Lista de Exercícios - Princípios SOLID e Boas Práticas em Java

1. Princípio da Responsabilidade Única (SRP)

1. Identifique os problemas de responsabilidade na classe abaixo e refatore-a para seguir o SRP:

```
class Relatorio {
    public void gerarRelatorio() {
        // Gera um relatório
    }
    public void salvarEmArquivo(String nomeArquivo) {
        // Salva o relatório em um arquivo
    }
}
```

2. Crie um exemplo de classe que viole o SRP e reescreva-o para que cada classe tenha uma única responsabilidade.

2. Princípio Aberto-Fechado (OCP)

3. O código abaixo viola o OCP. Reescreva-o para permitir novos descontos sem modificar a classe existente:

```
class CalculadoraDesconto {
    public double calcular(String tipoCliente, double valor) {
        if (tipoCliente.equals("comum")) {
            return valor * 0.95;
        } else if (tipoCliente.equals("vip")) {
            return valor * 0.90;
        } else {
            return valor;
        }
    }
}
```

4. Implemente um sistema de cálculo de imposto que permita adicionar novos tipos de impostos sem alterar as classes existentes.

3. Princípio da Substituição de Liskov (LSP)

5. O código abaixo viola o LSP. Qual é o problema? Corrija-o:

```
class Ave {
    public void voar() {
        System.out.println("Estou voando!");
    }
}
class Pinguim extends Ave {}
Pinguim p = new Pinguim();
p.voar();
```

6. Reescreva a hierarquia de classes para representar corretamente diferentes tipos de veículos (ex.: carro, bicicleta) sem violar o LSP.

4. Princípio da Segregação de Interfaces (ISP)

7. O código abaixo força classes a implementarem métodos desnecessários. Como refatorá-lo para seguir o ISP?

```
interface Trabalhador {
    void programar();
    void atenderCliente();
}
class Desenvolvedor implements Trabalhador {
    public void programar() {
        System.out.println("Escrevendo código...");
    }
    public void atenderCliente() {
        throw new UnsupportedOperationException("Desenvolvedores não atendem clientes.");
    }
}
```

8. Crie interfaces específicas para modelar corretamente uma cafeteria, onde baristas e atendentes possuem responsabilidades diferentes.

5. Princípio da Inversão de Dependência (DIP)

9. O código abaixo viola o DIP. Como corrigir isso?

```
class GmailSender {
    public void enviarEmail(String destinatario, String mensagem) {
        System.out.println("Enviando email para " + destinatario + ": " + mensagem);
```

```
    }
}
class EmailService {
    private GmailSender sender = new GmailSender();
    public void notificar(String destinatario, String mensagem) {
        sender.enviarEmail(destinatario, mensagem);
    }
}
```

10. Reescreva a implementação acima utilizando injeção de dependência para permitir o uso de diferentes serviços de envio de e-mail.