1. **Single Responsibility Principle (SRP)**

* **Definição**: O Princípio da Responsabilidade Única estabelece que uma classe deve ter uma única responsabilidade, ou seja, deve fazer apenas uma coisa. Se uma classe tem múltiplas responsabilidades, isso pode levar a um código que é mais difícil de entender e manter, pois uma mudança em uma parte do código pode afetar outras partes de maneira inesperada. O SRP ajuda a minimizar o acoplamento entre classes e melhora a coesão, facilitando a compreensão e a manutenção do código.



**2. Open/Closed Principle (OCP)**

* **Definição**: O Princípio da Abertura/Fechamento afirma que as entidades de software (classes, módulos, funções, etc.) devem ser abertas para extensão, mas fechadas para modificação. Isso significa que você deve ser capaz de adicionar novas funcionalidades ao seu sistema sem alterar o código existente. Essa abordagem é fundamental para reduzir o risco de introduzir bugs em funcionalidades já existentes e promove a reutilização de código. O uso de abstrações, como interfaces ou classes abstratas, é comum para permitir essa extensão.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

**3. Liskov Substitution Principle (LSP)**

* **Definição**: O Princípio da Substituição de Liskov afirma que as subclasses devem ser substituíveis por suas classes base sem alterar o comportamento correto do programa. Isso significa que, se você tem uma classe base e uma classe derivada, a classe derivada deve ser capaz de ser usada no lugar da classe base sem que o comportamento do sistema seja comprometido. O LSP ajuda a garantir que as hierarquias de classes sejam usadas corretamente e que o código que utiliza essas classes seja flexível e fácil de entender.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

**4. Interface Segregation Principle (ISP)**

* **Definição**: O Princípio da Segregação de Interfaces sugere que é melhor ter várias interfaces específicas do que uma única interface abrangente. Isso ocorre porque as classes não devem ser forçadas a implementar métodos que não utilizam. Ao criar interfaces menores e mais focadas, você melhora a coesão e a clareza do seu código, além de permitir que os clientes utilizem apenas as funcionalidades que realmente precisam. Isso também facilita a adaptação e evolução das classes que implementam essas interfaces.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

**5. Dependency Inversion Principle (DIP)**

* **Definição**: O Princípio da Inversão de Dependência afirma que as classes de alto nível não devem depender de classes de baixo nível, mas ambas devem depender de abstrações. Além disso, as abstrações não devem depender de detalhes, mas os detalhes devem depender de abstrações. Esse princípio promove um design onde as dependências são invertidas, facilitando a troca de implementações concretas sem afetar o código que as utiliza. O DIP é fundamental para alcançar uma arquitetura desacoplada, permitindo uma melhor testabilidade e flexibilidade.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente