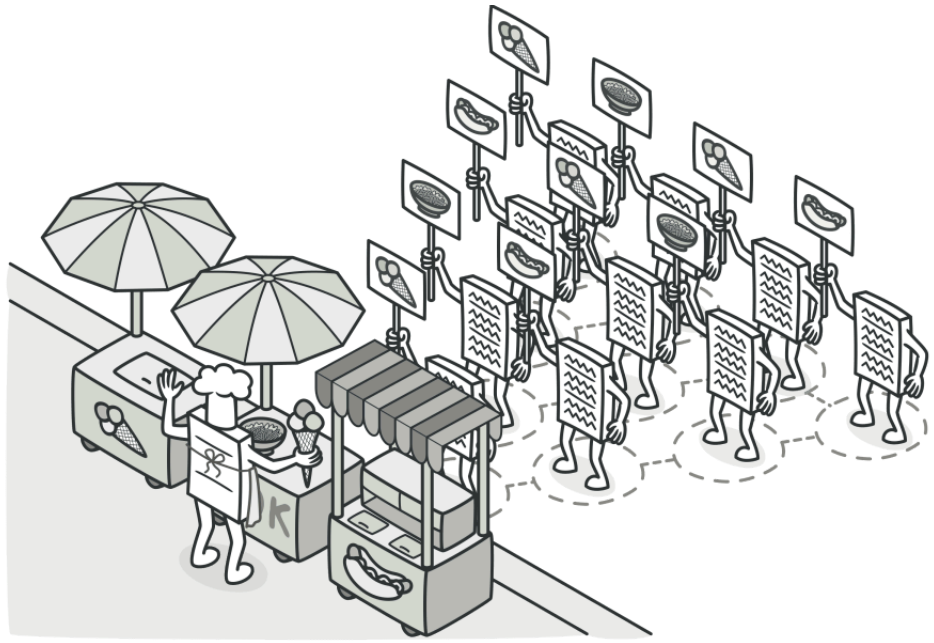
TÓPICO 20 - VISITOR

Design Patterns - Professor Ramon Venson - SATC 2026.1

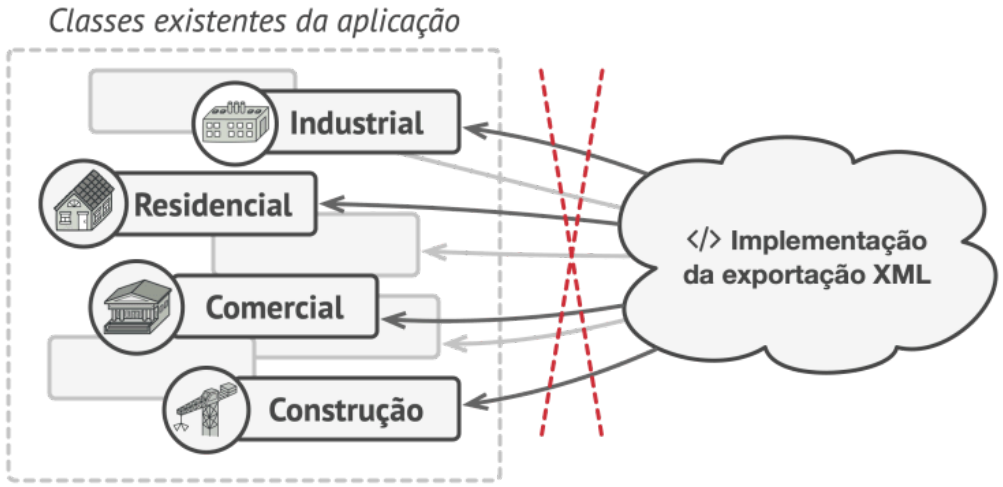
Motivação

Muitos sistemas têm uma estrutura com vários tipos de objetos e precisam executar novas operações sobre ela com frequência.

O desafio é adicionar esses comportamentos sem inflar as classes do domínio a cada novo requisito.



