PADRÕES DE PROJETO DE SOFTWARE

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro[a]

6 de maio de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

[a] Baseado nos slides de Kenneth M. Anderson

I think patterns as a whole can help people learn object-oriented thinking: how you can leverage polymorphism, design for composition, delegation, balance responsibilities, and provide pluggable behavior. Patterns go beyond applying objects to some graphical shape example, with a shape class hierarchy and some polymorphic draw method. You really learn about polymorphism when you've understood the patterns. So patterns are good for learning OO and design in general.

Erich Gamma

ORIGENS

- Em 1995, a “Gang of Four” (Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides) publicou um livro chamado de *Design Patterns*
- O livro aplicou o conceito de *padrões* (mais sobre isso adiante) no contexto de desenvolvimento de software
- Os padrões não foram inventados pelos autores; eles foram *identificados* em 3 sistemas de software “reais”

PADRÕES DE PROJETO ESTÃO EM TODA PARTE

- Muitos outros livros de padrões foram lançados desde então
  - muitos outros padrões foram catalogados
  - apesar de muitos autores terem abandonado a ideia de encontrar os padrões em mais de um sistema antes de publicá-lo
- Aprender padrões de projeto de software é fundamental para se tornar um especialista em análise e projetos de software orientado a objetos!

ANTROPOLOGIA CULTURAL

- Padrões de projetos tem sua origem intelectual na área de antropologia cultural
- Dentro de uma cultura, indivíduos concordam sobre o que é considerado um bom *design* (etnocentrismo)
- Padrões (estruturas e relações que aparecem repetidamente em muitos objetos diferentes) nos dão uma base objetiva para julgar o que é um “bom projeto”

What makes us know when an architectural design is good?
Is there an objective basis for such a judgement?

- Padrões de projetos em software foram inspirados nas ideias de um trabalho realizado nos anos 1970 pelo arquiteto Christopher Alexander
- O livro “The Timeless Way of Building”, lançado em 1979, propõe a seguinte reflexão: “A noção de qualidade é uma noção objetiva”?
- Christopher Alexander achava que **sim**, que é possível definir objetivamente conceitos como “boa qualidade” ou “beleza” de construções

- Ele estudou o problema de como identificar o que faz um projeto arquitetônico ser considerado um *projeto de excelente qualidade*, observando vários tipos de construções diferentes:
  - prédios, cidades, ruas, casas, etc.
- Quando ele encontrava um exemplo de projeto de boa qualidade, ele comparava aquele objeto a outros objetos de boa qualidade e procurava por características comuns
  - especialmente se os objetos eram usados para resolver o mesmo tipo de problema

- Ao estudar estruturas de boa qualidade que resolvem os mesmos tipos de problemas, ele pôde descobrir similaridades entre os projetos; estas similaridades eram o que ele chamava padrões

  "Cada padrão descreve um problema que ocorre repetidamente de novo e de novo em nosso ambiente, e então descreve a parte central da solução para aquele problema de uma forma que você pode usar esta solução um milhão de vezes, sem nunca implementa-la duas vezes da mesma forma."

Livros

- The Timeless Way of Building
- A Pattern Language: Towns, Buildings, and Construction

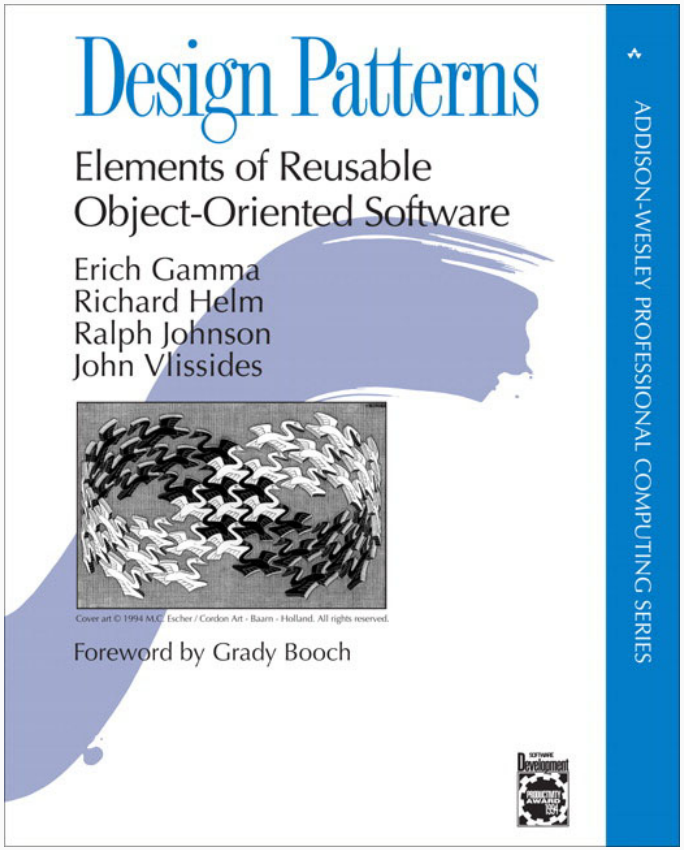
ELEMENTOS DE UM PADRÃO

- Alexander identificou quatro elementos que descrevem um padrão:
  - o nome do padrão
  - o propósito do padrão: qual problema ele resolve?
  - como resolver o problema
  - as restrições que devemos considerar na nossa solução
- Ele acreditava que aplicar múltiplos padrões ao mesmo tempo ajudaria a resolver problemas arquitetônicos complexos

