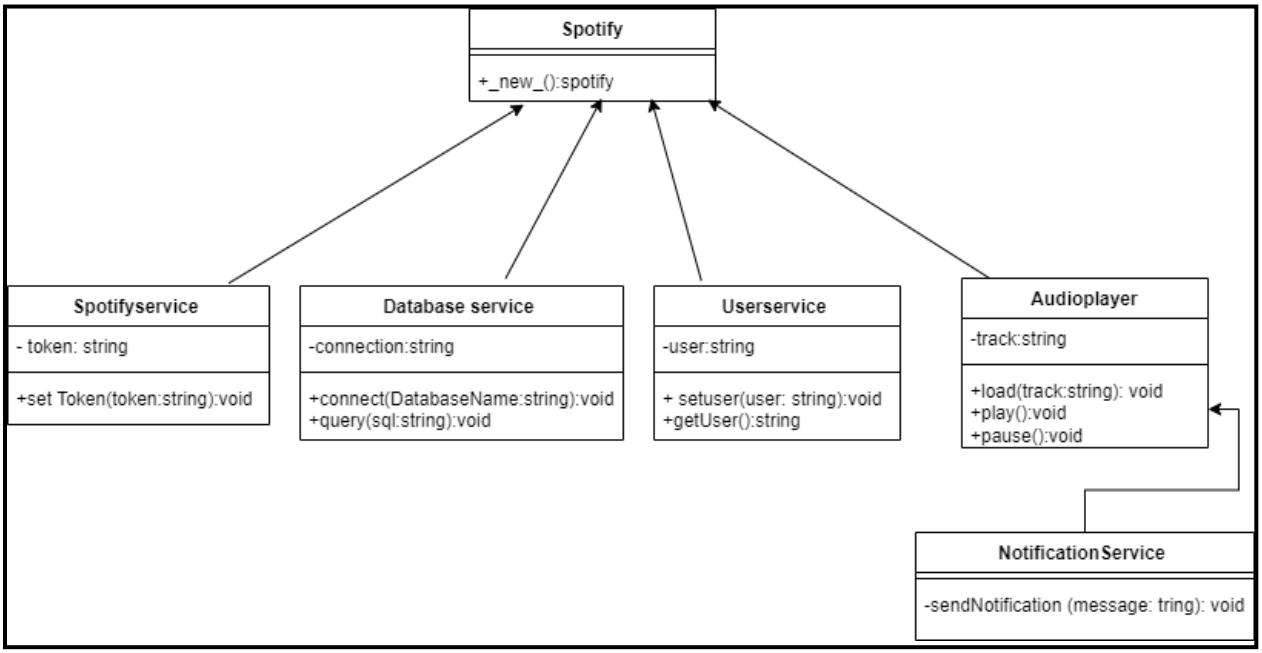
Database Service: The Database Service class represents a service responsible for database operations. It has a private property **connection** to store the database connection details.

The public methods **connect () and query ()** handle connecting to a specific database andexecuting SQL queries, respectively.

User Service: The User Service class manages user-related operations. It has a private property user to store user information. The public methods set User () and get User () allow setting and retrieving user information.

Audio Player: The Audio Player class represents the audio playback component in the Spotify application. It has a private property track to store the currently loaded track. The public methods load (), play(), and pause() handle loading a track, playing it, and pausing the playback, respectively.

Notification Service: The Notification Service class represents a service responsible for sending notifications. It provides a public method send Notification () to send a notification message.



The relationship between Audio Player and Notification Service indicates that the Audio Player class uses the Notification Service to send notifications. This could be used, for example, to notify the user about the playback status.

BENEFITS

- Single Instance
- Global Access
- Threat Safety
- Resource Management
- Flexibility
- Simplifies Dependency Management

FACADE DESIGN PATTERN

The Facade design pattern is a structural design pattern that provides a simplified interface to a complex subsystem or set of classes. It acts as a unified interface that encapsulates a group of interfaces or classes, making them easier to use and understand.

The **main purpose of the Facade pattern is to hide the complexity of a system** and **provide a simpler interface for clients to interact with**. The Facade pattern typically consists of a single class, called the facade, which serves as an entry point for the client code. The facade class delegates the client's requests to the appropriate methods or objects within the subsystem. It coordinates the interactions between the client and the subsystem, shielding the client from the complexities of the subsystem's components.

