

1. Identificação de Abstrações:

- Identifique as abstrações naturais deste problema e explique por que "Música", "Playlist" e "Catálogo" são boas escolhas para abstrações.
- Descreva como essas abstrações ajudam a ocultar detalhes específicos (como a reprodução de áudio ou a manipulação de listas) e simplificam o desenvolvimento do sistema.

→ Abstrações Naturais: Música, Playlist e Catálogo

Música: Representa uma faixa individual, com informações como título, artista e duração

Playlist: Um conjunto de Música escolhida pelo usuário

Catálogo: Conjunto de todas músicas disponíveis no sistema

→ Quando usamos abstrações:

• Ocultam detalhes:

→ Por exemplo, quem usa a classe Playlist não precisa saber como as músicas são guardadas (lista, dicionário etc.), só precisa saber que pode adicionar, remover e listar músicas

O mesmo vale para catálogo, quem usa não precisa saber como é feita a busca, só chama um método e recebe o resultado

• Simplificam o desenvolvimento

→ Cada parte do sistema fica separada, facilitando

mudanças futuras.

Ou seja, as **abstrações** deixam o código mais organizado, fácil de entender e de modificar, pois escondem os detalhes internos e oferecem uma interface simples para o resto do sistema.

2. Implementação Imperativa:

- Descreva como resolveria esse problema usando o paradigma imperativo, sem abstrações significativas.
- Crie um código imperativo que:
 - Use variáveis para representar a lista de músicas, playlists, e informações de cada música.
 - Utilize estruturas de controle (como if e while) para gerenciar a adição de músicas a playlists e a reprodução de músicas.
 - Manipule diretamente as variáveis para exibir informações de músicas e playlists.
- **Reflexão sobre Abstração:** Explique como a abordagem imperativa lida com abstração e quais limitações você observa se o sistema for ampliado para incluir novos recursos, como busca avançada por artistas ou gêneros.

Na **abordagem imperativa**, geralmente não há muitas abstrações.

→ Os dados são trabalhados de forma direta, usando listas e dicionários

→ As operações, como buscar, adicionar, remover) são feitas por funções soltas ou blocos de códigos, que manipulam essas listas diretamente

→ Não existe um tipo próprio para Música ou playlist, tudo é dicionário ou lista.

Limitações:

Se o sistema crescer, para incluir:

• Busca avançada por artista ou gênero

- filtros personalizados
- Mais infos por músicas (como álbum, ano)
- Outros tipos de playlist

As limitações serão:

1. Código repetitivo e difícil de manter

→ Cada nova função precisa lidar diretamente com listas de dicionários. Se mudar o formato do dicionário, precava ajustar o código em vários lugares

2. Baixa reutilização

→ Não é fácil reaproveitar funções, porque elas geralmente dependem de como outros dados estão estruturados
Naquele momento

3. Falta de encapsulamento

→ Qualquer parte do código pode mexer nos dados a qualquer momento, dificultando o controle e a prevenção de bugs

→ Na abordagem imperativa até funciona para sistemas pequenos, mas quando o sistema cresce, fica difícil de manter, testar e evoluir, pois tudo depende de manipular listas e dicionários de forma direta

3. Implementação Orientada a Objetos:

- Descreva como resolveria o problema usando o paradigma orientado a objetos, aproveitando a abstração.
- Crie um código orientado a objetos que:
 - Inclua uma classe Musica com atributos como titulo, artista, duracao.
 - Crie uma classe Playlist que contém uma lista de objetos Musica e métodos para adicionar músicas e exibir a lista de músicas.
 - Crie uma classe Catalogo que contém a lista completa de músicas disponíveis e fornece métodos para buscar e listar músicas.
 - Utilize encapsulamento para que detalhes como listas e métodos de manipulação sejam ocultados do usuário final.
- **Reflexão sobre Abstração:** Explique como a abstração no paradigma orientado a objetos ajuda a modularizar o código e tornar o sistema mais flexível para adições de novas funcionalidades, como recomendações ou filtros.

