# PRINCÍPIOS SOLID - Resumo baseado no PDF fornecido.

Estes princípios ajudam a criar um design de software flexível, sustentável e menos propenso a erros, facilitando a manutenção e evolução do sistema.

1. **SRP - Single Responsibility Principle (Princípio da Responsabilidade Única):**

Esse princípio define que uma classe deve ter apenas uma razão para mudar, o que significa que deve ter apenas uma responsabilidade. Por exemplo, o PDF ilustra o SRP como um jogo de boliche, inicialmente, a classe “Game” tratava tanto do acompanhamento de frames quanto do cálculo da pontuação. A refatoração separou essas responsabilidades em classes distintas “Game” e “Scorer”, tornando o design mais manutenível e menos propenso a erros quando os requisitos mudam. A ideia central é que cada responsabilidade representa um "eixo de mudança". Se uma classe tem múltiplas responsabilidades, mudanças em uma área podem acidentalmente quebrar outras. O exemplo de uma classe “Rectangle” com métodos de desenho e cálculo de área ilustra os problemas causados pela violação do SRP. Separar essas responsabilidades em classes “GeometricRectangle” e “Rectangle” resolve o problema. Separar responsabilidades é um aspecto crucial do design de software.

2. **OCP - Open/Closed Principle (Princípio do Aberto/Fechado):**

O Princípio Aberto-Fechado (OCP) defende que entidades de software devem estar abertas para extensão, mas fechadas para modificação. Segundo esse princípio, módulos de software devem estar abertos para extensão, mas fechados para modificação. Isso significa que novos recursos podem ser adicionados sem alterar o código existente, minimizando o risco de introduzir erros. Os mecanismos principais do OCP são abstração e polimorfismo. O padrão “Strategy” e o padrão “Template Method” ilustram como aplicar o OCP.

- O padrão Strategy usa uma classe base abstrata “ClientInterface” que define uma interface comum. Diferentes implementações (classes Server) podem ser criadas sem modificar o código do cliente.
- No padrão Template Method, a classe base “Policy” define uma estrutura de algoritmo, enquanto as subclasses implementam etapas específicas. O aplicativo “Shape”, uma abordagem procedural para desenhar formas viola o OCP, pois adicionar uma nova forma exige a modificação da função “DrawAllShapes”. Uma abordagem orientada a objetos, usando uma classe base abstrata “Shape”, demonstra como atender ao OCP adicionando novas classes de forma sem modificar a função principal de desenho.

No entanto, o fechamento completo contra todas as possíveis mudanças é irrealista. O OCP deve ser aplicado estrategicamente, focando em mudanças prováveis. Uma abordagem de "levar o primeiro tiro" é mais interessante, onde inicialmente um código simples é escrito, e abstrações são adicionadas apenas quando uma mudança necessária ocorre. Para antecipar mudanças necessárias, usar desenvolvimento orientado a testes é mais indicado, ciclos de desenvolvimento curtos e feedback frequente das partes interessadas.

3. **LSP - Liskov Substitution Principle (Princípio da Substituição de Liskov):**

O Princípio da Substituição de Liskov (LSP) garante que subtipos sejam substituíveis por seus tipos base. Isso significa que objetos de uma classe derivada devem poder substituir objetos de sua classe base sem quebrar a lógica do programa. Se um subtipo causa comportamento inesperado em uma função projetada para a classe base, ele viola o LSP. Isso frequentemente resulta no uso de Informação de Tipo em Tempo de Execução (RTTI), violando o OCP. Um exemplo envolve uma função "DrawShape" viola o OCP porque deve tratar explicitamente cada tipo de forma. Outro exemplo mostra uma violação mais sutil envolvendo classes "Rectangle" e "Square". Embora pareça razoável derivar "Square" de "Rectangle", isso leva a problemas porque modificar a largura de um Square também muda sua altura, violando a suposição de independência que uma função que usa "Rectangle" pode fazer. Um design auto consistente em isolamento pode ainda falhar em atender às expectativas do cliente (suposições razoáveis). O relacionamento IS-A em POO é sobre comportamento, não apenas a estrutura superficial. Design por Contrato (DBC) é uma maneira de tornar essas suposições explícitas. DBC envolve declarar pré-condições e pós-condições para cada método, ajudando a garantir que os subtipos aderem ao contrato de sua classe base. Testes unitários também são apresentados como uma maneira eficaz de especificar e validar esses contratos.

4. **DIP - Dependency Inversion Principle (Princípio da Inversão de Dependência):**

Esse princípio indica que módulos de alto nível não devem depender de módulos de baixo nível, ambos devem depender de abstrações. Abstrações não devem depender de detalhes, detalhes devem depender de abstrações. Este princípio inverte a estrutura de dependência tradicional encontrada na programação procedural, onde módulos de alto nível são fortemente acoplados com implementações de baixo nível. Uma heurística simples para aplicar o DIP é "Dependa de abstrações". Isso significa que variáveis, classes e métodos não devem depender diretamente de classes concretas, mas sim de interfaces abstratas. Essa heurística tem exceções (por exemplo, classes não voláteis como String), mas tem sua importância para gerenciar a volatilidade na maioria das classes específicas do aplicativo. Um exemplo simples de Button/Lamp ilustra os benefícios do DIP, contrastando um design ingênuo com um design que usa uma interface abstrata “ButtonServer”, permitindo que o botão controle vários dispositivos sem modificação. Concluindo, o DIP é muito importante para a criação de software robusto, reutilizável e manutenível, particularmente no contexto de frameworks.

5. **ISP - Interface Segregation Principle (Princípio da Segregação de Interfaces):**

O ISP aborda a questão de interfaces "gordas", onde uma classe tem métodos usados por diferentes conjuntos de clientes. Tais interfaces levam a dependências e fragilidades desnecessárias. Poluição de interface é um dos principais pontos que cooperam para a necessidade de aplicar o ISP, como no exemplo: Adicionar um recurso de temporização à classe "Door" necessita do uso de um Timer e de uma interface "TimerClient". Uma solução ingênua que faz "Door" herdar de "TimerClient" leva a problemas porque nem todas as implementações de porta precisam de temporização. A "força inversa" exercida pelos clientes sobre as interfaces, pode levar a mudanças na interface, e como isso pode causar problemas. O cerne do ISP é que os clientes não devem ser forçados a depender de métodos que não

usam. Os clientes não precisam interagir diretamente com a interface de um objeto. A delegação ou herança múltipla são boas soluções possíveis para este problema.

- Delegação: É uma técnica em que um objeto (delegante) passa a responsabilidade de uma tarefa a outro objeto (delegado). Em vez de herdar diretamente os métodos, isso evita uma ligação rígida entre classes e promove a composição sobre a herança. Segregar a interface em interfaces específicas do cliente melhora o design, reduzindo o impacto das mudanças.
- Herança múltipla: Permite que uma classe herde características de mais de uma classe base. A herança múltipla permite que uma classe combine o comportamento de múltiplas classes, mas pode causar complexidade adicional.