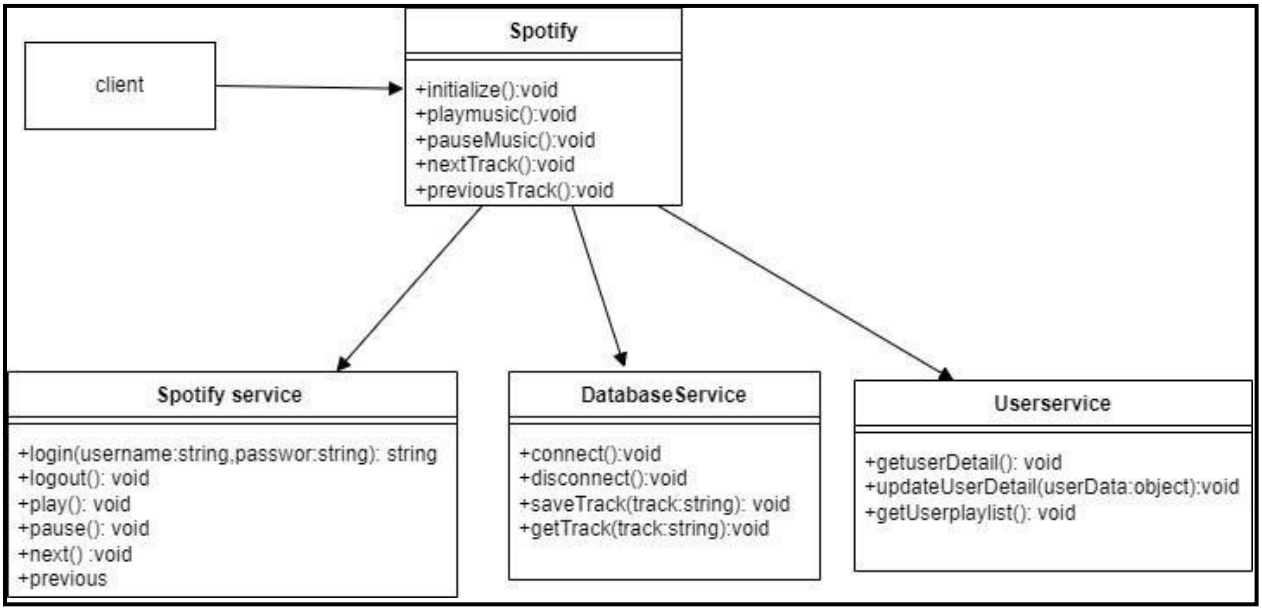
CLASSES

The Spotify class represents the facade, which provides a simplified interface to the complex subsystem of the Spotify application. It encapsulates the functionality of multiple underlying services and provides a unified set of methods to the client.

The **Spotify Service** class represents the service responsible for handling Spotify-related functionalities such as login, logout, playback control (play, pause, next, previous), and other Spotify-specific operations.

The **Database Service** class represents the service responsible for handling database-related functionalities such as connecting to the database, disconnecting, and performing operations related to saving and retrieving tracks.

The **User Service** class represents the service responsible for managing user-related functionalities, such as retrieving user details, updating user details, and accessing user playlists.



The Spotify class acts as an interface or entry point for the client to interact with the Spotify application. It encapsulates the complexity of the underlying services and provides simplified methods such as initialize, play Music, pause Music, next Track, and previous Track.

BENEFITS

- Simplified interface
- Encapsulation and abstraction
- Improved code readability and maintainability
- Reduced coupling and dependencies
- Performance optimization
- Enhanced test-ability
- Support for complex object creation

ITERATOR DESIGN PATTERN

The Iterator design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It separates the traversal of a collection from the specific structure of the collection, making it easier to iterate over various types of collections uniformly.

CLASSES

The **Song class** represents individual songs with attributes title and artist.

The **Playlist** class represents a playlist with an array of songs as its attribute. It provides methods to add songs and create an iterator.

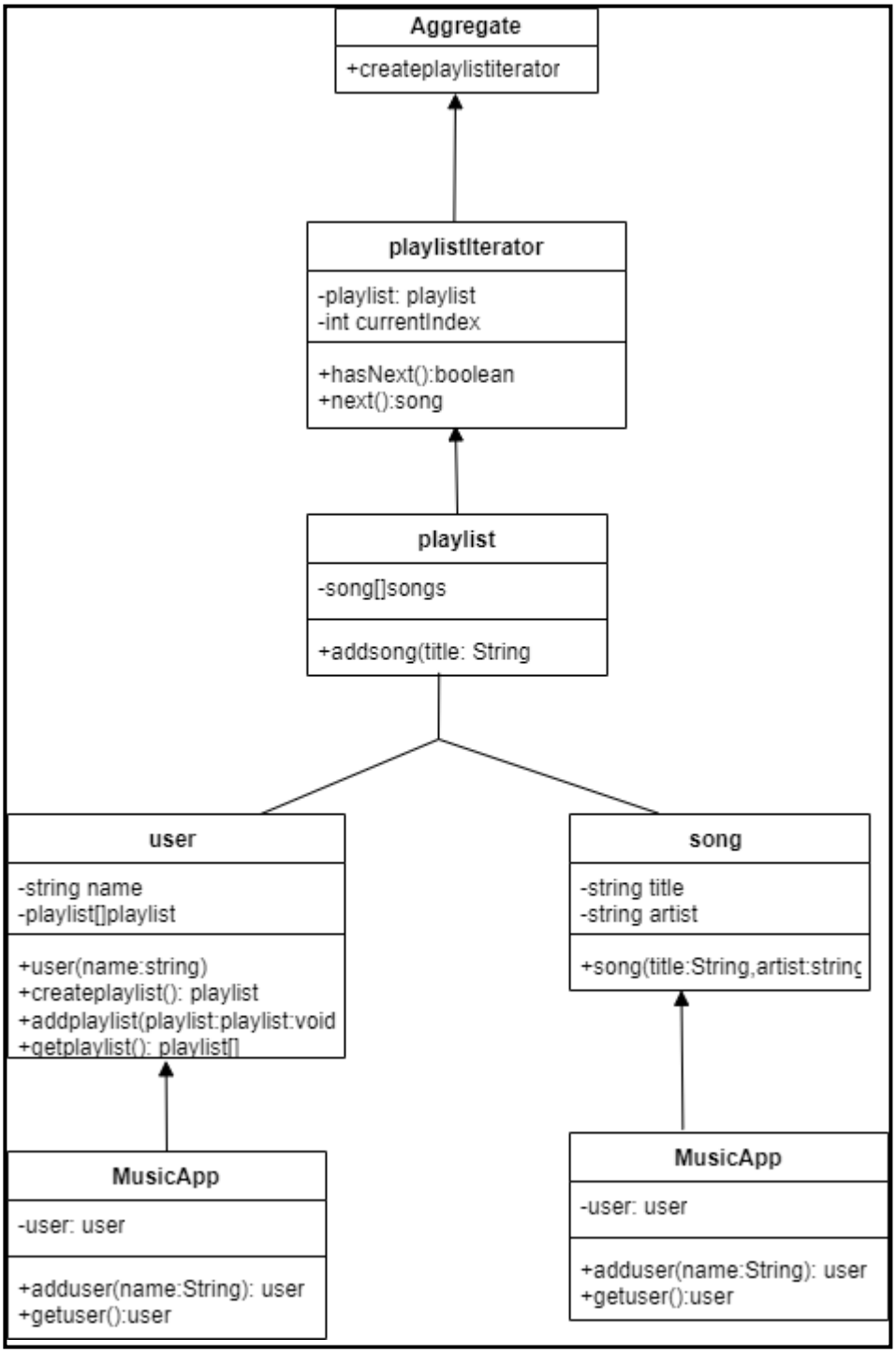
The **Playlist Iterator** class handles the iteration logic. It has attributes for the playlist and the current index, as well as methods to check if there are more songs and retrieve the next song.

The **User** class represents a user of the music app and has attributes `name` and an array playlists to store the playlists created by the user. It provides methods to create a new playlist, add a playlist to the user's collection, and retrieve the user's playlists.

The **Music App** class represents the music application itself. It has an array users to store the users of the app. It provides methods to add a new user to the app and retrieve the list of users.

BENEFITS

- Uniform interface
- Simplifies client code
- Encapsulation of iteration logic
- Supports different traversal algorithms
- Enables concurrent iteration


Aggregate
+createplaylistiterator
playlistIterator
-playlist: playlist
-int currentIndex
+hasNext():boolean
+next():song
playlist
-song[]songs
+addsong(title: String
user
-string name
-playlist[]playlist
+user(name:string)
+createplaylist(): playlist
+addplaylist(playlist:playlist:void
+getplaylist(): playlist[]
song
-string title
-string artist
+song(title:String,artist:string
MusicApp
-user: user
+adduser(name:String): user
+getuser():user
MusicApp
-user: user
+adduser(name:String): user
+getuser():user

BUILDER DESIGN PATTERN

The Builder design pattern is a creational pattern that provides a way to construct complex objects step by step. It separates the construction of an object from its representation, allowing the same construction process to create different representations.

