SOLID

**O que é o SOLID?**

SOLID é um conjunto de princípios usados na arquitetura de software, que visa melhorar a organização, a manutenção e a escalabilidade dos sistemas.

**S - Princípio da Responsabilidade Única (Single Responsibility Principle):**

Este princípio afirma que um componente (como uma classe ou módulo) deve ter **apenas uma única responsabilidade**. Isso significa que cada função, objeto ou classe deve ser responsável por uma única tarefa ou conjunto de tarefas relacionadas.

Por exemplo, uma classe que gerencia clientes não deve ser responsável pelo controle de estoque. Cada classe ou componente deve ter suas responsabilidades bem definidas e isoladas.

Além disso, as funções dentro de uma classe devem ser **bem segmentadas** para que cada uma execute uma tarefa específica. Por exemplo:

* Uma função para cadastro,
* Outra para busca,
* Outra para listagem,
* E outra para exclusão.

Dessa forma, cada função se complementa e contribui para a divisão de responsabilidades da classe.

Benefícios:

* As classes e funções se tornam mais específicas e fáceis de testar.
* O código fica mais claro e menos propenso a erros.
* As mudanças em uma funcionalidade impactam apenas a parte específica do código responsável por ela.

**Mas onde começam e terminam as responsabilidades?**

Definir claramente o limite das responsabilidades de cada componente pode ser um desafio e muitas vezes envolve tentativas e erros. Uma dica útil é tentar expressar no nome da função tudo o que ela faz. Se o nome começar a ficar muito longo ou complexo, é um sinal de que a função possui responsabilidades demais e deve ser dividida em partes menores.

**O - Princípio do Aberto e Fechado (Open Closed Principle)**

O princípio diz que classes entidades ou funções devem estar abertas para extensão mas fechadas para modificação.

segundo compreendi imagine que as funções são ferramentas. para que possa usar uma ferramenta não deve desaprender de como usar outra. na teoria até que entendi, mas nao prática não sei como usar.

**L - Princípio da Substituição de Liskov (Liskov Substitution Principle)**

O Princípio da Substituição de Liskov sugere que as subclasses devem ser substituíveis pelas suas classes base sem alterar o comportamento esperado do programa. Seguir esse princípio ajuda a garantir que suas abstrações estejam bem definidas e que o código seja mais consistente.

Para entender melhor, imagine uma classe chamada Ave, que possui um método voar. Em seguida, você cria uma subclasse Pardal, que herda esse método, o que faz sentido, já que pardais podem voar. No entanto, se você criar uma subclasse Pinguim que também herda de Ave, não faz sentido que ela tenha um método voar, pois pinguins não voam. Isso indica que Pinguim não deveria ser uma subclasse de Ave, já que viola o comportamento esperado de todas as "aves" na hierarquia.

Em resumo, o Princípio da Substituição de Liskov recomenda que as subclasses não alterem o comportamento básico definido pela classe base. Um método fundamental não deve, em subclasses, ter comportamentos inesperados ou lançar exceções que a classe base não possua. Isso garante que o sistema seja previsível e funcional em todos os níveis de herança.

**I - Princípio da Segregação de Interface (Interface Segregation Principle)**

O Princípio da Segregação de Interface estabelece que as classes não devem ser forçadas a implementar métodos que não utilizam. Em outras palavras, é melhor criar interfaces mais específicas e adaptadas às necessidades de cada classe, ao invés de uma única interface com métodos que podem ser desnecessários para algumas implementações.

Por exemplo, se uma interface possui métodos que algumas classes não precisam, essas classes acabam sobrecarregadas com implementações inúteis, o que prejudica a coesão e a clareza do código. Esse princípio está diretamente ligado aos outros princípios SOLID, pois incentiva a criação de abstrações que realmente atendem às responsabilidades específicas de cada classe, mantendo o design modular e consistente.

**D - Princípio da Inversão de Dependência (Dependency Inversion Principle)**

1. **Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.**
2. **Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.**

Isso significa que, em vez de um módulo de alto nível depender diretamente de um módulo de baixo nível, ambos devem interagir por meio de uma abstração (como uma interface ou classe abstrata). Essa abstração permite que a implementação detalhada do módulo de baixo nível seja alterada sem afetar o módulo de alto nível.

Estruture um exercício onde a cada instrução o aluno aprende conceitos de SOLID.

O exercício deve conter uma série de instruções para a construção de um programa, mas onde a cada etapa os conceitos de SOLID vão sendo aplicados.