



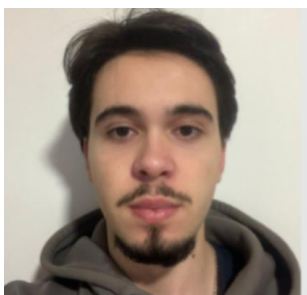
Universidade do Minho

Licenciatura em Engenharia Informática

Unidade Curricular de Programação Orientada a Objetos

Ano letivo 2024-2025

Grupo-25



Diogo Alves

A106904



Hugo Cunha

A106808



José Rocha

A106887

## Índice

<b>1. Introdução.....</b>	<b>4</b>
<b>2. Classes.....</b>	<b>4</b>
2.1 Musica.....	4
2.2 MusicaExplicita.....	5
2.3 MusicaMultimedia.....	5
2.4 Interface PlanoSubscricao .....	5
2.5 PlanoFree, PlanoPremiumBase e PlanoPremiumTop .....	6
2.6 Biblioteca .....	6
2.7 Utilizador .....	7
2.8 Playlist.....	7
2.9 PlaylistAleatoria.....	7
2.10 PlaylistConstruida.....	8
2.11 PlaylistFavoritos.....	8
2.12 PPlaylistGenero.....	9
2.13 Album .....	9
2.14 Artista .....	10
2.15 Reproducao .....	10
2.16 GestorFicheiros.....	10
2.17 Enum Cargo.....	11
2.18 Enum Genero .....	12
2.19 SpotifUM .....	12
2.20 Controller .....	15
2.21 ViewAdmin.....	15
2.22 ViewLogin.....	15
2.23 ViewPlaylist.....	16
2.24 ViewStats.....	16
2.25 ViewUtilizador .....	16
2.26 Main.....	16
2.27 Exceptions .....	17
2.28 Testes .....	17
<b>3. Diagrama de Classes .....</b>	<b>19</b>

<b>4. Estrutura do Projeto.....</b>	<b>20</b>
<b>5. Conclusão.....</b>	<b>21</b>

# 1. Introdução

Este projeto consistiu no desenvolvimento de uma aplicação de gestão e reprodução de música — **SpotifUM** — realizada no âmbito da Unidade Curricular de Programação Orientada a Objetos. Esta aplicação tem como objetivo simular algumas das principais funcionalidades das plataformas de streaming musical, como a criação de playlists, reprodução de músicas e análise de estatísticas de utilização, a aplicação suporta ainda a persistência de dados permitindo guardar e carregar o estado completo do sistema a partir de ficheiros.

Ao longo do relatório, serão apresentados a arquitetura de classes adotada, as principais funcionalidades implementadas, o diagrama de classes que representa visualmente a estrutura do sistema, a organização interna do projeto e, por fim, uma conclusão com as principais considerações sobre o sistema desenvolvido.

## 2. Classes

Nesta secção apresentamos as principais classes desenvolvidas para o projeto, organizadas por funcionalidades e hierarquias.

Grande parte das classes implementa a interface **Serializable**, para permitir que os objetos da classe possam ser convertidos para um fluxo de bytes. Assim, é possível guardar e recuperar o estado dos objetos em ficheiros.

Além disso, praticamente todas as classes possuem os construtores habituais (por omissão, parametrizado e de cópia), métodos getters e setters, bem como os métodos base equals, toString e clone, implementados conforme necessário para garantir a correta comparação, representação textual e clonagem de objetos.

### 2.1 Musica

```
public class Musica implements Serializable {  
    private String nome;  
    private Artista interprete;  
    private String nomeEditora;  
    private List<String> letra;  
    private List<String> musica;  
    private Genero genero;  
}
```

```
private int duracao;  
private int numReproducoes;
```

A classe **Musica** possui os atributos mínimos pedidos no enunciado, no entanto decidimos adicionar o **numReproducoes**, este será incrementado no método **reproduzir()** sempre que a música for reproduzida por um utilizador.

## 2.2 MusicaExplicita

```
public class MusicaExplicita extends Musica implements Serializable {  
    private String motivoAviso;  
    private int idadeMinima;
```

A classe **MusicaExplicita** é uma subclasse da classe **Musica**, que representa músicas com conteúdo explícito. Para além dos atributos herdados da classe **Musica** possui ainda:

- String **motivoAviso** – Indica o motivo da música ser classificada como explícita.
- Int **IdadeMinima** – Idade mínima para reproduzir a música.

