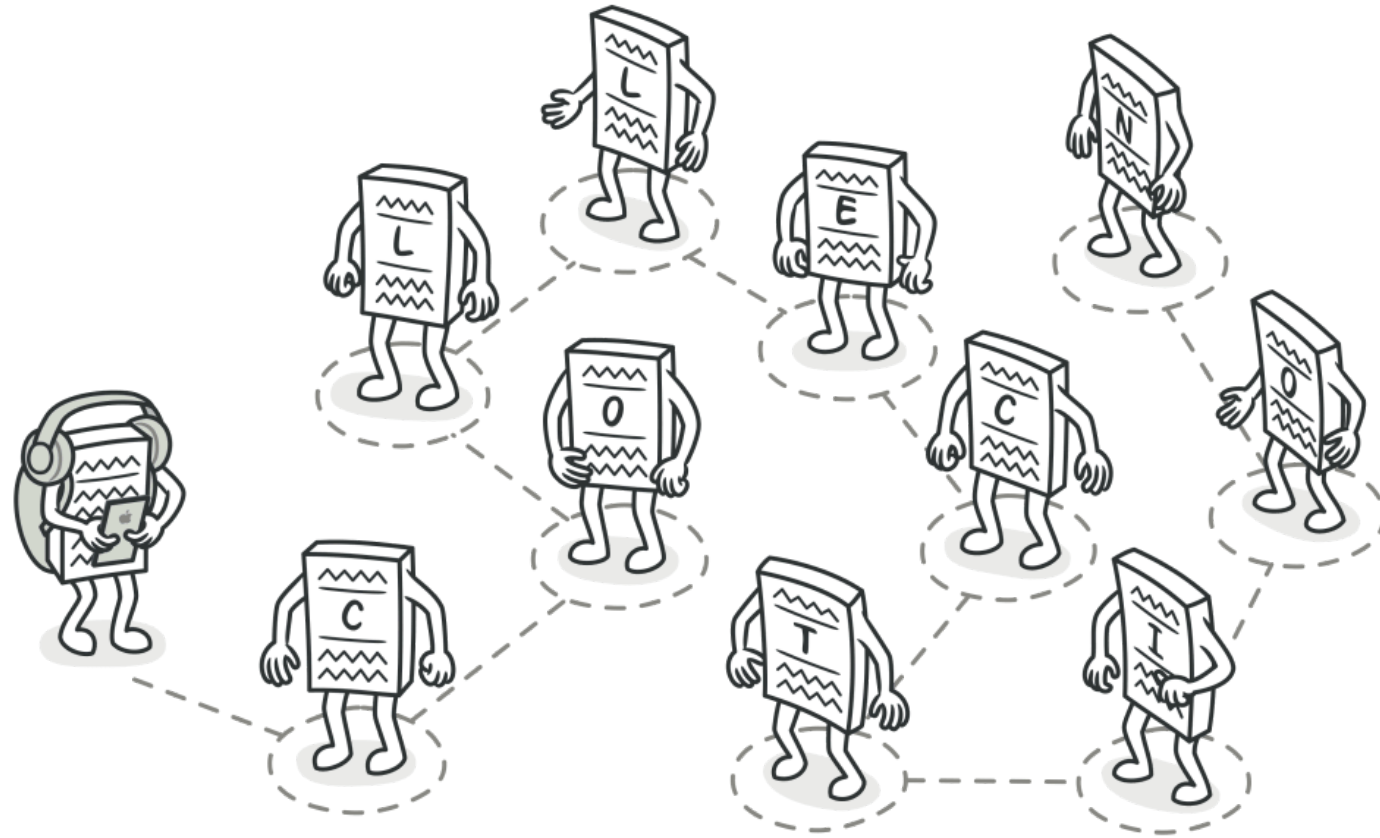


TÓPICO 13: ITERATOR

Design Patterns - Professor Ramon Venson - SATC 2026.2

Definição

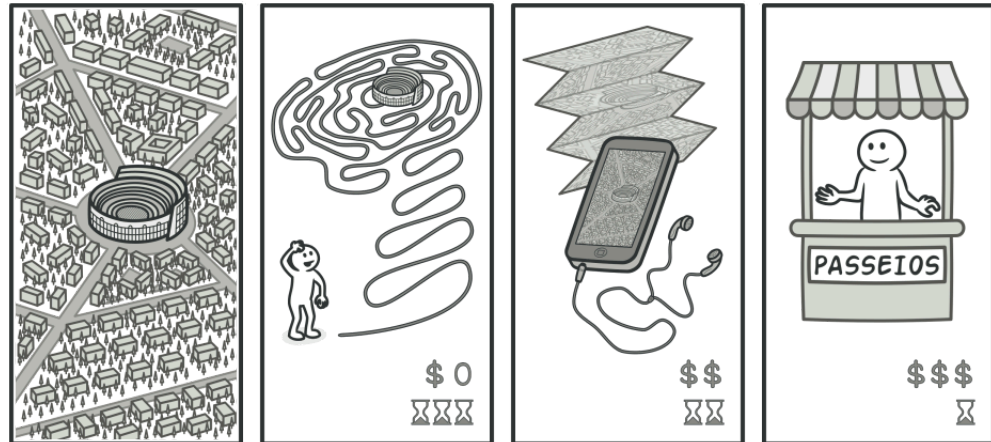
Iterator é um padrão de projeto **comportamental** que permite **percorrer os elementos de uma coleção** sem expor a sua **representação interna** (lista, array, árvore...).



Problema

Um app de música tem uma **playlist** e o cliente precisa passar por suas faixas. Sem o padrão, a travessia fica **espalhada** e presa à estrutura interna:

- a playlist pode ser uma `List`, um array ou nós ligados
- o cliente acessa os **detalhes internos** para percorrer
- cada forma de percorrer vira **código repetido**



```
public class Playlist {  
    private final List<String> faixas = new ArrayList<>();  
  
    public void adicionar(String faixa) {  
        faixas.add(faixa);  
    }  
  
    public List<String> getFaixas() {    // expõe a estrutura interna  
        return faixas;  
    }  
}  
  
// cliente conhece e manipula a lista por baixo  
for (String faixa : playlist.getFaixas()) {  
    System.out.println(faixa);  
}
```

Essa solução tem problemas sérios:

- **quebra o encapsulamento**: o cliente vê a estrutura interna
- trocar `List` por array ou árvore **quebraria** o cliente
- cada **modo de percorrer** duplica o laço por todo o sistema
- fica difícil oferecer travessias **simultâneas e independentes**
- o cliente precisa **conhecer** a forma de armazenamento

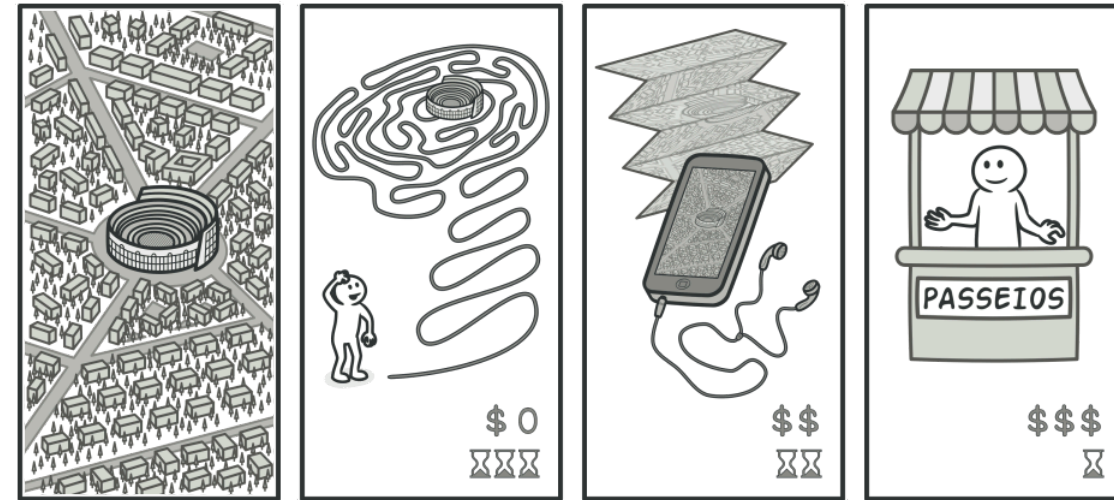
Solução

Vamos **extrair a travessia** para um objeto próprio: um **iterador** que sabe caminhar pela coleção, enquanto o cliente só vê o próximo elemento.

- **Iterator**: interface com `proxima()` e `temProxima()`
- **Concreteliterator**: mantém a **posição** e conhece a estrutura
- **Aggregate**: a coleção que **cria** o iterador
- O cliente **não sabe** como a coleção armazena os dados

A ideia central:

- a coleção **fabrica** um iterador para si
- o iterador guarda o **estado da travessia** (posição)
- o cliente avança **um elemento por vez**
- vários iteradores podem percorrer a **mesma coleção** sem interferir



Implementação

1. Primeiro, definimos a **interface do iterador**:

```
public interface IteradorPlaylist {  
    boolean temProxima();  
    String proxima();  
}
```

