UML — UNIFIED MODELING LANGUAGE E PADRÕES DE PROJETOS

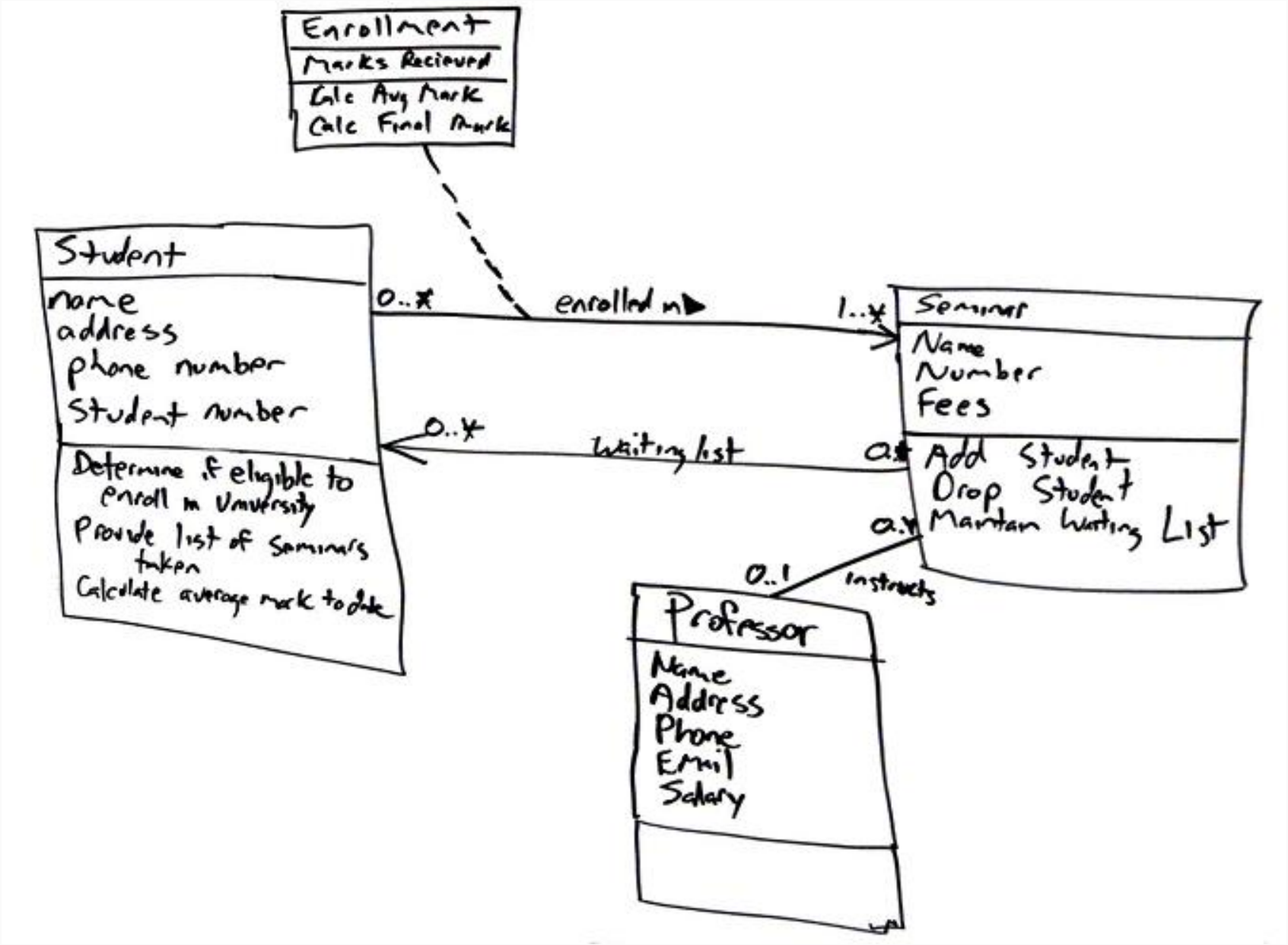
ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro[a]

Escola de Artes, Ciências e Humanidades | EACH | USP

[a] Baseado nos slides de Laëtitia Matignon (Université Claude Bernard Lyon 1) e Kenneth M. Anderson (University of Colorado Boulder)

EXEMPLO DE UML



O QUE É UML?

- Linguagem Unificada de Modelagem (*Unified Modeling Language*)
- Padrão definido pelo *Object Management Group* (OMG)

Objetivos segundo o OMG

O objetivo de UML é fornecer a arquitetos, desenvolvedores e engenheiros de software ferramentas (visuais) para a análise, projeto e implementação de sistemas de software, bem como para a modelagem de negócios e de processos.

O QUE É UML?

- Linguagem Unificada de Modelagem (*Unified Modeling Language*)
- Padrão definido pelo *Object Management Group* (OMG)

Diferença entre Linguagem e Método

- Linguagem de modelagem: notações, gramática, semântica, etc.
- Método: como utilizar uma linguagem de modelagem (análise de requisitos, concepção, desenvolvimento do software, validação, etc.)

UML é uma linguagem universal **para a modelagem de objetos**.

POR QUE MODELAR?

- A descrição de um programa orientado a objetos requer um trabalho conceitual: definição das classes, de como elas se relacionam, de seus atributos e operações (implementadas pelos seus métodos), das interfaces, etc.
- Precisamos de algo para ajudar a organizar as ideias, documentá-las e organizar o desenvolvimento
- A modelagem é uma etapa que precede a implementação

O QUE É UML?

- UML é uma linguagem universal de modelagem de objetos

O QUE É UML?

- UML é uma linguagem universal de modelagem de objetos
- UML é uma notação, uma ferramenta de comunicação visual (diagramas)

O QUE É UML?

- UML é uma linguagem universal de modelagem de objetos
- UML é uma notação, uma ferramenta de comunicação visual (diagramas)
- UML é uma linguagem de modelagem de aplicações orientadas a objetos

O QUE É UML?

- UML é uma linguagem universal de modelagem de objetos
- UML é uma notação, uma ferramenta de comunicação visual (diagramas)
- UML é uma linguagem de modelagem de aplicações orientadas a objetos
- UML não é uma linguagem de programação

O QUE É UML?

- UML é uma linguagem universal de modelagem de objetos
- UML é uma notação, uma ferramenta de comunicação visual (diagramas)
- UML é uma linguagem de modelagem de aplicações orientadas a objetos
- UML não é uma linguagem de programação
- UML não é um processo de desenvolvimento

O QUE É UML?

- UML é uma linguagem universal de modelagem de objetos
- UML é uma notação, uma ferramenta de comunicação visual (diagramas)
- UML é uma linguagem de modelagem de aplicações orientadas a objetos
- UML não é uma linguagem de programação
- UML não é um processo de desenvolvimento
- UML é independente de linguagem de programação

O QUE É UML?

- UML é uma linguagem universal de modelagem de objetos
- UML é uma notação, uma ferramenta de comunicação visual (diagramas)
- UML é uma linguagem de modelagem de aplicações orientadas a objetos
- UML não é uma linguagem de programação
- UML não é um processo de desenvolvimento
- UML é independente de linguagem de programação
- UML é um padrão mantido pela OMG

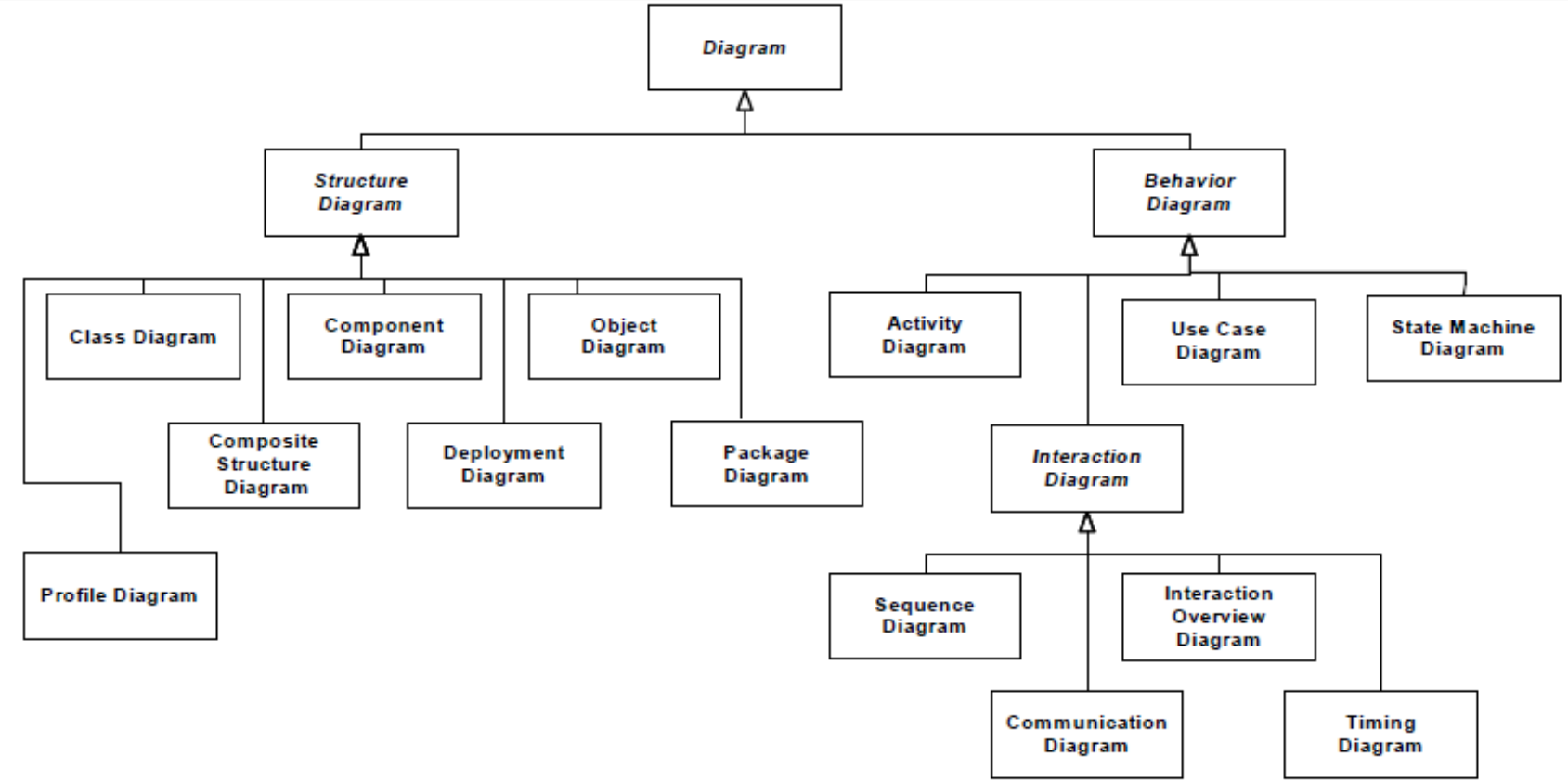
O QUE É UML?

- UML é uma linguagem universal de modelagem de objetos
- UML é uma notação, uma ferramenta de comunicação visual (diagramas)
- UML é uma linguagem de modelagem de aplicações orientadas a objetos
- UML não é uma linguagem de programação
- UML não é um processo de desenvolvimento
- UML é independente de linguagem de programação
- UML é um padrão mantido pela OMG
- Descrição exata: `http://www.omg.org/uml`

Diagramas elementos gráficos que representam o problema de acordo com a definição de seus pontos de vista

Pontos de vista descrevem um ponto de vista do sistema

Modelos dos elementos elementos básicos dos diagramas

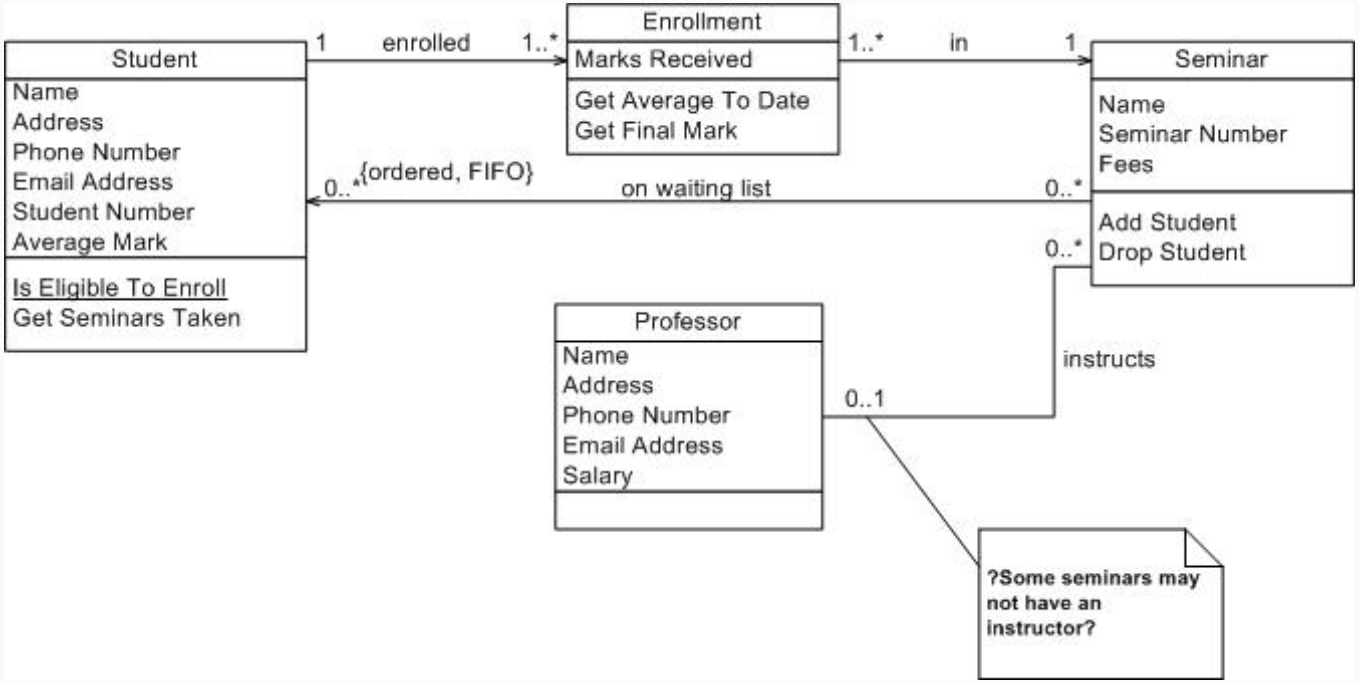


DIAGRAMAS ESTRUTURAIS

Usados para visualizar, especificar, construir e documentar aspectos estáticos de um sistema.

- diagrama de classes
- diagrama de pacotes
- diagrama de objetos
- diagrama de componentes
- diagrama de implantação

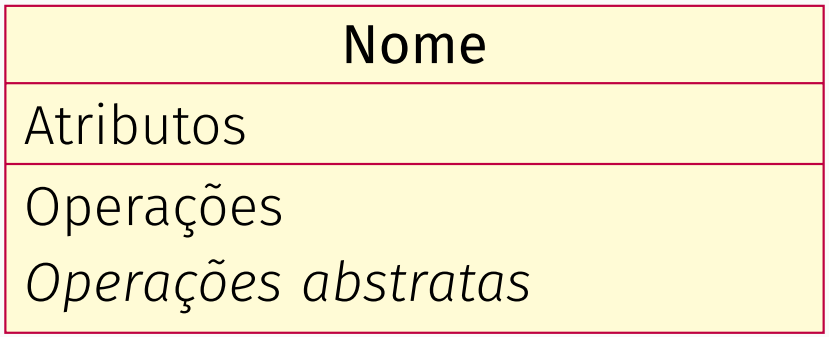
DIAGRAMA DE CLASSES



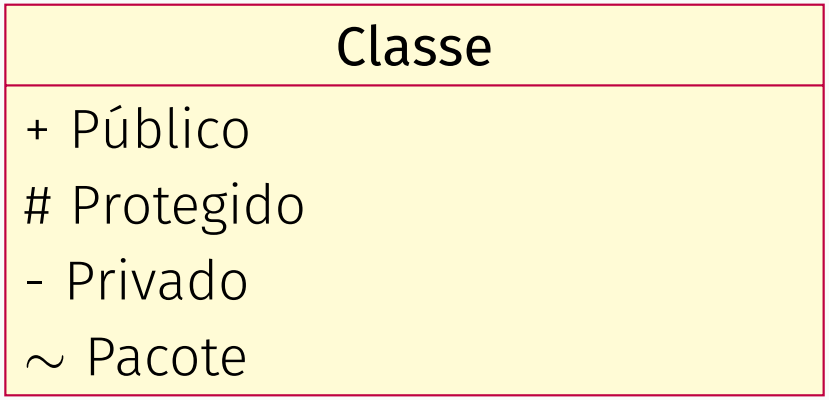
Usos comuns

- Modelar o vocabulário do sistema, em termos de quais abstrações fazem parte do sistema e quais caem fora de seus domínios
- Modelar as colaborações/interações (sociedades de elementos que trabalham em conjunto oferecendo algum comportamento cooperativo)
- Modelagem lógica dos dados manipulados pelo sistema (servindo de base para a definição formal do modelo da base de dados)

NOTAÇÃO PARA CLASSES



ESPECIFICAÇÃO DE ACESSO



ESPECIFICAÇÃO DE ACESSO

Classe
+ Público # Protegido - Privado ~ Pacote

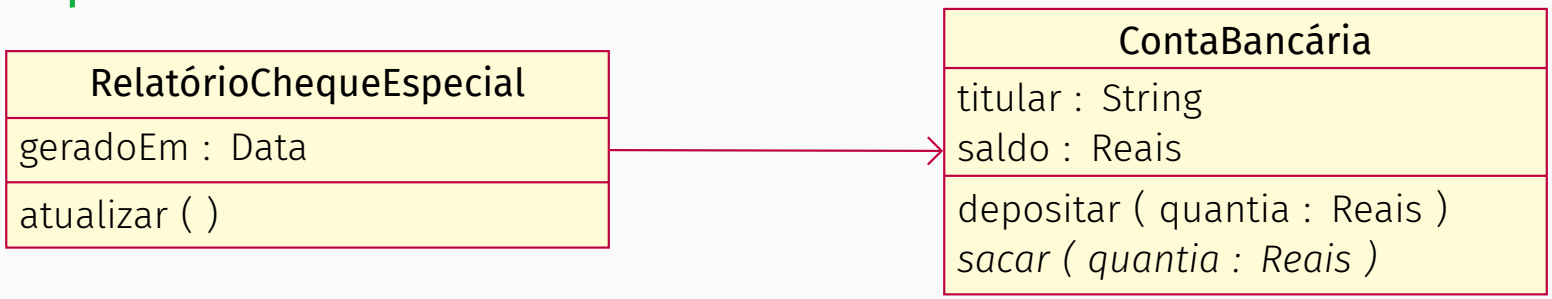
ContaBancária
+ titular : String + saldo : Reais
+ depositar (quantia : Reais) + *sacar (quantia : Reais)* # atualizarSaldo (novoSaldo : Reais)

São conexões entre classes:

1. dependência
2. generalização
3. associação

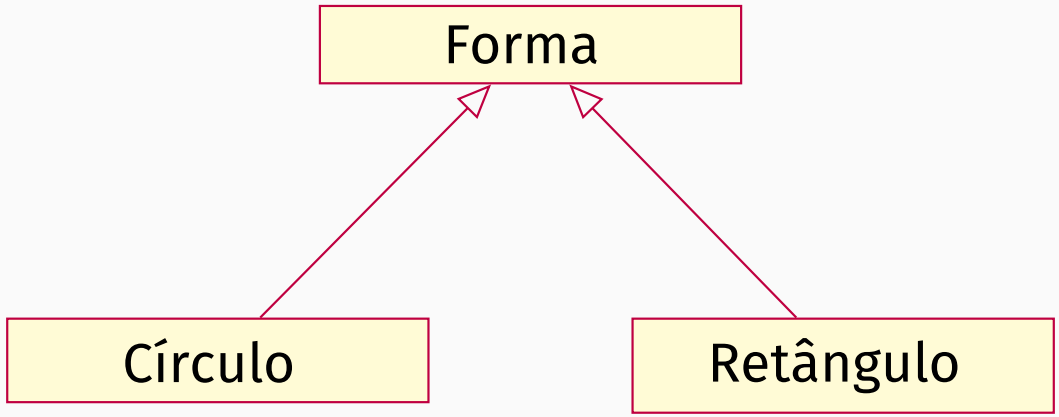
É uma relação do tipo “usa” na qual mudanças na implementação de uma classe podem causar efeitos em outra classe que a usa.

