PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Artur Monteiro Parreiras

Brenda Stefany de Oliveira Rocha

Bruna Letícia Silva

Eduarda Faria Pinheiro

Elenice Florentina de Oliveira dos Reis

Marco Aurélio Faria Ramos

Padrões de Projeto: Estrutura

Belo Horizonte
2025

0. Padrão Chain of Responsibility

O Chain of Responsibility é um padrão de projeto comportamental que permite passar uma solicitação ao longo de uma cadeia de manipuladores (handlers). Ao receber a solicitação, cada manipulador decide se processa a solicitação ou se a passa para o próximo manipulador na cadeia. A motivação é desacoplar o remetente de uma solicitação dos seus receptores. O remetente não precisa conhecer a estrutura da cadeia nem qual manipulador específico tratará sua solicitação, apenas que a solicitação será (idealmente) tratada.

Analogia da Vida Real: Caixa Automático de Banco

- **Solicitação:** O Cliente pede um saque de R$ 250.
- **Handlers (Cadeia):** O *Dispenser de R$ 100* (Handler 1) -> *Dispenser de R$ 50* (Handler 2) -> *Dispenser de R$ 20* (Handler 3).
- **Ação:** O Cliente envia a solicitação de R$ 250 para o primeiro handler (o de R$ 100).
- **Notificação (Processamento):**
    1. O *Dispenser de R$ 100* processa o que pode (entrega 2 notas, R$ 200) e **passa o resto (R$ 50)** para o próximo da cadeia.
    2. O *Dispenser de R$ 50* recebe os R$ 50, processa o que pode (entrega 1 nota, R$ 50) e **passa o resto (R$ 0)**.
    3. O *Dispenser de R$ 20* recebe R$ 0, vê que não há nada a fazer, e a solicitação termina.

Os Componentes Principais são:

- **Handler (Manipulador):**
    - Define a interface (ou classe abstrata) comum para todos os manipuladores (ex: aprovarDespesa()).
    - Mantém uma referência ao próximo Handler na cadeia (ex: protected Aprovador proximo;).
    - Fornece um método para configurar o próximo (ex: setProximo(Handler h)).
- **ConcreteHandler (Manipulador Concreto):**
    - Implementa a interface Handler.
    - Decide se pode tratar a solicitação. Se não puder, ele repassa a solicitação para o proximo, se ele existir.

- **Client (Cliente):**
    - É quem monta a cadeia (ligando os handlers) e envia a solicitação para o *início* da cadeia.
    -

Código

```
// O objeto da solicitação
public class Despesa {
   private double valor;
   public Despesa(double valor) { this.valor = valor; }
   public double getValor() { return valor; }
}

// 1. Handler Abstrato
public abstract class Aprovador {
   protected Aprovador proximo;

   public void setProximo(Aprovador proximo) {
      this.proximo = proximo;
   }

   public abstract void aprovarDespesa(Despesa despesa);
}

// 2. Concrete Handlers
class Gerente extends Aprovador {
   public void aprovarDespesa(Despesa despesa) {
      if (despesa.getValor() <= 1000) {
         System.out.println("Gerente aprovou R$" + despesa.getValor());
      } else if (proximo != null) {
         proximo.aprovarDespesa(despesa); // Passa para o próximo
      }
   }
}

class Diretor extends Aprovador {
   public void aprovarDespesa(Despesa despesa) {
      if (despesa.getValor() <= 5000) {
         System.out.println("Diretor aprovou R$" + despesa.getValor());
      } else if (proximo != null) {
         proximo.aprovarDespesa(despesa); // Passa para o próximo
      } else {
```

```
            System.out.println("Solicitação de R$" + despesa.getValor() + " não pôde ser aprovada.");
        }
    }
}

// 3. Exemplo de uso (Cliente)
public class Main {
    public static void main(String[] args) {
        // 1. Cria os handlers
        Aprovador gerente = new Gerente();
        Aprovador diretor = new Diretor();

        // 2. Monta a cadeia (gerente -> diretor)
        gerente.setProximo(diretor);

        // 3. Envia solicitações para o início da cadeia
        gerente.aprovarDespesa(new Despesa(800));  // Tratado pelo Gerente
        gerente.aprovarDespesa(new Despesa(3500)); // Tratado pelo Diretor
        gerente.aprovarDespesa(new Despesa(10000)); // Não tratado
    }
}
```

1. Padrão de comand

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.
Em resumo, o padrão Command separa quem solicita a execução de uma tarefa (o cliente) de quem realmente executa a tarefa (o receptor).