- Os padrões de projetos de software surgiram quando as pessoas começaram a se perguntar:
  - Existem problemas que aparecem durante o desenvolvimento de software toda hora e que podem ser resolvidos de um modo mais ou menos parecido?
  - É possível projetar software em termos de padrões?
- Muitos achavam que a resposta para essas perguntas eram "sim" e isso motivou a criação do livro *Design Patterns* pela *Gang of Four*
- O livro *Design Patterns* cataloga 23 padrões: soluções bem sucedidas para problemas comuns que aparecem durante o projeto de um software

GANG OF FOUR (GOF)

E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.



- Padrões de projetos garantem que a qualidade de um software possa ser medida objetivamente
  - O que está presente em um projeto de boa qualidade (X) que não está presente em um projeto de má qualidade?
  - O que está presente em um projeto de má qualidade (Y) que não está presente em um projeto de boa qualidade?
- Queremos maximizar a presença de X e minimizar a de Y em nossos projetos

FORMATO DOS PADRÕES NO GOF I

Nome (e número da página) um bom nome é essencial para que o padrão caia na boca do povo

Objetivo / Intenção

Motivação um cenário mostrando o problema e a necessidade da solução

Aplicabilidade como reconhecer as situações nas quais o padrão é aplicável

Estrutura uma representação gráfica da estrutura de classes do padrão

Participantes as classes e objetos que participam e quais suas responsabilidades

Colaborações como os participantes colaboram para exercer suas responsabilidades

Formato dos padrões no GoF II

Consequências vantagens e desvantagens, *trade-offs*

Implementação detalhes de implementação do padrão e aspectos específicos de cada linguagem

Exemplo de código em C++ (maioria) ou Smalltalk

Usos conhecidos exemplos de sistemas reais de domínios diferentes onde o padrão é utilizado

Padrões relacionados quais padrões devem ser usados em conjunto com esse e quais outros padrões são similares

POR QUE ESTUDAR PADRÕES DE PROJETO?

Padrões nos permitem:

- reutilizar soluções que funcionaram no passado; por que reinventar a roda toda vez?
- ter um vocabulário compartilhado para o projeto de software
  - permite que você possa dizer a um colega: “Eu usei o padrão Strategy aqui para permitir que o algoritmo usado para calcular essa expressão possa ser personalizado”
  - você não precisa perder tempo explicando o que você quis fazer já que vocês dois conhecem o padrão Strategy

POR QUE ESTUDAR PADRÕES DE PROJETO?

- Padrões de projetos **não oferecem reutilização de código** mas sim **reutilização de experiência**
- Conhecer conceitos como abstração, herança, polimorfismo não vão fazer de você um bom projetista, a não ser que você use esses conceitos para criar projetos flexíveis, fáceis de manter e que possa lidar com mudanças
- Padrões de projeto podem mostrar como aplicar esses conceitos para atingir esses objetivos

ANALOGIA DO CARPINTEIRO

- Padrões de projeto nos dão uma perspectiva de alto nível dos problemas resolvidos pela análise e desenvolvimento de programas OO
- Você poderá pensar mais abstratamente nos problemas, sem se preocupar com os detalhes de implementação
- O livro dá um exemplo excelente do que ele quer dizer com "perspectiva de alto nível". Imagine dois carpinteiros conversando:
  - Devo usar uma junta de mitra ou uma junta de meia esquadria?
  - ou: Devo fazer uma junta fazendo um corte na reto na madeira seguido de um outro em 45° e então ... ?