Exemplo: uma classe usa a outra



GENERALIZAÇÃO

É uma relação do tipo “é um” entre uma coisa geral (superclasse) e uma coisa mais específica (subclasse)



ASSOCIAÇÃO

É uma relação estrutural na qual classes ou objetos estão interconectados.

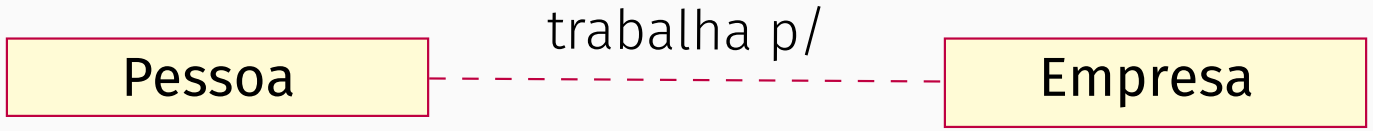
Uma associação entre objetos é chamada de ligação (*link*)

ORNAMENTOS PARA ASSOCIAÇÕES

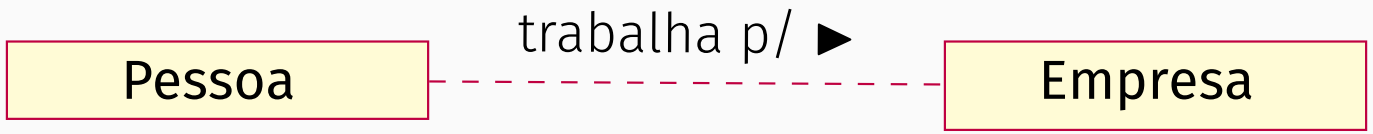
- nome
- papel
- multiplicidade
- agregação
- composição

NOME DA ASSOCIAÇÃO

descreve a natureza da relação:



pode indicar a direção:



PAPÉIS

Classes e objetos podem assumir papéis diferentes em diferentes momentos.

Valores possíveis: valor exato, intervalo ou * para “muitos”

Exemplo:

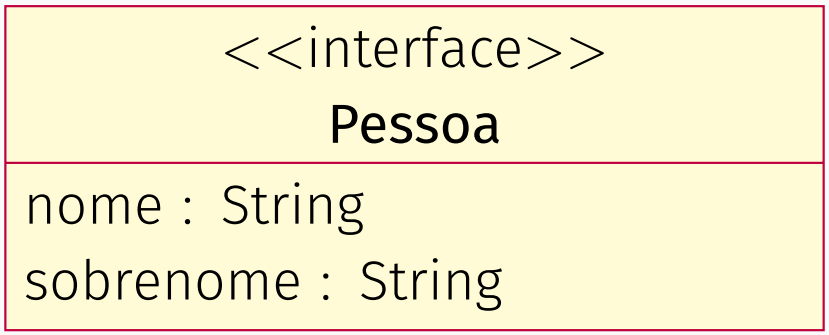
AGREGAÇÃO

É uma relação do tipo “todo/parte” ou “possui um” na qual uma classe representa uma coisa grande que é composta por coisas menores. Indicada por um diamante vazio.

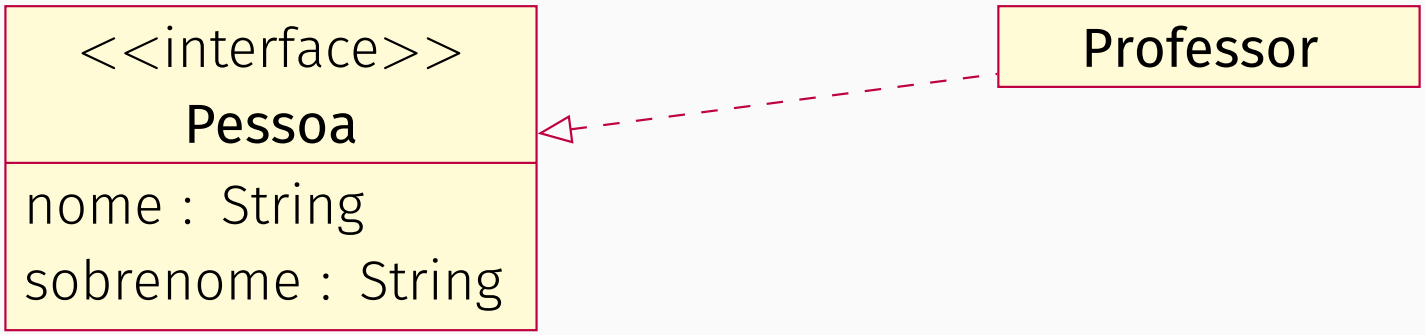
COMPOSIÇÃO

É um tipo especial de agregação na qual as partes são inseparáveis do todo. Indicada por um diamante cheio.

É uma coleção de operações que possui um nome. É usada para especificar um tipo de serviço sem ditar a sua implementação. Também podem participar de generalizações, associações e dependências.

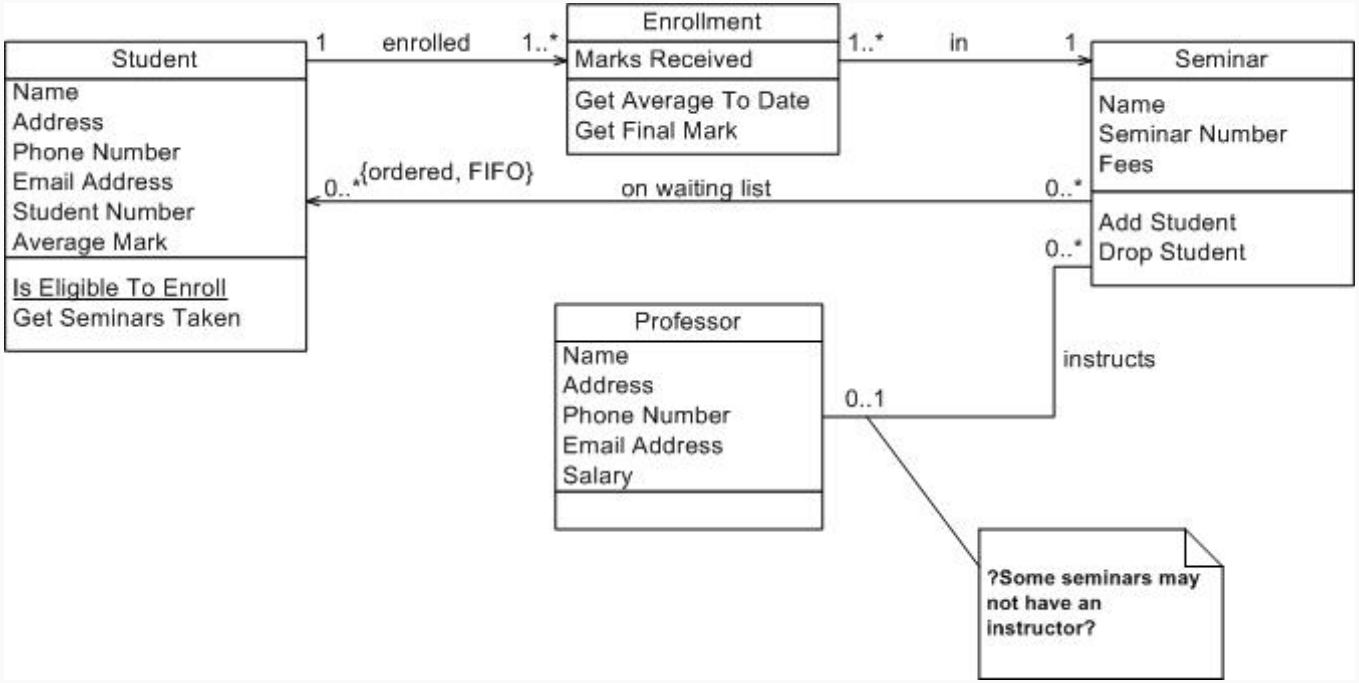


É uma relação entre uma interface e a classe que a implementa, i.e., que provê o serviço definido pela interface.



Uma classe pode realizar (implementar) várias interfaces.

DIAGRAMA DE CLASSES



PACOTES

- Um mecanismo para organizar elementos de um modelo (classes, diagramas, etc.) em grupos
- Cada elemento de um modelo pertence a um único pacote. O seu nome dentro do pacote deve ser único

