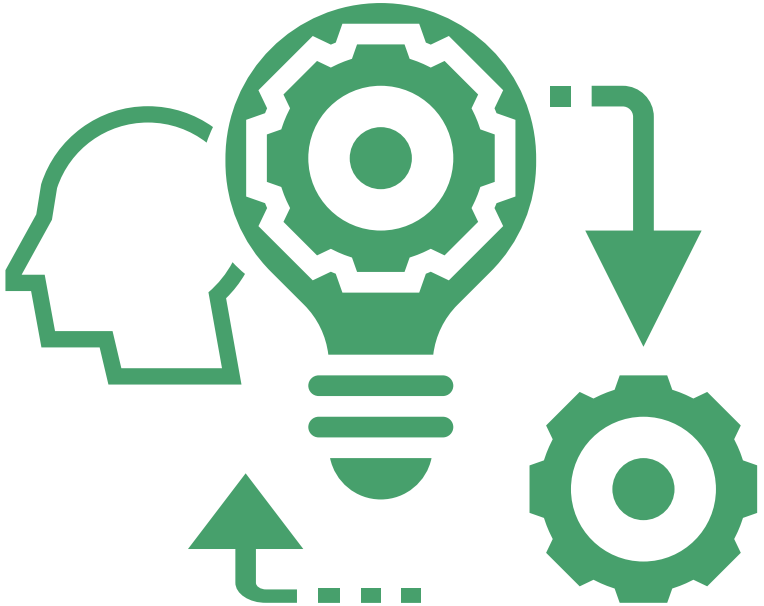


Paradigmas de Programação

Aula 05 – Como vai ser ???

- Entregáveis Aula 04
  - LSP e ISP

- **Let's Code**
  - Vamos tomar um "**Java Coffee**"???
  - E o **Eclipse**, é o solar ou o lunar???
  - **Spring**, mas já está chegando o outono...

Princípio de Substituição de Liskov

- **Significa dizer que classes derivadas devem poder substituídas por suas classes base e que classes base podem ser substituídas por qualquer uma das suas subclasses.**

- Uma subclasse deve sobrescrever os métodos da superclasse de forma que a funcionalidade do ponto de vista do cliente continue a mesma.

```java
abstract class FormaGeometrica {
    double lado;
    abstract double calculaArea();
    abstract double calculaVolume();
}
```

```java
class Cubo extends FormaGeometrica{
    @Override
    double calculaArea() {
        return 6 * (lado * lado);
    }
    @Override
    double calculaVolume() {
        return 3 * lado;
    }
}
```

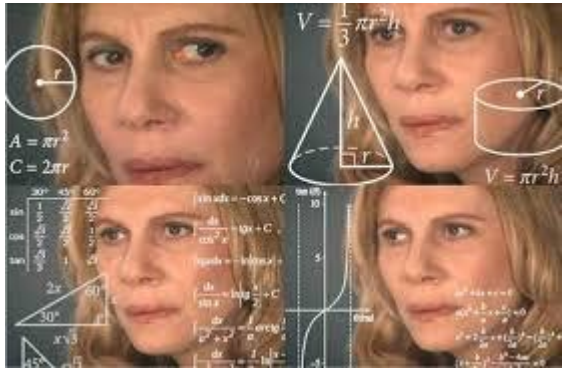
```java
class Quadrado extends FormaGeometrica {
    Quadrado(double lado) {
        this.lado = lado;
    }
    @Override
    double calculaArea() {
        return lado * lado;
    }
    @Override
    double calculaVolume() {
        //return 0;
        throw new UnsupportedOperationException(
            "Unsupported method 'calculaVolume'");
    }
}
```

```java
public class LSP {
    public static void main(String[] args) {
        Cubo cubo = new Cubo(2);
        imprimeArea(cubo);
        imprimeVolume(cubo);
        Quadrado ret = new Quadrado(3);
        imprimeArea(ret);
        imprimeVolume(ret);

    }
    static void imprimeArea(FormaGeometrica forma) {
        System.out.println(forma.calculaArea());
    }
    static void imprimeVolume(FormaGeometrica forma) {
        System.out.println(forma.calculaVolume());
    }
}
```

```
24.0
6.0
9.0
Exception in thread "main" java.lang.UnsupportedOperationException: Unsupported method 'calculaVolume'
        at Quadrado.calculaVolume(LSP.java:53)
        at LSP.imprimeVolume(LSP.java:15)
        at LSP.main(LSP.java:8)
```

Mas porque esse código viola o LSP??



