Análise, Projeto e Implementação

de Sistemas 2

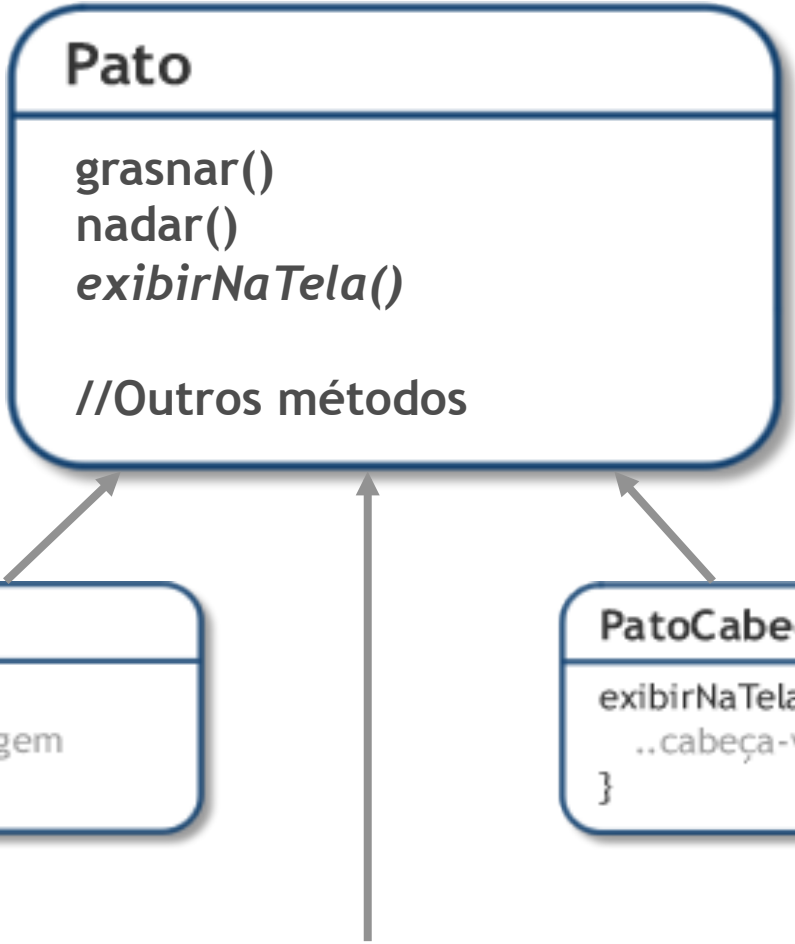
Professor: Eduardo Mendes

Jogo com Patos

- A simulação de um lago com patos
- SimUPato
- Grande variedade de espécies
- Técnicas OO
  - Superclasse: Pato
    - Herdada por todos os outros tipos de pato

Começando com Padrões

- Todos os patos grasnam e nadam
- A superclasse cuida da implementação

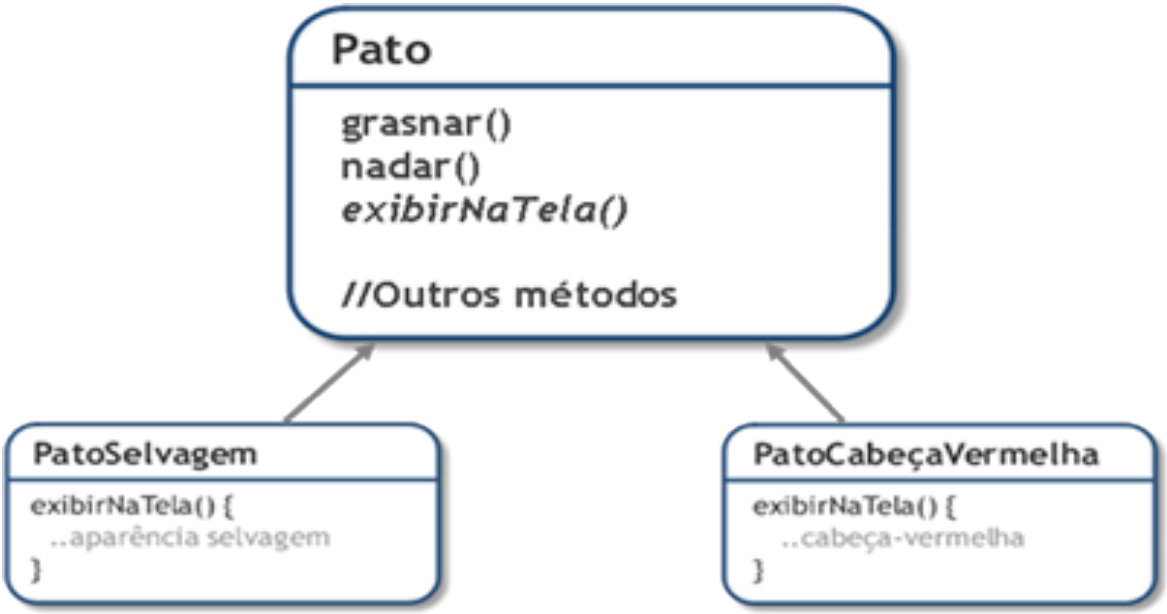


- O método exibirNaTela() é abstrato já que todos os subtipos de pato são diferentes

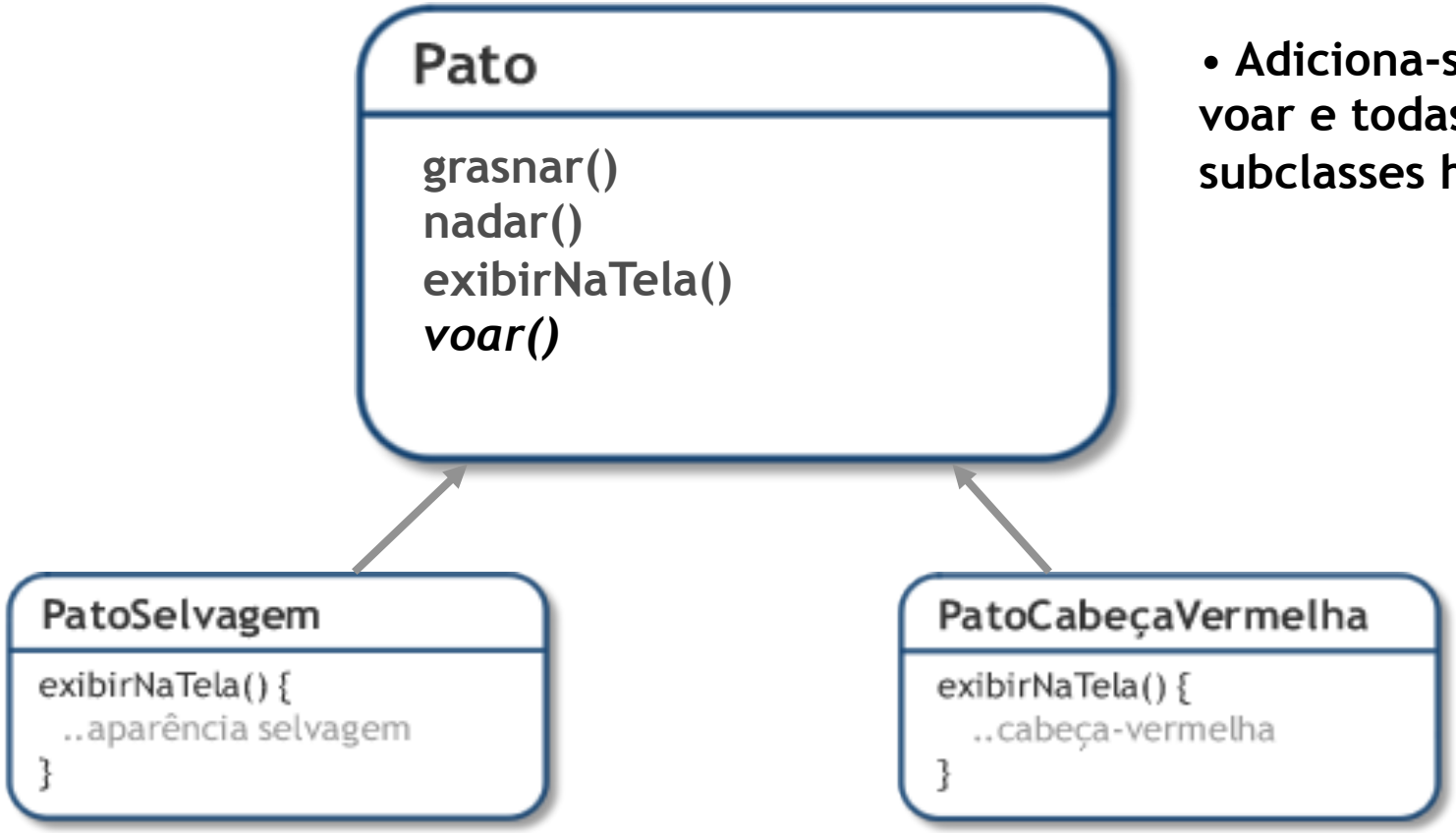
Requisitos

- Alteração nos Requisitos

- ADICIONAR PATOS QUE PRECISAM VOAR

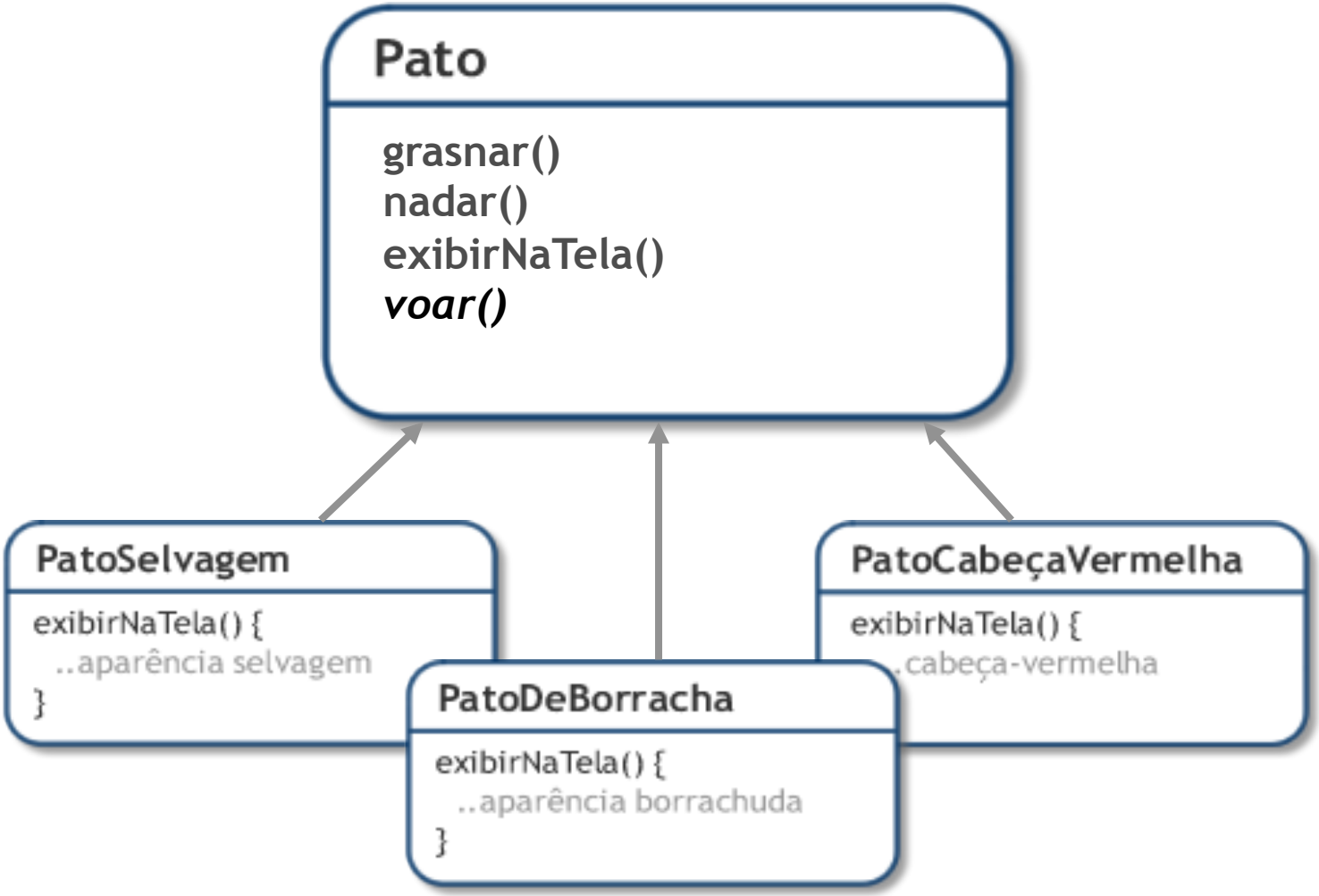


Começando com Padrões



- Adiciona-se o método voar e todas as subclasses herdam

Começando com Padrões



Problemas

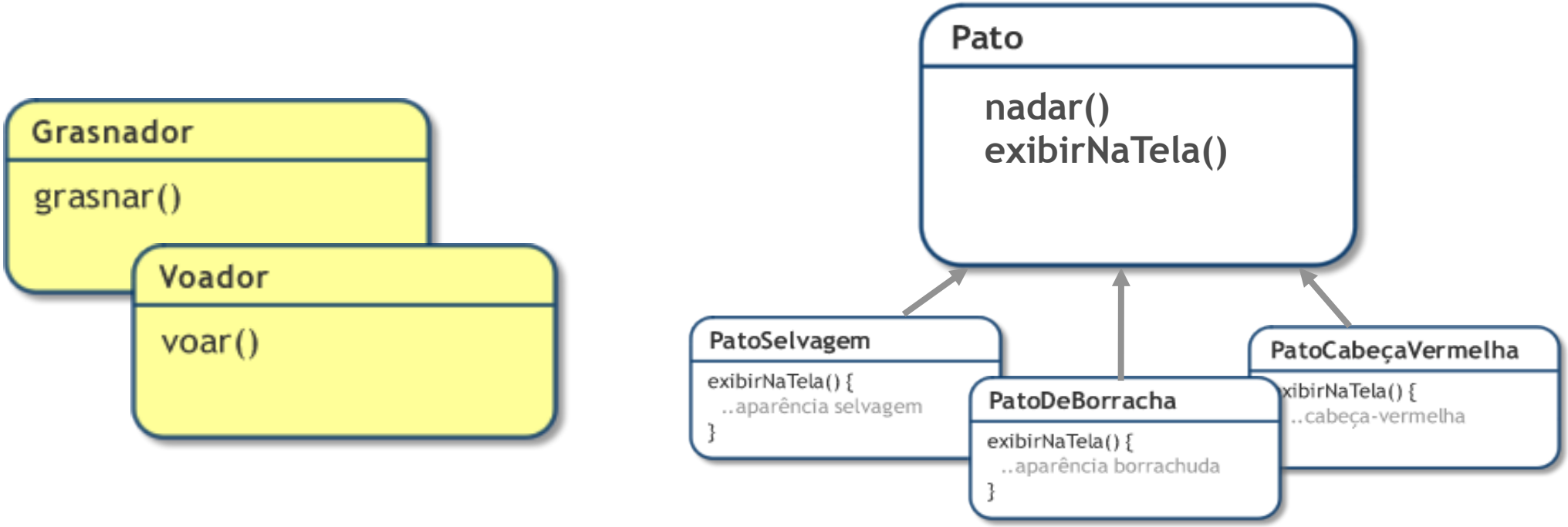
- O que pode parecer ser um excelente uso para a herança para fins de reutilização pode não ser tão eficiente quando se trata de manutenção

- Problema
  - Nem todas as classes deveriam voar
  - Uma atualização localizada no código causou um efeito colateral não local
    - Patos de borracha voadores

Opções

- Sobrescrever o método voar() do pato de borracha

- Que tal uma interface?????



ALTERAÇÃO

HERANÇA

- A HERANÇA não funcionou muito bem
  - Comportamento de patos ficam sempre alterando
  - Nem todas as subclasses necessitam ter o mesmo comportamento
  - Interfaces parecem “OK”
    - Não possuem implementação
      - Não há reutilização de código
    - Sempre que modificar o comportamento
      - Monitoração
      - Alteração

Princípio de Design

- "Identifique os aspectos de seu aplicativo que **variam** e **separe-os** do que permanece igual"
  - Pegue o que variar e "encapsule" para que isso não afete o restante do código
  - Menos conseqüências indesejadas
  - Mais flexibilidade

Mesma coisa

- “Pegue as partes que variam e encapsule-as para depois poder alterar ou estender as partes que variam sem afetar as que não variam”
- Base de quase todos os padrões de projeto
- Hora de voltar aos patos

Aos patos

- O que está variando?
  - voar()
  - grasnar()

  - Criaremos dois conjuntos de classes
  - Totalmente separados
  - 1º → Voar
  - 2º → Grasnar
  - Cada conjunto contém todas as implementações possíveis de comportamento