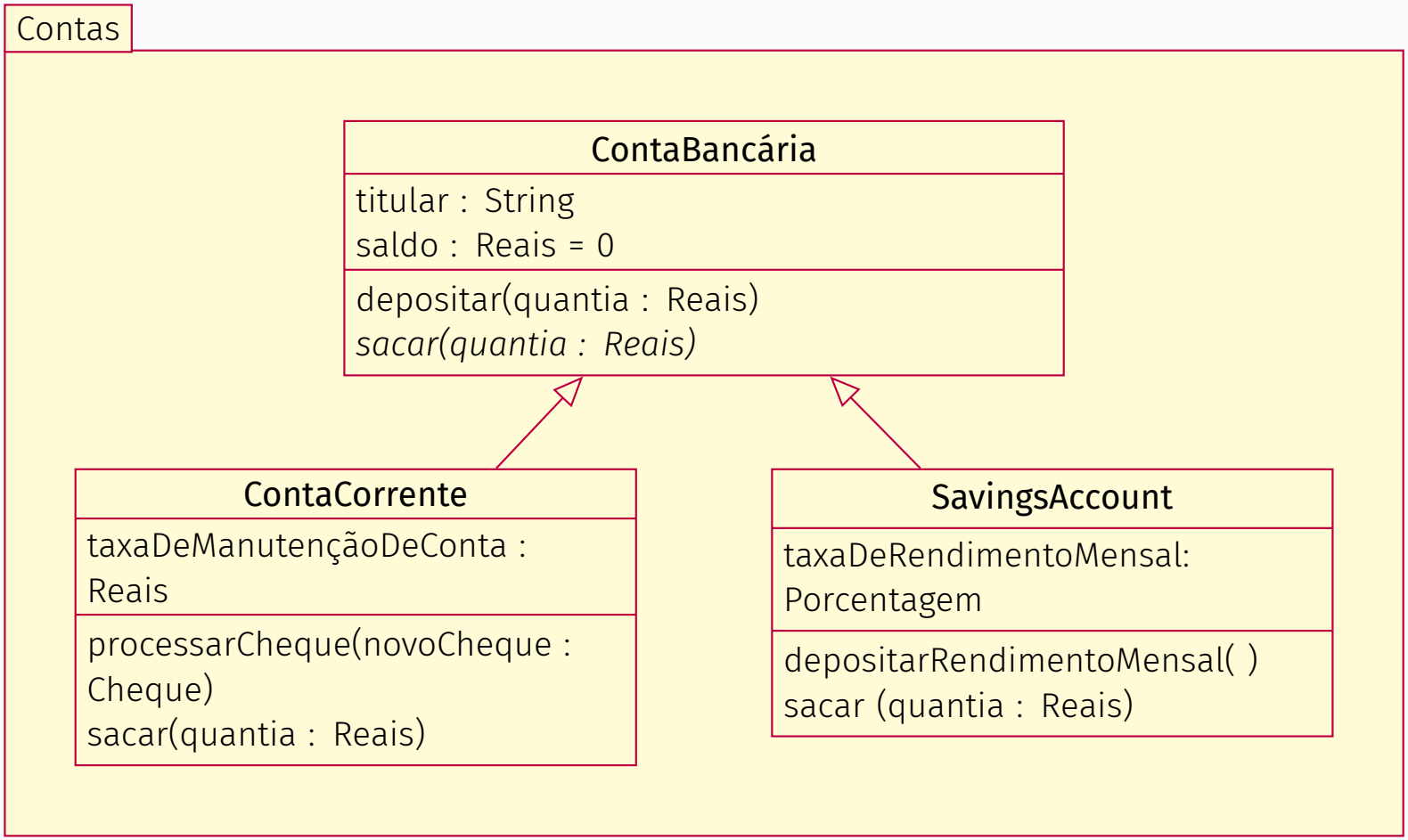
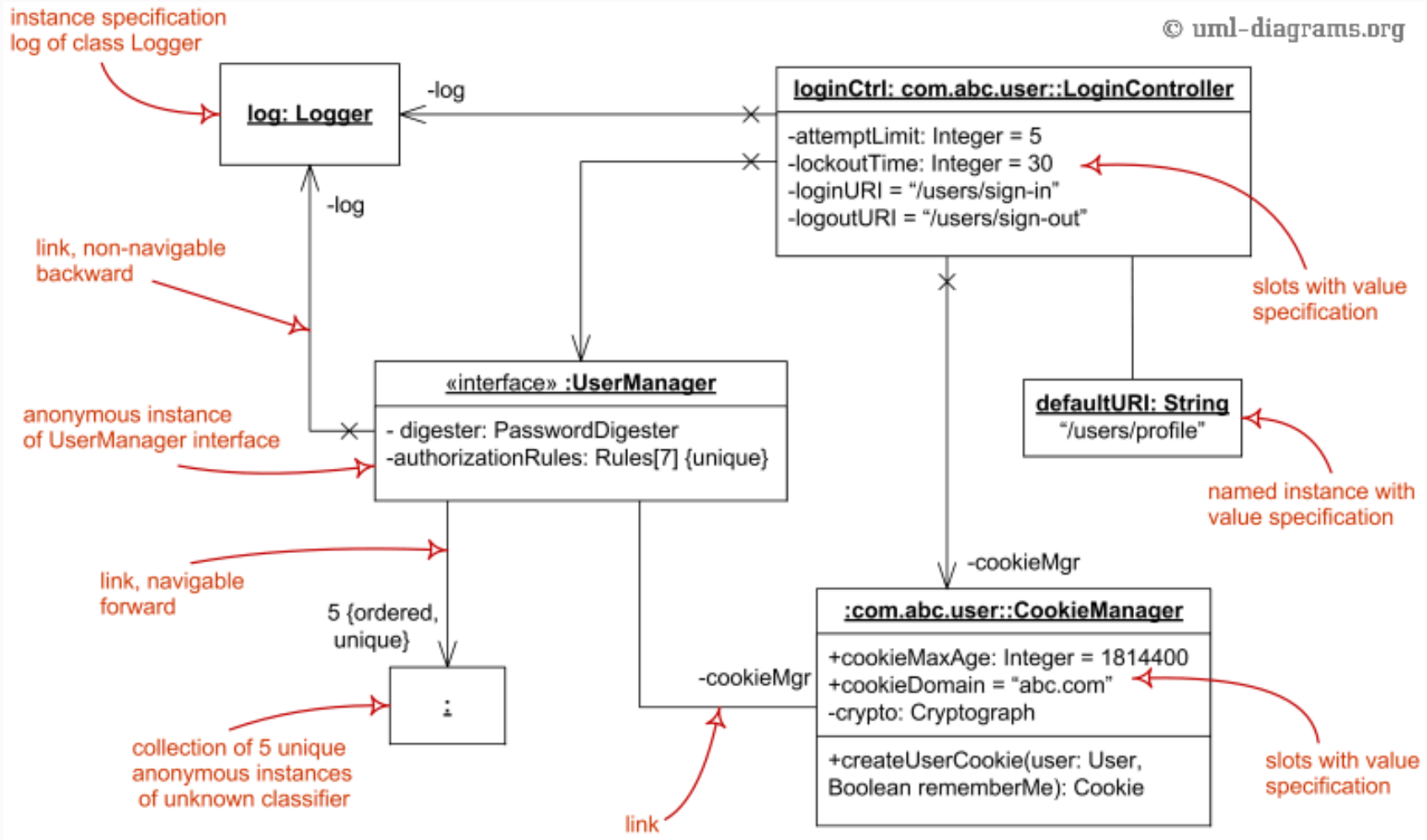


DIAGRAMA DE OBJETOS



Diagramas comportamentais

DIAGRAMA DE CASOS DE USO

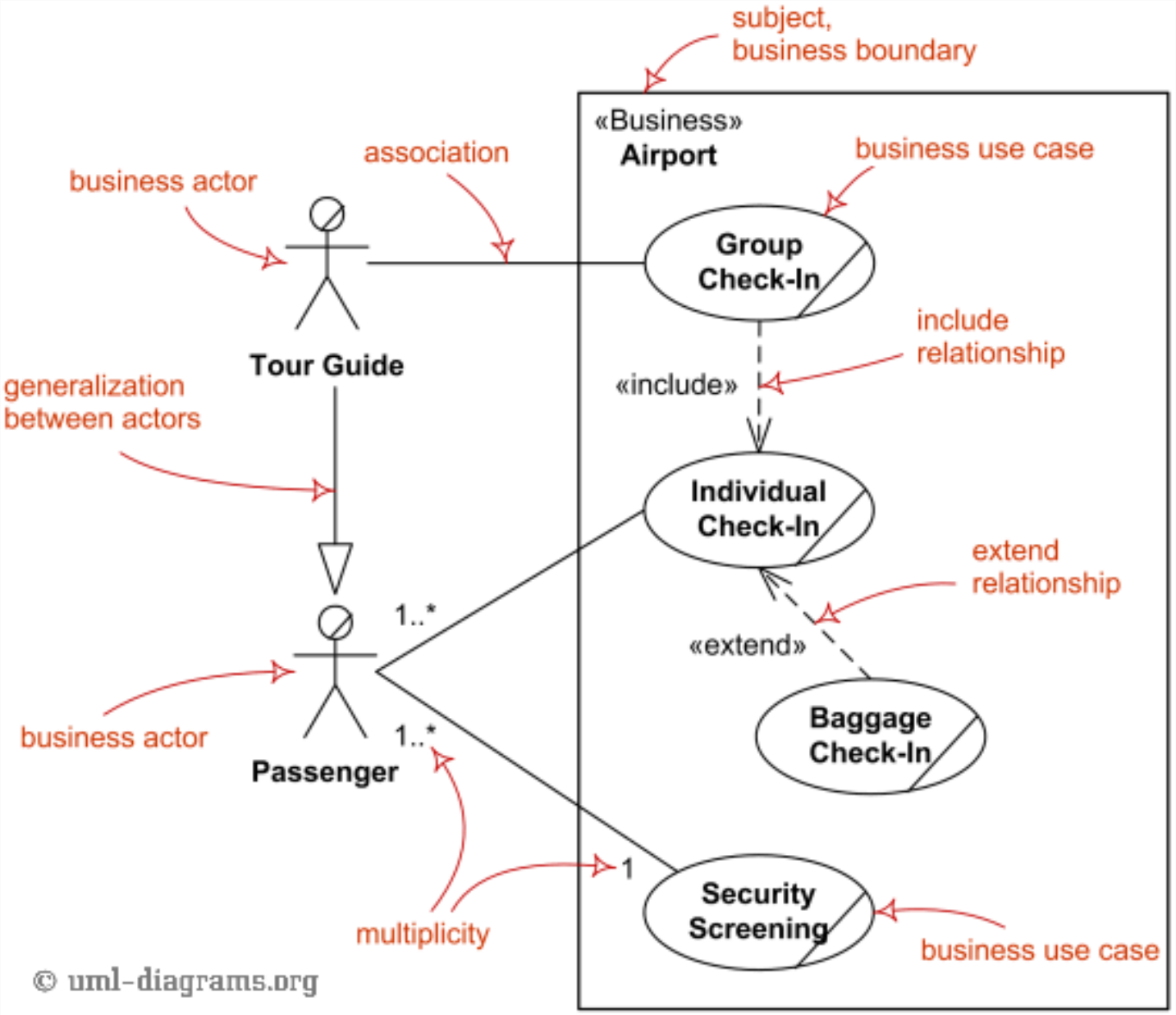
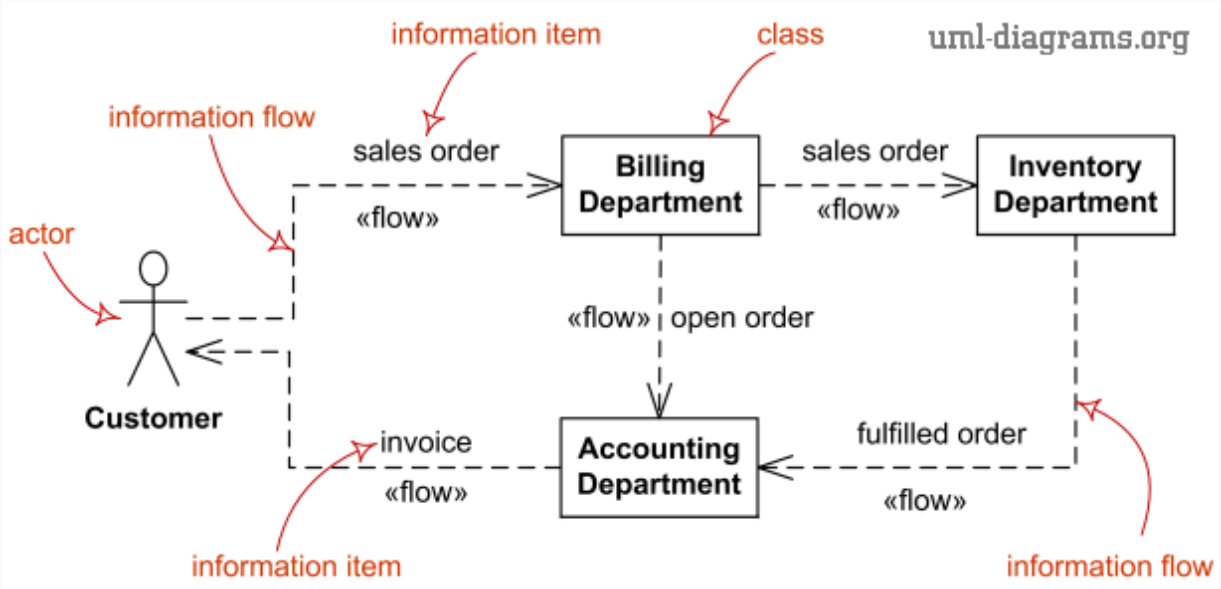


