

Princípios SOLID

O que são princípios SOLID

- S.O.L.I.D é um acrônimo para os cinco primeiros princípios de design orientado a objetos indicados por Robert C. Martin. Esses princípios facilitam o desenvolvimento de softwares, tornando-os fáceis de manter e estender. São eles:
 - 1) Single-responsibility Principle (Princípio da Responsabilidade Única)
 - 2) Open-closed Principle (Princípio Aberto/ Fechado)
 - 3) Liskov substitution principle (Princípio da Substituição de Liskov)
 - 4) Interface segregation principle (Princípio da Segregação da Interface)
 - 5) Dependency Inversion principle (Princípio da Inversão de Dependência)

1) Princípio da Responsabilidade Única

- **Definição:** Uma classe ou um método deve ser responsável por fazer apenas uma “coisa”.
- Esse princípio busca evitar classes ou métodos possuam diversas responsabilidades. Isso não só dificulta o entendimento mas também sua manutenção.
- Um exemplo de violação desse princípio seria, por exemplo, se colocássemos numa classe só Filme, FilmeFileDao e ControladoraFilmes (classes do “Aplicativo Completo”).
- Para funções/métodos um exemplo de violação desse principio pode ser encontrado quando uma funcionalidade faz, por exemplo, uma conta e também mostra seu resultado na tela. Exemplo:

```
public static void calcularIMC(double peso, double altura){  
    double resultado = peso/(altura*altura);  
    System.out.println("O valor do IMC é " + resultado);  
}
```

2) Princípio Aberto/Fechado

- **Definição:** classes, módulos, funções etc. devem estar abertas para extensão, mas fechadas para modificação.
- A forma de atender a esse princípio pode ser descrita da seguinte forma: Use uma interface para definir o comportamento extensível e inverta as dependências.
- O exemplo ao lado viola esse princípio:

**O método
calcularTotalPagamentos
deveria estar fechado para
modificação, mas qualquer
nova classe vai nos obrigar a
definir um novo "if"**

```
public abstract class Funcionario {}
```

```
public class Vendedor extends Funcionario{  
    public double totalizarComissoesVendas() {...3 lines }  
}
```

```
public class Celetista extends Funcionario {  
    public double obterSalario() {...3 lines }  
}
```

```
public class Pagamentos {  
    public double calcularTotalPagamentos(Funcionario[] funcionarios){  
        double total = 0;  
        for(int i = 0; i < funcionarios.length; i++){  
            if(funcionarios[i] instanceof Celetista)  
                total += ((Celetista)funcionarios[i]).obterSalario();  
            else if (funcionarios[i] instanceof Vendedor)  
                total += ((Vendedor)funcionarios[i]).totalizarComissoesVendas();  
        }  
        return total;  
    }  
  
    public static void main(String[] args) {...3 lines }  
}
```

2) Princípio Aberto/Fechado

- Solução (para não violar o princípio):

```
public interface Pagavel {  
    public double gerarPagamento();  
}
```

```
public abstract class Funcionario implements Pagavel {}
```

**Novas classes vão estender
Funcionario e definir gerarPagamento
não mudando o método calcularTotalPagamentos**

```
public class Celetista extends Funcionario {  
    @Override  
    public double gerarPagamento() { ...3 lines }  
}
```

```
public class Vendedor extends Funcionario {  
    @Override  
    public double gerarPagamento() { ...3 lines }  
}
```

```
public class Pagamentos {  
    public double calcularTotalPagamentos(Pagavel[] funcionarios) {  
        double total = 0;  
        for(int i = 0; i < funcionarios.length; i++) {  
            total += funcionarios[i].gerarPagamento();  
        }  
        return total;  
    }  
  
    public static void main(String[] args) { ...3 lines }  
}
```

3) Princípio da substituição de Liskov

- **Definição:** se um programa depende de um objeto de uma classe específica, ao trocá-lo por um objeto de uma subclasse dessa classe específica o comportamento do programa permanece inalterado.
- Tipos de situações onde esse princípio é violado:
 - Lançar uma exceção não prevista pela classe pai;
 - Sobrescrever um método “sem código”;
 - Retornar, em um método, valor de tipo diferente do esperado pela classe pai.
- As situações acima descritas podem não ser implementáveis em algumas linguagens. Ou seja, pode não ser possível ocorrerem algumas dessas situações simplesmente porque a linguagem utilizada não dá suporte a esse tipo de situação. Por exemplo, linguagens fortemente tipadas podem simplesmente não permitir retornos de tipos diferentes do esperado pela classe pai.

