Instituto Federal Catarinense *(Campus Blumenau)*

Professor: Ricardo de La Rocha Ladeira

Matéria: Padrões de Projeto

Nomes: Gabrielli Danker

Turma: BCC 2025.1

Data de entrega: 27 de Fevereiro de 2025

Exercícios

1. (FGV – SEFAZ-AM 2022 – Analista de Tecnologia da Informação da Fazenda Estadual) Os padrões GRASP descrevem princípios fundamentais de atribuição de responsabilidades em projetos de software orientado a objeto. De acordo com esses padrões, o polimorfismo é um princípio orientador para atribuir responsabilidades
   1. a classes artificiais que não representem nada no domínio do problema de maneira a obter uma coesão alta.
   2. a classes abstratas que representem o comportamento de classes concretas para permitir ao sistema lidar com vários tipos de maneira homogênea.
   3. a uma classe controladora global para centralizar o tratamento de todas as mensagens e eventos do sistema.
   4. a classes que têm a informação necessária para satisfazer à responsabilidade e colaborar para que o acoplamento permaneça alto.
   5. a classes estáticas, altamente coesas e que são utilizadas no sistema sem a necessidade de instanciar objetos.

RESPOSTA: Letra B) a classes abstratas que representem o comportamento de classes concretas para permitir ao sistema lidar com vários tipos de maneira homogênea.

1. (CESPE – SEDF 2017 – Analista de Gestão Educacional – Tecnologia da Informação) Julgue a seguinte frase a respeito de padrões de projetos: *No padrão GRASP, a alta coesão (high cohesion) serve para mensurar quão fortemente uma classe está conectada a outras classes*. Certo ou Errado?

RESPOSTA: Errado.

1. Comente o código [TestaRelatoriosDePagamento.java](https://drive.google.com/file/d/1-Nrc9unL0K7CCDXO4QuL4zd1N_VAyQ1W/view?usp=drive_link) e indique onde e quais padrões GRASP estão sendo utilizados no código.

RESPOSTA:

// Interface Relatorio aplicando o padrão Polymorphism (Polimorfismo)

// Permite que diferentes classes implementem a mesma interface de forma independente.

interface Relatorio {

void gerarRelatorio();

}

// Classe RelatorioPagamento implemetando a interface Relatorio

// Aplicando Polymorphism e High Cohesion (Alta Coesão)

// Esta classe tem a única responsabilidade de gerar relatórios de pagamento.

class RelatorioPagamento implements Relatorio {

private Pagamento pagamento;

// Aplicando Creator (Criador)

// A classe recebe um objeto Pagamento para trabalhar com ele.

public RelatorioPagamento(Pagamento pagamento) {

this.pagamento = pagamento;

}

@Override

public void gerarRelatorio() {

System.out.println("Gerando relatório de pagamento...");

System.out.println("Pagamento de " + pagamento.getValor() + " realizado.");

}

}

// Classe Pagemneto aplicando High Cohesion

// Responsável apenas por armazenar e fornecer o valor do pagamento.

class Pagamento {

private double valor;

public Pagamento(double valor) {

this.valor = valor;

}

public double getValor() {

return valor;

}

}

// Classe GerenciadorDeLog aplicando High Cohesion

// Tem a única responsabilidade de registrar logs do sistema.

class GerenciadorDeLog {

public void logar(String mensagem) {

System.out.println("Log: " + mensagem);

}

}

// Classe ControladorPagamento aplicando Controller e Low Coupling (Baixo Aclopamento)

// Atua como um intermediário que gerencia a lógica de pagamento e usa injeção de dependência.

class ControladorPagamento {

private GerenciadorDeLog log;

private Relatorio relatorio;

public ControladorPagamento(GerenciadorDeLog log, Relatorio relatorio) {

this.log = log;

this.relatorio = relatorio;

}

// Método processarPagamento aplicando Controller

public void processarPagamento(Pagamento pagamento) {

log.logar("Iniciando o processamento de pagamento...");

relatorio.gerarRelatorio();

log.logar("Pagamento processado com sucesso!");

}

}

// Classe principal TesteRelatorioDePagamento aplicando Creator (Criador)

// Responsável por criar as instâncias e e iniciar o processo.

public class TestaRelatoriosDePagamento {

public static void main(String[] a) {

// Criando as intâncias necessárias

Pagamento pagamento = new Pagamento(500.00);

Relatorio relatorio = new RelatorioPagamento(pagamento);

GerenciadorDeLog log = new GerenciadorDeLog();

ControladorPagamento controlador = new ControladorPagamento(log, relatorio);

// Iniciando o processamento do pagamento.

controlador.processarPagamento(pagamento);

}

}

1. Desenvolva um código orientado a objetos que utilize no mínimo cinco princípios GRASP. Comente o código e deixe claro onde os princípios estão sendo aplicados.

