TIC em trilhas

Programação Orientada a Objetos

Módulo 3

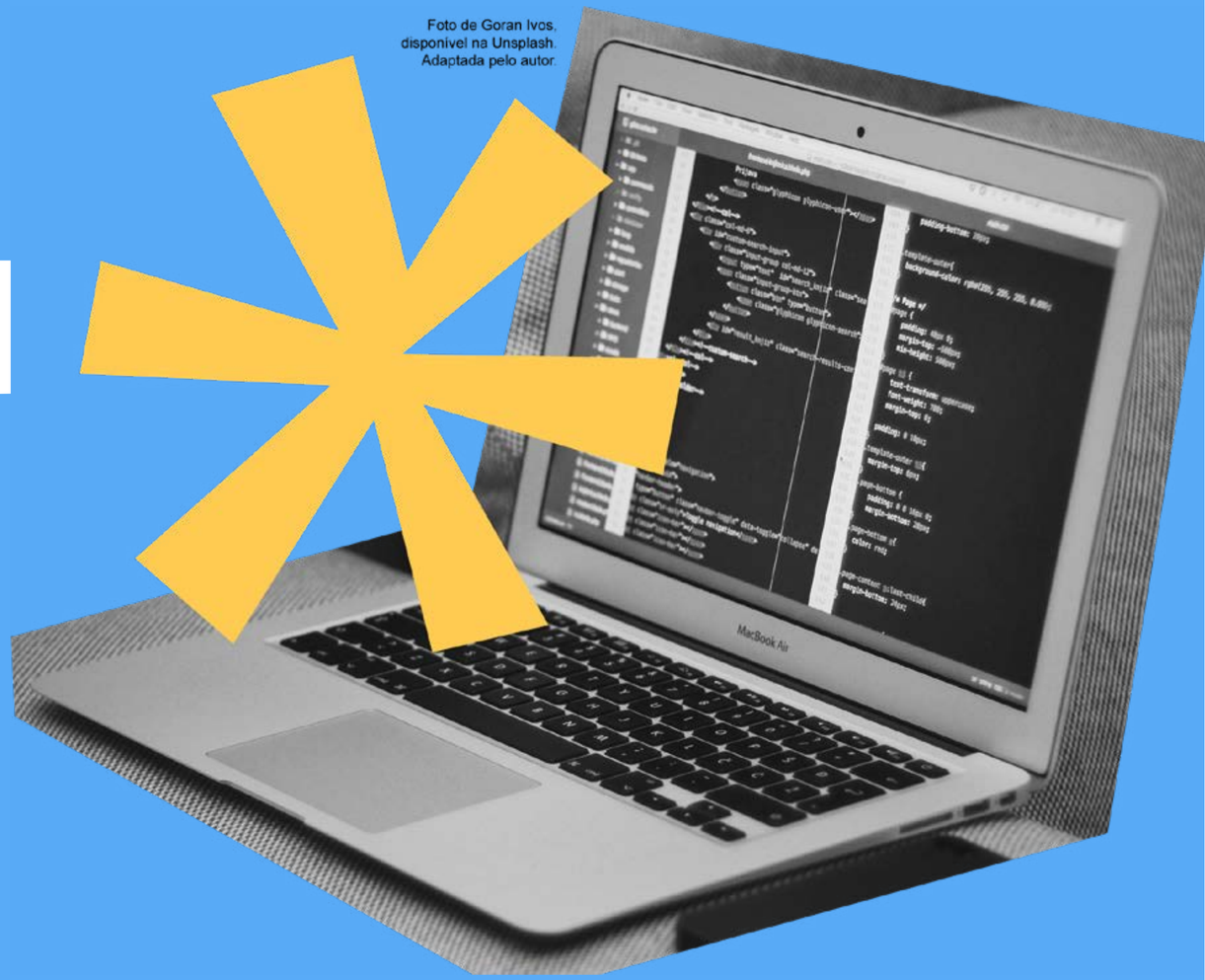


Foto de Goran Ivos, disponível na Unsplash. Adaptada pelo autor.