Retira-se os métodos da classe e cria-se classes para estes comportamentos

Classe Pato

Comportamentos de vôo

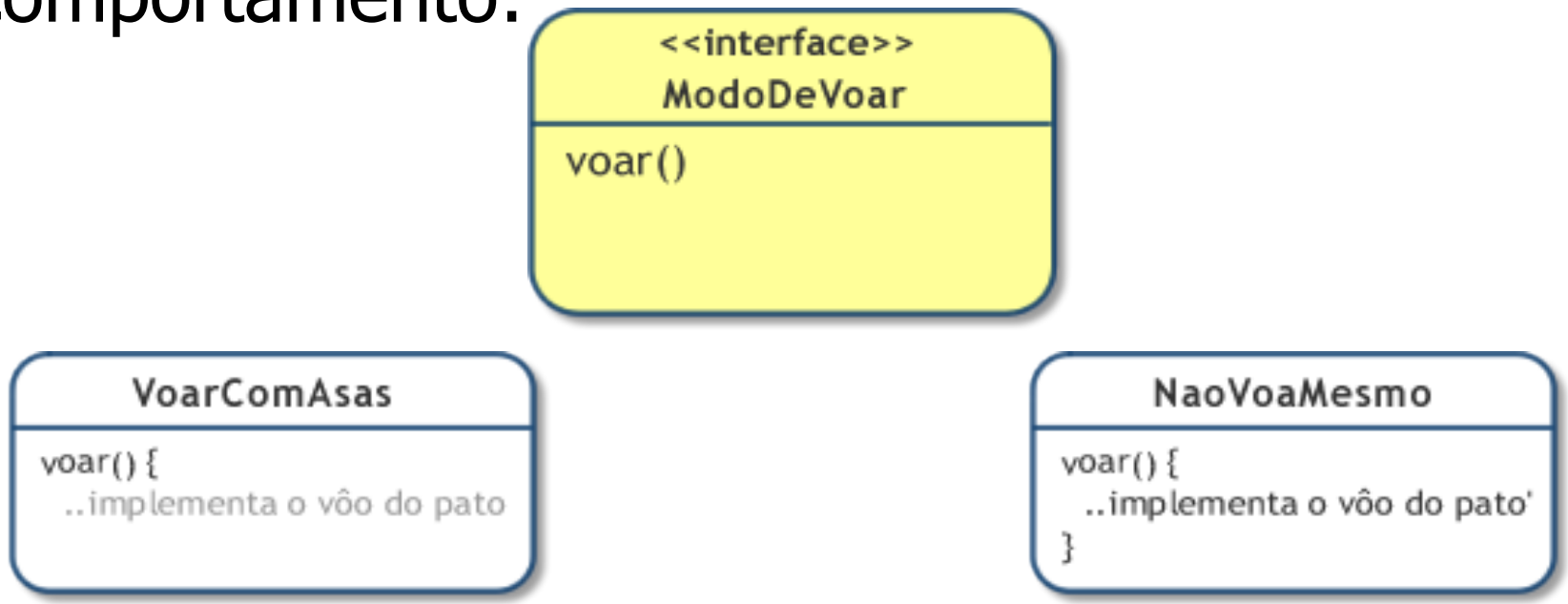
Comportamentos de grasnar

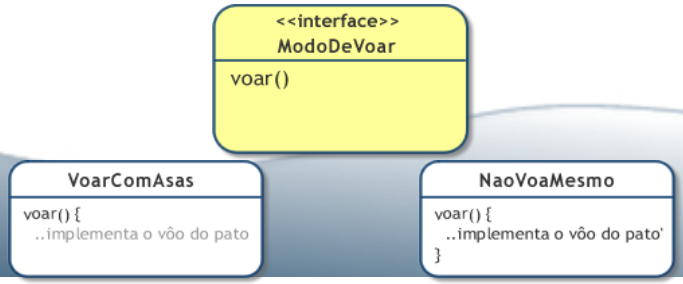
Flexibilidade

- Como desenvolver comportamento e manter a flexibilidade?
- Podemos alterar o comportamento de forma dinâmica?
- É possível alterar o comportamento em tempo de execução?

Princípio de Design

- "Programe para uma interface e não para uma implementação"
  - Utilizaremos interfaces para representar o cada comportamento:





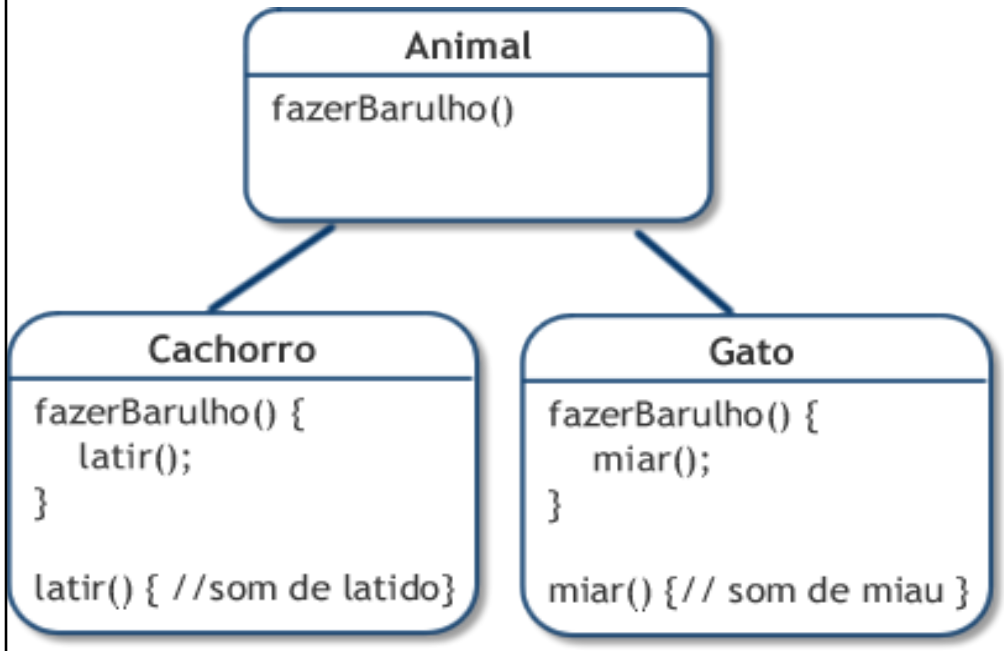
O que é diferente???

- Não é a classe Pato que implementa o comportamento
- Não existe uma implementação concreta destes comportamentos na classe Pato
- Isso nos deixava preso a estas implementações específicas
- As classes Pato irão usar um comportamento externo

Programar para uma *interface*

- Significa
  - Programar para um supertipo
- É possível programar desta maneira sem usar uma interface em Java
- Polimorfismo
  - O **tipo declarado** → **supertipo**
  - **Exemplo** →

