Engenharia de Softwares Escaláveis

Design Patterns e Domain-Driven Design com Java

Agenda

Etapa 2: Desenvolver Software Aplicando Design Patterns.

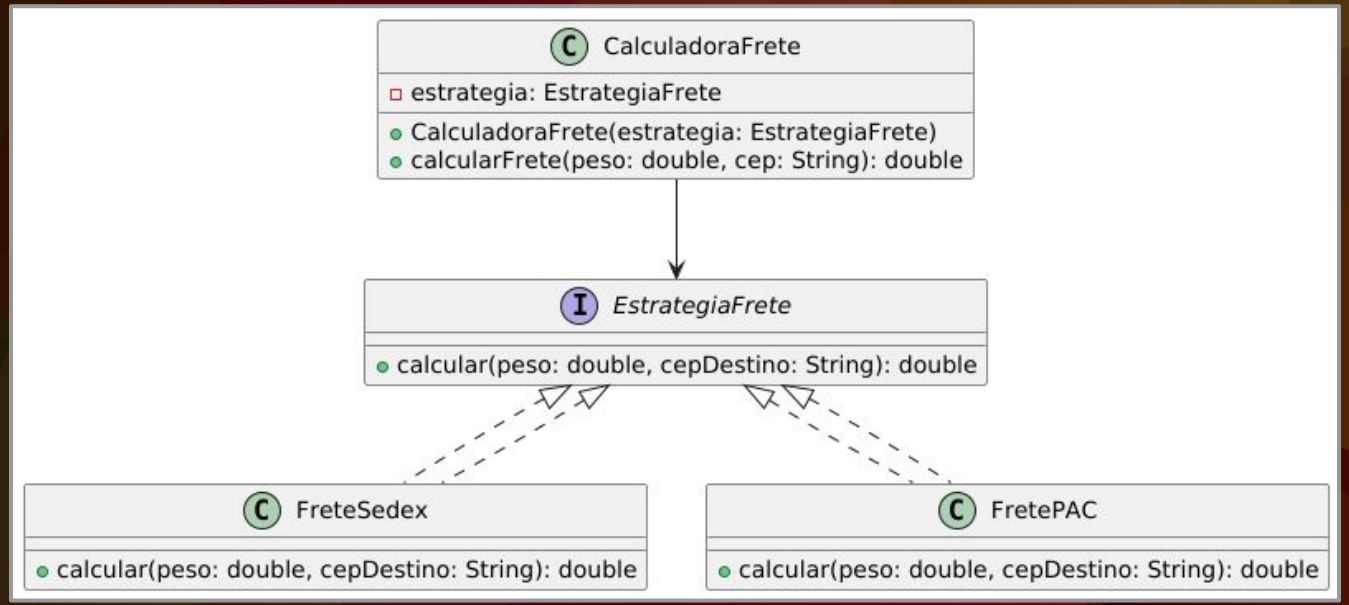
- Padrões Comportamentais.

Padrões Comportamentais

Strategy

O Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.

Permite encapsular comportamentos ou algoritmos em classes separadas chamadas estratégias para que seja possível alterar dinamicamente o comportamento sem mudar o código.



```java
public interface EstrategiaFrete {
    double calcular(double peso, String cepDestino);
}

public class FreteSedex implements EstrategiaFrete {
    public double calcular(double peso, String cepDestino) {
        return peso * 12.0;
    }
}

public class CalculadoraFrete {
    private EstrategiaFrete estrategia;

    public CalculadoraFrete(EstrategiaFrete estrategia) {
        this.estrategia = estrategia;
    }

    public double calcularFrete(double peso, String cep) {
        return estrategia.calcular(peso, cep);
    }
}
```

```java
@Test
public void testTrocaDeEstrategia() {
    CalculadoraFrete calculadora = new CalculadoraFrete(new FreteSedex());
    double valor1 = calculadora.calcularFrete(1.0, "12345-678");
    assertEquals(12.0, valor1, 0.01);

    calculadora = new CalculadoraFrete(new FretePAC());
    double valor2 = calculadora.calcularFrete(1.0, "12345-678");
    assertEquals(8.0, valor2, 0.01);
}
```

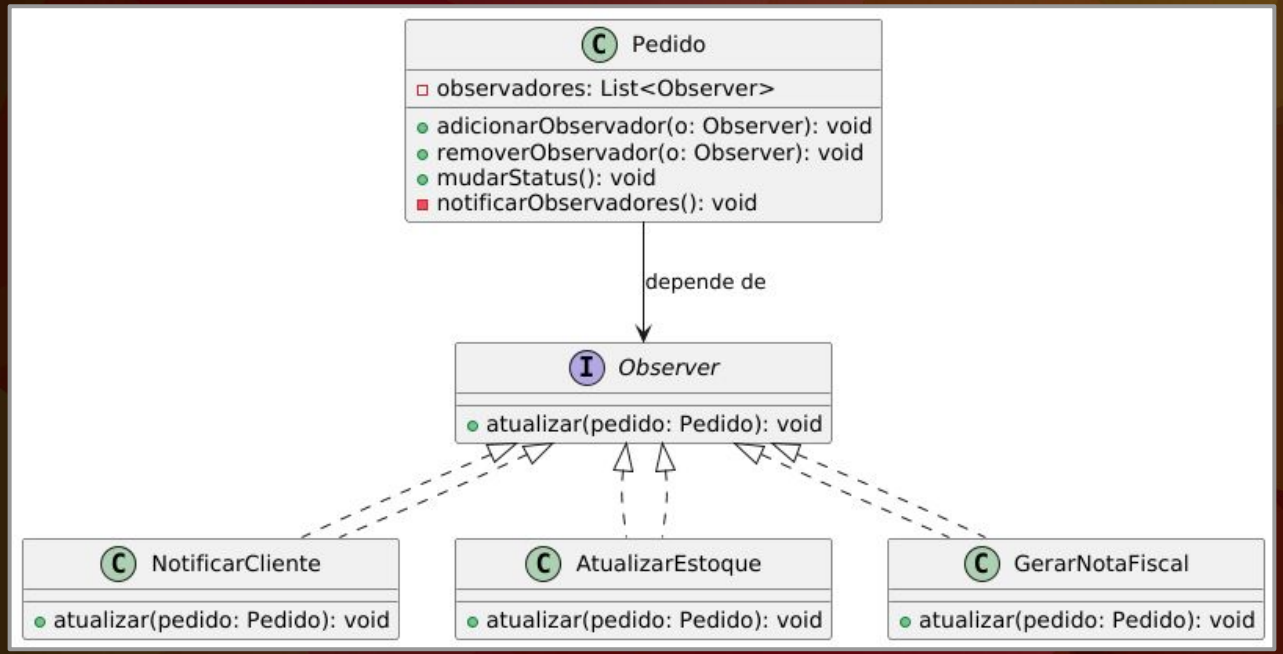
OCP O Strategy permite adicionar novos comportamentos (estratégias) sem modificar o código existente.

DIP A CalculadoraFrete depende da abstração EstrategiaFrete, e não das classes concretas (FreteSedex, FretePAC, etc.), permitindo que o código principal funcione com qualquer implementação da interface.

Observer

Um objeto sujeito (Subject) mantém uma lista de observadores (Observers) que ao mudar de estado, notifica automaticamente todos os observadores registrados.

O objetivo é desacoplar o emissor de eventos dos receptores, promovendo um design mais flexível.



```java
public interface Observer {
    void atualizar();
}

public class Pedido {
    private List<Observer> observers = new ArrayList<>();

    public void adicionarObserver(Observer o) {
        observers.add(o);
    }

    public void mudarStatus() {
        // Lógica do pedido...
        notificarObservers();
    }

    private void notificarObservers() {
        for (Observer o : observers) {
            o.atualizar();
        }
    }
}
```

```
@Test
void deveNotificarObserversQuandoStatusMudar() {
     Pedido pedido = new Pedido();
     TestObserver observador1 = new TestObserver();
     TestObserver observador2 = new TestObserver();
     pedido.adicionarObservador(observador1);
     pedido.adicionarObservador(observador2);
     pedido.mudarStatus("ENVIADO");
     assertTrue(observador1.foiNotificado());
     assertTrue(observador2.foiNotificado());
     assertEquals("ENVIADO", observador1.getStatusRecebido());
     assertEquals("ENVIADO", observador2.getStatusRecebido());
}
```

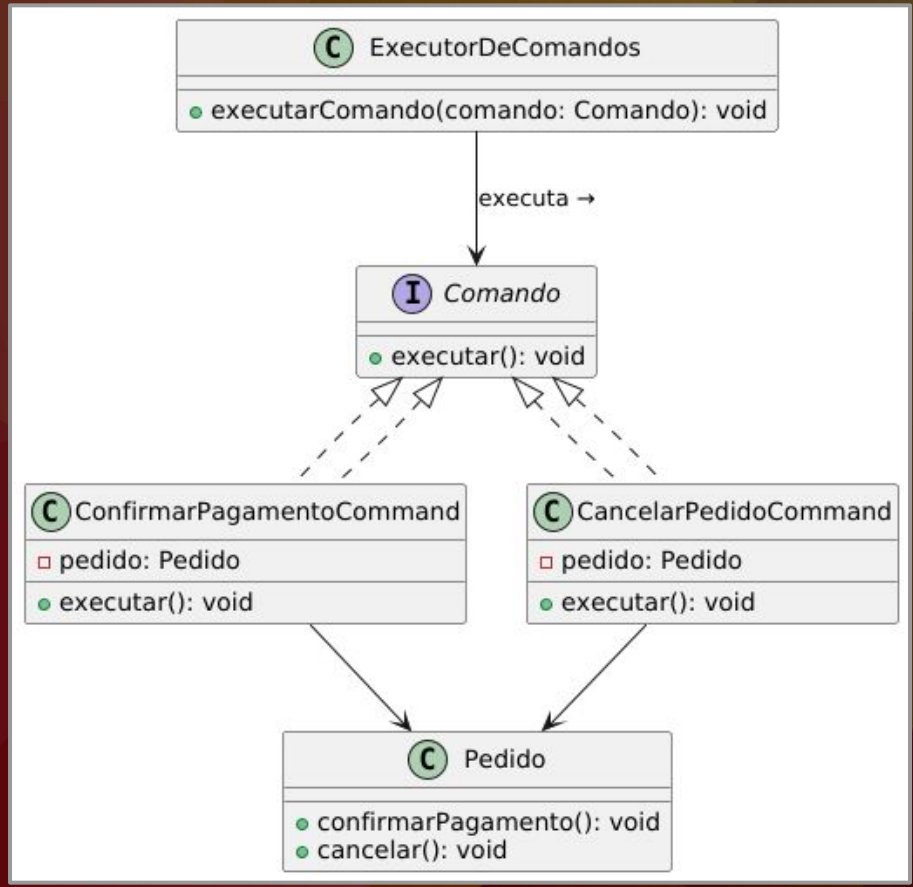
SRP O Subject (ou emissor de eventos) cuida apenas da lógica de seu próprio estado e da mecânica de notificação e os Observers cuidam de como reagir à notificação, cada um de forma independente.

OCP O Subject não conhece as implementações concretas dos observers, pois depende de uma abstração (interface Observer) que pode ser implementada por qualquer classe.

Command

Encapsular uma solicitação como um objeto, desta forma permitindo que você parametrize clientes com diferentes solicitações, enfileire ou registre solicitações e suporte operações de desfazer (undo).

