Orientação a Objetos

Fernando Camargo

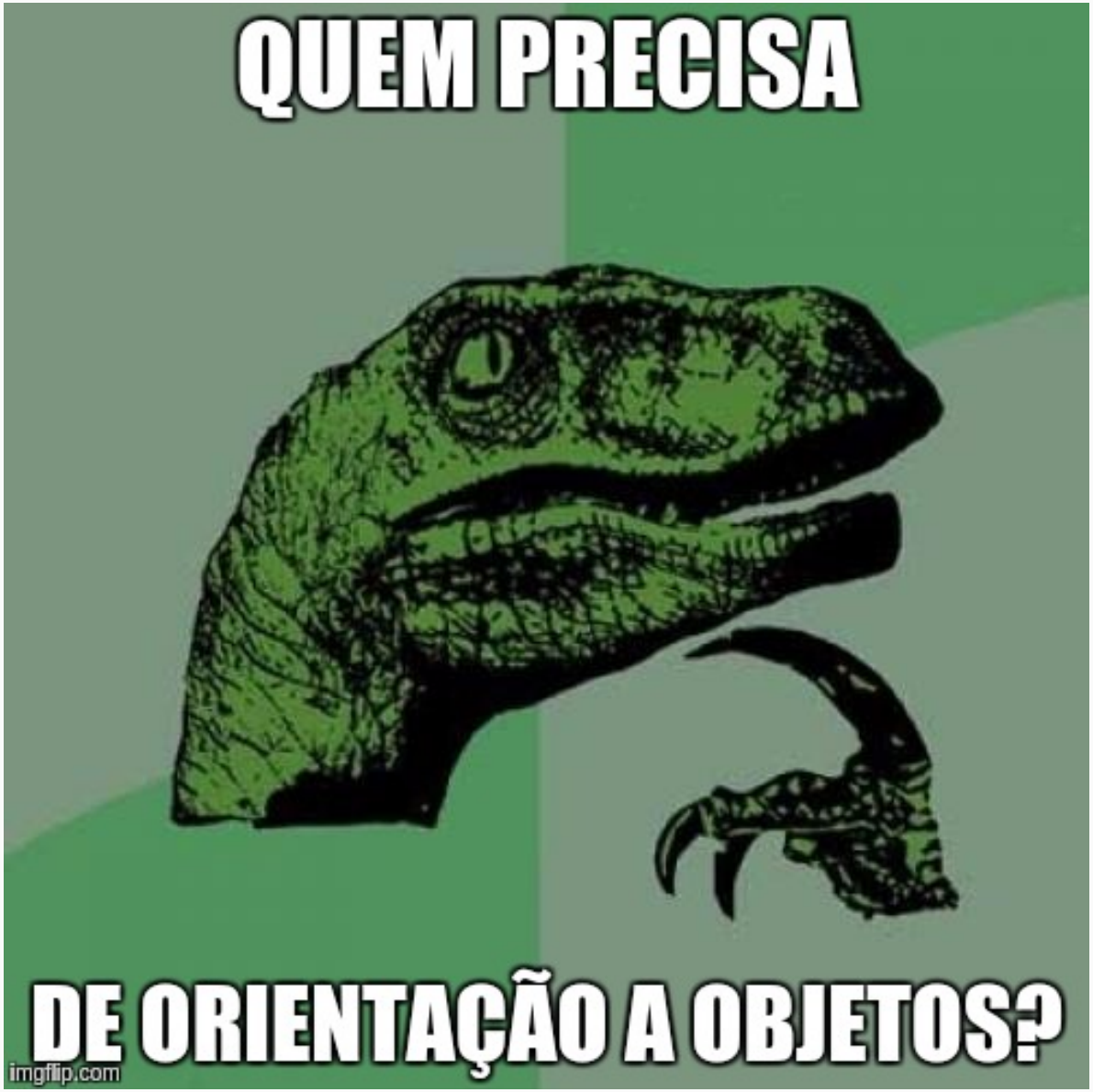
1 de junho de 2017

ZG Soluções

ZG SOLUÇÕES

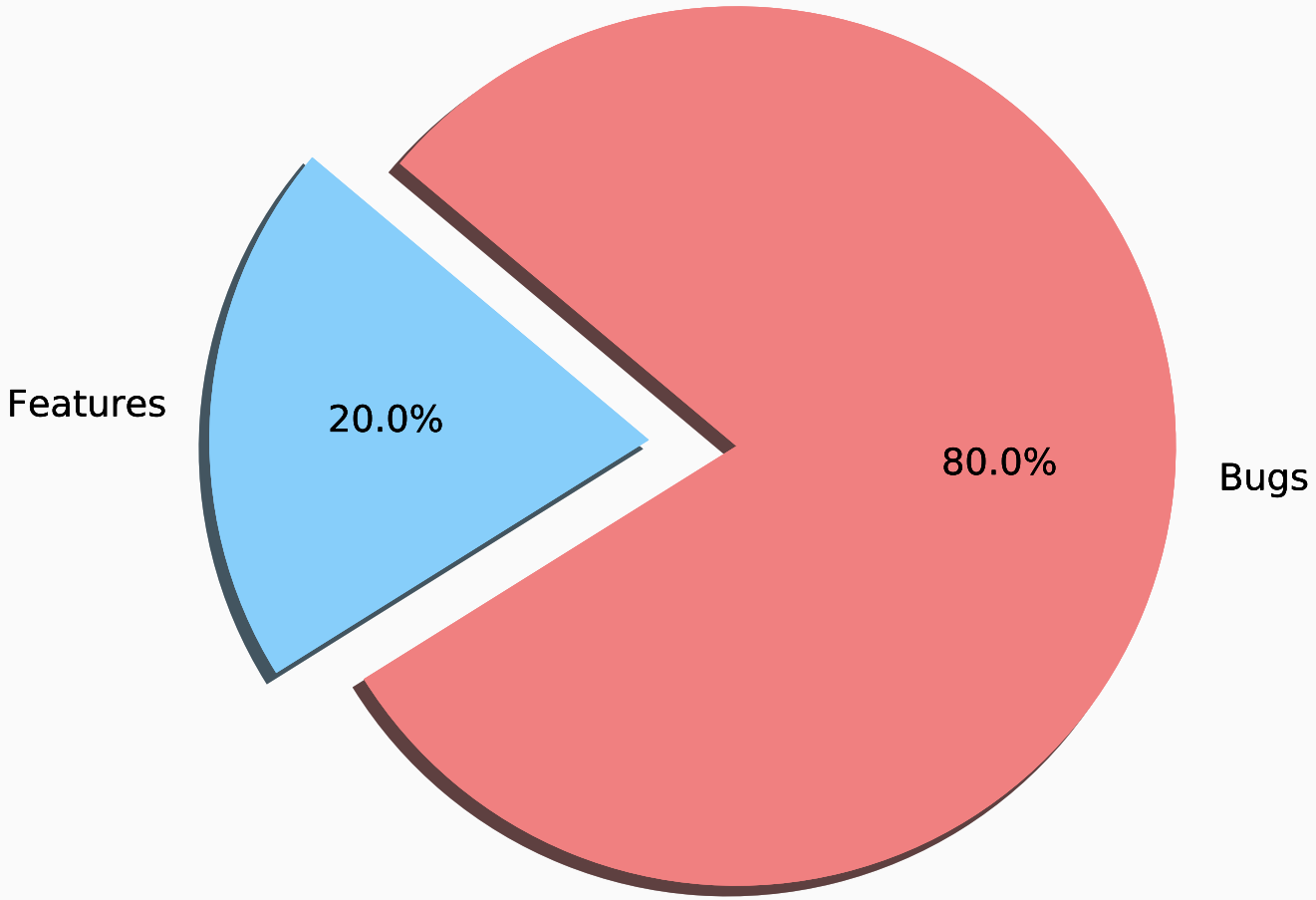
Por que um tema tão básico?

Por que um tema tão básico?

