

# Orientação a Objetos

---

Fernando Camargo

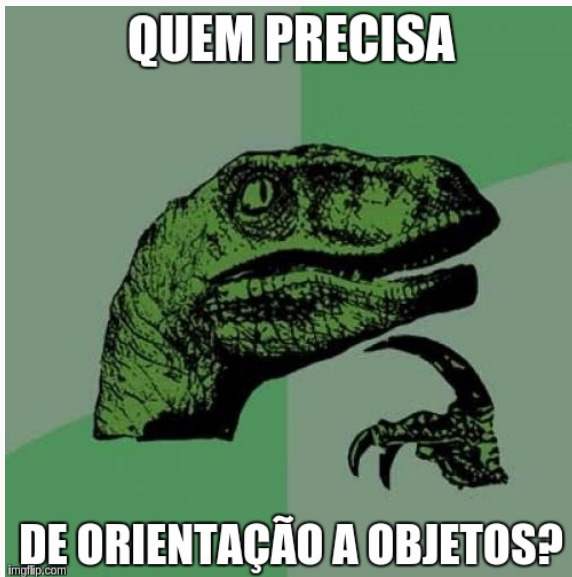
2 de junho de 2017

ZG Soluções

**Por que um tema tão básico?**

---

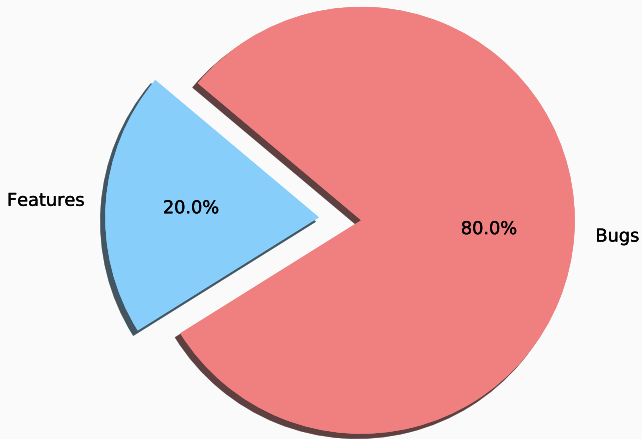
Por que um tema tão básico?



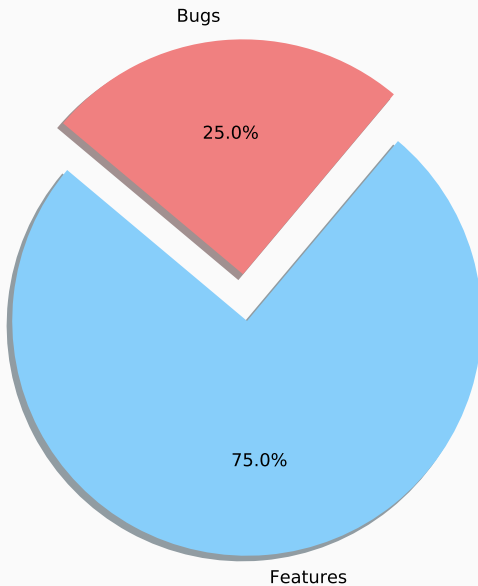
## **Vantagens da Qualidade de Código**

---

## Tempo gasto com código de má qualidade



## Tempo gasto com código de boa qualidade



## Avaliando um Código OO

---

**E SE EU IMPLEMENTASSE**

**TODAS FUNCIONALIDADES NESSA CLASSE?**

imgflip.com



## Repetição de código (DRY)

- "Don't Repeat Yourself"

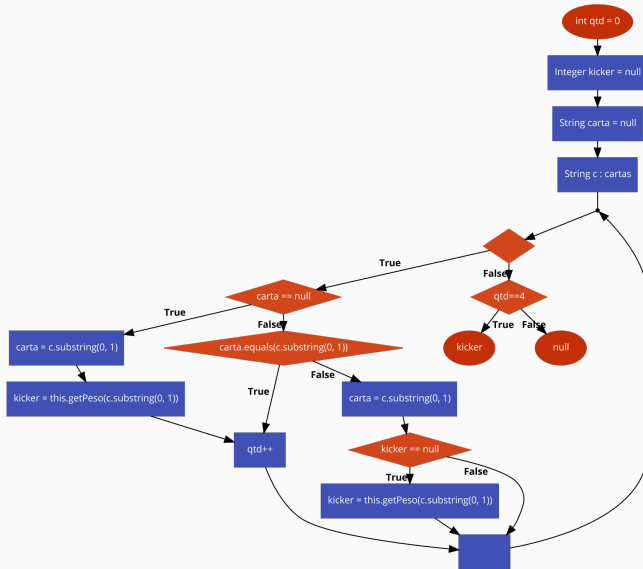
## Repetição de código (DRY)

