x

ISABELY CRISTINA CAMARGO MARTINS
GIOVANA GIORGI BIGATI
YSADORA SISTINI TEODORO RAMOS DE OLIVEIRA

BOAS PRÁTICAS DE PROGRAMAÇÃO: RELATÓRIO SOLID

Londrina
2025

1. INTRODUÇÃO

Durante as aulas de Boas Práticas de Programação, aprendemos como escrever códigos mais organizados, reutilizáveis e fáceis de manter. Um dos principais conteúdos foi o conjunto de princípios SOLID, que servem como guia para construir sistemas bem estruturados e preparados para mudanças.

Nesta atividade, analisamos quatro exemplos de código que apresentavam problemas relacionados aos princípios SOLID (exceto o D, que também resolvemos, mas não era exigido no enunciado). Em grupo, dividimos as responsabilidades para entender cada caso, aplicar melhorias no código e elaborar as explicações para o relatório.

Isabely ficou responsável pelo princípio S (Responsabilidade Única) e pela montagem geral do relatório;

Ysadora ficou responsável pelos princípios O (Aberto/Fechado) e L (Substituição de Liskov);

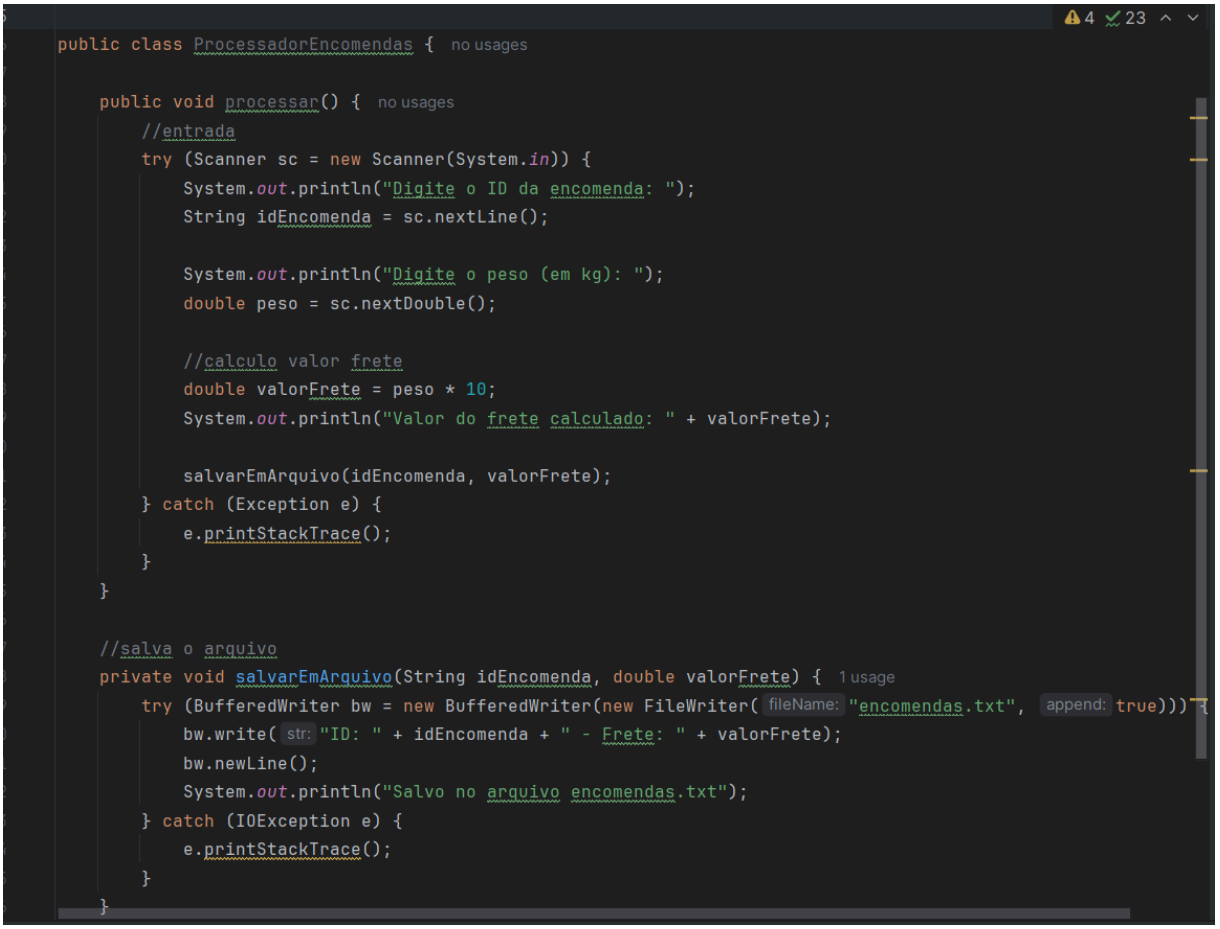
Giovana ficou responsável pelos princípios I (Segregação de Interfaces) e D (Inversão de Dependência).