- Esse comportamento viola o LSP porque:
- Ao tentar substituir *FormaGeometrica* por uma instância de *Quadrado*, o método *imprimeVolume()* quebra o programa ao lançar uma exceção.
  - Atenção: Caso alterarmos o método *calculaVolume()* da classe *Quadrado* para retornar um valor qualquer (exemplo: zero), também é considerado uma violação, pois se podemos invocar o método, então espera-se que o retorno seja realmente o volume daquela forma.
- Isso sugere que a interface da classe base (FormaGeometrica) define um comportamento que nem todas as subclasses podem cumprir.

E como podemos corrigir???

- Uma solução seria separar as classes em duas hierarquias diferentes, por exemplo, criando uma classe base *FormaBidimensional* para formas como *Quadrado* que não possuem volume
- E uma classe *FormaTridimensional* para formas como *Cubo*, que têm volume.
- Isso garantiria que cada subclasse respeite completamente os métodos definidos pela sua classe base.

```
abstract class FormaBidimensional extends FormaGeometrica {
}
abstract class FormaTridimensional extends FormaGeometrica {
    abstract double calculaVolume();
}
```

```
class Cubo extends FormaTridimensional {
```

```
class Quadrado extends FormaBidimensional {
```

```
    public static void main(String[] args) {
        Cubo cubo = new Cubo(2);
        imprimeArea(cubo);
        imprimeVolume(cubo);
        Quadrado ret = new Quadrado(3);
        imprimeArea(ret);
        imprimeVolume(ret);
    }
    static void imprimeArea(FormaGeometrica forma) {
        System.out.println(forma.calculaArea());
    }
    static void imprimeVolume(FormaTridimensional forma) {
        System.out.println(forma.calculaVolume());
    }
```

Princípio da Segregação de Interface

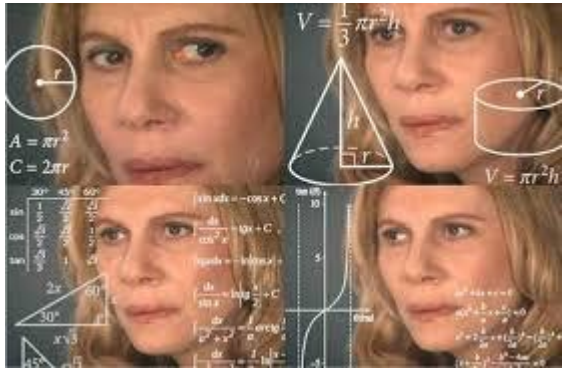
- **Este princípio estabelece que uma classe classe não deve ser forçada a implementar métodos que não usa.**
- De acordo com o Princípio da Segregação da Interface, uma interface deve ser coesa e ter apenas o mínimo necessário para seus clientes.
- As interfaces devem ser segregadas de forma a cada cliente depender apenas dos métodos que precisa utilizar, evitando assim a dependência de funcionalidades desnecessárias.

```java
interface Ave {
    void comer();
    void voar();
}
```

```java
class Canarinho implements Ave {
    @Override
    public void comer() {
        System.out.println("Canarinho comendo");
    }
    @Override
    public void voar() {
        System.out.println("Canarinho voando");
    }
}
```

```java
class Pinguim implements Ave {
    @Override
    public void comer() {
        System.out.println("Pinguim comendo");
    }
    @Override
    public void voar() {
        throw new UnsupportedOperationException("Pinguim não voa");
    }
}
```

Mas porque esse código viola o ISP??



- A interface *Ave* é muito ampla, pois impõe que todas as aves, independentemente de suas habilidades, implementem o método *voar()*.

- Isso viola o ISP, já que *Pinguim* é uma ave que não deveria ser obrigada a implementar algo que não é relevante para ela.

E como podemos corrigir???