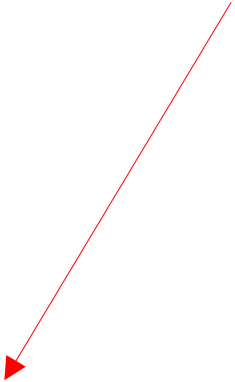
3) Princípio da substituição de Liskov

```
public class Atleta {  
    public void aquecer() {  
        System.out.println("Estou aquecendo");  
    }  
}
```

```
public class PiranhaException extends RuntimeException {  
    @Override  
    public String getMessage() {  
        return "Tem piranha";  
    }  
}
```

```
public class Nadador extends Atleta {  
    @Override  
    public void aquecer() {  
        // Suponha alguma condição que disparasse a exceção abaixo  
        throw new PiranhaException();  
    }  
}
```

Exceção não prevista na classe Atleta, seu disparo é “inesperado”

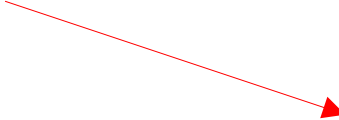


3) Princípio da substituição de Liskov

```
public class Atleta {  
    public void aquecer() {  
        System.out.println("Estou aquecendo");  
    }  
}
```

aquecer não faz nada

```
public class Nadador extends Atleta {  
    @Override  
    public void aquecer() {  
    }  
}
```



3) Princípio da substituição de Liskov

- Para retornar um valor diferente do esperado numa classe pai utilizaremos a linguagem Dart:

Esperava-se um número inteiro

```
1 class A{
2   f(){
3     return 1;
4   }
5 }
6
7 class B extends A{
8   f(){
9     return "Isso não é um número";
10  }
11 }
12
13 void main() {
14   A a = A();
15   B b = B();
16   print("O valor de f em em 'a' é ${a.f()} e em 'b' é ${b.f()}");
17 }
```

4) Princípio da Segregação da Interface

- **Definição:** Uma classe não deve ser obrigada a implementar interfaces e métodos que não irá utilizar.
- Uma solução para a violação desse princípio consiste em dividir uma interface em interfaces mais específicas (subdividir uma interface em 2 ou mais).

```
public class Pardal implements Ave{
    @Override
    public void botarOvo() {
        System.out.println("Pardais botão ovos pequenos");
    }
    @Override
    public void voar() {
        System.out.println("Estou voando");
    }
}

public interface Ave {
    public void botarOvo();
    public void voar();
}

public class Avestruz implements Ave {
    @Override
    public void botarOvo() {
        System.out.println("Avestruz bota ovo grande");
    }
    @Override
    public void voar() {
    }
}
```



Violação

4) Princípio da Segregação da Interface

- Solução:

```
public interface Ave {  
    public void botarOvo();  
}
```

```
public interface AveQueVoa extends Ave {  
    public void voar();  
}
```

```
public class Pardal implements AveQueVoa {  
    @Override  
    public void botarOvo() {  
        System.out.println("Pardais botão ovos pequenos");  
    }  
    @Override  
    public void voar() {  
        System.out.println("Estou voando");  
    }  
}
```

```
public class Avestruz implements Ave {  
    @Override  
    public void botarOvo() {  
        System.out.println("Avestruz bota ovo grande");  
    }  
}
```

5) Princípio da Inversão de Dependência

- **Definição:** Dependenda de abstrações não de implementações.
- No código abaixo a classe DAO possui um alto nível de acoplamento (dependência entre a classe DAO e a ConexaoMySQL).

```
public class DAO {  
  
    private ConexaoMySQL conexao;  
  
    public DAO() {  
        this.conexao = new ConexaoMySQL();  
    }  
}
```

5) Princípio da Inversão de Dependência

- **Injeção de Dependência** (nesse caso, via construtor): se passarmos o objeto de conexão como parâmetro para o construtor então isso diminuirá a dependência que a classe DAO possui da ConexaoMySQL. É importante notar que, no exemplo anterior, a classe DAO é obrigada a conhecer como uma ConexaoMySQL deve ser criada, ou seja, há uma grande dependência, um grande acoplamento. Se ao invés de criarmos a ConexaoMySQL no construtor de DAO passarmos a ConexaoMySQL como parâmetro para o construtor então diminuiremos essa dependência (o objeto de ConexaoMySQL já “virá pronto” e será “injetado” como parâmetro do construtor).

```
public class DAO {  
  
    private ConexaoMySQL conexao;  
  
    public DAO(ConexaoMySQL conexao) {  
        this.conexao = conexao;  
    }  
}
```

5) Princípio da Inversão de Dependência

- A injeção de dependência ajudou mas ainda não resolveu o problema da inversão de dependência. Note que ainda dependemos da implementação de `ConexaoMySQL`.
- Para resolver essa questão precisaremos depender de uma “interface de conexão” (interfaces normalmente não possuem implementação, digo normalmente por conta dos métodos *default*).

```
public interface Conexao {
    public void conectar();
}

public class ConexaoMySQL implements Conexao{
    @Override
    public void conectar() {
        // Conectando com MySQL
    }
}

public class ConexaoSQLServer implements Conexao{
    @Override
    public void conectar() {
        // Conectando com SQLServer
    }
}
```

```
public class DAO {

    private Conexao conexao;

    public DAO(Conexao conexao) {
        this.conexao = conexao;
    }
}
```

Note que para trocar a implementação basta trocar o objeto criado, mas isso não muda nada no código de DAO

Note também que, se tivermos um banco novo basta criar sua classe de conexão e usar

```
public class Principal {
    public static void main(String[] args) {
        DAO daol = new DAO(new ConexaoMySQL());
        // DAO daol = new DAO(new ConexaoSQLServer());
    }
}
```

Dúvidas?