Fluxo: Client → Invoker → Command → Receiver

Exemplo:

Imagine um controle remoto que pode ligar ou desligar uma luz.

Cada botão representa um comando, e o controle em si não precisa saber *como* a luz é ligada — apenas *que deve ser ligada*.

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."
   }
```

}

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.

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.

2. Padrão de 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.

Exemplo prático (cenário)

Suponha uma linguagem simples que interpreta expressões matemáticas, como:

(5 + 10) - 3

O objetivo é fazer o programa entender e calcular o resultado.

Código explicado

```
// 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);
    }
}

// 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
    }
```

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.

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.

3. Padrão de Iterator (Iterador)

O padrão Iterator é um padrão de projeto comportamental que fornece uma maneira de acessar os elementos de um objeto agregado (uma coleção) sequencialmente, sem expor sua representação interna. A motivação é mover a responsabilidade de "como percorrer" a coleção (lógica de travessia) da própria coleção para um objeto Iterator separado. Isso permite que o código cliente percorra diferentes tipos de coleções (como Arrays, Listas, Mapas ou até mesmo resultados de banco de dados) usando a mesma interface, sem precisar saber como a coleção está estruturada por baixo dos panos.

Analogia Técnica (Mundo Real): Cursor de Banco de Dados

- **Aggregate (Agregado):** O ResultSet de um banco de dados (o conjunto de resultados de uma query).
- **Iterator (Iterador):** Um Cursor (o objeto que você recebe para "andar" pelos resultados).
- **Ação (Cliente):** O Cliente (seu código) executa uma query no banco e recebe um Cursor.
- **Notificação (Processo):** O Cliente usa métodos padronizados como cursor.hasNext() e cursor.next() para ler linha por linha. 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.

Os Componentes Principais são:

- **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.
    - Inclui um método "fábrica" para criar o iterador (ex: createIterator()).
- **ConcreteAggregate (Agregado Concreto):**
    - Implementa a interface Aggregate.
    - Sabe como instanciar e retornar o ConcreteIterator correto que sabe como percorrer *aquela* coleção específica.

Código

Java

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

// 1. Interface Iterator
public interface Iterator<T> {
    boolean hasNext();
    T next();
}

// 2. Interface Aggregate (A Coleção)
public interface Aggregate<T> {
    Iterator<T> createIterator();
}

// 3. ConcreteAggregate (A Coleção Concreta)
public class Biblioteca implements Aggregate<String> {
    private String[] livros; // Representação interna (poderia ser um List, Map, etc.)

    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;
    }

    @Override
    public boolean hasNext() {
        // Verifica se o índice está dentro dos limites do array
        return index < livros.length;
    }

    @Override
    public String next() {
        if (this.hasNext()) {
            return livros[index++];
        }
        return null; // Ou lançar uma exceção
    }
}

// 5. Exemplo de uso (Cliente)
public class Main {
    public static void main(String[] args) {
        String[] meusLivros = {"Livro A", "Livro B", "Livro C"};
        Biblioteca minhaBiblioteca = new Biblioteca(meusLivros);

        // O cliente pede o iterador para a coleção
        Iterator<String> it = minhaBiblioteca.createIterator();

        // O cliente usa o iterador, sem NUNCA saber que
        // por baixo dos panos existe um 'String[]'
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

4. Padrão Mediator (Mediador)

- Definição

O padrão Mediator propõe a criação de um objeto intermediário, denominado *mediador*, responsável por controlar e coordenar a comunicação entre diversos objetos.

Assim, elimina-se a dependência direta entre eles, reduzindo o acoplamento e melhorando a coesão do sistema.

O Mediator define um objeto que encapsula a forma como um conjunto de objetos interage, promovendo uma comunicação desacoplada e organizada.

- Estrutura e Funcionamento

A estrutura básica é composta por:

Mediator (Interface): declara as operações de comunicação entre componentes.
ConcreteMediator: implementa a lógica específica de interação.
Componentes Colegas: comunicam-se exclusivamente com o mediador.

Representação conceitual:
[ComponentA] → [Mediator] ← [ComponentB]

O *mediador* centraliza a troca de mensagens, evitando dependências diretas entre os componentes.

- Código

