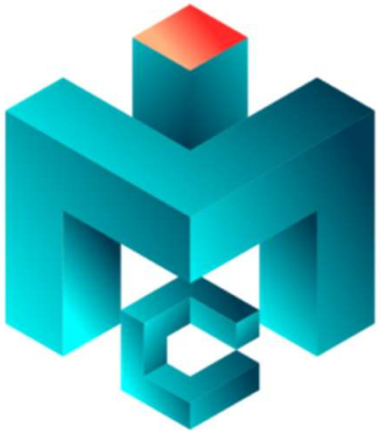


Aula – Introdução a Padrões - Strategy

Disciplina: COM221 – Computação Orientada a Objetos II

Prof: Phyllipe Lima
phyllipe@unifei.edu.br

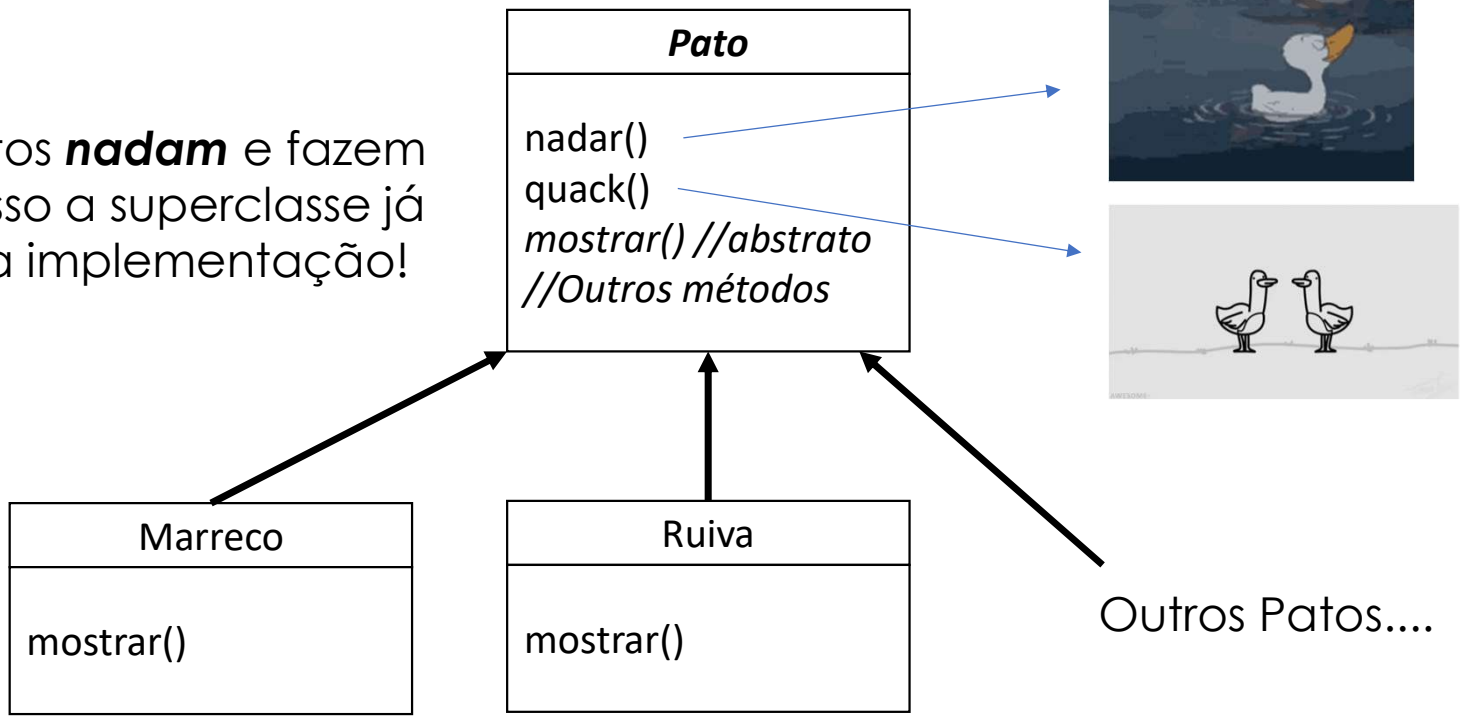
Universidade Federal de Itajubá – UNIFEI
IMC – Instituto de Matemática e Computação

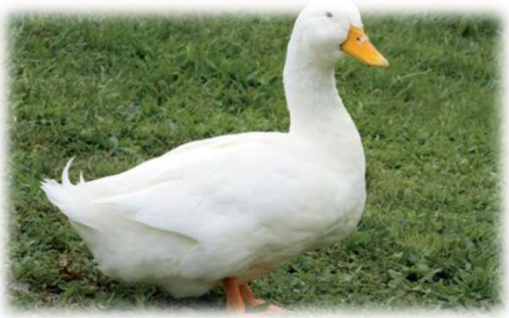
Agenda

- Entender o problema do reuso pela herança
- Conhecer o que são padrões de projetos
- *Strategy* – Meu Primeiro Padrão

DuckSimulator 2.0: Versão Underground

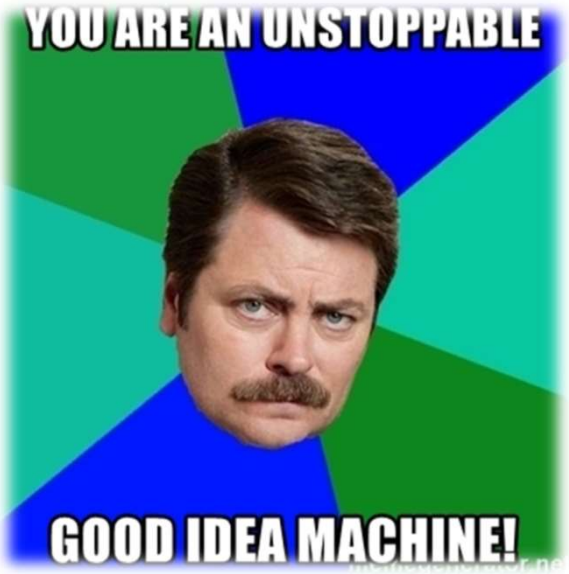
Todos os patos ***nadam*** e fazem ***quack***. Por isso a superclasse já fornece uma implementação!





Ganhando da concorrência!

Patos Voadores!



Agora todos os patos ***voam!*** Problema resolvido!

Pato
nadar() quack() **voar()** *mostrar() //abstrato* *//Outros métodos*

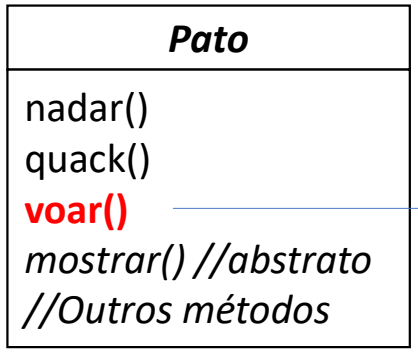
Marreco
mostrar()

Ruiva
mostrar()

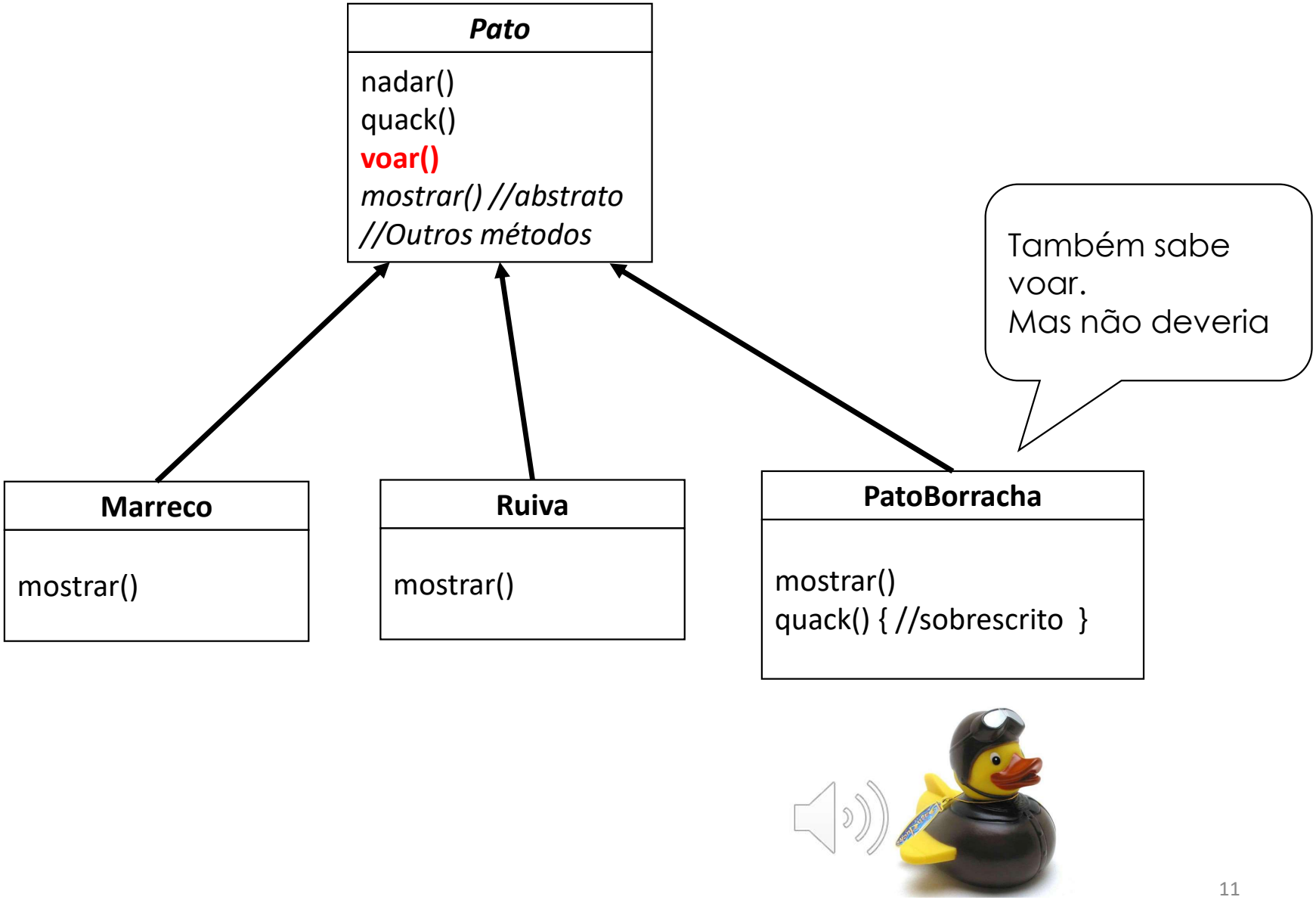
Outros Patos....

Reunião para apresentar a nova versão

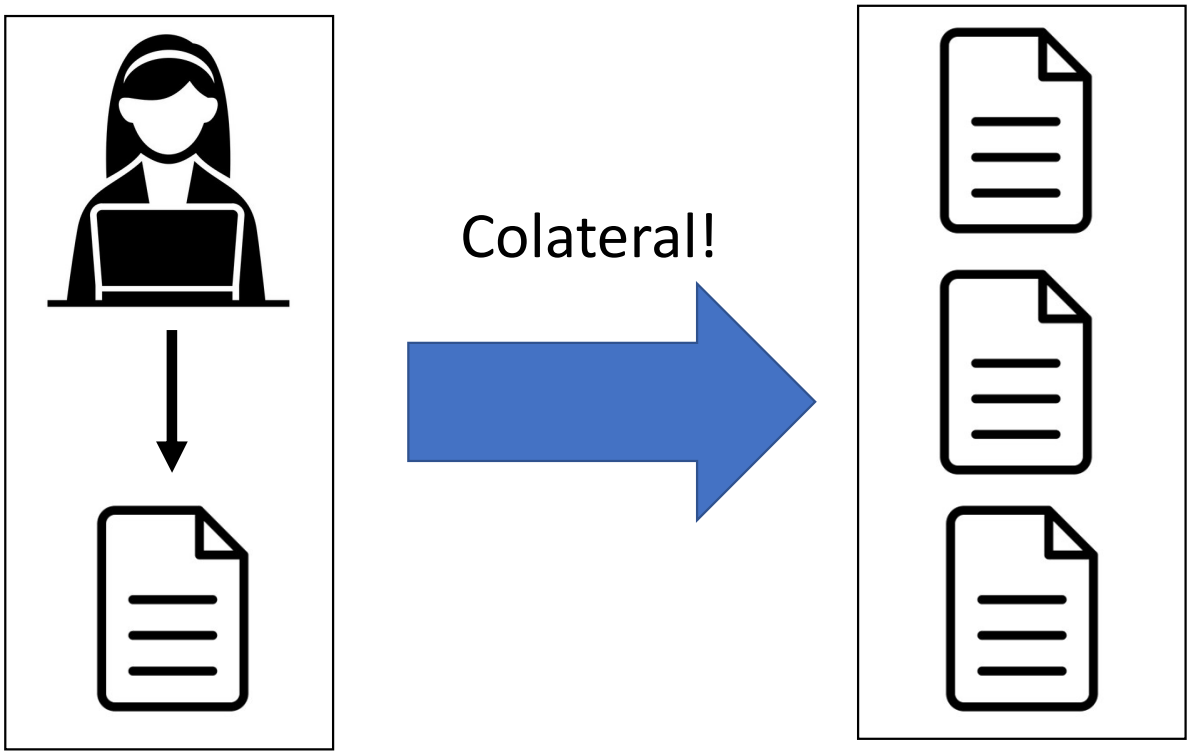
Pato de Borracha voando?



Inserindo comportamento na superclasse, toda subclasse passa a ter esse comportamento também


Pato
nadar()
quack()
voar()
mostrar() //abstrato
//Outros métodos
Também sabe voar.
Mas não deveria
Marreco
mostrar()
Ruiva
mostrar()
PatoBorracha
mostrar()
quack() { //sobrescrito }

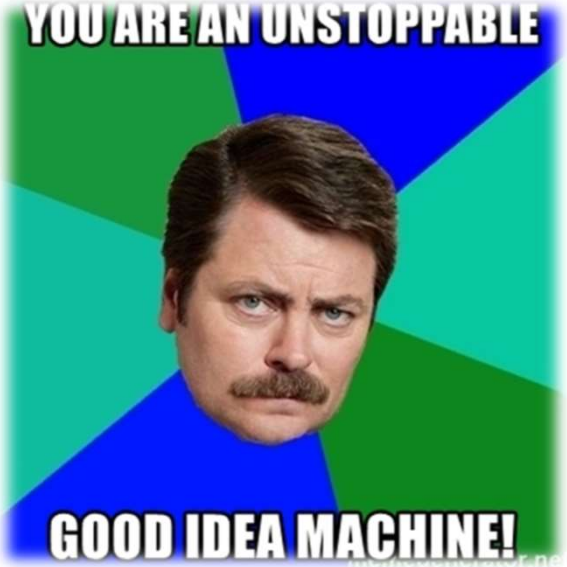
Uma modificação em um trecho do código causou um ***efeito colateral*** em outro trecho

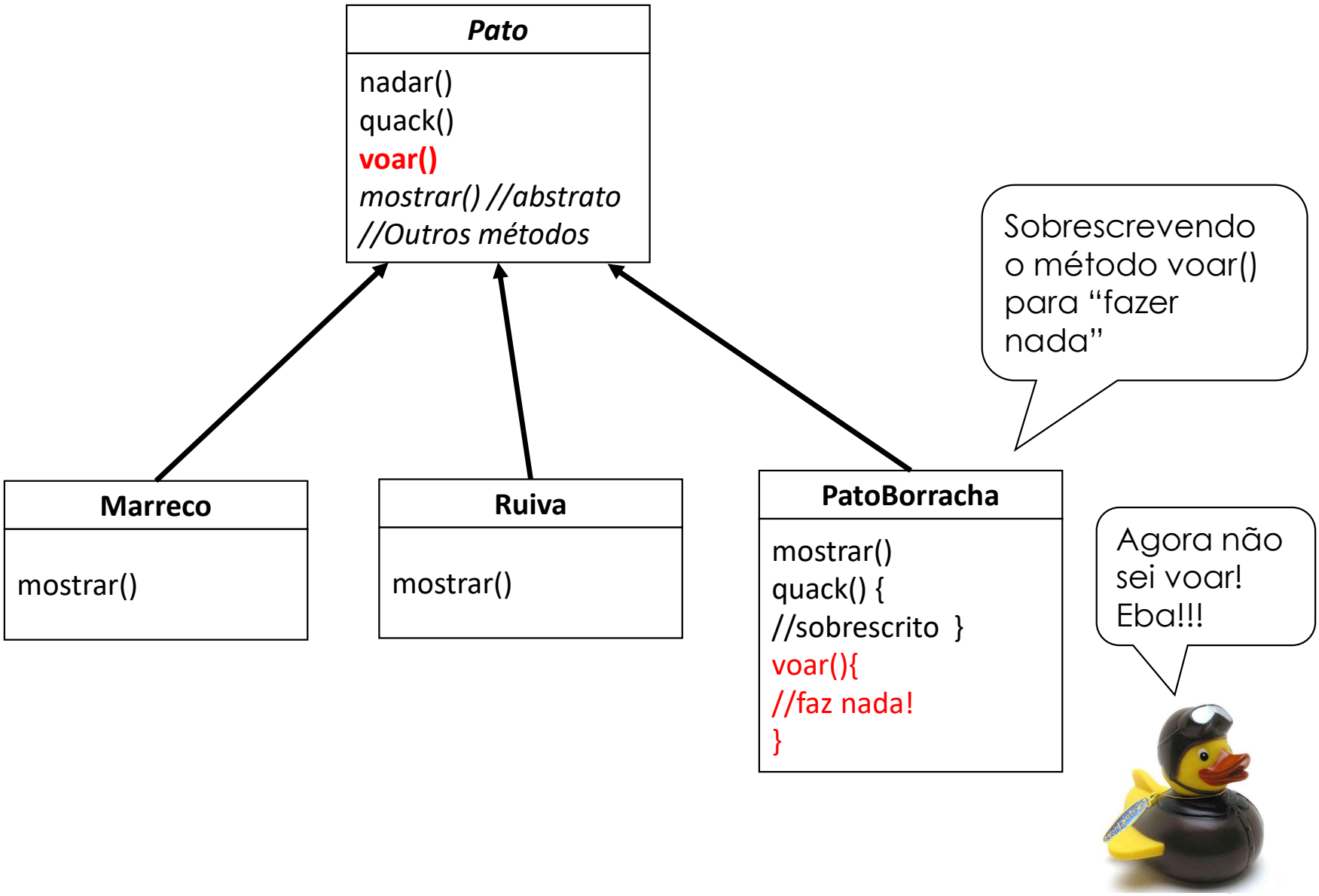


A herança trouxe reuso, mas atrapalhou a ***manutenção***!

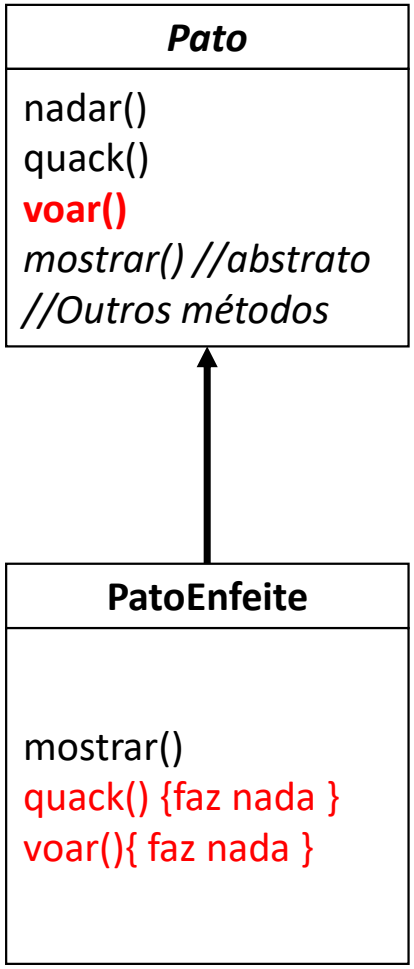
Manutenção

Vamos sobrescrever!




Pato
nadar()
quack()
voar()
mostrar() //abstrato
//Outros métodos
Marreco
mostrar()
Ruiva
mostrar()
PatoBorracha
mostrar()
quack() {
//sobrescrito }
voar(){
//faz nada!
}
Sobrescrevendo o método voar() para "fazer nada"
Agora não sei voar! Eba!!!

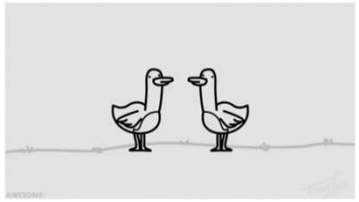
Não estamos criando mais problemas?

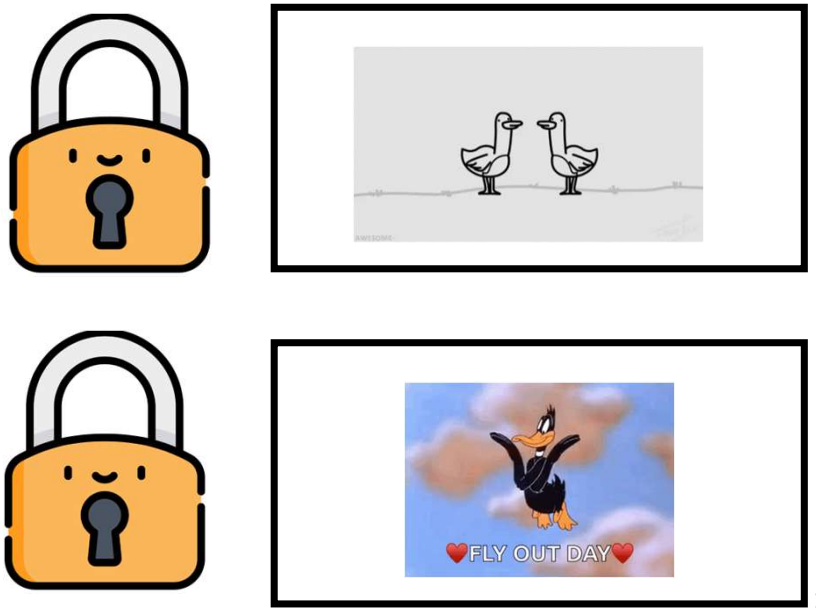


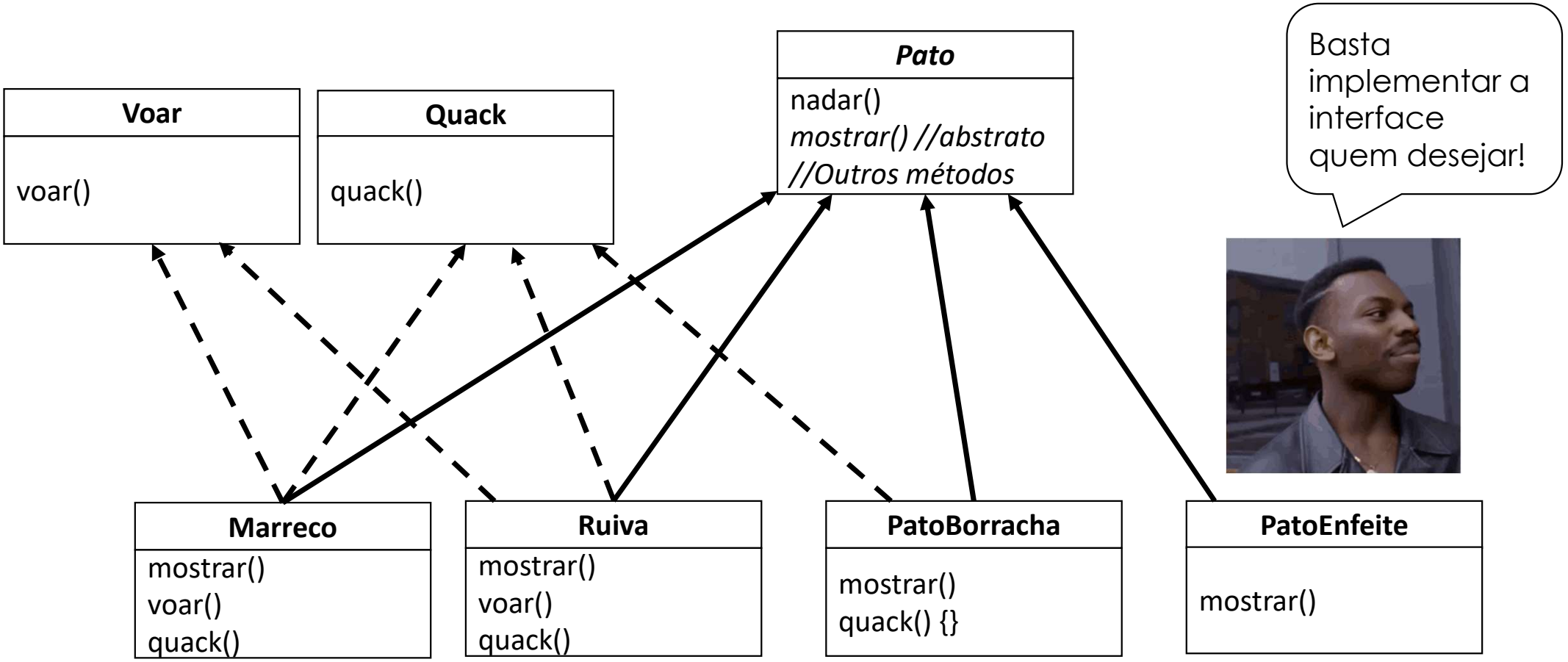
Eu também não sei voar! Sou de enfeite. Alias, nem falar eu sei!

Além de violar princípios, estamos criando um problema grave de manutenção!

Voar e ***Falar(Quack)*** são comportamentos!
Quando queremos ***encapsular*** comportamento utilizamos interfaces!



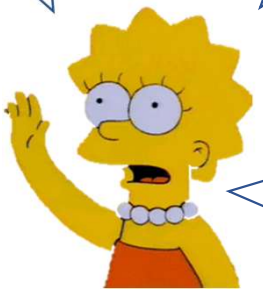



Voar
voar()
Quack
quack()
Pato
nadar()
mostrar() //abstrato
//Outros métodos
Basta implementar a interface quem desejar!
Marreco
mostrar()
voar()
quack()
Ruiva
mostrar()
voar()
quack()
PatoBorracha
mostrar()
quack() {}
PatoEnfeite
mostrar()

Chega de Pato de Borracha voando!

Não haverá duplicação de código?

E se mais patos *voam* da mesma forma?



Acabou o reuso!

Chega de Pato de Borracha voando!

Não conseguimos mais reaproveitar comportamento!
Continuamos com problema de manutenção!

Qual é única certeza que temos em software?

Mudança!

Como o comportamento constantemente se altera nas subclasses, herança não está resolvendo

Usando ***interface*** perdemos a oportunidade de reusar código, pois não existe implementação dentro da interface!

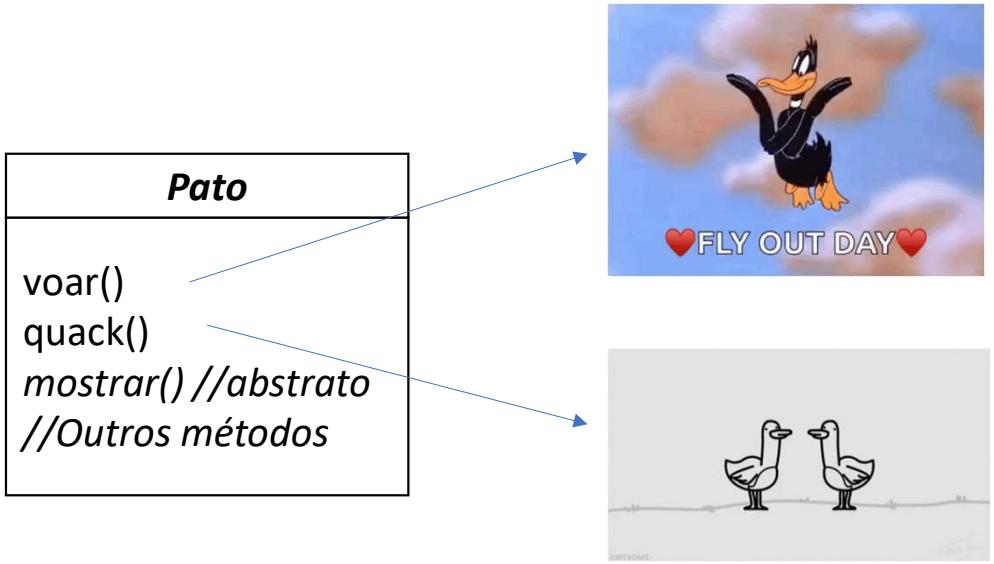
Princípio:

Identifique as partes que se alteram com frequência e separe das mais estáveis!

Assim, minimizamos possíveis efeitos colaterais de mudança de código!

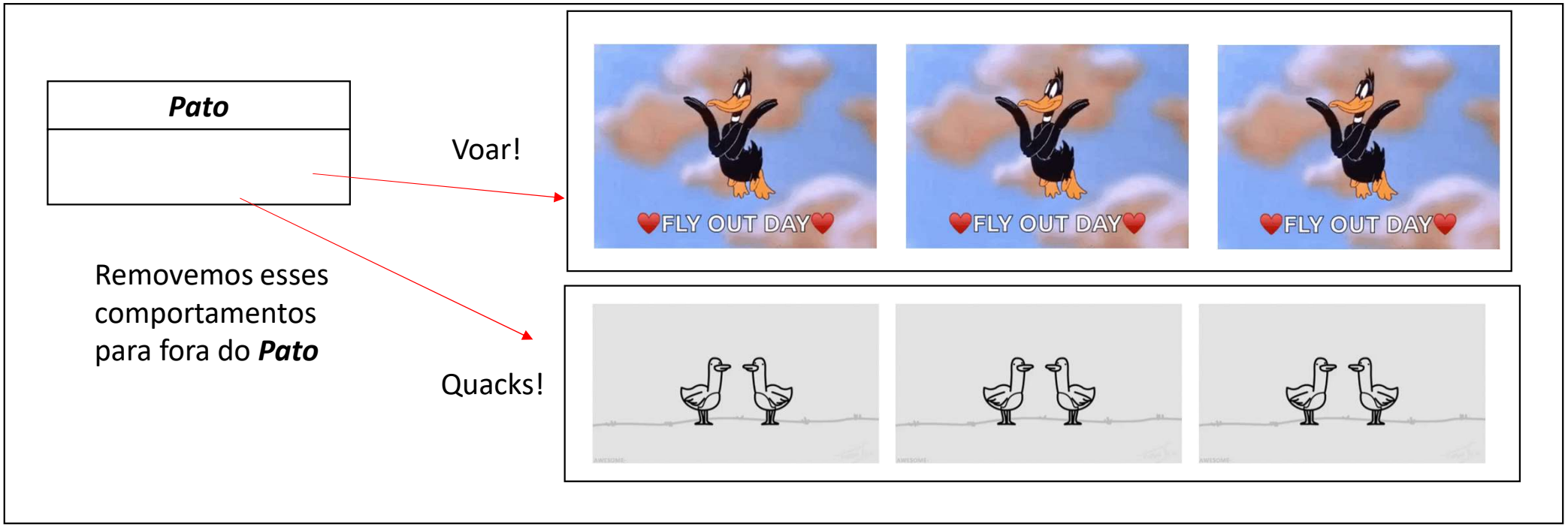
Mais flexibilidade

Nesse exemplo o que se modifica com frequência?

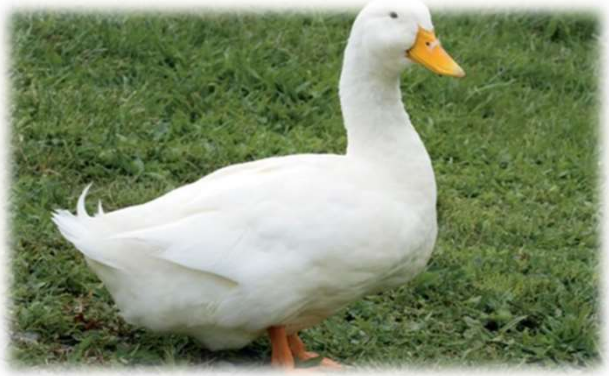


Vamos remover esse "comportamento"

Voar e ***Quack*** ganharão um conjunto de classes dedicadas!



Como queremos flexibilidade, iremos ***atribuir*** comportamento.

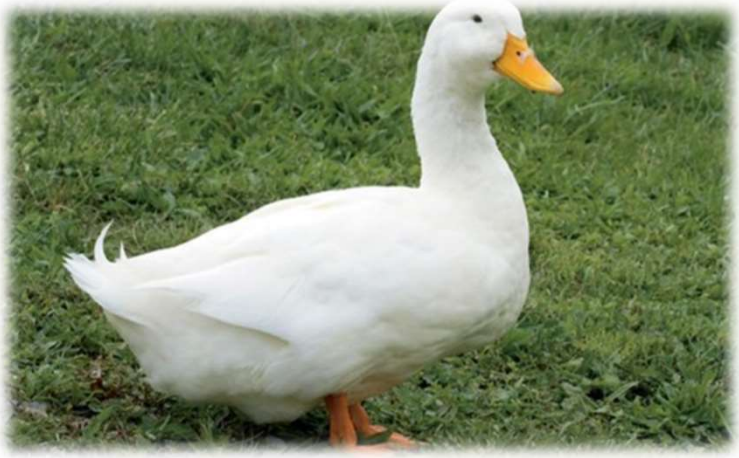


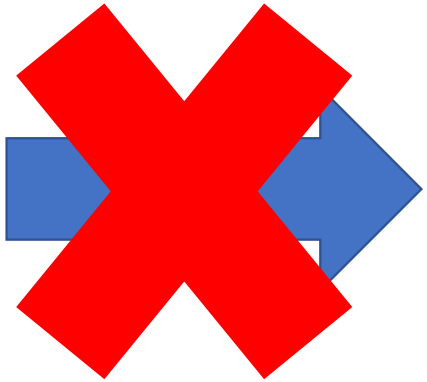
Podemos ***inicializar*** um pato com um tipo de voo, mas queremos ***mudar*** isso em ***tempo de execução***!

Princípio:

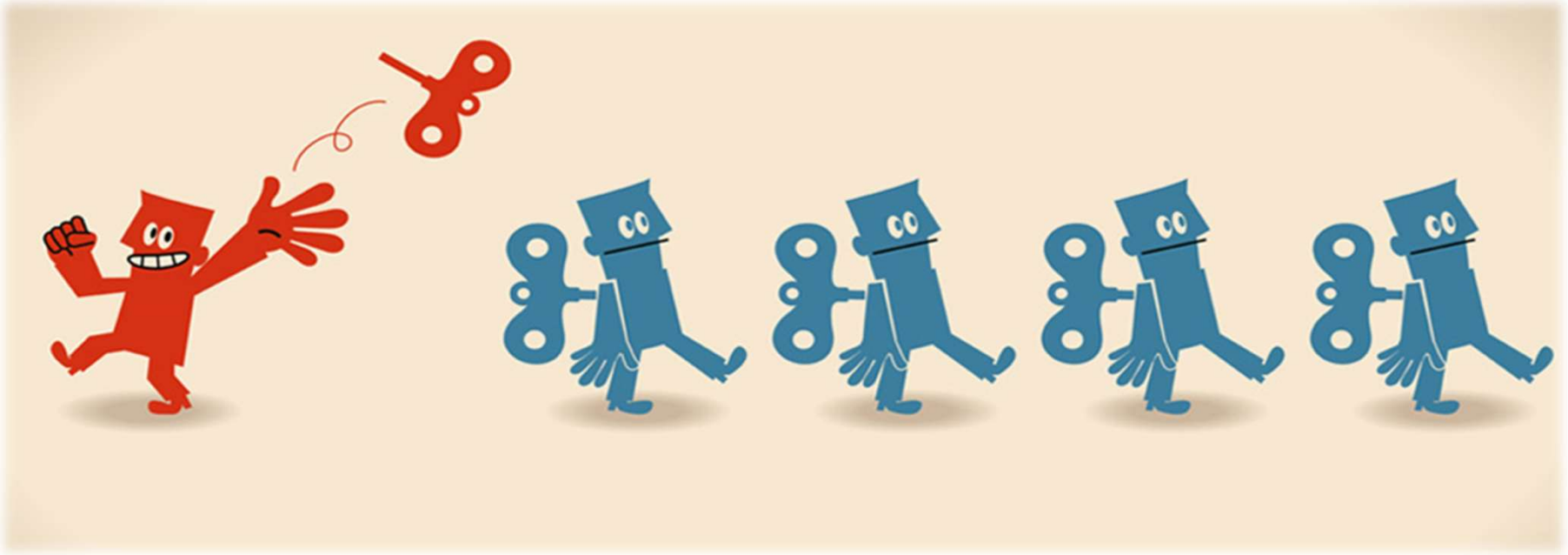
Programe para uma interface (super-tipo) e não uma implementação concreta!

Cada comportamento será uma ***interface***. Mas não serão as classes do pato que irão implementar.





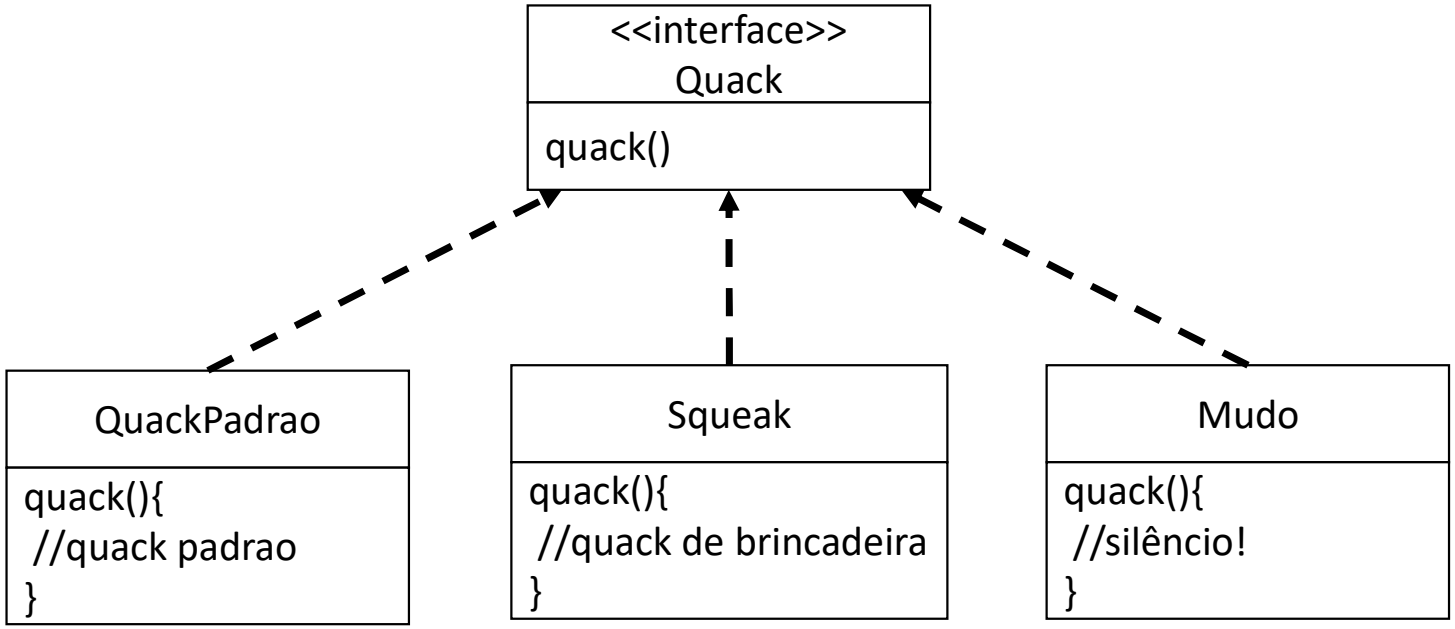
Isso é o contrário do que fizemos antes, onde a implementação estava na própria superclasse (Pato) ou em alguma subclasse.

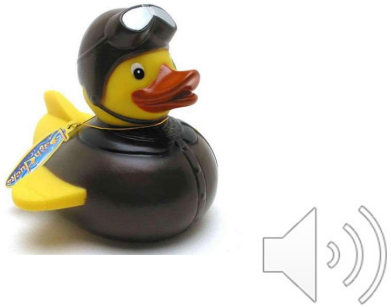


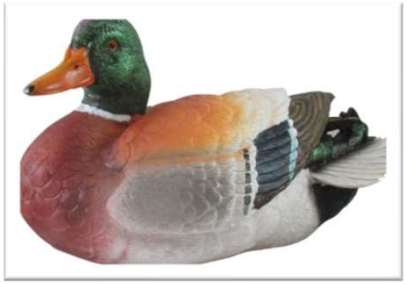
Em ambos os casos o comportamento estava travado sem oportunidade de mudar, a não ser através de escrita de código.


❤️FLY OUT DAY❤️

<<interface>>
Voar
voar()
VoarComAsas
voar(){
//voando com
asas
}
NaoPodeVoar
voar(){
//não voa. Faz
nada!
}

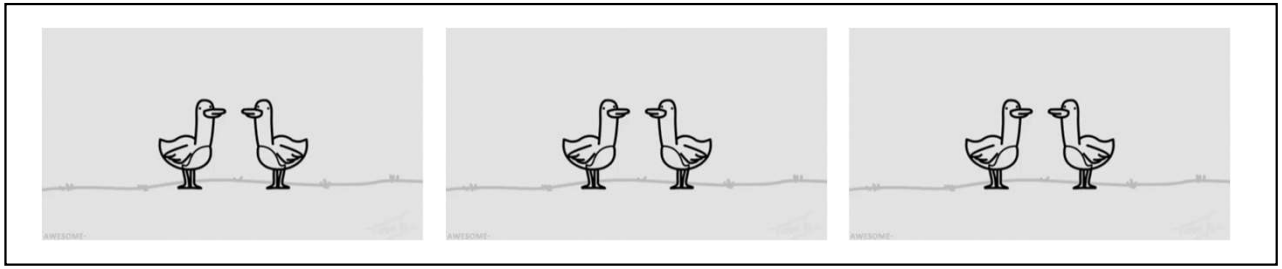

<<interface>>
Quack
quack()
QuackPadrao
quack(){
//quack padrao
}
Squeak
quack(){
//quack de brincadeira
}
Mudo
quack(){
//silêncio!
}





Unindo os detalhes!

Pato
Voar voar Quack quack
executaVoo() executaQuack() ***//outros métodos da //superclasse***



Como inicializar?

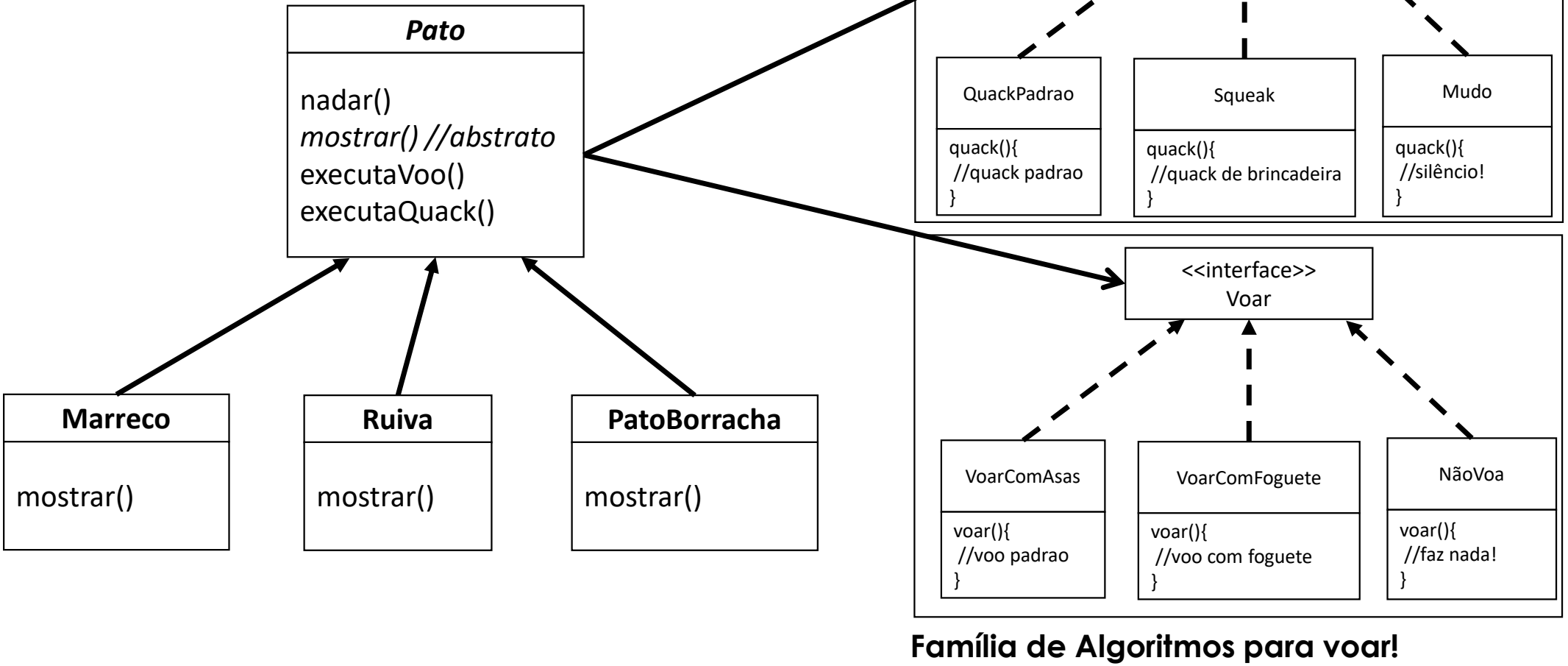
```java
public class Marreco extends Pato {

    public Marreco() {
        voar = new VoarComAsas();
        quack = new QuackPadrao();
    }

    @Override
    public void mostrar() {
        System.out.println("Eu sou um Marreco!");
    }
```

Podemos inicializar dentro do construtor o comportamento padrão!

Temos também *setters* para modificar em tempo de execução!

DuckSimulator 2.0: Versão Underground

O *Pato* usa uma família de algoritmos que estão encapsulados!



Observe alguns relacionamentos:

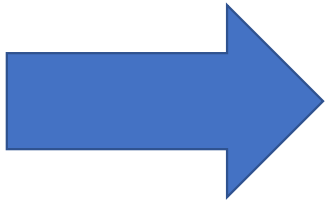
- Marreco ***É UM*** Pato
- VoarComAsas ***IMPLEMENTA*** Voar
- Pato ***TEM UM*** Voar

No relacionamento ***TEM UM*** as classes estão usando a ***composição***
Ao invés de ***herdar*** o comportamento, ele está sendo composto

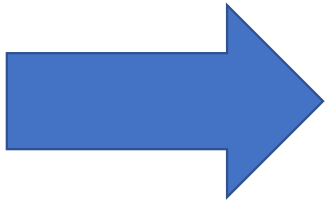
Princípio:

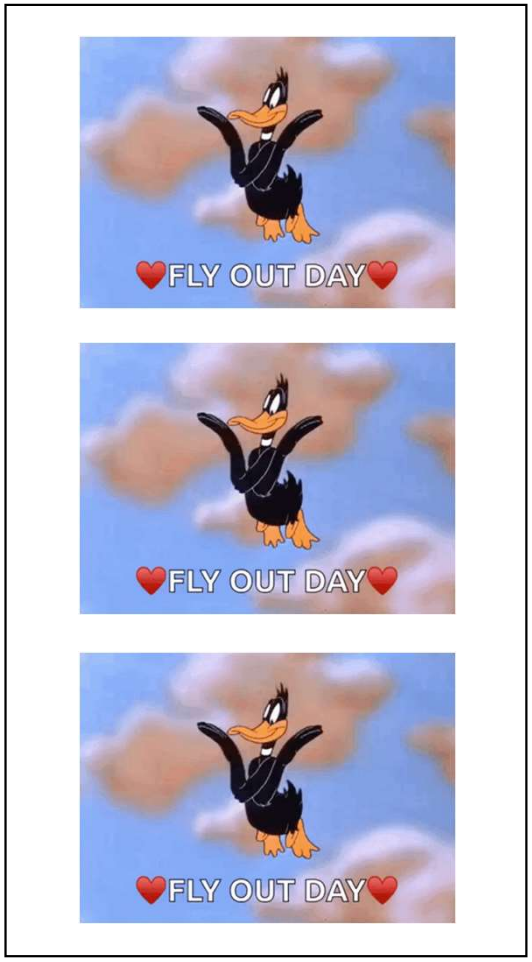
Prefira composição à herança

A composição nos fornece muita flexibilidade!



Encapsula uma família de algoritmos



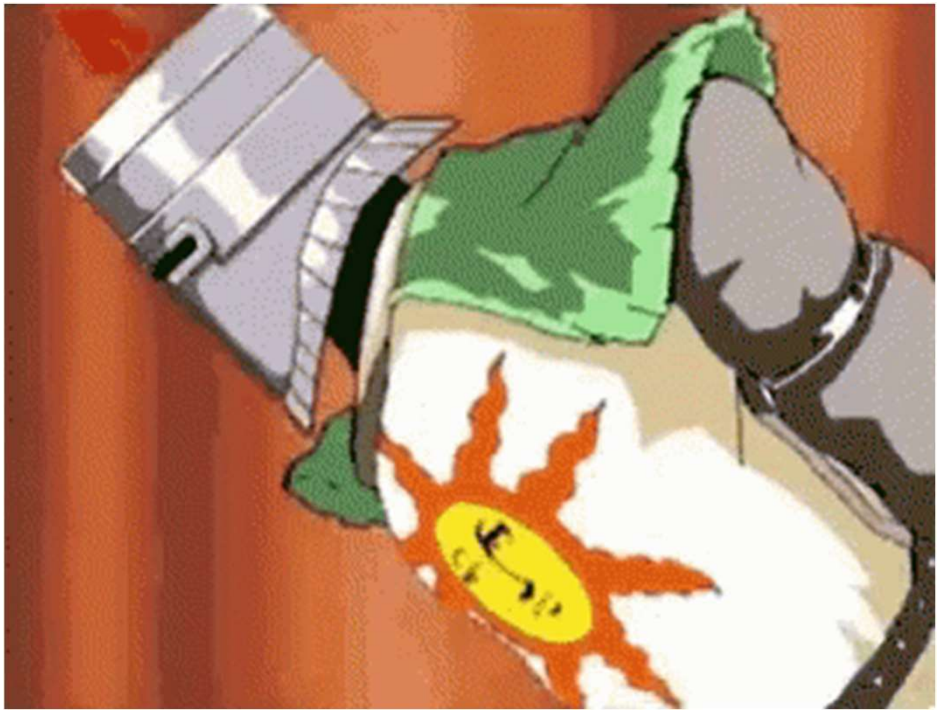


Permite mudar o algoritmo em tempo de execução

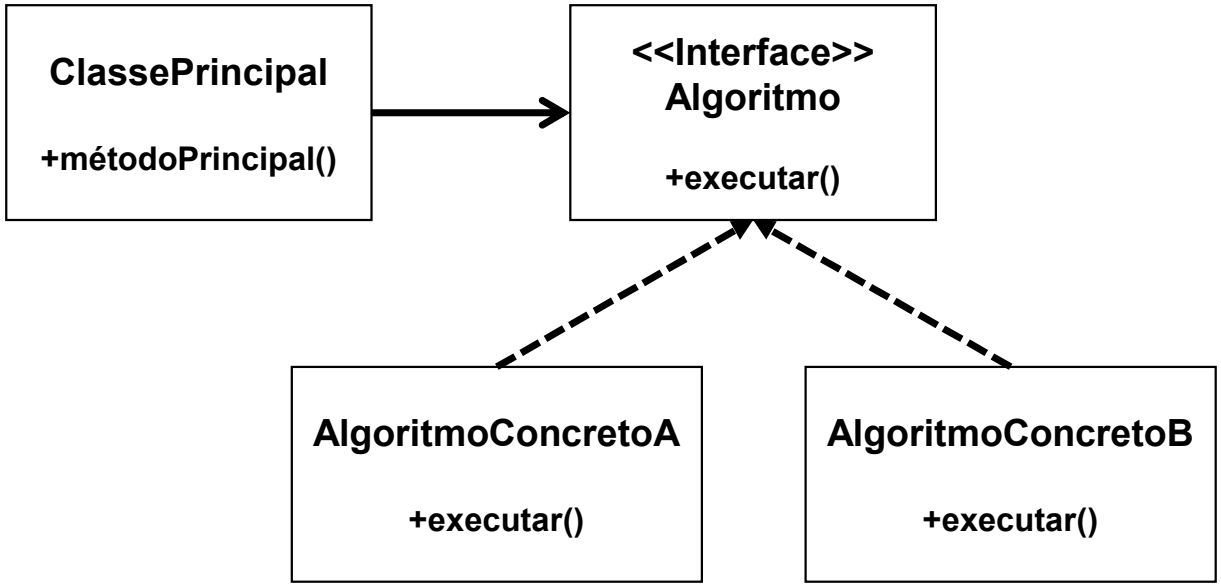
Parabéns :)
Acabamos de usar um ***design pattern*** para resolver o problema!

O primeiro de muitos!

Strategy



UML genérico do ***Strategy***!



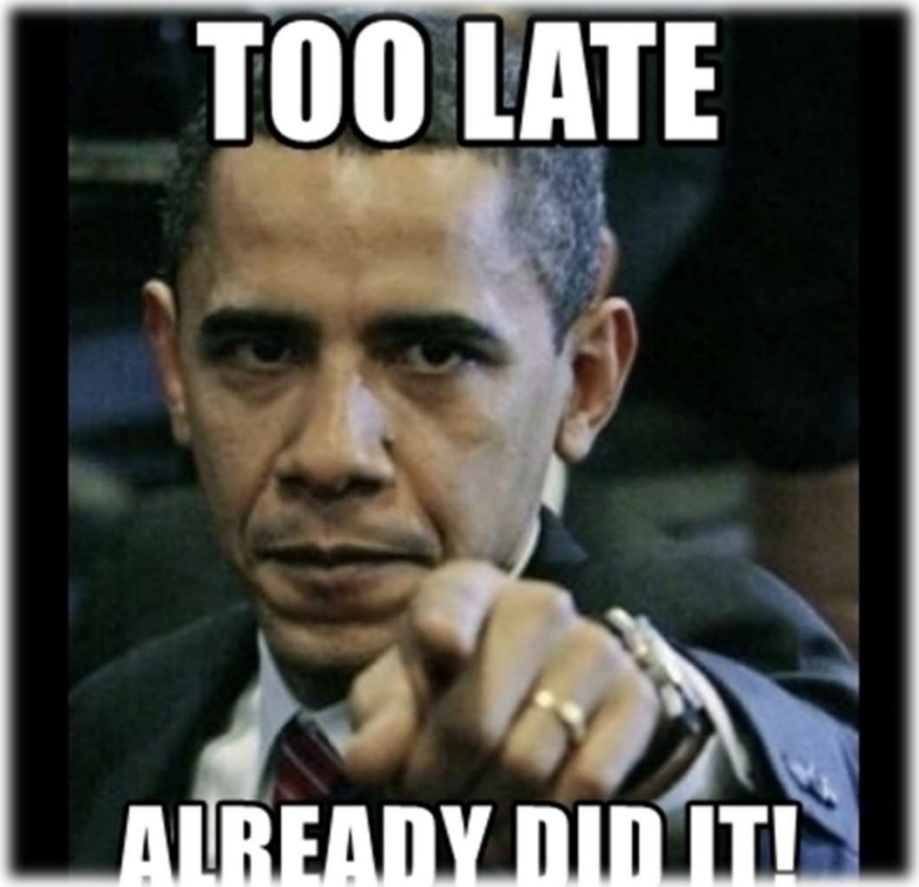
Strategy:

Defina uma família de algoritmos, as encapsule e tornem intercambiáveis.
Cada algoritmo pode variar de forma ***independente*** do cliente!

O que é um padrão?

Uma solução para um determinado problema em um contexto.

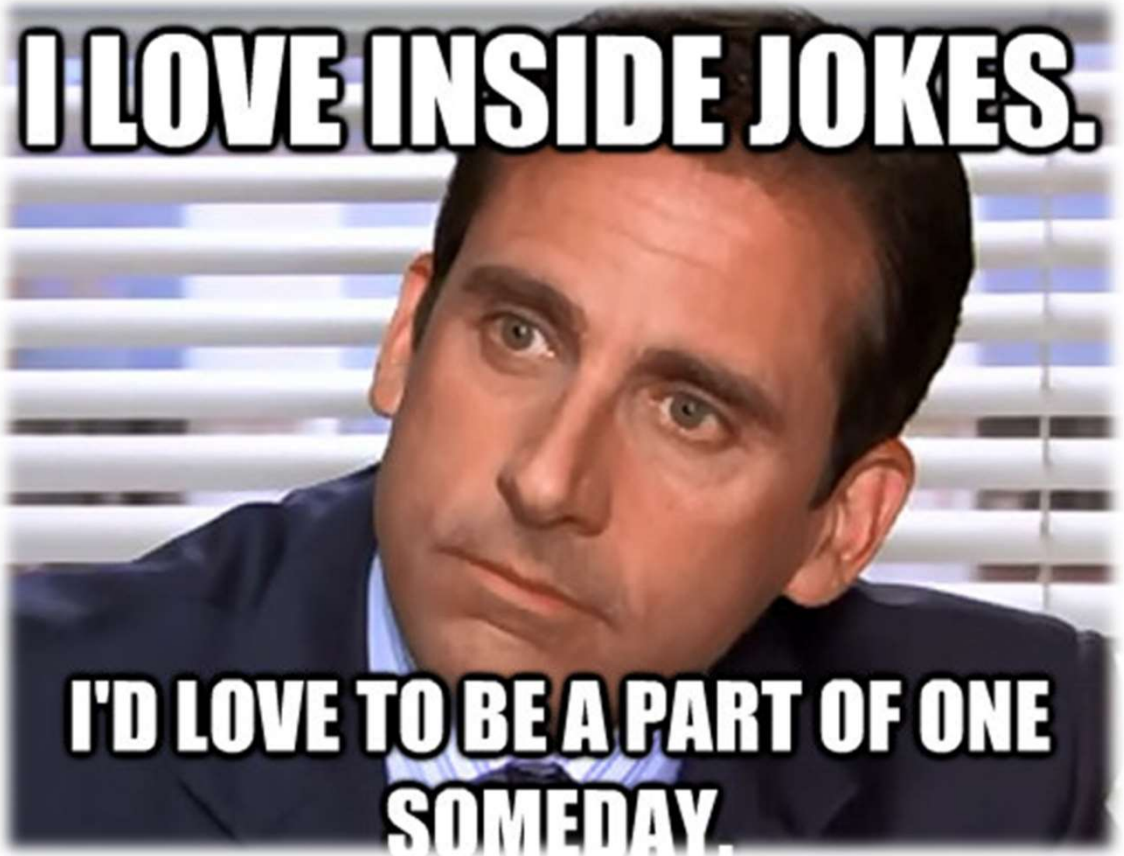
Uma solução que já tenha sido utilizada com sucesso.



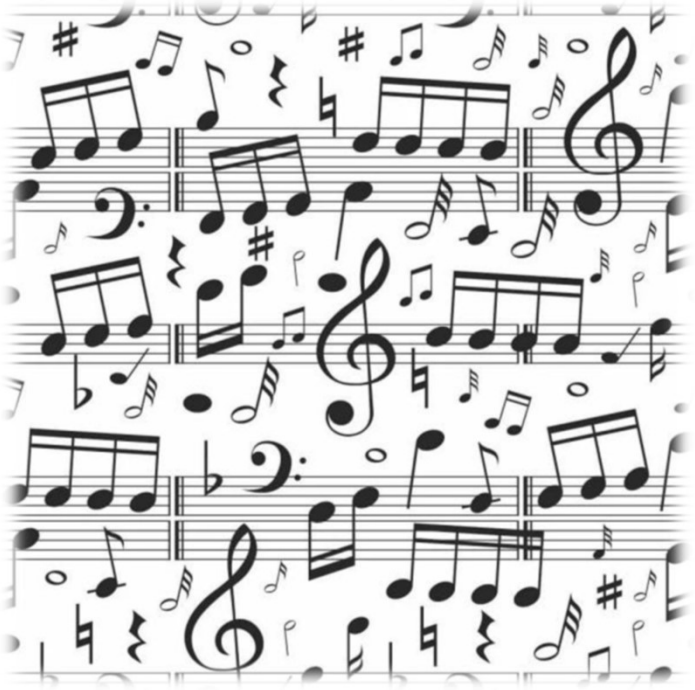
Não descreve soluções novas.

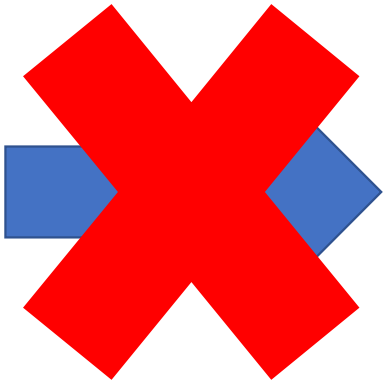
A ideia de padrões vem da Engenharia Civil e foi disseminada na comunidade de software pelo livro escrito pelo GoF (***Gang of Four***).

Conhecer os padrões criam um vocabulário comum!



