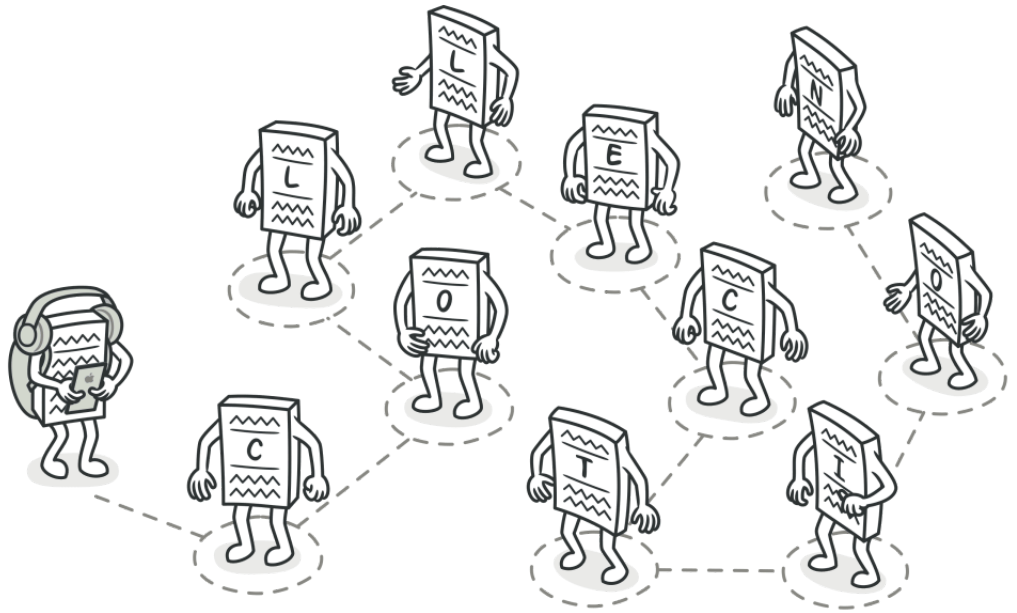
TÓPICO 17 - ITERATOR

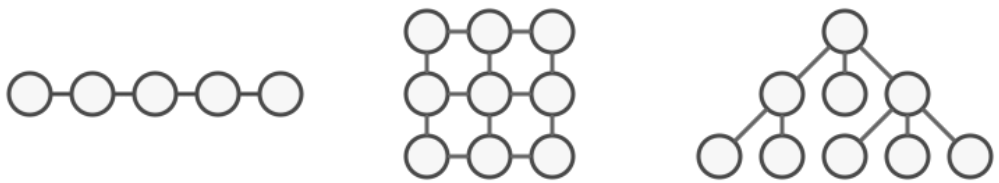
Design Patterns - Professor Ramon Venson - SATC 2026.1

Motivação

Coleções aparecem o tempo todo em software.

O problema começa quando o cliente precisa conhecer **como** cada coleção organiza seus dados para conseguir percorrê-la.