- Uma solução seria dividir a interface *Ave* em interfaces mais específicas, como *AveVoadora* e *AveComedora*. Dessa forma, apenas as aves que realmente voam (como *Canarinho*) implementariam a interface *AveVoadora*, enquanto o Pinguim apenas implementaria a interface que contém o comportamento de *comer()*.

```java
interface AveVoadora {
    void voar();
}
interface AveComedora {
    void comer();
}
```

```java
class Canarinho implements AveVoadora, AveComedora {
    @Override
    public void comer() {
        System.out.println("Canarinho comendo");
    }
    @Override
    public void voar() {
        System.out.println("Canarinho voando");
    }
}
```

```java
class Pinguim implements AveComedora {
    @Override
    public void comer() {
        System.out.println("Pinguim comendo");
    }
}
```

Entregáveis Aula 04 – Questão 05

- Explique se o uso dessas implementações da interface *List* pode ferir o Princípio de Substituição de Liskov (LSP).

- Avalie se a Interface Segregation Principle (ISP) está sendo violado ao utilizar essas listas, considerando as operações permitidas e não permitidas nas mesmas.

```java
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Lista imutável usando List.of()
        List<String> listaImutavel = List.of("A", "B", "C");

        // Lista de tamanho fixo usando Arrays.asList()
        List<String> listaTamanhoFixo = Arrays.asList("X", "Y", "Z");

        // Testando operações de adição e remoção
        listaImutavel.add("D"); // O que acontece aqui?
        listaTamanhoFixo.remove("X"); // E aqui?
    }
}
```

```
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.base/java.util.ImmutableCollections.uoe(ImmutableCollections.java:142)
        at java.base/java.util.ImmutableCollections$AbstractImmutableCollection.add(ImmutableCollections.java:147)
        at Main.main(Main.java:13)
```

```
Exception in thread "main" java.lang.UnsupportedOperationException: remove
        at java.base/java.util.Iterator.remove(Iterator.java:102)
        at java.base/java.util.AbstractCollection.remove(AbstractCollection.java:285)
        at Main.main(Main.java:14)
```

Bom senso é a chave!!!

- A aplicação de princípios, padrões e boas práticas, deve ser guiada pelo bom senso e pela consideração do contexto em que o código será mantido e evoluído.

- Se a violação resulta em um código mais limpo, intuitivo e fácil de manter, pode ser preferível aceitar a violação em vez de criar métodos adicionais que apenas obscurecem a simplicidade do design.

Let’s code!!!

spring®

What Spring can do

Microservices

Quickly deliver production-grade features with independently evolvable microservices.

Reactive

Spring's asynchronous, nonblocking architecture means you can get more from your computing resources.

Cloud

Your code, any cloud—we've got you covered. Connect and scale your services, whatever your platform.

Web apps

Frameworks for fast, secure, and responsive web applications connected to any data store.

Serverless

The ultimate flexibility. Scale up on demand and scale to zero when there's no demand.

Event Driven

Integrate with your enterprise. React to business events. Act on your streaming data in realtime.

Batch

Automated tasks. Offline processing of data at a time to suit you.

Spring Framework e Spring Boot

- O **Spring Framework** é um framework de código aberto que fornece um ambiente de desenvolvimento abrangente para aplicativos Java. Ele foi lançado pela primeira vez em 2002 e desde então se tornou uma escolha popular para o desenvolvimento de aplicativos corporativos. Algumas características-chave do Spring Framework incluem:

  - **Injeção de Dependência (Dependency Injection - DI)**
  - **Programação Orientada a Aspectos (Aspect-Oriented Programming – AOP)**
  - **MVC (Model-View-Controller**)
  - **Integração com Banco de Dados**
  - **Segurança**

Spring Framework e Spring Boot

- O **Spring Boot** é uma extensão do Spring Framework que simplifica muito o desenvolvimento de aplicativos Java. Ele foi projetado para acelerar o processo de configuração e desenvolvimento, permitindo que os desenvolvedores se concentrem mais na lógica de negócios e menos na configuração. Algumas características do Spring Boot incluem:

  - **Configuração Automática**
  - **Embedded Web Server**
  - **Pronto para Produção**
  - **Spring Boot Starter**
  - **Facilidade de Teste**
  - **Ampla Comunidade**

Ciência da
Computação