ANALOGIA DO CARPINTEIRO

- O primeiro permite uma conversa mais rica sobre o problema
- O seguindo depende do conhecimento compartilhado entre os carpinteiros
  - eles sabem que juntas de esquadria tem mais qualidade do que as juntas de mitra, mas são mais caras
  - sabendo disso eles podem debater se é mesmo necessário garantir mais qualidade nessa situação

ANALOGIA DO CARPINTEIRO

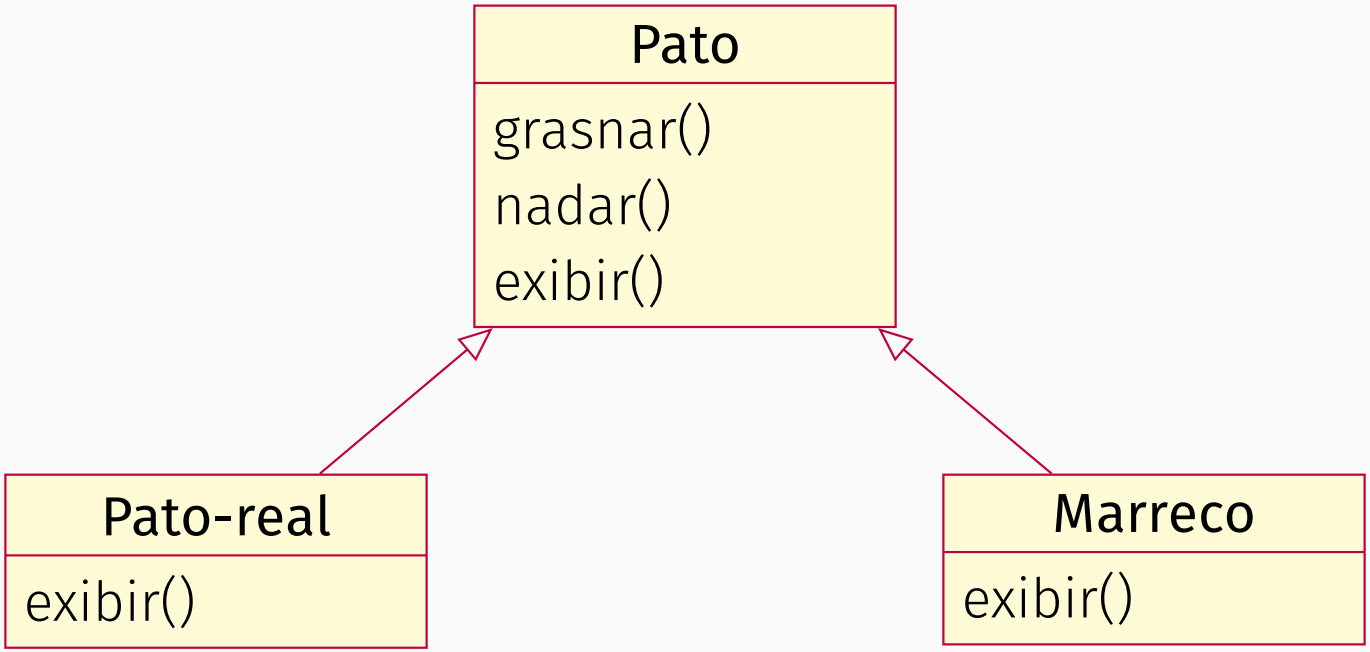
- “Eu tenho um objeto que possui uma informação importante e há outros objetos que precisam saber quando uma informação muda. Esses outros objetos são criados e destruídos dinamicamente. Eu acho que eu deveria separar notificação e o registro de quem quer notificado da funcionalidade do objeto e permitir que a implementação do objeto se concentre em armazenar e manipular a informação corretamente. Você concorda?

vs.

- “Eu estou pensando em usar o padrão Observer. Você concorda?”

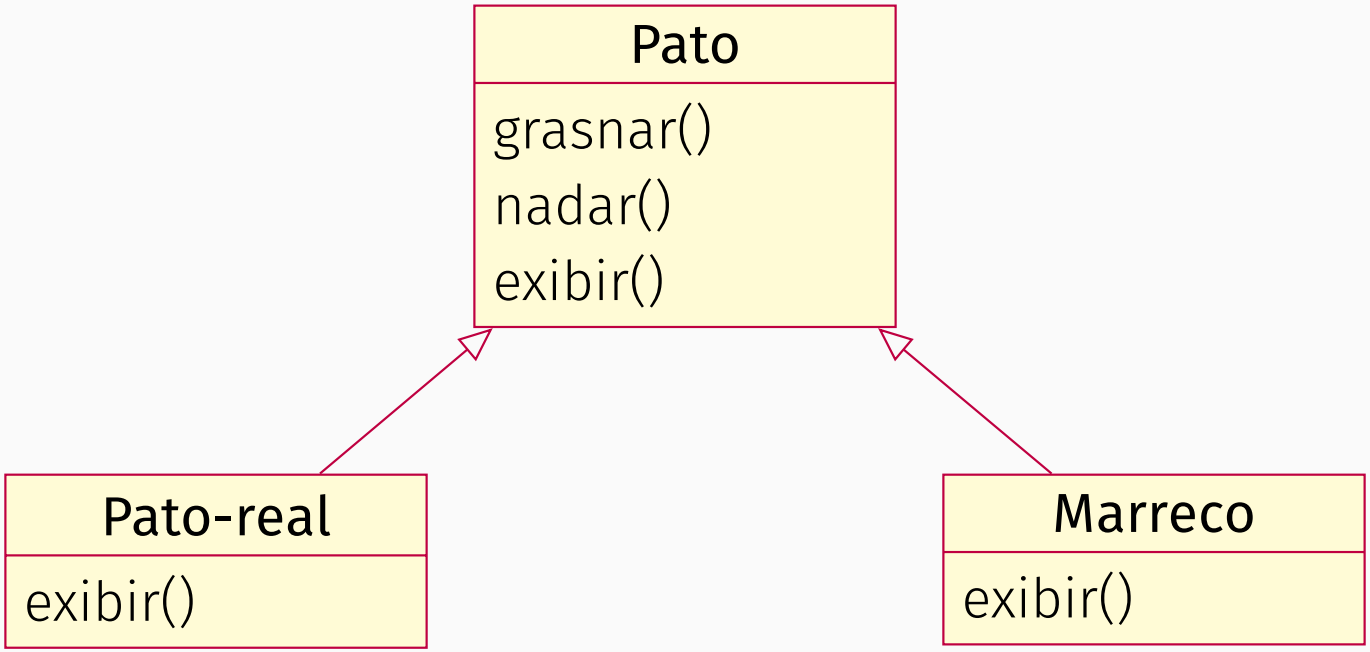
Exemplo:

Imagine um “simulador de patos em uma lagoa”, que é capaz de exibir uma grande variedade de espécies de patos nadando e grasnando.



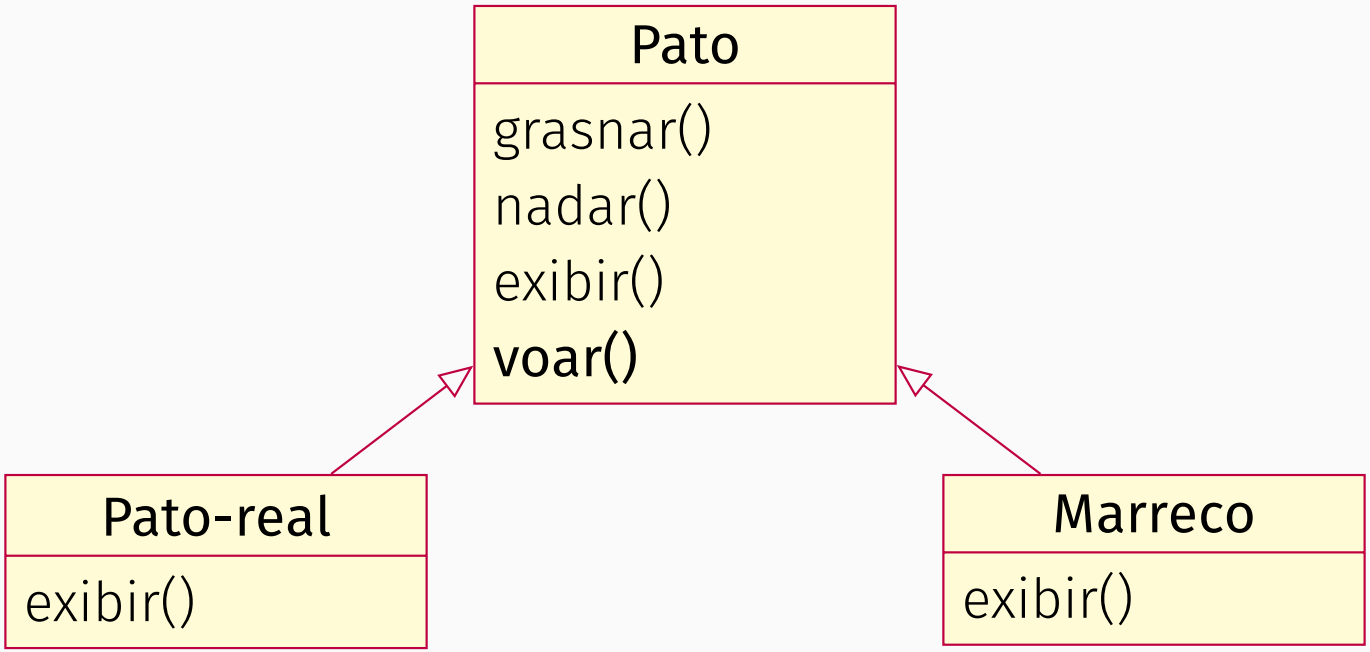
Exemplo:

Imagine um “simulador de patos em uma lagoa”, que é capaz de exibir uma grande variedade de espécies de patos nadando e grasnando.