2. Criamos o **iterador concreto** da travessia em ordem:

```
public class IteradorSequencial implements IteradorPlaylist {  
    private final List<String> faixas;  
    private int posicao = 0;  
  
    public IteradorSequencial(List<String> faixas) {  
        this.faixas = faixas;  
    }  
  
    public boolean temProxima() {  
        return posicao < faixas.size();  
    }  
  
    public String proxima() {  
        return faixas.get(posicao++);  
    }  
}
```

3. A **coleção** (Aggregate) cria o iterador, sem expor os detalhes:

```
public class Playlist {  
    private final List<String> faixas = new ArrayList<>();  
  
    public void adicionar(String faixa) {  
        faixas.add(faixa);  
    }  
  
    public IteradorPlaylist criarIterador() {  
        return new IteradorSequencial(faixas);  
    }  
}
```

4. Para embaralhar, basta um **novo iterador**, a `Playlist` nem muda:

```
public class IteradorEmbaralhado implements IteradorPlaylist {  
    private final List<String> copia;  
  
    public IteradorEmbaralhado(List<String> faixas) {  
        this.copia = new ArrayList<>(faixas);  
        Collections.shuffle(copia);  
    }  
  
    public boolean temProxima() { return !copia.isEmpty(); }  
  
    public String proxima() { return copia.remove(0); }  
}
```

Utilização

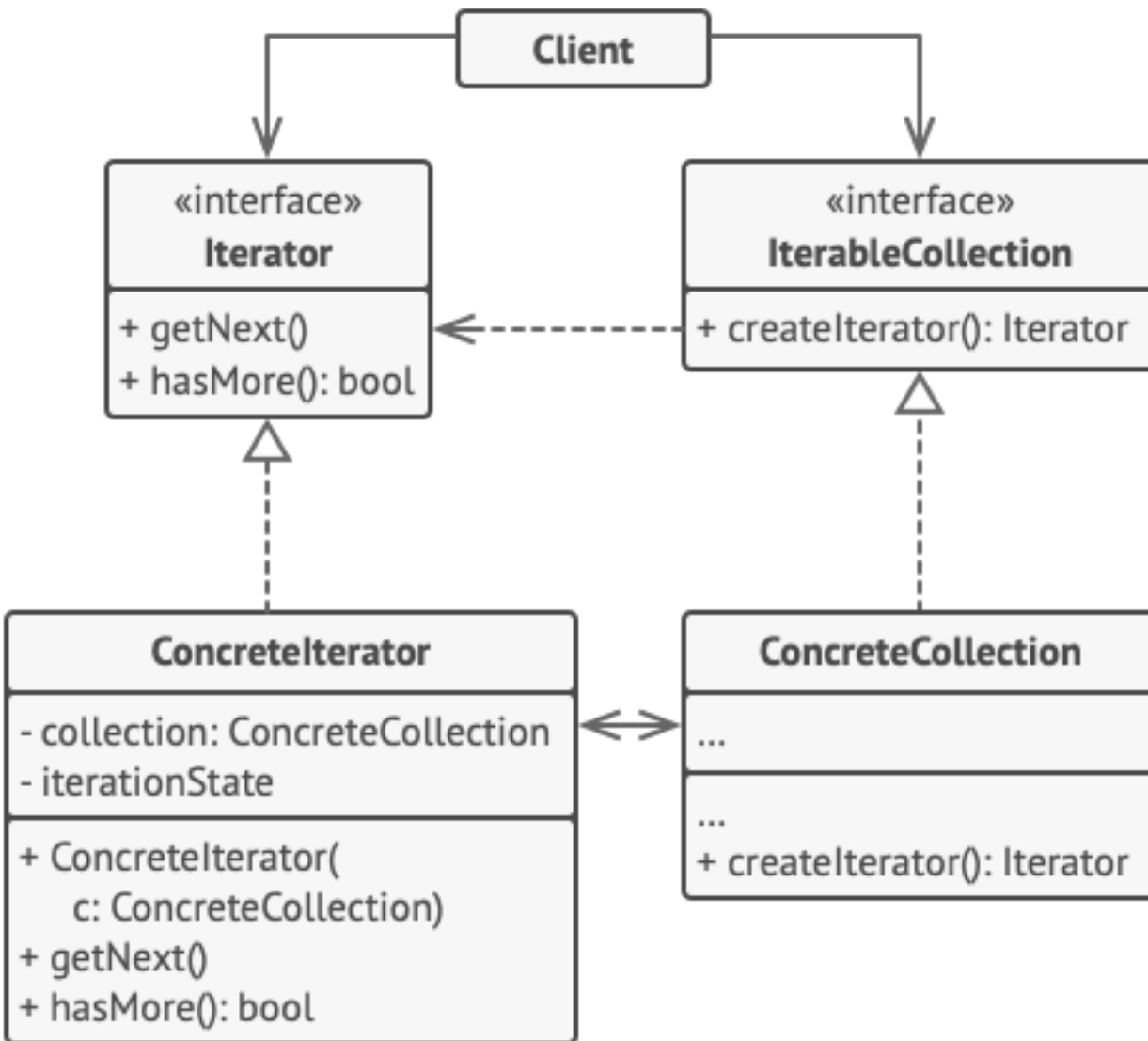
O cliente percorre com **um único protocolo**, sem conhecer a estrutura:

```
Playlist playlist = new Playlist();  
playlist.adicionar("Faixa A");  
playlist.adicionar("Faixa B");  
  
IteradorPlaylist it = playlist.criarIterador();  
while (it.temProxima()) {  
    System.out.println(it.proxima());  
}
```