Você pode criar uma interface Comando, implementar cada ação como um ConcreteCommand e deixar um Invoker (como um agendador ou botão na interface) executar o comando.



```java
public class Pedido {
    private final String id;
    private boolean pagamentoConfirmado;
    private boolean cancelado;

    public void confirmarPagamento() {
        this.pagamentoConfirmado = true;
    }

    public void cancelar() {
        this.cancelado = true;
    }

    public boolean isPagamentoConfirmado() {
        return pagamentoConfirmado;
    }

    public boolean isCancelado() {
        return cancelado;
    }
}
```

1

```java
public interface Comando {
    void executar();
}

public class ConfirmarPagamentoCommand implements Comando {
    private final Pedido pedido;

    public ConfirmarPagamentoCommand(Pedido pedido) {
        this.pedido = pedido;
    }

    @Override
    public void executar() {
        pedido.confirmarPagamento();
    }
}
```

2

```
public class CancelarPedidoCommand implements Comando {
    private final Pedido pedido;

    public CancelarPedidoCommand(Pedido pedido) {
        this.pedido = pedido;
    }

    @Override
    public void executar() {
        pedido.cancelar();
    }
}

public class ExecutorDeComandos {
    public void executarComando(Comando comando) {
        comando.executar();
    }
}
```

3

```java
public class CommandTest {

    private Pedido pedido;
    private ExecutorDeComandos executor;

    @BeforeEach
    void setup() {
        pedido = new Pedido("123");
        executor = new ExecutorDeComandos();
    }

    @Test
    void deveConfirmarPagamentoDoPedido() {
        Comando confirmar = new
ConfirmarPagamentoCommand(pedido);
        executor.executarComando(confirmar);

        assertTrue(pedido.isPagamentoConfirmado());
    }
}
```

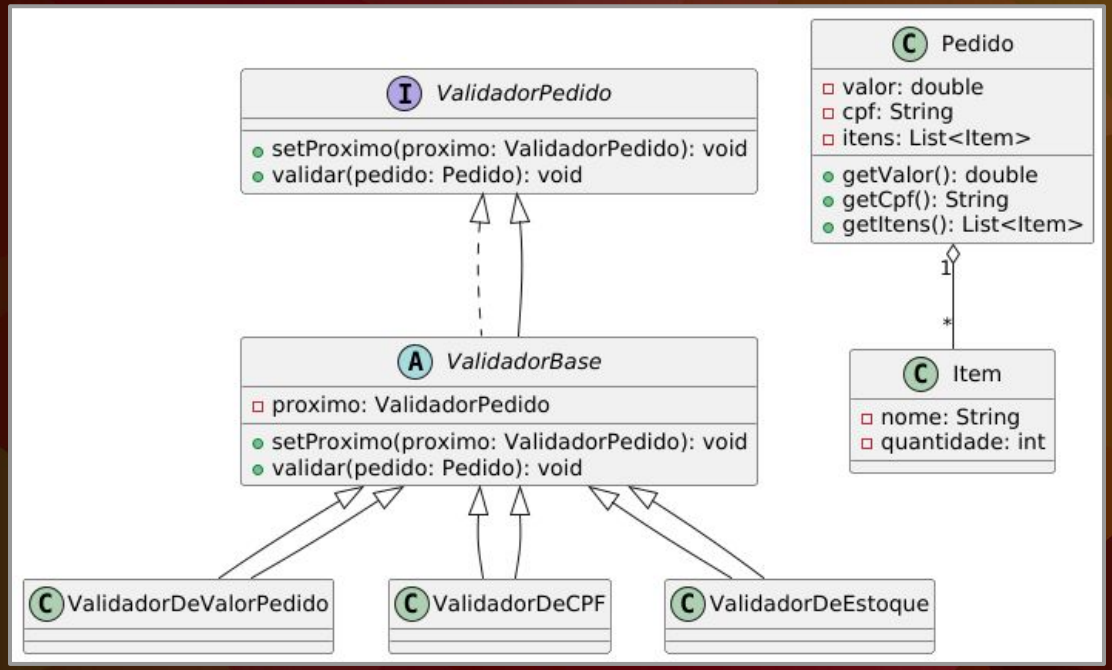
SRP Cada classe ConcreteCommand encapsula uma única ação específica, como EnviarPedidoCommand ou CancelarPedidoCommand. O Invoker não precisa saber o que o comando faz nem como ele funciona.