Sumário

Ilustração feita por upklyak, disponível no Freepik. Adaptada pelo autor.

SOLID

É o acrônimo de cinco princípios de arquitetura de software que embasam o desenvolvimento de código. Esses princípios embasam os conceitos de fundamentos de orientação a objetos vistos nesta trilha.

Estes princípios foram introduzidos por Robert C. Martin e tem como objetivo ajudar a estruturar o código e a compreender o código de outras pessoas, bem como dar manutenção nos softwares.

Esses princípios definem as responsabilidades de cada classe individual e entre as classes, influenciando a maneira como elas interagem umas com as outras.

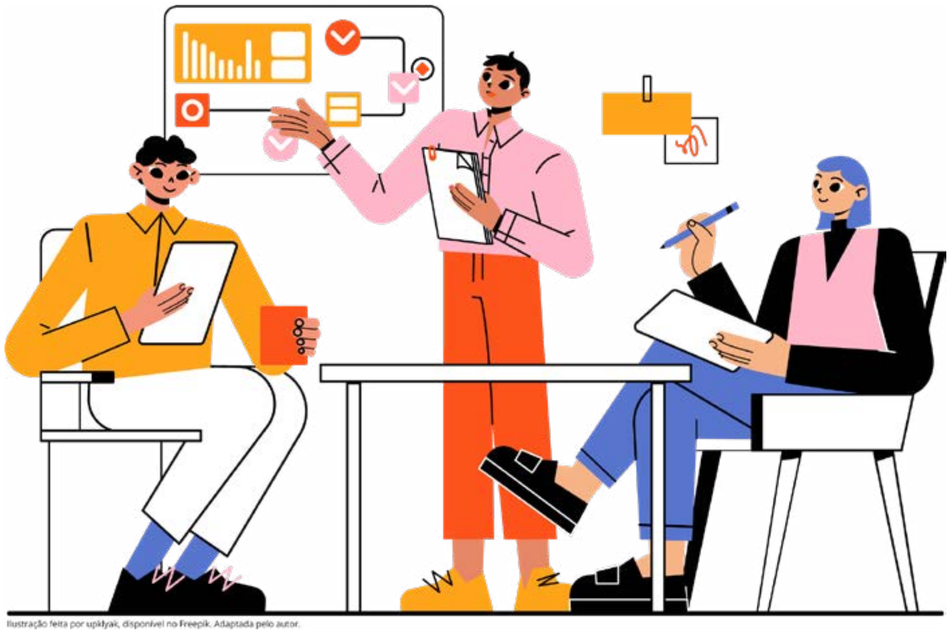


Ilustração feita por upklyak, disponível no Freepik. Adaptada pelo autor.

Esses princípios são os seguintes:

- Princípio da Responsabilidade única - Single Responsibility Principle
- Princípio Aberto e Fechado - Open Closed Principle
- Princípio da Substituição de Liskov - Liskov Substitution Principle
- Princípio da Segregação de Interface - Interface Segregation Principle
- Princípio da Inversão de Dependência - Dependency Inversion Principle

Cada um desses princípios são associado as seguintes letras respectivamente: S, O, L, I, D resultando no acrônimo SOLID.

Barbadinha! Vamos identificar cada um desses princípios para entender melhor onde eles se encaixam dentro do que já vimos.

Princípio da Responsabilidade Única (S)

Significa que uma classe deve ter somente uma responsabilidade dentro de sua construção. Basicamente, uma classe deve representar características e ações de apenas uma coisa.

Por exemplo, a classe responsável por conectar ao banco de dados, não deve ser a mesma que gera números randômicos.

Parece bem nítido que são coisas diferentes. As etapas de código para realizar uma conexão com um banco de dados realmente não tem nada a ver com as etapas de código que geram números aleatórios.

Claro que essa separação nem sempre é tão obvia, mas é a prática que vai te mostrando qual a melhor forma de construir o código, e quando necessário refatore, não há problemas nisso.

Princípio Aberto e Fechado (O)

Significa que uma classe, módulo ou função deve possibilitar extensões e ao mesmo tempo não permitir modificações. O objetivo é poder alterar um código que já esta estável e funcionando, sem necessidade de mudar o que já foi desenvolvido.

Basicamente, quando temos um método que testa um parâmetro e conforme o valor do parâmetro decide sobre um caminho ou outro (if else if else if else). Caso necessite acrescentar novas situações de teste, acabamos por ter de modificar o código já existente, acrescentando um outro else ao código, por exemplo. Essa é uma situação a ser evitada.

Ilustração feita por upklyak, disponível no Freepik. Adaptada pelo autor.

Princípio da Substituição de Liskov (L)

Propõe que uma classe filha (derivada) devem ser substituíveis por suas classes base (mãe) sem gerar erros de execução que possa ‘quebrar’ o código, ou ter um comportamento incorreto.

Analisa esta situação. Uma classe quadrado que herda de uma classe retângulo.

```
class Retangulo:
    def __init__(self, altura, largura):
        self.altura = altura
        self.largura = largura
    def get_area(self):
        return self.altura * self.largura

class Quadrado(Retangulo):
    def __init__(self, lado):
        super().__init__(lado, lado)

retangulo = Retangulo(5, 4)
print(retangulo.get_area()) # saída retângulo

quadrado = Quadrado(4)
print(quadrado.get_area())  # saída quadrado
```

Consegue perceber que saída retângulo e saída quadrado podem gerar resultados diferentes?

A forma correta deste princípio para esse exemplo seria:

```
class FormaGenerica:
    def get_area(self):
        pass

class Retangulo(FormaGenerica):
    def __init__(self, altura, largura):
        self.altura = altura
        self.largura = largura
    def get_area(self):
        return self.altura * self.largura

class Quadrado(FormaGenerica):
    def __init__(self, lado):
        self.lado = lado
    def get_area(self):
        return self.lado * self.lado

retangulo = Retangulo(5, 4)
print(retangulo.get_area())

quadrado = Quadrado(4)
print(quadrado.get_area())
```

Acredito que esse seja um bom exemplo de como implementar um código tendo uma forma de pensar e depois perceber que é possível melhorar. Não se preocupe, erros vão acontecer e temos que lidar com eles. Como eu digo. Barbadinha! É só refatorar!

Princípio da Segregação de Interfaces (I)

Este princípio aborda a ideia de que uma classe não deve ser “forçada” a implementar um método do qual não utiliza.

Alguma situação onde na classe mãe existe algum método qualquer e, por sua vez, este método deve ser implementado na classe filha. Isto pode acontecer ao implementar uma classe de interface, pois nas classes que representam interfaces todos os métodos declarados devem ser implementados.

Um exemplo bem simples é:

```
class Animal:
    def comer(self):
        pass
    def voar(self):
        pass
    def nadar(self):
        pass

class Pinguim(Animal):
    def comer(self):
        # Implementação do código
        pass
    def voar(self):
        raise NotImplementedError(“Pinguins não voam.”)
    def nadar(self):
        # Implementação do código
        pass
```

Barbadinha! Se pinguins não voam, a classe pinguim não deveria ter esse método.