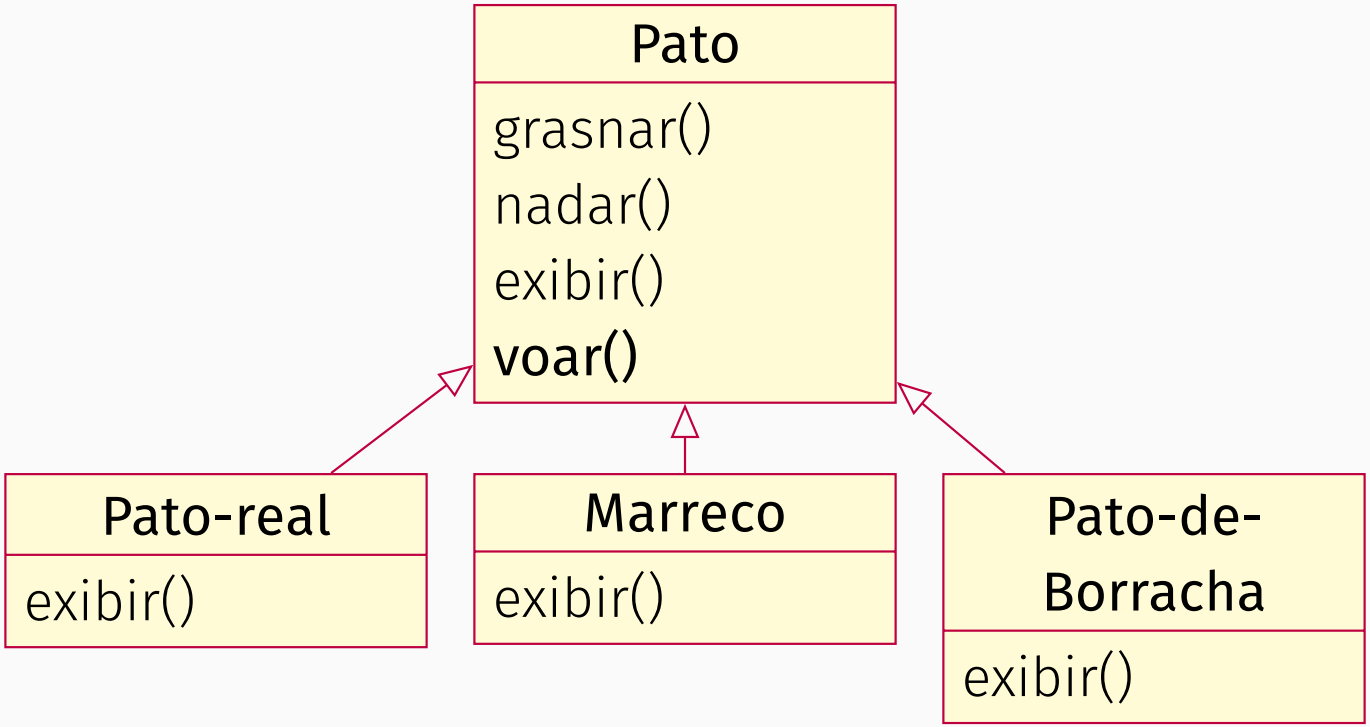
Novo requisito

Agora os patos devem conseguir voar!



Ops

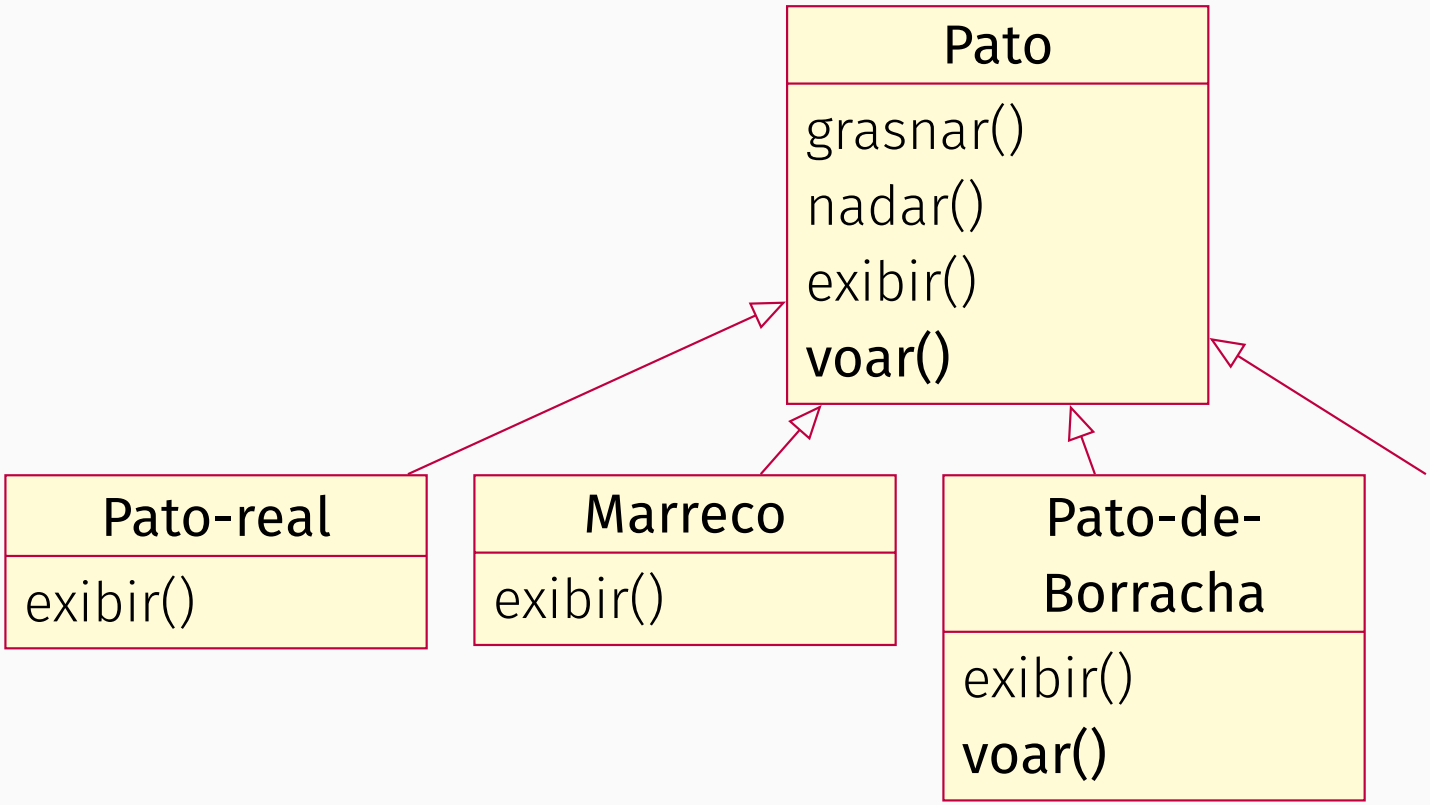
Patos de borracha não deveriam poder voar!