The main idea behind the Builder pattern is to encapsulate the construction logic of an object within a separate builder object. The builder object contains methods for setting the desired properties or attributes of the object being built. Each method typically returns the builder object itself, allowing method chaining and creating a fluent interface.

CLASSES

User Song: Represents a song in the context of a user. It has attributes title and artist and provides methods to get the title and artist of the song. User Playlist: Represents a playlist in the context of a user. It has attributes name and a list of

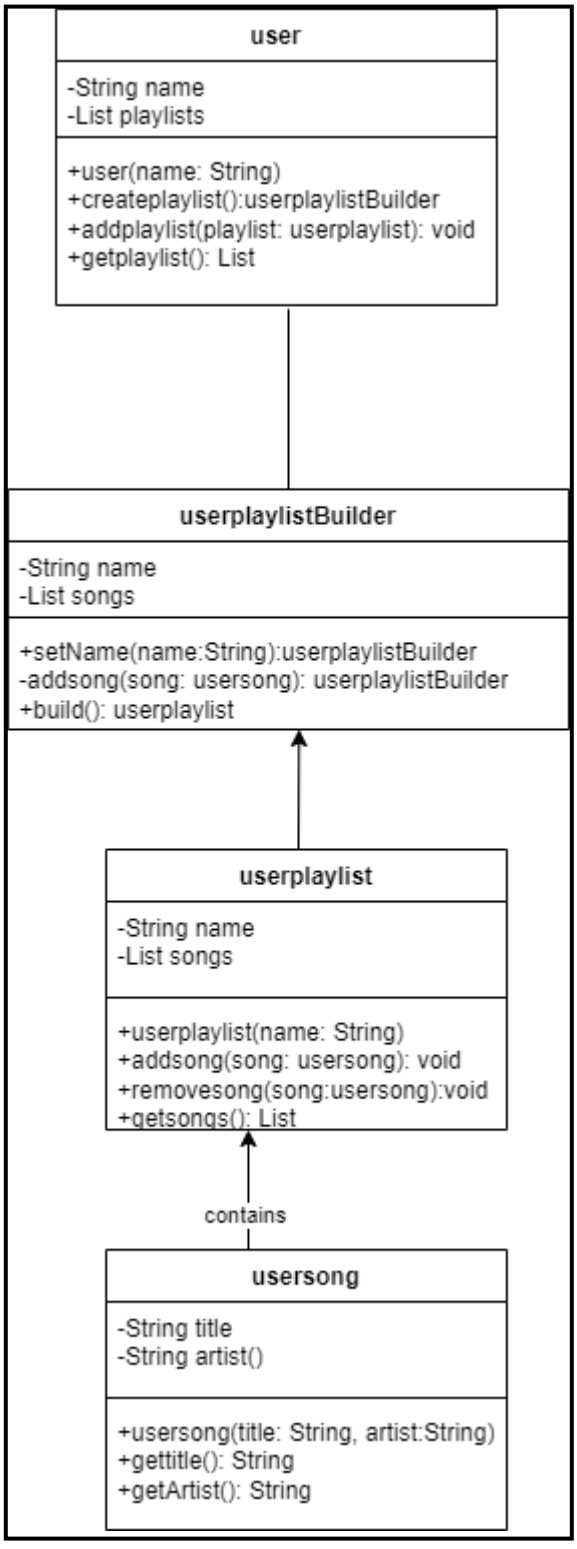
UserPlaylistBuilder: Implements the builder pattern for creating user playlists. It has attributes for name and a list of User Song objects being added. It provides methods to set the name of the playlist, add songs to it, and build the final playlist object.

User: Represents a user in the Spotify application. It has attributes name and a list of User Playlist objects associated with the user. It provides methods to create a playlist using the builder pattern, add playlists to the user's collection, and retrieve the user's playlists.

Spotify: Represents the Spotify application. It has a list of User objects registered on the platform. It provides methods to add users to the application and retrieve the list of users.

BENEFITS

- Improved code readability
- Encapsulated construction logic
- Separation of construction and representation


user
-String name
-List playlists
+user(name: String)
+createplaylist():userplaylistBuilder
+addplaylist(playlist: userplaylist): void
+getplaylist(): List
userplaylistBuilder
-String name
-List songs
+setName(name:String):userplaylistBuilder
-addsong(song: usersong): userplaylistBuilder
+build(): userplaylist
userplaylist
-String name
-List songs
+userplaylist(name: String)
+addsong(song: usersong): void
+removesong(song:usersong):void
+getsongs(): List
contains
usersong
-String title
-String artist()
+usersong(title: String, artist:String)
+gettitle(): String
+getArtist(): String