Programação orientada a objeto
Aula 3

Fernando Duwe

v 1.0

As interfaces têm papel fundamental no desenvolvimento de software orientado a objetos. Por várias vezes, nós precisamos, de alguma forma, especificar um conjunto de métodos que um grupo de classes deverá, obrigatoriamente, implementar. Para atingir este efeito, utilizamos as interfaces.

Basicamente, uma interface define um “contrato” a ser seguido por outras classes. Este contrato é composto por cláusulas, que descrevem determinados comportamentos que este grupo de classes deverá seguir.

Ver mais

Uma classe abstrata é um tipo de classe especial que não pode ser instanciada, apenas herdada. Sendo assim, uma classe abstrata não pode ter um objeto criado a partir de sua instância. Essas classes são muito importantes quando não queremos criar um objeto a partir de uma classe “geral”, apenas de suas “subclasses”.

Ver mais

```
abstract class Carro {
  void acelerar();

  void frear();
}

class Fusca implements Carro {
  @override
  void acelerar() {

  }

  @override
  void frear() {

  }
}
```

Os princípios SOLID são cinco princípios do design de classes orientado a objetos. Eles são um conjunto de regras e práticas recomendadas a serem seguidas na criação de uma estrutura de classe.

- O **S**ingle Responsibility Principle (Princípio da responsabilidade única);
- O **O**pen-Closed Principle (Princípio aberto/fechado);
- O **L**iskov Substitution Principle (Princípio da substituição de Liskov);
- O **I**nterface Segregation Principle (Princípio da segregação da interface);
- O **D**ependency Inversion Principle (Princípio da inversão da dependência);

SOLID - 1º SRP — Single Responsibility Principle

Ver mais

Princípio da Responsabilidade Única — Uma classe deve ter um, e somente um, motivo para mudar.

Esse princípio declara que uma classe deve ser especializada em um único assunto e possuir apenas uma responsabilidade dentro do software, ou seja, a classe deve ter uma única tarefa ou ação para executar.

A violação do Single Responsibility Principle pode gerar alguns problemas, sendo eles:

- Falta de coesão — uma classe não deve assumir responsabilidades que não são suas;
- Alto acoplamento — Mais responsabilidades geram um maior nível de dependências, deixando o sistema engessado e frágil para alterações;
- Dificuldades na implementação de testes automatizados — É difícil de “mockar” esse tipo de classe;
- Dificuldades para reaproveitar o código;

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

Alterando uma classe já existente para adicionar um novo comportamento, corremos um sério risco de introduzir bugs em algo que já estava funcionando.

Princípio da substituição de Liskov — Uma classe derivada deve ser substituível por sua classe base.

O princípio da substituição de Liskov foi introduzido por Barbara Liskov em sua conferência “Data abstraction” em 1987. A definição formal de Liskov diz que:

Se para cada objeto o1 do tipo S há um objeto o2 do tipo T de forma que, para todos os programas P definidos em termos de T, o comportamento de P é inalterado quando o1 é substituído por o2 então S é um subtipo de T

Simplificando a definição de Liskov, podemos dizer que, Se a class A é um subtipo de B, poderemos então utilizar os objetos do tipo A, para substituir os objetos do tipo B sem que seja necessário efetuar alterações no programa.

Alguns exemplos de violação do LSP:

- Sobrescrever/implementar um método que não faz nada;
- Lançar uma exceção inesperada;
- Retornar valores de tipos diferentes da classe base;

Princípio da Segregação da Interface — Uma classe não deve ser forçada a implementar interfaces e métodos que não irão utilizar.

Esse princípio basicamente diz que é melhor criar interfaces mais específicas ao invés de termos uma única interface genérica.

O Princípio de Inversão de Dependência possui duas definições:

- Módulos de alto nível não devem depender de módulos de baixo nível e ambos devem depender de abstrações;
- Abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações;