PADRÕES DE PROJETO – COMPORTAMENTO

Artur Monteiro Parreiras
Brenda Stefany de Oliveira Rocha
Bruna Letícia Silva
Eduarda Faria Pinheiro
Elenice Florentina de Oliveira dos Reis
Marco Aurélio Faria Ramos

Padrão Chain of Responsibility

- Define uma cadeia de objetos "handlers" (manipuladores) por onde uma solicitação pode passar.
- Quando um objeto (handler) recebe uma solicitação, ele decide se processa a solicitação ou se a passa para o próximo na cadeia.
- Permite desacoplar quem envia a solicitação de quem a recebe, já que o remetente não sabe quem vai tratar.

Padrão Chain of Responsibility

- Solicitação: Pedido de saque de R$ 250.
- Handlers: Dispenser de R$ 100 -> Dispenser de R$ 50 -> Dispenser de R$ 20.
- Ação: O cliente envia a solicitação de R$ 250 para o primeiro handler (o de R$ 100).
- Processamento em Cadeia:
- O handler de R$ 100 processa (libera 2 notas) e passa o resto (R$ 50) para o próximo.
- O handler de R$ 50 processa (libera 1 nota) e passa o resto (R$ 0).
- O cliente não sabe quantos handlers existem, ele apenas envia a solicitação para o início da cadeia.

Padrão Chain of Responsibility

- Handler (Manipulador): interface (ou classe abstrata) que define o método handleRequest() e mantém uma referência (proximo) para o próximo handler.
- ConcreteHandler (Manipulador Concreto): implementa o Handler. Decide se pode tratar o pedido. Se não puder, chama proximo.handleRequest().
- Client (Cliente): monta a cadeia (ligando um handler no setProximo() do outro) e envia a solicitação para o primeiro handler.

Padrão Chain of Responsibility – Código

```
// 1. O Objeto da Solicitação
public class Despesa {
    private double valor;
    public Despesa(double v) { this.valor = v; }
    public double getValor() { return valor; }
}

// 2. O Handler Abstrato
abstract class Aprovador {
    protected Aprovador proximo;
    public void setProximo(Aprovador p) { this.proximo = p; }
    public abstract void aprovar(Despesa d);

// 3. Os Concrete Handlers
class Gerente extends Aprovador {
    public void aprovar(Despesa d) {
        if (d.getValor() <= 1000) {
            System.out.println("Gerente aprovou R$" + d.getValor());
        } else if (proximo != null) {
            proximo.aprovar(d); // Passa para o próximo
        }
    }
}
class Diretor extends Aprovador {
    public void aprovar(Despesa d) {
        if (d.getValor() <= 5000) {
            System.out.println("Diretor aprovou R$" + d.getValor());
        } else if (proximo != null) {
            proximo.aprovar(d); // Passa para o próximo
        }
    }
}

}// 4. O Cliente
public class Main {
    public static void main(String[] args) {
        Aprovador gerente = new Gerente();
        Aprovador diretor = new Diretor();
        gerente.setProximo(diretor); // Monta a cadeia

        // Envia solicitações para o início da cadeia
        gerente.aprovar(new Despesa(800));
        gerente.aprovar(new Despesa(3500));
    }
}
```

Padrão Command

- O Command é um padrão de projeto comportamental que tem como objetivo encapsular uma solicitação (ação) como um objeto.
- Isso permite que ações sejam passadas como parâmetros, armazenadas, desfeitas ou executadas posteriormente.

Padrão Command

- Command : Interface que declara o método execute().
- ConcreteCommand : Implementa o comando específico e liga o comando ao receptor.
- Receiver : Objeto que executa a ação real.
- Invoker : Responsável por solicitar a execução do comando.
- Client : Cria o comando e associa o invoker ao comando.