Conhecer os princípios OO não nos torna bom programador OO,





Design OO deve ser reutilizável, extensível e manutenível

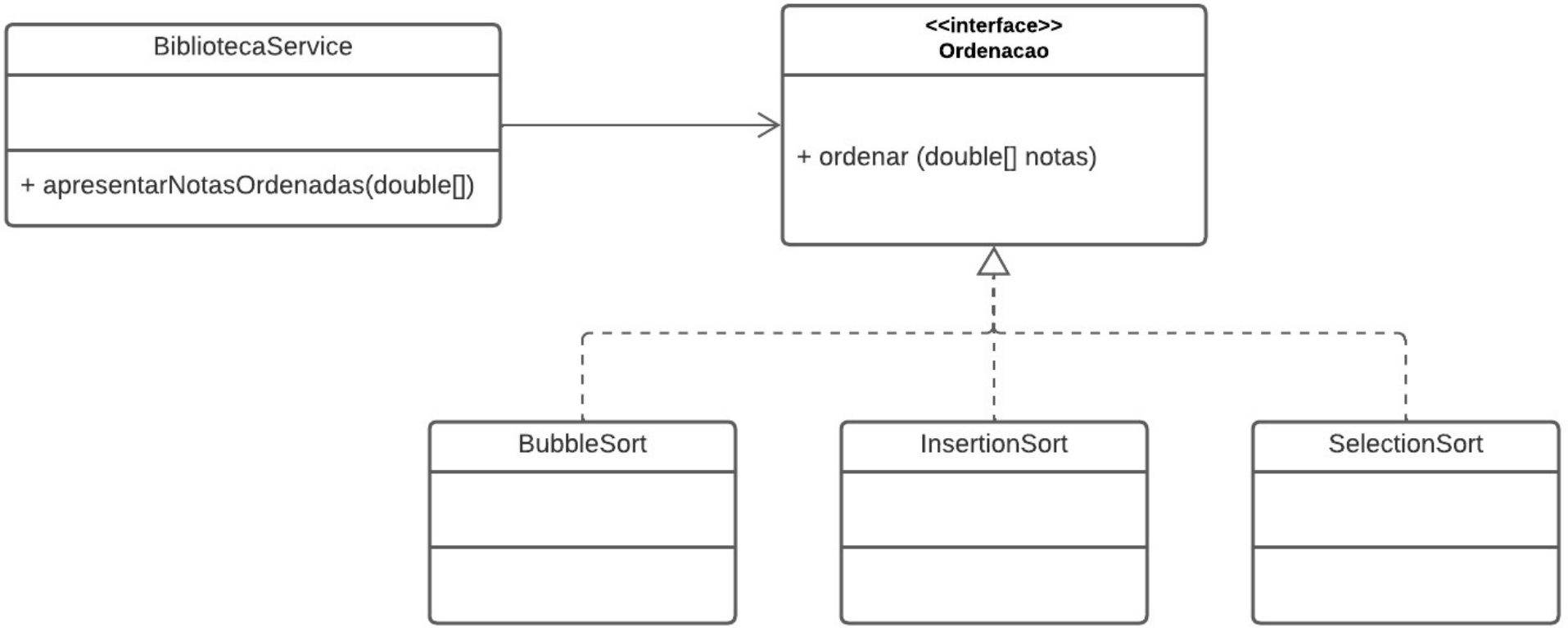
A maioria dos padrões tentam resolver o problema de mudança

Padrão não é um código, é uma solução

Exercício Proposto:

- Utilize o ***design pattern*** Strategy para implementar um sistema capaz de ordenar dados.
- O programa deverá permitir trocar o algoritmo de ordenação em tempo de execução.
- Utilize sua linguagem de programação favorita.
- Implemente os algoritmos Bubble Sort, Selection Sort e Insertion Sort

Diagrama:


BibliotecaService
+ apresentarNotasOrdenadas(double[])
<<interface>>
Ordenacao
+ ordenar (double[] notas)
BubbleSort
InsertionSort
SelectionSort

Bubble Sort:

```
public void ordenar(double[] notas) {

   double aux, fim = notas.length;
   boolean continua = false;
   do {
      continua = false;
      for(int i = 0; i < fim-1; i++) {
         if(notas[i] > notas[i+1]){
            aux = notas[i];
            notas[i] = notas[i+1];
            notas[i+1] = aux;
            continua = true;
         }
      }
      fim--;
   }while(continua != false);
}
```

Selection Sort:

```
public void ordenar(double[] notas) {

   int menor;
   double aux;
   int tam = notas.length;
   for (int i = 0; i <  tam - 1; i++){
      menor = i;
      for(int j = i + 1; j < tam; j++){
         if(notas[j] < notas[menor]){
            menor = j;
         }
      }
      if(menor != i){
         aux = notas[i];
         notas[i] = notas[menor];
         notas[menor] = aux;
      }
   }
}
```

Insertion Sort:

```
public void ordenar(double[] notas) {
    double atual;
    int j, tam = notas.length;
    for (int i = 1; i < tam; i++){
        atual = notas[i];
        for(j = i; (j > 0) && (atual < notas[j-1]); j--){
            notas[j] = notas[j-1];
        }
        notas[j] = atual;
    }
}
```

Referência

[1] This was once revealed to me in a dream.

- Capítulo 6 do livro Engenharia de Software Moderna
  - Padrões de Projeto
  - https://engsoftmoderna.info/cap6.html

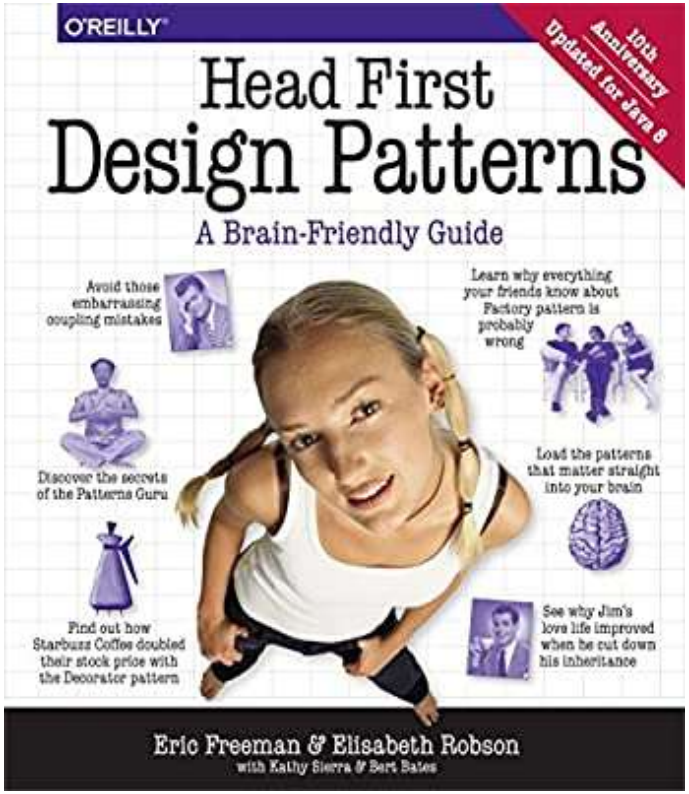
Referência - Complementar

[1] This was once revealed to me in a dream.

- Design Patterns com Java
- Cap 1 – Intro Design Pattern

Referência - Complementar

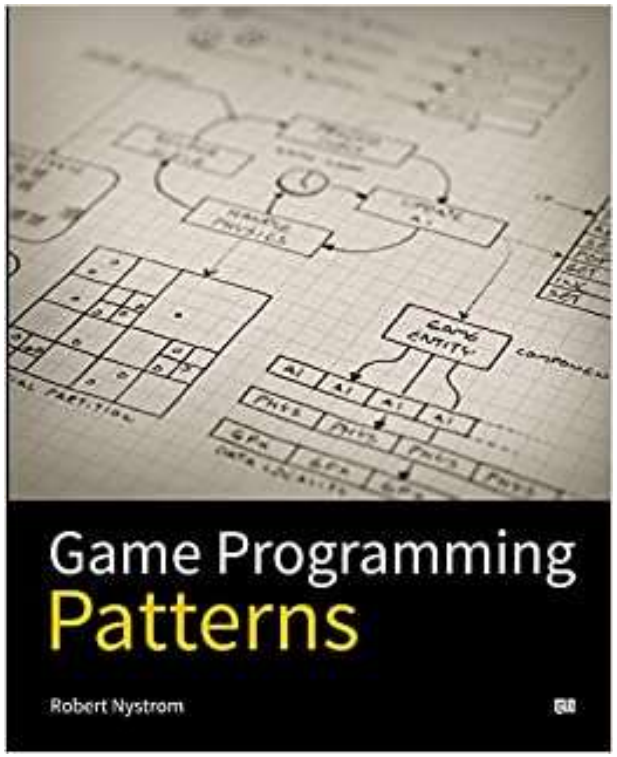
[1] This was once revealed to me in a dream.



- Head First Design Patterns
- Edição: 2
- Cap 1

Referência - Complementar

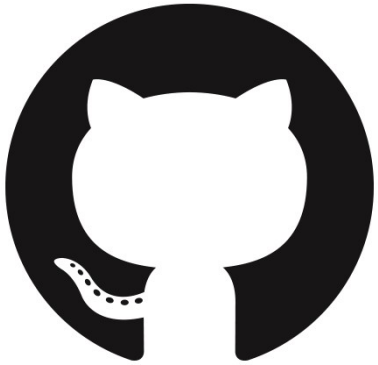
[1] This was once revealed to me in a dream.

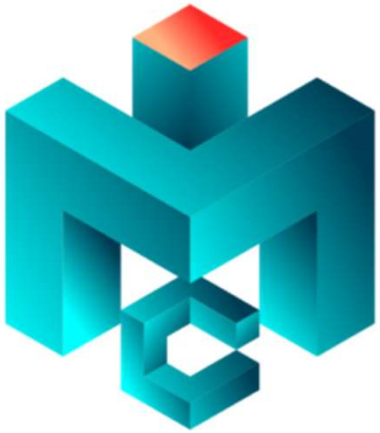


- Game Programming Patterns
- Versão HTML
  - https://gameprogrammingpatterns.com

Implementações

- https://github.com/phillima-unifei/COM221





Aula – Introdução a Padrões - Strategy

Disciplina: COM221 – Computação Orientada a Objetos II

Prof: Phyllipe Lima
phyllipe@unifei.edu.br

Universidade Federal de Itajubá – UNIFEI
IMC – Instituto de Matemática e Computação