A seguir, apresentamos as comparações entre as versões antigas e as implementações corrigidas de cada exemplo, explicando de forma simples como as alterações ajudaram a deixar o código mais limpo, flexível e fácil de entender.

2. **S**OLID - Single Responsibility Principle

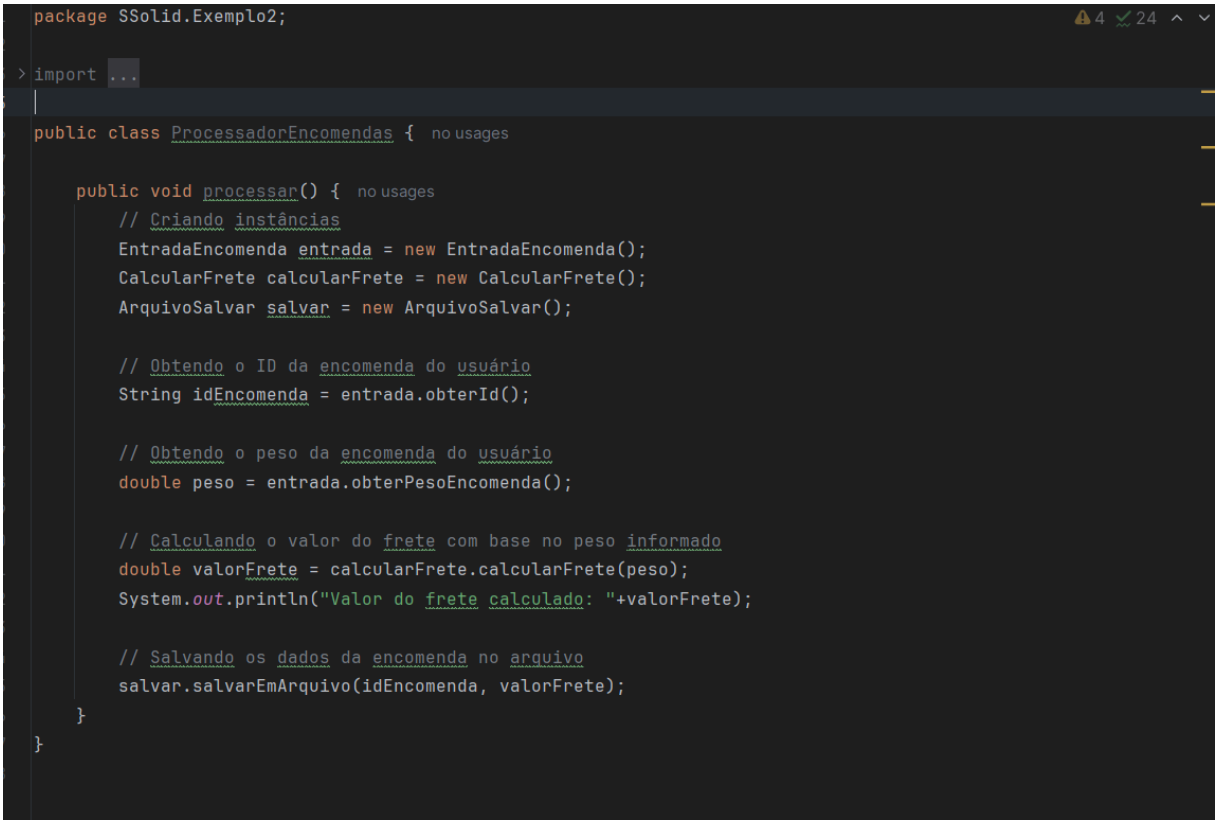
O princípio S do SOLID é o Princípio da Responsabilidade Única (Single Responsibility Principle, SRP). Ele afirma que uma classe deve ter apenas uma razão para mudar, ou seja, ela deve ter apenas uma responsabilidade. A classe **ProcessadorEncomendas** está fazendo mais de uma coisa: ela está processando a entrada do usuário, calculando o valor do frete e salvando essa informação em um arquivo.