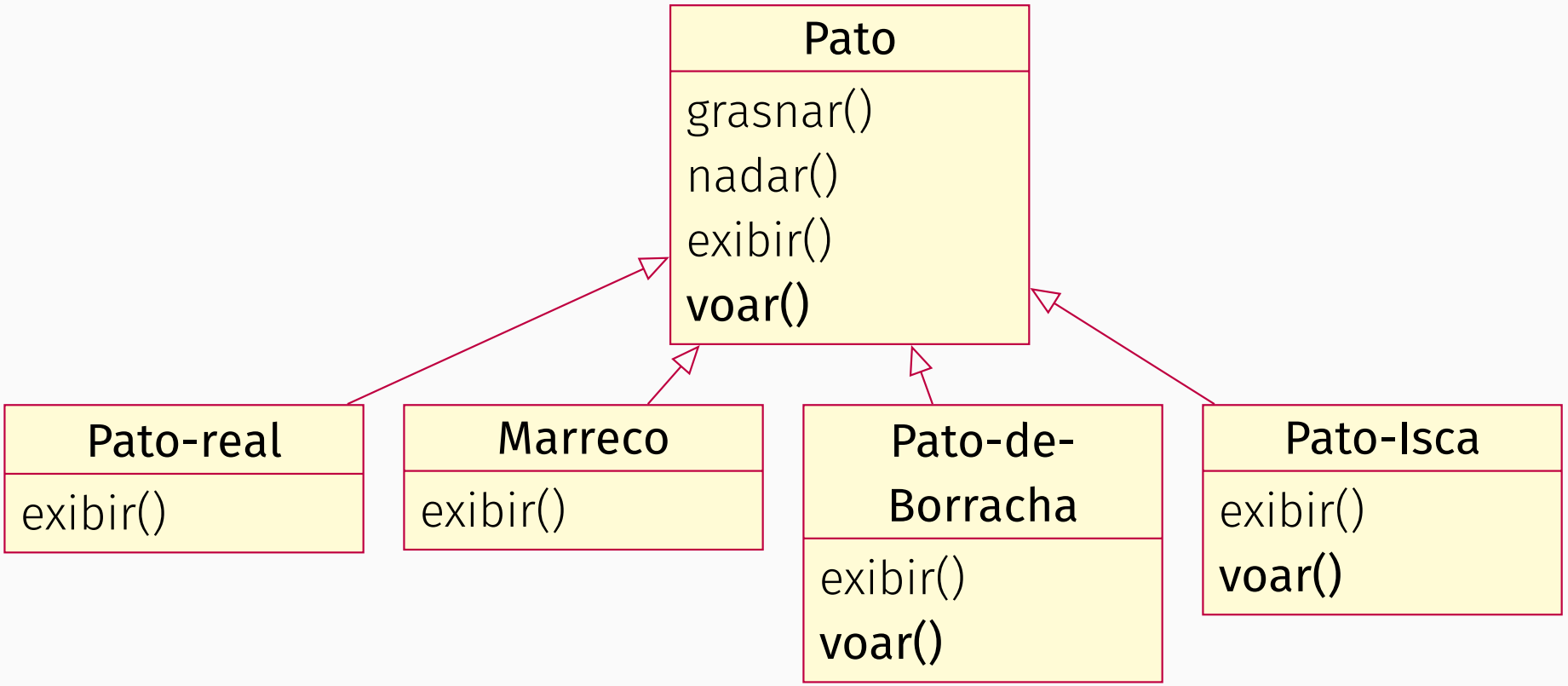
Ops

Patos de borracha não deveriam poder voar!

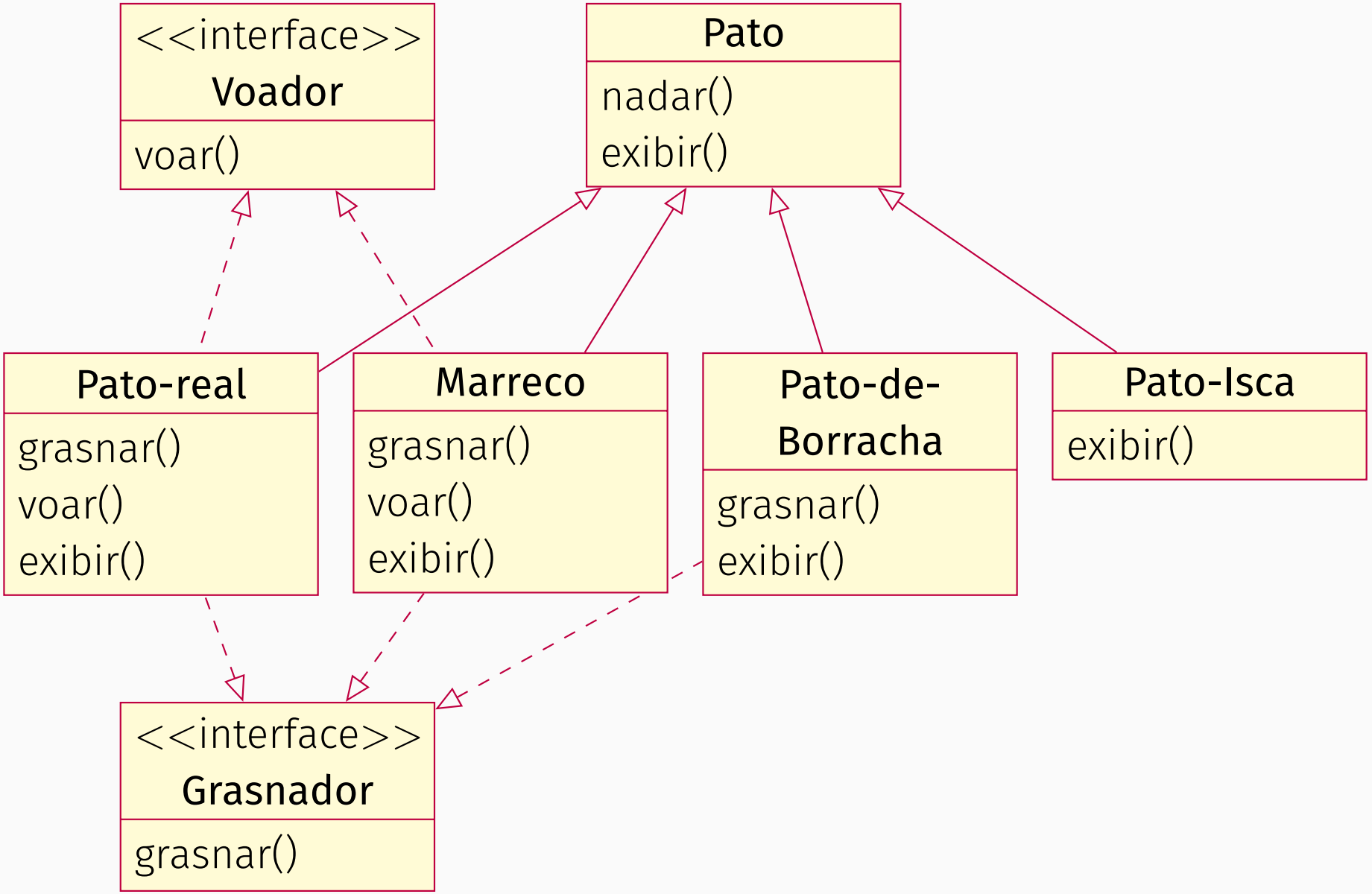
NÃO ESTÁ MAIS TÃO FÁCIL



NÃO ESTÁ MAIS TÃO FÁCIL



E SE USARMOS INTERFACES?



