**Projeto de Desenvolvimento de Sistema Bancário com Paradigma Orientado a Aspectos**

**Introdução**

Este projeto visa implementar um sistema bancário utilizando os paradigmas de Programação Orientada a Objetos (POO) e Programação Orientada a Aspectos (POA). O objetivo principal é aplicar uma preocupação transversal, que é a verificação de saldo antes de qualquer operação de saque, em diferentes tipos de contas (corrente, poupança, investimento).

**Estrutura do Projeto**

1. **Classes de Conta**
   * **Conta**
     + **Atributos**:
       - tipoConta
       - tipoCliente
       - dataAbertura
       - saldo
     + **Métodos**:
       - sacar(double valor)
       - depositar(double valor)
       - calculaValorRefinanciamento()
       - anutencao()
       - getSaldo()
   * **ContaPoupanca**
     + Estende Conta
     + Implementa lógica específica para conta de poupança
   * **ContaInvestimento**
     + Estende Conta
     + Implementa lógica específica para conta de investimento
2. **Aspecto de Verificação de Saldo**
   * **BalanceCheckAspect**
     + Utiliza @Around para interceptar chamadas a sacar()
     + Verifica se o saldo é suficiente antes de permitir a operação
     + Registra erros usando SLF4J
3. **Exceção Personalizada**
   * **InsufficientBalanceException**
     + Lançada quando o saldo é insuficiente para realizar um saque
4. **Configuração e Testes**
   * **Main**
     + Contém testes para demonstrar a funcionalidade do sistema
     + Cria instâncias de diferentes tipos de contas e executa operações de saque

**Implementação Detalhada**

**Classe Base: Conta**

public class Conta {

// Atributos

private TipoConta tipoConta;

private TipoCliente tipoCliente;

private Date dataAbertura;

private double saldo;

// Construtor

public Conta(TipoConta tipoConta, TipoCliente tipoCliente, Date dataAbertura, double saldo) {

this.tipoConta = tipoConta;

this.tipoCliente = tipoCliente;

this.dataAbertura = dataAbertura;

this.saldo = saldo;

}

// Métodos

public synchronized void sacar(double valor) throws InsufficientBalanceException {

if (saldo >= valor) {

saldo -= valor;

} else {

throw new InsufficientBalanceException("Saldo insuficiente para realizar o saque de " + valor + " em " + this);

}

}

public void depositar(double valor) {

saldo += valor;

}

public double calculaValorRefinanciamento() {

// Lógica de cálculo de refinanciamento

return 0.0;

}

public void anutencao() {

// Lógica de anutenção

}

public double getSaldo() {

return saldo;

}

}

**Subclasses: ContaPoupanca e ContaInvestimento**

public class ContaPoupanca extends Conta {

public ContaPoupanca(TipoConta tipoConta, TipoCliente tipoCliente, Date dataAbertura, double saldo) {

super(tipoConta, tipoCliente, dataAbertura, saldo);

}

@Override

public synchronized void sacar(double valor) throws InsufficientBalanceException {

super.sacar(valor);

// Lógica adicional para conta de poupança

}

}

public class ContaInvestimento extends Conta {

public ContaInvestimento(TipoConta tipoConta, TipoCliente tipoCliente, Date dataAbertura, double saldo) {

super(tipoConta, tipoCliente, dataAbertura, saldo);

}

@Override

public synchronized void sacar(double valor) throws InsufficientBalanceException {

super.sacar(valor);

// Lógica adicional para conta de investimento

}

}

**Aspecto: BalanceCheckAspect**

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

@Aspect

public class BalanceCheckAspect {

private static final Logger logger = LoggerFactory.getLogger(BalanceCheckAspect.class);

@Around("execution(\* Conta+.sacar(..))")

public void aroundSacar(ProceedingJoinPoint joinPoint) throws Throwable {

Conta conta = (Conta) joinPoint.getTarget();

double valor = (double) joinPoint.getArgs()[0];

if (conta.getSaldo() >= valor) {

joinPoint.proceed();

} else {

logger.error("Saldo insuficiente para realizar o saque de {} em {}", valor, conta);

throw new InsufficientBalanceException("Saldo insuficiente para realizar o saque de " + valor + " em " + conta);

}

}

}

**Exceção Personalizada**

public class InsufficientBalanceException extends Exception {

public InsufficientBalanceException(String message) {

super(message);

}

}

**Classe Principal: Main**

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class Main {

private static final Logger logger = LoggerFactory.getLogger(Main.class);

public static void main(String[] args) {

Conta contaCorrente = new Conta(TipoConta.CORRENTE, TipoCliente.PESSOA\_FISICA, new Date(), 1000.0);

ContaPoupanca contaPoupanca = new ContaPoupanca(TipoConta.POUPANCA, TipoCliente.PESSOA\_JURIDICA, new Date(), 500.0);

try {

contaCorrente.sacar(1500.0); // Deve lançar InsufficientBalanceException

} catch (InsufficientBalanceException e) {

logger.error(e.getMessage());

}

try {

contaPoupanca.sacar(300.0); // Deve ser bem-sucedido

logger.info("Novo saldo da conta poupança: " + contaPoupanca.getSaldo());

} catch (InsufficientBalanceException e) {

logger.error(e.getMessage());

}

}

}

**Considerações Finais**

* **Encapsulamento e Segurança de Concorrência**: Os métodos sacar() são sincronizados para evitar problemas de concorrência.
* **Logging**: O aspecto utiliza SLF4J para registrar erros, facilitando a monitoração e debugging.
* **Extensibilidade**: A estrutura POO permite a fácil adição de novos tipos de contas e funcionalidades específicas.
* **Aspectos Transversais**: A POA permite a separação de preocupações transversais, como a verificação de saldo, mantendo o código principal limpo e focado na lógica de negócios.

Este projeto demonstra uma abordagem eficaz para a integração de POO e POA, proporcionando um sistema modular, mantível e escalável.