Desenvolvimento de Sistemas Orientados a Objetos I

Associação, Agregação, Composição e Coleções

Jean Carlo Rossa Hauck, Dr.

jean.hauck@ufsc.br

http://www.inf.ufsc.br/~jeanhauck

Conteúdo Programático

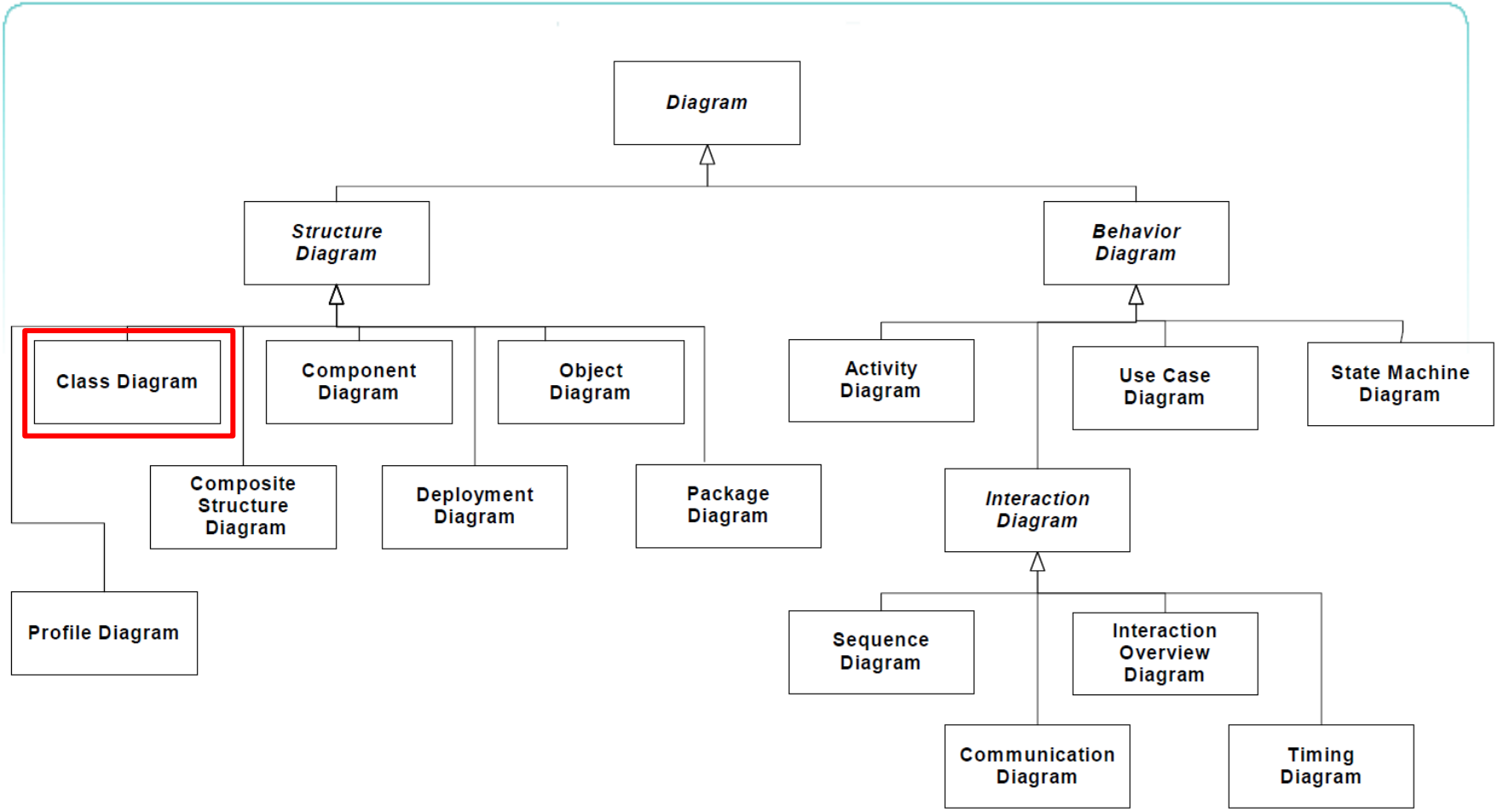
- Conceitos e mecanismos da programação orientada a objetos
  - Objetos e classes
  - Diagramas de classes
    - Herança, Associação, Agregação, Composição
- Técnicas de uso comum em sistemas orientados a objetos
  - Coleções

UML (*Unified Modeling Language*)

- Linguagem de Modelagem Unificada → padrão OMG (*Object Management Group*) desde 1997 que unificou em uma linguagem comum, diferentes notações existentes na época
- Oferece uma notação gráfica baseada em vários diagramas que permitem a modelagem visual de programas orientados a objeto
- Independente de linguagem de programação

[http://www.uml.org/]

Visão geral da notação UML



[OMG, 2015]

Acoplamento entre Classes

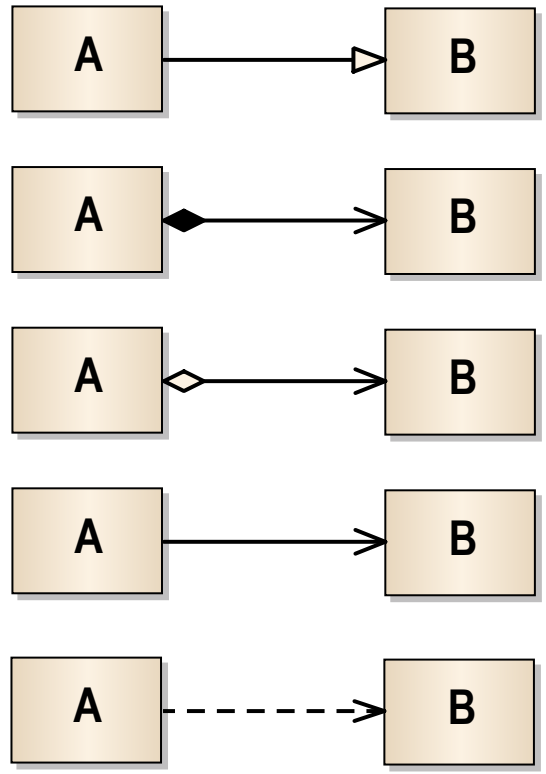
Generalização:

Composição:

Agregação:

Associação:

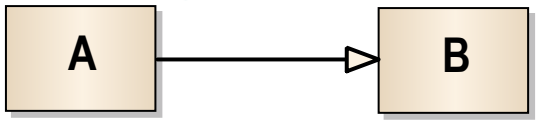
Dependência:



Grau de Acoplamento

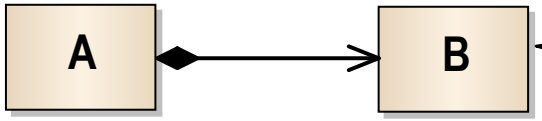
RESUMINDO

Generalização:



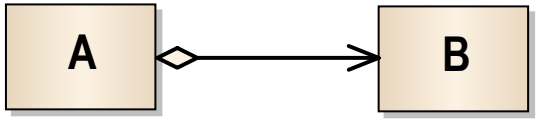
A é um tipo de **B**

Composição:



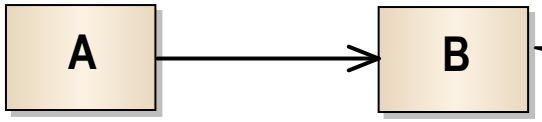
A contém **B**, **A** é composto por **B**
B é propriedade exclusiva de **A**
A instancia e destrói **B**

Agregação:



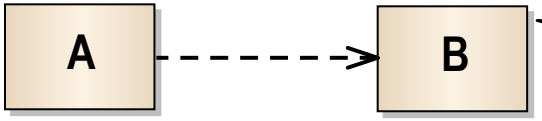
A contém **B**, **A** é formado por **B**
B é compartilhável

Associação:



A tem atributo
do tipo **B**

Dependência:



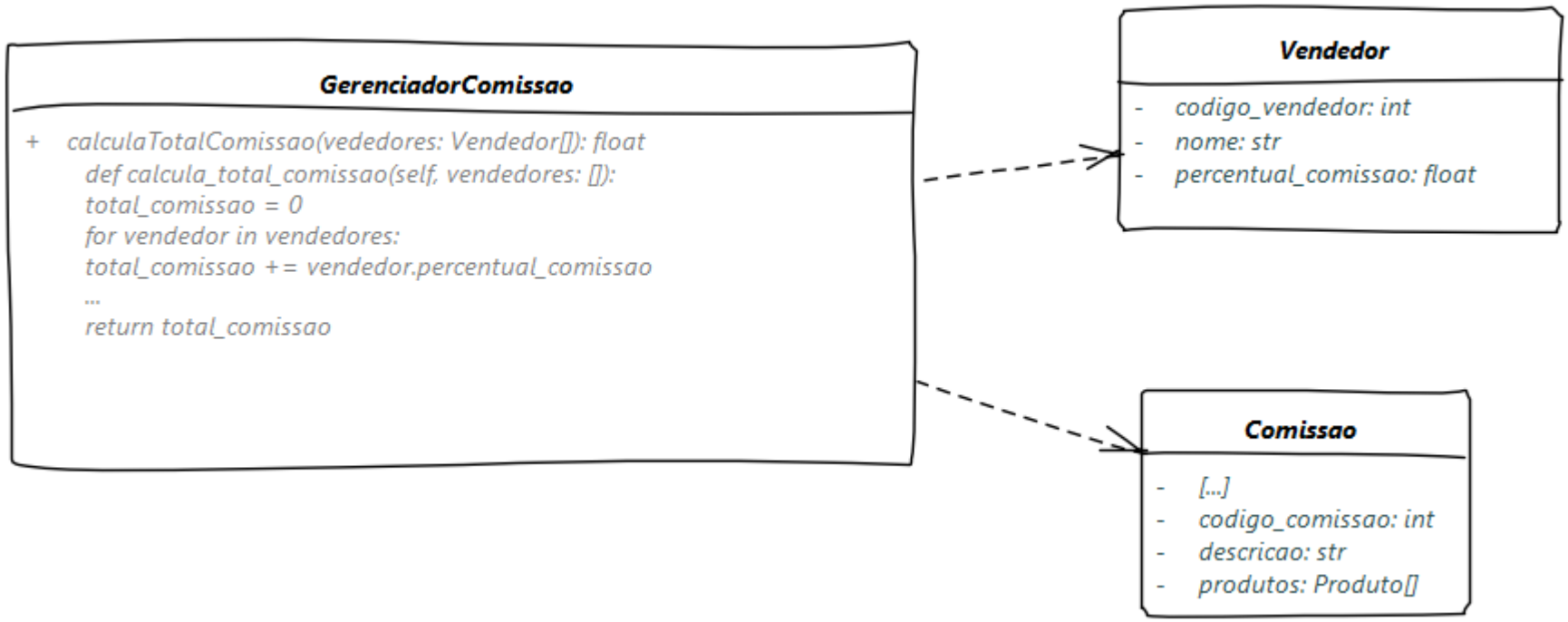
Dependência de **A** para **B**
não é permanente