## 2.3 MusicaMultimedia

```
public class MusicaMultimedia extends Musica implements Serializable {  
    private String nomeVideo;  
    private String formato;
```

A classe **MusicaMultimedia** também é uma subclasse da classe **Musica**. Para além dos atributos herdados da classe **Musica**, possui ainda:

- String **nomeVideo** – Indica o nome do vídeo associado à música.
- String **formato** – Indica o formato do vídeo.

## 2.4 Interface PlanoSubscricao

Esta interface possui os métodos necessários para a implementação dos diferentes planos de subscrição, incluindo:

- `int pontosPorMusica(int pontosAtuais, booleana musicaNova)` - Que irá atualizar os pontos do utilizador de acordo com o seu plano e se a música for reproduzida pela primeira vez, no caso do **PlanoPremiumTop**.
- `boolean podeCriarPlayLists()` - Que verifica a permissão do usuário para criar uma playlist.
- `boolean podeAcederAFavoritos()` - Que verifica a permissão do usuário aceder às músicas favoritas.
- `String getNomePlano()` - Que retorna o nome do plano de subscrição.

## 2.5 PlanoFree, PlanoPremiumBase e PlanoPremiumTop

As classes **PlanoFree**, **PlanoPremiumBase** e **PlanoPremiumTop** implementam a interface **PlanoSubscricao**.

Cada uma representa um tipo distinto de subscrição, diferenciando-se pela forma como os pontos são atribuídos e pelas permissões concedidas ao utilizador (como criar playlists ou aceder a favoritos).

- **PlanoFree**: 5 pontos por música; sem acesso a favoritos ou playlists.
- **PlanoPremiumBase**: 10 pontos por música; acesso a favoritos, mas não à criação de playlists.
- **PlanoPremiumTop**: Bónus inicial de 100 pontos e 2.5% dos pontos acumulados por música nova; acesso total a funcionalidades.

## 2.6 Biblioteca

```
public class Biblioteca implements Serializable {
    private ArrayList<Playlist> playlist;
    private ArrayList<Album> albuns;
```

A classe Biblioteca é utilizada para armazenar as playlists do utilizador assim como os seus albuns, para isso inclui os argumentos:

- `ArrayList<Playlist> playlist`
- `ArrayList<Album> albuns`

Nesta classe também são incluídos métodos de adição e remoção de playlists e albuns.

## 2.7 Utilizador

```
public class Utilizador implements Serializable {  
    private String nome;  
    private String email;  
    private String morada;  
    private String password;  
    private int pontos;  
    private PlanoSubscricao planoSubscricao;  
    private Cargo cargo;  
    private Biblioteca biblioteca;
```

A classe **Utilizador** representa o utilizador da aplicação, contendo as suas informações pessoais, plano de subscrição e biblioteca associada.

O email serve como identificador único.

Inclui o método **ganharPontos()** que atualiza os pontos com base no plano.

## 2.8 Playlist

```
public abstract class Playlist implements Serializable {  
    private String nome;  
    private Utilizador utilizador;  
    private LocalDate dataCriacao;  
    private boolean publica;  
    private ArrayList<Musica> listaMusicas;
```

Na classe **Playlist** é uma classe abstrata que define a estrutura base de uma playlist.

Possui métodos que permitem definir se é pública ou privada, calcular a sua duração total, reproduzir as suas músicas, e controlar o seu acesso.

## 2.9 PlaylistAleatoria

A classe **PlaylistAleatoria** é uma subclasse da classe **Playlist**.

Esta para além dos métodos base também inclui um método `ArrayList<Musica> embaralhar()`, que organiza as músicas em ordem aleatória.

## 2.10 PlaylistConstruida

```
public class PlaylistConstruida extends Playlist implements Serializable {  
  
    private int indiceAtual = 0;  
    private boolean aleatorio = false;  
    private ArrayList<Musica> ordemReproducao;  
}
```

A classe **PlaylistConstruida** é uma subclasse da classe **Playlist**.

Esta classe representa uma playlist construída manualmente por um utilizador Premium ou por um administrador. Permite controlar a ordem de reprodução das músicas e alternar entre dois modos: **sequencial** e **aleatório**. Para isso, inclui os seguintes atributos adicionais:

- int **indiceAtual** – Índice da música atual na ordem de reprodução.
- boolean **aleatorio** – Indica se o modo aleatório está ativo.
- ArrayList<Musica> **ordemReproducao** – Lista interna que representa a ordem de reprodução, que pode ser embaralhada.

Além dos métodos base herdados de Playlist, esta classe inclui os seguintes métodos específicos:

- String **reproduzirAtual()** – Reproduz a música atual com base no indiceAtual, lança uma exceção PlaylistVaziaException se não houver músicas.
- String **avancar()** – Avança para a próxima música. Se for a última, reinicia a partir da primeira.
- String **retroceder()** – Retrocede para a música anterior. Se for a primeira, vai para a última.
- void **ativarModoAleatorio()** – Ativa o modo aleatório e embaralha a ordem de reprodução.
- void **desativarModoAleatorio()** – Desativa o modo aleatório e restaura a ordem original.

## 2.11 PlaylistFavoritos

Por sua vez a classe **PlaylistFavoritos** é também uma subclasse da classe **Playlist**, não possui atributos adicionais nem métodos adicionais para além dos métodos base, serve apenas para classificação da playlist onde serão armazenadas as músicas favoritas do utilizador.



## 2.12 PlaylistGenero

```
public class PlaylistGenero extends Playlist implements Serializable {  
    private Genero genero;  
    private int duracaoMaxima;
```

A classe **PlaylistGenero** é também uma subclasse da classe **Playlist**.

Esta playlist permite criar uma lista de músicas filtrada por género musical e com uma duração máxima definida. É útil para utilizadores que pretendem ouvir apenas músicas de um determinado género dentro de um limite de tempo.

Para isso, esta classe inclui os seguintes atributos adicionais:

- Género **genero** – Representa o género musical da playlist.
- int **duracaoMaxima** – Duração máxima permitida (em segundos) para a playlist.

## 2.13 Album

```
public class Album implements Serializable {  
    private String nome;  
    private LocalDate dataLancamento;  
    private Artista autor;  
    private ArrayList<Musica> musicas;
```

A classe **Album** representa um conjunto de músicas lançado por um artista numa determinada data. Permite a organização e gestão de músicas por parte de artistas, facilitando o acesso e reprodução em grupo.

Esta classe inclui os seguintes atributos:

- String **nome** – Nome do álbum.
- LocalDate **dataLancamento** – Data de lançamento do álbum.
- Artista **autor** – Artista responsável pelo álbum.
- ArrayList<Musica> **musicas** – Lista de músicas que compõem o álbum.

Foram implementados os seguintes métodos, para além dos base:

- int **getDuracaoTotal()** – Calcula e retorna a duração total (em segundos) de todas as músicas presentes no álbum.

- void **adicionarMusica(Musica musica)** – Adiciona uma música à lista de músicas do álbum, criando uma cópia da música recebida.
- void **removerMusica(Musica musica)** – Remove uma música específica da lista de músicas, caso ela exista no álbum.

## 2.14 Artista

```
public class Artista implements Serializable {
    String nome;
    String pais;
```

A classe **Artista** representa o interprete das músicas, contendo as suas informações de nome e pais. Inclui apenas os métodos base.

## 2.15 Reproducao

```
public class Reproducao implements Serializable {
    private Utilizador utilizador;
    private Musica musica;
    private LocalDateTime dataHora;
```

A classe **Reproducao** representa uma reprodução de uma música por um utilizador, armazenando informações sobre a música que foi reproduzida, o utilizador que a ouviu e o momento da reprodução. Inclui apenas os métodos base.

## 2.16 GestorFicheiros

```
public class GestorFicheiros {
    public static final String SPOTIFUMFILE = "src/Ficheiros/SpotifUM.dat";
    public static final String SCRIPTFILE = "src/Ficheiros/scriptExemplo.txt";
```

A classe **GestorFicheiros** é responsável pela manipulação de ficheiros para armazenar e recuperar o estado do sistema, bem como processar comandos a partir de um ficheiro de script.

Constantes:

- **SPOTIFUMFILE**: Caminho do ficheiro onde o estado do sistema é armazenado.

- **SCRIPTFILE:** Caminho do ficheiro de script de exemplo.