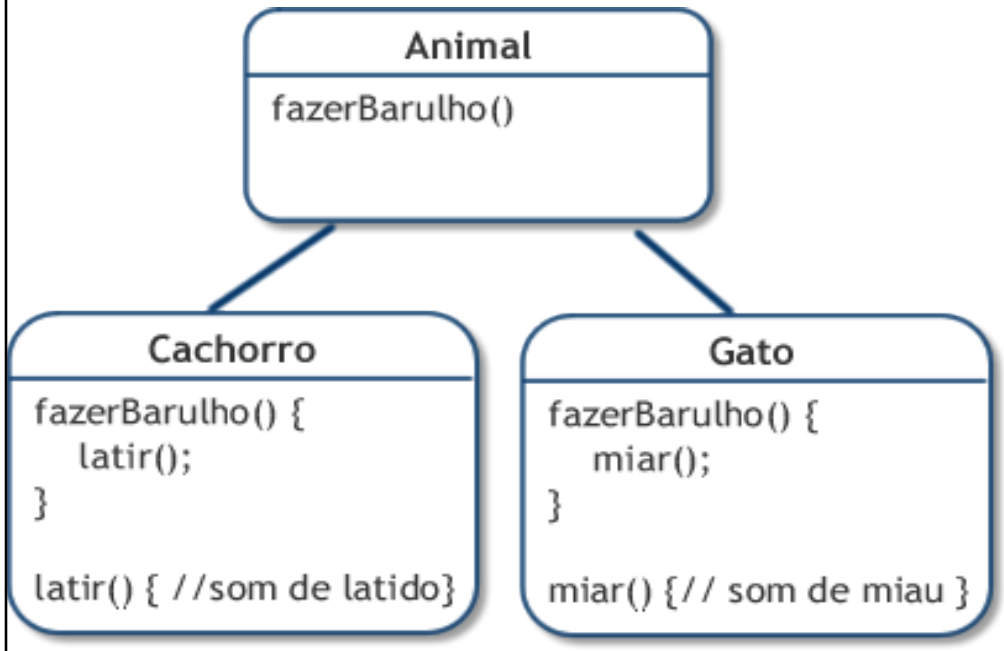
Programando Interface x Implementação



- **Para Implementação:**

```
Cachorro c = new Cachorro();

c.latir();
```

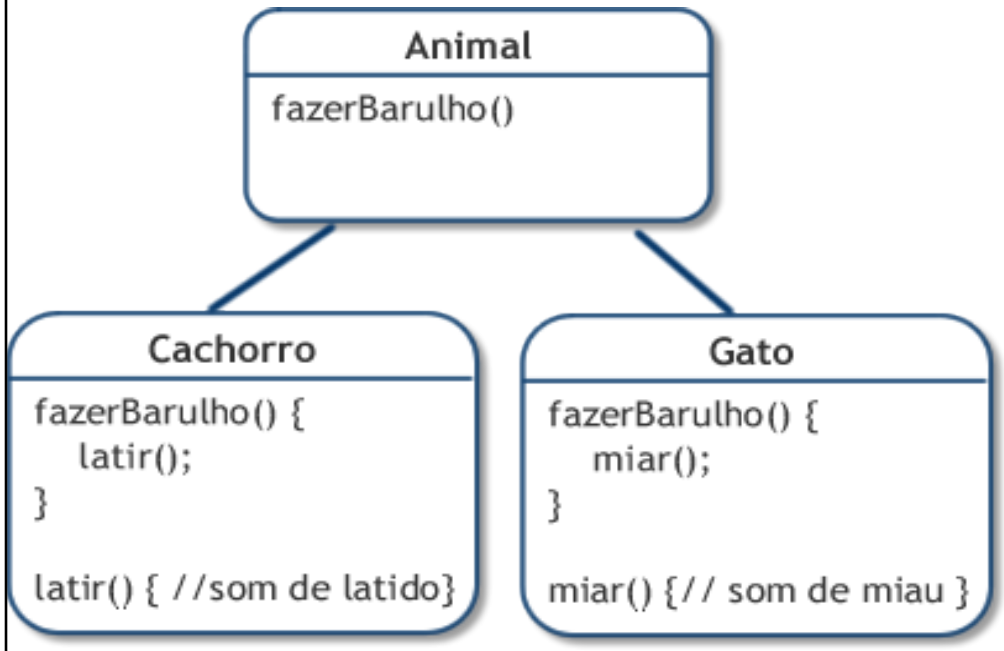
Programando Interface x Implementação



- **Para interface/ supertipo:**

```
Animal c = new Cachorro();

c.fazerBarulho();
```

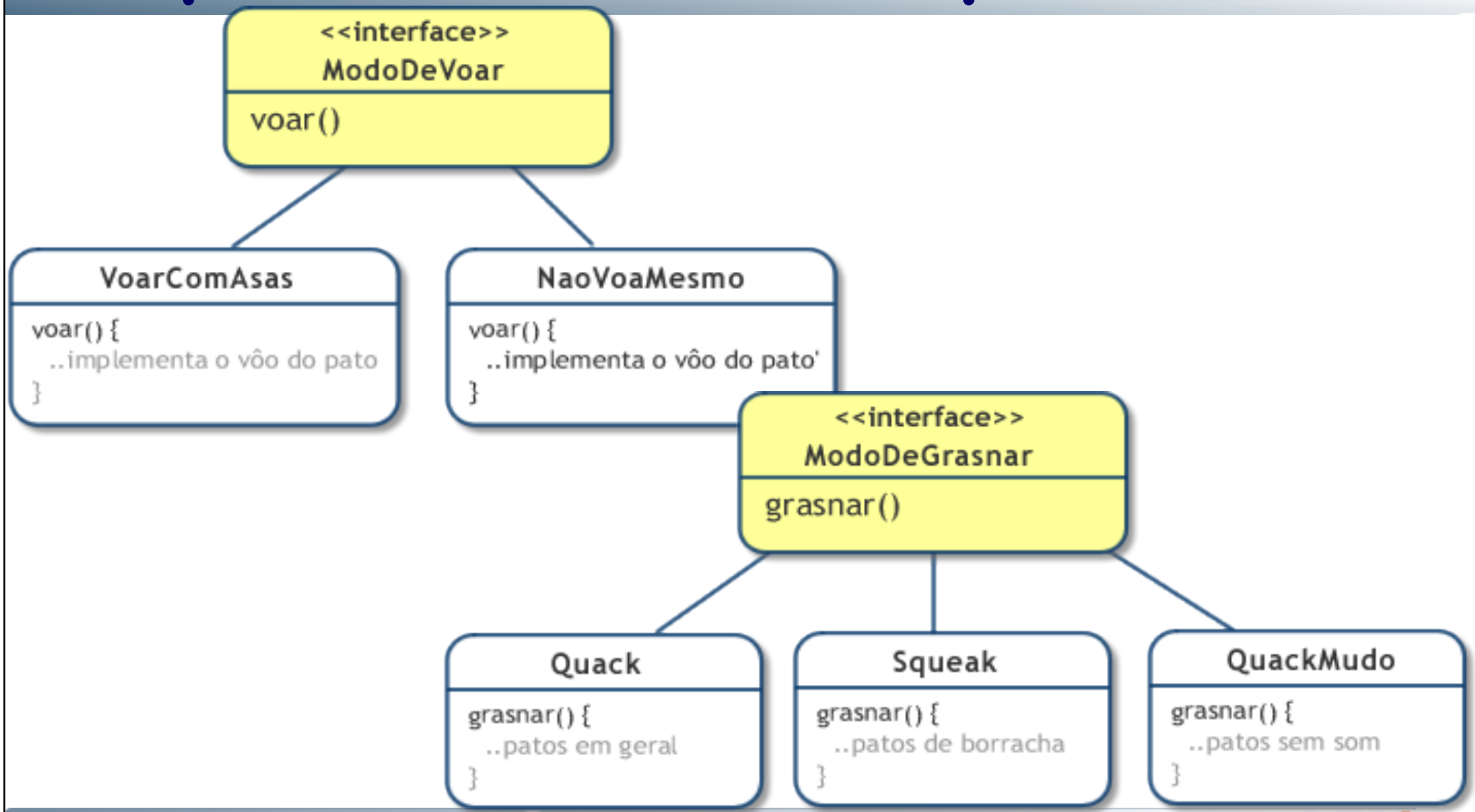
Programando Interface x Implementação



- **Melhorando**

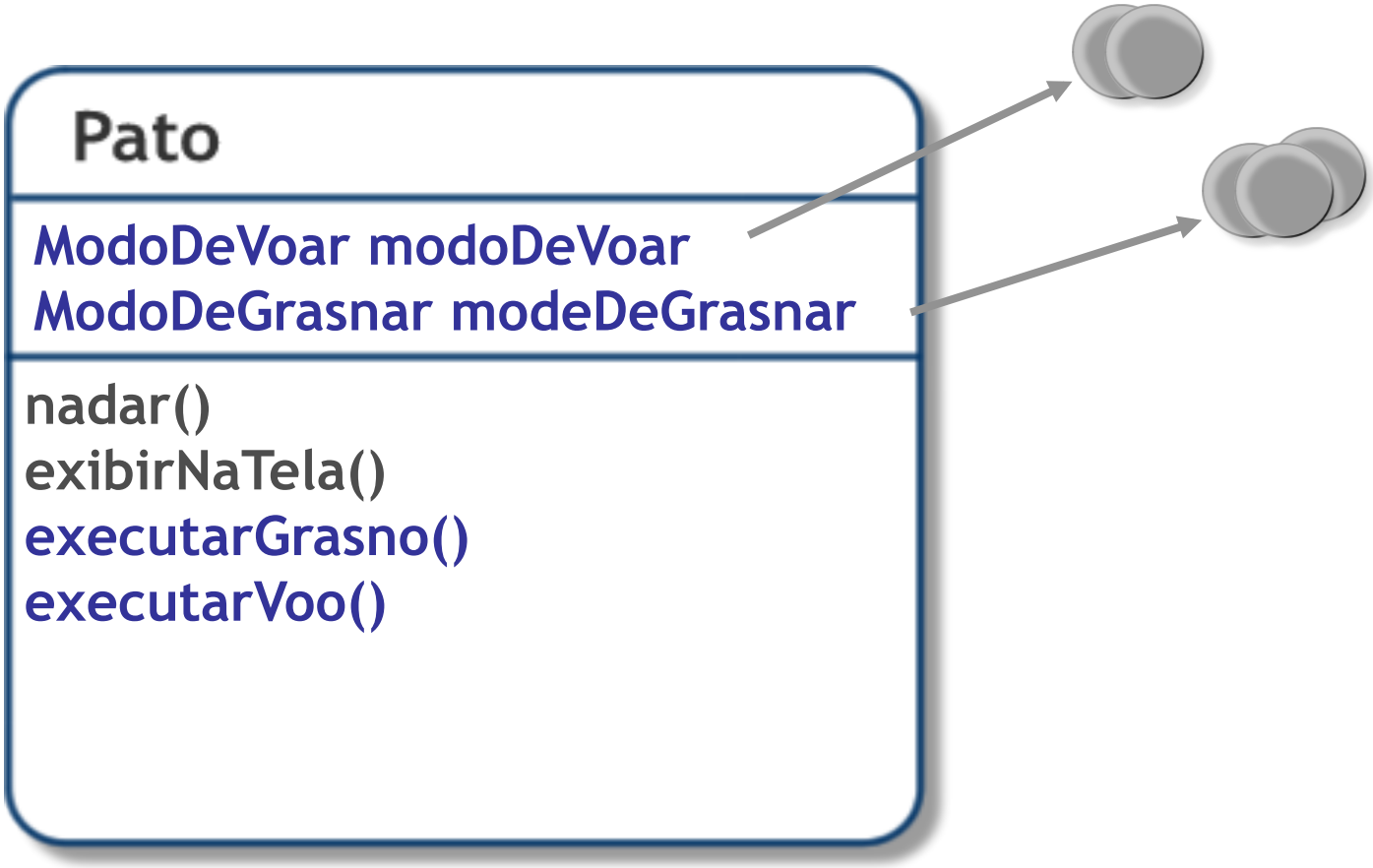
```
Animal c = getAnimal();

c.fazerBarulho();
```

Implementando os comportamentos



Integrando o comportamento

1. Adicionar 2 variáveis de instância ao Pato



