Documentação de Princípios SOLID - by Joaquim Guilherme

Definição

Os princípios SOLID podem ser definidos como 5 boas práticas que facilitam a manutenção e expansão de um software, tornando-se cruciais para o bom desenvolvimento de programas orientados a objetos (POO).

Eles são subdivididos em:

S - Single Responsibility Principle

O - Open/Closed Principle

L - Liskov Substitution Principle

I - Interface Segregation Principle

D - Dependency Inversion Principle

Podem ser aplicados em qualquer linguagem de programação que utilize este paradigma (POO).

Definição de POO

Pode ser definido como um paradigma baseado no conceito de “objetos”, que são representados por instâncias e abstrações de conceitos do mundo real, e funcionam como elementos para a construção de soluções.

O paradigma permite organizar o código em elementos chamados de objetos, que são frutos da instanciação de uma classe, e possuem seus próprios atributos e métodos.

Relação entre POO e SOLID

O desenvolvimento de programas orientados a objetos, alinhado com os princípios definidos pelo SOLID, permitem o desenvolvimento de programas modulares, flexíveis, abstratos e principalmente, de fácil manutenção e compreensão. Estes fatores são cruciais quando o objetivo é a construção de softwares que exigem qualidade, manutenção e alta complexidade.

S - Single Responsibility Principle

O primeiro princípio, traduzido para Princípio da Responsabilidade Única, define que as classes devem ter responsabilidades únicas, ou seja, cada classe deve ter um, e somente um, motivo para mudar.

Se a classe agrupa diversas responsabilidades, simples alterações, como mudar requisitos de um projeto podem desencadear em diversas modificações na classe, o que torna mais difícil a manutenção e reutilização do código.

Solução para o SRP

Com a finalidade de manter a classe com uma responsabilidade única, é necessária a análise da classe e definição de todas as responsabilidades que ela agrupa. Depois, devemos construir novas classes, que, baseadas na análise posterior, terão suas respectivas responsabilidades únicas.

Dessa forma, o princípio é respeitado, a medida que o código se torna propício a fácil manutenção, reutilização e compreensão.

O - Open/Closed Principle

Traduzido para Princípio do Aberto/Fechado, define que uma classe deve estar aberta a extensões, mas fechada a modificações.

Quando uma classe está aberta a modificações, a adição de novas funcionalidades a torna cada vez mais complexa e acoplada.

Dessa forma, o ideal é adaptar o código para que essa classe não precise ser alterada, mas esteja aberta a extensão de novos recursos, tornando o código mais semelhante ao mundo real, praticando de maneira sólida a orientação a objetos.

Solução para o OCP

Visando o desacoplamento do código, o melhor caminho para a aplicação de códigos abertos a extensão, mas fechados a modificações é a abstração do código.

Com a criação de interfaces e classes abstratas, e aplicando as estratégias implements e extends, o código se torna extensível, possibilitando que a adição de novas funcionalidades não exijam alterações diretas no objeto de origem.

Assim, o código se torna muito mais flexível, fácil de ler, e incentiva, indiretamente, a aplicação de Design Patterns, neste caso, o Strategy, alinhando várias boas práticas de desenvolvimento.

L - Liskov Substitution Principle

O Princípio de Substituição de Liskov implica que, quando necessário , uma classe-filha (herança) deve ser capaz de substituir e reproduzir todos os comportamentos da sua classe-mãe.

Dessa forma, sempre que implementamos subclasses que estendem superclasses, devemos garantir que sejam capazes de substituir a classe mãe. Caso um método da classe-filha tenha um retorno muito diferente do da classe mãe, ou lance uma exceção, são sinais de que o princípio está sendo violado.

Solução para o LSP

Para evitar a violação deste princípio, deve sempre ocorrer a análise do comportamento proposto pela superclasse e do comportamento implementado nas subclasses.

Quando se trata de LSP, a maioria das violações ocorre na tentativa de implementação de um novo método na classe-mãe, que acaba por romper o padrão já estabelecido e estendido para classes-filhas já existentes.

Dessa forma, quando violado, a melhor solução é a criação de novas subclasses, de forma que, ao invés de um rompimento do padrão estabelecido na superclasse com a criação de um novo método ou funcionalidade, seja criada uma subclasse, que manterá o comportamento adequado, e a nova implementação específica para essa classe.

I - Interface Segregation Principle

O Princípio de Segregação de Interface implica que nunca devemos implementar uma interface genérica, mas sim específica, independente do tamanho do código.

Dessa forma, o princípio afirma que uma classe nunca deve ser obrigada a implementar métodos e interfaces que não se apliquem para o seu propósito ou comportamento.

Solução para o ISP

Assim, para evitar a violação do ISP devemos sempre nos atentar a implementação de interfaces específicas, visando que cada classe cumpra apenas os contratos que obedecem seus comportamentos e propósitos.

Portanto, é uma boa prática sempre evitar a utilização de interfaces genéricas, promovendo a coesão e flexibilidade dos sistemas.

D - Dependency Inversion Principle

Definido como Princípio da Injeção de Dependência, ele determina que os códigos devem ser desacoplados, ou seja, módulos de alto nível não devem depender de módulos de baixo nível, ambos devem depender de abstrações.

Assim, uma classe deve depender de classes abstratas e interfaces, mas não de classes concretas.

Solução para o DIP

Para seguir este princípio, devemos sempre analisar se uma classe depende diretamente da instância(classe concreta) de outra classe.

Dessa forma, quando encontradas dependência concretas, devemos, a partir da implementação de interfaces ou classes abstratas, abstrair estas dependências (injeção de dependência), tornando o código desacoplado e flexível a novas extensões.

Injeção de Dependência X Inversão de Dependência

Injeção de Dependência e Inversão de Dependência são DISTINTOS, mas se relacionam diretamente.

Enquanto a Inversão de Dependência se trata apenas de um conceito, a Injeção de Dependência é uma prática.

A Inversão de Dependência se refere diretamente ao conceito de que classes de alto e de baixo nível devem sempre depender de abstrações.

Já a injeção de dependência está diretamente ligada à aplicação deste conceito. Ela é a implementação para instanciar as classes após a realização da abstração.

Portanto, a injeção refere-se ao ato de entregar dependências a um objeto, em vez de o próprio objeto definir suas dependências (classes concretas), o que violaria o princípio.