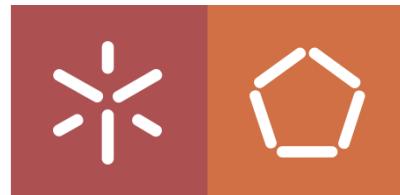


UNIVERSIDADE DO MINHO

Licenciatura em Engenharia Informática



Programação Orientada aos Objetos

2024/2025

Relatório Trabalho Prático

GRUPO 03



a106804, Alice Soares



a106853, Ana Beatriz Freitas



a106877, José Cação

Maio 2025

Índice

1	Introdução	1
2	Arquitetura de Classes	2
2.1	Visão Geral da Arquitetura	2
2.2	Diagrama de Classes UML	3
2.3	Design Patterns Aplicados	3
2.3.1	Strategy Pattern	4
2.3.2	Factory Pattern (Simple Factory)	6
2.3.3	Observer Pattern	7
2.3.4	Model-View-Controller (MVC) Pattern	9
2.3.5	Marker Interface Pattern	10
2.3.6	Facade Pattern:	11
2.4	Outras Decisões Relevantes	12
2.4.1	Manutenção do Encapsulamento nas Entidades do Modelo	12
2.4.2	Escolha de Estruturas de Dados	12
2.4.3	Estratégia de Tratamento e Propagação de Exceções Personalizadas	13
2.4.4	Arquitetura da Camada View com Classes Base Genéricas	14
3	Descrição da Aplicação Desenvolvida	16
3.1	Funcionalidades Implementadas	16
3.1.1	Gestão de Entidades	16
3.1.2	Reprodução de Conteúdo	16
3.1.3	Biblioteca Pessoal e Playlists Públicas	17
3.1.4	Criação e geração de Playlists	17
3.1.5	Sistema de Pontos	18
3.1.6	Estatísticas	18
3.1.7	Persistência de Dados	18
3.2	Interface com o Utilizador	19
3.2.1	Navegação por Menus	19
3.2.2	Tratamento de Input e Feedback	19

4 Conclusão	21
4.1 Trabalhos Futuros e Melhorias	21

Capítulo 1

Introdução

Este relatório aborda o desenvolvimento do trabalho prático da disciplina de Programação Orientada aos Objetos, implementado em linguagem *Java*, inserido no 2ºano da licenciatura em Engenharia Informática.

O projeto SpotifUM consiste no desenvolvimento de uma aplicação de gestão e reprodução de músicas, inspirada em plataformas de *streaming* como o *Spotify*. A aplicação permite a criação e organização de músicas, álbuns, playlists e utilizadores com diferentes tipos de subscrição.

Entre as funcionalidades implementadas destacam-se a reprodução de músicas, a criação de playlists com base nas preferências dos utilizadores, o sistema de pontuação e a recolha de estatísticas de utilização.

A aplicação inclui ainda mecanismos de persistência para guardar e carregar o estado do sistema, garantindo que os dados dos utilizadores e das músicas se mantêm entre sessões.

Capítulo 2

Arquitetura de Classes

2.1 Visão Geral da Arquitetura

A arquitetura do SpotifUM segue o padrão *Model-View-Controller* (MVC), assegurando uma separação clara entre os dados, a lógica de aplicação e a interface com o utilizador.

Na camada de modelo, encontram-se as entidades principais, como *Musica* (e as suas variantes *MusicaExplicita* e *MusicaMultimedia*), *Album*, *Playlist* (com extensões como *PlaylistAleatoria* e *PlaylistPersonalizada*), *Utilizador*, *Reproducao* e *RegistoDeReproducao*. A associação de utilizadores a planos de subscrição é feita através da classe abstrata *PlanoSubscricao*, com implementações concretas como *Free*, *PremiumBase* e *PremiumTop*. A classe *SpotifUMModel* atua como fachada, centralizando os dados em memória e oferecendo operações de gestão, além de gerar estatísticas e playlists.

A camada de controlo, representada por *SpotifUMController*, recebe os comandos da interface, valida entradas e interage com o modelo. Esta camada isola a lógica de aplicação das operações de apresentação, promovendo um desenho limpo e modular.

Por fim, a camada de interface, *View*, consiste num conjunto de ecrãs em modo texto. Estes apresentam opções ao utilizador, recolhem os seus inputs e exibem os resultados de forma simples e funcional.

Esta arquitetura modular e orientada a interfaces permite uma evolução facilitada do sistema. Funcionalidades como novos tipos de playlists ou planos de subscrição podem ser integradas sem impacto significativo nas outras camadas, promovendo a extensibilidade e a manutenção.

2.2 Diagrama de Classes UML

A seguir apresenta-se o diagrama de classes UML que representa a arquitetura final do sistema SpotifUM. Este diagrama oferece uma visão global da estrutura do programa, evidenciando as principais entidades, as relações entre elas e a organização modular adotada.

As relações de herança, composição, agregação e associação são representadas segundo a notação UML padrão, facilitando a compreensão da estrutura e dependências entre componentes. Os atributos e métodos principais estão incluídos com o objetivo de documentar as responsabilidades fundamentais de cada classe, sem comprometer a legibilidade do diagrama.

Este diagrama serviu como base para guiar a implementação e contribuiu para garantir uma arquitetura coesa, extensível e de fácil manutenção.

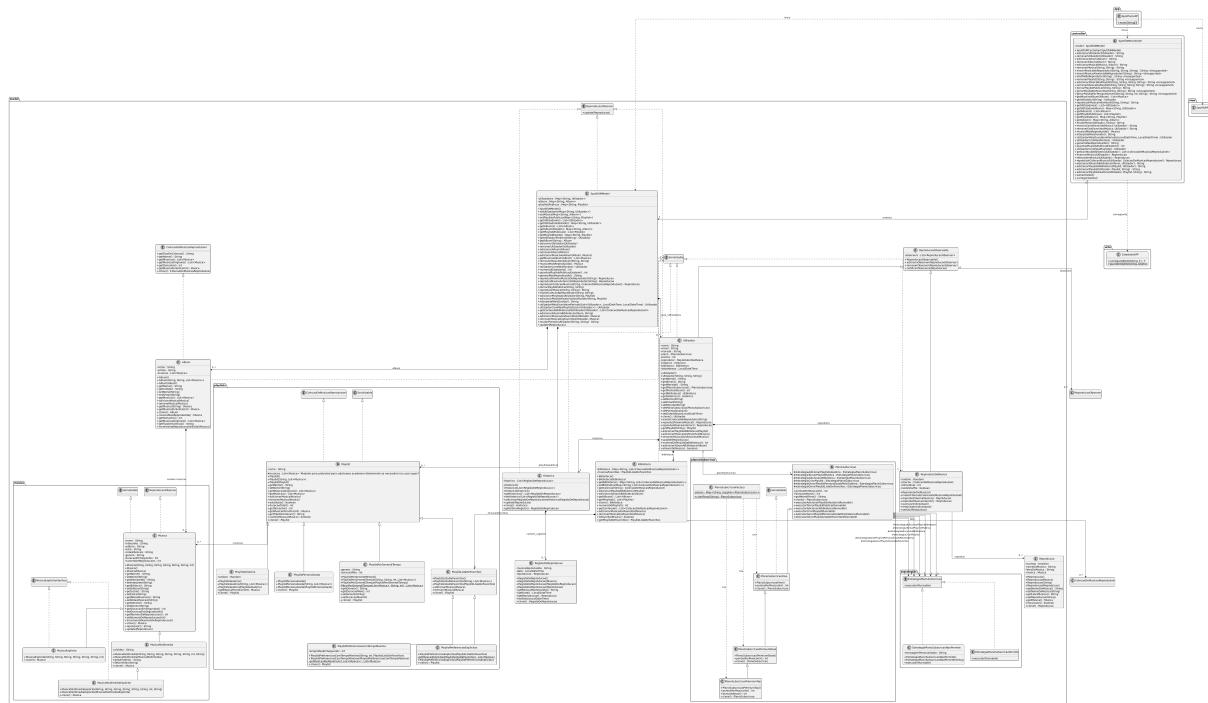


Figura 2.1: Diagrama de Classes UML do sistema SpotifUM - *Model e Controller*.



Figura 2.2: Diagrama de Classes UML do sistema SpotifUM - *View*.