Integrando o comportamento

2 Implementamos, por exemplo, o executarGrano()

```
public class Pato {
   ModoDeGrasnar modoDeGrasnar;

   public void executarGrasno() {
      modoDeGrasnar.grasnar();
   }
}
```

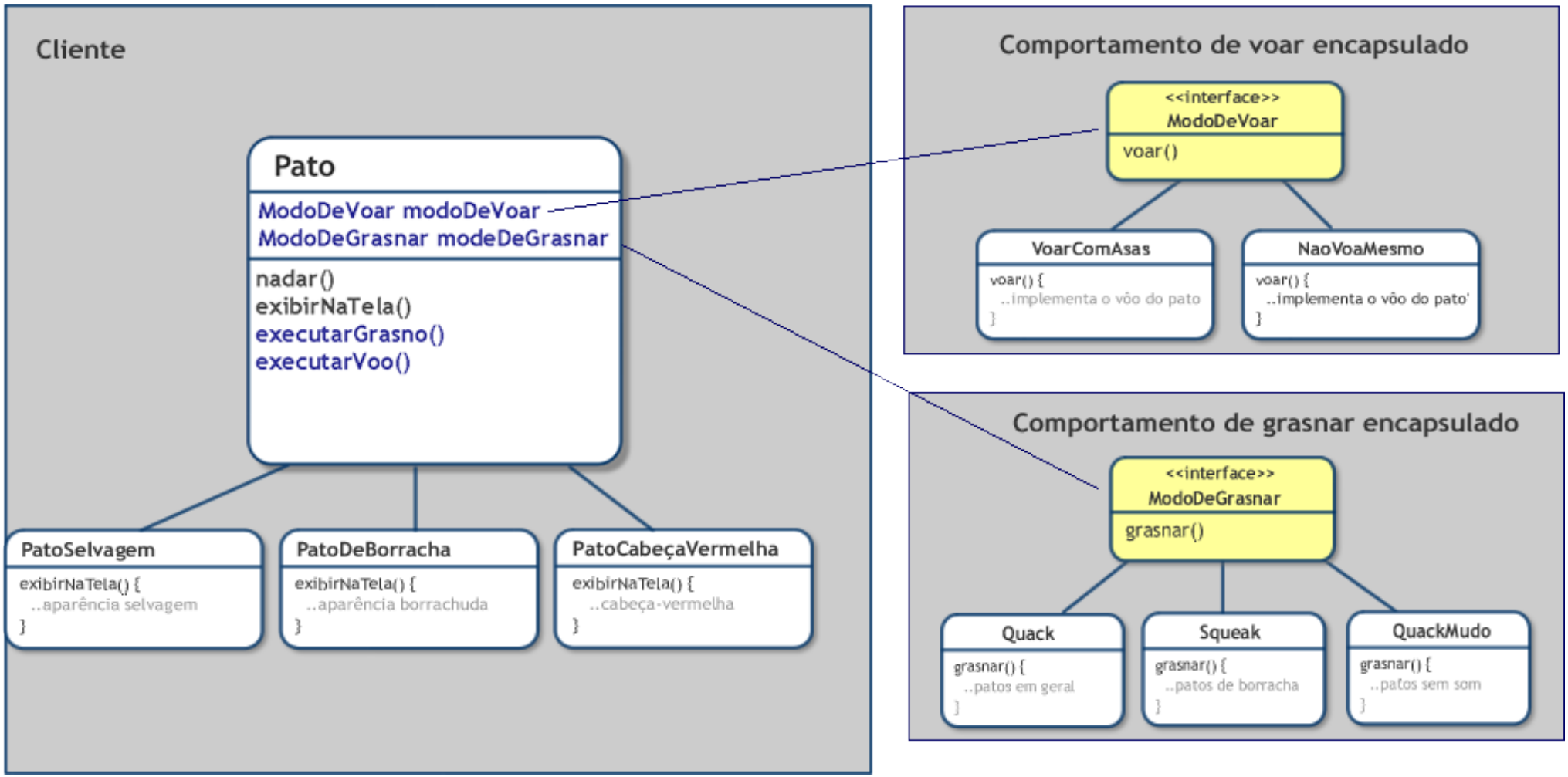
Delegou o comportamento para outra classe

