07/04/2025

Práticas com SOLID

Renato Teixeira Barreto

S - SOLID:

Código sem implementação:

```
package SSolid.Exemplo2;

import java.io.*;
import java.util.Scanner;

public class ProcessadorEncomendas {

    public void processar() {
        try (Scanner sc = new Scanner(System.in)) {
            System.out.println("Digite o ID da encomenda: ");
            String idEncomenda = sc.nextLine();

            System.out.println("Digite o peso (em kg): ");
            double peso = sc.nextDouble();

            double valorFrete = peso * 10;
            System.out.println("Valor do frete calculado: " + valorFrete);


            salvarEmArquivo(idEncomenda, valorFrete);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void salvarEmArquivo(String idEncomenda, double valorFrete) {
        try (BufferedWriter bw = new BufferedWriter(new
FileWriter("encomendas.txt", true))) {
            bw.write("ID: " + idEncomenda + " - Frete: " + valorFrete);
            bw.newLine();
            System.out.println("Salvo no arquivo encomendas.txt");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Código com implementação:

```
package codigos_implementados;

import java.io.*;
import java.util.Scanner;

class EntradaUsuario {
    public String obterIdEncomenda() {
        Scanner sc = new Scanner(System.in);
        System.out.println("Digite o ID da encomenda: ");
        return sc.nextLine();
    }

    public double obterPeso() {
        Scanner sc = new Scanner(System.in);
        System.out.println("Digite o peso (em kg): ");
        return sc.nextDouble();
    }
}

class CalculadoraFrete {
    public double calcular(double peso) {
        return peso * 10;
    }
}

class SalvadorArquivo {
    public void salvar(String idEncomenda, double valorFrete) {
        try (BufferedWriter bw = new BufferedWriter(new
FileWriter("encomendas.txt", true))) {
            bw.write("ID: " + idEncomenda + " - Frete: " + valorFrete);
            bw.newLine();
            System.out.println("Salvo no arquivo encomendas.txt");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class ProcessadorEncomendas {
    private final EntradaUsuario entradaUsuario = new EntradaUsuario();
    private final CalculadoraFrete calculadoraFrete = new CalculadoraFrete();
    private final SalvadorArquivo salvadorArquivo = new SalvadorArquivo();

    public void processar() {
        try {
            String idEncomenda = entradaUsuario.obterIdEncomenda();
            double peso = entradaUsuario.obterPeso();

            double valorFrete = calculadoraFrete.calcular(peso);
            System.out.println("Valor do frete calculado: " + valorFrete);

            salvadorArquivo.salvar(idEncomenda, valorFrete);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Explicação das alterações:

O Single Responsibility Principle (SRP), ou Princípio da Responsabilidade Única afirma que "uma classe deve ter apenas um motivo para mudar", ou seja, deve estar focada em uma única responsabilidade específica. Quando uma classe assume múltiplas funções ou responsabilidades, ela se torna mais difícil de manter, testar e evoluir, contrariando os objetivos principais do SOLID, que são tornar o design de software mais compreensível, flexível e fácil de manter.

No código original da classe **`ProcessadorEncomendas`**, identificamos múltiplas responsabilidades concentradas em uma única classe: entrada de dados via terminal, cálculo do valor do frete com base no peso e, por fim, a gravação dessas informações em um arquivo de texto. Cada uma dessas ações representa uma responsabilidade distinta. Se qualquer uma dessas partes sofrer uma alteração no futuro (por exemplo, mudar a forma como o frete é calculado ou como os dados são persistidos), será necessário modificar a mesma classe, violando assim o SRP.

Para corrigir isso e alinhar o código ao SRP, realizamos uma refatoração que divide essas responsabilidades em classes separadas. A entrada de dados foi delegada para a classe **`EntradaUsuario`**, que encapsula toda a lógica de interação com o usuário pelo console. O cálculo do valor do frete foi movido para uma nova classe chamada **`CalculadoraFrete`**, que fica responsável exclusivamente por essa operação. A tarefa de salvar os dados em arquivo foi delegada à classe **`SalvadorArquivo`**, que agora lida com a persistência dos dados. Por fim, a classe **`ProcessadorEncomendas`** passou a exercer o papel de coordenadora das operações, reunindo as dependências das outras classes e orquestrando o fluxo da aplicação.

Com essas alterações, cada classe tem agora apenas um motivo para mudar, conforme preconizado pelo SRP. Isso melhora significativamente a manutenibilidade do código, uma das vantagens destacadas no PDF, além de favorecer a extensibilidade e a testabilidade, já que cada responsabilidade está isolada e pode ser modificada ou testada de forma independente. A aplicação do SRP também contribui para uma alta coesão e baixo acoplamento, princípios que são destacados como fundamentais para a qualidade de sistemas orientados a objetos.