Principais relacionamentos entre Classes

- **Generalização** (herança): um dos princípios da OO, permite a reutilização, uma nova classe pode ser definida a partir de outra já existente
- **Agregação e Composição**: especializações de uma associação, onde um objeto todo é relacionado com suas partes (relacionamento “parte-de” ou “contenção”)
- **Associação**: relação entre ocorrências (objetos) das classes
- **Dependência**: um objeto depende de alguma forma de outro (relacionamento de utilização)

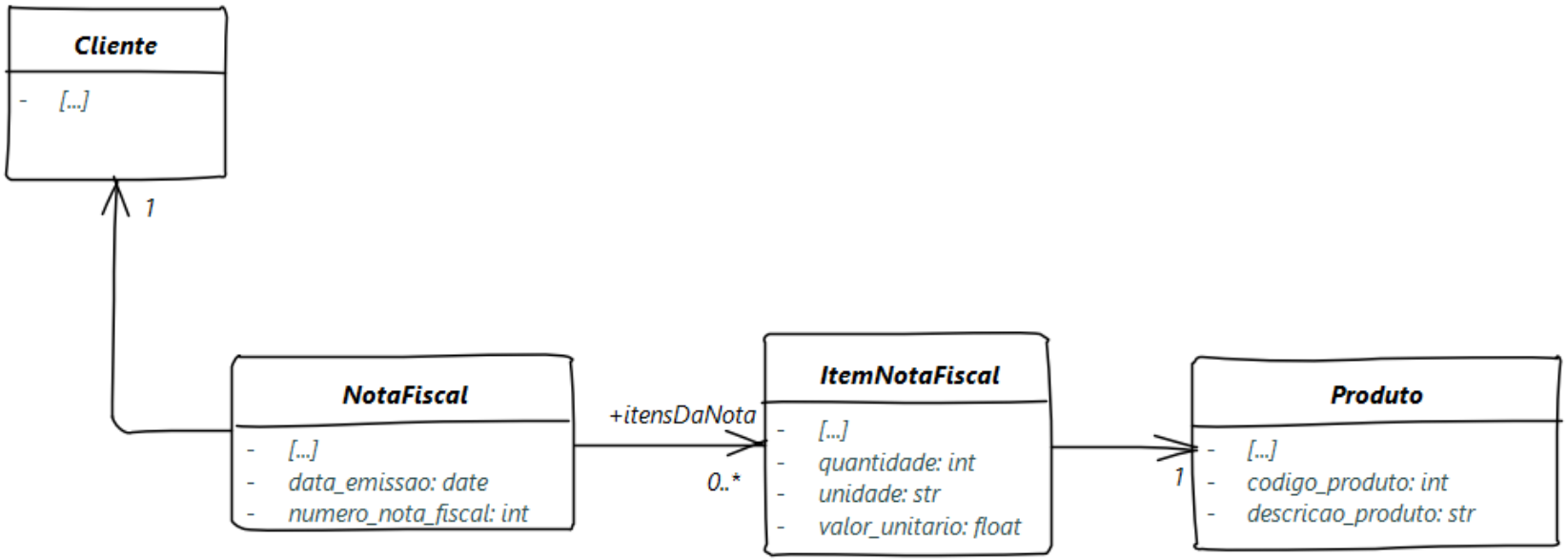
Exemplos

Dependência



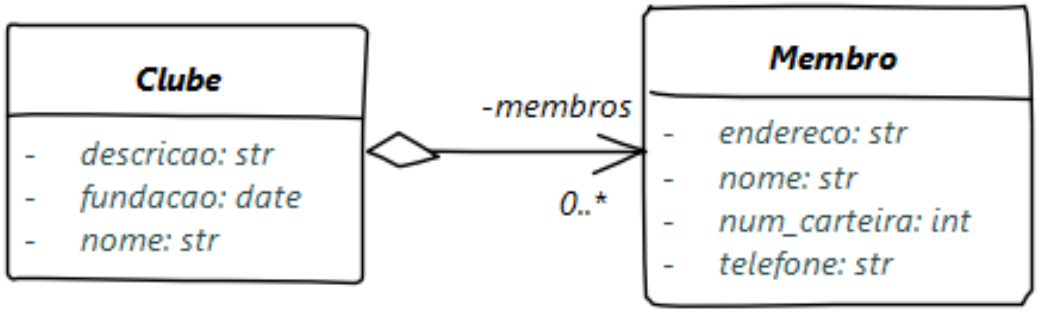
Exemplos

Associação



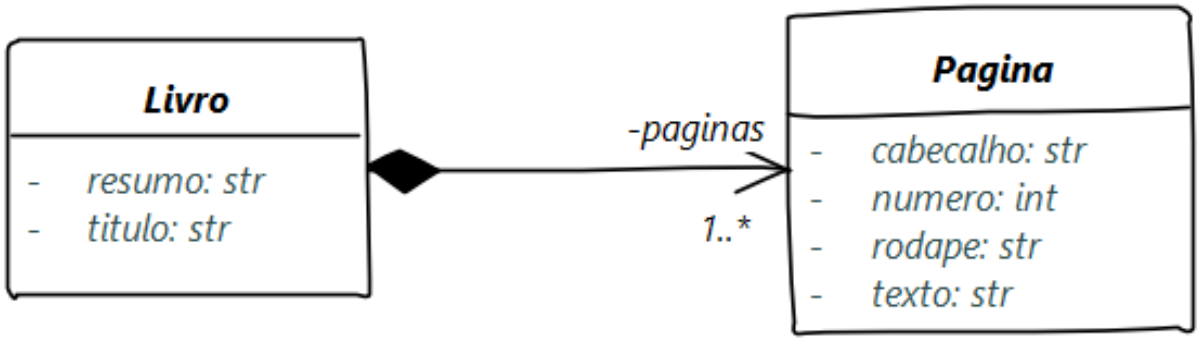
Exemplos

Agregação



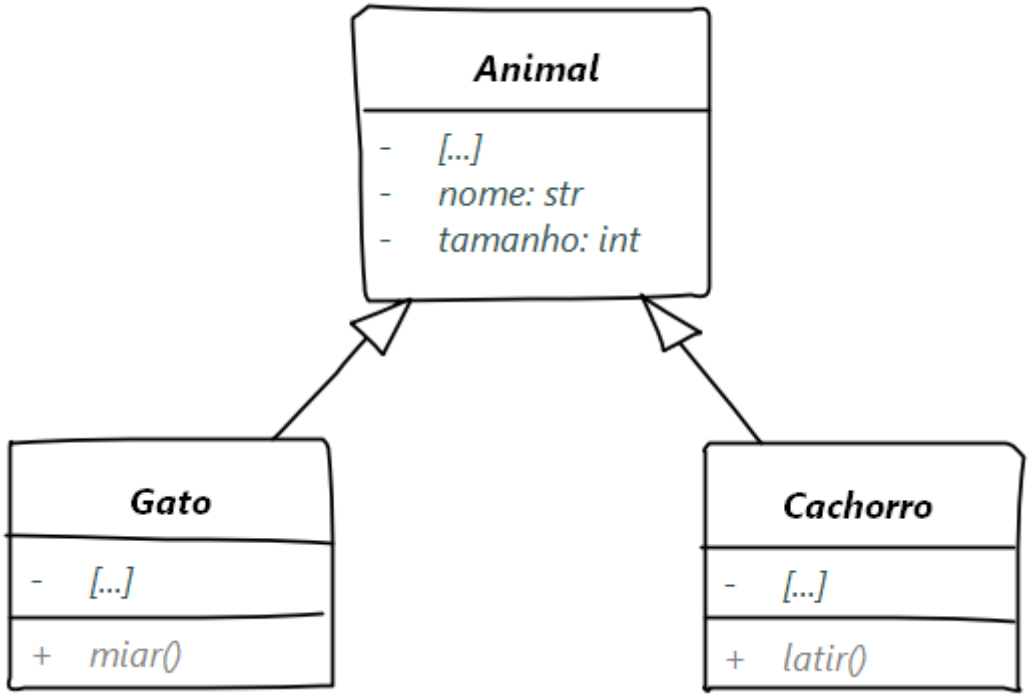
Exemplos

Composição



Exemplos

Generalização



Mais detalhes sobre:
ASSOCIAÇÃO

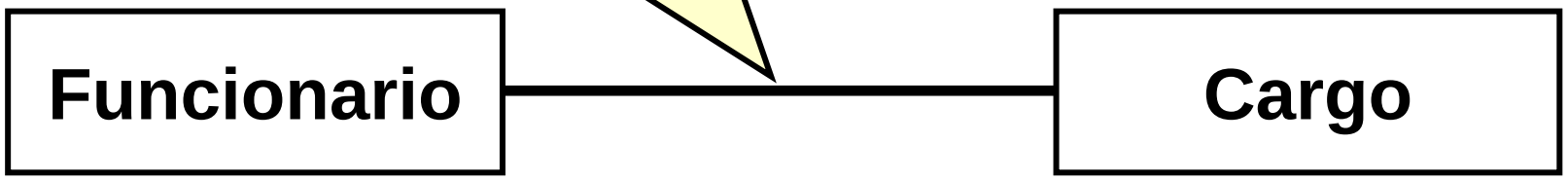
Associação

- Associações representam **relações entre objetos**
- Cada associação tem **duas pontas de associação**
- Cada ponta de associação é **ligada a uma das classes** na associação
- Os dados podem fluir **em uma ou em ambas as direções** através da associação