PRÓS E CONTRAS DO PROJETO

Com herança:

PRÓS E CONTRAS DO PROJETO

Com herança:

- (pró) reutilização de código, apenas um método `voar()` e `grasnar()`

PRÓS E CONTRAS DO PROJETO

Com herança:

- (pró) reutilização de código, apenas um método `voar()` e `grasnar()`
- (contra) o comportamento padrão do pai não era tão padrão assim no final

Com interfaces:

PRÓS E CONTRAS DO PROJETO

Com herança:

- (pró) reutilização de código, apenas um método `voar()` e `grasnar()`
- (contra) o comportamento padrão do pai não era tão padrão assim no final

Com interfaces:

- (pró) especificidade: apenas as subclasses que precisavam do método `voar()` eram obrigadas a implementá-lo

PRÓS E CONTRAS DO PROJETO

Com herança:

- (pró) reutilização de código, apenas um método `voar()` e `grasnar()`
- (contra) o comportamento padrão do pai não era tão padrão assim no final

Com interfaces:

- (pró) especificidade: apenas as subclasses que precisavam do método `voar()` eram obrigadas a implementá-lo
- (contra) não há reutilização de código; interfaces só definem assinaturas

PRÓS E CONTRAS DO PROJETO

Com herança:

- (pró) reutilização de código, apenas um método `voar()` e `grasnar()`
- (contra) o comportamento padrão do pai não era tão padrão assim no final

Com interfaces:

- (pró) especificidade: apenas as subclasses que precisavam do método `voar()` eram obrigadas a implementá-lo
- (contra) não há reutilização de código; interfaces só definem assinaturas

Outra possibilidade:

Usar uma classe abstrata ao invés de interface. Você poderia implementar `Voador` e `Grasnador` como classes abstratas e fazer as subclasses de `Pato` usá-las. Requer herança múltipla

Encapsule o que variar

- para esse problema em particular, “o que varia” são os comportamentos entre as subclasses de `Pato`
- precisamos conseguir retirar esses comportamentos que variam entre as subclasses e colocá-lo em sua própria classe (ou seja, encapsulá-lo!)

Resultado:

Menos efeitos colaterais indesejados causados por modificações no código (ex: adicionar o método `voar()`) e código mais flexível

IDEIA

- Mover todo comportamento que varia entre as subclasses de `Pato` e removê-los da classe `Pato`
  - `Pato` não terá mais os métodos `voar()` e `grasnar()`
- Programe para uma interface
  - podemos seguir esse princípio e obrigar que todo membro siga implemente uma interface determinada:
    - em `ComportamentoDePato` teremos `Grasnar`, `Chiar`, `Silêncio`
    - em `ComportamentoDeVoo` teremos `VoarComAsas`, `VoarQuandoArremessado`, `NãoPodeVoar`
- Benefícios extras:
  - outras classes podem ganhar acesso a esses comportamentos (se isso fizer sentido) e podemos adicionar novos comportamentos sem impactar as outras classes

PROGRAMAR PARA UMA INTERFACE ≠ PROGRAMAR UMA INTERFACE JAVA

- Abusamos da palavra “interface” quando falamos em programar para uma interface:
  - podemos seguir uma interface definindo uma interface Java e fazendo as classes implementar essa interface
  - ou podemos “seguir um supertipo” e definir uma classe abstrata que será acessada por outras classes via herança
- Podemos dizer que “programar para uma interface” implica que o objeto que usar a interface terá uma variável cujo tipo é o supertipo da interface ou classe abstrata e, portanto:
  - pode usar qualquer implementação daquele supertipo
  - e está protegida dos nomes específicos das classes
    - um `Pato` usará o comportamento de voar usando uma variável do tipo `ComportamentoDeVoo` ao invés de usar `VoarComAsas`
    - o código ficará menos acoplado

JUNTANDO TUDO: COMPOSIÇÃO

- Para se aproveitar desses novos comportamentos, precisamos modificar a classe `Pato` para delegar seus comportamentos para as outras classes (ao invés de implementá-los internamente)
- Vamos adicionar dois atributos para armazenar os comportamentos desejados e renomear `voar()` e `grasnar()` para `realizarVoo()` e `realizarGrasnada()`
  - esse último passo é só para salientar que não faz sentido um `Pato-Isca` ter métodos como `voar()` ou `grasnar()` como parte de sua interface
  - ao invés disso, ele irá herdar esses novos métodos e associar os comportamentos de `NãoPodeVoar` e `Silêncio` para garantir que ele fará a coisa certa se esses métodos forem chamados
- Esse é um exemplo do princípio favoreça composição em relação à herança

NOVO DIAGRAMA DE CLASSE