DIAGRAMA DE FLUXOS



- O fluxo de eventos principal descreve o caso em que tudo corre bem.
- Fluxos de eventos excepcionais cobrem as variações que podem ocorrer quando diferentes coisas dão errado ou quando algo pouco comum acontece.

DIAGRAMA DE SEQUÊNCIA

- É um diagrama de interações que enfatiza a ordem temporal das mensagens
- Uma linha de vida é uma linha tracejada vertical que representa o tempo de vida de um objeto
- Um foco de controle é um retângulo fino vertical sobreposto à linha de vida que mostra o período durante o qual um objeto está realizando uma ação

DIAGRAMA DE SEQUÊNCIA

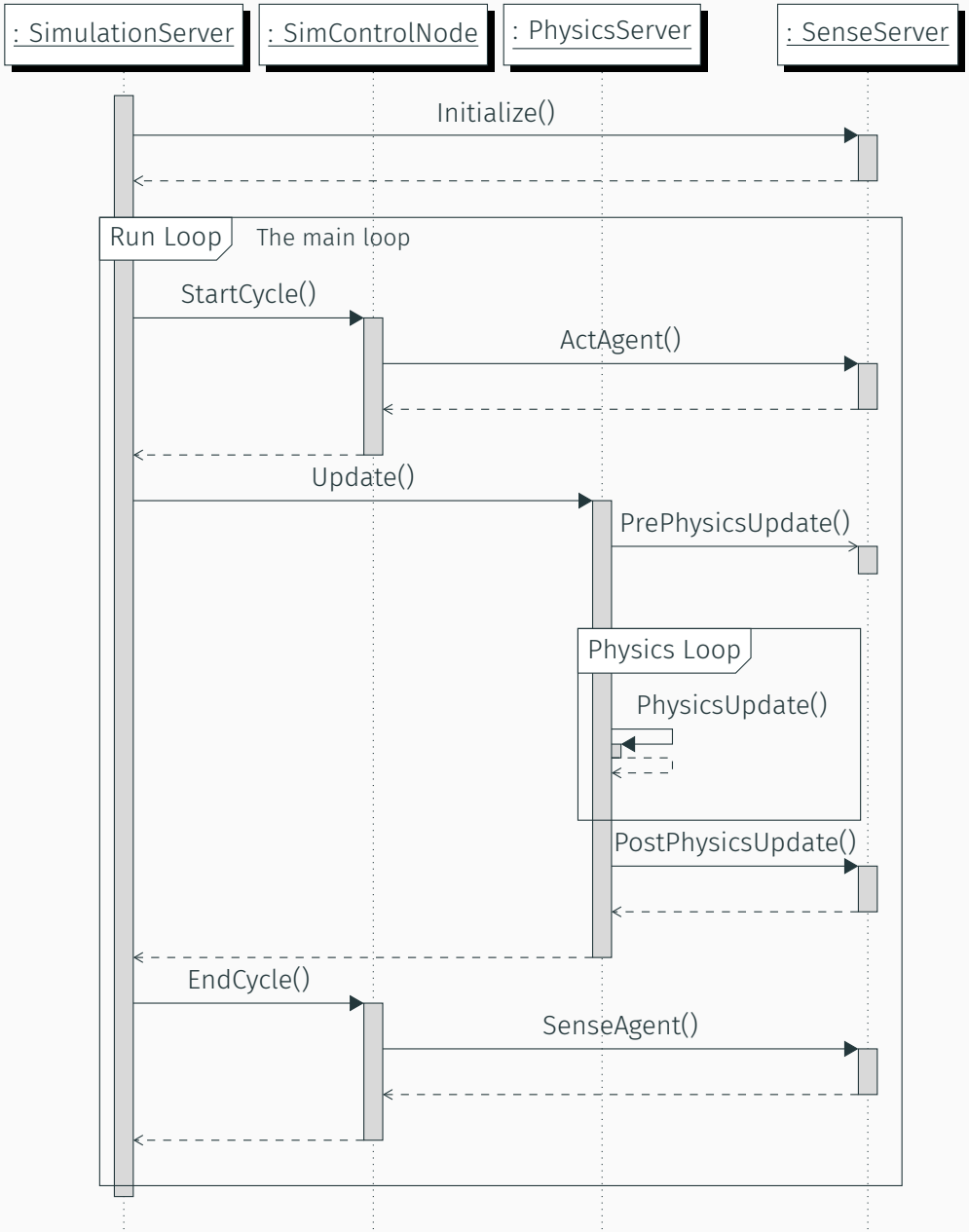
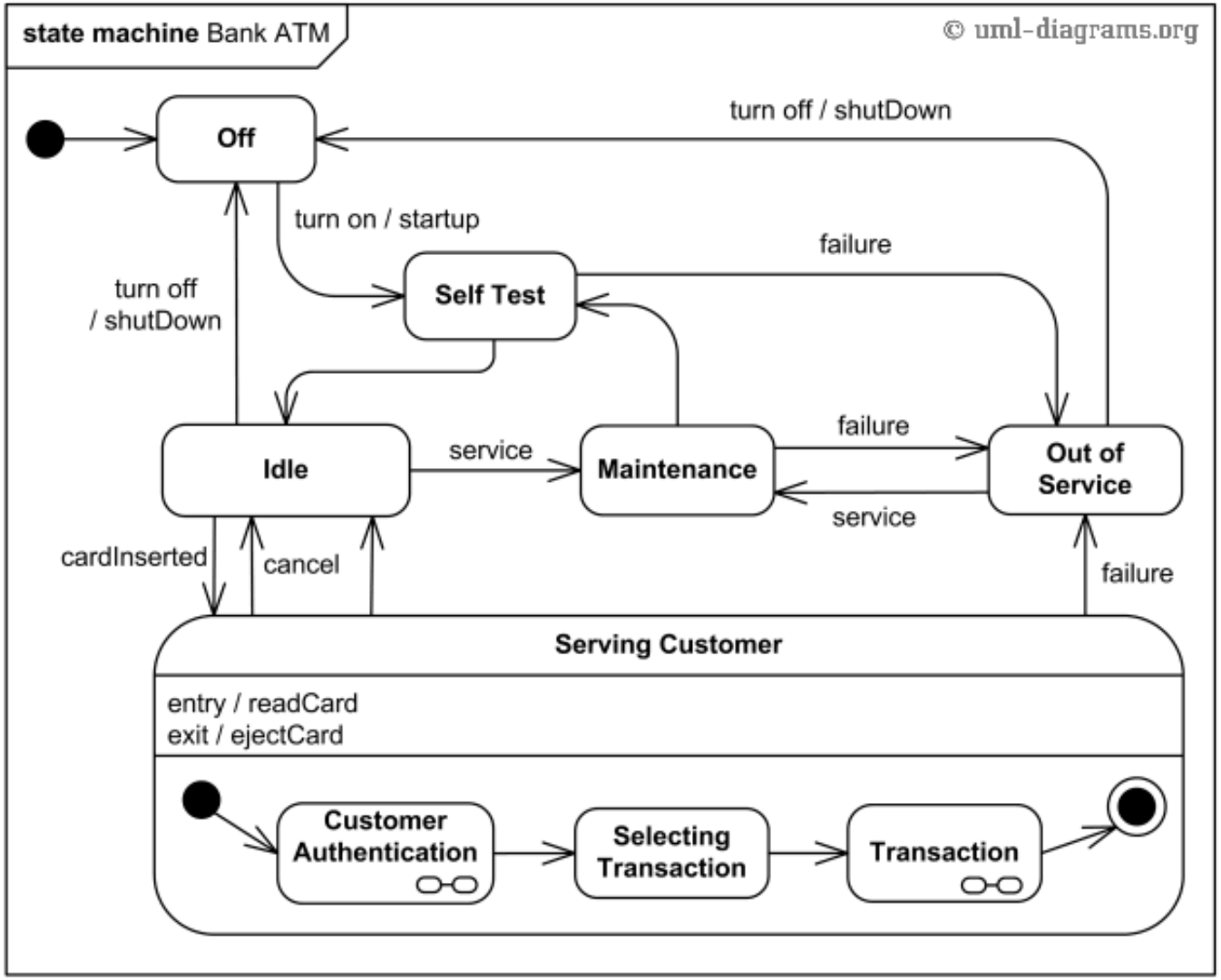


DIAGRAMA DE ESTADOS



USOS DE UML: COMO UM ESBOÇO

- Uso mais comum de UML
- Usado para comunicar algum aspecto do sistema para melhor entendê-lo
- Usado tanto para a engenharia do sistema (ou seja, desenhar os diagramas antes de escrever código) como para engenharia reversa (ou seja, desenhar os diagramas para entender melhor o código)
- Se esforça para ser informal e dinâmico
- Enfatiza apenas as classes, atributos, operações e relações de interesse
- Mais preocupado com a comunicação seletiva do que com a especificação completa

USOS DE UML: COMO O DESENHO COMPLETO DO SISTEMA

- O objetivo é completude
- Tem caráter definitivo, enquanto que o esboço tem caráter exploratório
- Usado para descrever o desenho do sistema em detalhes para que o programador possa seguir o desenho e escrever o código correspondente
- A notação tem que ser suficiente para que um programador possa seguir o desenho do sistema
- Às vezes usado por arquitetos de software para desenvolver um modelo de alto nível que mostra apenas as interfaces dos subsistemas ou classes (desenvolvedores ficam responsáveis pelos detalhes de implementação)
- Quando criado como produto de engenharia reversa, os diagramas transmitem com mais facilidade informações sobre o código-fonte

USOS DE UML: COMO LINGUAGEM DE PROGRAMAÇÃO

- Especifica o sistema completo de forma que o código possa ser gerado automaticamente
- Trata UML da perspectiva do software e não da perspectiva do modelo conceitual do problema
- Diagramas são compilados diretamente para código executável, de modo que o UML se torna o código-fonte do programa
- O desafio é fazer com que o UML seja mais produtivo do que uma linguagem de programação
  - na prática não é viável!
  - modelar comportamento é complicado (mesmo com o uso de diagramas de iteração, estado e atividades)

COMPARAÇÃO DOS USOS DE UML

- Esboços em UML são muito úteis tanto no desenvolvimento como na engenharia reversa; tanto no nível conceitual como na perspectiva de software
- UML como desenho completo de um sistema são difíceis de fazer corretamente e tendem a diminuir o ritmo de desenvolvimento; além disso, a implementação acaba mostrando novas necessidades de mudanças
- UML como desenho completo de um sistema feito com engenharia reversa pode ser útil, mas depende de auxílio de ferramentas: se for exibido dinamicamente pode ser muito útil, mas como documentação é um desperdício de tempo e recursos
- o uso de UML como linguagem de programação provavelmente nunca será significante

Padrões de Projetos

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

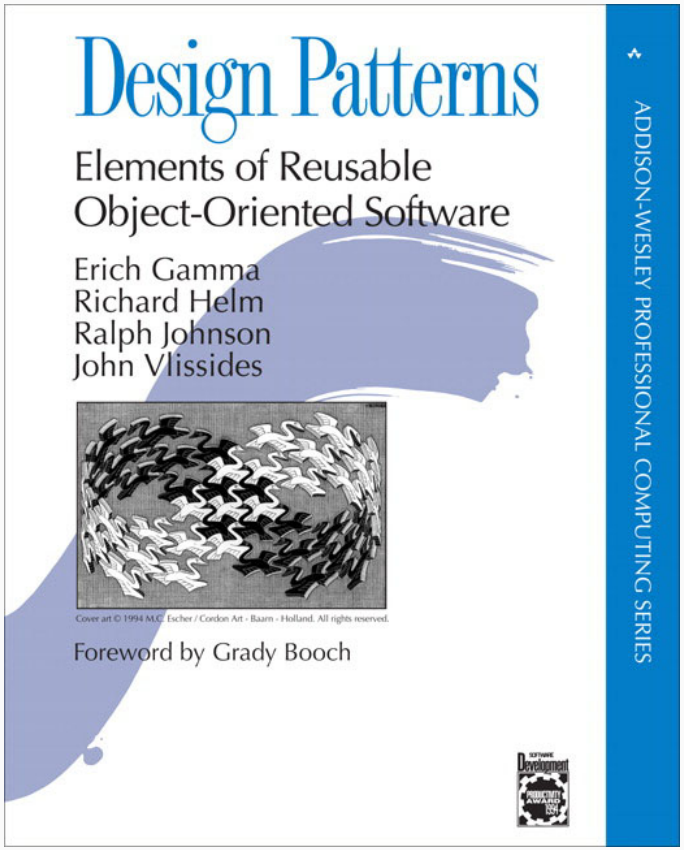
ELEMENTOS DE UM PADRÃO

- 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 toda hora durante o desenvolvimento de software 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 segundo 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

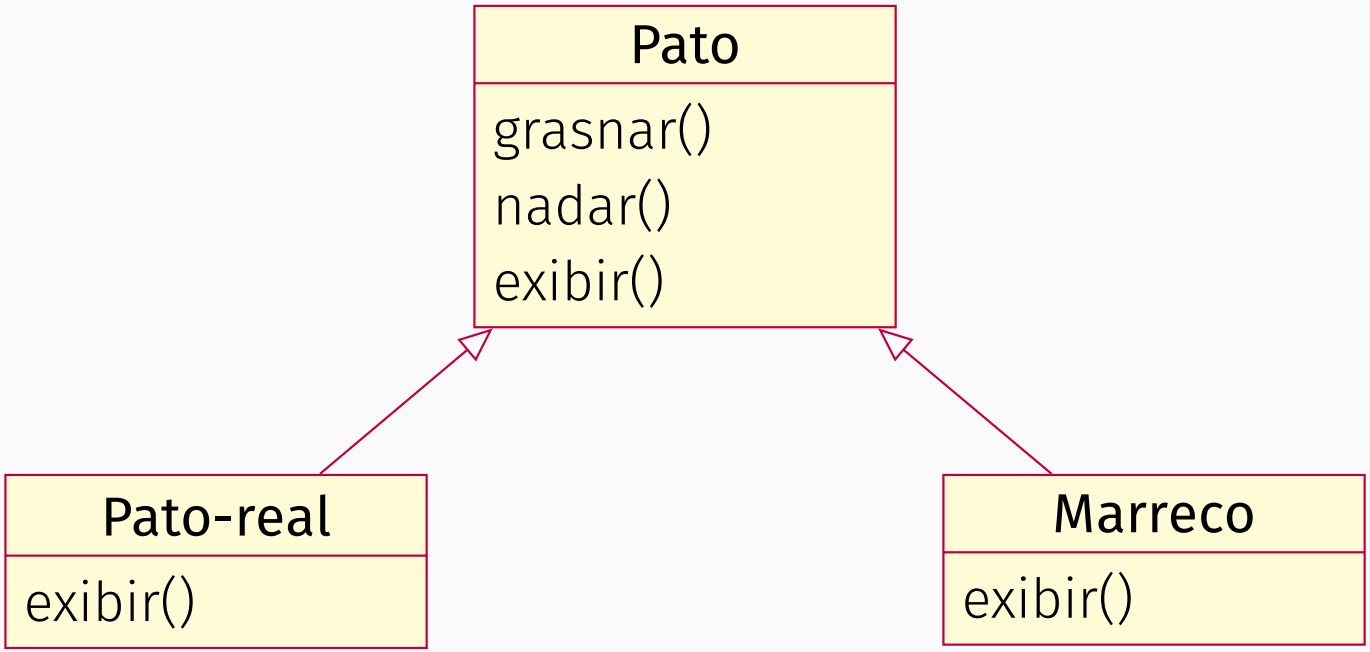
ANALOGIA DO CARPINTEIRO

- “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 a notificação e o registro de quem quer ser notificado da funcionalidade do objeto e permitir que a implementação do objeto se concentre em armazenar e manipular a informação corretamente. Você concorda?

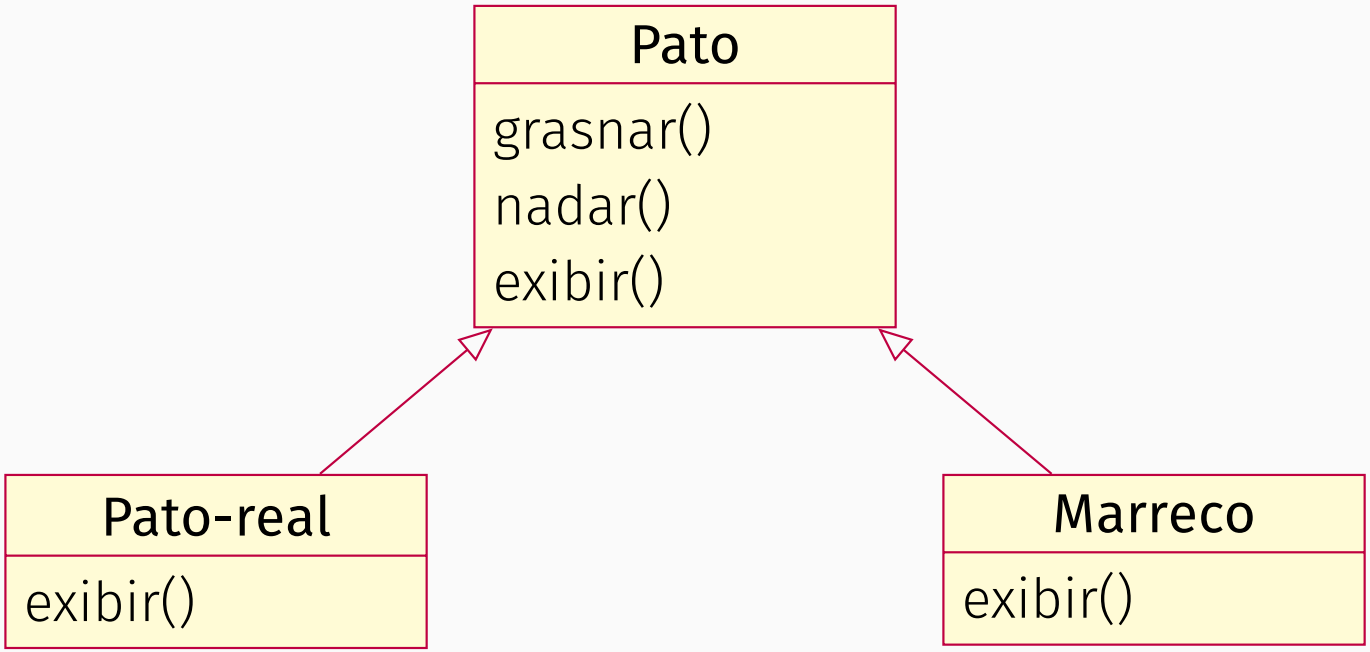
vs.