Versão anterior



```java
public class ProcessadorEncomendas {  no usages

    public void processar() {  no usages
        //entrada
        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();

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

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

    //salva o arquivo
    private void salvarEmArquivo(String idEncomenda, double valorFrete) {  1 usage
        try (BufferedWriter bw = new BufferedWriter(new FileWriter( fileName: "encomendas.txt",  append: true))) {
            bw.write( str: "ID: " + idEncomenda + " - Frete: " + valorFrete);
            bw.newLine();
            System.out.println("Salvo no arquivo encomendas.txt");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Versão atualizada



```java
package SSolid.Exemplo2;

import ...

public class ProcessadorEncomendas {  no usages

    public void processar() {  no usages
        // Criando instâncias
        EntradaEncomenda entrada = new EntradaEncomenda();
        CalcularFrete calcularFrete = new CalcularFrete();
        ArquivoSalvar salvar = new ArquivoSalvar();

        // Obtendo o ID da encomenda do usuário
        String idEncomenda = entrada.obterId();

        // Obtendo o peso da encomenda do usuário
        double peso = entrada.obterPesoEncomenda();

        // Calculando o valor do frete com base no peso informado
        double valorFrete = calcularFrete.calcularFrete(peso);
        System.out.println("Valor do frete calculado: "+valorFrete);

        // Salvando os dados da encomenda no arquivo
        salvar.salvarEmArquivo(idEncomenda, valorFrete);
    }
}
```

Na versão atualizada dividimos essa classe em outras que tenham responsabilidades mais claras e distintas.

1. **Responsabilidade da Entrada do Usuário:** A primeira implemetação foi a criação da nova clase **EntradaEncomenda,** onde movemos a responsabilidade da entrada do usuário que estava na classe **ProcessadorEncomendas.**

```java
package SSolid.Exemplo2;

import java.util.Scanner;

//Responsável por coletar os dados de entrada (ID da encomenda e peso).
public class EntradaEncomenda {  2 usages

    public String obterId(){  1 usage
        try (Scanner sc = new Scanner(System.in)) {
            System.out.println("Digite o ID da encomenda: ");
            return sc.nextLine();
        }
    }

    public double obterPesoEncomenda(){  1 usage
        try (Scanner sc = new Scanner(System.in)) {
            System.out.println("Digite o peso (em kg): ");
            return sc.nextDouble();
        }
    }
}
```

2. **Responsabilidade de Calcular o Frete:** Após isso foi criado outra classe **CalcularFrete** que vai ser responsável por calcular o frete com base no peso.

```java
package SSolid.Exemplo2;

public class CalcularFrete {  2 usages

    // Responsável pelo cálculo do valor do frete com base no peso.
    public double calcularFrete(double peso){  1 usage
        return peso * 10;
    }
}
```

3. **Responsabilidade de Salvar em Arquivo:** A classe **ArquivoSalvar** vai ser responsável por salvar a informação no arquivo.

```java
package SSolid.Exemplo2;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

//Responsável por salvar a informação no arquivo.
public class ArquivoSalvar {  2 usages
        public void salvarEmArquivo(String idEncomenda, double valorFrete){  1 usage
            try (BufferedWriter bw = new BufferedWriter(new FileWriter( fileName: "encomendas.txt",  append: true)
                bw.write( str: "ID: " + idEncomenda + " - Frete: " + valorFrete);
                bw.newLine();
                System.out.println("Salvo no arquivo encomendas.txt");
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
}
```

3. S**O**LID - Open-Closed Principle

O princípio O do SOLID é o Princípio Aberto-Fechado. Ele afirma que uma Entidades de software devem estar abertas para extensão, mas fechadas para modificação, isto é, deve ser possível estender o comportamento, porém sem alterar o código fonte original.

Versão anterior

```java
package OSolid.Exemplo2;

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!");
        }
    }
}
```

A classe **SistemaPagamento** violava o princípio OCP porque, sempre que um novo método de pagamento fosse adicionado, seria preciso **editar essa classe**, adicionando novos if/else. Isso torna o sistema mais suscetível a erros com o tempo.

Versão atualizada

Classe de Sistema de Pagamento:

```java
package OSolid.Exemplo2;

public class SistemaPagamento {
    public void acionarpagamento(double valor, ISistemaPagamento metodoPagamento) {
        metodoPagamento.pagar(valor);
    }
}
```

Interface:

```java
package OSolid.Exemplo2;

public interface ISistemaPagamento {
    public void pagar(double valor);
}
```

Um dos metódos de implementação:

```java
package OSolid.Exemplo2;

public class MetodoPix implements ISistemaPagamento{
    @Override
    public void pagar (double valor) {
        System.out.println("Pagamento de R$" + valor + " realizado via PIX.");

    }
}
```

A nova versão foi separada em 6 classe, usando **polimorfismo e abstração com interface**, possibilitando a implementação de novos metodos de pagamento sem a modficação da classe original.

Com essa refatoração, a classe **SistemaPagamento** está fechada para modificação e aberta para extensão, pois agora qualquer novo método de pagamento só precisa implementar a interface **ISistemaPagamento**, sem alterar código já existente. Isso segue corretamente o princípio **O** do SOLID.

4. SO**L**ID - Liskov Substitution Principle

O princípio L do SOLID é o Princípio da Substituição de Liskov. Ele afirma que uma subclasse não pode alterar o comportamento esperado da superclasse, isto é, se B é uma subclasse de A, então devemos poder usar B no lugar de A sem problemas. A subclasse não deve quebrar o contrato estabelecido pela superclasse.

Versão anterior

Superclasse

```java
package LSOLID.Exemplo2;

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;
    }
}
```

Subclasse

```java
package LSOLID.Exemplo2;

public class ContaPoupanca extends ContaBancaria {

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

A classe **ContaPoupanca** herdava a **ContaBancaria**, mas sobrescrevia o método sacar para lançar uma exceção, pois saques não eram permitidos. Isso quebra o princípio L, já que qualquer código que espera uma ContaBancaria pode falhar se receber uma ContaPoupanca.

Versão atualizada

Interface Básica:

```java
package LSOLID.Exemplo2;

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

Interface Estendida (com saque):

```java
package LSOLID.Exemplo2;

public interface ContaBasicaComSaque extends ContaBasica {
    void sacar(double valor);
}
```

Classe ContaBancaria (com saque):

```java
package LSOLID.Exemplo2;

public class ContaBancaria implements ContaBasicaComSaque {
    protected double saldo;

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

    public double getSaldo() {
        return saldo;
    }

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

Classe ContaPoupanca (sem saque):

```java
package LSOLID.Exemplo2;

public class ContaPoupanca implements ContaBasica {
    protected double saldo;

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

    public double getSaldo() {
        return saldo;
    }

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

}
```

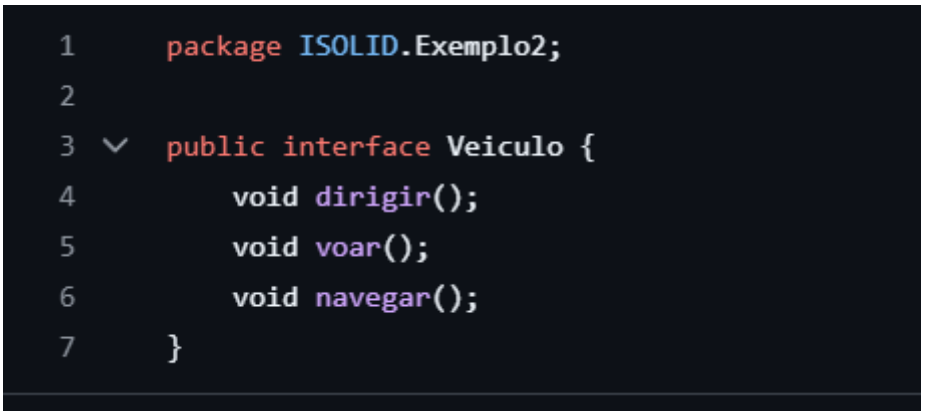
A nova versão foi separada em 4 classes, sendo que, o comportamento de saque foi separado na classe **ContaBasicaComSaque,** para que apenas as contas que suportam saque implementem essa funcionalidade. Assim, ContaPoupanca não precisa sobrescrever sacar com uma exceção.

Com essa refatoração, **cada classe implementa apenas as operações que realmente oferece.**

5. SOL**I**D - Interface Segregation Principle

O princípio I do SOLID é o Princíipio da Segregação de Interfaces. Ele afirma que´e melhor ter várias interfaces específicas do que uma única interface “inchada”, isto é, as interfaces não deveriam obrigar as classes a depender de métodos que não utilizam.