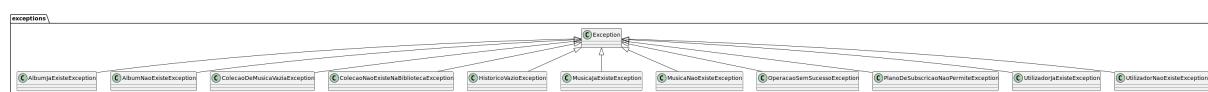


Figura 2.3: Excerto Diagrama de classes UML: *Exceptions*

2.3 Design Patterns Aplicados

No desenvolvimento do sistema SpotifUM, a aplicação consciente de diversos *design patterns* foi fundamental para enfrentar desafios específicos e para alcançar os requisitos das funcionalidades, assim como

a reutilização de componentes, a flexibilidade para futuras extensões e uma maior clareza.

Em seguida, detalhamos os principais padrões identificados e implementados no nosso SpotifUM. Para cada padrão, será apresentado o contexto do problema que motivou a sua utilização, a forma como foi concretizado na arquitetura de classes, os benefícios obtidos com a sua aplicação, e a sua localização no diagrama de classes UML para facilitar a visualização da sua estrutura e interações. A discussão que se segue visa demonstrar como estes padrões contribuíram para uma solução adaptável às necessidades do projeto.

2.3.1 Strategy Pattern

O sistema SpotifUM suporta o conceito de planos de subscrição associados aos utilizadores. Dessa forma, era necessário uma forma flexível e extensível para gerir os diferentes comportamentos e permissões associadas aos vários tipos de PlanoSubscricao (*Free*, *PremiumBase*, *PremiumTop*). Cada plano concede direitos distintos aos utilizadores para realizar certas ações, como adicionar álbuns à biblioteca, criar playlists personalizadas ou tornar playlists públicas. O desafio consistia em implementar estas variações de comportamento de uma maneira que evitasse condicionais complexas espalhadas pelo código e que facilitasse a modificação de permissões existentes ou a adição de novos planos à aplicação de forma simples, no futuro.

Ao deparamo-nos com este problema, o nosso primeiro instinto foi considerar a utilização de atributos *booleanos* dentro de cada classe de plano para controlar as permissões. No entanto, queríamos encontrar uma forma de encapsular a lógica de execução da ação (ou a sua negação) de forma mais coesa e controlada, idealmente permitindo que a própria classe PlanoSubscricao gerisse a execução da ação com base na permissão.

Após ponderação, e com o objetivo de ter um maior controlo sobre a execução das ações condicionadas pelas permissões do plano, optamos por aplicar o *Strategy Pattern*.

A implementação evoluiu da seguinte forma:

1. **Abordagem Inicial:** Numa primeira implementação, considerámos criar uma interface Strategy separada para cada tipo de ação que necessitava de verificação de permissão (ex: EstrategiaAdicionarBiblioteca, EstrategiaTornarPlaylistPublica), cada uma com as suas respetivas classes concretas de PermiteAdicionarBiblioteca / NaoPermiteAdicionarBiblioteca, etc. Contudo, rapidamente nos apercebemos que esta abordagem levaria a uma proliferação de interfaces e classes de estratégia muito similares, resultando em bastante repetição de código e numa potencial quebra de encapsulamento nos parâmetros que passávamos.
2. **Refactoring para uma Estratégia Genérica:** Para otimizar e simplificar, reformulamos a solução. Em vez de múltiplas interfaces de estratégia, criámos uma única interface EstrategiaPlanoSubscricao. Esta interface define um método `executar(Runnable acao)`, que recebe a própria ação a ser executada como um parâmetro do tipo Runnable (uma interface funcional em Java que representa uma ação sem argumentos e sem return).

As classes e interfaces chave na implementação final são:

- **PlanoSubscricao:** Atua como Classe Abstrata. Cada subclasse de plano (PlanoSubscricaoFree, PlanoSubscricaoPremiumBase, PlanoSubscricaoPremiumTop) é construída com instâncias específicas das estratégias para cada tipo de operação controlada.
- **EstrategiaPlanoSubscricao:** É a interface Strategy, com o método `void executar(Runnable acao) throws PlanoDeSubscricaoNaoPermiteException;`.
- **EstrategiaPlanoSubscricaoPermite:** Uma Concrete Strategy que simplesmente executa a Runnable passada como argumento (`acao.run();`).

- **EstrategiaPlanoSubscricaoNaoPermite:** Outra Concrete Strategy que não executa a ação, mas lança uma exceção `PlanoDeSubscricaoNaoPermiteException`, indicando que a operação não é autorizada pelo plano. Esta classe pode ser construída com uma mensagem de erro personalizada.

A aplicação do Strategy Pattern, especialmente após o *refactoring* para uma estratégia mais genérica que aceita uma `Runnable`, trouxe diversos benefícios:

- **Flexibilidade e Extensibilidade:** Tornou-se trivial adicionar novas permissões ou modificar as existentes. Basta adicionar um novo campo de estratégia na classe base `PlanoSubscricao`, inicializá-lo nas subclasses concretas, e adicionar o método correspondente. Novos planos podem ser criados e configurados com suas próprias combinações de permissões sem alterar o código das ações em si.
- **Coesão e Encapsulamento:** A lógica de "permitir" ou "não permitir" uma ação está encapsulada nas classes de estratégia. A ação real a ser executada é passada como um `Runnable`, o que significa que a estratégia não precisa conhecer os detalhes da ação.
- **Redução de Código Duplicado:** O *refactoring* evitou a criação de múltiplas classes de estratégia com lógica quase idêntica, simplificando a estrutura.
- **Adesão ao Princípio Aberto/Fechado:** As classes `PlanoSubscricao` estão abertas para extensão (novos planos podem ser adicionados) mas fechadas para modificação (não é necessário alterar o código existente de um plano para mudar a lógica de permissão se esta for encapsulada numa nova estratégia).
- **Clareza:** A intenção do código torna-se mais clara, pois a configuração das permissões em cada plano é explícita através da instanciação das estratégias.

