

## 12. Os princípios SOLID de design orientado a objetos

O desenvolvimento de Software orientado a objetos tem como objetivo principal produzir sistemas mais compreensíveis, flexíveis e de fácil manutenção. Entretanto, quando mal estruturado, o código pode trazer problemas como:

- **Rigidez** - Quando uma alteração em uma parte do programa pode quebrar a outro.
- **Fragilidade** - Quando as coisas quebram em lugares não relacionados.
- **Imobilidade** - Quando o código não pode ser reutilizado fora do seu contexto original.

Para resolver isso, Robert C. Martin, formulou os princípios SOLID, que são hoje um pilar do desenvolvimento O.O.

### 1) SRP: Princípio da Responsabilidade Única

"Uma classe deve ter apenas um motivo para mudar". Isso significa que cada classe ou método deve ter responsabilidade sobre uma única parte da funcionalidade do sistema.

• **Problema**: Uma classe chamada "Calculadora" era responsável por somar as áreas de várias formas e imprimir o resultado total no tel.

• **Mudança**: Se houvesse a necessidade de alterar o formato de saída, a classe que continha o lógico de cálculo, precisaria ser modificada, recompilada e reimplantada. As 2 responsab. estariam acopladas.

• **Solução**: A responsab. de saída foi para classe "OutputArea", agora ela apenas calcula e a outra recebe o resultado, podendo ser alterada sem impacto na outra classe.

## 2) OCP: Princípio de Abertura / Fechamento

Esse princípio estabelece que as classes devem estar abertas para extensões, mas fechadas para modificação. Isso significa que, ao invés de alterar diretamente uma classe já existente para adicionar novos comportamentos, deve-se projetar o sistema de forma que seja possível estender suas funcionalidades por meio de subclasses ou composição. Um exemplo é o uso de uma abstrato de formas geométricas, onde novos tipos de formas podem ser adicionados sem modificar o código já escrito. Dessa forma, o sistema pode crescer sem comprometer o que já funciona.

## 3) LSP: Princípio de Substituição de Liskov

Defende que subclasses devem poder substituir suas superclasses sem causar efeitos inesperados. Em termos práticos, isso significa que, se um código funciona com uma classe mãe, também deve funcionar com qualquer uma de suas subclasses. Um caso clássico de violação é o do quadrado tratado como subclasse do retângulo: embora matematicamente faça sentido, em código isso gera inconsistência, pois um quadrado não se comporta exatamente como um retângulo. A solução é tratar cada um como classes distintas, evitando heranças que comprometam a lógica.

## 4) IPS: Princípio de Segregação de Interface

Esse princípio diz que é melhor ter várias interfaces pequenas e específicas do que uma única interface grande e genérica. Quando uma interface obriga uma classe a implementar métodos que ela não precisa, cria-se um problema de design. Para resolver isso, cada comportamento deve ser definido em uma interface própria, permitindo que as classes implementem apenas o que realmente necessitam.

## DIP - Princípio da Inversão de Dependência

Estabelece que o código deve depender de abstrações e não de implementações concretas. Em outros palavras, classes de alto nível não devem depender diretamente de classes de baixo nível, mas sim de contratos definidos por interfaces ou classe abstratas. Isso permite que o comportamento do sistema seja modificado um tempo de execução por meio da injeção de dependências, sem precisar alterar o código já existente. Dessa forma, componentes ficam mais desacoplados e podem ser reutilizados em diferentes contextos.

- Inversão de dependências - O princípio de inverter as dependências
- Injeção de dependências - O ato de inverter as dependências
- Injeção de construtor - Executor injeção de dependência pelo construtor
- Injeção de parâmetros - Realize injeção de dependência por meio dos parâmetros de um método, como um setter.

## Conclusão

Os princípios SOLID funcionam como diretrizes que ajudam a criar sistemas mais organizados, reutilizáveis e fácil manutenção. Aplicados corretamente, eles reduzem o acoplamento e aumentam a coesão, permitindo que o software evolua de forma mais eficiente e segura.