Atividade Prática: Princípios SOLID em Java

1- b) Uma classe deve ter apenas um motivo para mudar.

2- b) Princípio Aberto-Fechado (OCP)
3- a) A subclasse deve herdar todos os métodos da classe base e adicionar novos métodos sem alterar os existentes.

4- b) Criar interfaces específicas para que as classes implementem apenas os métodos que realmente utilizam.

5- c) Tanto classes de alto nível quanto de baixo nível devem depender de abstrações.

6- b) SRP - Princípio da Responsabilidade Única

7- b) Maior acoplamento entre as classes.

8- b) Princípio Aberto-Fechado (OCP)

9- c) LSP - Princípio da Substituição de Liskov

10- a) As classes clientes são forçadas a depender de métodos que não utilizam.
11- d) A classe UrnaEleitoral apresenta uma quebra do SRP, uma vez que possui responsabilidades que deveriam ser de componentes distintos do software.

12- Primeiro deve ser fazer uma analise do código e verificar as responsabilidades de cada classe, para resolver deverá dividir a classe para que não haja muita responsabilidade em uma única classe, fazer uma refatoração do código e rever os métodos da classe.

13- É necessário refatorá-la, dividindo-a em classes menores, cada uma com uma única função bem definida. Além disso, deve-se evitar métodos que realizam muitas operações diferentes e extrair responsabilidades para classes ou métodos auxiliares. Isso melhora a clareza

14- Quando uma classe depende apenas de métodos essenciais para sua funcionalidade, o código se torna menos acoplado, Isso melhora a manutenibilidade, pois as alterações podem ser feitas em módulos específicos

sem riscos de quebrar funcionalidades que não estão relacionadas. Além disso, torna o código mais legível

15- O acoplamento é indesejado em projetos orientados a objetos porque ele aumenta a dependência entre classes, o que torna o código mais difícil de manter, modificar e escale. Quando as classes estão fortemente acopladas, ou seja, quando uma classe depende excessivamente de outra, qualquer mudança em uma classe pode exigir alterações em muitas outras partes do sistema. Isso aumenta o risco de erros e quebra de funcionalidades, além de dificultar a compreensão e a evolução do código.

16- Para resolver o problema de acoplamento, é fundamental desacoplar as classes utilizando práticas como a injeção de dependência, onde as dependências são fornecidas de fora, e o uso de interfaces ou abstrações, permitindo que as classes dependam apenas de comportamentos e não de implementações específicas.

17- O Princípio Aberto-Fechado (OCP) nos ajuda a escrever classes mais flexíveis ao permitir que o comportamento de uma classe seja estendido sem modificar seu código original. Isso significa que, ao seguir esse princípio, uma classe deve ser "aberta" para extensão, mas "fechada" para modificação.
18- OCP e DIP é que o DIP facilita a aplicação do OCP. Ao depender de abstrações (e não de implementações concretas), o código se torna mais fácil de estender sem alterar a base existente, já que novas implementações podem ser adicionadas e substituídas sem afetar o comportamento das classes de alto nível. Assim, o DIP ajuda a criar a infraestrutura necessária para aplicar o OCP de forma eficaz.

19- Em vez de criar uma hierarquia rígida com herança, a composição permite que objetos sejam formados por diferentes comportamentos, tornando o código mais flexível e modular. Esse padrão evita os problemas de acoplamento e complexidade excessiva típicos da herança, permitindo que as classes compartilhem funcionalidades sem depender de uma estrutura hierárquica rígida.

20- O Princípio de Substituição de Liskov (LSP) afirma que objetos de uma classe base devem poder ser substituídos por objetos de suas subclasses sem alterar o comportamento esperado do sistema. Ou seja, uma subclasse deve herdar a classe base de forma que continue a cumprir as expectativas e invariantes da classe original, garantindo que o código que usa a classe base funcione corretamente também com suas subclasses. Isso assegura que a hierarquia de classes seja consistente e que as subclasses não violem o

contrato da classe base.

Explicação do código fonte: O código fornecido apresentava varias violações dos princípios de SOLID, dificultava a manutenção, extensão e teste do sistema.

A principal violação foi a de SRP, que é o principio da responsabilidade única além da violação do DRY, “Dont Repeat Yourself”

Agora Justificando o SOLID: Cada classe agora tem uma única responsabilidade bem definida, facilitando a manutenção e o teste.Agora é possível adicionar novas formas de calcular o total do pedido (por exemplo, adicionar descontos) sem modificar o código existente na classe Order. Garante que as subclasses de uma classe base possam ser usadas sem alterar a correção do programa. Promove a criação de interfaces mais coesas e evita que as classes implementem métodos que não utilizam. Facilita a troca de implementações e torna o sistema mais flexível. Embora não tenhamos usado interfaces explícitas neste exemplo, essa seria a abordagem em um sistema real.