- o cliente só conhece `temProxima()` e `proxima()`
- trocar a travessia = **trocar o iterador**, não o cliente
- a `Playlist` pode mudar de `List` para array **sem quebrar** ninguém
- percorrer favoritas ou embaralhado vira **só mais um iterador**

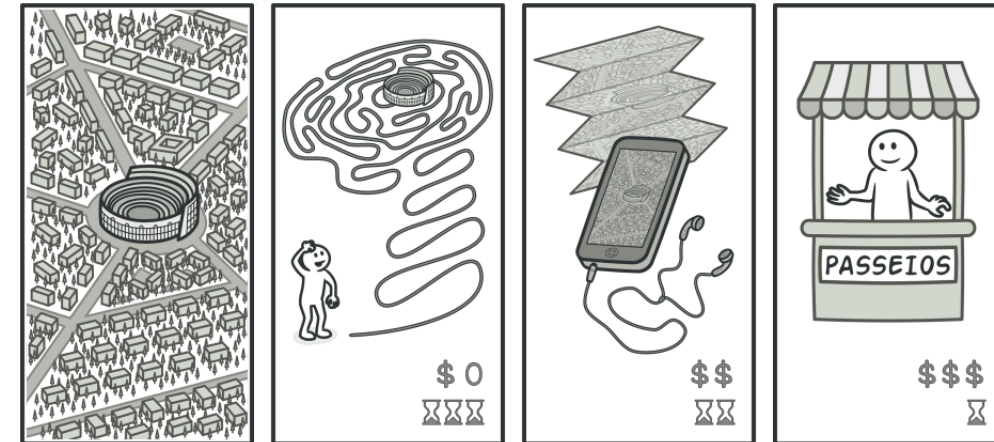
Diagramas

- **Iterator**: interface de travessia
(`proxima` , `temProxima`)
- **Concreteliterator**: implementa a travessia e guarda a posição
- **Aggregate**: interface da coleção, com `criarIterador()`
- **ConcreteAggregate**: cria o iterador concreto
- **Client**: percorre pela interface, sem ver a estrutura



Analogias

- **Álbum de figurinhas:**
 - você vira **uma figurinha por vez**, sempre sabendo se ainda há páginas
 - não precisa saber como o álbum está **encadernado** por dentro
 - dá para duas pessoas folhearem o **mesmo álbum** em ritmos diferentes
 - **Limitação:** se o álbum ganhar novas figurinhas durante o folhear, a posição atual pode "sair do lugar"



Aplicações

- Percorrer coleções **sem expor** a estrutura interna
- Oferecer **várias travessias** sobre a mesma coleção
- Permitir **vários iteradores simultâneos** e independentes
- Padronizar o acesso a estruturas diferentes com **um só protocolo**

Exemplos típicos:

- `java.util.Iterator` e o **for-each** (`Iterable`)
- percorrer **árvores**: pastas, DOM, cenas gráficas
- **paginação** e leitura de dados em **streaming**
- iteradores de banco de dados (cursor por linhas)

Prós

- **SRP**: a travessia sai da coleção para o iterador
- **OCP**: novas travessias entram sem alterar a coleção
- **Encapsulamento**: o cliente não vê a estrutura interna
- Vários iteradores **independentes** na mesma coleção
- Protocolo **uniforme** para estruturas diferentes

Contras

- Pode ser **overengineering** para uma coleção simples e fixa
- **Mais classes** (interface + iteradores concretos)
- Risco de **invalidação** ao alterar a coleção durante a travessia
- Iteradores internos podem ser **menos flexíveis** que um laço direto

Regra: use quando a **travessia** é responsabilidade própria ou a estrutura deve ficar escondida.

Resumo

- **Iterator**: percorre uma coleção **sem expor** a estrutura interna
- A coleção **cria** o iterador; ele guarda o **estado da travessia**
- Vários iteradores podem percorrer a **mesma coleção** sem interferir
- É o padrão por trás do `for-each` e de `java.util.Iterator`
- Diferença do **Composite**: o Composite monta a estrutura parte-todo; o Iterator apenas **caminha** por ela (ou por qualquer coleção)