Práticas com SOLID

Pedro Henrique Tavares
Raphael Garcia Palma

Princípio Aplicado: ISOLID (Interface Segregation Principle)

Pacote Analisado

ISolid.Exemplo2

Problema Identificado

O código original violava o Princípio da Segregação de Interfaces (ISP) porque a interface Veiculo era demasiadamente abrangente, obrigando implementações desnecessárias. A classe Carro, por exemplo, era forçada a implementar os métodos voar() e navegar(), que não faziam sentido para esse tipo de veículo. Isso resultava em lançamentos de exceções como UnsupportedOperationException, tornando o código rígido, acoplado e difícil de manter.

Código Antes

Interface Veiculo

```
package ISolid.Exemplo2;

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

Classe Carro

```java
package ISolid.Exemplo2;

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!");
    }
}
```

Solução Aplicada

Para corrigir a violação do ISP, a interface Veiculo foi dividida em interfaces menores e mais coesas:

- Dirigivel: Representa veículos que podem dirigir.
- Voador: Representa veículos que podem voar.
- Navegavel: Representa veículos que podem navegar.

Com isso, a classe Carro agora implementa apenas Dirigivel, respeitando o seu real comportamento e responsabilidade.

Código Depois

Interface Dirigível

```java
package ISolid.Exemplo2;

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

Interface Voador

```java
package ISolid.Exemplo2;

public interface Voador {
    void voar();
}
```

Interface Navegável

```java
package ISolid.Exemplo2;

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

Classe Carro

```java
package ISolid.Exemplo2;

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

Justificativa da Solução

- Segregação de Interfaces: Agora, cada classe implementa apenas as interfaces necessárias, promovendo menor acoplamento.
- Código Limpo e Funcional: Remoção de métodos inúteis e exceções desnecessárias.
- Flexibilidade: Facilita a expansão do sistema com novas classes como Aviao ou Barco, que podem implementar Voador ou Navegavel conforme apropriado.
- Modularidade: O sistema torna-se mais modular, coeso e fácil de manter.

==========

Princípio Aplicado: LSOLID (Liskov Substitution Principle)

Pacote Analisado

LSolid.Exemplo2

Problema Identificado

O código original violava o Princípio da Substituição de Liskov (LSP) porque a classe ContaPoupanca herdava de ContaBancaria, mas sobrescrevia o método sacar para lançar uma exceção. Isso impedia que ContaPoupanca fosse substituída por ContaBancaria em contextos que esperavam o comportamento de saque, quebrando a expectativa de substituição sem alterar o comportamento correto do programa.

Código Antes

Classe ContaBancaria

```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;
    }
}
```

Classe ContaPoupanca

```java
package LSolid.Exemplo2;

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

Solução Aplicada

Para corrigir a violação do LSP, foram introduzidas interfaces específicas para separar os comportamentos esperados:

- ContaBasica: Define operações básicas (depósito e consulta de saldo).
- ContaComSaque: Estende ContaBasica para contas que permitem saques.

ContaBancaria implementa ContaComSaque, enquanto ContaPoupanca implementa apenas ContaBasica, removendo a herança problemática.

Código Depois

Interface ContaBasica

```java
package LSolid.Exemplo2;

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

Interface ContaComSaque

```java
package LSolid.Exemplo2;

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

Classe ContaBancaria

```java
package LSolid.Exemplo2;

public class ContaBancaria implements ContaComSaque {
    protected double saldo;

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

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

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

Classe ContaPoupanca

```java
package LSolid.Exemplo2;

public class ContaPoupanca implements ContaBasica {
    protected double saldo;

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

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

Justificativa da Solução

- Substituição Garantida: ContaBancaria pode ser usada onde ContaComSaque é esperado, e ContaPoupanca onde ContaBasica é esperado, sem exceções inesperadas.

- Separação de Comportamentos: A remoção da herança evita que subclasses violem o contrato da superclasse.
- Flexibilidade: O sistema agora suporta diferentes tipos de contas sem comprometer a integridade do LSP.

==========

Relatório SOLID - OSOLID (Princípio Aberto/Fechado)

Pacote: OSolid.Exemplo2

Problema Identificado

A classe SistemaPagamento original violava o Princípio Aberto/Fechado (OCP) porque estava fechada para extensão e aberta para modificação. A lógica de pagamento era implementada com uma estrutura if-else que verificava o tipo de método de pagamento. Adicionar um novo método (ex.: "TRANSFERÊNCIA") exigiria alterar diretamente o método realizarPagamento, o que contraria o OCP.

Código Antes