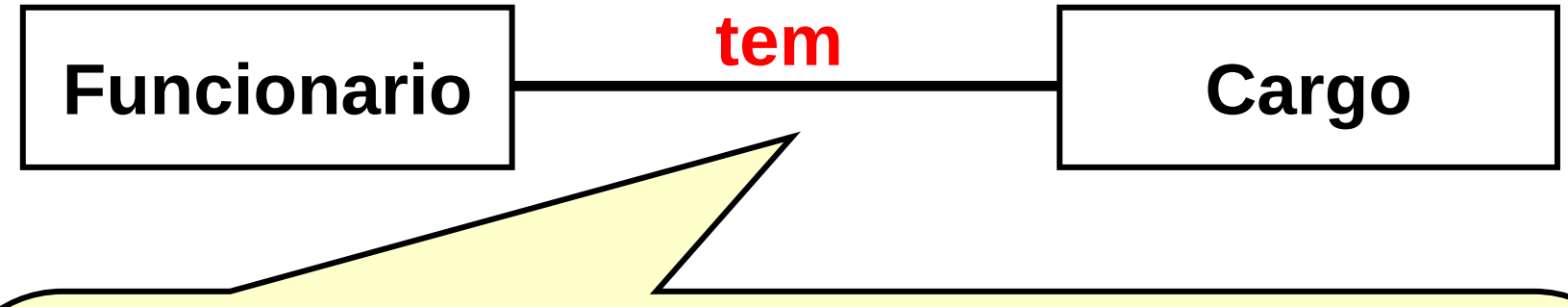
Associação

- Associações representam **relações entre objetos**
- Cada ass **de associaç**
- Cada pon **a uma das class**
- Os dados **em ambas as direções** atr da associação

Mas qual é o significado da associação entre estas classes?



Entendendo uma associação



Indica que o objeto de uma das classes tem objeto(s) da outra classe

Quantos um pode ter do outro?

Mas quem tem quem?

Entendendo uma associação

- Considerando o sentido **Funcionario → Cargo**
  - 1 objeto Funcionario está associado com **1** e somente **1** objeto Cargo
- Considerando o sentido **Cargo → Funcionario**
  - 1 objeto Cargo está associado com **vários (*)** objetos Funcionario

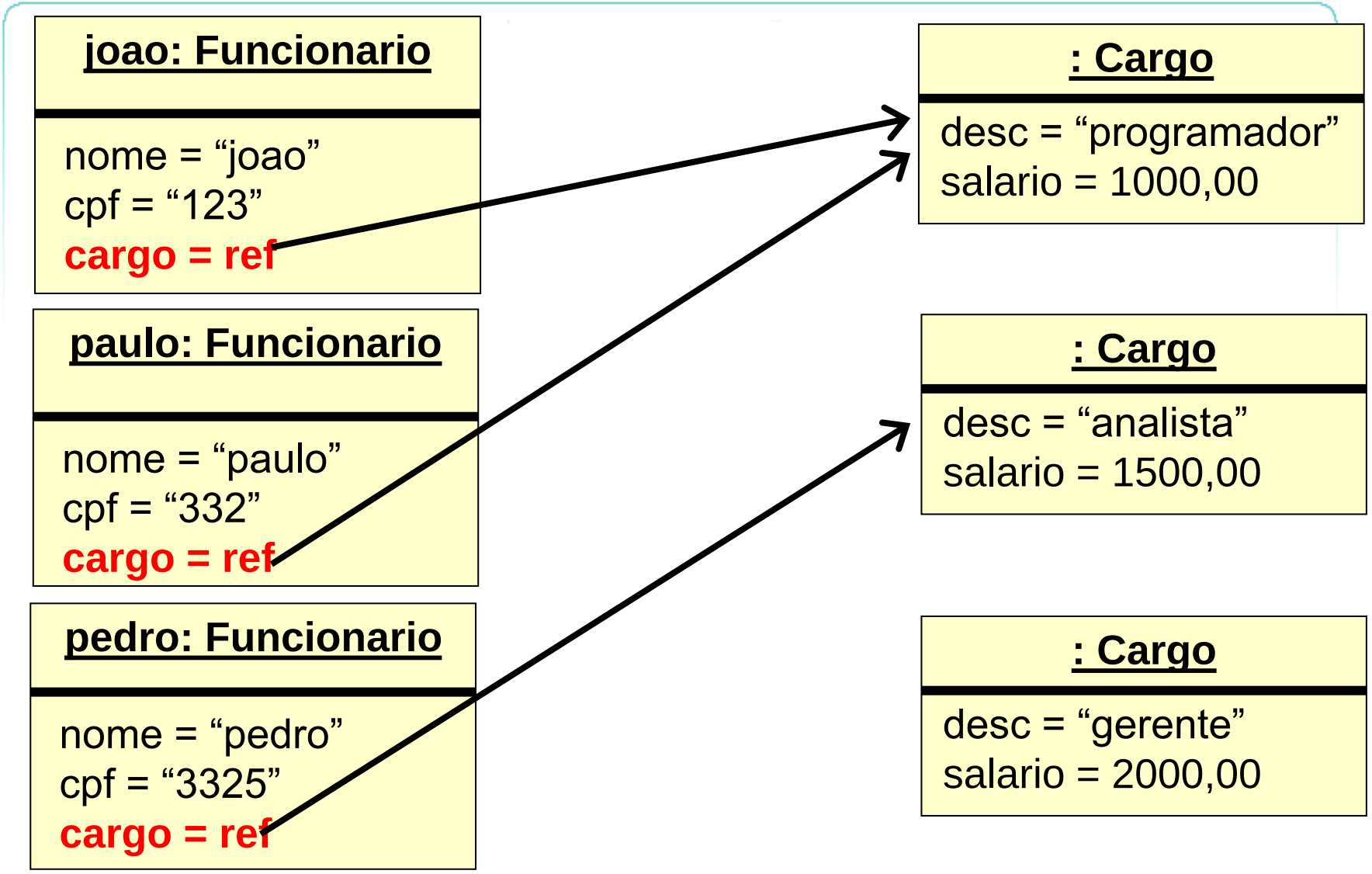
Entendendo uma associação

- Um objeto Funcionario **aponta** para 1 objeto Cargo
- Um objeto Cargo **pode ser apontado por** vários objetos Funcionario

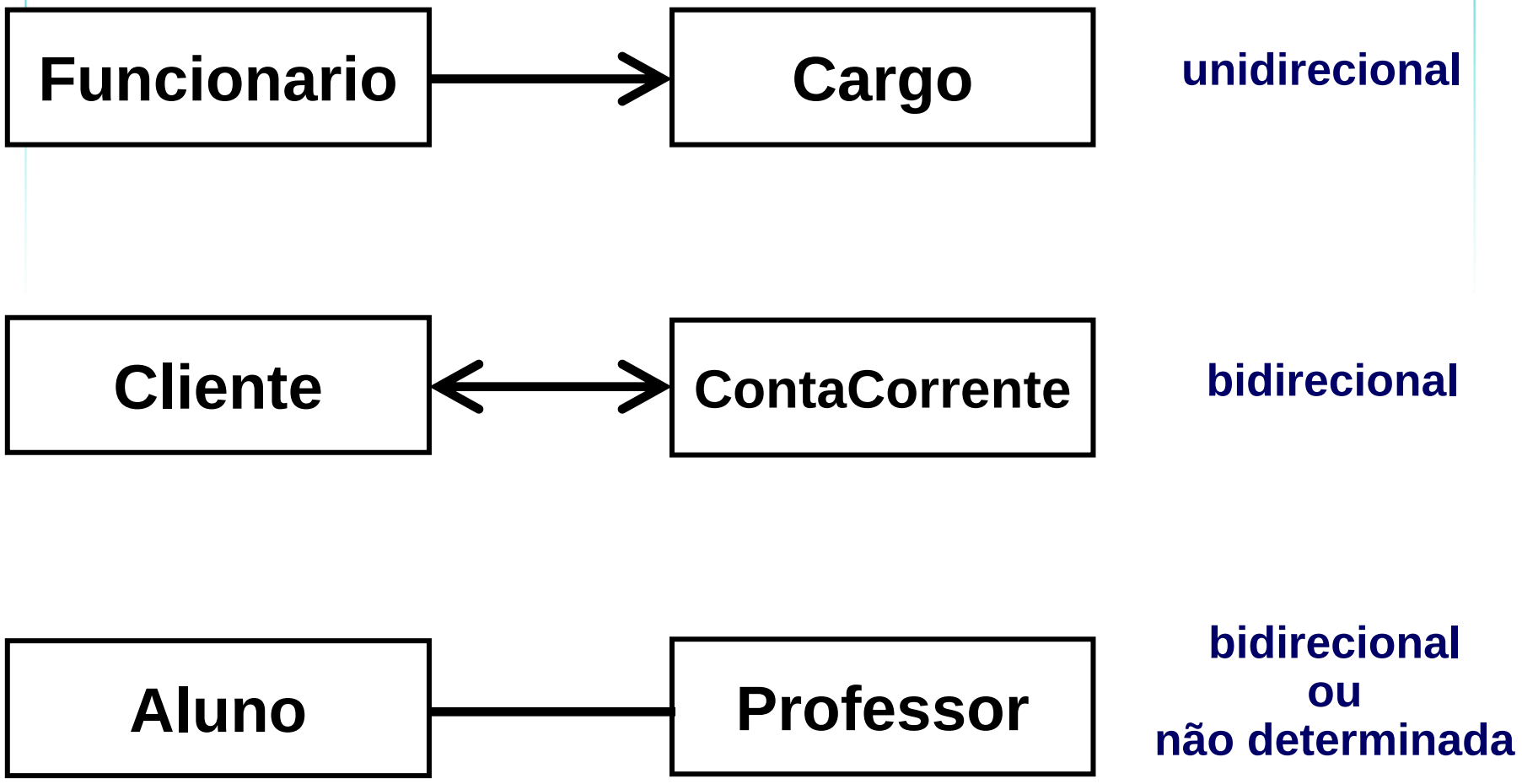
```
class Funcionario:

    def __init__(self, cargo: Cargo):
        self.__cargo = cargo
```

```
class Cargo:
    pass
```