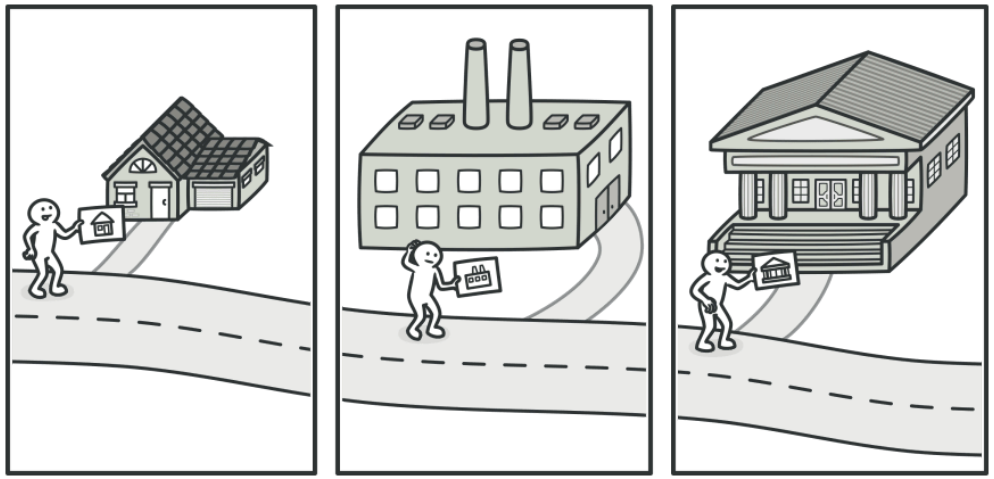
Problema

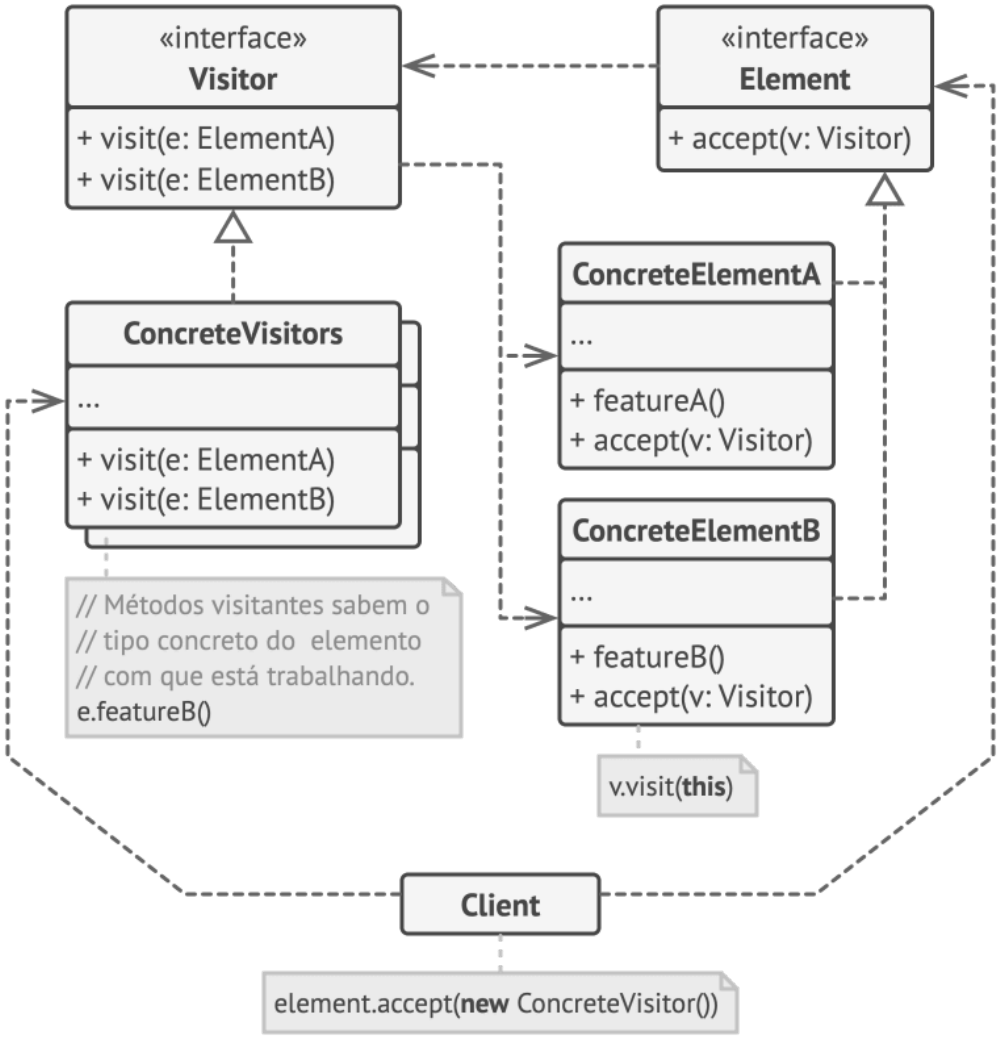


- Há vários tipos de elementos na mesma estrutura
- Novas operações aparecem com frequência
- Colocar tudo nas classes principais reduz coesão
- Cada novo comportamento exige mudar muitos arquivos
- `if`, `instanceof` e lógica espalhada viram tentação constante

