Práticas com SOLID

Grupo: Victor Scheller Zuccoli, Vitor Hugo de Oliveira Paloco, Diego Vinicius dos Santos

Letra S – Single Responsibility Principle

Cada classe deve ter apenas uma única responsabilidade.

No primeiro exemplo, o código original misturava responsabilidades: ler entrada do usuário, calcular frete e salvar dados no arquivo. Para respeitar esse princípio, separamos essas tarefas em três classes diferentes. Cada uma agora tem uma função clara: uma para calcular o frete, uma para salvar os dados, e uma principal que faz a leitura e coordena as outras.

Código ajustado:

`CalcularFrete.java`

```
package SSOLID.Exemplo2;

public class CalcularFrete {
    public double executar(double peso) {
        return peso * 10;
    }
}
```

`SalvarArquivo.java`

```
package SSOLID.Exemplo2;

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

public class SalvarArquivo {
    public void executar(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();
        }
    }
}
```

ProcessarEncomenda.java

```
package SSOLID.Exemplo2;

import java.util.Scanner;

public class ProcessarEncomenda {
    private final CalcularFrete calcularFrete = new CalcularFrete();
    private final SalvarArquivo salvarArquivo = new SalvarArquivo();

    public void executar() {
        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 = calcularFrete.executar(peso);
            System.out.println("Valor do frete calculado: " +
valorFrete);

            salvarArquivo.executar(idEncomenda, valorFrete);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Letra O – Open/Closed Principle

As classes devem estar abertas para extensão, mas fechadas para modificação.

Neste exemplo, originalmente a classe de pagamento fazia verificações com `if` para saber qual método de pagamento usar. Para corrigir isso, criamos uma interface comum e implementamos uma classe para cada tipo de pagamento. Assim, o sistema pode ser estendido com novos métodos sem precisar alterar código existente.

Código ajusatado:

FormaPagamento.java

```
package OSolid.Exemplo2;

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

Cartao.java

```
package OSolid.Exemplo2;

public class Cartao implements FormaPagamento {
    @Override
    public void pagar(double valor) {
        System.out.println("Pagamento de R$" + valor + " realizado com
CARTÃO.");
    }
}
```

Pix.java

```
package OSolid.Exemplo2;

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

```
Boleto.java
package OSolid.Exemplo2;

public class Boleto implements FormaPagamento {
    @Override
    public void pagar(double valor) {
        System.out.println("Pagamento de R$" + valor + " realizado via
```

```
BOLETO.");
    }
}
```

SistemaPagamento.java

```
package OSolid.Exemplo2;

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

Main.java

```
package OSolid.Exemplo2;

public class Main {
    public static void main(String[] args) {
        SistemaPagamento sistema = new SistemaPagamento();

        sistema.realizarPagamento(100.0, new Pix());
        sistema.realizarPagamento(200.0, new Boleto());
        sistema.realizarPagamento(300.0, new Cartao());
    }
}
```

Letra L – Liskov Substitution Principle

Uma subclasse deve poder ser usada no lugar da sua superclasse sem causar erros.

O problema neste exemplo era que a classe `ContaPoupanca` herdava de uma classe `ContaBancaria`, mas não permitia saque, o que fazia o sistema lançar exceções quando esse método era chamado. Para resolver, criamos interfaces separadas: uma para contas que permitem saque e outra só para depósito. Com isso, cada tipo de conta implementa apenas o que realmente faz sentido.

Código ajustado:

Deposito.java

```java
package LSOLID.Exemplo2;

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

Saque.java

```java
package LSOLID.Exemplo2;

public interface Saque extends Deposito {
    void sacar(double valor);
}
```

ContaCorrente.java

```java
package LSOLID.Exemplo2;

public class ContaCorrente implements Saque {
    private double saldo;

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

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

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

ContaPoupanca.java

```
package LSOLID.Exemplo2;

public class ContaPoupanca implements Deposito {
    private double saldo;

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

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

Letra I – Interface Segregation Principle

As classes não devem ser forçadas a implementar métodos que não utilizam.

No exemplo original, a interface `Veiculo` obrigava todas as classes a implementar os métodos `dirigir()`, `voar()` e `navegar()`, mesmo que isso não fizesse sentido para todas. A solução foi dividir essa interface em três menores. Agora, cada classe implementa só as funcionalidades que realmente tem.

Código ajustado:

Dirigivel.java

```
package ISOLID.Exemplo2;

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

Aereo.java

```
package ISOLID.Exemplo2;

public interface Aereo {
```

```
    void voar();
}
```

Navegador.java

```
package ISOLID.Exemplo2;

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

```
Carro.java
package ISOLID.Exemplo2;

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

Aviao.java

```
package ISOLID.Exemplo2;

public class Aviao implements Dirigivel, Aereo {
    @Override
    public void dirigir() {
        System.out.println("Avião está dirigindo na estrada...");
    }

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

Navio.java

```
package ISOLID.Exemplo2;
```

```java
public class Navio implements Navegador {
    @Override
    public void navegar() {
        System.out.println("Navio está navegando em alto mar...");
    }
}
```

Main.java

```java
package ISOLID.Exemplo2;

public class Main {
    public static void main(String[] args) {
        Dirigivel carro = new Carro();
        carro.dirigir();

        Aviao aviao = new Aviao();
        aviao.dirigir();
        aviao.voar();

        Navegador navio = new Navio();
        navio.navegar();
    }
}
```