Entendendo uma associação



Navegabilidade

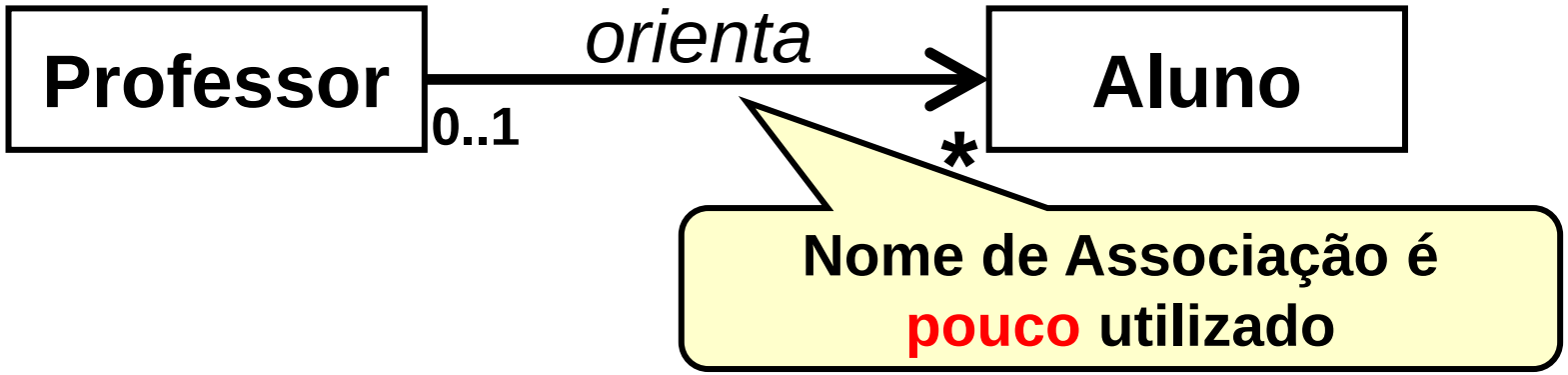


Nomeando associações

- Para facilitar seu entendimento, uma associação precisa ser **nomeada**
- O nome é representado como um **rótulo** colocado ao longo da linha de associação
- Um nome de associação é usualmente um **verbo** ou uma **frase verbal**

Nomeando associações

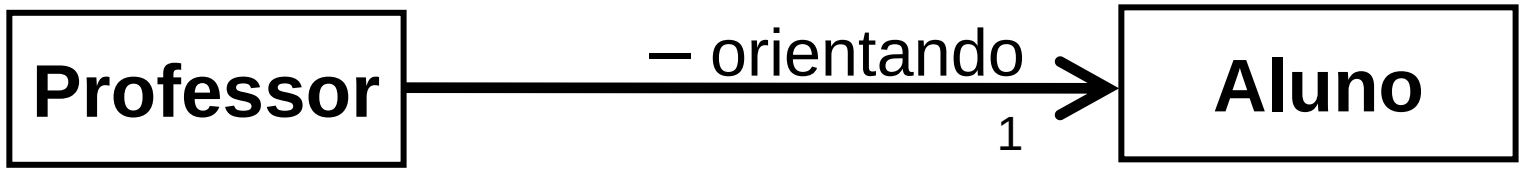
- Para facilitar seu entendimento, uma associação precisa ser **nomeada**
- O nome é representado como um **rótulo** colocado ao longo da linha de associação
- Um nome de associação é usualmente um **verbo** ou uma **frase verbal**



Definindo papéis (*rolenames*)

- As pontas das associação podem ser rotuladas. Este rótulo é chamado **nome de papel**

- Um papel oferece uma interpretação da classe (objeto) para a qual ele está apontando

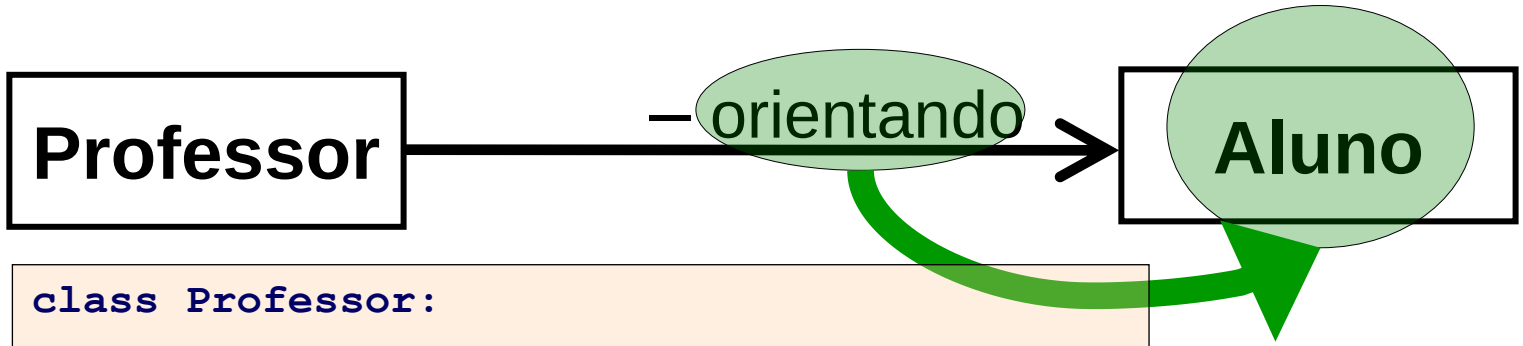



```
class Professor:

    def __init__(self, orientando: Aluno):
        self.__orientando = orientando
```

Definindo papéis (*rolenames*)

- As pontas das associação podem ser rotuladas. Este rótulo é chamado **nome de papel**

- Um papel oferece uma interpretação da classe (objeto) para a qual ele está apontando



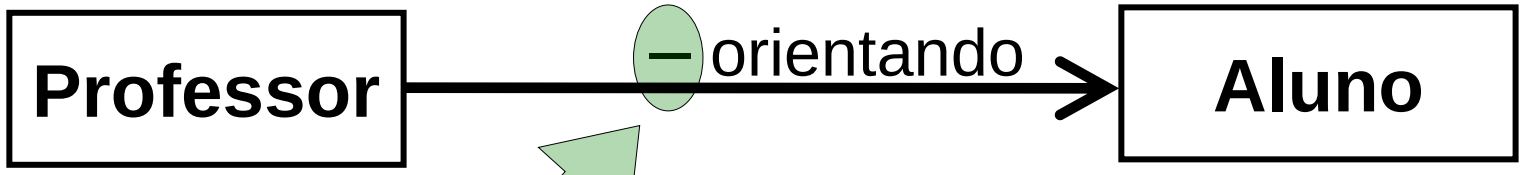

```
class Professor:

    def __init__(self, orientando: Aluno):
        self.__orientando = orientando
```

orientando é um papel de Aluno

Definindo papéis (*rolenames*)

- As pontas das associação podem ser rotuladas. Este rótulo é chamado **nome de papel**

- Um papel oferece uma interpretação da classe (objeto) para a qual ele está apontando

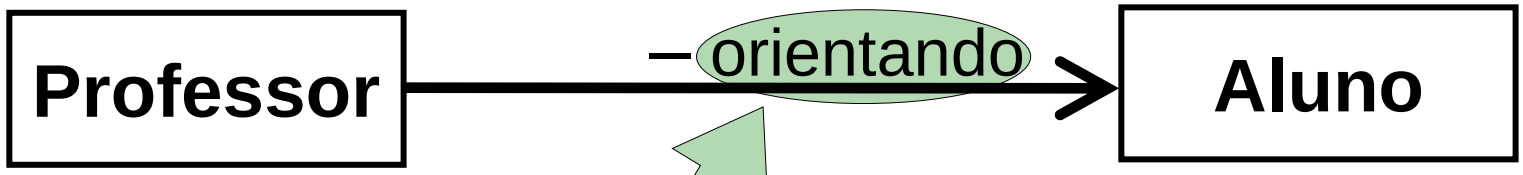



```
class Professor:

    def __init__(self, orientando: Aluno):
        self.__orientando = orientando
```

Definindo papéis (*rolenames*)

- As pontas das associação podem ser rotuladas. Este rótulo é chamado **nome de papel**

- Um papel oferece uma interpretação da classe (objeto) para a qual ele está apontando

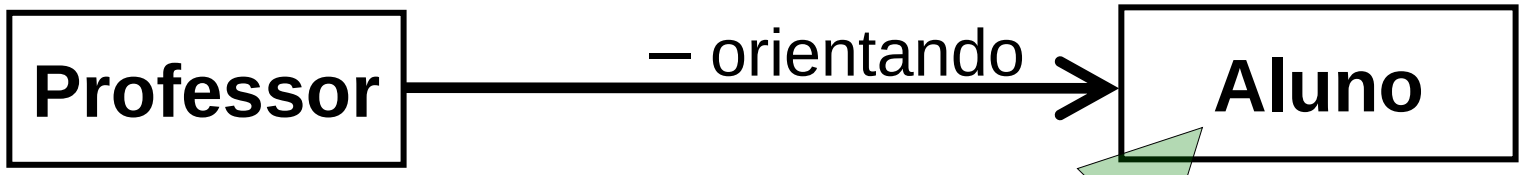



```
class Professor:

    def __init__(self, orientando: Aluno):
        self.__orientando = orientando
```

Definindo papéis (*rolenames*)

- As pontas das associação podem ser rotuladas. Este rótulo é chamado **nome de papel**

- Um papel oferece uma interpretação da classe (objeto) para a qual ele está apontando




```
class Professor:

    def __init__(self, orientando: Aluno):
        self.__orientando = orientando
```

Definindo papéis (*rolenames*)

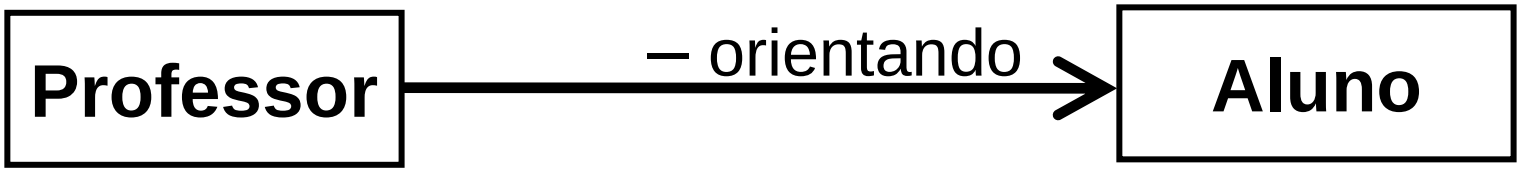
- As pontas das associação podem ser rotuladas. Este rótulo é chamado **nome de papel**
- Um
  ação da
  clas
  stá apontando

Python não obriga parâmetro a ser do tipo Aluno


```
class Professor:

    def __init__(self, orientando: Aluno):
        self.__orientando = orientando
```

Definindo papéis (*rolenames*)

- As pontas das associação podem ser rotuladas. Este rótulo é chamado **nome de papel**
- Um papel oferece uma interpretação da classe (objeto) para a qual ele está apontando



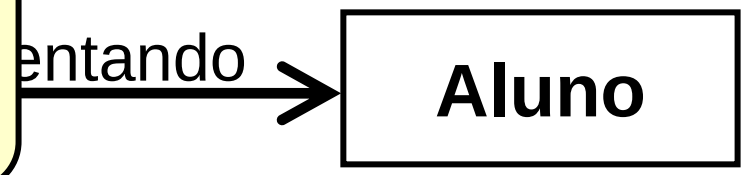

```
class Professor:

    def __init__(self, orientando: Aluno):
        if isinstance(orientando, Aluno):
            self.__orientando = orientando
```

Definindo papéis (*rolenames*)

- As pontas das associação podem ser rotuladas. Este rótulo é chamado **nome de papel**

- Um papel oferece uma interpretação da classe (objeto) para a qual ele está apontando

Mas é possível testar o tipo




```
clas        essor:

    def __init__(self, orientando: Aluno):
        if isinstance(orientando, Aluno):
            self.__orientando = orientando
```

Definindo papéis (*rolenames*)

- As pontas das associação podem ser rotuladas. Este rótulo é chamado **nome de papel**

- Um papel oferece uma interpretação da classe (objeto) para a qual ele está apontando

Professor

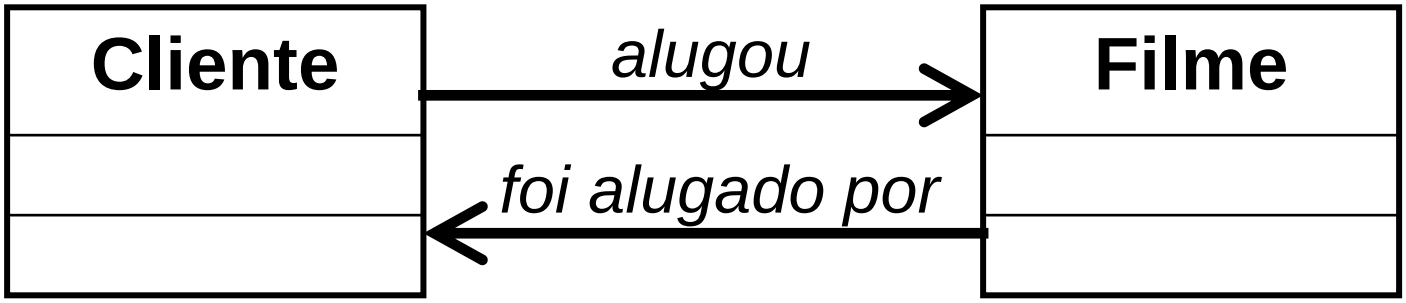
Esta abordagem será discutida quando tratarmos do princípio: **EAFP** (*Easier to ask for forgiveness than permission*)*

```
class Professor:

    def __init__(self, orientando: Aluno):
        if isinstance(orientando, Aluno):
            self.__orientando = orientando
```

* [https://docs.python.org/3.4/glossary.html]

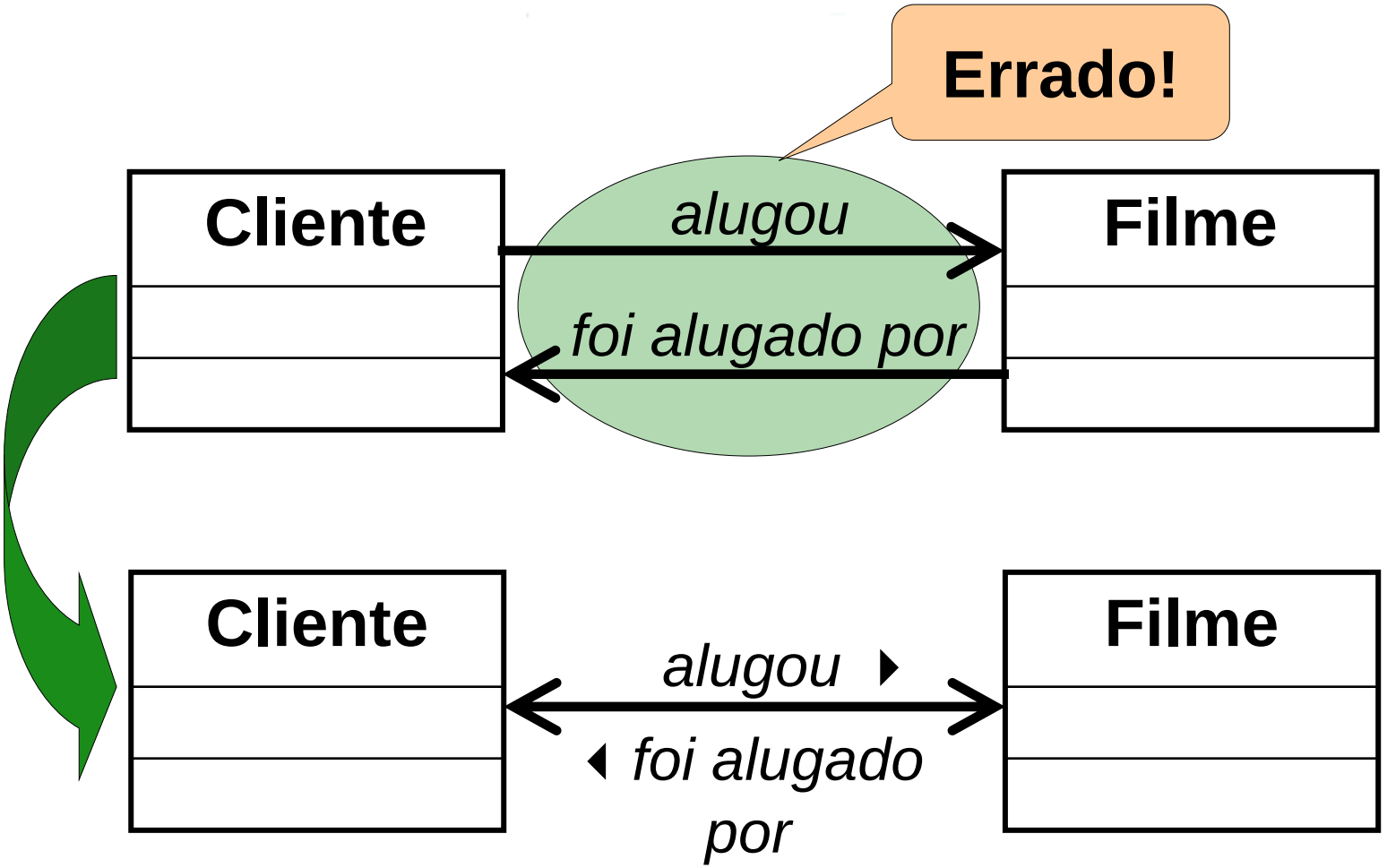
Múltiplas associações

- Podem existir várias associações entre duas classes
- Se há mais que uma associação entre duas classes, então elas precisam ser nomeadas
- **Cuidado**: não mapear por mensagens



São duas associações diferentes?

Entendendo a semântica da associação



Multiplicidade para associações

- Multiplicidade é o número de instâncias de uma classe relacionada com uma instância de outra classe
- Para cada associação, há duas decisões a fazer: uma para cada lado da associação
- Exemplos:
  - Para cada instância de Cliente, podem ocorrer muitas (zero ou mais) instâncias de Filme
  - Para cada instância de Filme, pode ocorrer exatamente uma instância de Cliente

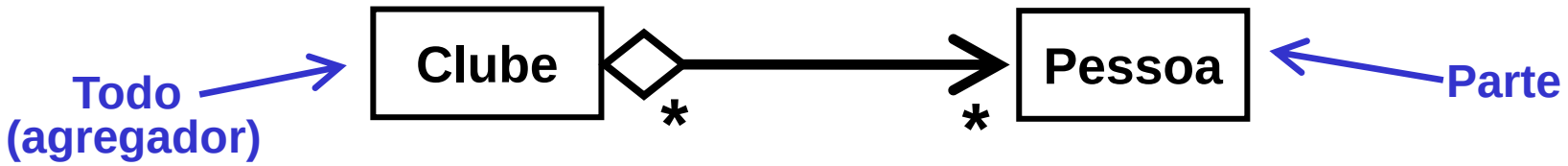
Indicadores de multiplicidade

Muitos/Vários/Zero, um ou mais	*
Muitos/Vários/Zero, um ou mais	0..*
Um ou mais	1..*
Zero ou um	0..1
Exatamente um	1
Faixa especificada	2..4, 6..8