Métodos principais:

- **guardarEstado(SpotifUM spotifUM, String nomeFicheiro):**  
Salva o estado do sistema SpotifUM em um ficheiro, usando serialização.
- **carregarEstado(String nomeFicheiro):**  
Carrega o estado do sistema a partir de um ficheiro previamente serializado.
- **carregarScript(SpotifUM sistema):**  
Lê e processa um ficheiro de script linha por linha, realizando ações no sistema.
- **processarLinha(SpotifUM sistema, String linha):**  
Processa uma linha do ficheiro de script, interpretando o comando a ser executado.
- **processarComandoAdmin(SpotifUM sistema, String comando):**  
Executa comandos específicos para o administrador, como criação de artistas, músicas, álbuns e registo de usuários.

Esses métodos permitem que o sistema carregue e guarde seu estado e também processe comandos a partir de um ficheiro de script para alterar ou interagir com o sistema.

## 2.17 Enum Cargo

```
public enum Cargo {  
    USER, ADMIN  
}
```

A enumeração **Cargo** define os diferentes cargos que um utilizador pode ter no sistema. Neste caso, existem dois cargos possíveis:

- **USER:** Representa um utilizador comum.
- **ADMIN:** Representa um utilizador administrador.

A enumeração **Cargo** é usada para atribuir e verificar o cargo de um utilizador no sistema, permitindo a diferenciação entre os direitos e permissões de cada tipo de utilizador.

## 2.18 Enum Genero

```
public enum Genero {  
    ROCK,  
    POP,  
    JAZZ,  
    HIPHOP,  
    CLASSICA,  
    ELETRONICA,  
    FUNK,  
    METAL,  
    OUTRO  
}
```

A enumeração **Genero** define os diferentes gêneros musicais possíveis no sistema. Cada valor dentro da enumeração corresponde a um gênero musical específico, o que facilita a organização e a filtragem de músicas dentro do sistema.

## 2.19 SpotifUM

```
public class SpotifUM implements Serializable {  
    private HashMap<String, Utilizador> utilizadores;  
    private ArrayList<Reproducao> reproducoes;  
    private ArrayList<Musica> musicas;  
    private ArrayList<Album> albuns;  
    private ArrayList<Playlist> playlists;  
    private HashMap<String, Artista> artistas;
```

A classe **SpotifUM** representa o centro da aplicação. É responsável por armazenar e manipular toda a informação relativa a utilizadores, artistas, músicas, álbuns, playlists e reproduções. Além disso, centraliza toda a lógica de funcionamento associada à interação entre estes elementos.

Para o armazenamento dos dados, foram utilizadas as estruturas HashMap e ArrayList, escolhidas pela sua eficiência e adequação às operações mais comuns do sistema.

Os HashMap permitem acessos rápidos a elementos através de chaves únicas, como o e-mail de um utilizador ou o nome de um artista, garantindo um desempenho eficiente em operações de procura e inserção.

Por sua vez, os ArrayList são utilizados para manter coleções ordenadas e

dinâmicas de elementos, como listas de músicas, álbuns ou reproduções, sendo ideais para iteração sequencial e manipulação de listas.

Esta classe inclui os métodos habituais, garantindo a encapsulação e integridade dos dados. Adicionalmente, estão implementados vários métodos específicos, que podem ser divididos pelas seguintes áreas:

#### *Gestão de Utilizadores*

Trata da criação, verificação e login de utilizadores. Permite registar novos utilizadores no sistema, seja por interface ou por script, bem como realizar o login e verificar permissões associadas ao plano de subscrição.

- `registarUtilizador(...)`
- `registarUtilizadorPorScript(...)`
- `loginUtilizador(...)`
- `validarUtilizadorPremium(...)`

#### *Acesso e Consulta de Objetos*

Métodos de pesquisa que permitem obter entidades específicas com base em atributos chave, como o nome ou o e-mail. Estes métodos são essenciais para procurar dados do sistema.

- `getUtilizadorPorEmail(...)`
- `getArtistaPorNome(...)`
- `getMusicaPorNome(...)`
- `getAlbumPorNome(...)`
- `getPlaylistPorNome(...)`

#### *Adição de Conteúdos*

Permite adicionar novos elementos ao sistema. Estes métodos garantem que conteúdos como músicas, artistas, álbuns e playlists são integrados corretamente nas respetivas listas.

- `adicionarMusica(...)`
- `adicionarArtista(...)`
- `adicionarAlbum(...)`
- `adicionarPlaylist(...)`

### *Reprodução de Músicas*

Reproduzir músicas e registar as reproduções feitas por cada utilizador. A classe também verifica se uma música já foi ouvida, e atribui pontos conforme a lógica definida no sistema.

- `jaOuviuMusica(...)`
- `reproduzirMusica(...)`

### *Listar Dados*

Métodos simples que devolvem representações textuais de conteúdos presentes no sistema, como músicas, playlists ou a biblioteca pessoal de um utilizador. Útil para mostrar informação ao utilizador de forma organizada.

- `listarMusicas(...)`
- `listarPlaylist(...)`
- `listarBiblioteca(...)`

### *Gerar Playlists*

Permite gerar playlists personalizadas com base nos hábitos de audição dos utilizadores. A lógica baseia-se no género musical favorito identificado a partir do histórico de reproduções e pode considerar restrições como tempo máximo ou o conteúdo explícito.

- `obterGeneroFavorito(...)`
- `gerarPlaylistFavorita(...)`
- `gerarPlaylistPorTempo(...)`
- `gerarPlaylistFavoritaExplicita(...)`

### *Outros*

Métodos de apoio usados para conversões e manipulação de listas, como transformar listas de nomes de músicas em objetos **Musica**, ou criar seleções aleatórias.

- `musicasToArray(...)`

- `criaArrayMusicasAleatorio(...)`

## 2.20 Controller

A classe **Controller** atua como o intermediário principal entre a interface de utilizador (views) e a lógica da aplicação SpotifUM. É responsável por receber os comandos provenientes das diferentes views (como **ViewLogin**, **ViewAdmin**, **ViewPlayList**, etc...), interpretar esses comandos e invocar os métodos adequados do model (**SpotifUM**).

Além disso, facilita a manutenção e expansão da aplicação, permitindo a alteração ou adição de funcionalidades sem impactar diretamente a camada de apresentação.

Entre as suas principais funcionalidades estão:

- Gerir o login e o registo de utilizadores;
- Controlar a navegação entre as diferentes views;
- Invocar métodos do model de acordo com os comandos recebidos;
- Coordenar as permissões e comportamentos com base no plano de subscrição e cargo do utilizador.

A classe **Controller** é, assim, essencial para o funcionamento geral da aplicação.

## 2.21 ViewAdmin

A classe **ViewAdmin** implementa a interface de texto dedicada ao administrador da aplicação. Através de um menu interativo, permite aceder a diversas funcionalidades de gestão, nomeadamente a adição de músicas (normais, explícitas ou multimédia), artistas e álbuns, bem como a geração de playlists personalizadas e a listagem de todas as entidades do sistema.

## 2.22 ViewLogin

A classe **ViewLogin** é a interface principal da aplicação responsável por gerir o acesso dos utilizadores. Permite realizar o login, registar novos utilizadores, consultar estatísticas, carregar scripts de exemplo e sair do programa. Após o login, direciona o utilizador para a interface correspondente, seja a de administrador ou utilizador.

Também é responsável por carregar o estado salvo da aplicação no arranque e guardar os dados ao encerrar, garantindo a persistência da informação.

## 2.23 ViewPlaylist

A classe **ViewPlaylist** é a interface responsável por interagir com o utilizador na criação, reprodução e gestão de playlists. Permite criar vários tipos de playlists, incluindo normais, favoritas, por género e playlists explícitas, tanto para utilizadores comuns como para administradores. Também suporta a reprodução das playlists, com modos sequencial ou aleatório, adaptando-se ao plano do utilizador

## 2.24 ViewStats

A classe **ViewStats** é a interface que apresenta ao utilizador um menu para consultar várias estatísticas da aplicação. Permite visualizar dados como a música mais reproduzida, o utilizador que ouviu mais músicas num determinado período ou desde sempre, o artista mais escutado, o género musical com mais reproduções, o utilizador com mais pontos, o número de playlists públicas e o utilizador com mais playlists.

O utilizador pode seleccionar uma destas opções e receber os resultados diretamente no terminal.

## 2.25 ViewUtilizador

A classe **ViewUtilizador** é a interface de texto destinada aos utilizadores da aplicação. Ela oferece um menu interativo para realizar várias operações, como ouvir músicas, listar músicas disponíveis, criar playlists (se o plano permitir), visualizar e ouvir playlists, consultar a biblioteca pessoal, adicionar playlists ou álbuns à biblioteca e trocar o plano de subscrição.