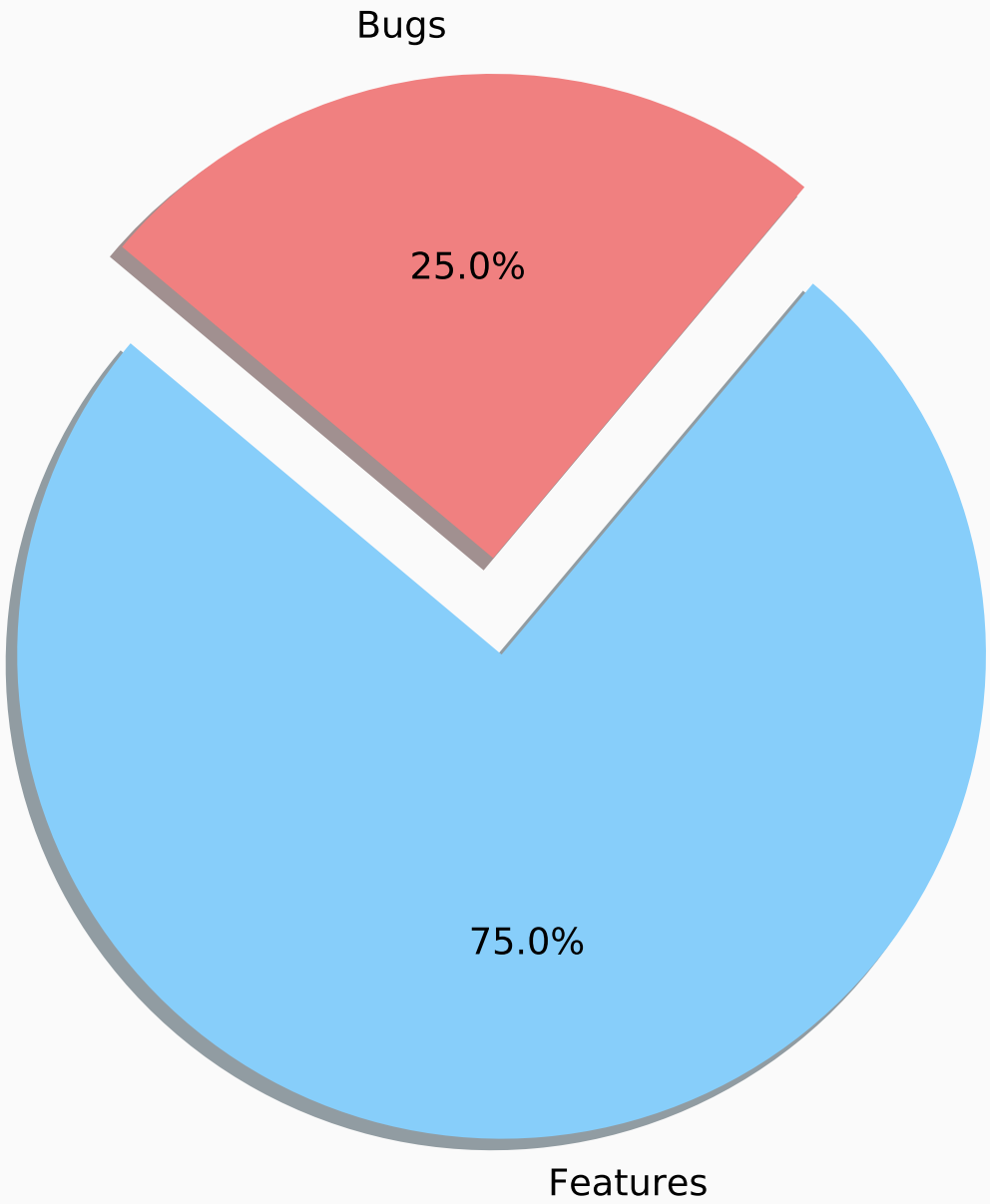


Vantagens da Qualidade de Código

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

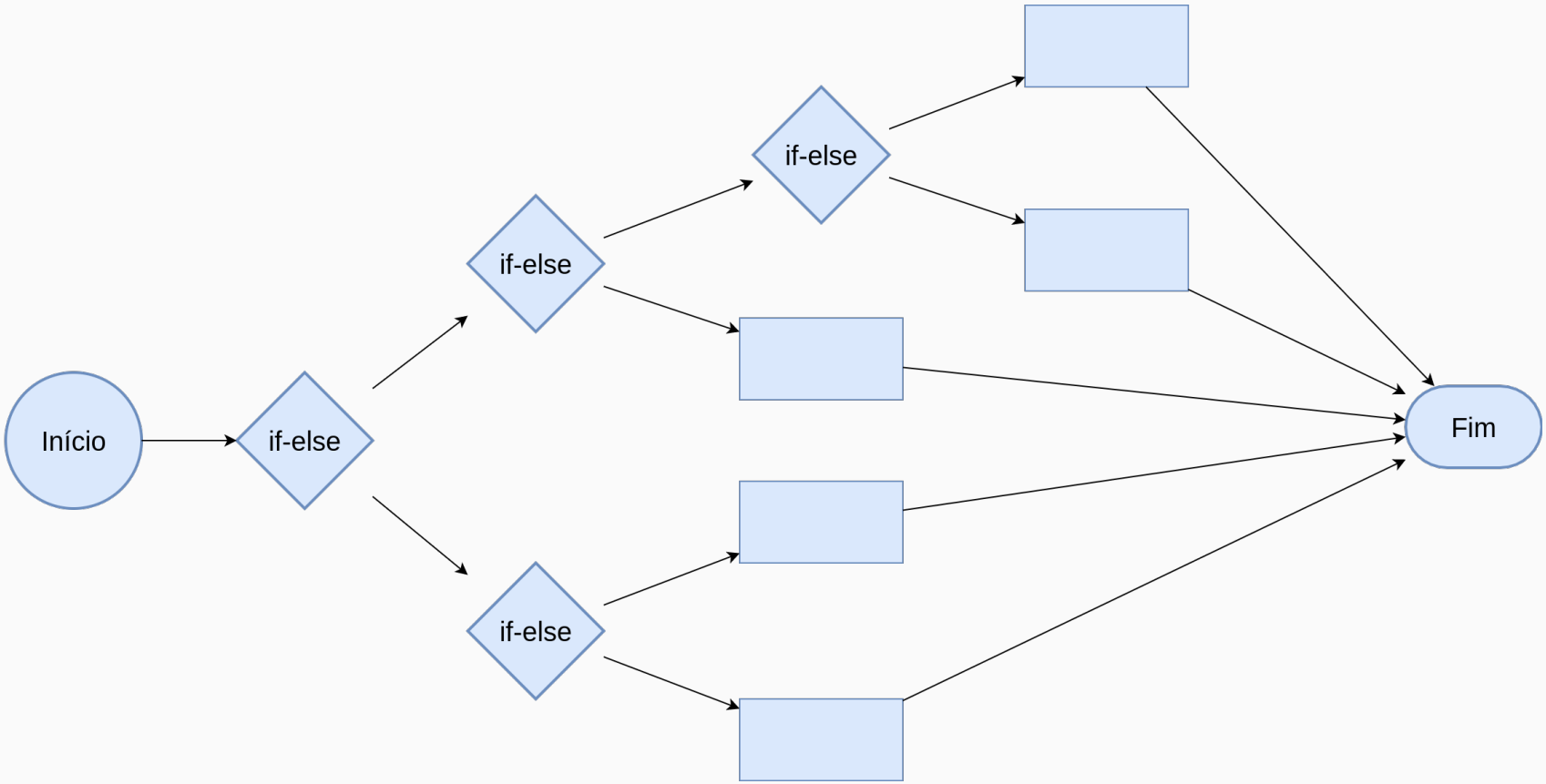
Repetição de código (DRY)

- ”Don’t Repeat Yourself”

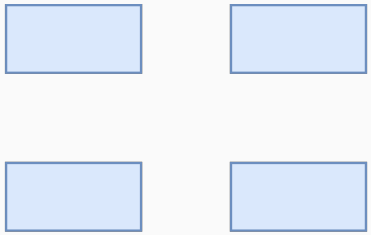
Repetição de código (DRY)

- ”Don’t Repeat Yourself”
- Lógica duplicada deve ser eliminada via abstração

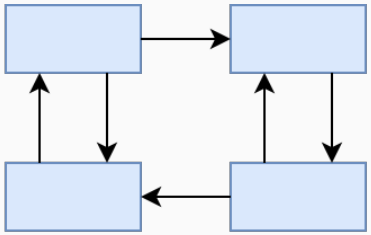
Complexidade de código



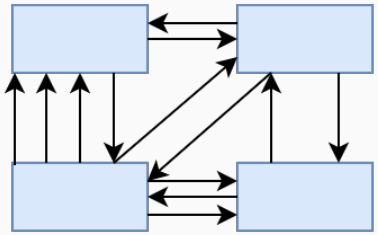
Acoplamento



Sem dependências



Baixo acoplamento
Algumas dependências



Alto acoplamento
Muitas dependências

Acoplamento

- Não existe zero acoplamento

Acoplamento

- Não existe zero acoplamento
- Baixo acoplamento $\rightarrow$ alterações pontuais

Acoplamento

- Não existe zero acoplamento
- Baixo acoplamento $\rightarrow$ alterações pontuais
- Alto acoplamento $\rightarrow$ 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)

Coesão

- Classes

Coesão

- Classes
  - Baixa coesão $\rightarrow$ múltiplos métodos com responsabilidades/tarefas não relacionadas

Coesão

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

Coesão

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

Coesão

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

Coesão

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

Sintomas de Projeto de Classes em Degradação

Sintomas

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

Sintomas

- 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

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
- Viscosidade: fácil fazer a "coisa errada"e difícil fazer a "coisa certa"

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
- Viscosidade: fácil fazer a "coisa errada"e difícil fazer a "coisa certa"
- Complexidade desnecessária: muitos elementos inúteis ou não utilizados (dead code)

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
- 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

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
- 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 $\rightarrow$ mudanças nas responsabilidades

Single Responsibility Principle

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

Single Responsibility Principle

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

Single Responsibility Principle

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

Single Responsibility Principle

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

Open/Closed Principle

Open/Closed Principle

"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 $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ adição de código novo sem alteração de código existente
- Como?
  - Abstrações que permitam um grupo ilimitado de possíveis comportamentos $\rightarrow$ classes abstratas e interfaces

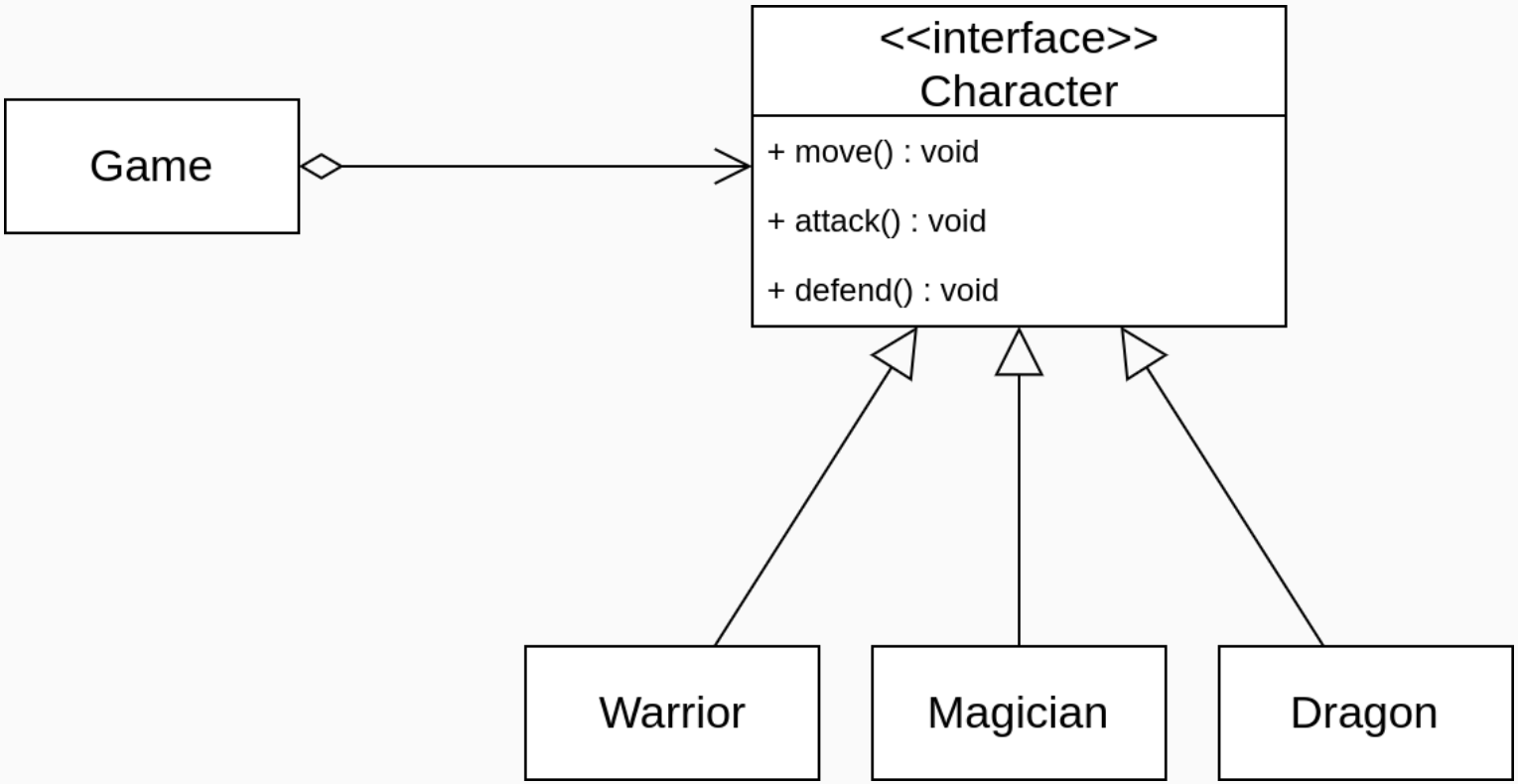
Open/Closed Principle

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

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

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 forçacos 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 forçacos a depender de interfaces que eles não usam"
    - Dependência de interfaces "gordas"gera acoplamento entre implementações
- Diferentes clientes (implementações com diferentes responsabilidades) $\rightarrow$ 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