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

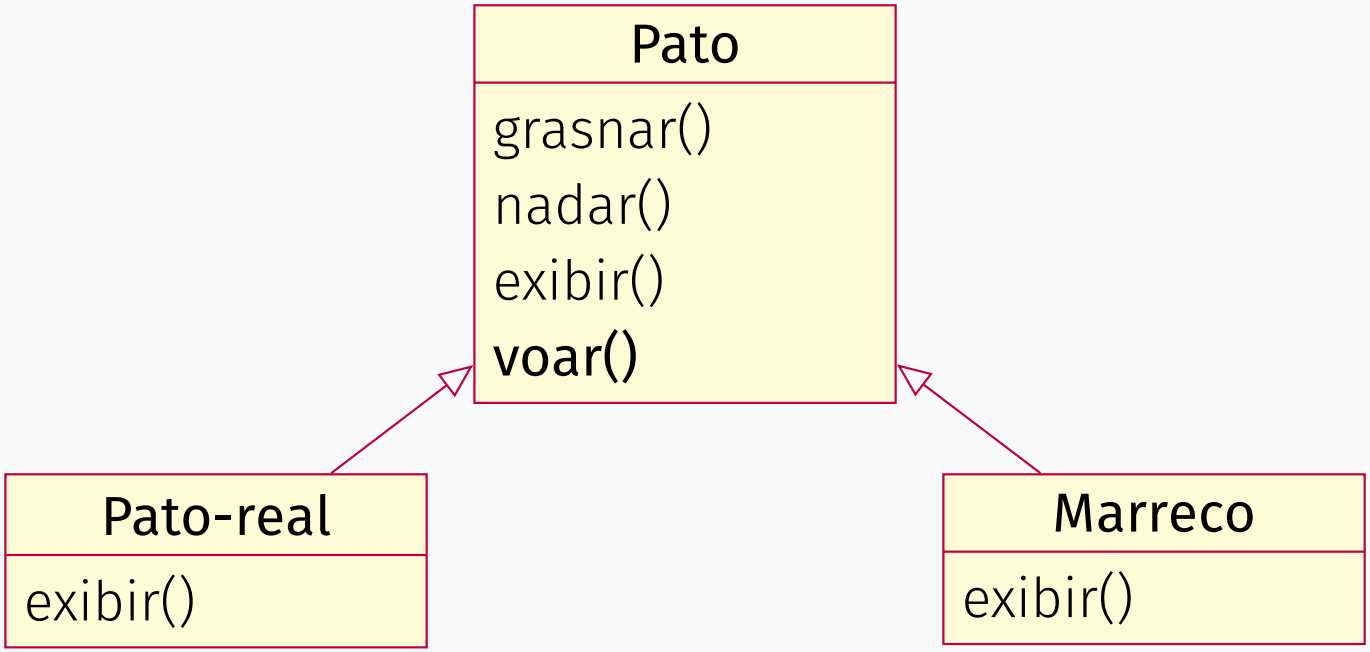
Exemplo:
Imagine um “simulador de patos em uma lagoa”, que é capaz de exibir uma grande variedade de espécies de patos nadando e grasnando.

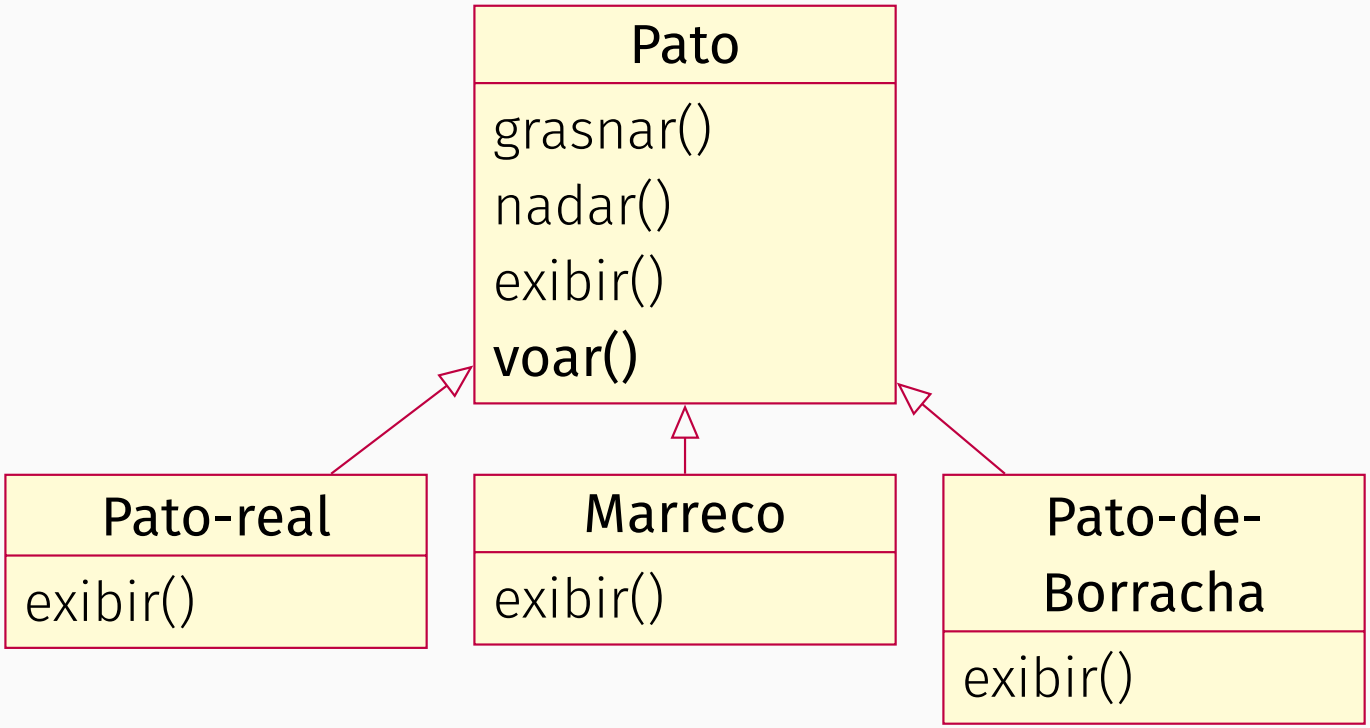


Exemplo:
Imagine um “simulador de patos em uma lagoa”, que é capaz de exibir uma grande variedade de espécies de patos nadando e grasnando.



Novo requisito
Agora os patos devem conseguir voar!

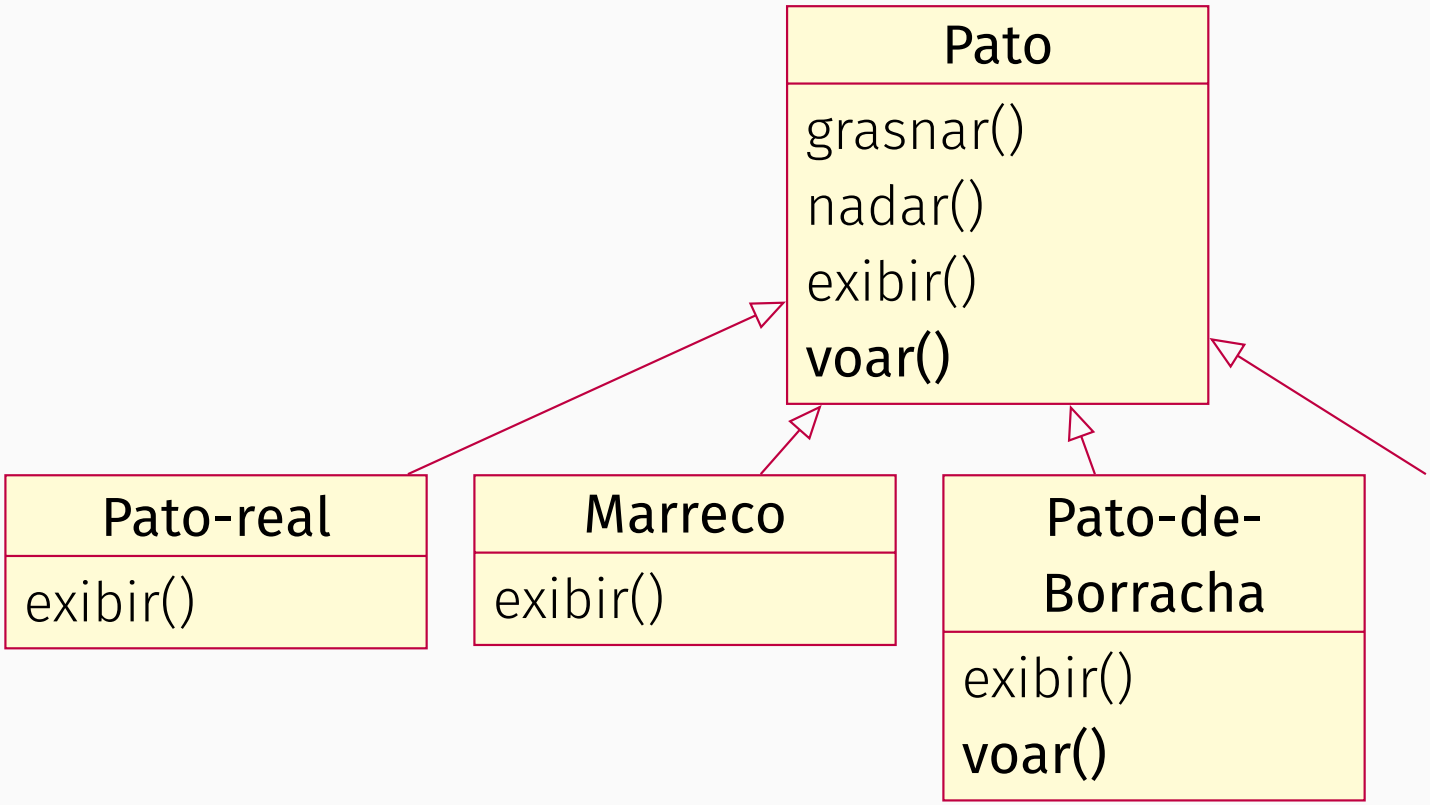

Pato
grasnar()
nadar()
exibir()
voar()
Pato-real
exibir()
Marreco
exibir()



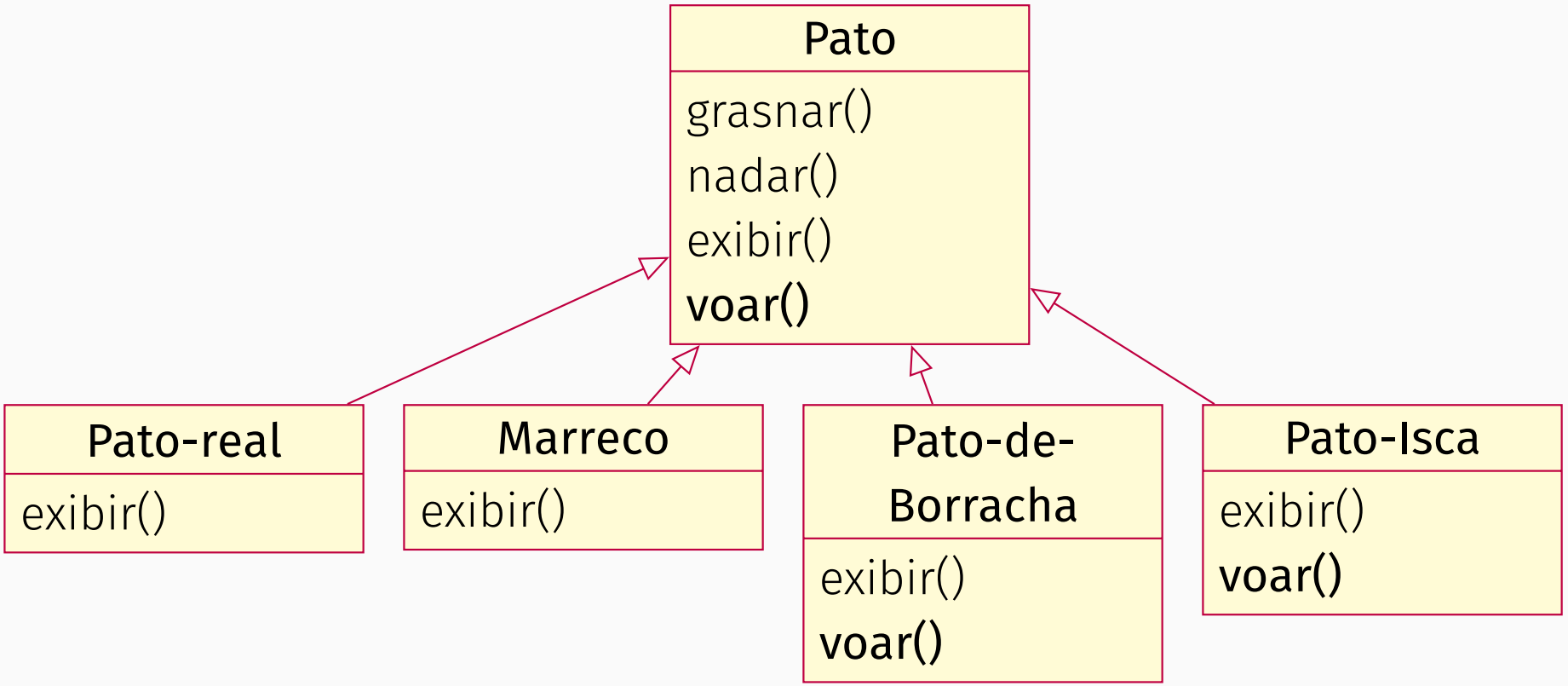
Ops

Patos de borracha não deveriam poder voar!

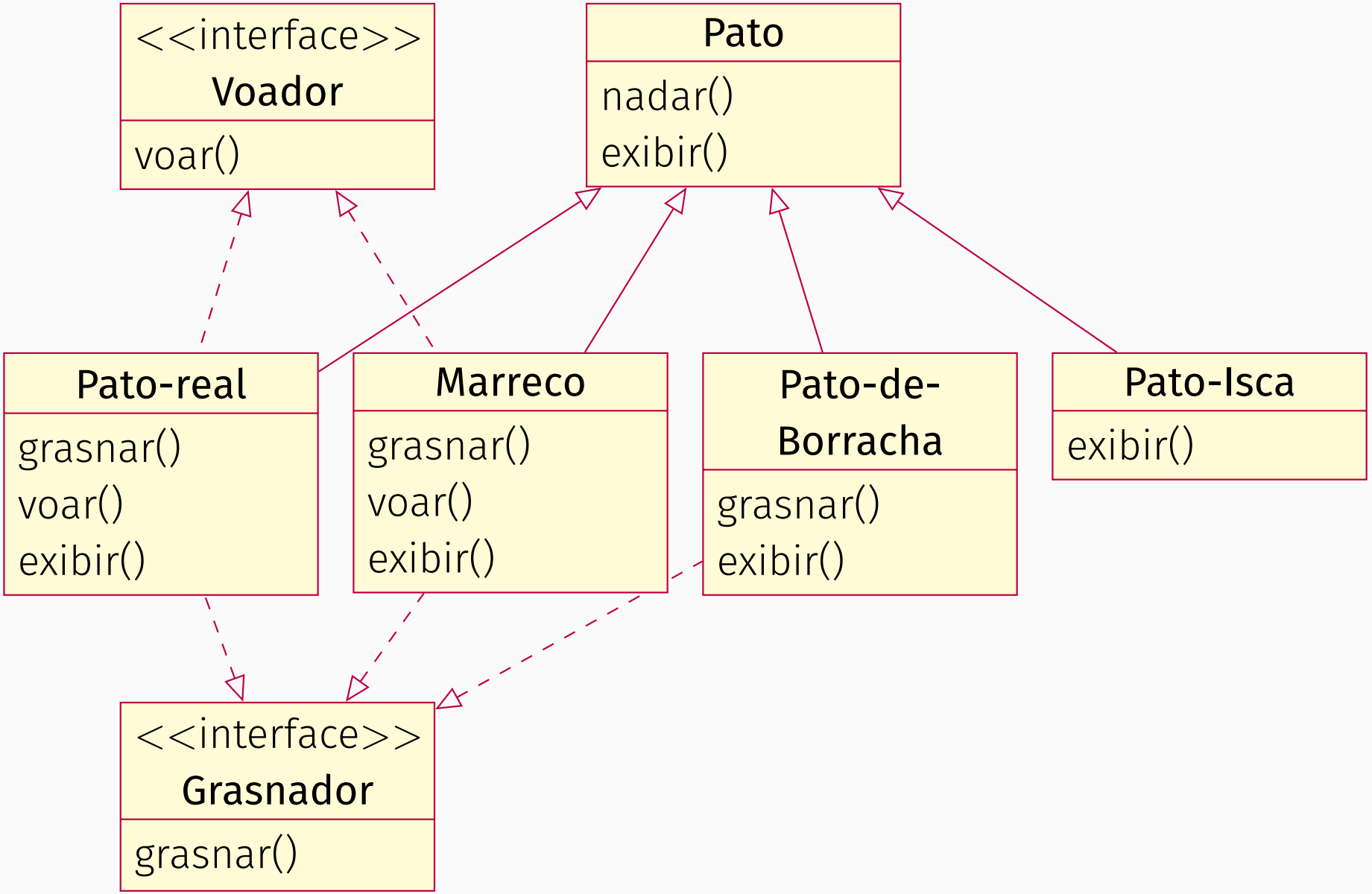
NÃO ESTÁ MAIS TÃO FÁCIL



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

- 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 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

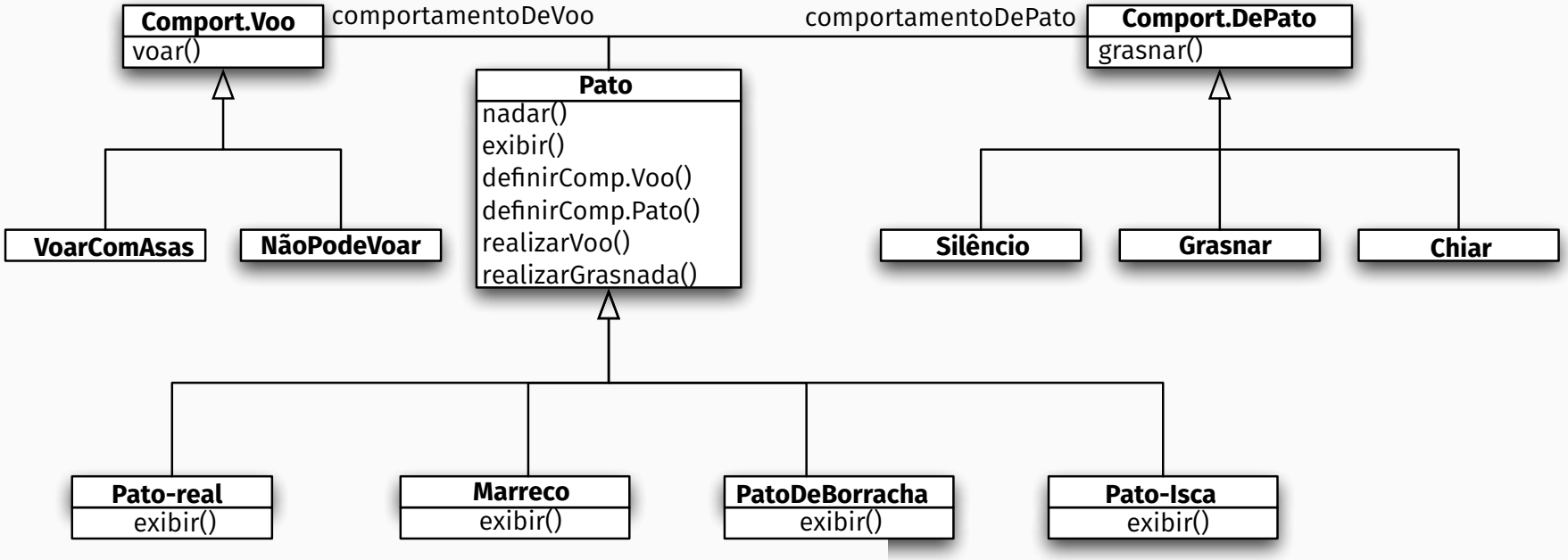
PROGRAMAR PARA UMA INTERFACE ≠ PROGRAMAR UMA INTERFACE JAVA

- 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



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

BIBLIOGRAFIA

- Martin Fowler. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, 2003.
- Scott W. Ambler. Introduction to the Diagrams of UML 2.X `http://www.agilemodeling.com/essays/umlDiagrams.htm`
- Fabio Kon. Uma Visão Geral de UML. `http://www.ime.usp.br/~kon/presentations/UMLIntro.pdf`
- `http://www.uml-diagrams.org/`
- The Gang of Four Book, ou GoF: E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns — Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.