

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

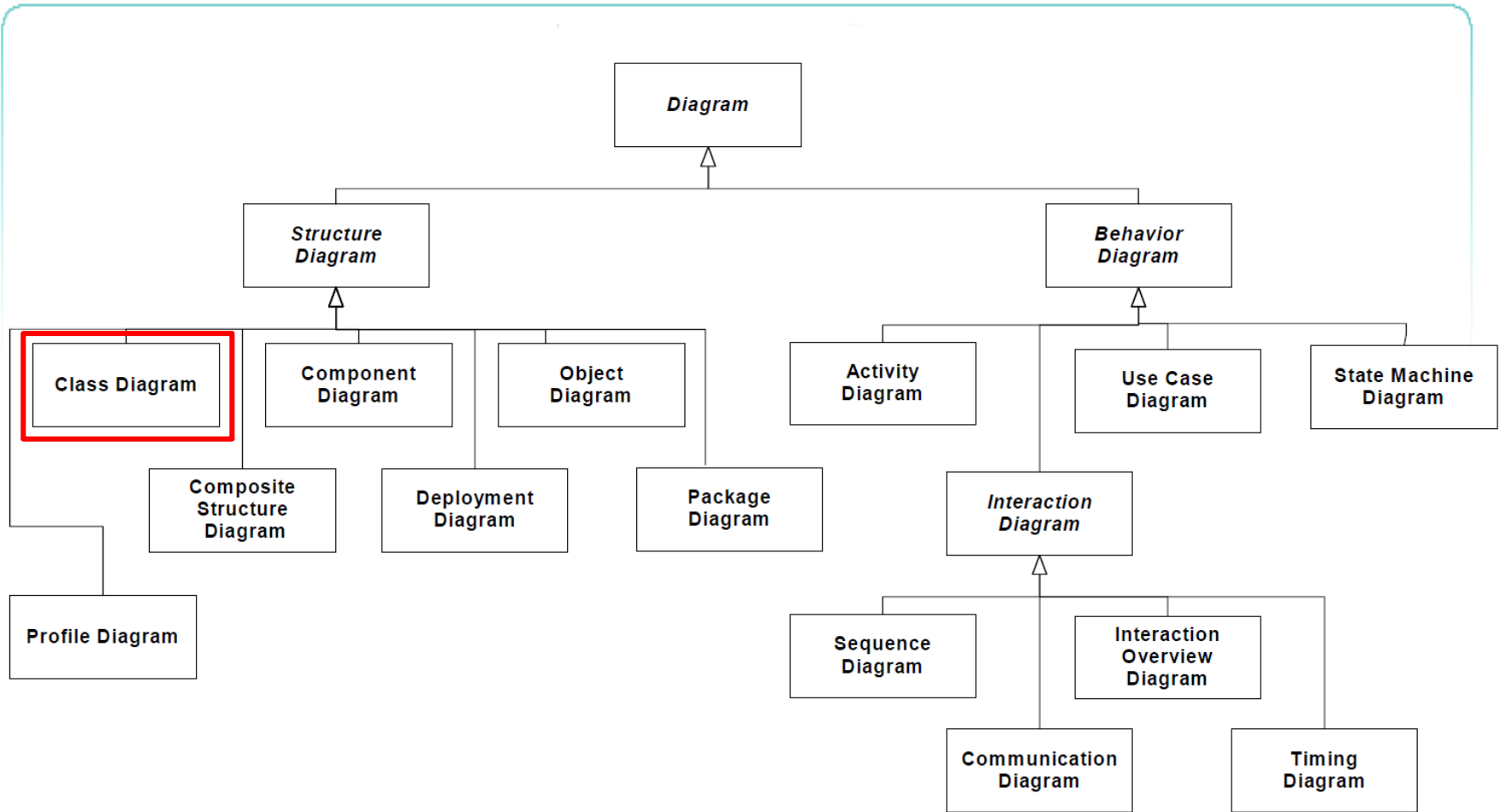
UML (*Unified Modeling Language*)

- 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/>]