RESPOSTA:

// 1. Classe Abstrata para Pagamento (Polimorfismo e Alta Coesão)

abstract class Pagamento {

public abstract void processarPagamento(double valor);

}

// 2. Classe que representa o pagamento por Cartão de Crédito

class PagamentoCartaoCredito extends Pagamento {

@Override

public void processarPagamento(double valor) {

System.out.println("Pagamento de R$ " + valor + " realizado com Cartão de Crédito.");

}

}

// 3. Classe que representa o pagamento por Dinheiro

class PagamentoDinheiro extends Pagamento {

@Override

public void processarPagamento(double valor) {

System.out.println("Pagamento de R$ " + valor + " realizado com Dinheiro.");

}

}

// 4. Classe que representa o pagamento por Pix

class PagamentoPix extends Pagamento {

@Override

public void processarPagamento(double valor) {

System.out.println("Pagamento de R$ " + valor + " realizado com Pix.");

}

}

// 5. Classe Pedido (Controller)

class Pedido {

private double valorTotal;

private Pagamento pagamento;

public Pedido(double valorTotal) {

this.valorTotal = valorTotal;

}

// 6. Método para definir o método de pagamento (Controller)

public void definirPagamento(Pagamento pagamento) {

this.pagamento = pagamento;

}

// 7. Método para processar o pagamento (Information Expert)

public void finalizarPedido() {

if (pagamento != null) {

pagamento.processarPagamento(valorTotal);

} else {

System.out.println("Método de pagamento não definido.");

}

}

}

public class SistemaPagamento {

public static void main(String[] args) {

// Criando um pedido de R$ 100,00

Pedido pedido = new Pedido(100.0);

// Definindo o pagamento como Cartão de Crédito

pedido.definirPagamento(new PagamentoCartaoCredito());

pedido.finalizarPedido(); // Processa o pagamento com Cartão de Crédito

// Criando outro pedido de R$ 50,00

Pedido pedido2 = new Pedido(50.0);

// Definindo o pagamento como Dinheiro

pedido2.definirPagamento(new PagamentoDinheiro());

pedido2.finalizarPedido(); // Processa o pagamento com Dinheiro

// Criando outro pedido de R$ 20,00

Pedido pedido3 = new Pedido(20.0);

// Definindo o pagamento como Pix

pedido3.definirPagamento(new PagamentoPix());

pedido3.finalizarPedido(); // Processa o pagamento com Pix

}

}

1. Altere o exemplo 2 para
   1. permitir um novo tipo de desconto: o Desconto VIP, que deve permitir descontos de 40%.

RESPOSTA:

class CalculadoraDeDesconto {

public double calcularDesconto(String tipo, double valor) {

if (tipo.equals("estudante"))

return valor \* 0.2;

else if (tipo.equals("idoso"))

return valor \* 0.3;

else if (tipo.equals("VIP"))

return valor \* 0.4;

else

return valor \* 0.1;

}

}

public class TestaCalculadora {

public static void main(String[] args) {

CalculadoraDeDesconto calc = new CalculadoraDeDesconto();

System.out.println("Desconto estudante: " + calc.calcularDesconto("estudante", 100.0));

System.out.println("Desconto idoso: " + calc.calcularDesconto("idoso", 100.0));

System.out.println("Desconto VIP: " + calc.calcularDesconto("VIP", 100.0));

System.out.println("Desconto padrão: " + calc.calcularDesconto("padrão", 100.0));

}

}

* 1. extinguir a superclasse Desconto, de forma a criar um novo tipo (desconto padrão). Há vantagens e/ou desvantagens ao aplicar esta alteração?

RESPOSTA:

#### **Vantagens:**

* **Simplicidade**: O código fica mais simples, sem a necessidade de lidar com herança. Você não precisa mais de classes adicionais como uma superclasse Desconto ou subclasses.
* **Facilidade de manutenção**: Se o código de cálculo de descontos não precisa ser ampliado com outras funcionalidades, a abordagem direta no método calcularDesconto pode ser mais fácil de entender e modificar.

#### **Desvantagens:**

* **Escalabilidade**: Se você precisar adicionar mais tipos de desconto no futuro ou modificar regras de descontos para tipos diferentes de forma independente, pode ser mais difícil gerenciar tudo dentro de um único método, sem a abstração fornecida por uma superclasse. Usar a herança pode tornar o código mais flexível para mudanças ou extensões futuras.