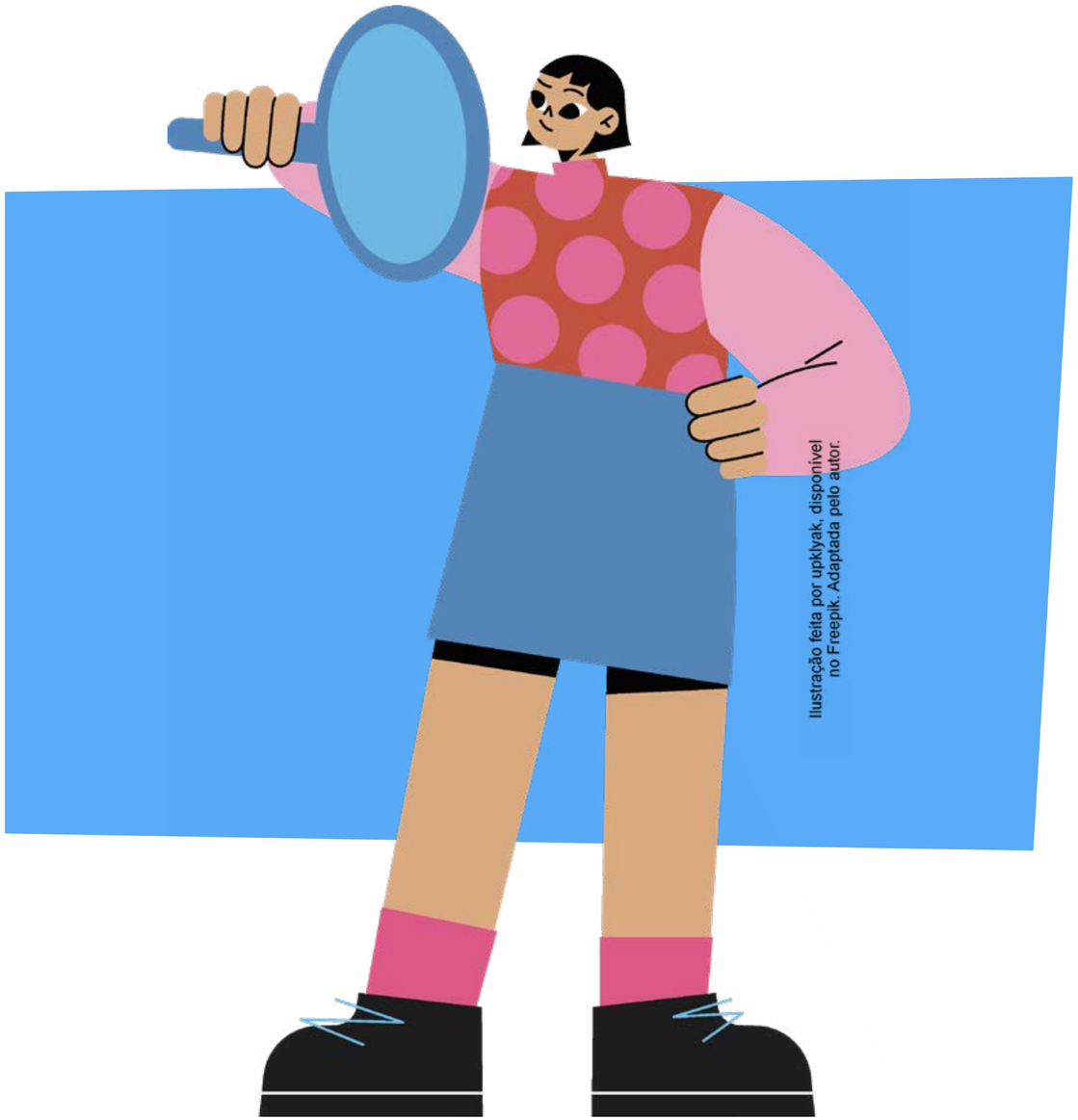
Princípio da Inversão de Dependência (D)

Diz que as classes mãe, não devem depender de classes filhas, outra forma de dizer isso é que abstrações não devem depender de detalhes de implementação. Por exemplo, um interruptor de luz não deve depender da existência de uma lâmpada, mas acender ou apagar uma lâmpada depende sim do interruptor.

Ilustração feita por upklyak, disponível no Freepik. Adaptada pelo autor.

Finalizando

Beleza! Esses princípios são norteadores, regras baseadas em experiências no desenvolvimento de software, você que está iniciando, muito provavelmente, vai ferir várias dessas regras, mas não se preocupe, é só refatorar o código e seguir em frente. Isso é assim mesmo.



Material de apoio

Código limpo (Clean Code)

https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html