Conceito principal

- **Visitor** é um padrão comportamental
- Ele separa algoritmos da estrutura visitada
- Os elementos recebem um visitante externo
- O visitante aplica lógica específica para cada tipo concreto
- Isso facilita adicionar novas operações sem mexer na hierarquia principal





Papéis do padrão

- **Elemento**: declara `aceitar(visitor)`
- **Elemento concreto**: redireciona para o método correto do visitante
- **Visitor**: declara um método para cada tipo concreto
- **Visitor concreto**: implementa um algoritmo específico
- **Cliente**: percorre a estrutura e inicia a visita

Double dispatch

- O cliente chama `elemento.aceitar(visitor)`
- O elemento concreto sabe quem ele é
- Então ele invoca algo como `visitor.visitarTexto(this)`
- A operação final depende do visitante **e** do elemento concreto
- Isso evita cadeias de condicionais para descobrir o tipo do objeto

Fluxo de uso

- Crie um visitante concreto, como `ExportadorHtmlVisitor`
- Percorra a estrutura de objetos
- Cada elemento aceita o visitante
- O visitante executa a lógica específica daquele tipo
- O mesmo conjunto de elementos pode receber vários visitantes diferentes

Código: interface base

```
interface Visitor {
    void visitarTexto(Texto texto);
    void visitarTabela(Tabela tabela);
}

interface ElementoRelatorio {
    void aceitar(Visitor visitor);
}
```

Código: elementos concretos

```
class Texto implements ElementoRelatorio {
    @Override
    public void aceitar(Visitor visitor) {
        visitor.visitarTexto(this);
    }
}

class Tabela implements ElementoRelatorio {
    @Override
    public void aceitar(Visitor visitor) {
        visitor.visitarTabela(this);
    }
}
```

Código: visitante concreto

```
class ExportadorMarkdownVisitor implements Visitor {
    @Override
    public void visitarTexto(Texto texto) {
        System.out.println(texto.getConteudo());
    }

    @Override
    public void visitarTabela(Tabela tabela) {
        System.out.println("## Tabela: " + tabela.getTitulo());
    }
}
```

Quando usar

- Há muitas operações sobre a mesma hierarquia
- A estrutura de elementos é relativamente estável
- Você quer limpar a lógica de negócio de comportamentos auxiliares
- A aplicação percorre árvores ou coleções heterogêneas
- Exportação, validação e métricas mudam mais que os tipos do modelo

Quando não usar

- A hierarquia ganha novos tipos o tempo todo
- Há poucas operações e solução simples resolve
- O visitante exigiria expor dados sensíveis demais
- O padrão seria usado apenas por catálogo
- O custo de manutenção supera o ganho de organização

Vantagens

- Facilita adicionar novos comportamentos
- Melhora separação de responsabilidades
- Centraliza o algoritmo em uma classe visitante
- Funciona bem com árvores e estruturas compostas
- Permite acumular estado durante a travessia

Desvantagens

- Novo tipo de elemento obriga atualizar todos os visitantes
- Pode aumentar acoplamento com classes concretas
- Às vezes pede acesso extra a dados internos
- Adiciona indireção e mais tipos ao modelo
- Pode ser excesso de engenharia em cenários pequenos

Relações com outros padrões

- **Composite**: Visitor percorre bem árvores de objetos
- **Iterator**: pode ajudar a caminhar pela estrutura visitada
- **Command**: encapsula ação, mas Visitor atua sobre vários tipos de elemento
- **Interpreter**: árvores sintáticas costumam usar Visitor para análise e geração
- **Double Dispatch**: é a base técnica da seleção do método correto

Resumo final

- Visitor separa operações da estrutura de objetos
- Cada elemento aceita o visitante e delega a chamada correta
- O padrão favorece novos comportamentos com menos impacto na hierarquia
- Ele funciona melhor quando os elementos mudam menos que as operações
- Use com critério, porque novos tipos de elemento custam mais caro