- "Don't Repeat Yourself"
- Lógica duplicada deve ser eliminada via abstração

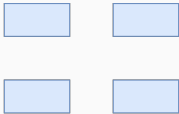
# Complexidade de código

```
int qtd = 0;
Integer kicker = null;
String carta = null;
for(String c:cartas){
    if(carta == null){
        carta = c.substring(0, 1);
        kicker = this.getPeso(c.substring(0, 1));
        qtd++;
    }else{
        if(carta.equals(c.substring(0, 1))){
            qtd++;
        }else{
            carta = c.substring(0, 1);
            if(kicker == null){
                kicker = this.getPeso(c.substring(0, 1));
            }
        }
    }
}
if(qtd==4){
    return kicker;
}
return null;
```

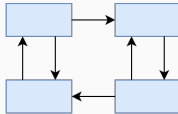
# Complexidade de código



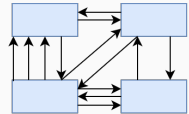
# Acoplamento



Sem dependências



Baixo acoplamento  
Algumas dependências



Alto acoplamento  
Muitas dependências

- Não existe zero acoplamento

- Não existe zero acoplamento
- Baixo acoplamento → alterações pontuais

- Não existe zero acoplamento
- Baixo acoplamento → alterações pontuais
- Alto acoplamento → alterações em todo o código/em cascata



## O que causa alto acoplamento?

- Classes sabem demais sobre as outras

## O que causa alto acoplamento?

- Classes sabem demais sobre as outras
  - Acesso direto de propriedades

## O que causa alto acoplamento?

- Classes sabem demais sobre as outras
  - Acesso direto de propriedades
  - Construção de dependências

## O que causa alto acoplamento?

- Classes sabem demais sobre as outras
  - Acesso direto de propriedades
  - Construção de dependências
  - Uso de implementações ao invés de interfaces

## O que causa alto acoplamento?

- Classes sabem demais sobre as outras
  - Acesso direto de propriedades
  - Construção de dependências
  - Uso de implementações ao invés de interfaces
- Falta de organização estruturada das classes (separação de camadas)

- Classes

- Classes
  - Baixa coesão → múltiplos métodos com responsabilidades/tarefas não relacionadas

- Classes
  - Baixa coesão → múltiplos métodos com responsabilidades/tarefas não relacionadas
  - Alta coesão → classe possui uma única responsabilidade/tarefa, com métodos relacionados a ela



- Classes
  - Baixa coesão → múltiplos métodos com responsabilidades/tarefas não relacionadas
  - Alta coesão → classe possui uma única responsabilidade/tarefa, com métodos relacionados a ela
- Métodos

- Classes
  - Baixa coesão → múltiplos métodos com responsabilidades/tarefas não relacionadas
  - Alta coesão → classe possui uma única responsabilidade/tarefa, com métodos relacionados a ela
- Métodos
  - Baixa coesão → método realiza várias tarefas

- Classes
  - Baixa coesão → múltiplos métodos com responsabilidades/tarefas não relacionadas
  - Alta coesão → classe possui uma única responsabilidade/tarefa, com métodos relacionados a ela
- Métodos
  - Baixa coesão → método realiza várias tarefas
  - Alta coesão → método com uma única tarefa, podendo chamar métodos que a complementem

# Sintomas de Projeto de Classes em Degradação

---

- Rigidez: toda mudança causa uma cascata de mudanças subsequentes em módulos dependentes

- Rigidez: toda mudança causa uma cascata de mudanças subsequentes em módulos dependentes
- Fragilidade: mudanças acarretam em quebras em muitos lugares diferentes

## Sintomas

- Rigidez: toda mudança causa uma cascata de mudanças subsequentes em módulos dependentes
- Fragilidade: mudanças acarretam em quebras em muitos lugares diferentes
- Imobilidade: impossibilidade de reusar módulos em outros projetos

- Rigidez: toda mudança causa uma cascata de mudanças subsequentes em módulos dependentes
- Fragilidade: mudanças acarretam em quebras em muitos lugares diferentes
- Imobilidade: impossibilidade de reusar módulos em outros projetos
- Viscosidade: fácil fazer a "coisa errada" e difícil fazer a "coisa certa"



- Rigidez: toda mudança causa uma cascata de mudanças subsequentes em módulos dependentes
- Fragilidade: mudanças acarretam em quebras em muitos lugares diferentes
- Imobilidade: impossibilidade de reusar módulos em outros projetos
- Viscosidade: fácil fazer a "coisa errada" e difícil fazer a "coisa certa"
- Complexidade desnecessária: muitos elementos inúteis ou não utilizados (dead code)