OCP Você pode adicionar novos comandos implementando a interface Command sem modificar o código existente do Invoker ou do Client.

Chain of Responsibility

Permite que mais de um objeto tenha a chance de tratar uma solicitação, evitando o acoplamento entre o remetente e o receptor da solicitação.

Os objetos são encadeados e a solicitação passa pela cadeia até que algum objeto a processe.



```java
public class Pedido {
    private double valor;
    private String cpfCliente;
    private List<Item> itens;

    public Pedido(double valor, String cpfCliente, List<Item> itens) {
        this.valor = valor;
        this.cpfCliente = cpfCliente;
        this.itens = itens;
    }

    public double getValor() {
        return valor;
    }

    public String getCpfCliente() {
        return cpfCliente;
    }

    public List<Item> getItens() {
        return itens;
    }
}
```

1

```java
public class Item {
    private String nome;
    private int quantidade;

    public Item(String nome, int quantidade) {
        this.nome = nome;
        this.quantidade = quantidade;
    }

    public String getNome() {
        return nome;
    }

    public int getQuantidade() {
        return quantidade;
    }
}
```

2

```java
public interface ValidadorPedido {
    void setProximo(ValidadorPedido proximo);
    void validar(Pedido pedido);
}

public abstract class ValidadorBase implements ValidadorPedido {
    protected ValidadorPedido proximo;

    @Override
    public void setProximo(ValidadorPedido proximo) {
        this.proximo = proximo;
    }

    @Override
    public void validar(Pedido pedido) {
        executarValidacao(pedido);
        if (proximo != null) {
            proximo.validar(pedido);
        }
    }

    protected abstract void executarValidacao(Pedido pedido);
}
```

3

```java
public class ValidadorDeValorPedido extends ValidadorBase {
    @Override
    protected void executarValidacao(Pedido pedido) {
        if (pedido.getValor() <= 0) {
            throw new RuntimeException("Erro: O valor do pedido inválido.");
        }
    }
}
```

4

```java
public class ValidadorDeCPF extends ValidadorBase {
    @Override
    protected void executarValidacao(Pedido pedido) {
        if (!pedido.getCpfCliente().matches("\\d{11}") ||
            pedido.getCpfCliente().equals("99999999999")) {
            throw new RuntimeException("Erro: CPF do cliente é inválido.");
        }
    }
}
```

```java
@Test
public void deveValidarPedidoComSucesso() {
    Pedido pedido = new Pedido(150.0, "12345678900",
                               List.of(new Item("Notebook", 1)));
    ValidadorPedido validadorValor = new ValidadorDeValorPedido();
    ValidadorPedido validadorCpf = new ValidadorDeCPF();
    ValidadorPedido validadorEstoque = new ValidadorDeEstoque();
    validadorValor.setProximo(validadorCpf);
    validadorCpf.setProximo(validadorEstoque);
    assertDoesNotThrow(() -> validadorValor.validar(pedido));
}
```