## 2.26 Main

A classe **Main** é o ponto de entrada da aplicação. Ela inicializa o modelo principal **SpotifUM**, cria o **Controller** para gerir a lógica de negócio e um **Scanner** para ler a entrada do utilizador. Em seguida, instância a **ViewLogin**, que apresenta a interface inicial para login, registo e outras opções, e inicia a aplicação utilizando o método `start()`.



## 2.27 Exceptions

Foram implementadas várias exceções personalizadas com o objetivo de garantir um tratamento robusto e específico dos diferentes erros que podem ocorrer durante a execução da aplicação. Estas exceções permitem identificar e reagir adequadamente a situações inesperadas, melhorando a experiência do utilizador e a estabilidade do sistema.

As exceções abrangem diversos cenários, tais como:

- **Elementos inexistentes:** AlbumInexistenteException, ArtistaInexistenteException, MusicalInexistenteException, PlaylistInexistenteException.
- **Entradas duplicadas:** AlbumJaNaBibliotecaException, PlaylistJaNaBibliotecaException, EmailExistenteException.
- **Erros relacionados com playlists:** PlaylistIsNotAleatoriaException, PlaylistVaziaException.
- **Ausências de reproduções:** SemReproducoesException.
- **Gestão de utilizadores:** UtilizadorFaltaPontosException, UtilizadorFreeNaoPossuiBibliotecaException, UtilizadorNaoTemPermissoesException.

## 2.28 Testes

Foram realizados testes unitários a diversas classes fundamentais do sistema, com o objetivo de garantir a fiabilidade e o correto funcionamento dos métodos implementados.

As classes testadas foram:

- **Músicas:** MusicaTest, MusicaExplicitaTest e MusicaMultimediaTest.
- **Planos de Subscrição:** PlanoFreeTest, PlanoPremiumBaseTest e PlanoPremiumTopTest.
- **Playlists:** PlaylistAleatoriaTest, PlaylistConstruidaTest, PlaylistFavoritosTest e PlaylistGeneroTest.
- **Outras Classes:** AlbumTest, ArtistaTest, BibliotecaTest, ReproducaoTest e UtilizadorTest.

Estes testes permitiram validar o comportamento esperado de cada classe, identificar eventuais erros e assegurar que as alterações realizadas ao longo do desenvolvimento não comprometeram funcionalidades existentes.