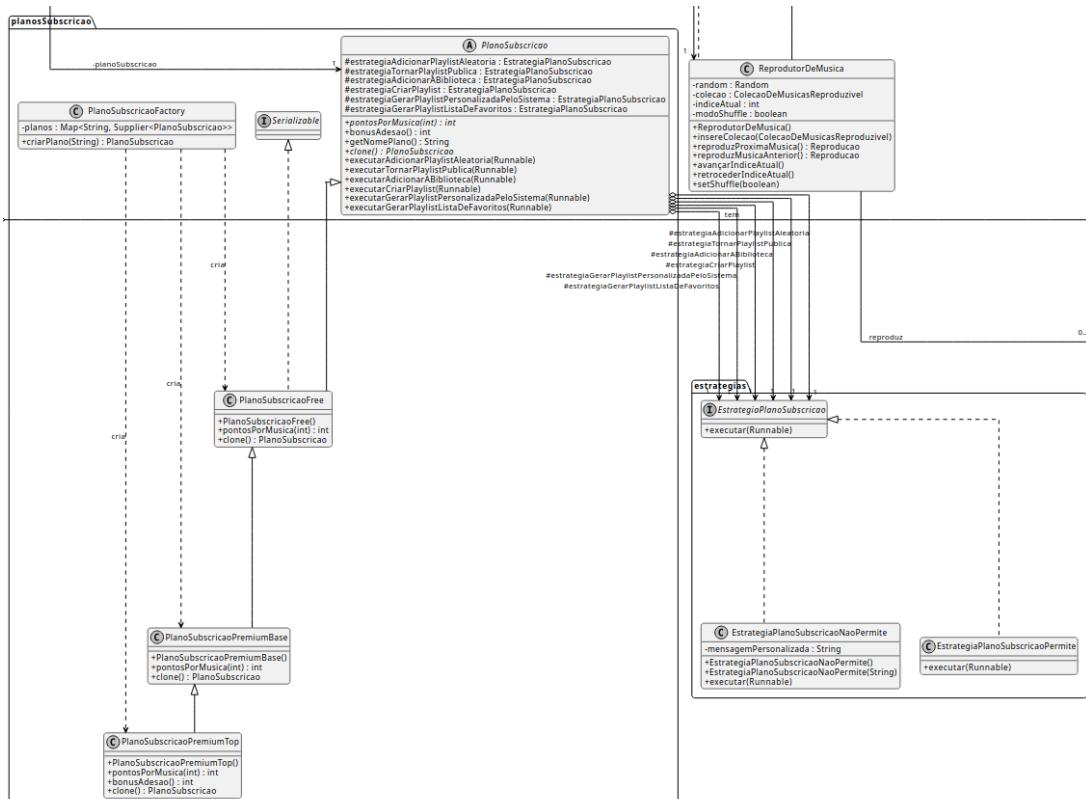


Figura 2.4: Excerto Diagrama de classes UML: *Strategy Pattern*

2.3.2 Factory Pattern (Simple Factory)

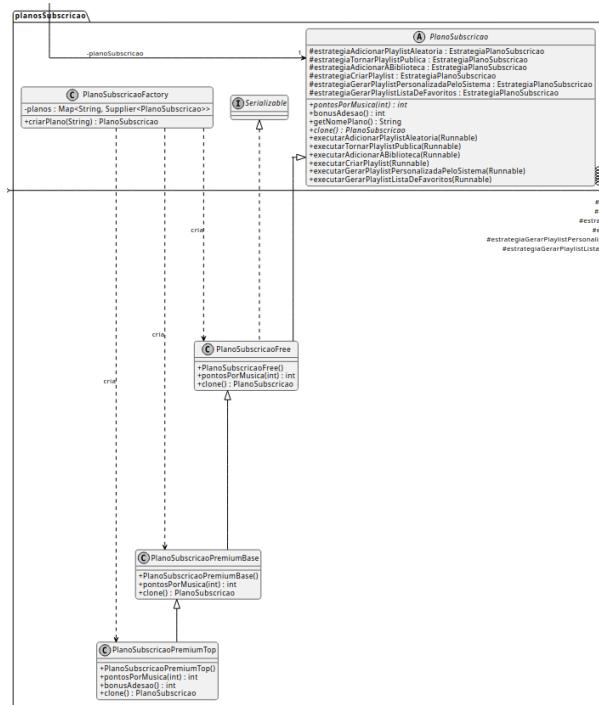
Ao implementar a funcionalidade de trocar o plano de subscrição dos utilizadores, percebemos que a aplicação necessitava de um mecanismo para criar diferentes tipos de planos de subscrição de forma flexível e centralizada. Inicialmente, a lógica de selecionar e instanciar o plano correto ao mudar o plano de um utilizador estava implementada através de estruturas condicionais (neste caso, o `switch-case`) diretamente no controller. Esta abordagem direta, embora funcional para um pequeno número de tipos de planos, torna-se difícil de manter e estender à medida que novos planos são introduzidos. Cada alteração ou adição de um novo plano exigiria a modificação dessas estruturas condicionais, aumentando o risco de erros e violando o Princípio Aberto/Fechado (Open/Closed Principle).

Para resolver este problema, foi introduzida a classe `PlanoSubscricaoFactory`. Esta classe implementa o padrão Simple Factory, fornecendo um método estático `criarPlano(String nome)`. Este método encapsula toda a lógica de decisão sobre qual subclasse de `PlanoSubscricao` deve ser instanciada com base no nome do plano fornecido como argumento. Internamente, a fábrica utiliza um mapa (`Map<String, Supplier<PlanoSubscricao>`) que associa os nomes dos planos aos seus respectivos construtores de instâncias.

Quando o método `criarPlano` é invocado, ele consulta este map para encontrar o fornecedor correspondente ao nome do plano e, se encontrado, utiliza-o para criar e retornar a nova instância. Isto abstrai completamente o controller do processo de seleção e instanciação das classes concretas de `PlanoSubscricao`.

A implementação do Simple Factory através da `PlanoSubscricaoFactory` trouxe diversos benefícios:

- **Centralização da Lógica de Criação:** Toda a lógica para criar os diferentes tipos de planos reside num único local, a fábrica. Isto simplifica a gestão e a compreensão de como os planos são criados.
- **Redução de dependências:** O model já não precisa de conhecer todas as subclasses concretas de `PlanoSubscricao`. Ele apenas interage com a fábrica e a interface/classe base `PlanoSubscricao`.
- **Facilidade de Extensão (Escalabilidade):** Introduzir novos tipos de planos no futuro tornou-se significativamente mais simples. Basta criar a nova classe de plano e registá-la na fábrica (adicionar uma nova entrada ao mapa). Não é necessário modificar o código que utiliza a fábrica, que anteriormente teria um `switch-case` a ser alterado. Esta abordagem adere melhor ao Princípio Aberto/Fechado.
- **Manutenção Simplificada:** Se a lógica de instanciação de um plano precisar ser alterada (por exemplo, se um construtor mudar), a modificação é feita apenas dentro da fábrica, num local controlado, minimizando o impacto noutras partes do sistema.
- **Clareza do Código:** A remoção de blocos `switch-case` ou longas cadeias de `if-else if` do código torna-o mais limpo e focado na sua responsabilidade principal, delegando a criação de objetos à fábrica.


 Figura 2.5: Excerto Diagrama de classes UML: *Factory Pattern*

2.3.3 Observer Pattern

Em várias partes da aplicação, diferentes objetos precisam reagir a um mesmo evento central, nomeadamente, a reprodução de uma música. Isto porque, quando uma música é tocada:

- A própria instância da **Musica** precisa de ter o seu contador de reproduções incrementado.
- O **Historico** do utilizador que está a ouvir a música deve registar esta nova reprodução.
- O **Utilizador** pode necessitar de atualizar a sua pontuação com base no seu plano de subscrição.

O padrão Observer foi implementado para resolver este problema, permitindo que múltiplos objetos (Observers) se registem para serem notificados das reproduções que lhes interessam. Desta forma, quando uma música é reproduzida, em vez de o ReprodutorDeMusica ter lógica de atualização espalhada, cada classe responsável por essas atualizações observa o evento de reprodução e reage de forma autónoma e encapsulada. Os componentes envolvidos são:

- **Observable:** A classe **RetrovisorObservable** atua como a classe base para os observáveis. A instância que origina o evento de reprodução e notifica os observers é o **RetrovisorDeMusica**, que herda de **RetrovisorObservable**. Quando o **RetrovisorDeMusica** finaliza a reprodução de uma música, ele cria um objeto **Retrovisor** e invoca o seu método **notificarObservers(Retrovisor reproducao)**.
- **Interface Observer:** A interface **RetrovisorObserver** é implementada em todos os objetos que desejam ser notificados. Ela declara um único método, **update(Retrovisor reproducao)**, que é chamado pelo Observable quando o evento ocorre.
- **Concrete Observers:** São as classes que implementam a interface **RetrovisorObserver** e contêm a lógica específica para reagir ao evento. Na aplicação, temos:

- **Historico:** No seu método `update(Reproducao reproducao)`, cria um novo `RegistroDeReproducao` a partir do objeto `Reproducao` e adiciona-o à sua lista interna de registos.
- **Utilizador:** No seu método `update(Reproducao reproducao)`, calcula e adiciona os pontos ganhos pela reprodução da música, de acordo com o `PlanoSubscricao` do utilizador.
- **SpotifumModel:** No seu método `update(Reproducao reproducao)`, percorre as músicas e incrementa o número de reproduções ás que foram reproduzidas.

- **Mecanismo de Subscrição e Notificação:**

- **Subscrição:** O `ReprodutorDeMusica` (na classe `Utilizador`, ao ser inicializado) regista os seus observers interessados (o `Historico` do utilizador e o próprio `Utilizador`). Já o `SpotifumModel` é adicionado como observer ao utilizador aquando a adição do mesmo à aplicação.
- **Notificação:** Quando uma música é efetivamente reproduzida pelo `ReprodutorDeMusica`, este chama `notificarObservers(reproducao)`, que itera sobre a lista de observers registados e invoca o método `update(reproducao)` em cada um deles, passando o objeto `Reproducao` como argumento.

A utilização do padrão Observer neste contexto é crucial para:

- **Diminuição de dependências:** O `ReprodutorDeMusica` não precisa de conhecer as classes dos seus Observers. Ele apenas interage com eles através da interface `ReproducaoObserver`. Isto significa que os Observers podem variar independentemente do Reprodutor.
- **Coesão e Responsabilidade Única:** Cada Concrete Observer encapsula a sua própria lógica de reação ao evento de reprodução. Isto evita que outra classe acumule múltiplas responsabilidades que não lhe pertencem diretamente.
- **Flexibilidade e Extensibilidade:** É fácil adicionar novos comportamentos em resposta à reprodução de uma música. Basta criar uma nova classe que implemente `ReproducaoObserver` e registá-la no `ReprodutorDeMusica`. Não são necessárias grandes modificações no código do Reprodutor nem nos Observers existentes.