Exemplo: com controle remoto

Não sabe como a luz funciona nem como e ligada, apenas sabe que ele deve ligar ela

Padrão Command

Vantagens

- Desacopla o emissor (Invoker) do executor (Receiver).
- Permite undo/redo (basta armazenar comandos executados).
- Facilita filas e logs de comandos executados.
- É flexível e extensível — novos comandos podem ser adicionados facilmente.

Desvantagens

- Pode aumentar a complexidade do código, pois cada ação precisa de uma classe de comando separada.
- Em sistemas com muitas ações, o número de classes pode crescer rapidamente.

Padrão Command

Quando usar

- Quando deseja desacoplar o solicitante de uma operação do executor.
- Quando é necessário desfazer/refazer ações.
- Quando ações precisam ser armazenadas, enfileiradas ou agendadas.

Padrão Command - Código

```
// Interface Command
public interface Command {
  void execute(); // Método que todos os comandos devem implementar
}

// Receiver
public class Luz {
  public void ligar() { System.out.println("Luz ligada."); }
  public void desligar() { System.out.println("Luz desligada."); }
}

// ConcreteCommand
public class LigarLuzCommand implements Command {
  private Luz luz;
  public LigarLuzCommand(Luz luz) { this.luz = luz; }
  public void execute() { luz.ligar(); }
}
```

```
// Invoker
public class ControleRemoto {
  private Command comando;
      public void setComando(Command comando) { this.comando =
comando; }
  public void pressionarBotao() { comando.execute(); }
}

// Client
public class Main {
  public static void main(String[] args) {
    Luz luz = new Luz(); // Receptor
        Command ligarLuz = new LigarLuzCommand(luz); // Cria comando
concreto
    ControleRemoto controle = new ControleRemoto(); // Invoker
    controle.setComando(ligarLuz); // Associa comando ao botão
    controle.pressionarBotao(); // Executa o comando -> "Luz ligada."
  }
}
```

Padrão interpret

- O Interpreter é um padrão de projeto comportamental usado para interpretar sentenças de uma linguagem específica (DSL).
- Ele define uma gramática e usa classes para representar regras e expressões dessa gramática.
- É como construir um mini interpretador ou avaliador de expressões dentro do seu código

Padrão interpret

- AbstractExpression: Interface base com o método interpret (Context).
- TerminalExpression: Representa elementos simples (números, variáveis, etc).
- NonTerminalExpression: Representa combinações de expressões (operações, regras).
- Context: Armazena informações globais necessárias para a interpretação.
- Client : Constrói as expressões e solicita sua interpretação.

Exemplo: matemática

Suponha uma linguagem simples que interpreta expressões matemáticas, como: (5 + 10) – 3
O objetivo é fazer o programa entender e calcular o resultado.

Padrão interpret

Vantagens

- Facilita a criação de linguagens específicas.
- Permite extensão fácil da gramática.
- Separa interpretação da estrutura da linguagem.

Desvantagens

- Pode gerar muitas classes em gramáticas complexas.
- O desempenho é menor em comparações com interpretadores otimizados.
- Difícil de manter em linguagens muito grandes.

Padrão interpret

Quando usar

- Quando há uma linguagem simples a ser interpretada (como regras, expressões ou comandos).
- Quando precisa avaliar expressões dinâmicas ou definir regras configuráveis em tempo de execução.

Padrão interpret – Código

```
// Contexto
public class Context {
  // Pode conter variáveis, funções, etc.
}

// Expressão abstrata
public interface Expressao {
  int interpretar(Context contexto);
}

// Expressão terminal (valor fixo)
public class Numero implements Expressao {
  private int numero;
  public Numero(int numero) { this.numero = numero; }
  public int interpretar(Context contexto) { return numero; }
}
```

```
// Expressão não terminal (soma)
public class Soma implements Expressao {
  private Expressao esquerda, direita;
  public Soma(Expressao esquerda, Expressao direita) {
    this.esquerda = esquerda;
    this.direita = direita;
  }
  public int interpretar(Context contexto) {
    return esquerda.interpretar(contexto) + direita.interpretar(contexto);
  }
}

// Expressão não terminal (subtração)
public class Subtracao implements Expressao {
  private Expressao esquerda, direita;
  public Subtracao(Expressao esquerda, Expressao direita) {
    this.esquerda = esquerda;
    this.direita = direita;
  }
  public int interpretar(Context contexto) {
    return esquerda.interpretar(contexto) - direita.interpretar(contexto);
  }
}
```