```
interface Mediator {
    void notify(Component sender, String event);
}
class DialogMediator implements Mediator {
    private Button button;
    private TextBox textBox;
```

```
    public void setButton(Button b) { this.button = b; }
    public void setTextBox(TextBox t) { this.textBox = t; }
    @Override
    public void notify(Component sender, String event) {
        if (event.equals("click")) {
            textBox.setText("Botão clicado!");
        }
    }
}
abstract class Component {
    protected Mediator mediator;
    public Component(Mediator mediator) { this.mediator = mediator; }
}
class Button extends Component {
    public Button(Mediator mediator) { super(mediator); }
    public void click() { mediator.notify(this, "click"); }
}
class TextBox extends Component {
    private String text;
    public TextBox(Mediator mediator) { super(mediator); }
    public void setText(String value) { text = value; }
}
```

- Aplicabilidade

O padrão Mediator é recomendado quando:

Há múltiplos objetos interdependentes, tornando o sistema difícil de manter;
Deseja-se centralizar o controle da comunicação entre componentes, como em interfaces gráficas (botões, caixas de texto, formulários);
É necessário reduzir o acoplamento entre classes que frequentemente trocam mensagens.

- Benefícios e Limitações

- Vantagens:

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

- Desvantagens:

O mediador pode se tornar complexo e sobrecarregado em sistemas com muitas interações.

5. Padrão Memento (*Lembrança*)

- Definição

O padrão Memento permite capturar e restaurar o estado interno de um objeto sem violar o encapsulamento.
Esse padrão fornece o mecanismo para implementar operações de desfazer e restaurar estados anteriores de objeto.

- Estrutura e Funcionamento

A estrutura é composta por três elementos principais:

Originator: objeto cujo estado será salvo e restaurado.
Memento: armazena o estado interno do *Originator*.
Caretaker: solicita e armazena os *mementos* sem conhecer seus detalhes internos.
Representação conceitual:

[Originator] → cria → [Memento]

[Originator] ← restaura ← [Memento]

[Caretaker] → armazena → [Memento]

- Código

```
class Editor {

    private String text = "";