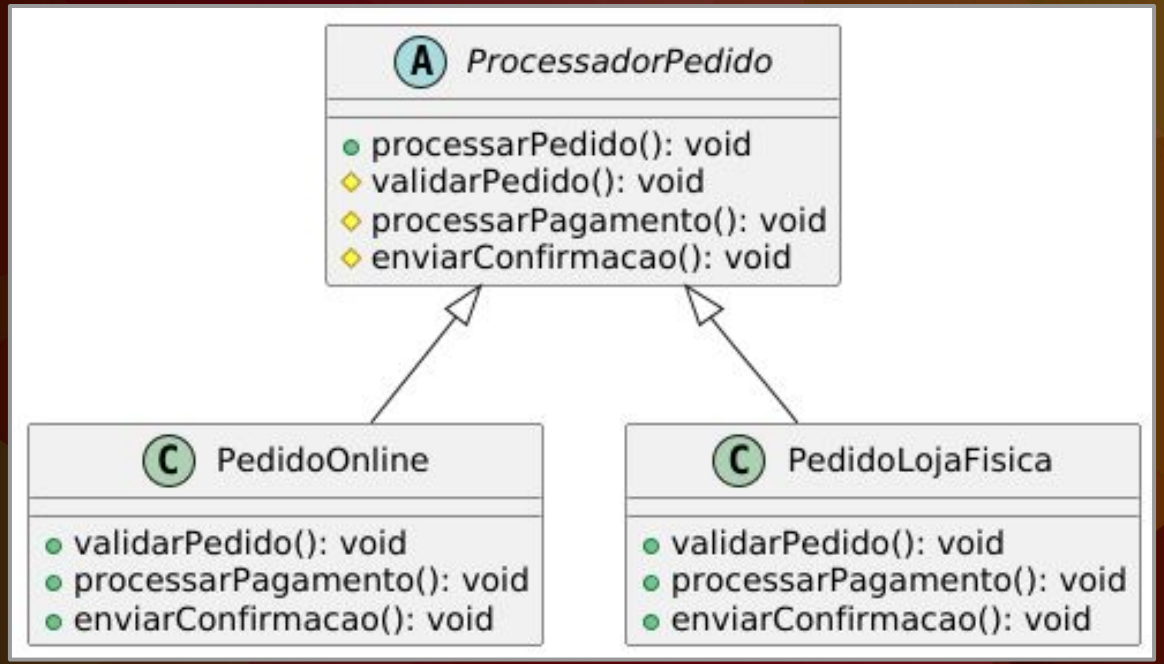
SRP Cada handler (tratador) da cadeia tem uma única responsabilidade: validar um aspecto específico de uma solicitação.

OCP A cadeia pode ser estendida com novos validadores sem alterar os existentes: não é preciso modificar os validadores já implementados para incluir novas regras.

Template Method

É a implementação de um modelo de algoritmo em uma operação, adiando algumas etapas para subclasses.

O modelo permite que subclasses redefinam certas etapas de um algoritmo sem alterar sua estrutura.



```java
public abstract class ProcessadorDePedido {

    public final void processar() {
        validarDados();
        verificarEstoque();
        processarPagamento();
        emitirNotaFiscal();
    }

    protected abstract void validarDados();
    protected abstract void verificarEstoque();
    protected abstract void processarPagamento();

    protected void emitirNotaFiscal() {
        System.out.println("Nota fiscal emitida.");
    }
}
```

1

2

```
public class PedidoOnline extends ProcessadorDePedido {
    protected void validarDados() {
        System.out.println("Validando dados online...");
    }
    protected void verificarEstoque() {
        System.out.println("Checando estoque online...");
    }
    protected void processarPagamento() {
        System.out.println("Processando cartão de crédito...");
    }
}
```

```
@Test
void testPedidoOnline() {
     ByteArrayOutputStream output = new ByteArrayOutputStream();
     System.setOut(new PrintStream(output));
     ProcessadorPedido pedido = new PedidoOnline();
     pedido.processarPedido();
     String result = output.toString();
     assertTrue(result.contains("Validando pedido online"));
     assertTrue(result.contains("Processando pagamento via cartão"));
     assertTrue(result.contains("Enviando confirmação por e-mail"));
}
```

SRP A classe base define a estrutura fixa de um algoritmo (o template), concentrando-se apenas no fluxo genérico da operação. As subclasses são responsáveis por detalhar comportamentos específicos (implementações dos métodos abstratos ou hooks): com isso, cada classe tem uma única responsabilidade.

OCP A classe base não precisa ser modificada quando novas variações de comportamento são necessárias. Isso evita que o algoritmo geral seja alterado (mantendo a classe base fechada para modificação) e permite a criação de novos comportamentos (aberta para extensão).