→ No paradigma **Orientado a Objeto**, usamos classes para representar conceitos do sistema (como Música, Playlist, Catálogo). Essas classes **encapsulam** os dados e as operações relacionadas a elas.

Como modularizar?

- Cada parte do sistema fica responsável só pelo próprio comportamento (Ex: só a classe Playlist sabe como gerenciar suas músicas).
- O código é dividido em blocos independentes (módulos), que se comunicam por interfaces simples.
- Mudança em uma parte (por exemplo, como armazenar músicas) não afeta o resto do sistema, desde que a interface da classe não mude.

Como isso torna o sistema mais flexível?

- Para adicionar **novas funcionalidades** (como recomendações, filtros, busca avançada), basta criar novas classes ou métodos, sem precisar mexer no código que já funciona.
- Podemos criar diferentes tipos de Playlist (inteligentes, colaborativas) apenas herdando e adaptando classes existentes.
- É fácil testar e dar manutenção, pois cada módulo

tem uma responsabilidade bem definida

→ A **Abstração Orientada a Objetos** permite dividir o sistema em partes pequenas, bem organizadas e independentes, facilitando a evolução e adição de novas funções sem bagunçar o código já existente. Isso deixa o sistema mais seguro, reutilizável e fácil de crescer.

4. Comparação e Discussão:

- Compare as duas implementações e responda às perguntas a seguir:
 - Qual implementação é mais fácil de modificar, caso um novo recurso (como filtro por gênero) precise ser adicionado?
 - Qual abordagem permite um controle mais seguro e abstrato das músicas e playlists? Justifique.
 - Em qual caso o fluxo do programa ficou mais fácil de entender?
- Discuta como o conceito de abstração é tratado em cada paradigma, focando nas vantagens de organizar e ocultar detalhes em um sistema que pode crescer em complexidade.

3) A **Versão Orientada a objeto** é muito mais fácil de modificar. Se quisermos adicionar um filtro por gênero, basta incluir o campo "genero" na classe **Song** e adicionar um método de filtro na classe **Catalog**. Todo resto do código pode continuar igual, pois a manipulação está centralizada nessas classes.

Na **Versão Imperativa**, seria necessário adicionar o

tempo "gênero" em todos os dicionários de música e
criar funções extras que percorrem listas de dicionários,
o que aumenta a chance de erros e torna o código
repetitivo

2) A abordagem orientada a objeto permite um controle
mais seguro e abstrato.

- O acesso aos dados é feito por métodos das classes, e não diretamente pelas estruturas internas (lista ou dicionário)
- Só as próprias classes sabem como modificar ou acessar suas informações, o que evita bugs e alterações indevidas
- O resto do sistema só interage por interface (método) sem conhecer os detalhes internos

Na abordagem imperativa, qualquer parte do código pode alterar diretamente as listas e dict, o que torna difícil evitar bugs e manter o controle dos dados

3) No paradigma orientado a objetos, o fluxo fica mais fácil de entender. O código fica organizado em módulos (classes / arquivos), cada um com sua responsabilidade.
É mais claro ver o que é uma música, uma playlist e o que cada parte faz.

No **Imperativo**, o fluxo depende da ordem das funções e do uso de variáveis globais, o que pode confundir quando o programa cresce.

4) Imperativo

- ~ **Pouca abstração**. O programador trabalha diretamente com listas e dict.
- ~ Não existe separação clara entre o que é música, Playlist, catálogo; são só estruturas de dados soltas.
- ~ Detalhes de implementação (como armazenar ou filtrar músicas) aparecem em todo lugar do código.
- ~ **Dificulta o crescimento do sistema** e aumenta o risco de erros.

Orientado a Objeto

- ~ **Alta abstração**. Os conceitos principais viram classes com métodos próprios.
- ~ Os detalhes internos de cada classe ficam ocultos (^{en-}capsulados).
- ~ O programador usa métodos simples, sem se preocupar com como a lista de música está organizada.
- ~ **Facilita crescimento**, manutenção, teste e reutilização do sistema.