    public void setText(String newText) { text = newText; }
```

```
    public String getText() { return text; }
    public Memento save() { return new Memento(text); }
    public void restore(Memento memento) { text = memento.getSavedText(); }
    static class Memento {
        private final String text;
        private Memento(String text) { this.text = text; }
        private String getSavedText() { return text; }
    }
}
class Caretaker {
    private Stack<Editor.Memento> history = new Stack<>();
    public void backup(Editor editor) { history.push(editor.save()); }
    public void undo(Editor editor) {
        if (!history.isEmpty()) editor.restore(history.pop());
    }
}
```

- Aplicabilidade

O padrão Memento é apropriado quando:

É necessário registrar o histórico de estados de um objeto (por exemplo, editores de texto ou jogos);

Deseja-se implementar operações de desfazer e refazer.
Precisa-se preservar o encapsulamento do objeto cujo estado será salvo.

- Benefícios e Limitações

- Vantagens:

Preserva o encapsulamento do objeto original.
Facilita a restauração de estados anteriores.
Implementa de forma simples o conceito de histórico de alterações.

- Desvantagens:

Pode gerar alto consumo de memória, caso muitos estados sejam armazenados.
A gestão de *mementos* pode se tornar complexa em sistemas grandes.

6. Padrão Observer (Observador)

O padrão Observer define uma dependência um-para-muitos entre objetos. Quando um objeto (o "Subject" ou "Observável") muda de estado, todos os seus dependentes (os "Observers" ou "Observadores") são notificados e atualizados automaticamente. A motivação é manter a consistência entre objetos relacionados sem criar um acoplamento rígido entre eles. O padrão permite que o objeto "Subject" não tenha conhecimento sobre as classes concretas de seus "Observers" (assinantes), apenas que eles implementam uma interface comum. O Observer deve ser utilizado quando uma mudança em um objeto exigir que outros objetos sejam alterados, mas você não quer que esses objetos sejam fortemente acoplados.

Analogia da Vida Real: Inscrição em um Canal do YouTube

- **Subject (Observável):** O Canal do YouTube (ex: "Canal de Culinária").
- **Observers (Observadores):** Você e todos os outros inscritos nesse canal.
- **Ação (Mudança de Estado):** O canal posta um vídeo novo.
- **Notificação:** O YouTube (a plataforma) envia uma notificação para todos os inscritos (Observers) sobre o vídeo novo.

O Canal de Culinária não precisa saber quem você é. Ele apenas tem uma "lista de inscritos" e um botão de "notificar todos". Você pode se inscrever (adicionar-se à lista) ou cancelar a inscrição (remover-se da lista) a qualquer momento.

Os Componentes Principais são:

2. **Subject (Sujeito):**
    - Mantém uma lista de Observers.
    - Fornece métodos para attach() (adicionar um observer) e detach() (remover um observer).
    - Possui um método notify() que percorre a lista e chama o método de atualização de cada observer.
3. **Observer (Observador):**
    - Define uma interface (ou classe abstrata) com um método update().
    - Este é o método que o Subject chamará quando houver uma mudança.

4. **ConcreteSubject:**
    - Armazena o estado que interessa aos observers.
    - Chama notify() quando seu estado muda.
5. **ConcreteObserver:**
    - Implementa a interface Observer.
    - Define o que fazer quando for notificado (o que acontece dentro do update()).

Código

```
import java.util.ArrayList;
import 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!");
    }
}
```

7. Padrão Strategy

O padrão Strategy é um padrão de projeto comportamental que encapsula uma família de algoritmos em objetos intercambiáveis, permitindo que o algoritmo varie independentemente do cliente que o utiliza. Ele é útil para lidar com diferentes variantes de um mesmo algoritmo, isolar regras de negócio e evitar estruturas de código complexas com muitos ifs. O Strategy deve ser usado quando se tem várias maneiras de fazer a *mesma coisa* (vários algoritmos) e quer permitir que o cliente ou o sistema escolha qual usar em tempo de execução, sem usar um monte de if/else ou switch/case.

Analogia da Vida Real: Aplicativo de Mapas (GPS)

- **Context (Contexto):** Aplicativo de GPS (ex: Google Maps, Waze).
- **Strategy (Estratégia):** A forma como você quer calcular a rota.
- **ConcreteStrategies (Estratégias Concretas):**
    - EstrategiaRotaMaisRapida (evita trânsito)
    - EstrategiaRotaMaisCurta (menor quilometragem)
    - EstrategiaRotaAPe (usa calçadas e parques)
    - EstrategiaRotaTransportePublico (usa ônibus e metrô)

O aplicativo (Contexto) *possui* uma referência a uma das estratégias. Quando você pede para "calcular rota", o aplicativo delega essa tarefa para a estratégia que está selecionada no momento. O mais importante: você pode trocar a estratégia em tempo de execução (mudar de "carro" para "a pé") sem mudar o aplicativo em si.

Os Componentes Principais são:

1. **Context (Contexto):**
    - A classe que precisa de um algoritmo.
    - Mantém uma referência a um objeto Strategy.
    - Não sabe os detalhes do algoritmo, apenas chama um método (ex: executeStrategy()).
    - Geralmente fornece um método setStrategy() para trocar a estratégia.
2. **Strategy (Interface da Estratégia):**
    - Define uma interface comum para todos os algoritmos suportados.
    - Declara o método que o Contexto chamará (ex: execute()).
3. **ConcreteStrategy:**
    - Implementa a interface Strategy com um algoritmo específico.
    - (Ex: PagamentoCartaoCredito, PagamentoPix, PagamentoBoleto).

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);
    }
}
```

Saída:

Total a pagar: R$90.0

Total a pagar: R$80.0

8. Padrão Strategy

O Template Method é um padrão de projeto comportamental que define o esqueleto de um algoritmo em uma superclasse, mas permite que as subclasses redefinam (sobrescrevam) certos passos desse algoritmo sem alterar a estrutura geral do algoritmo. A motivação é

permitir que subclasses customizem partes de um algoritmo sem duplicar código, movendo a parte comum para a superclasse. O padrão inverte o controle, onde a classe mãe (template) é quem chama os métodos das subclasses, e não o contrário. O Template Method deve ser utilizado para implementar as partes invariantes de um algoritmo, deixando as subclasses implementarem o comportamento que pode variar.

Analogia da Vida Real: Receita de Bolo (em um livro de receitas)

- **AbstractClass (Template):** A "Receita Mestre de Bolo". Ela define os passos fixos e a ordem deles: 1. Pré-aqueça o forno, 2. Misture ingredientes secos(), 3. Misture ingredientes molhados(), 4. Combine misturas, 5. Asse. Os passos 2 e 3 são abstratos (o *o quê* fazer).
- **ConcreteClass (Implementação):** A Receita de Bolo de Chocolate ou Receita de Bolo de Cenoura. Elas *implementam* os passos abstratos (ex: misturarIngredientesSecos() no bolo de chocolate adiciona cacau; no de cenoura, não).
- **Ação (Template Method):** Você (o Cliente) decide fazer um "Bolo de Chocolate". Você pega essa receita e segue o *mesmo esqueleto* da Receita Mestre.
- **Resultado:** O esqueleto (a ordem dos passos) é sempre o mesmo, mas o resultado (o sabor do bolo) é diferente, pois as subclasses forneceram os detalhes específicos.

Os Componentes Principais são:

- **AbstractClass (Classe Abstrata):**
    - Contém o **templateMethod()** (o "esqueleto"), que é final (não pode ser sobrescrito).
    - Este método chama os "passos primitivos" na ordem correta.
    - Define os "passos primitivos" que as subclasses devem implementar (métodos abstract) ou podem implementar (métodos "hooks", ou ganchos, que têm uma implementação padrão).
- **ConcreteClass (Classe Concreta):**
    - Implementa a AbstractClass.
    - Ela sobrescreve os "passos primitivos" (os métodos abstratos) para fornecer a implementação específica de cada passo do algoritmo.

Código

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

    // O "Template Method" (final = esqueleto é imutável)
    public final void prepararBebida() {
        ferverAgua();
        prepararIngrediente(); // Passo abstrato
        colocarNaXicara();
        adicionarCondimentos(); // "Hook" (opcional)
    }

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

    // Passos customizáveis (abstratos ou hooks)
    protected abstract void prepararIngrediente();

    // "Hook" (Gancho) - Um método opcional para as subclasses
    protected void adicionarCondimentos() {
        // Implementação padrão vazia
    }
}

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

    // Sobrescreve o "hook"
    @Override
    protected void adicionarCondimentos() {
        System.out.println("Adicionando açúcar e leite");
    }
}

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

```
    // Não sobrescreve o "hook" (adicionarCondimentos)
}

// 3. Exemplo de uso (Cliente)
public class Main {
    public static void main(String[] args) {
        System.out.println("--- Preparando Café ---");
        PreparadorDeBebida cafe = new PreparadorDeCafe();
        cafe.prepararBebida(); // Chama o Template Method

        System.out.println("\n--- Preparando Chá ---");
        PreparadorDeBebida cha = new PreparadorDeCha();
        cha.prepararBebida(); // Chama o mesmo Template Method
    }
}
```

9. Padrão State

Padrão State (Estado) O padrão State é um padrão de projeto comportamental que permite a um objeto alterar seu comportamento quando seu estado interno muda. O objeto parece mudar de classe. A motivação é organizar o código que lida com diferentes estados de um objeto em classes separadas. Em vez de o objeto principal (Contexto) ter um grande if/else ou switch para verificar seu próprio estado e decidir o que fazer, ele delega a responsabilidade de comportamento para um objeto de "estado" separado. O padrão State deve ser usado quando um objeto tem um comportamento complexo que depende de seu estado, e esse estado muda em tempo de execução.