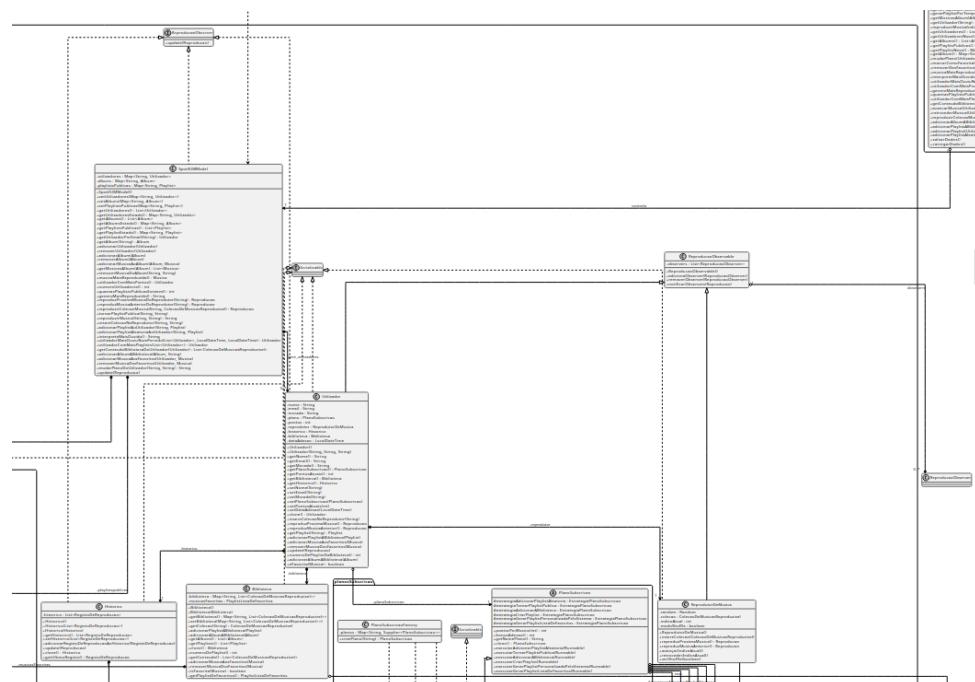


Figura 2.6: Excerto Diagrama de classes UML: *Observer Pattern*

2.3.4 Model-View-Controller (MVC) Pattern

Como a aplicação é interativa era fundamental utilizar uma estrutura organizada que separa as diferentes preocupações do sistema. A ausência de uma arquitetura clara podia levar a um código difícil de entender, manter e evoluir, onde a lógica, a interface do utilizador e o controlo de fluxo estão fortemente emaranhados.

A aplicação SpotifUM adota o padrão arquitetural Model-View-Controller (MVC) para estruturar os seus componentes principais. Esta divisão é claramente refletida na organização dos pacotes:

- **Model:** Localizado no pacote `model`, encapsula os dados da aplicação e a lógica de negócio. Inclui classes centrais como `SpotifUMModel` (que gera as coleções de utilizadores, álbuns, etc.), e as entidades de domínio como `Utilizador`, `Musica`, `Album`, `Playlist`, e `PlanoSubscricao`. O Model é responsável por manipular os dados e aplicar a lógica necessária, independentemente de como essa informação é apresentada ou como o utilizador interage com ela.
- **View:** O pacote `view` contém as classes responsáveis pela apresentação dos dados ao utilizador e pela captura das suas interações. Isto inclui `SpotifUMView` (que gere a transição entre ecrãs), a hierarquia de ecrãs baseada em `BaseScreen` (como `MenuBaseScreen`, `ListarEntidadesBaseScreen`), e as implementações de ecrãs para diferentes funcionalidades (e.g., `MenuPrincipalScreen`, `CriarUtilizadorScreen`).
- **Controller:** Representado pela classe `SpotifUMController` no pacote `controller`. Atua como o intermediário entre o Model e a View. Ele recebe informação da View (ações do utilizador), interpreta-a e invoca as operações apropriadas no Model para processar os dados ou alterar o estado da aplicação. Após o Model processar a solicitação, o Controller pode instruir a View a atualizar a sua apresentação para refletir as mudanças no Model.

Na nossa View, em algumas partes da sua implementação (por exemplo, em classes como `MenuEntidadeBaseScreen<T>` ou `CriarEntidadeBaseScreen<T, O>`), manipula diretamente instâncias de entidades do Model (o tipo genérico T ou O frequentemente refere-se a classes como `Utilizador`, `Album`, `Musica`). Por exemplo, `MenuAlbum` recebe um objeto `Album`, e `CriarMusicaScreen` recebe um `Album` como alvo e cria uma `Musica`.

Esta abordagem, onde a View tem conhecimento direto de algumas entidades do Model, é uma variação comum em certas implementações de MVC, especialmente em cenários onde a View precisa de apresentar dados específicos dessas entidades ou preencher os seus campos.

- **Pragmatismo vs. Pureza do Padrão:** Numa aplicação MVC "pura", a View idealmente apenas conheceria representações de dados mais simples (como DTOs - Data Transfer Objects) ou seria atualizada inteiramente pelo Controller. No entanto, passar as próprias entidades do Model (ou clones delas, como implementamos) para a View pode ser mais direto e pragmático para aplicações de consola ou interfaces onde a complexidade da View não justifica uma camada adicional de DTOs.
- **Leitura vs. Escrita:** O aspeto crucial é que a View, mesmo tendo acesso a estas entidades, não deve modificar diretamente o estado do Model. Ela utiliza estas entidades primariamente para *leitura* (para apresentar informações) ou para *recolher dados* (como nos ecrãs de criação). A modificação do estado do Model (salvar uma nova entidade, apagar uma existente) é sempre delegada ao Controller.
- **Encapsulamento via Clones:** A utilização de métodos `clone()` ao passar estas entidades para a View ou ao obtê-las do Controller (que por sua vez as obtém do Model) ajuda a mitigar os riscos, garantindo que a View trabalha com cópias, protegendo o estado original no Model.

Portanto, embora a View tenha conhecimento de entidades do Model, isto não é crítico, pois mantemos a responsabilidade de modificar o estado do Model no Controller e que o encapsulamento seja mantido.

A adoção do MVC é fundamental para:

- **Separação de Responsabilidades:** Mantém a distinção clara entre a lógica de negócio (Model), a apresentação (View) e o controlo de fluxo (Controller).
- **Manutenção Melhorada:** Alterações na forma como os dados são apresentados na View têm menos probabilidade de afetar a lógica do model, e vice-versa, desde que a View não altere diretamente o estado do Model.
- **Desenvolvimento Paralelo:** Permite que diferentes aspectos da aplicação sejam desenvolvidos com alguma independência.

2.3.5 Marker Interface Pattern

Existe a necessidade de categorizar certos tipos de músicas como sendo do tipo Explícita para que a aplicação possa, potencialmente, tratá-las de forma diferente. No entanto, esta categorização não adiciona necessariamente novos métodos ou comportamentos que precisem ser implementados por estas músicas, mas sim apenas uma marcação de tipo.

A interface `MusicaExplicitaInterface` é utilizada como uma "marker interface". Esta interface não declara nenhum método, apenas marca uma classe como sendo do mesmo tipo. As classes `MusicaExplicita` e `MusicaMultimediaExplicita` implementam esta interface. Podemos assim verificar se um objeto `Musica` é do tipo explícito e aplicar lógicas baseadas nessa verificação.

Justificação e Benefícios:

- **Marcação de Tipo em Tempo de Execução:** Permite que objetos sejam "marcados" como possuindo uma certa propriedade ou característica sem a necessidade de adicionar novos métodos à sua interface pública.
- **Clareza Semântica:** Torna explícito no tipo da classe que ela possui uma característica particular (neste caso, ser "explícita").
- **Polimorfismo Condicional:** Facilita a aplicação de lógica específica a objetos que implementam a marker interface, usando `instanceof`. Por exemplo, um filtro de reprodução poderia verificar `if (musica instanceof MusicaExplicitaInterface) { // aplicar restrição }`.