Integrando o comportamento

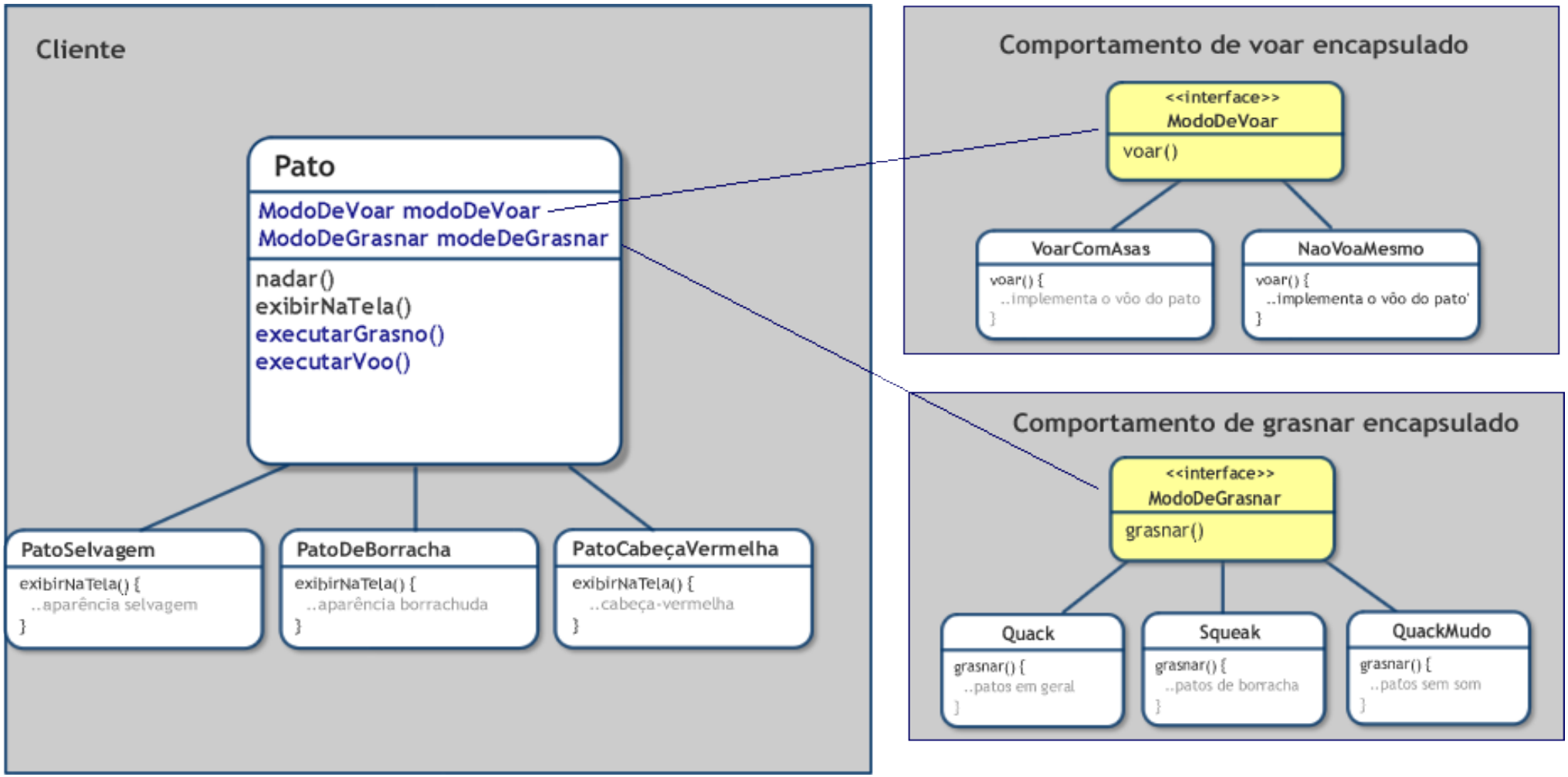
3. Como definir as variáveis de instância do comportamento?

```
public class PatoSelvagem extends Pato {

   public PatoSelvagem() {
     modoDeVoar = new VoarComAsas();
     modoDeGrasnar = new Quack();
   }

   public void exibirNaTela() {
     System.out.println("Eu sou um pato selvagem");
   }
}
```

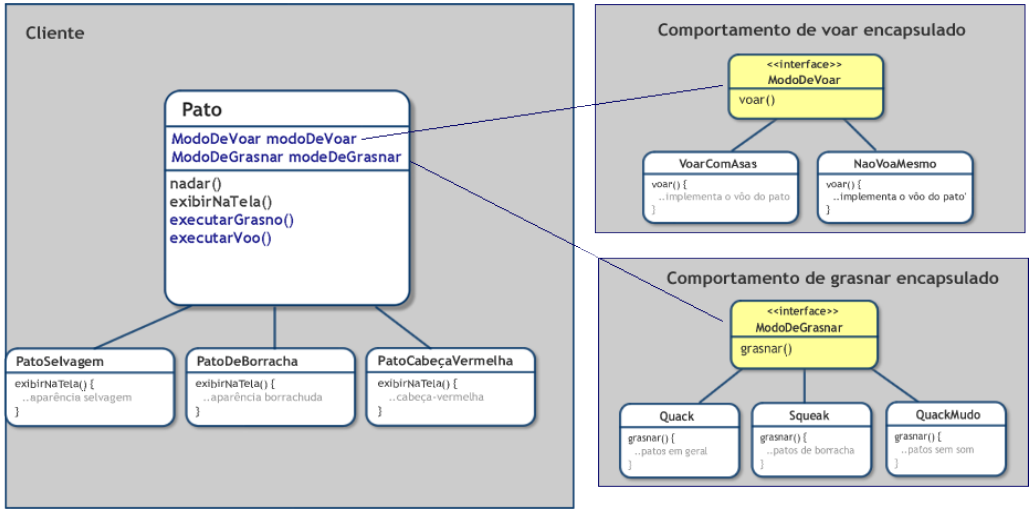
Tudo junto


Cliente
Pato
ModoDeVoar modoDeVoar
ModoDeGrasnar modeDeGrasnar
nadar()
exibirNaTela()
executarGrasno()
executarVoo()
PatoSelvagem
exibirNaTela() {
..aparência selvagem
}
PatoDeBorracha
exibirNaTela() {
..aparência borrachuda
}
PatoCabeçaVermelha
exibirNaTela() {
..cabeça-vermelha
}
Comportamento de voar encapsulado
<<interface>>
ModoDeVoar
voar()
VoarComAsas
voar() {
..implementa o vôo do pato
}
NaoVoaMesmo
voar() {
..implementa o vôo do pato'
}
Comportamento de grasnar encapsulado
<<interface>>
ModoDeGrasnar
grasnar()
Quack
grasnar() {
..patos em geral
}
Squeak
grasnar() {
..patos de borracha
}
QuackMudo
grasnar() {
..patos sem som
}