Analogia da Vida Real:

Player de Música Context: O aplicativo de música (Player). State: A interface EstadoPlayer. ConcreteStates: O EstadoTocando e o EstadoPausado. Ação: O usuário clica no botão principal. Notificação:

1. O Player está no EstadoPausado. O usuário clica no botão. O Player delega a ação para o EstadoPausado, que executa "tocar" e **muda o estado do Player** para EstadoTocando.

2. O Player está agora no EstadoTocando. O usuário clica no mesmo botão. O Player delega a ação para o EstadoTocando, que executa "pausar" e **muda o estado do Player** de volta para EstadoPausado.

O Player em si não sabe o que fazer; ele apenas pergunta ao seu estado atual.

Os Componentes Principais são:

- **Context (Contexto):**
  - A classe que possui um estado (ex: Player).
  - Mantém uma referência para o objeto State atual.
  - Fornece um método para trocar o estado (ex: setEstado(EstadoNovo)).
  - Delega as solicitações de comportamento para o objeto de estado atual.
- **State (Interface ou Classe Abstrata de Estado):**
  - Define a interface comum para todos os estados concretos (ex: tocar(), pausar()).
- **ConcreteState (Estado Concreto):**
  - Implementa a interface State (ex: EstadoTocando, EstadoPausado).
  - Cada estado implementa o comportamento específico para quando o Contexto está nesse estado.
  - É responsável por trocar o estado do Contexto quando uma transição deve ocorrer.

Código

Java

```
// 1. State (Interface de Estado)
public interface EstadoPlayer {
   void pressionarPlay(Player player);
   void pressionarPause(Player player);
}

// 2. ConcreteStates
class EstadoTocando implements EstadoPlayer {
   public void pressionarPlay(Player player) {
      System.out.println("Já está tocando.");
```

```
    }

    public void pressionarPause(Player player) {
        System.out.println("Pausando a música.");
        // Transição de estado
        player.setEstado(new EstadoPausado());
    }
}

class EstadoPausado implements EstadoPlayer {
    public void pressionarPlay(Player player) {
        System.out.println("Iniciando a música.");
        // Transição de estado
        player.setEstado(new EstadoTocando());
    }

    public void pressionarPause(Player player) {
        System.out.println("Já está pausado.");
    }
}

// 3. Context (Contexto)
public class Player {
    private EstadoPlayer estadoAtual;

    public Player() {
        // Estado inicial
        this.estadoAtual = new EstadoPausado();
    }

    public void setEstado(EstadoPlayer estado) {
        this.estadoAtual = estado;
    }

    // O contexto delega o comportamento para o estado atual
    public void tocar() {
        estadoAtual.pressionarPlay(this);
```

```
    }

    public void pausar() {
        estadoAtual.pressionarPause(this);
    }
}

// 4. Exemplo de uso (Cliente)
public class Main {
    public static void main(String[] args) {
        Player player = new Player();

        player.tocar();   // Saída: Iniciando a música.
        player.tocar();   // Saída: Já está tocando.
        player.pausar();  // Saída: Pausando a música.
        player.pausar();  // Saída: Já está pausado.
        player.tocar();   // Saída: Iniciando a música.
    }
}
```

10. Padrão Visitor

Padrão Visitor (Visitante) O padrão Visitor é um padrão de projeto comportamental que permite adicionar novos comportamentos (operações) a um conjunto de classes (elementos) sem modificar essas classes. O padrão funciona "visitando" cada elemento de uma estrutura de objetos e executando uma operação específica para o tipo daquele elemento. A motivação é separar um algoritmo da estrutura de objetos em que ele opera. O Visitor deve ser usado quando você tem uma estrutura de objetos estável (as classes raramente mudam), mas precisa executar operações diferentes e complexas sobre essa estrutura (novas operações são adicionadas com frequência).