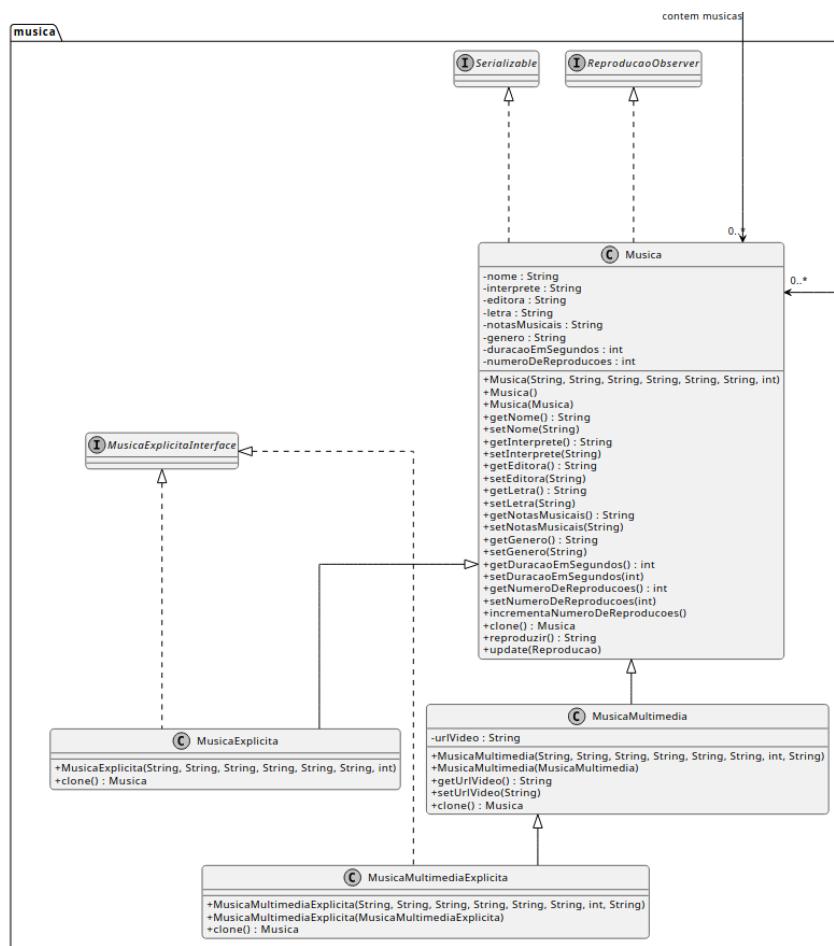


Figura 2.7: Excerto Diagrama de classes UML:Marker Interface Pattern

2.3.6 Facade Pattern:

O subsistema do modelo (model) pode ser composto por várias classes interligadas (Utilizador, Album, Playlist, ReprodutorDeMusica, etc.), cada uma com suas próprias responsabilidades e interfaces. Para a camada do controller interagir diretamente com todas estas classes do modelo pode ser complexo e criar muitas dependências.

A própria classe **SpotifUMModel** pode ser vista como uma Facade para as coleções de dados que ela gera (utilizadores, álbuns, playlists públicas). Em vez de o Controller aceder diretamente a estas coleções (que são mapas internos), eles utilizam os métodos públicos de **SpotifUMModel**, que encapsulam a lógica de acesso e manipulação dessas coleções.

- **Simplificação da Interface:** O **SpotifUMModel** fornece uma interface mais simples e coesa para as operações comuns que o Controller precisa realizar, escondendo a complexidade das interações internas do Model.
- **Redução de dependências:** O Controller depende do **SpotifUMModel** e não diretamente de todas as classes do Model.
- **Organização em Camadas:** Promove uma melhor organização em camadas, onde o Model serve como um ponto de entrada bem definido para a lógica e dados encapsulados.

2.4 Outras Decisões Relevantes

Para além dos padrões de projeto formalmente identificados, diversas outras escolhas de implementação foram cruciais para moldar a arquitetura e o comportamento da aplicação SpotifUM.

2.4.1 Manutenção do Encapsulamento nas Entidades do Modelo

O encapsulamento para as entidades do modelo (como `Utilizador`, `Album`, `Musica`, `Playlist`) manifesta-se principalmente através de:

- Atributos privados: A maioria dos campos das classes do modelo são declarados como `private`, restringindo o acesso direto de fora da classe.
- Getters e Setters controlados: O acesso e a modificação dos atributos são feitos através de métodos públicos (getters e setters), permitindo que a classe controle como os seus dados são acedidos e alterados, e possibilitando a inclusão de lógica de validação ou efeitos secundários controlados.
- Uso extensivo de clonagem: Conforme detalhado anteriormente, sempre que objetos do modelo ou coleções de objetos do modelo são retornados por métodos (especialmente em getters do `SpotifUMModel` ou de entidades agregadoras como `Album` e `Playlist`), são retornadas cópias (clones) em vez das referências originais. Isto é vital para evitar que o estado interno do modelo seja modificado accidentalmente por código externo que obtenha uma referência a um objeto.

Com isto obtemos:

- **Integridade dos Dados:** Protege o estado interno dos objetos contra modificações não autorizadas ou inconsistentes, garantindo que as entidades do modelo permaneçam num estado válido.
- **Baixa dependência:** Reduz as dependências diretas de outras partes do sistema nos detalhes internos de implementação das entidades do modelo. As alterações na representação interna de uma entidade (desde que a sua interface pública seja mantida) não deverão impactar o código cliente.
- **Flexibilidade e fácil manutenção:** Facilita a reestruturação e a evolução das classes do modelo, pois as alterações internas podem ser feitas com menor risco de afetar outras partes do sistema.
- **Prevenção de Efeitos Secundários:** Ao retornar clones, evita-se que modificações feitas numa instância "obtida" afetem a instância original no modelo.

2.4.2 Escolha de Estruturas de Dados

A aplicação utiliza diferentes estruturas de dados para gerir coleções de entidades, sendo as mais proeminentes:

- `Map<String, Entidade>`: Utilizado no `SpotifUMModel` para armazenar as coleções principais de `Utilizador` (chave: email), `Album` (chave: nome do álbum) e `Playlist` públicas (chave: nome da playlist).
- `List<Entidade>`: Utilizado dentro de classes como `Album` e `Playlist` para armazenar a lista de `Musica`, em `Historico` para a lista de `RegistoDeReproducao`, e em `Biblioteca` para listas de `ColecaoDeMusicasReproduzivel`.
- **Uso de Map para Acesso Rápido por Chave Única:**

- Para entidades como **Utilizador** (identificado unicamente pelo email) e **Album** ou **Playlist** pública (identificados unicamente pelo nome), o uso de **HashMap** oferece um desempenho eficiente (tempo médio constante, $O(1)$) para operações de busca, inserção e remoção baseadas na chave.
 - Garante a unicidade das entidades baseada na sua chave natural (email do utilizador, nome do álbum/playlist).
- **Uso de List para Coleções Ordenadas:**

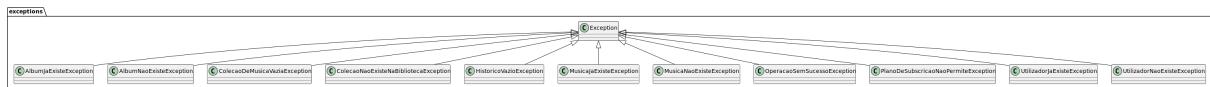
- Dentro de um **Album** ou **Playlist**, a ordem das músicas pode ser importante (especialmente para playlists personalizadas). **ArrayList** mantém a ordem de inserção e permite acesso por índice.
- Para o **Historico**, uma **List** (como **ArrayList**) é adequada para manter a sequência cronológica dos registo de reprodução.
- Embora uma música possa não ser duplicada dentro de um mesmo álbum, uma **List** é uma escolha geral flexível para conter as músicas. Se a unicidade fosse um requisito estrito e a ordem não importasse, um **Set** poderia ser considerado, mas a **List** oferece acesso por índice, o que é útil para a reprodução sequencial.