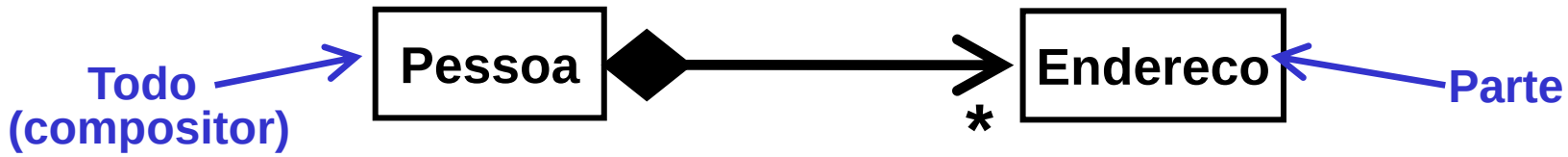
Mais detalhes sobre: AGREGAÇÃO E COMPOSIÇÃO

Agregação e Composição

- Uma **agregação** é uma associação que representa um relacionamento todo-parte; sua notação é um losango vazio (sem cor) no final da conexão, anexado à classe agregadora



- Uma **composição** é uma forma mais forte de associação na qual o compositor tem responsabilidade exclusiva sobre gerenciar suas partes, assim como sua criação e destruição; sua notação é um losango preenchido no final da conexão, anexado à classe compositora

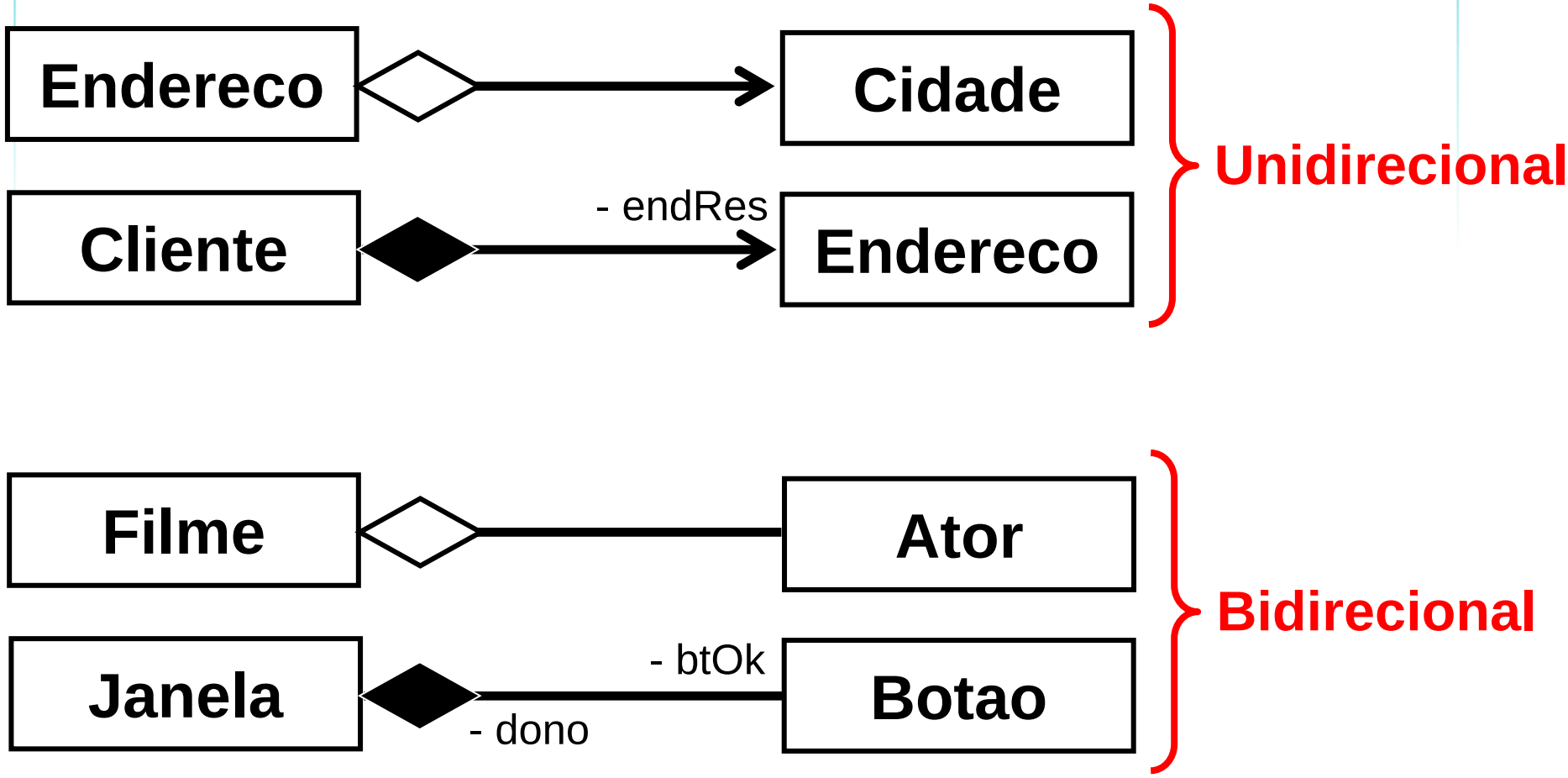


Agregação e Composição

- Na **agregação**, um objeto parte pode ser **compatilhado** (***shared***) por mais de um objeto todo (no exemplo anterior, uma pessoa pode pertencer a mais de um clube)
  - Sua aplicação é praticamente idêntica a de uma associação

- Na **composição**, um objeto parte é **exclusivo** de um objeto todo (***not shared***)
  - Quando o objeto todo é destruído, todas as partes são também destruídas
  - Não há necessidade de explicitar a multiplicidade no lado do compositor, pois o valor será "**0..1**" ou "**1**"

Agregação e Composição: Navegação



Quando usar agregação e composição?

- O relacionamento é descrito com uma frase “**parte de**”:
  - Um botão é “parte de” uma janela
- Algumas operações no todo são automaticamente aplicadas a suas partes?
  - Mover a janela, mover o botão
- Alguns valores de atributos são propagados do todo para todos ou algumas de suas partes?
  - A fonte da janela é Arial, a fonte do botão é Arial
- Existe uma assimetria inerente no relacionamento onde uma classe é subordinada a outra?
  - Um botão É parte de uma janela, uma janela NÃO É parte de um botão

Associação ou agregação/composição?

- **Agregação/Composição**: se dois objetos são altamente acoplados por um relacionamento todo-parte
- **Associação**: se dois objetos são usualmente considerados como independentes, mesmo eles estejam frequentemente ligados

Implementação da composição

```
class Endereco:

    def __init__(self, rua: str = "", complemento: str = "", bairro: str = "", cidade: str = "", cep: str = ""):
        self.rua = rua
        self.complemento = complemento
        self.bairro = bairro
        self.cidade = cidade
        self.cep = cep


class Cliente:

    def __init__(self):
        self.__enderecos = []
        ...

    def add_endereco(self, rua: str = "", complemento: str = "", bairro: str = "", cidade: str = "", cep: str = ""):
        novo_endereco = Endereco(rua, complemento, bairro, cidade, cep)
        self.__enderecos.append(novo_endereco)

    ...
```

Implementação da composição

```
class Endereco:

    def __init__(self, rua: str = "", complemento: str = "", bairro: str = "", cidade: str = "", cep: str = ""):
        self.rua = rua
        self.complemento = complemento
        self.bairro = bairro
        self.cidade = cidade
        self.cep = cep


class Cliente:

    def __init__(self):
        self.__enderecos = []
        ...

    def add(self, rua: str = "", complemento: str = "", bairro: str = "", cidade: str = "", cep: str = ""):
        novo_endereco = Endereco(rua, complemento, bairro, cidade, cep)
        self.__enderecos.append(novo_endereco)

    ...
```

Não está adicionando um endereço!

Implementação da composição

```
class Endereco:

    def __init__(self, rua: str = "", complemento: str = "", bairro: str = "", cidade: str = "", cep: str = ""):
        self.rua = rua
        self.complemento = complemento
        self.bairro = bairro
        self.cidade = cidade
        self.cep = cep


class Cliente:

    def __init__(self):
        self.__enderecos = []
        ...

    def add_endereco(self, rua: str = "", complemento: str = "", bairro: str = "", cidade: str = "", cep: str = ""):
        novo_endereco = Endereco(rua, complemento, bairro, cidade, cep)
        self.__enderecos.append(novo_endereco)

    ...
```

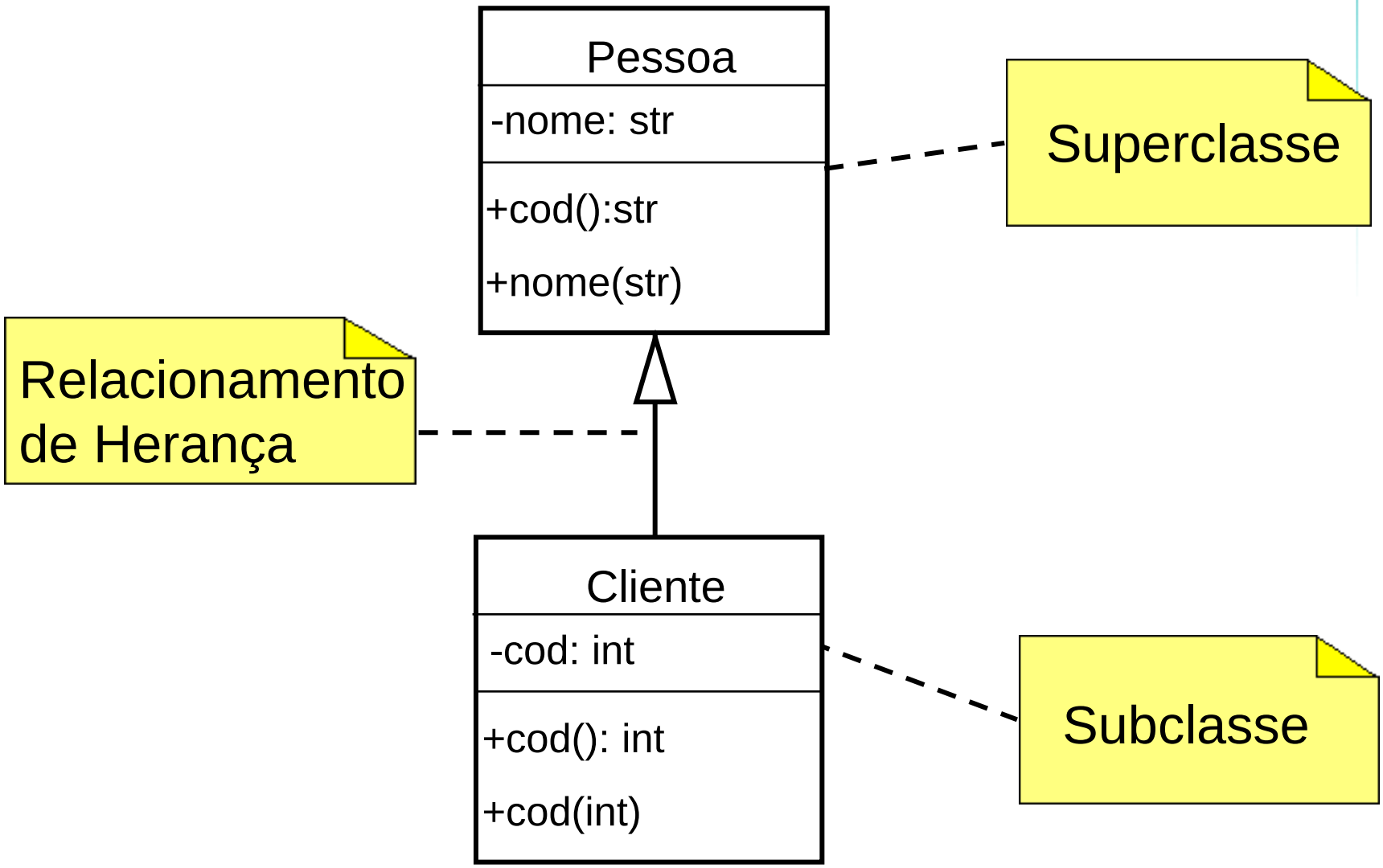
O Endereço é criado dentro do método e pertence unicamente a este objeto da classe Cliente

