**SOLID** é um acrônimo criado por *Michael Feathers,* e consiste em cinco princípios da programação orientada à objetos, que ajudam o desenvolvedor a separar responsabilidades, diminuir acoplamentos e refatorar mais facilmente seus códigos.

**1 - SRP: Single Responsability Principle:** Princípio da responsabilidade única. De acordo com ele, uma classe deve ter um, e apenas um, motivo para mudar. Além disso, a classe deve ser especialista em um único assunto e possuir uma responsabilidade única dentro do software, ou seja, uma única ação para executar.

Quando damos à uma classe muita responsabilidade e acabamos criando uma classe que faz de tudo, a chamamos de *god class*, e devemos evitar a criação delas o máximo possível pois, por mais que pareçam boas em um primeiro momento, em um longo prazo será difícil modificá-la sem comprometer outras classes.

A violação desse princípio pode levar à alguns problemas, tais como a falta de coesão, alto acoplamento e dificuldade na implementação de testes automatizados (por ser difícil “mockar” uma *god class)*.

O princípio não se limita somente a classes, podendo ser aplicado também em métodos, ou seja, tudo que é responsável por executar uma ação, deve ser responsável apenas por aquilo que se propõe a fazer.

**2 – OCP – Open Closed Principle:** Princípio aberto-fechado. De acordo com ele, objetos ou entidades devem ser abertos para extensão, porém fechados para modificação. Ou seja, quando novos comportamentos e recursos precisam ser adicionados ao software, devemos estender e não alterar o código original.

Para adicionarmos uma nova implementação sem precisar mudar algo já existente, separamos o comportamento extensível por trás de uma interface e, então invertemos as dependências (exemplo mostrado no código do projeto).

Esse princípio é a base para o padrão *strategy* e tem como vantagem, principalmente, a facilidade na adição de novos requisitos, com menor chance de introduzir *bugs*, pois o comportamento fica isolado e, o que funcionava anteriormente continuará funcionando.

**3 – LSP – Liskov Substitution Principle:** Princípio da substituição de Liskov. De acordo com ele, uma classe derivada deve ser substituível por sua classe base. Basicamente, *Barbara Liskov* diz que se S é um subtipo de T, em um programa, os objetos de T podem ser substituídos por objetos de S sem que seja necessário alterar as propriedades deste programa.

Alguns exemplos de violação do LSP são: Sobrescrever/implementar um método que não faz nada, lançar uma exceção inesperada e retornar valores de tipos diferentes da classe base.

Ao sobrescrevermos um método que nada faz, por exemplo, estamos violando o princípio pois, ao chamarmos o mesmo método para as duas classes diferentes, uma derivada da outra, obteremos retornos de tipos diferentes, podendo quebrar o código. O mesmo acontece quando lançamos uma exceção inesperada ou retornamos valores de tipos diferentes da nossa classe base, tornando nosso código mais suscetível a problemas no futuro, se precisarmos substituir uma classe por sua classe derivada.

Para não violarmos o LSP, além de estruturar bem as abstrações, em alguns casos será necessário utilizar injeção de dependência e também utilizar bem outros princípios do SOLID. Além disso, o LSP nos permite utilizar o polimorfismo com mais confiança, pois podemos chamar nossas classes derivadas referindo-se à classe base sem preocupações.

**4 – ISP – Interface Segretation Principle:** Princípio da segregação da Interface. De acordo com ele, uma classe não deve ser forçada a implementar interfaces e métodos que não irá utilizar. Basicamente, é melhor criar interfaces mais específicas do que uma única interface genérica.