Padrão interpret – Código

```
// Client
public class Main {
  public static void main(String[] args) {
    Context contexto = new Context();

    // (5 + 10) - 3
    Expressao expressao = new Subtracao(
      new Soma(new Numero(5), new Numero(10)),
      new Numero(3)
    );

                                    System.out.println("Resultado:          "          +
expressao.interpretar(contexto));
    // Saída: Resultado: 12
  }
```

Padrão Iterator (Iterador)

- Define uma maneira de acessar os elementos de um objeto agregado (uma coleção) sequencialmente.
- Permite acessar os elementos sem expor sua representação interna (seja um Array, List, Map, etc.).
- Move a responsabilidade de "como percorrer" a coleção da própria coleção para um objeto Iterator separado.
- Permite que o código cliente percorra diferentes tipos de coleções usando a mesma interface (hasNext(), next()).

Padrão Iterator – Conceito

- Abstração (Aggregate): Um ResultSet de um banco de dados (o conjunto de resultados de uma query).
- Implementação (Iterator): Um Cursor (o objeto que você recebe para "andar" pelos resultados).
- Ação: O Cliente (seu código) executa uma query no banco e recebe um Cursor.
- Processo: O Cliente usa métodos padronizados como cursor.hasNext() e cursor.next() para ler linha por linha.
- Desacoplamento: O Cliente não precisa saber como o banco de dados armazena ou busca esses dados (se é um índice, uma tabela temporária, etc.). Ele apenas usa a interface do Cursor (o Iterator) para navegar pelos dados.

Iterator – Componentes

- Iterator (Interface): Define a interface para a navegação. Os métodos mais comuns são hasNext() (verifica se há um próximo) e next() (retorna o próximo elemento).
- ConcreteIterator (Iterador Concreto): implementa a interface Iterator. Ele mantém o estado da iteração (ex: a posição atual, currentIndex) e sabe como navegar na coleção concreta.
- Aggregate (Agregado - Interface): Define a interface para a coleção, incluindo um método "fábrica" para criar o iterador (ex: createIterator()).
- ConcreteAggregate (Agregado Concreto): implementa a interface Aggregate e seu método createIterator(), retornando uma instância do ConcreteIterator que sabe como percorrer aquela coleção específica.

Padrão Iterator – Código

```
import java.util.ArrayList;
import java.util.List;

// 1. Interface Iterator
interface Iterator<T> {
    boolean hasNext();
    T next();
}
// 2. Interface Aggregate (A Coleção)
interface Aggregate<T> {
    Iterator<T> createIterator();
}
// 3. ConcreteAggregate (A Coleção Concreta)
class Biblioteca implements Aggregate<String> {
    private String[] livros; // Representação interna
    public Biblioteca(String[] livros) {
        this.livros = livros;
    }
@Override
 public Iterator<String> createIterator() {
 // Retorna o iterador que sabe como percorrer um array
 return new LivroIterator(this.livros);
 }
}
 4. ConcreteIterator
class LivroIterator implements Iterator<String> {
 private String[] livros;
 private int index = 0;
 public LivroIterator(String[] livros) {
 this.livros = livros;
 }
```

Padrão Iterator – Código

```
import java.util.ArrayList;
import java.util.List;

// 1. Interface Iterator
interface Iterator<T> {
    boolean hasNext();
    T next();
}
// 2. Interface Aggregate (A Coleção)
interface Aggregate<T> {
    Iterator<T> createIterator();
}
// 3. ConcreteAggregate (A Coleção Concreta)
class Biblioteca implements Aggregate<String> {
    private String[] livros; // Representação interna
    public Biblioteca(String[] livros) {
        this.livros = livros;
    }
@Override
 public Iterator<String> createIterator() {
 // Retorna o iterador que sabe como percorrer um array
 return new LivroIterator(this.livros);
 }
}
 4. ConcreteIterator
class LivroIterator implements Iterator<String> {
 private String[] livros;
 private int index = 0;
 public LivroIterator(String[] livros) {
 this.livros = livros;
 }
```