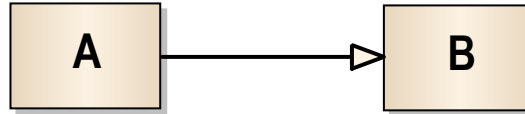
Visão geral da notação UML



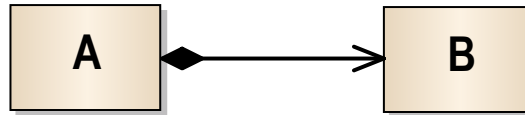
[OMG, 2015]

Acoplamento entre Classes

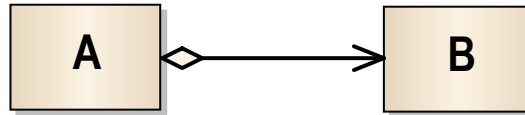
Generalização:



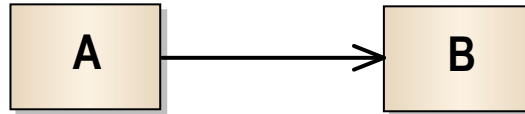
Composição:



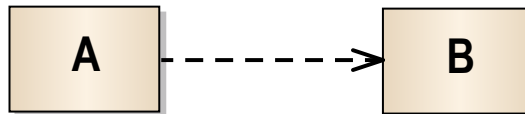
Agregação:



Associação:



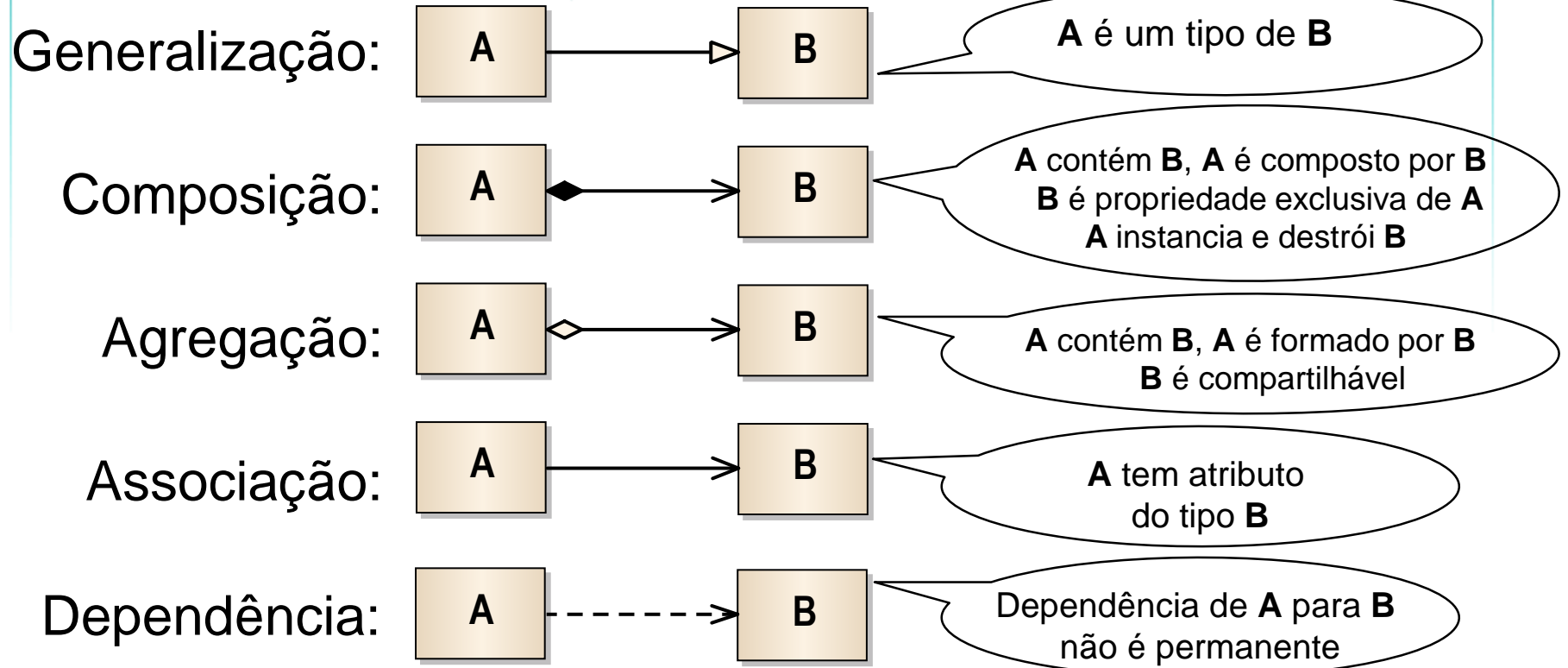
Dependência:



Grau de Acoplamento



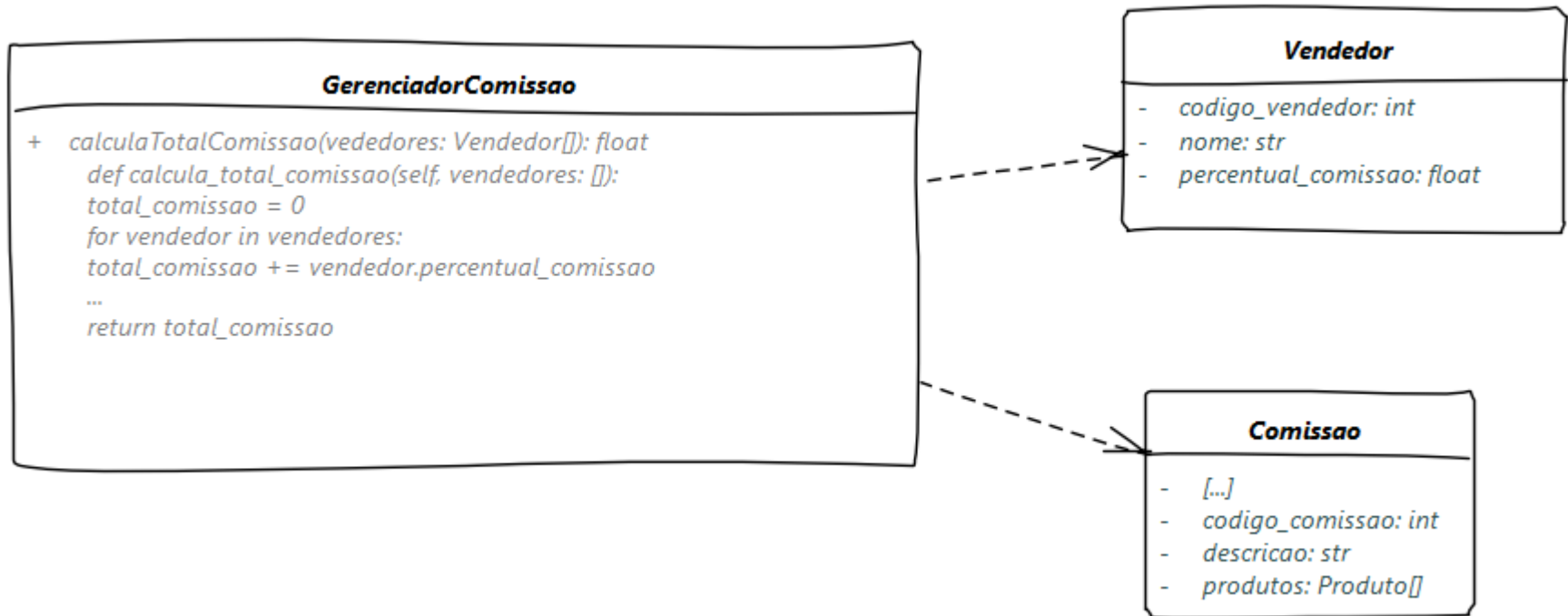
RESUMINDO



Principais relacionamentos entre Classes

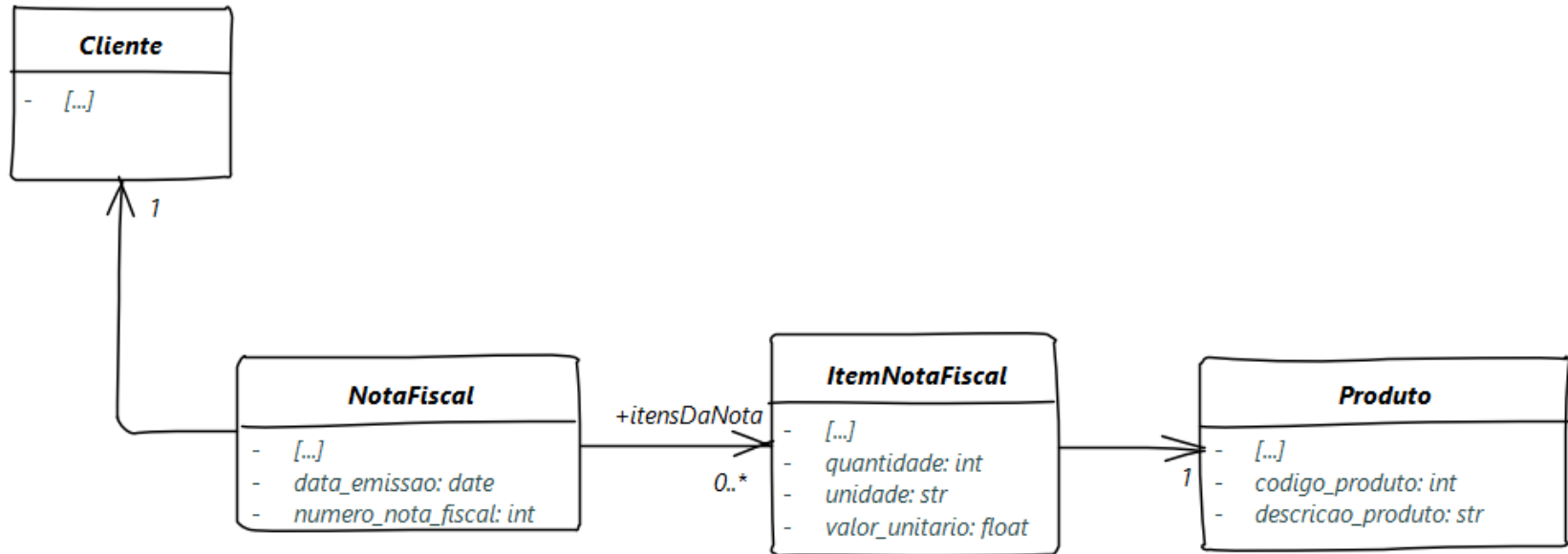
- **Generalização** (herança): um dos princípios da OO, permite a reutilização, uma nova classe pode ser definida a partir de outra já existente
- **Agregação e Composição**: especializações de uma associação, onde um objeto todo é relacionado com suas partes (relacionamento “parte-de” ou “contenção”)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

Dependência



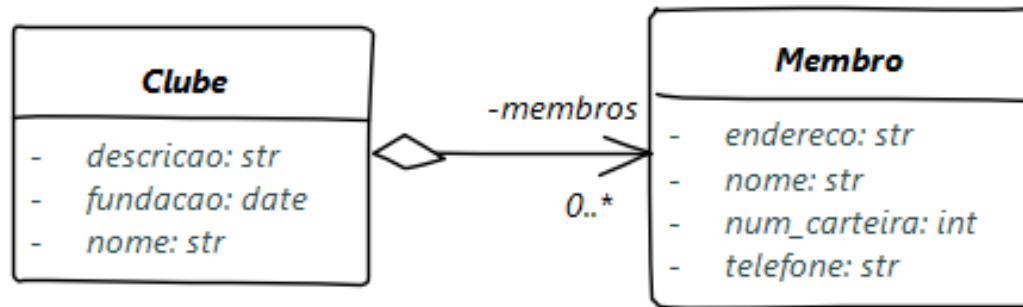
Exemplos

Associação



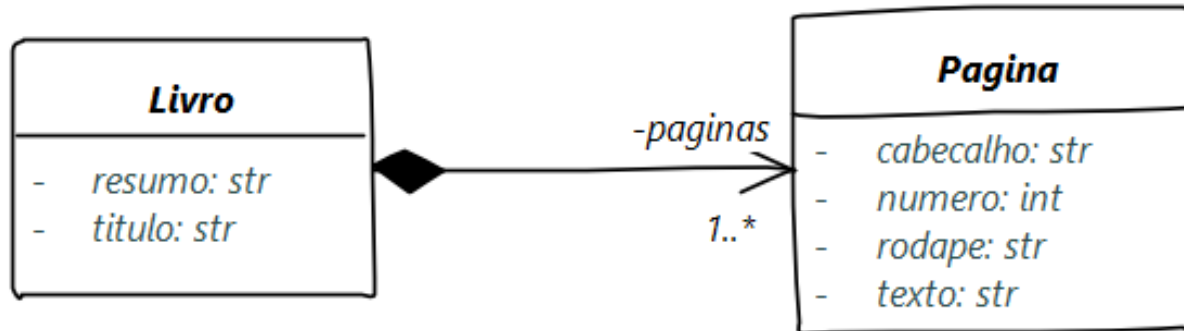
Exemplos

Agregação



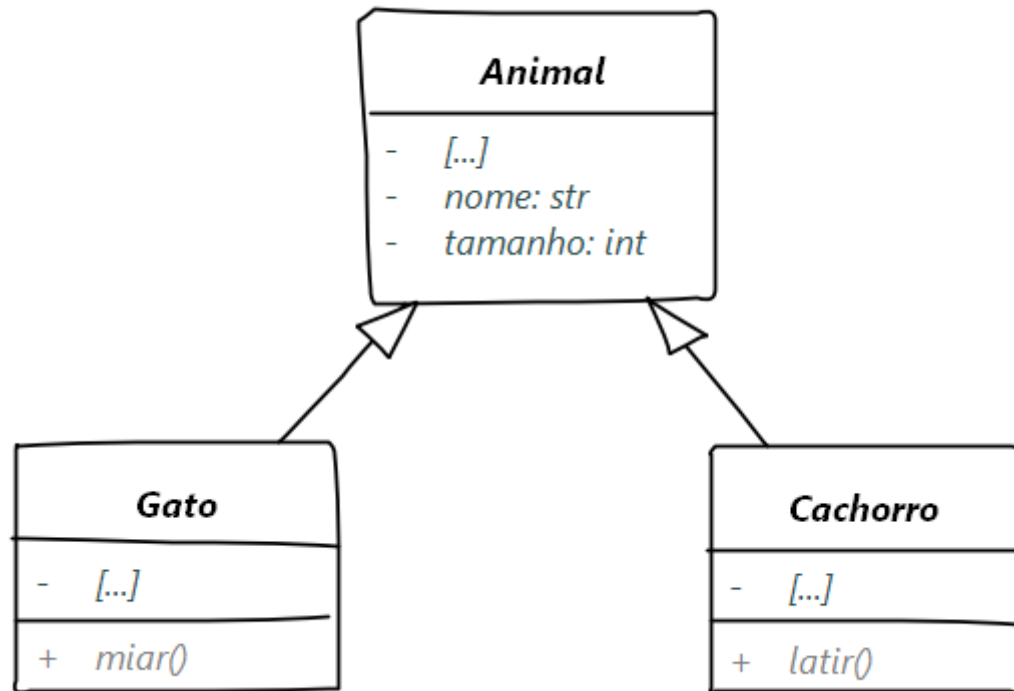
Exemplos

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

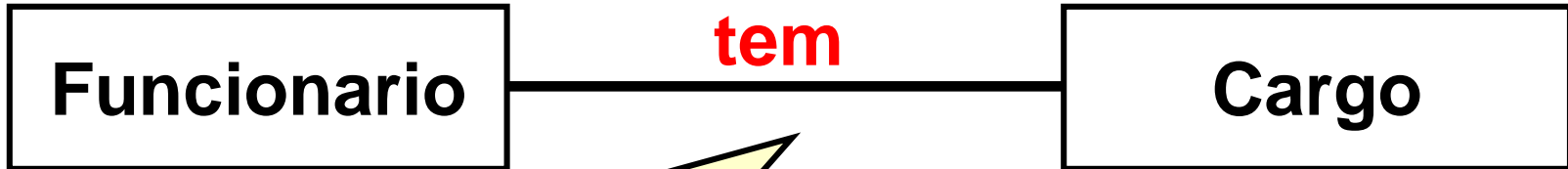
- Associações representam relações entre objetos
- Cada associação representa de
- Cada ponto da associação representa a uma
- Os dados da associação são atribuídos em ambas as direções da associação

Mas qual é o significado da associação entre estas classes?