O - SOLID:

Código sem implementação:

```
package codigos_antigos;
public class SistemaPagamento {

    public void realizarPagamento(double valor, String metodo) {
        if ("CARTAO".equalsIgnoreCase(metodo)) {
            System.out.println("Pagamento de R$" + valor + " realizado com
CARTÃO.");
        } else if ("PIX".equalsIgnoreCase(metodo)) {
            System.out.println("Pagamento de R$" + valor + " realizado via PIX.");
        } else if ("BOLETO".equalsIgnoreCase(metodo)) {
            System.out.println("Pagamento de R$" + valor + " realizado via
BOLETO.");
        } else {
            System.out.println("Método de pagamento não suportado!");
```

```
        }
    }
}
```

Código com implementação:

```
package codigos_implementados;

interface MetodoPagamento {
    void pagar(double valor);
}

class PagamentoCartao implements MetodoPagamento {
    public void pagar(double valor) {
        System.out.println("Pagamento de R$" + valor + " realizado com
CARTÃO.");
    }
}

class PagamentoPIX implements MetodoPagamento {
    public void pagar(double valor) {
        System.out.println("Pagamento de R$" + valor + " realizado via PIX.");
    }
}

class PagamentoBoleto implements MetodoPagamento {
    public void pagar(double valor) {
        System.out.println("Pagamento de R$" + valor + " realizado via
BOLETO.");
    }
}

public class SistemaPagamento {
    public void realizarPagamento(double valor, MetodoPagamento
metodo) {
        metodo.pagar(valor);
    }
}
```

Explicação das alterações:

Foi aplicado ao código fornecido o segundo princípio do SOLID, o **Open/Closed Principle (OCP)**, ou Princípio Aberto/Fechado. Esse princípio estabelece que "as classes devem estar abertas para extensão, mas fechadas para modificação", ou seja, a estrutura do código deve permitir que novos comportamentos sejam adicionados sem que seja necessário alterar o código já existente. O objetivo é minimizar os efeitos colaterais e riscos ao sistema quando novas funcionalidades são implementadas.

No código original da classe **`SistemaPagamento`**, há uma violação clara do OCP. A lógica de pagamento está toda centralizada em uma única função **`realizarPagamento`**, que verifica o método de pagamento por meio de estruturas condicionais `if-else if`. Para cada novo método de pagamento que se deseje adicionar (como PayPal, débito automático, etc.), seria necessário modificar essa mesma função, o que contraria diretamente o princípio de estar "fechado para modificação".

Para corrigir essa violação e tornar o sistema aderente ao OCP, realizamos uma refatoração com base no uso de **polimorfismo e abstração**. Criamos uma interface **`MetodoPagamento`**, que define o contrato `pagar(double valor)`. A partir dela, implementamos classes específicas para cada tipo de pagamento, como **`PagamentoCartao`**, **`PagamentoPIX`** e **`PagamentoBoleto`**, cada uma encapsulando sua própria lógica de pagamento. A classe **`SistemaPagamento`**, por sua vez, foi alterada para receber um objeto do tipo **`MetodoPagamento`**, chamando seu método `pagar` sem precisar conhecer sua implementação interna.

L - SOLID:

Códigos sem implementação:

```
package codigos_antigos;

public class ContaBancaria {
    protected double saldo;
```

```
    public void depositar(double valor) {
        saldo += valor;
    }

    public void sacar(double valor) {
        saldo -= valor;
    }

    public double getSaldo() {
        return saldo;
    }
}
```

```
package codigos_antigos;

public class ContaPoupanca extends ContaBancaria {

    @Override
    public void sacar(double valor) {
        throw new UnsupportedOperationException("Resgate não é permitido
direto.");
    }
}
```

Códigos com implementação:

```
package codigos_implementados;

public interface Conta {
    void depositar(double valor);
    double getSaldo();
}
```

```
class ContaBancaria implements Conta {
    protected double saldo;

    @Override
    public void depositar(double valor) {
```

```
        saldo += valor;
    }

    public void sacar(double valor) {
        saldo -= valor;
    }

    @Override
    public double getSaldo() {
        return saldo;
    }
}


class ContaPoupanca implements Conta {
    protected double saldo;

    @Override
    public void depositar(double valor) {
        saldo += valor;
    }

    @Override
    public double getSaldo() {
        return saldo;
    }


    public void resgatar(double valor) {

        System.out.println("Resgate agendado no valor de: R$" + valor);
    }
}
```

Explicação das alterações:

Foi aplicado aos códigos fornecidos o terceiro princípio do SOLID, conhecido como **Liskov Substitution Principle (LSP)**, ou Princípio da Substituição de Liskov. Esse princípio estabelece que **subtipos devem ser substituíveis por seus tipos base sem que o comportamento esperado**

da aplicação seja alterado. Em outras palavras, uma subclasse deve ser capaz de substituir sua superclasse em qualquer contexto, **sem causar efeitos colaterais inesperados ou quebrar a lógica do sistema**.

No cenário apresentado, temos a classe **`ContaBancaria`** com os métodos `depositar`, `sacar` e `getSaldo`, funcionando como uma interface comum para contas bancárias genéricas. Entretanto, a subclasse **`ContaPoupanca`**, que herda de **`ContaBancaria`**, **quebra esse contrato** ao sobrescrever o método `sacar` com uma exceção (`UnsupportedOperationException`), alegando que o saque não é permitido. Essa violação do LSP é crítica porque compromete a **substituibilidade**: qualquer código que espera trabalhar com uma **`ContaBancaria`** comum e tenta chamar o método `sacar` em uma instância de `ContaPoupanca` será interrompido por uma exceção inesperada, causando falhas de execução ou necessidade de tratamentos adicionais.

Para resolver essa violação e alinhar o código ao LSP conforme recomendado no PDF, realizamos uma refatoração que reorganiza a hierarquia de classes para refletir comportamentos distintos de forma adequada. Ao invés de permitir que uma classe (`ContaPoupanca`) herde de outra (`ContaBancaria`) e **remova uma funcionalidade essencial**, criamos uma abstração superior — uma interface `Conta` — que define comportamentos comuns às contas, e implementamos comportamentos específicos em classes diferentes, respeitando as regras de cada tipo de conta.

I - SOLID:

Códigos sem implementação:

```
package codigos_antigos;

public interface Veiculo {
   void dirigir();
   void voar();
   void navegar();
}
```

```
package codigos_antigos;


public class Carro implements Veiculo {
    @Override
    public void dirigir() {
        System.out.println("Carro está dirigindo na estrada...");
    }


    @Override
    public void voar() {
        throw new UnsupportedOperationException("Carro não voa!");
    }


    @Override
    public void navegar() {
        throw new UnsupportedOperationException("Carro não navega!");
    }
}
```

Códigos com implementação:

```
package codigos_implementados;

interface Veiculo {
    void dirigir();
}
```

```
interface VeiculoVoador {
    void voar();
}

interface VeiculoNautico {
    void navegar();
}



class Carro implements Veiculo {
    @Override
    public void dirigir() {
        System.out.println("Carro está dirigindo na estrada...");
    }
}

class Aviao implements Veiculo, VeiculoVoador {
    @Override
    public void dirigir() {
        System.out.println("Avião taxiando na pista...");
    }

    @Override
    public void voar() {
        System.out.println("Avião está voando...");
    }
}

class Barco implements VeiculoNautico {
    @Override
    public void navegar() {
        System.out.println("Barco está navegando no mar...");
    }
}
```

Explicação das alterações:

Aplicamos aos códigos fornecidos o quarto princípio do SOLID: o Interface Segregation Principle (ISP), ou Princípio da Segregação de Interfaces. Este princípio estabelece que nenhuma classe deve ser forçada a implementar métodos que não utiliza. Interfaces devem ser pequenas e específicas, contendo apenas as funcionalidades diretamente relacionadas a um tipo de comportamento. Assim, evitamos a criação de interfaces “inchadas” que forçam implementações incoerentes ou desnecessárias.

No código original, a interface `Veiculo` contém três métodos: `dirigir`, `voar` e `navegar`. A intenção é representar diferentes capacidades que um veículo pode ter. No entanto, ao forçar a classe `Carro` a implementar essa interface, temos um problema: o carro não voa nem navega. Para cumprir o contrato da interface, o desenvolvedor se vê obrigado a lançar exceções (`UnsupportedOperationException`) para os métodos `voar` e `navegar`, o que viola o ISP, pois a classe está sendo forçada a implementar métodos que não fazem sentido para seu domínio.

Essa prática vai contra o que o ISP recomenda: uma classe nunca deve ser obrigada a depender de métodos que ela não usa. Além disso, isso compromete a manutenção, legibilidade e robustez do código, podendo causar erros de execução se um cliente da interface `Veiculo` tentar chamar `voar()` em um carro, por exemplo.

Para aplicar corretamente o ISP, devemos quebrar a interface `Veiculo` em interfaces menores e mais específicas, com métodos relacionados apenas a um tipo de comportamento. Com isso, cada classe pode implementar apenas as interfaces que realmente refletem suas funcionalidades reais.

07 de Abril de 2025, Londrina PR