**Lista de Exercícios – Princípios SOLID**

**1. **SRP – Single Responsibility Principle****

Explique, com suas palavras, o que significa o princípio da responsabilidade única.

R: É o principio que diz que toda classe precisa somente de uma resposabilidade para existir, muitas responsabilidades em uma classe pode gerar problemas futuros.

Quais problemas podem surgir quando uma classe possui múltiplas responsabilidades?

R: Manutencao de codigo pode ser mais dificultosa e problematica

**Exercício prático**  
Uma classe Funcionario realiza múltiplas operações: calcula salário, gera relatório de desempenho e envia e-mails de aniversário.

class Funcionario {

void calcularSalario() { /\* ... \*/ }

void gerarRelatorioDesempenho() { /\* ... \*/ }

void enviarEmailAniversario() { /\* ... \*/ }

}

Quantas responsabilidades essa classe possui?

R: 3 responsabilidades

Reestruture-a em classes separadas, cada uma com uma única responsabilidade.

public class Email { public class Relatorio {

void enviarEmail() {} void gerarRelatorio() {}

} }

public class Salario { public class Funcionario {

void calculaSalario() {} }

}

**2. OCP – Open/Closed Principle**

O que significa uma classe estar "aberta para extensão e fechada para modificação"?

R: significa nao mudar o core da funcionalidade dela e sim adicionar uma funcionaldade

Cite um exemplo de sistema onde esse princípio seria essencial.

R: navegador e extensoes

**Exercício prático**

class CalculadoraDesconto {

double calcular(String tipoCliente, double valor) {

if (tipoCliente.equals("VIP")) return valor \* 0.8;

if (tipoCliente.equals("Estudante")) return valor \* 0.9;

return valor;

}

}

Onde o OCP é violado?

R: Na criacao de ifs para adaptar para novos clientes, ao inves de adicionar ele modificou o metodo calcular

Reestruture o código de forma que seja fácil adicionar novos tipos de cliente sem modificar a classe principal.

public class DescontoVip {

double calcular(double valor) {

return valor \* 0.8;

}

}

public class DescontoEstudante {

double calcular(double valor) {

return valor \* 0.9;

}

}

**3. LSP – Liskov Substitution Principle**

O que significa "subtipos devem ser substituíveis por seus tipos base"?

Quais sinais indicam que o LSP está sendo violado em uma hierarquia de classes?

**Exercício prático**

class Ave {

void voar() { /\* ... \*/ }

}

class Pato extends Ave { /\* voa normalmente \*/ }

class Pinguim extends Ave { /\* não voa \*/ }

Qual é o problema com essa hierarquia?

R: Penguim é uma ave, mas ele nao voa, mas é obrigado a implementar o metodo voar

Refaça a hierarquia respeitando o LSP.

public class Ave {

}

public interface PodeVoar {

void voar();

}

public class Pato extends Ave implements PodeVoar {

@Override

public void voar() {

}

}

public class Penguim extends Ave {

}

**4. ISP – Interface Segregation Principle**

Por que interfaces grandes e genéricas podem prejudicar a reutilização do código?

R: porque as classes nao sao obrigadas a implementar todos os metodos que a interface oferece

Como o ISP ajuda a reduzir o acoplamento entre classes?

R: Separando os metodos em diversas interfaces

**Exercício prático**

interface Impressora {

void imprimir();

void escanear();

void enviarFax();

}

Quais problemas essa interface pode causar em classes que só imprimem (por exemplo, uma impressora térmica)?

R: Porque uma impressora termica nao e obrigada a implementar enviarFax nem escanear ja que ela nao faz essas funcoes

Reescreva as interfaces de forma mais coesa, aplicando o ISP.

public interface Escaner {

void escanear();}

public interface Fax {

void enviarFax();}

public interface Imprecao {

void imprimir();}

public class ImpressoraTermica implements Imprecao {

@Override

public void imprimir() {

}

}

**5. DIP – Dependency Inversion Principle**

Qual a diferença entre depender de classes concretas e depender de abstrações?

R: Porque depender de classes concretas pode gerar relacinamentos forte e fracos que podem dificultar mudancas e testes, ja depender de classes abstratas torna tudo mais flexivel

Como o DIP facilita a manutenção e a testabilidade de um sistema?

R: Com isso o sistema se torna muito mais facil de ser mantido e testado

**Exercício prático**

class EmailService {

void enviar(String mensagem) {

System.out.println("Enviando e-mail: " + mensagem);

}

}

class Notificacao {

private EmailService email = new EmailService();

void notificar(String mensagem) {

email.enviar(mensagem);

}

}

Responda:

Onde está a violação do DIP?

R: Na criacao de uma instancia de email, prendendo email a classe notificacao

Reestruture o código para que a classe Notificacao dependa de uma abstração, permitindo trocar o tipo de notificação (e-mail, SMS, push).

public interface NotificacaoService {

void enviar(String mensagem);

}

class EmailService implements NotificacaoService {

@Override

public void enviar(String mensagem) {

}

}

class SMSService implements NotificacaoService {

@Override

public void enviar(String mensagem) {

}

}

class Notificacao {

private final NotificacaoService servico;

public Notificacao(NotificacaoService servico) {

this.***servico*** = servico;

}

public void notificar(String mensagem) {

this.***servico***.enviar(mensagem);

}

}