- Rigidez: toda mudança causa uma cascata de mudanças subsequentes em módulos dependentes
- Fragilidade: mudanças acarretam em quebras em muitos lugares diferentes
- Imobilidade: impossibilidade de reusar módulos em outros projetos
- Viscosidade: fácil fazer a "coisa errada" e difícil fazer a "coisa certa"
- Complexidade desnecessária: muitos elementos inúteis ou não utilizados (dead code)
- Repetição desnecessária: falta de abstração apropriada para evitar repetição de código

- Rigidez: toda mudança causa uma cascata de mudanças subsequentes em módulos dependentes
- Fragilidade: mudanças acarretam em quebras em muitos lugares diferentes
- Imobilidade: impossibilidade de reusar módulos em outros projetos
- Viscosidade: fácil fazer a "coisa errada" e difícil fazer a "coisa certa"
- Complexidade desnecessária: muitos elementos inúteis ou não utilizados (dead code)
- Repetição desnecessária: falta de abstração apropriada para evitar repetição de código
- Opacidade: código difícil de ser entendido

**SOLID**

---

## Single Responsibility Principle



## Single Responsibility Principle

"Uma classe deve ter um, e somente um, motivo para mudar."

## Single Responsibility Principle

- Mudanças de requisitos → mudanças nas responsabilidades

## Single Responsibility Principle

- Mudanças de requisitos  $\rightarrow$  mudanças nas responsabilidades
- Classes com múltiplas responsabilidades:



# Single Responsibility Principle

- Mudanças de requisitos → mudanças nas responsabilidades
- Classes com múltiplas responsabilidades:
  - Múltiplos motivos de mudança

# Single Responsibility Principle

- Mudanças de requisitos → mudanças nas responsabilidades
- Classes com múltiplas responsabilidades:
  - Múltiplos motivos de mudança
  - Acoplamento das responsabilidades → difícil alteração

# Single Responsibility Principle

- Mudanças de requisitos → mudanças nas responsabilidades
- Classes com múltiplas responsabilidades:
  - Múltiplos motivos de mudança
  - Acoplamento das responsabilidades → difícil alteração
- Conclusão: uma classe deve ter uma **única** responsabilidade



"Entidades de Software (classes, módulos, funções, etc.) devem ser abertas para extensão, mas fechadas para modificação."

## Open/Closed Principle

- Uma mudança deve resultar em uma cascata de mudanças em classes dependentes.

## Open/Closed Principle

- ~~Uma mudança deve resultar em uma cascata de mudanças em classes dependentes.~~
- Mudanças de requisito → adição de código novo sem alteração de código existente

## Open/Closed Principle

- ~~Uma mudança deve resultar em uma cascata de mudanças em classes dependentes.~~
- Mudanças de requisito → adição de código novo sem alteração de código existente
- Como?



## Open/Closed Principle

- ~~Uma mudança deve resultar em uma cascata de mudanças em classes dependentes.~~
- Mudanças de requisito → adição de código novo sem alteração de código existente
- Como?
  - Abstrações que permitam um grupo ilimitado de possíveis comportamentos → classes abstratas e interfaces

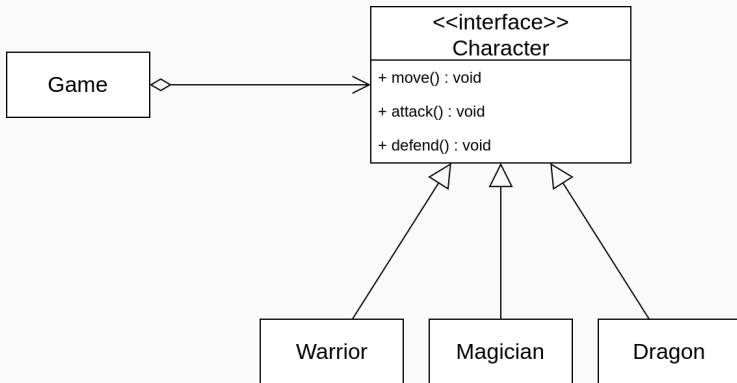
# Open/Closed Principle

- ~~Uma mudança deve resultar em uma cascata de mudanças em classes dependentes.~~
- Mudanças de requisito → adição de código novo sem alteração de código existente
- Como?
  - Abstrações que permitam um grupo ilimitado de possíveis comportamentos → classes abstratas e interfaces
  - Novos comportamentos adicionados por herança ou implementação de interface

## Open/Closed Principle

- ~~Uma mudança deve resultar em uma cascata de mudanças em classes dependentes.~~
- Mudanças de requisito → adição de código novo sem alteração de código existente
- Como?
  - Abstrações que permitam um grupo ilimitado de possíveis comportamentos → classes abstratas e interfaces
  - Novos comportamentos adicionados por herança ou implementação de interface
  - Classes dependem da abstração (fixa), não da implementação

# Open/Closed Principle





## Liskov Substitution Principle

"Uma classe base deve poder ser substituída pela sua classe derivada."

# Liskov Substitution Principle

- Extensão do Open/Closed Principle

# Liskov Substitution Principle

- Extensão do Open/Closed Principle
- Classes derivadas não podem alterar o comportamento de classes base



# Liskov Substitution Principle

```
class Square extends Rectangle {
    public void setWidth(int width){
        this.width = width;
        this.height = width;
    }

    public void setHeight(int height){
        this.width = height;
        this.height = height;
    }
}

class LspTest {
    private static Rectangle getNewRectangle() {
        return new Square();
    }

    public static void main (String args[]) {
        Rectangle r = LspTest.getNewRectangle();

        r.setWidth(5);
        r.setHeight(10);

        System.out.println(r.getArea()); // Resultado: 100 ao invés de 50
    }
}
```

## Interface Segregation Principle

**DEIXA EU COLOCAR**



**SÓ MAIS UM MÉTODO AQUI**

## Interface Segregation Principle

"Muitas interfaces específicas são melhores do que uma interface única."

## Interface Segregation Principle

- Interfaces poluídas prejudicam a coesão

# Interface Segregation Principle

- Interfaces poluídas prejudicam a coesão
- "Clientes não devem ser forçados a depender de interfaces que eles não usam"
  - Dependência de interfaces "gordas" gera acoplamento entre implementações

# Interface Segregation Principle

- Interfaces poluídas prejudicam a coesão
- "Clientes não devem ser forçados a depender de interfaces que eles não usam"
  - Dependência de interfaces "gordas" gera acoplamento entre implementações
- Diferentes clientes (implementações com diferentes responsabilidades) → diferentes interfaces

# ERRADO!

```
interface Worker {  
    void work();  
    void eat();  
}  
  
class FactoryWorker implements Worker {  
    public void work() { /* implementation */ }  
    public void eat() { /* implementation */ }  
}  
  
class Robot implements Worker {  
    public void work() { /* implementation */ }  
    public void eat() { /* ??? */ }  
}
```

# CORRETO!

```
interface Workable {  
    public void work();  
}  
  
interface Feedable{  
    public void eat();  
}  
  
class FactoryWorker implements Workable, Feedable {  
    public void work() { /* implementation */ }  
    public void eat() { /* implementation */ }  
}  
  
class Robot implements Workable {  
    public void work() { /* implementation */ }  
}
```



## Dependency Inversion Principle



# Dependency Inversion Principle

"Dependa de uma abstração e não de uma implementação."

# Dependency Inversion Principle

- Implementações de baixo nível podem ser alteradas

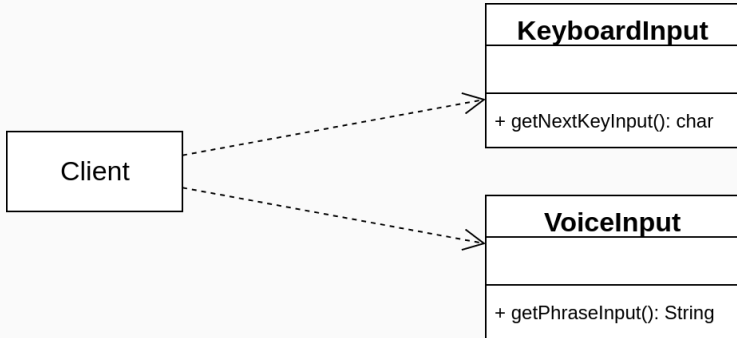
# Dependency Inversion Principle

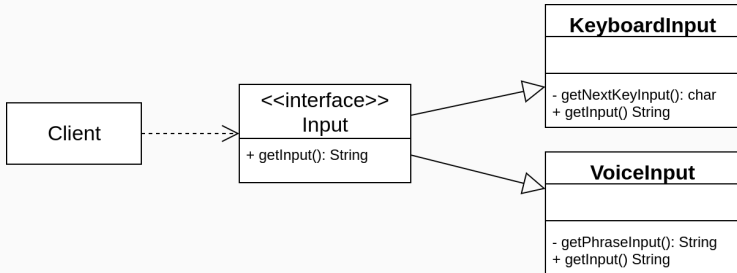
- Implementações de baixo nível podem ser alteradas
- Uso dessas implementações → alto acoplamento → alterações de dependentes

# Dependency Inversion Principle

- Implementações de baixo nível podem ser alteradas
- Uso dessas implementações → alto acoplamento → alterações de dependentes
- Uso de abstrações de alto nível (interfaces) → baixo acoplamento

# ERRADO!





## Conclusões

---



- Evite repetições de código

- Evite repetições de código
- Aplique os princípios SOLID para aumentar coesão e reduzir acoplamento

- Evite repetições de código
- Aplique os princípios SOLID para aumentar coesão e reduzir acoplamento
- Estude Padrões de Projeto