Mais detalhes sobre: GENERALIZAÇÃO

Generalização (herança)

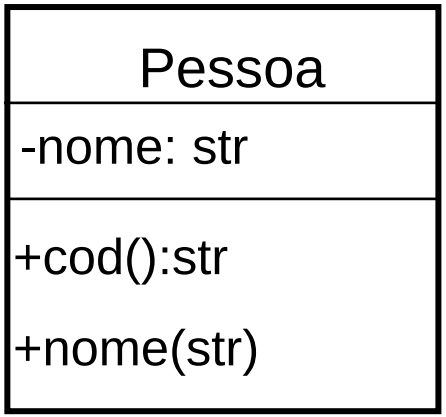
- Mecanismo que permite a reutilização daquilo que já foi implementado
- Define um relacionamento entre classes, onde verifica-se aquilo que é comum entre determinadas classes
- Uma classe compartilha a estrutura e/ou comportamento de uma ou mais classes
- É um relacionamento de especialização/ generalização (“**é um**” ou “**tipo de**”)

Representação da generalização



Superclasse Pessoa em Python

```python
class Pessoa:

    def __init__(self, nome: str):
        self.__nome = nome

    @property
    def nome(self):
        return self.__nome

    @nome.setter
    def nome(self, nome):
        self.__nome = nome
```



Subclasse Cliente em Python

```
class Cliente(Pessoa):

    def __init__(self, cod: int, nome: str):
        super().__init__(nome)
        self.__cod = cod

    @property
    def cod(self):
        return self.__cod

    @cod.setter
    def cod(self, cod):
        self.__cod = cod
```

Cliente
-cod: int
+cod(): int +cod(int)

Subclasse Cliente em Python

```
class Cliente(Pessoa):

    def __init__(self,
        super().__init__(
        self.__cod = cod

    @property
    def cod(self):
        return self.__cod

    @cod.setter
    def cod(self, cod):
        self.__cod = cod
```

Indica a herança

A classe Cliente especializa (herda) a classe Pessoa

A classe Pessoa é a superclasse

Subclasse Cliente em Python

```
class Cliente(Pessoa):

    def __init__(self, co
        super().__init__(nom
        self.__cod = cod

    @property
    def cod(self):
        return self.__cod

    @cod.setter
    def cod(self, cod):
        self.__cod = cod
```

Pode ser necessário ter uma cláusula “import” para o pacote onde a classe Pessoa está implementada

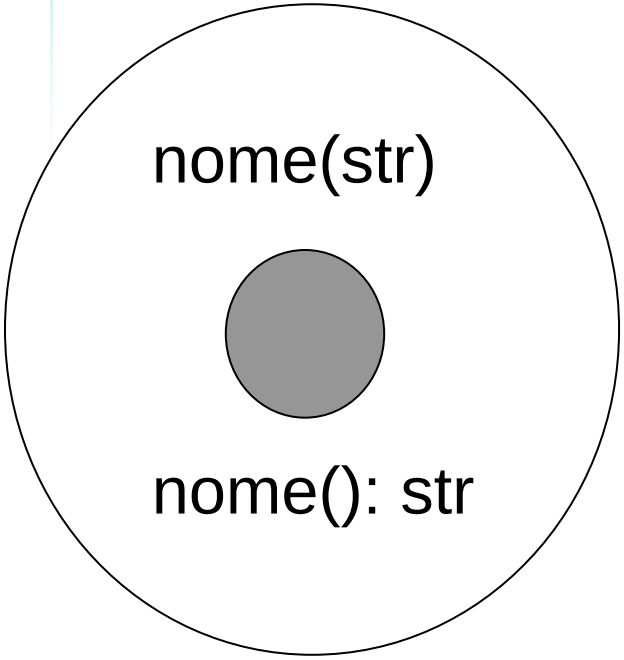
Subclasse Cliente em Python

```
class Cliente(Pessoa):

    def __init__(self, cod: int, nome: str):
        super().__init__(nome)
        self.__cod = cod

    @property
    def cod(self):
        return self.__cod

    @cod.setter
    def cod(self, cod):
        self.__cod = cod
```

Repassa os valores dos atributos que pertencem à superclasse

Generalização: objeto da superclasse

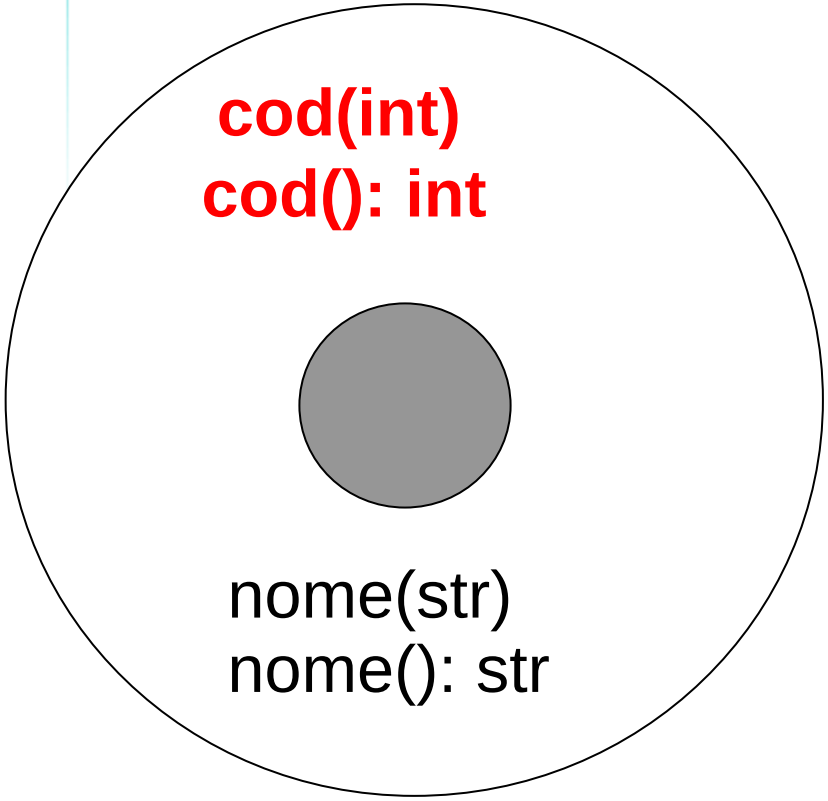
alguem: Pessoa




```
alguem = Pessoa("Jean")
...
alguem.nome = "Jean Hauck"
...
print(alguem.nome)
```

Generalização: objeto da subclasse

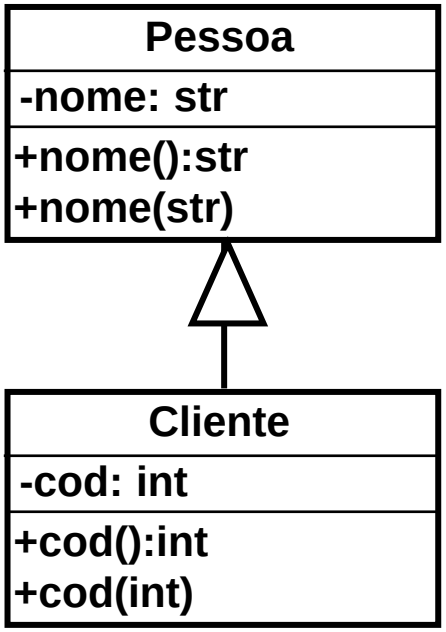
cliente: Cliente




```
cliente = Cliente(1, "Jean")
...
cliente.cod = 123
cliente.nome = "Jean Hauck"
...
print(cliente.cod)
print(cliente.nome)
```

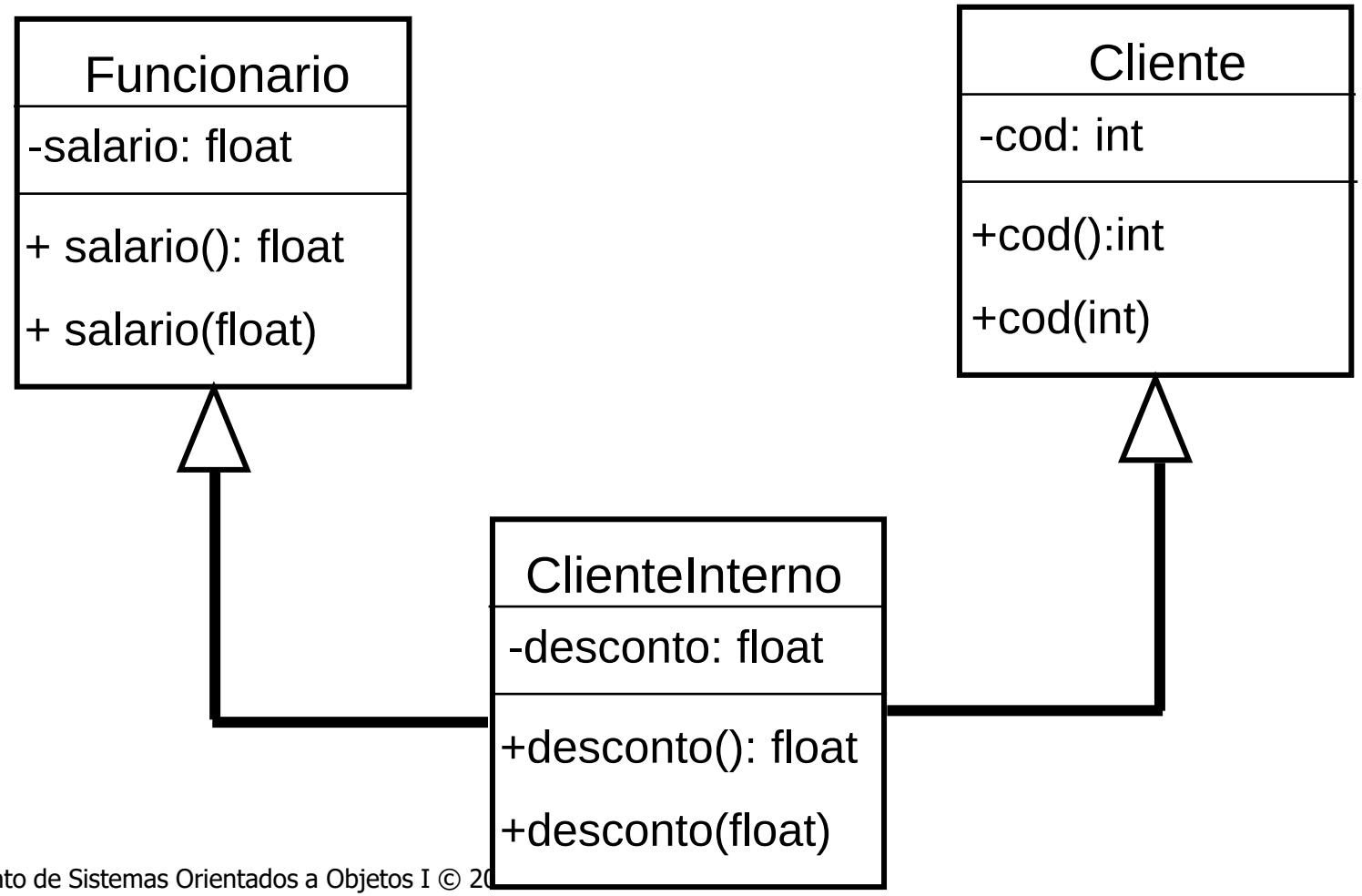
Tipos de herança

- Herança define uma hierarquia de abstrações na qual uma subclasse herda de uma ou mais superclasses:
  - **Herança simples**: a subclasse herda de uma única superclasse



Herança múltipla

- **Herança múltipla**: a subclasse herda de duas ou mais superclasses

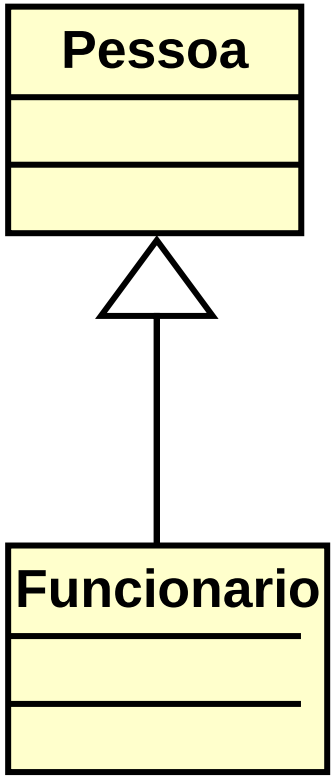


Modularidade e Herança

Qual é o sentido da dependência?

Qual é a classe mais independente?

Qual é a classe com maior reusabilidade?



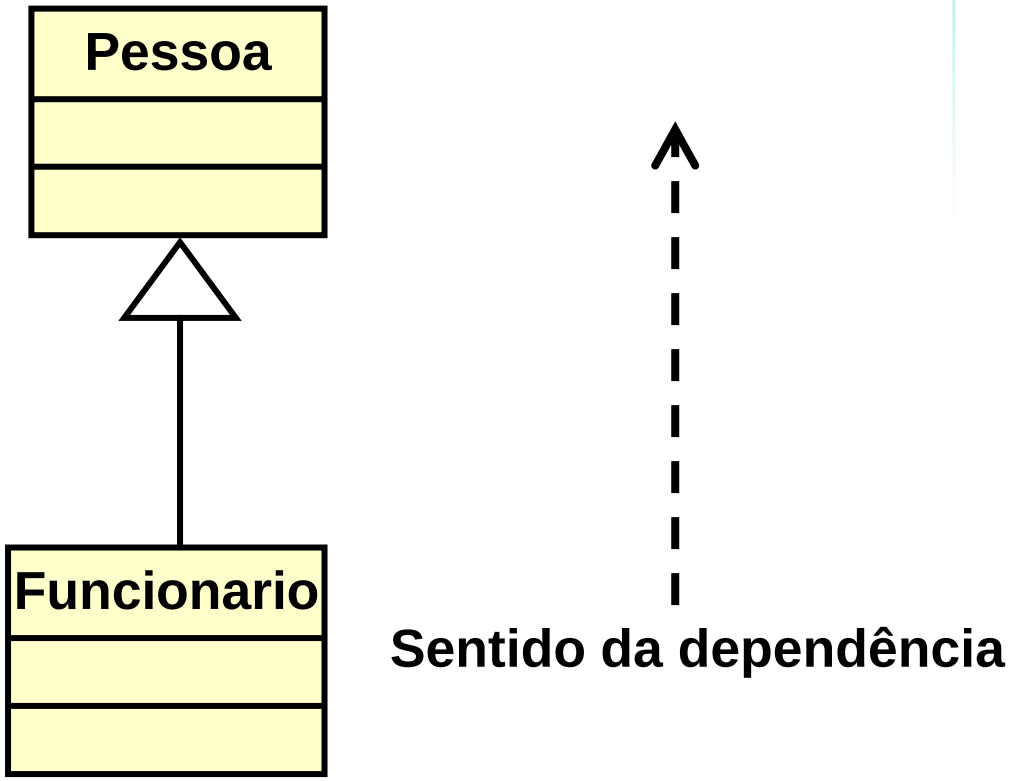
Modularidade e Herança

Maior reusabilidade, Mais genérico
Maior abstração

Qual é o sentido da dependência?

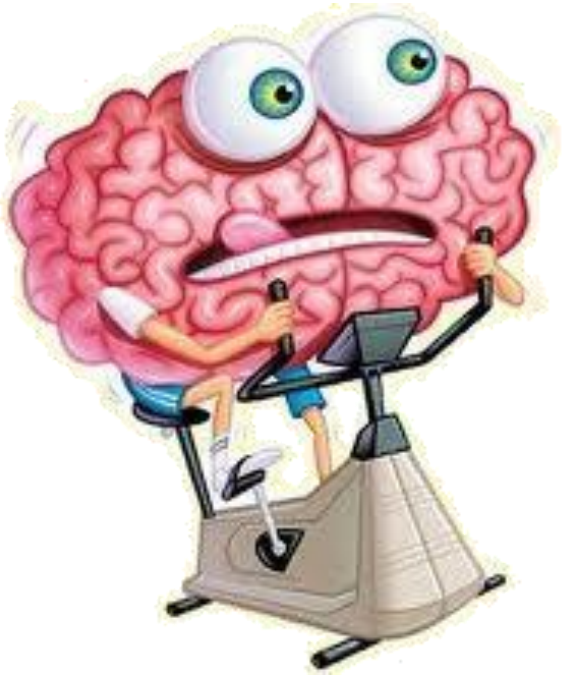
Qual é a classe mais independente?

Qual é a classe com maior reusabilidade?



Menor reusabilidade, Mais específico,
Menor abstração

Agora vamos exercitar ...



Implemente os exercícios no Moodle!

Referências

THIRY, M. Apresentações de aula. Univali, 2014.

ALCHIN, Marty. Pro Python. New York: Apress, 2010. Disponível em: <https://link.springer.com/book/10.1007%2F978-1-4302-2758-8#about>

HALL, Tim; STACEY, J. P. Python 3 for absolute beginners. Apress, 2010. Disponível em: <https://link.springer.com/book/10.1007%2F978-1-4302-1633-9>

BOOCH, G., Object-Oriented Design. Benjamin/Cumminggs Pub. 1998.

WAZLAWICK, Raul S. Introdução a Algoritmos e Programação com Python. São Paulo: Elsevier, 2017.

WAZLAWICK, Raul S. Análise e Projeto de Sistemas de Informação Orientados a Objetos. São Paulo: Campus. 2004.

Agradecimento

Agradecimento ao prof. Marcello Thiry pelo material cedido.

marcello.thiry@gmail.com

Licença

creative commons
COMMONS DEED

Atribuição-Uso-Não-Comercial-Compartilhamento pela Licença 2.5 Brasil

Você pode:
- copiar, distribuir, exibir e executar a obra
- criar obras derivadas

Sob as seguintes condições:
Atribuição — Você deve dar crédito ao autor original, da forma especificada pelo autor ou licenciante.
Uso Não-Comercial — Você não pode utilizar esta obra com finalidades comerciais.
Compartilhamento pela mesma Licença — Se você alterar, transformar, ou criar outra obra com base nesta, você somente poderá distribuir a obra resultante sob uma licença idêntica a esta.

Para ver uma cópia desta licença, visite http://creativecommons.org/licenses/by-nc-sa/2.5/br/ ou mande uma carta para Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.