Padrão Mediator (Mediador)

Definição e Propósito

- O padrão Mediator propõe a criação de um objeto intermediário responsável por coordenar a comunicação entre diversos objetos.
- Elimina a dependência direta entre eles, reduzindo o acoplamento e melhorando a coesão do sistema.
- Encapsula a forma como os objetos interagem, tornando a comunicação mais organizada.

[ComponentA] → [Mediator] ← [ComponentB]

Padrão Mediator (Mediador)

Estrutura do Padrão Mediator

Conteúdo (tópicos curtos):

- Mediator (Interface): define métodos de comunicação.
- ConcreteMediator: implementa a lógica específica.
- Componentes: comunicam-se apenas com o mediador.

Quando aplicar:

- Quando há muitos objetos interdependentes.
- Para centralizar a comunicação entre componentes.
- Para reduzir o acoplamento entre classes.

Padrão Mediator –Código

```
   // Exemplo simplificado do padrão Mediator
interface Mediator { void notify(String event); }

class DialogMediator implements Mediator {
   private TextBox t;
   public void set(TextBox t) { this.t = t; }

   // Centraliza a comunicação entre componentes
   public void notify(String e) {
      if (e.equals("click")) t.setText("Botão clicado!");
   }
}
class Button {
   private Mediator m;
   public Button(Mediator m) { this.m = m; }
    public void click() { m.notify("click"); } // Envia evento ao
mediador
}
class TextBox {
   public void setText(String v) { System.out.println("Texto:
" + v);
}
public class Main {
   public static void main(String[] a) {
      DialogMediator m = new DialogMediator();
      TextBox t = new TextBox();
      m.set(t);
      new Button(m).click(); // Saída: Botão clicado!
   }
}
}
```

O Mediador centraliza a comunicação entre componentes — o botão envia um evento ao mediador, que atualiza a caixa de texto sem que eles se comuniquem diretamente.

Padrão Mediator (Mediador)

O padrão Mediator centraliza a comunicação entre objetos, trazendo benefícios de organização e manutenção, mas exige cuidado para evitar o aumento da complexidade do sistema

Vantagens

- Reduz o acoplamento entre classes.
- Facilita a manutenção e expansão do sistema.
- Centraliza a lógica de comunicação.

Desvantagens

- O mediador pode se tornar complexo em sistemas grandes.
- Exige cuidado com interações extensas.
- Pode comprometer o desempenho se mal projetado.

Padrão Memento (LembranÇas)

Definição e Propósito

- O padrão Memento permite capturar e restaurar o estado interno de um objeto sem violar seu encapsulamento.
- Ele fornece o mecanismo para desfazer (undo) e restaurar estados anteriores de um objeto.
- É muito usado em editores de texto, jogos e sistemas que precisam registrar histórico de alterações.

[Originator] → cria → [Memento]
[Originator] ← restaura ← [Memento]
[Caretaker] → armazena → [Memento]

Captura e restaura o estado de um objeto sem violar o encapsulamento.

Padrão Memento (LembranÇas)

Estrutura e Funcionamento

Elementos principais:

- Originator: cria e restaura estados.
- Memento: armazena o estado do objeto.
- Caretaker: guarda os mementos.

Funcionamento:

- O Originator gera um Memento com seu estado.
- O Caretaker armazena o Memento.
- O Originator pode restaurar o estado salvo.

Quando aplicar:

- Para implementar undo/redo.
- Para manter histórico de estados.
- Para preservar o encapsulamento.

Padrão Memento –Código

```
// Exemplo simplificado do padrão Memento
class Editor {
   private String text = "";

   // Define o novo estado
   public void setText(String newText) { text = newText; }

   // Salva o estado atual
   public Memento save() { return new Memento(text); }

   // Restaura o estado anterior
   public void restore(Memento m) { text = m.getText(); }

   // Classe interna que armazena o estado
   static class Memento {
      private final String text;
      private Memento(String t) { text = t; }
      private String getText() { return text; }
   }
}
```

```
// Armazena os estados (histórico)
class Caretaker {
     private Stack<Editor.Memento> history = new Stack<>
();

   public void backup(Editor e) { history.push(e.save()); }
   public void undo(Editor e) {
      if (!history.isEmpty()) e.restore(history.pop());
   }
}
```

O Memento protege o encapsulamento, permitindo salvar e restaurar estados sem expor os dados internos do objeto.

Padrão Memento (LembranÇas)

Vantagens e Desvantagens

O padrão Memento facilita o controle de versões de um objeto, mas pode aumentar o consumo de memória se muitos estados forem armazenados.

Vantagens

- Preserva o encapsulamento do objeto original.
- Permite restaurar estados anteriores com facilidade.
- Implementa de forma simples o histórico de alterações.

Desvantagens

- Pode gerar alto consumo de memória com muitos estados.
- A gestão dos mementos pode se tornar complexa em sistemas grandes.
- Exige cuidados com o desempenho em operações de histórico.

Padrão Observer (Observador)

- Define uma dependência um-para-muitos entre objetos.
- Quando um objeto (Subject) muda de estado, todos os seus Observers são notificados e atualizados automaticamente.
- Permite manter consistência entre objetos relacionados sem acoplamento rígido.

Padrão Observer (Observador)

- Subject: Canal do YouTube (ex: Canal de Culinária)
- Observers: inscritos (você e outros)
- Ação: novo vídeo postado
- Notificação: todos recebem aviso automaticamente
- O canal não sabe quem você é — apenas mantém uma lista de inscritos.

Padrão Observer

- Subject: Canal do YouTube (ex: Canal de Culinária)
- Observers: inscritos (você e outros)
- Ação: novo vídeo postado
- Notificação: todos recebem aviso automaticamente
- O canal não sabe quem você é — apenas mantém uma lista de inscritos.

Padrão Observer – Componentes

- **Subject**: mantém lista de observers e notifica mudanças.
- **Observer**: interface com método update().
- **ConcreteSubject**: armazena o estado e chama notify().
- •**ConcreteObserver**: implementa update() e define ação ao ser notificado.

Padrão Observer – Código

```
import java.util.ArrayList;
impaort java.util.List;
// Interface Observer
interface Observer {
    void atualizar(String mensagem);
}
// Sujeito que será observado
class CanalNoticia {
    private List<Observer> assinantes = new ArrayList<>();
    public void adicionar(Observer o) { assinantes.add(o); }
    public void remover(Observer o) { assinantes.remove(o); }
    public void publicarNoticia(String noticia) {
        for (Observer o : assinantes) {
            o.atualizar(noticia);
        }
    }
}
```

```
// Observador concreto
class Assinante implements Observer {
    private String nome;
    public Assinante(String nome) { this.nome = nome; }
    public void atualizar(String noticia) {
        System.out.println(nome + " recebeu: " + noticia);
    }
}
// Exemplo de uso
public class Main {
    public static void main(String[] args) {
        CanalNoticia canal = new CanalNoticia();
        Observer bruna = new Assinante("Bruna");
        Observer lucas = new Assinante("Lucas");
        canal.adicionar(bruna);
        canal.adicionar(lucas);
        canal.publicarNoticia("Nova notícia publicada!");
    }
}
```

Padrão Strategy

- Encapsula uma família de algoritmos em objetos intercambiáveis.
- Permite alterar o algoritmo em tempo de execução sem mudar o código cliente.
- Útil para evitar estruturas com muitos if/else e facilitar extensões.