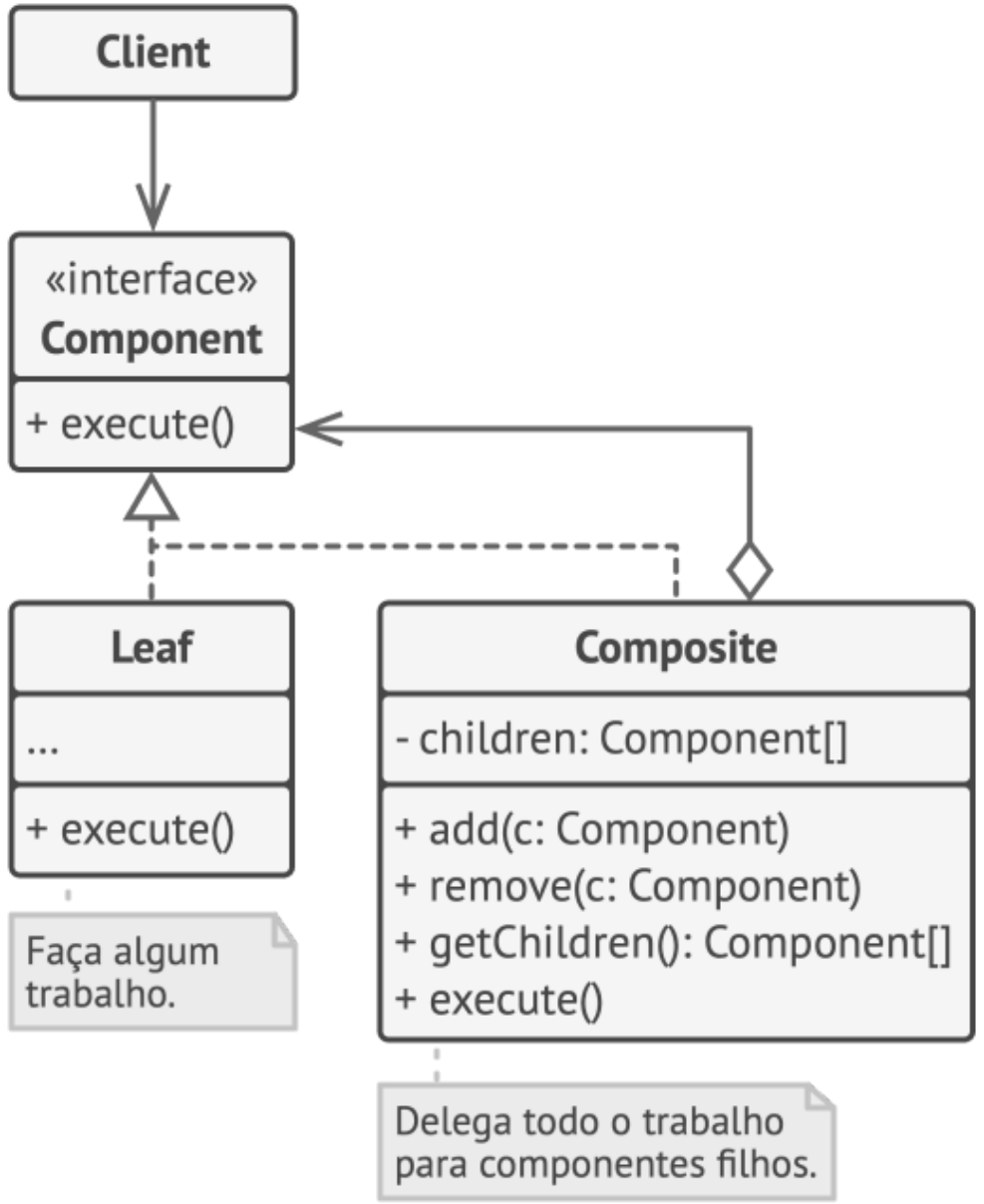


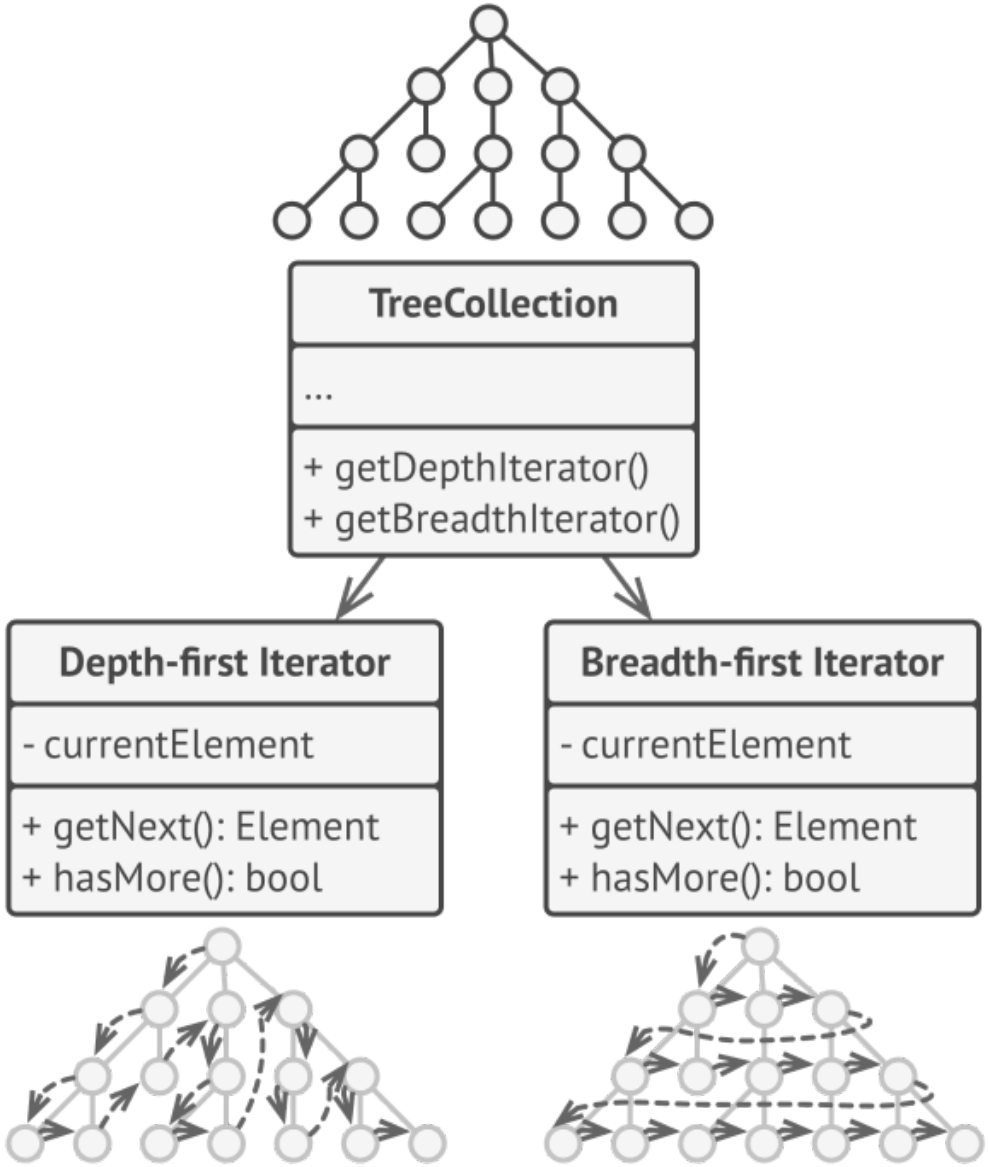
Problema

- Uma lista pode ser percorrida facilmente
- Uma árvore já exige outra lógica
- Um grafo pode exigir filtros e controle extra
- O cliente não deveria saber esses detalhes
- A travessia espalhada gera acoplamento e duplicação

Conceito principal

- **Iterator** é um padrão comportamental
- Ele separa travessia de armazenamento
- A coleção cria o iterador
- O iterador controla posição e avanço
- O cliente usa interface comum





Como funciona

- A coleção expõe um método para obter iteradores
- Cada iterador guarda seu próprio estado
- O cliente chama `temProximo()` e `proximo()`
- Diferentes percursos podem coexistir
- A coleção permanece encapsulada

Estrutura / Componentes

- **Iterator**: define operações de navegação
- **ConcreteIterator**: implementa o percurso
- **Aggregate**: expõe criação de iteradores
- **ConcreteAggregate**: mantém os dados
- **Cliente**: usa a interface do iterador

Código: interface e coleção

```
interface IteradorDisciplina {
    boolean temProximo();
    String proximo();
}

class GradeCurricular {
    private final List<String> disciplinas = new ArrayList<>();

    public IteradorDisciplina criarIterador() {
        return new GradeCurricularIterador(disciplinas);
    }
}
```

Código: iterador concreto

```
class GradeCurricularIterador implements IteradorDisciplina {
    private final List<String> disciplinas;
    private int posicaoAtual = 0;

    public boolean temProximo() {
        return posicaoAtual < disciplinas.size();
    }

    public String proximo() {
        return disciplinas.get(posicaoAtual++);
    }
}
```

Quando usar

- A estrutura interna não deve ser exposta
- Existem vários modos de percorrer dados
- O cliente usa diferentes coleções
- Há duplicação de lógica de travessia
- A iteração precisa ter estado próprio

Quando não usar

- A coleção é muito simples
- O custo da abstração não compensa
- Não há variação real de percurso
- A equipe só quer seguir o catálogo
- Um `for` simples já resolve com clareza

Vantagens

- Reduz acoplamento com coleções concretas
- Centraliza lógica de travessia
- Permite percursos alternativos
- Suporta iterações paralelas independentes
- Facilita evolução do design

Desvantagens

- Adiciona mais classes e interfaces
- Pode ser excesso em casos simples
- Introduz indireção extra
- Exige cuidado com estado interno
- Pode dificultar depuração inicial

Relações com outros padrões

- **Composite** usa percursos sobre árvores
- **Factory Method** pode criar iteradores
- **Memento** pode salvar estado da iteração
- **Visitor** combina percurso com operações
- Todos reforçam encapsulamento e extensibilidade

Resumo final

- Iterator separa acesso e armazenamento
- O cliente percorre elementos por interface comum
- A coleção continua encapsulada
- Novos percursos surgem sem quebrar clientes
- O padrão ajuda quando iterar ficou um problema de design