Verifica-se a composição



Princípio de Design

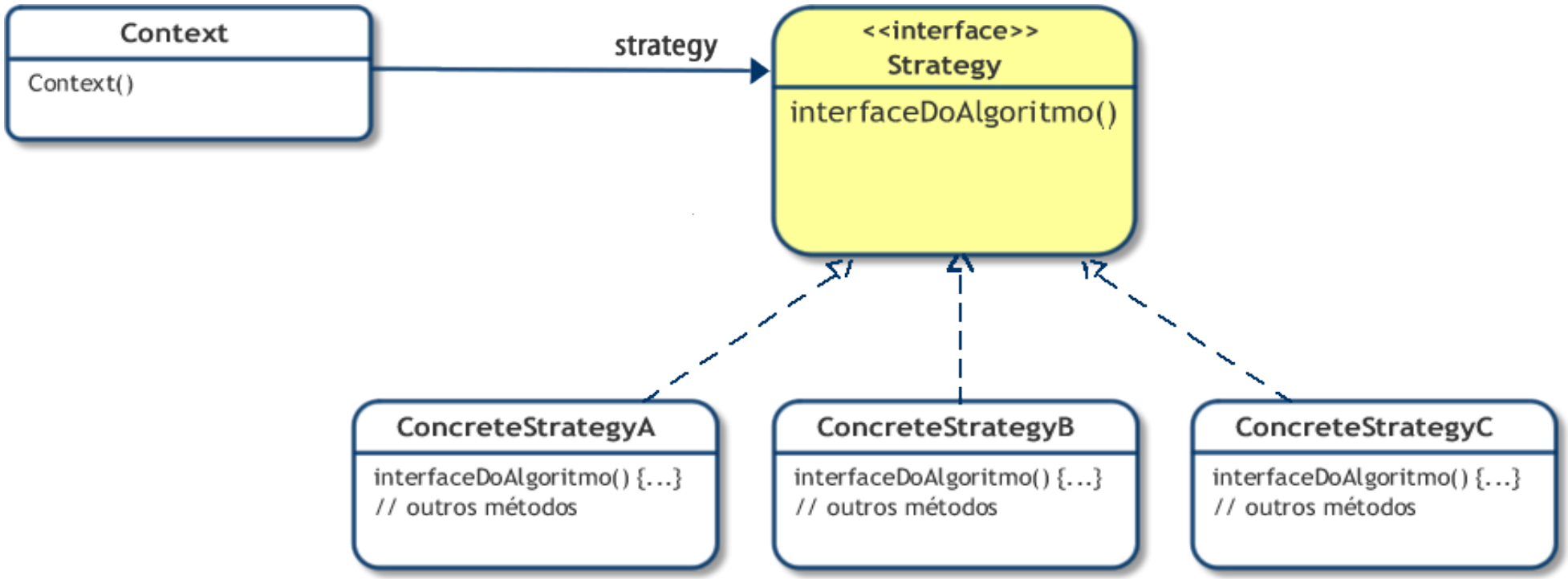


- "Dar prioridade à composição ao invés da herança"
  - Maior flexibilidade
  - Conjunto de algoritmos encapsulados em uma família de classes
  - Alteração do comportamento em tempo de execução

Primeiro Padrão → STRATEGY

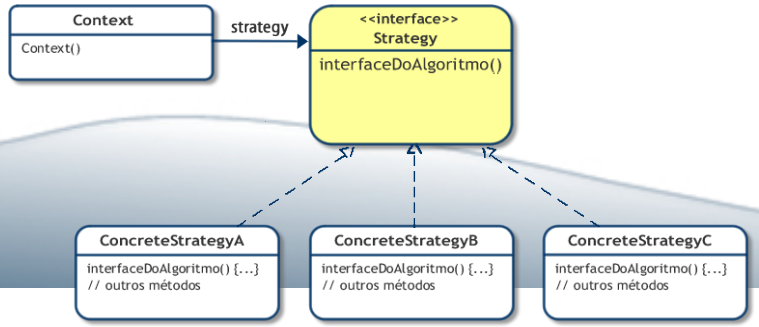
Define uma **família** de **algoritmos**, **encapsula** cada um deles e os torna **intercambiáveis**. O **Strategy** permite que os algoritmos **variem** independentemente dos clientes que o utilizam.

Strategy
Diagrama de Classes



Aplicabilidade

- Use o padrão Strategy quando:
  - Muitas classes relacionadas diferem somente no seu comportamento
    - O *Strategy* fornecem uma maneira de configurar uma classe com um, dentre muitos comportamentos
  - Precisa-se de variantes de um algoritmo
  - Um algoritmo usa dados de que os clientes não deveriam ter conhecimento
    - O *Strategy* evita a exposição de estruturas de dados complexos específicas de um algoritmo
  - Uma classe define muitos comportamentos e estes aparecem em suas operações como múltiplos comandos condicionais da linguagem
    - Mova os condicionais para sua classe *Strategy*



Participantes

- **Context**
  - Possui uma referência para um objeto **Strategy**
  - É configurado com um objeto **ConcreteStrategy**
  - Pode definir uma interface que permite o **Strategy** acessar seus dados
- **Strategy**
  - Define uma interface comum para todos os algoritmos suportados. **Context** usa esta interface para chamar um algoritmo definido por um **ConcreteStrategy**
- **ConcreteStrategy**
  - Implementa o algoritmo usando a interface **Strategy**

Colaborações

- Strategy e Context interagem para implementar o algoritmo escolhido
- Um Context repassa solicitações dos seus clientes para seu Strategy

Consequências

- Famílias de algoritmos relacionados
- Alternativa ao uso de subclasses
- Possibilidade da escolha de implementações

Consequências

- Os clientes devem conhecer diferentes Strategies
  - O cliente deve compreender qual a diferença entre os Strategies
  - Usar o padrão Strategy somente quando a variação em comportamento é importante
- Custo de comunicação entre Strategy e Context
- Aumento do número de objetos