Padrão Strategy

- Context: aplicativo de GPS (Google Maps, Waze)
- Strategy: modo de cálculo de rota
- ConcreteStrategies:

1. Rota mais rápida
2. Rota mais curta
3. Rota a pé
4. Transporte público

- Você pode trocar a estratégia sem alterar o app.

Padrão Strategy – componentes

- Context: classe que usa o algoritmo.
- Strategy: interface comum com o método execute().
- ConcreteStrategy: implementa a lógica específica.
- Método setStrategy() troca o comportamento dinamicamente.

Padrão Strategy - código

```
interface Desconto {
   double calcular(double valor);
}
class DescontoNormal implements Desconto {
     public double calcular(double valor) { return
valor; }
}
class DescontoVip implements Desconto {
     public double calcular(double valor) { return
valor * 0.9; } // 10% de desconto
}
class DescontoEstudante implements Desconto {
     public double calcular(double valor) { return
valor * 0.8; } // 20% de desconto
}
```

```
class Loja {
 private Desconto desconto;
 public Loja(Desconto desconto) {
 this.desconto = desconto;
 }
 public void finalizarCompra(double valor) {
 System.out.println("Total a pagar: R$" + desconto.calcular(valor));
 }
}
public class Main {
 public static void main(String[] args) {
 Loja cliente1 = new Loja(new DescontoVip());
 cliente1.finalizarCompra(100);
 Loja cliente2 = new Loja(new DescontoEstudante());
 cliente2.finalizarCompra(100);
 }
}
```

Template Method

- Define o esqueleto de um algoritmo em uma classe mãe (superclasse).
- Permite que as subclasses redefinam certos passos desse algoritmo sem mudar a estrutura (a ordem) do algoritmo em si.
- Permite manter a parte invariante do algoritmo na superclasse, evitando duplicação de código.

Padrão Template Method

- Abstração (Template): Uma classe abstrata RelatorioService. Ela define o "esqueleto" do algoritmo (ex: conectarDB(), executarQuery(), processarDados(), formatarRelatorio()).
- Subclasses (Implementação): As classes concretas, como RelatorioVendasPDF e RelatorioEstoqueCSV.
- Ação: Gerar um relatório de vendas em PDF.
- Processo: O "esqueleto" (a ordem dos passos) é o mesmo, definido na classe mãe. A subclasse (RelatorioVendasPDF) apenas fornece a implementação dos passos que mudam (ex: executarQuery() faz um SELECT específico e formatarRelatorio() gera um PDF).
- Controle: A classe mãe (RelatorioService) controla o fluxo principal e chama os métodos (passos) da subclasse na ordem correta, garantindo que o algoritmo seja sempre seguido.

Template Method – Componentes

- AbstractClass (Classe Abstrata): contém o templateMethod() (o "esqueleto"), que é final e define a ordem dos passos.
- ConcreteClass (Classe Concreta): implementa (sobrescreve) os "passos primitivos" que foram definidos como abstract na classe mãe.
- Primitive Operations (Passos Primitivos): são os métodos chamados pelo template. Podem ser abstratos (obrigando a subclasse a implementar) ou "Hooks" (opcionais, com implementação padrão).

Template Method - código

```
// 1. AbstractClass (com o Template)
public abstract class PreparadorDeBebida {

    // O "Template Method" (final)
    public final void prepararBebida() {
        ferverAgua();
        prepararIngrediente(); // Passo abstrato
        colocarNaXicara();
    }

    // Passos comuns
        private void ferverAgua() { System.out.println("Fervendo
água"); }
    private void colocarNaXicara() { System.out.println("Colocando
na xícara"); }

    // Passo customizável (obrigatório)
    protected abstract void prepararIngrediente();
}

// 2. ConcreteClasses
class PreparadorDeCafe extends PreparadorDeBebida {
    @Override
    protected void prepararIngrediente() {
        System.out.println("Coando o café");
    }
}

class PreparadorDeCha extends PreparadorDeBebida {
    @Override
    protected void prepararIngrediente() {
        System.out.println("Colocando o sachê de chá");
    }
}

// 3. Exemplo de uso (Cliente)
public class Main {
    public static void main(String[] args) {
        PreparadorDeBebida cafe = new PreparadorDeCafe();
        cafe.prepararBebida(); // Chama o Template

        System.out.println("---"); // Separador

        PreparadorDeBebida cha = new PreparadorDeCha();
        cha.prepararBebida(); // Chama o mesmo Template
    }
}
```

Padrão State

- Padrão State (Estado)
- Definição: Permite que um objeto mude seu comportamento quando seu estado interno muda.
- Analogia: O objeto "parece" mudar de classe.
- Motivação Principal:
- Evitar if/else ou switch complexos baseados no estado atual.
- Organizar o código, separando cada estado em sua própria classe.

Padrão State

- Analogia: Player de Música (Play/Pause)
- Contexto: O Player de música.
- Estados Concretos: EstadoTocando, EstadoPausado.
- Ação: Clicar no botão.
- Comportamento:
- Se o estado é Pausado, o clique resulta em "Tocar" e muda o estado para Tocando.
- Se o estado é Tocando, o clique resulta em "Pausar" e muda o estado para Pausado.
- Componentes:
- Context (Player): A classe que tem um estado.
- State (Interface): Define os comportamentos (ex: pressionarPlay()).
- ConcreteState (EstadoTocando): Implementa o comportamento e a transição para o próximo estado.

Padrão State - Código

```
public class Player {
 private EstadoPlayer estadoAtual;
 // O Contexto delega a ação para o objeto de estado
 public void tocar() {
 estadoAtual.pressionarPlay(this);
}
// Permite que o estado seja trocado
public void setEstado(EstadoPlayer estado) {
 this.estadoAtual = estado;
}
}

class EstadoPausado implements EstadoPlayer {
    public void pressionarPlay(Player player) {
        System.out.println("Iniciando a música.");

        // A mágica: O próprio estado realiza a transição
        player.setEstado(new EstadoTocando());
    }
}
```

Padrão Visitor

- Padrão Visitor (Visitante)
- Definição: Permite adicionar novas operações a classes existentes sem modificá-las.
- Motivação Principal:
- Separar um algoritmo da estrutura de objetos onde ele opera.
- Evitar "poluir" as classes de elementos (ex: Circulo, Quadrado) com muitas operações (ex: calcularArea, exportarXML, imprimir).
- Quando usar? Quando as classes de elementos são estáveis, mas novas operações são frequentes.

Padrão Visitor

- Padrão Visitor: Estrutura e Analogia
- Analogia: Guia Turístico
- Elementos: Pontos turísticos (Museu, Restaurante).
- Visitors: Guias especializados (GuiaDeArquitetura, GuiaGastronomico).
- Ação: O Museu não sabe sobre gastronomia; ele apenas "aceita" um visitante (accept(Guia g)).
- Comportamento: O GuiaGastronomico sabe o que fazer ao visitar o Museu (falar do café).
- Componentes:
- Visitor (Interface): Declara métodos visit() para cada tipo de elemento (ex: visit(Circulo c)).
- Element (Interface): Declara o método accept(Visitor v).
- ConcreteVisitor: Implementa a operação (ex: CalculadorDeArea).

Padrão Visitor - Código

```
// 1. O Elemento (Circulo) aceita o visitante
class Circulo implements Forma {
public void accept(Visitante visitante) {
// ... e chama o método "visit" correto,
// passando a si mesmo (this)
visitante.visitar(this);
}
}
```

```
// 2. O Visitor tem um método para cada tipo
public interface Visitante {
void visitar(Circulo c);
void visitar(Quadrado q);
}
```

```
// 3. A operação é executada
class CalculadorDeArea implements Visitante {
public void visitar(Circulo c) {
// Lógica de área específica para Círculo
}
}
```

Obrigado !