Versão anterior



```
1   package ISOLID.Exemplo2;
2
3   public interface Veiculo {
4       void dirigir();
5       void voar();
6       void navegar();
7   }
```

A interface Veiculo estava mal implemetada, pois exigia que qualquer classe que a implementasse fornecesse todos os métodos: dirigir, voar e navegar, ou seja, a classe era forçada a implementar **métodos que ela não precisa**.

Versão Atualizada

Classe VeiculoTerrestre:

```java
package ISOLID.Exemplo2;

public interface VeiculoTerrestre {
    void dirigir();
}
```

Classe VeiculoAquatico:

```java
package ISOLID.Exemplo2;

public interface VeiculoAquatico {
    void navegar();
}
```

Classe Carro:

```java
package ISOLID.Exemplo2;

public class Carro implements VeiculoTerrestre { // assim a classe não precisa implementar métodos que não preci
    @Override
    public void dirigir() {
        System.out.println("Carro está dirigindo na estrada...");
    }

}
```

A solução foi separar a interface **Veiculo** em interfaces menores e mais específicas, de acordo com o tipo de veículo, assim, não deixando a interface “inchada”. Logo, cada classe só implementa o que realmente faz sentido para ela.

6. SOLI**D** - Dependency Inversion Principle

O princípio D do SOLID é o Princípio da Inversão de Dependência. Ele afirma que classes importantes (de alto nível) não devem depender diretamente de coisas simples (de baixo nível). Mas, o que defini classes de alto nível e de baixo nível? Classe de alto nível são classes que contém a lógica principal do sistema, já as classes de baixo nível são as que fazer tarefas específicas, geralmente de apoio, que podem ser trocadas sem afetar o resto do sistema, isto é, a lógica principal (alto nível) **deveria funcionar sozinha**, sem depender diretamente das classes "de apoio". Senão, ela **fica engessada** e difícil de mudar.

Versão anterior

```java
package DSOLID.Exemplo2;

public class ServicoPagamento {
    private ConsoleLogger logger = new ConsoleLogger();

    public void pagar(double valor) {
        // Lógica de pagamento
        logger.log("Pagamento de R$" + valor + " realizado com sucesso!");
    }
}
```

Nessa versão o **ServicoPagamento** estava dependendo diretamente da classe **ConsoleLogger**. Uma forma de entender melhor, é ver o **ServicoPagamento** como uma atendente do caixa e o **ConsoleLogger** como o tipo de caneta que ele usa pra assinar os recibos.

A forma que está agora da a entender que a atendente **só sabe trabalhar com uma caneta azul específica**, a forma errada.

Versão Atualizada

ServicoPagamento

```java
package DSOLID.Exemplo2;

public class ServicoPagamento {
    private ILogger iLogger;

    public ServicoPagamento(ILogger iLogger) {
        this.iLogger = iLogger;
    }

    public void pagar(double valor) {
        // Lógica de pagamento, com dependência indireta
        iLogger.log("Pagamento de R$" + valor + " realizado com sucesso!");
    }
}
```

ILogger

```java
package DSOLID.Exemplo2;

public interface ILogger {
    void log(String mensagem);
}
```

ConsoleLogger

```java
package DSOLID.Exemplo2;

public class ConsoleLogger implements ILogger {
    @Override
    public void log(String mensagem) {
        System.out.println("LOG: " + mensagem);
    }
}
```

A solução foi criar uma interface chamada **ILogger**, que representa o que qualquer tipo de "caneta" deve ser capaz de fazer. A classe **ConsoleLogger** agora implementa essa interface, ou seja, é uma das possíveis "canetas" que podem ser usadas. Com isso, o **ServicoPagamento** passou a depender apenas da interface **ILogger**, e não mais diretamente da **ConsoleLogger**. Voltando ao exemplo: agora a atendente do caixa pede uma caneta para assinar os recibos, e não se importa com qual caneta será entregue. Seguindo o princípio **DIP**.

7. Conclusão

Estudar e aplicar os princípios SOLID mostrou como algumas mudanças simples podem deixar o código muito mais organizado e fácil de entender. Ao corrigir os exemplos, conseguimos enxergar como separar bem as responsabilidades, evitar repetições desnecessárias e permitir que novas funcionalidades sejam adicionadas sem bagunçar o que já estava funcionando.

No fim das contas, o SOLID ajuda a escrever um código que "conversa melhor" com quem vai ler ou mexer nele depois, seja outra pessoa ou até a gente no futuro. É como deixar tudo mais limpo, claro e pronto pra crescer sem dor de cabeça.