A escolha de **HashMap** para as coleções principais no **SpotifyModel** privilegia a performance de busca, que é uma operação frequente. A escolha de **ArrayList** dentro das entidades para as suas sub-coleções oferece um bom equilíbrio entre flexibilidade e performance para as operações típicas realizadas nessas coleções (iteração, adição no final, acesso por índice).

2.4.3 Estratégia de Tratamento e Propagação de Exceções Personalizadas

A aplicação define e utiliza um conjunto de exceções personalizadas que herdam de **Exception**. Estas exceções são lançadas pelas classes do Model quando uma condição de erro específica do domínio da aplicação ocorre (e.g., tentar adicionar um utilizador com um email que já existe). O Controller, ao chamar métodos do Model, captura estas exceções e geralmente converte as suas mensagens em strings que são retornadas para a View para serem apresentadas ao utilizador.

- **Semântica Clara de Erros:** Exceções personalizadas tornam o código mais legível e expressivo. Fica claro qual tipo de erro ocorreu (e.g., **AlbumNaoExisteException** é mais informativo do que uma **NullPointerException** genérica).
- **Tratamento Específico de Erros:** Permite que o código cliente (neste caso, o Controller ou até mesmo outros serviços) capture e trate tipos específicos de erros de forma diferenciada, se necessário, embora na implementação atual o Controller tenda a tratar a maioria delas de forma similar (retornando a mensagem).
- **Obrigatoriedade de Tratamento (Checked Exceptions):** Ao herdar de **Exception**, estas exceções são verificadas, o que força os métodos a lidar com elas (seja com um bloco **try-catch** ou declarando-as com **throws**). Isto pode ajudar a evitar que erros importantes sejam ignorados.
- **Desacoplamento do Tratamento de Erros da Lógica Principal:** A lógica no Model pode focar-se na sua tarefa principal e lançar uma exceção quando algo dá errado, delegando o tratamento do erro para as camadas superiores.
- **Comunicação de Erros à View:** O Controller usa as mensagens das exceções para informar a View (e, consequentemente, o utilizador) sobre o que correu mal, de uma forma relativamente compreensível.

Figura 2.8: Excerto Diagrama de classes UML: *Exceptions*

2.4.4 Arquitetura da Camada View com Classes Base Genéricas

A camada de visualização (View) da aplicação SpotifUM, implementada em modo consola, foi arquitetada utilizando um conjunto de classes base abstratas e genéricas localizadas no pacote `view.bases`. Esta abordagem visa promover a reutilização e a consistência na forma como os diferentes ecrãs da aplicação são construídos e se comportam. As principais classes base incluem:

- **BaseScreen:** É a classe fundamental da qual todos os outros ecrãs herdam. Fornece funcionalidades comuns como um título, gestão do ecrã anterior (`previous`), acesso ao `SpotifUMController` e à `SpotifUMView` (para navegação), um `Scanner` partilhado para input, e métodos como `render()` (que chama o método abstrato `run()`), `esperarEnter()`, e `setNextScreen(BaseScreen nextScreen)` para controlar o fluxo de navegação entre ecrãs.
- **MenuBaseScreen:** Herda de `BaseScreen` e é especializada para ecrãs que apresentam um menu de opções ao utilizador. Define um método abstrato `configurarMenu()` que as subclasses devem implementar para fornecer a lista de `MenuItem` (outra classe base que representa uma opção de menu com uma ação associada). A lógica de apresentar o menu e processar a escolha do utilizador está centralizada nesta classe base.
- **MenuEntidadeBaseScreen<T>:** Herda de `MenuBaseScreen` e é parametrizada com um tipo genérico `T`, representando uma entidade do modelo (e.g., `Utilizador`, `Album`). Esta classe é usada para ecrãs de menu que operam no contexto de uma entidade específica. Adiciona a capacidade de mostrar um resumo da entidade (`resumoEntidade(T entidade)`) antes de apresentar o menu. Exemplos de uso incluem `MenuUtilizador<Utilizador>` ou `MenuAlbum<Album>`.
- **ListarEntidadesBaseScreen<T>:** Herda de `BaseScreen` e também é genérica (`T`). É projetada para ecrãs que listam entidades de um determinado tipo. As subclasses devem implementar `getEntidades()` (para obter a lista de entidades através do Controller), `resumoEntidade(T entidade)` (para formatar como cada entidade é mostrada na lista), e `aoSelecionarEntidade(T entidade)` (para definir o que acontece quando o utilizador seleciona uma entidade da lista).
- **SelecionarEntidadeDeListagemScreen<T>:** Herda de `ListarEntidadesBaseScreen<T>` e refina-a para cenários onde o objetivo é selecionar uma entidade de uma lista para executar uma ação específica sobre ela (passada como um `Consumer<T> acao`) e depois navegar para um próximo ecrã. Utilizada, por exemplo, em `SelecionarAlbum` e `SelecionarPlaylist`.
- **CriarEntidadeBaseScreen<T, O>:** Herda de `BaseScreen` e é parametrizada com dois tipos genéricos: `T` para o tipo da entidade a ser criada e `O` para o tipo do "alvo" ou contexto no qual a entidade será criada (e.g., criar uma `Musica` (`T`) para um `Album` (`O`)). As subclasses definem como criar uma nova entidade (`novaEntidade()`), quais campos devem ser preenchidos (`obterCampos()`, usando a classe auxiliar `CampoEntidade<T>`), e como processar a entidade após o preenchimento (`processarEntidade(T entidade, O alvo)`).
- **CampoEntidade<T>:** Classe auxiliar utilizada por `CriarEntidadeBaseScreen` para definir cada campo a ser solicitado ao utilizador, incluindo a pergunta, uma função para converter o input do utilizador, e um `BiConsumer` para definir o valor na entidade.
- **MenuItem:** Representa uma opção num menu, contendo a descrição da opção, a ação a ser executada (`Runnable`), e opcionalmente uma condição de visibilidade (`Supplier<Boolean>`).

As classes concretas de ecrãs (e.g., `MenuPrincipalScreen`, `CriarUtilizadorScreen`, `ListagemAlbumsScreen`) herdam destas classes base e implementam os métodos abstratos, fornecendo o comportamento específico para cada ecrã. A classe `SpotifUMView` gera a apresentação e transição entre estes ecrãs.

- **Reutilização de Código:** A lógica comum a muitos ecrãs (como navegação, apresentação de menus, listagem de itens, recolha de dados para criação de entidades) é implementada uma única vez nas classes base. Isto reduz significativamente a duplicação de código.
- **Consistência da Interface do Utilizador:** Ao reutilizar as classes base, garante-se uma experiência de utilizador mais consistente em termos de como os menus são apresentados, como as listas são navegadas e como os dados são solicitados.
- **Desenvolvimento Simplificado de Novos Ecrãs:** Para criar um novo ecrã de menu ou de listagem, o programador pode herdar da classe base apropriada e focar-se apenas na lógica específica desse novo ecrã (definir os itens de menu, como buscar e apresentar as entidades, etc.), aproveitando a funcionalidade já implementada.
- **Manutenção fácil:** Se for necessário alterar uma funcionalidade comum a todos os menus (e.g., a forma como as opções são numeradas ou como o input inválido é tratado), a alteração pode ser feita na `MenuBaseScreen`, e todos os ecrãs de menu herdarão essa mudança.
- **Uso de Genéricos para Flexibilidade de Tipo:** A utilização de tipos genéricos (`<T>`, `<O>`) nas classes base como `MenuEntidadeBaseScreen`, `ListarEntidadesBaseScreen` e `CriarEntidadeBaseScreen` permite que estas classes sejam reutilizadas para diferentes tipos de entidades do modelo sem a necessidade de reescrever a lógica base para cada tipo. Isto aumenta a segurança de tipo e a clareza do código.
- **Programação Funcional para Ações e Conversores:** O uso de interfaces funcionais como `Runnable` (para ações de `MenuItem`), `Consumer` (para ações ao selecionar entidade), `Function` (para conversores em `CampoEntidade`), e `BiConsumer` (para setters em `CampoEntidade`) permite uma definição concisa e flexível do comportamento dinâmico.

Esta arquitetura da View, baseada numa hierarquia de classes abstratas e genéricas, exemplifica o princípio DRY (Don't Repeat Yourself) e promove um design mais robusto e extensível para a interface de utilizador em consola. Embora a View interaja com entidades do Model (como discutido na secção MVC), esta estrutura ajuda a organizar essas interações de forma padrão.