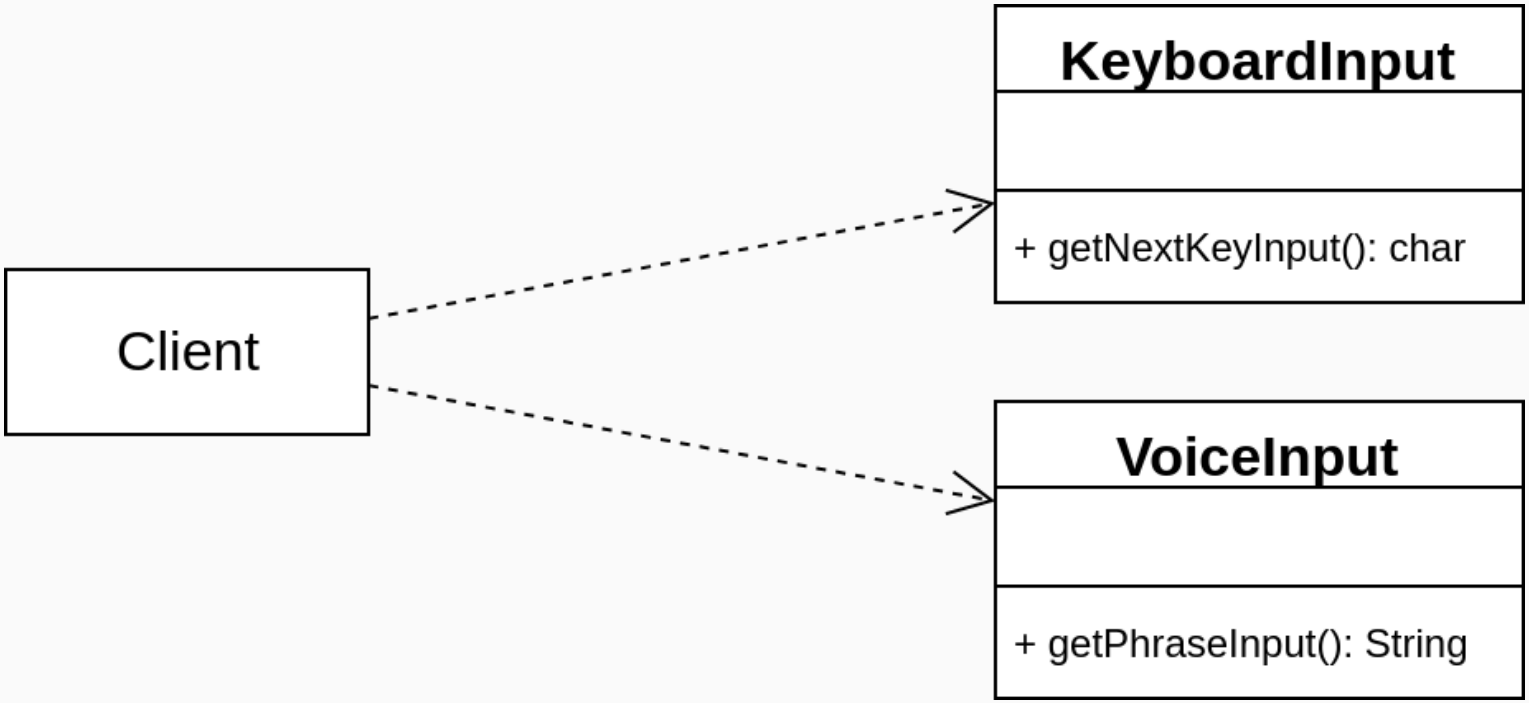
Dependency Inversion Principle

- Implementações de baixo nível podem ser alteradas
- Uso dessas implementações $\rightarrow$ alto acoplamento $\rightarrow$ alterações de dependentes

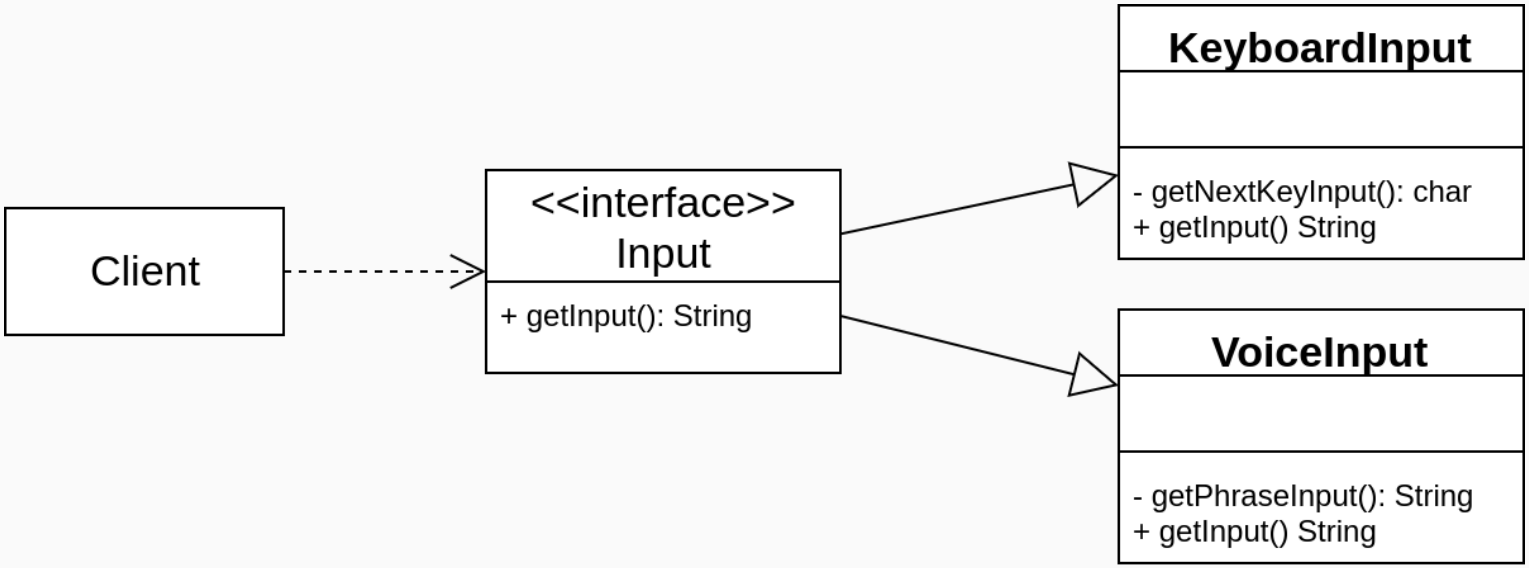
Dependency Inversion Principle

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

ERRADO!


KeyboardInput
+ getNextKeyInput(): char
Client
VoiceInput
+ getPhraseInput(): String

CORRETO!


Client
<<interface>>
Input
+ getInput(): String
KeyboardInput
- getNextKeyInput(): char
+ getInput() String
VoiceInput
- getPhraseInput(): String
+ getInput() String

Conclusões

- Evite repetições de código

Conclusões

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

Conclusões

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