```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!");
        }
    }
}
```

Solução Aplicada
Para corrigir a violação do OCP, foi introduzida uma interface MetodoPagamento e classes específicas para cada tipo de pagamento. A classe SistemaPagamento agora delega a execução para o método implementado, tornando o sistema aberto para extensão (novas classes) e fechado para modificação. Código Depois Interface MetodoPagamento

```java
package OSolid.Exemplo2;

public interface MetodoPagamento {
    void realizarPagamento(double valor);
}
```

Pagamento Cartão

```java
package OSolid.Exemplo2;

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

Pagamento Pix

```java
package OSolid.Exemplo2;

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

Pagamento Boleto

```java
package OSolid.Exemplo2;

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

Sistema Pagamento

```java
package OSolid.Exemplo2;

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

Justificativa

- Aberto para Extensão: Novos métodos de pagamento podem ser adicionados criando novas classes que implementem MetodoPagamento, sem alterar SistemaPagamento.
- Fechado para Modificação: A classe SistemaPagamento não precisa ser modificada para suportar novos métodos, eliminando a necessidade de if-else.
- Flexibilidade: O sistema agora é mais modular e fácil de manter ou expandir.

==========

Relatório SOLID - SSOLID (Princípio da Responsabilidade Única)

Pacote: SSolid.Exemplo2

Problema Identificado

A classe ProcessadorEncomendas original violava o Princípio da Responsabilidade Única (SRP) ao ter duas responsabilidades distintas:

1. Coletar entrada do usuário e calcular o frete.
2. Salvar os dados em um arquivo (encomendas.txt).

Isso significa que mudanças na lógica de cálculo ou no formato de salvamento exigiriam alterações na mesma classe, aumentando a complexidade e o risco de erros.

Código Antes

```java
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();
        }
    }
}
```

Solução Aplicada

Para corrigir a violação do SRP, as responsabilidades foram separadas em duas classes:

- ProcessadorEncomendas: Lida com entrada do usuário e cálculo do frete.
- GravadorEncomendas: Lida com a persistência dos dados em arquivo.

Código Depois

ProcessadorEncomendas

```java
package SSolid.Exemplo2;

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 = calcularFrete(peso);
            System.out.println("Valor do frete calculado: " + valorFrete);
            GravadorEncomendas gravador = new GravadorEncomendas();
            gravador.salvar(idEncomenda, valorFrete);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private double calcularFrete(double peso) {
        return peso * 10;
    }
}
```

GravadorEncomendas

```java
package SSolid.Exemplo2;

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

public class GravadorEncomendas {
    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();
        }
    }
}
```

Justificativa

- Responsabilidade Única: Cada classe agora tem uma única razão para mudar.
- Manutenção: Alterações no cálculo do frete ou na persistência podem ser feitas isoladamente.
- Reusabilidade: GravadorEncomendas pode ser usado por outras classes, se necessário.

Como Usar

- Copie esse código Markdown e cole no seu arquivo (ex.: README.md) no repositório público.
- Se quiser incluir a parte anterior ("Problema Identificado"), é só pedir que eu a converto também!
- Para os outros pacotes (OSOLID, LSOLID, ISOLID), você pode seguir o mesmo formato quando refatorá-los.

Está pronto para subir ao GitHub assim! Se precisar de mais ajustes, é só avisar.

Observações

- Mantive a estrutura exatamente como você passou, incluindo o título "GravadorEncomendas" e a seção "Justificativa".
- O bloco de código Java foi corretamente delimitado com ```java para destacar a sintaxe no Markdown.
- A seção "Como Usar" foi preservada como instrução final, formatada em Markdown.