Capítulo 3

Descrição da Aplicação Desenvolvida

3.1 Funcionalidades Implementadas

3.1.1 Gestão de Entidades

O SpotifUM implementa a gestão de entidades através de operações CRUD (Criar, Ler, Atualizar, Apagar) para os principais componentes do sistema:

- **Utilizadores:** Os utilizadores são criados com dados pessoais e credenciais de acesso. É possível consultar os seus dados, atualizar informações como plano de subscrição, e remover contas. Cada utilizador é associado a um plano de subscrição (free, premium base ou premium top), que influencia funcionalidades disponíveis.
- **Músicas:** As músicas são geridas com base em atributos como nome, artista, género e duração. Podem ser adicionadas à biblioteca, consultadas individualmente ou em listas, atualizadas e removidas da base de dados.
- **Álbuns:** Um álbum agrupa várias músicas e é identificado pelo artista. Suporta operações de criação, leitura e consulta, atualização (como adicionar novas músicas), e eliminação.
- **Playlists:** Existem dois tipos principais: personalizadas e aleatórias. Ambas podem ser criadas pelo utilizador, populadas com músicas da biblioteca, modificadas, consultadas, ou eliminadas. A personalização permite ordenação manual, enquanto as aleatórias reordenam-se automaticamente. Existem ainda playlists baseadas nas preferências dos utilizadores, isto é, na Lista de Favoritos.

Em todos os casos, clones das instâncias garantem que alterações posteriores não afetem versões originais nem dados já guardados.

3.1.2 Reprodução de Conteúdo

A reprodução de conteúdo em SpotifUM é orquestrada pelo método *reproduzMusica(...)* no modelo, que recebe o nome da música e o respetivo álbum, obtém o objeto Musica, incrementa o contador de reproduções e devolve a letra, simulando, desta forma, a reprodução. Quando o utilizador escolhe reproduzir um álbum, a aplicação itera sobre todas as músicas associadas a esse Album, chamando internamente o mesmo método para cada faixa, numa ordem sequencial baseada na lista interna do álbum. Já ao reproduzir uma playlist, a lógica depende do tipo de playlist e do plano do utilizador:

- **Utilizador Free:** só pode aceder a playlists aleatórias (instâncias de PlaylistAleatoria). Em cada chamada a *reproduzProximaMusica()*, o índice é gerado aleatoriamente, sem possibilidade

de avançar para a próxima ou retroceder manualmente. A ordem de execução é inteiramente determinada pelo sistema, e o utilizador não pode interromper ou selecionar uma faixa específica numa playlist aleatória.

- **Utilizador PremiumBase:** pode criar ou guardar playlists personalizadas. Para playlists personalizadas, a reprodução segue a ordem definida por quem criou a playlist, e o utilizador pode invocar *reproduzProximaMusica()* ou *reproduzMusicaAnterior()* para avançar ou retroceder manualmente na lista. Se ativar o modo aleatório, *ativarModoAleatorio()*, as músicas são reproduzidas aleatoriamente, mas o utilizador mantém o controlo de poder avançar.
- **Utilizador PremiumTop:** inclui todas as funcionalidades do PremiumBase e, adicionalmente, pode aceder às playlists geradas automaticamente com base nas suas preferências de reprodução, bem como às versões filtradas, tempo máximo ou apenas músicas explícitas. A reprodução destas playlists de favoritos obedece ao mesmo comportamento de navegação que as playlists personalizadas, permitindo ao utilizador PremiumTop percorrer livremente as músicas recomendadas.

Em qualquer plano, a cada invocação de reprodução, seja de música individual, álbum ou playlist, o contador de reproduções do objeto Musica aumenta, alimentando as estatísticas do sistema como música mais reproduzida, intérprete mais ouvido, etc. Esta distinção clara entre funcionalidades para Free e Premium assegura que cada utilizador recebe uma experiência adequada ao seu nível de subscrição.

3.1.3 Biblioteca Pessoal e Playlists Públicas

Cada utilizador Premium possui uma biblioteca pessoal onde pode guardar álbuns e playlists criadas ou tornadas públicas. No modelo, a classe Biblioteca mantém internamente duas coleções: uma para álbuns (mapeada pela chave “album”) e outra para playlists (chave “playlist”), bem como a playlist de favoritos.

Quando um utilizador PremiumBase ou PremiumTop adiciona um álbum ou uma playlist à sua biblioteca, invoca-se o método *adicionarAlbumABiblioteca(Album)* ou *adicionarPlaylistABiblioteca(Playlist)*, que faz um clone da entidade e armazena-o na lista correspondente dentro do objeto Utilizador. Desta forma, cada utilizador conserva o seu próprio conjunto isolado de coleções.

A funcionalidade de tornar uma playlist pública está disponível apenas para utilizadores Premium. Quando um utilizador opta por partilhar a sua playlist com todos, o método *tornarPlaylistPublica(...)* do modelo move uma cópia da playlist do repositório privado do utilizador para um repositório global de playlists públicas. Esta operação não remove a playlist da biblioteca pessoal, mas adiciona uma réplica à lista central supervisionada por SpotifUMModel.

Outros utilizadores, sejam Free ou Premium, podem então aceder às playlists públicas consultando o método *getPlaylistsPublicas()*, que devolve clones de todas as playlists publicadas. Assim, um utilizador pode incorporar qualquer playlist pública na sua própria biblioteca com *adicionarPlaylistABiblioteca(...)*, criando um clone local que passa a fazer parte da sua coleção privada. Esse mecanismo assegura que as playlists publicadas fiquem disponíveis de forma persistente para toda a comunidade, enquanto cada utilizador mantém a sua versão individual para edições futuras.

3.1.4 Criação e geração de Playlists

O SpotifUM suporta vários tipos de playlists, cada uma gerada segundo regras específicas, mas todas baseadas em listas fornecidas pelo utilizador ou na coleção de músicas favoritas do próprio. A *PlaylistAleatória* seleciona músicas aleatoriamente a cada reprodução, acessível mesmo a utilizadores Free, sem possibilidade de avanço ou retrocesso manual. A *PlaylistPersonalizada*, criada pelo utilizador Premium, mantém uma ordem definida pelo criador e permite avançar ou retroceder manualmente.

A *PlaylistPorGeneroETempo* parte de um conjunto genérico de músicas e inclui somente músicas cujo género corresponde ao desejado e cuja duração acumulada permanece abaixo de um valor limite estipulado pelo utilizador.

Por outro lado, a *PlaylistListaDeFavoritos* agrupa todas as músicas que o utilizador marcou como favoritas, reproduzindo-as na ordem em que foram adicionadas. A partir desses favoritos, a *PlaylistPreferenciasComTempoMaximo* filtra músicas até que a soma das durações não ultrapasse um limite em segundos.

De forma semelhante, a *PlaylistPreferenciasExplicitas* seleciona apenas músicas que implementam a interface *MusicaExplicitaInterface*, formando uma lista composta unicamente por músicas marcadas como explícitas.

3.1.5 Sistema de Pontos

O sistema de pontos do SpotifUM atribui valores diferentes consoante o plano de subscrição do utilizador. No plano Free, cada reprodução de música gera 5 pontos. Utilizadores em PremiumBase recebem 10 pontos por cada música reproduzida. Já no plano PremiumTop, o utilizador ganha 100 pontos imediatos na adesão e, a cada nova reprodução, recebe 2,5 % do total de pontos acumulados até então. Esta lógica é implementada no método *pontosPorMusica(...)* de cada classe concreta de PlanoSubscricao, garantindo que a atribuição de pontos siga as regras de cada plano sem alterar o resto do sistema.

3.1.6 Estatísticas

O SpotifUM permite a consulta de várias estatísticas a partir dos registo de reprodução armazenados no *RegistoDeReproducao*. As estatísticas atualmente disponíveis são:

- **Música mais reproduzida:** Percorre todas as músicas existentes, compara os contadores de reprodução e devolve a faixa com maior número de execuções.
- **Intérprete mais ouvido:** Agrupa as músicas por intérprete e soma o total de reproduções de cada um, devolvendo o artista mais ouvido.
- **Utilizador que mais músicas ouviu:** Para um dado intervalo de tempo (ou “desde sempre”), soma o número de reproduções por utilizador e identifica aquele com o maior total.
- **Utilizador com mais pontos:** Compara o valor do atributo pontos entre todos os utilizadores e devolve o que possui o maior valor acumulado.
- **Género mais reproduzido:** Agrupa as músicas por género musical, soma o número de reproduções em cada género e devolve o género mais popular.
- **Número de playlists públicas:** Consulta o repositório de playlists públicas e devolve a contagem atual.
- **Utilizador com mais playlists:** Conta quantas playlists cada utilizador tem e devolve aquele com maior número.