### 3. Diagrama de Classes

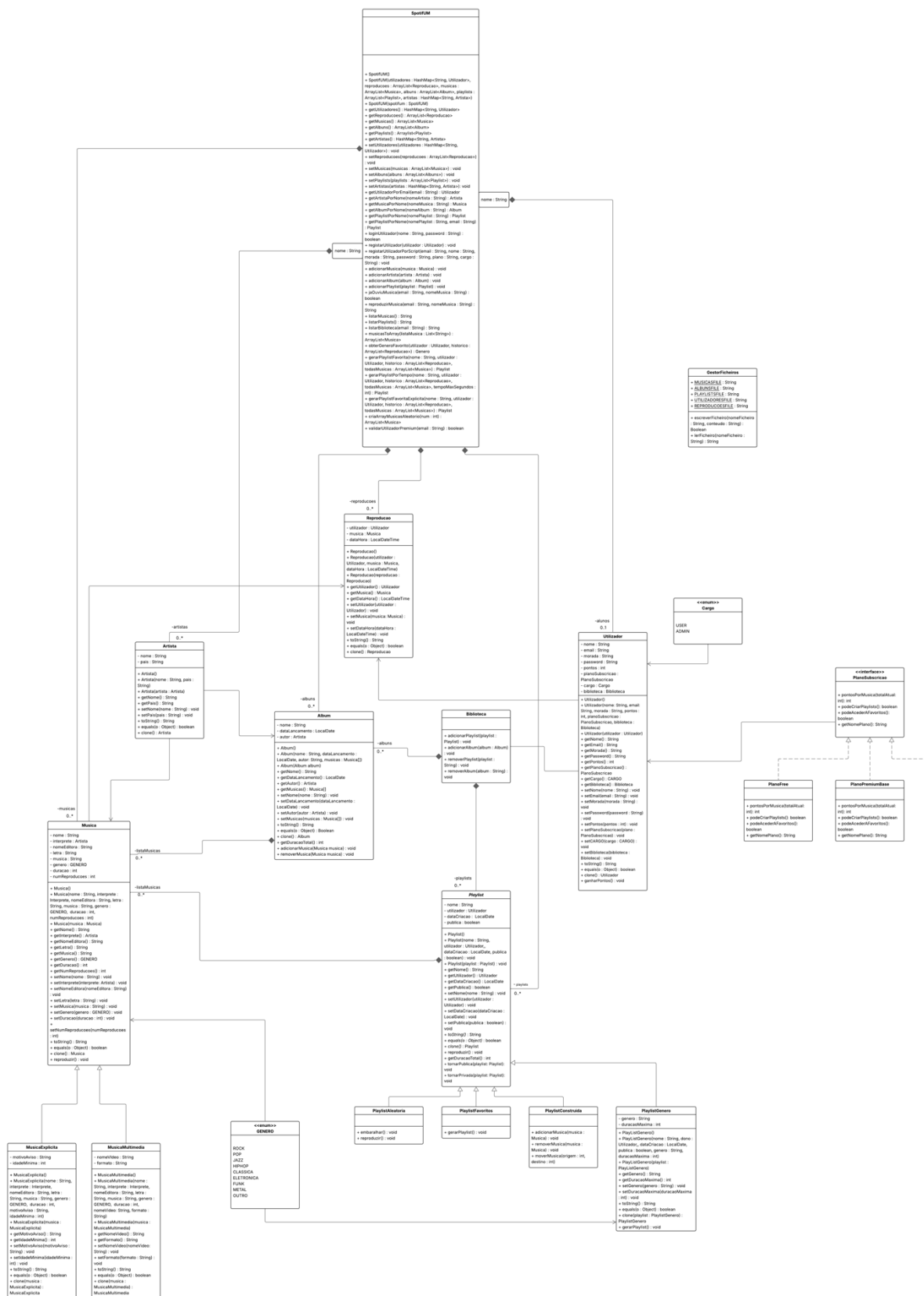


Figura 1- Diagrama de Classes

## 4. Estrutura do Projeto

A aplicação **SpotifUM** foi desenvolvida com base numa estrutura modular e organizada, seguindo o padrão **MVC (Model-View-Controller)**, que permite uma separação clara entre a interface com o utilizador, a lógica da aplicação e os dados que a suportam.

O projeto está dividido da seguinte forma:

- **Model (SpotifUM):** Contém as classes que representam os elementos centrais da aplicação, como músicas, playlists, utilizadores, álbuns e os diferentes tipos de plano (Free, Premium e PremiumTop). Neste pacote encontra-se também a classe SpotifUM, que funciona como núcleo da aplicação, reunindo e gerindo todos os componentes.
- **View:** Reúne as classes responsáveis pela interação com o utilizador através do terminal. Estão incluídas várias interfaces específicas, como ViewLogin, ViewUtilizador, ViewPlaylist e ViewStats, cada uma dedicada a uma parte distinta da aplicação.
- **Controller:** Atua como intermediário entre o modelo e as interfaces de utilizador. Esta camada trata os pedidos do utilizador, valida os dados introduzidos e executa as ações correspondentes sobre o modelo.

A classe Main dá início à aplicação, criando as instâncias necessárias do model, controller, views, e inicia o ciclo de execução da aplicação.

A estrutura foi pensada para ser clara, de fácil manutenção e preparada para futuras expansões. Adicionalmente, a utilização de exceções personalizadas permite uma melhor gestão de erros, tornando a aplicação mais robusta e fácil de utilizar.

## 5. Conclusão

Tendo em conta os objetivos definidos para a criação desta plataforma e os requisitos estabelecidos no enunciado, consideramos que a aplicação desenvolvida cumpre de forma eficaz e intuitiva todas as funcionalidades propostas, proporcionando uma utilização simples e compreensível a todos os utilizadores.

Destacamos a utilização de encapsulamento total e a preocupação em garantir a reutilização e manutenção facilitada do código, através da aplicação de hierarquias de classes e interfaces. Este cuidado permite que futuras alterações ou expansões à aplicação possam ser feitas de forma controlada e eficiente, sem necessidade de grandes modificações na estrutura existente.

Com a realização deste projeto, aprofundámos significativamente os nossos conhecimentos de Programação Orientada a Objetos, colocando em prática os conceitos lecionados nas aulas teóricas e práticas. Ao longo do desenvolvimento, enfrentámos vários desafios que nos permitiram também explorar outras áreas da linguagem Java, nomeadamente a gestão de ficheiros e a implementação de persistência de dados, consolidando assim competências importantes para projetos futuros.