Defina o que é cada princípio do S.O.L.I.D:

1) Single Responsibility – Princípio da Responsabilidade Única

Ao criar uma classe e/ou método, a estrutura deve ter uma única responsabilidade, clara e bem definida, para centralizar as responsabilidades e pontos de mudança em lugares centrais e fáceis de serem identificados, tornando a manutenção do código muito mais simples.

Exemplo: Utilizando o trabalho do PetShop dado em aula, supondo que em um memos métodos a gente tivesse o banho, o atendimento clínico e a vacinação.

O método estaria fazendo tudo, logo, para realizar qualquer ponto de mudança, seria necessário alterar todo o método. Fica claro que está sobrecarregado.

Tem-se ainda o problema de criar um cenário automatizado para receber vacina, sem passar pelo banho ou atendimento clínico, por exemplo, caso tenha algum erro no banho, a execução do teste vai parar e mostrar um erro em vacina, o que caracteriza um falso negativo.

Sendo assim aplicando o Single Responsibility, o ideal é criar um método para cada funcionalidade.

2) Open-Closed – Princípio Aberto-Fechado

Objetos ou entidades devem estar abertos para extensão, mas fechados para modificações, ou seja, quando novos comportamentos e recursos precisam ser adicionados no software, devemos estender e não alterar o código fonte original.

Exemplo: Supondo que no trabalho sobre PetShop tivéssemos somente uma classe responsável pelo cadastro de animais do tipo cachorro. Passado algum tempo, o PetShop decide receber repteis.

Neste caso seria necessário modificar a classe, visto que um réptil possui características e precisa de cuidados diferentes de um cachorro.

Aparentemente a inserção dos novos campos na classe seria mais simples, porém, ao editar uma classe já existente aumenta os riscos de introduzir bugs em algo que funcional.

Aplicando o Open-Closed, o ideal seria criar um comportamento extensível atrás de uma interface, ou seja, criar uma interface com nome ‘CaracteristicaAnimal’ contendo método ‘caracteristica()’ e fazer com que as classes referente aos animais Gato, Cachorro, implementem essa interface. Assim conseguimos colocar regras de características para cada classe, dentro do método ‘caracteristica()’, fazendo com que a classe ‘Animal’ dependa somente da interface criada.

Sendo assim, a classe ‘Animal’ não precisa saber quais métodos chamar pra adicionar características do animal.

Qualquer nova espécie que o PetShop decida atender no futuro, desde que seja implementada a interface CaracteristicaAnimal, as características serão preenchidas sem a necessidade de alterar o código fonte.

3) Liskov Substitution – Princípio da Substituição de Liskov

Uma classe derivada deve ser substituível por sua classe base.

Se para cada objeto *O1*, do tipo S, há um objeto *O2*, do tipo T, de tal forma que um programa P definido em termos de T, o comportamento de P não é alterado quando *O1* é substituído por *O2*. Então S é um subtipo de T.

Resumindo, se tenho duas classes Animal e Cachorro e ao passá-las para um código, nada precisa ser alterado nesse código, então elas são subtipos uma da outra. Ou seja, o programa que recebe esses objetos não pode precisar saber qual o tipo exato ele está recebendo e tão pouco precisar ser modificado por este motivo.

Além de estruturar bem as abstrações, em alguns casos é necessário usar a injeção de dependência e usar outros princípios do SOLID, como por exemplo, o Open-Closed Principle e o Interface Segregation Principle.

Seguir o LSP nos permite usar o polimorfismo com mais confiança. Podemos chamar nossas classes derivadas referindo-se à sua classe base sem preocupações com resultados inesperados.

4) Interface Segregation – Princípio da Segregação da Interface

Uma classe não deve ser forçada a implementar interfaces e métodos que não irão utilizar. Basicamente diz que é melhor criar interfaces mais específicas ao invés de termos uma única interface genérica.

Exemplo: Supondo que no nosso projeto do PetShop vamos receber aves, sendo assim, criamos uma interface ‘Aves’, para abstrair as características desse animais e em seguida as classes criadas devem implementar essa interface.

O pinguim é uma ave, implementar um método ‘setAltitude()’ na classe ‘Pinguim’ da interface ‘Aves’ seria uma violação do Interface Segregation Principle e do LSP, pois pinguins não voam. O ideal então, seria criar interfaces mais específicas.

Uma solução para o problema no exemplo, seria criar uma nova interface com nome ‘Aves que voam’ e adicionar o método ‘setAltitude()’ nela, assim conseguimos respeitar o princípio de segregação das interfaces, isolando as características de forma correta.

5) Dependency Inversion – Princípio da Inversão de Dependência

Um módulo de alto nível não deve depender de módulos de baixo nível, ambos devem depender da abstração. Abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações. Sintetizando: Dependa de abstrações e não de implementações.

Depender de abstrações permite que classes dependam menos umas das outras, tornando-as bem definidas e reduzindo a necessidade de modificá-las devido a alterações em outras classes. Facilita o reuso, pois depende menos de detalhes de implementação e reduz a necessidade de código duplicado para os casos excepcionais onde é muito difícil reaproveitar o código. Por fim, é um bom aliado na hora de entender o código e testar, porém se o nível de abstração foi muito alto o aumenta o tempo de compreensão e debug de algum problema no código.

Componentes que formam o sistema são simplificados, de forma que um saiba o mínimo possível sobre o outro, o que consequentemente aumenta as chances de reaproveitamento de trechos de código e das definições de suas funções.

Fontes:

https://www.treinaweb.com.br/blog/principios-solid-single-responsability-principle
https://medium.com/desenvolvendo-com-paixao/o-que-%C3%A9-solid-o-guia-completo-para-voc%C3%AA-entender-os-5-princ%C3%ADpios-da-poo-2b937b3fc530
https://www.blogdoft.com.br/2020/03/15/solid-de-verdade-liskov-substitution-principle-lsp/
https://www.blogdoft.com.br/2020/02/03/solid-de-verdade-dependency-inversion-principle-dip/
https://campuscode.com.br/conteudos/s-o-l-i-d-principio-de-inversao-de-dependencia