3.1.7 Persistência de Dados

Persistência dos dados da aplicação (utilizadores, álbuns, playlists públicas) é gerida pela classe utilitária *EstadodaAPP* (no pacote *utils*). Esta classe utiliza a serialização de objetos Java para guardar e carregar o estado da aplicação em ficheiros binários (com extensão *.dat*).

- O método `EstadoDaAPP.guardarObjeto(String caminho, Object objeto)` serializa um objeto para um ficheiro especificado.
- O método `EstadoDaAPP.carregarObjeto(String caminho, T valorPorDefeito)` deserializa um objeto de um ficheiro, retornando um valor por defeito se o carregamento falhar (e.g., ficheiro não encontrado, erro de desserialização).

O `SpotifUMModel` utiliza estes métodos no seu construtor para carregar o estado inicial e a classe `SpotifumAPP` (e `MenuModoDeAppScreen`) utiliza-os para guardar o estado ao encerrar a aplicação.

3.2 Interface com o Utilizador

A interface com o utilizador da aplicação SpotifUM é implementada inteiramente em modo texto, operando através de uma série de ecrãs na consola. Esta abordagem, embora simples, permite ao utilizador navegar pelas diversas funcionalidades da aplicação de forma estruturada. A interação é gerida pela camada View, que coordena a apresentação dos ecrãs e a recolha de input.

3.2.1 Navegação por Menus

O fluxo principal da aplicação é orientado por menus. Cada ecrã apresenta ao utilizador uma lista de opções numeradas. O utilizador interage inserindo o número correspondente à opção desejada.

- **Estrutura Hierárquica:** Os menus são organizados hierarquicamente. A partir de um menu principal, o utilizador pode navegar para submenus que agrupam funcionalidades relacionadas (e.g., gestão de utilizadores, gestão de álbuns). A opção "0" é convencionalmente utilizada para "Voltar" ao menu anterior ou sair de um contexto.
- **Consistência:** A apresentação dos menus é consistente em toda a aplicação, graças à utilização das classes base da View.
- **Feedback Imediato:** Após uma seleção, a aplicação geralmente fornece feedback imediato, seja navegando para um novo ecrã, apresentando uma mensagem de sucesso/erro, ou exibindo dados solicitados.

Ao iniciar, o utilizador é apresentado com o `MenuModoDeAppScreen`. A escolha aqui direciona o utilizador para o modo de administração do sistema ou para o modo de simulação de um utilizador final.

3.2.2 Tratamento de Input e Feedback

- **Input Numérico e Textual:** A maioria das interações de menu requer input numérico. Para a criação ou edição de entidades, é solicitado input textual. A classe `Scanner` é utilizada para ler este input.
- **Confirmações:** Operações destrutivas (como apagar um álbum ou utilizador) geralmente apresentam um ecrã de confirmação (e.g., `ApagarAlbumScreen`) onde o utilizador deve confirmar a ação ("Sim"/"Não").
- **Mensagens de Erro e Sucesso:** Após cada operação, o sistema fornece uma mensagem indicando se a operação foi bem-sucedida ou se ocorreu um erro (muitas vezes derivado das mensagens de exceções personalizadas lançadas pelo Model e tratadas pelo Controller).

- **Pausa para Leitura:** O método `esperarEnter()` é frequentemente utilizado após a exibição de mensagens ou resultados, garantindo que o utilizador tenha tempo para ler antes que o ecrã seja limpo ou atualizado para o próximo estado.

A interface, embora simples, é funcional e cobre as principais funcionalidades da aplicação de forma organizada através da sua estrutura de menus e ecrãs especializados, facilitada pela arquitetura da camada View discutida anteriormente.

Capítulo 4

Conclusão

Neste trabalho, desenvolvemos o SpotifUM, uma aplicação em *Java* para gerir músicas, álbuns, utilizadores e playlists. Adotando o padrão arquitetural Model-View-Controller (MVC), procurámos criar uma estrutura modular que separasse as responsabilidades da lógica de negócio, da apresentação dos dados e do controlo da interação do utilizador. Ao longo do desenvolvimento, implementámos funcionalidades essenciais como a criação, edição e remoção das principais entidades do sistema, um sistema de planos de subscrição com diferentes níveis de permissões e pontuações, mecanismos de reprodução de músicas de forma sequencial e aleatória, a capacidade de os utilizadores criarem e gerirem as suas playlists, e a recolha e apresentação de estatísticas de uso da plataforma.

A aplicação de outros padrões de projeto como Observer, Strategy, Simple Factory e Prototype foi fundamental para alcançar um design mais flexível e manutenível. O padrão Observer permitiu um baixo acoplamento na notificação de eventos, como a reprodução de músicas, atualizando dinamicamente o estado de diferentes componentes. O Strategy facilitou a gestão das diferentes lógicas de permissão dos planos de subscrição, enquanto o Simple Factory centralizou a criação destes planos. A utilização intensiva de clonagem foi crucial para proteger o encapsulamento e a integridade dos dados do modelo.

Os principais desafios encontrados durante o desenvolvimento incluíram a manutenção de uma separação clara entre a lógica do sistema (Model) e a sua apresentação (View), mesmo com uma interface de consola. A classe `SpotifUMModel` atuou como uma fachada para a lógica, mas a complexidade inerente à atualização do estado correto das entidades, especialmente após operações de clonagem (como garantir que o contador de reproduções da instância original de uma música é incrementado quando um clone é reproduzido), exigiu atenção particular. A persistência de dados através da serialização Java, embora simples de implementar, também apresentou desafios no que diz respeito à manutenção da coerência dos diversos mapas de dados (utilizadores, álbuns e playlists) e à gestão de versões das classes serializadas.

Em suma, o desenvolvimento da SpotifUM permitiu aplicar e explorar diversos conceitos de engenharia de software e padrões de projeto na construção de uma aplicação funcional. A arquitetura escolhida, embora com espaço para refinamentos, provou ser uma base sólida para as funcionalidades implementadas.

4.1 Trabalhos Futuros e Melhorias

Apesar das funcionalidades implementadas, existem diversas áreas onde a aplicação SpotifUM poderia ser melhorada e expandida:

- **Refinamento do Model:**

- **Atualização de Estatísticas e Contadores:** A forma como o `SpotifUMModel` lida com a atualização das estatísticas poderia ser menos pesada, implementando uma classe dedicada

para gerir as estatísticas globais, que atuaria como um observer de eventos relevantes (reproduções, criação de playlists, etc.), poderia centralizar e otimizar a recolha destes dados, em vez de calcular estatísticas no momento a partir do estado completo do Model.

- **Melhorias na Interface do Utilizador (View):**

- **Validação de Input na View:** Adicionar validações mais robustas e feedback mais imediato na própria View para os inputs do utilizador antes de serem enviados para o Controller.
- **Paginação e Pesquisa em Listagens:** Para grandes volumes de dados (muitos álbuns, músicas, etc.), as listagens atuais tornar-se-iam impraticáveis. Implementar paginação e funcionalidades de pesquisa e filtragem na View seria essencial.
- **Mais Funcionalidades Expostas:** Nem todas as funcionalidades do Model que foram testadas com sucesso (como tipos avançados de criação de playlist baseadas em preferências ou tempo máximo, que existem nas classes de playlist) estão completamente expostas ou são facilmente acessíveis através dos menus atuais da View. Devíamos expandir os menus para cobrir todas as capacidades do sistema.

- **Qualidade de Código e Testes:**

- **Cobertura de Testes:** Aumentar a cobertura dos testes unitários e adicionar testes de integração para garantir a robustez da aplicação.
- **Reestruturação Contínua:** Continuar a reestruturar e limpar o código para melhorar a clareza e remover duplicação.
- **Documentação:** Melhorar a documentação interna do código (JavaDocs) para todas as classes e métodos públicos.

Estas melhorias poderiam transformar a nossa SpotifUM numa aplicação mais completa, robusta e com uma melhor experiência de utilizador.