Analogia da Vida Real:

Guia Turístico Especializado ObjectStructure: Uma cidade com vários pontos turísticos (ex: Museu, Parque, Restaurante). Essas classes (Museu, Parque) são estáveis. Element: A

interface PontoTuristico. Visitor: A interface Visitante (um guia). ConcreteVisitors: GuiaDeArquitetura, GuiaGastronomico. Ação: Um guia (Visitor) decide "visitar" a cidade (ObjectStructure). Notificação:

- O GuiaDeArquitetura visita o Museu e fala sobre "o design da fachada".
- O GuiaGastronomico visita o Museu e fala sobre "a qualidade do café".
- O GuiaDeArquitetura visita o Restaurante e fala sobre "a estrutura do prédio".
- O GuiaGastronomico visita o Restaurante e fala sobre "o prato do dia".

O Museu não precisa saber sobre gastronomia ou arquitetura. Ele apenas "aceita" um visitante e deixa o visitante fazer seu trabalho.

Os Componentes Principais são:

- **Visitor (Interface do Visitante):**
    - Declara um método visit() para cada classe ConcreteElement na estrutura (ex: visit(Circulo c), visit(Quadrado q)).
- **ConcreteVisitor (Visitante Concreto):**
    - Implementa a interface Visitor.
    - Cada ConcreteVisitor representa uma operação diferente a ser executada nos elementos.
- **Element (Interface do Elemento):**
    - Define um método accept(Visitor v). Este é o ponto de entrada para o visitante.
- **ConcreteElement (Elemento Concreto):**
    - Implementa a interface Element.
    - Seu método accept() geralmente chama o método visit() do visitante, passando a si mesmo como argumento (ex: visitor.visit(this)).
- **ObjectStructure (Estrutura de Objetos):**
    - Uma coleção de objetos Element.
    - Fornece uma maneira de iterar sobre seus elementos e aplicar o Visitor a cada um.

Código

Java

```
// 1. Element (Interface do Elemento)
```

```
public interface Forma {
    void accept(Visitante visitante);
}

// 2. ConcreteElements
class Circulo implements Forma {
    private double raio;
    public Circulo(double raio) { this.raio = raio; }
    public double getRaio() { return raio; }

    @Override
    public void accept(Visitante visitante) {
        visitante.visitar(this); // Chama o método específico para Circulo
    }
}

class Quadrado implements Forma {
    private double lado;
    public Quadrado(double lado) { this.lado = lado; }
    public double getLado() { return lado; }

    @Override
    public void accept(Visitante visitante) {
        visitante.visitar(this); // Chama o método específico para Quadrado
    }
}

// 3. Visitor (Interface do Visitante)
public interface Visitante {
    void visitar(Circulo c);
    void visitar(Quadrado q);
}

// 4. ConcreteVisitors
class CalculadorDeArea implements Visitante {
    private double areaTotal = 0;
```

```
    public void visitar(Circulo c) {
        areaTotal += 3.14159 * c.getRaio() * c.getRaio();
    }

    public void visitar(Quadrado q) {
        areaTotal += q.getLado() * q.getLado();
    }

    public double getAreaTotal() { return areaTotal; }
}

class ExportadorXML implements Visitante {
    private StringBuilder xml = new StringBuilder();

    public void visitar(Circulo c) {
        xml.append("<circulo raio=\"" + c.getRaio() + "\"/>\n");
    }

    public void visitar(Quadrado q) {
        xml.append("<quadrado lado=\"" + q.getLado() + "\"/>\n");
    }

    public String getXml() { return xml.toString(); }
}

// 5. Exemplo de uso (Cliente e ObjectStructure)
public class Main {
    public static void main(String[] args) {
        // 5. ObjectStructure (uma lista de formas)
        Forma[] formas = { new Circulo(10), new Quadrado(5), new Circulo(2) };

        // Aplicando o primeiro Visitor (CalculadorDeArea)
        CalculadorDeArea calculador = new CalculadorDeArea();
        for (Forma f : formas) {
            f.accept(calculador);
        }
        System.out.println("Área Total: " + calculador.getAreaTotal());
```

```
        // Aplicando o segundo Visitor (ExportadorXML)
        ExportadorXML exportador = new ExportadorXML();
        for (Forma f : formas) {
            f.accept(exportador);
        }
        System.out.println("\nExportação XML:\n" + exportador.getXml());
    }
}
```