Funcionario

Cargo

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



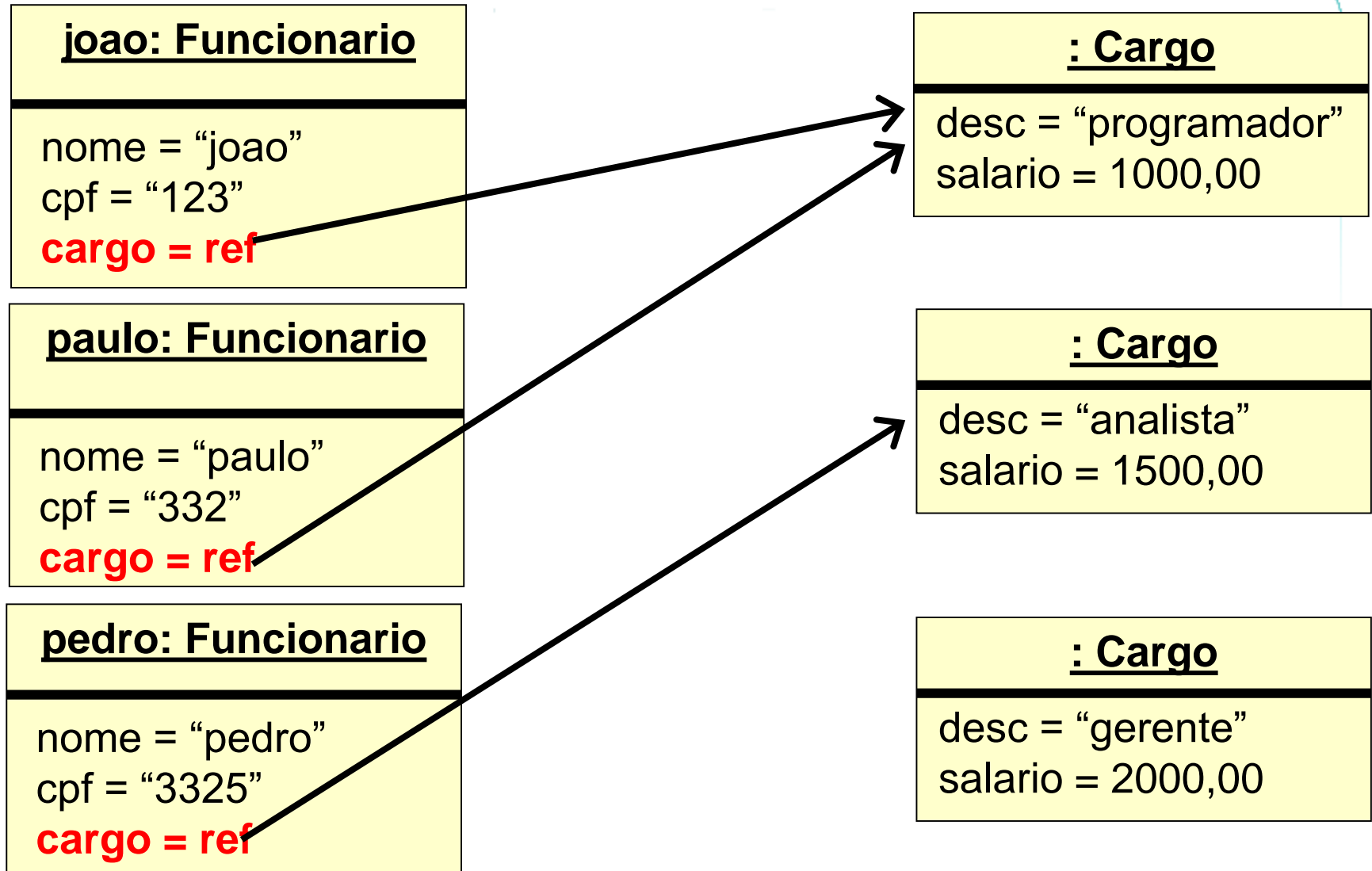
- Um objeto Funcionario **aponta** para 1 objeto Cargo
- Um objeto Cargo **pode ser apontado por** vários objetos Funcionario

```
class Funcionario:

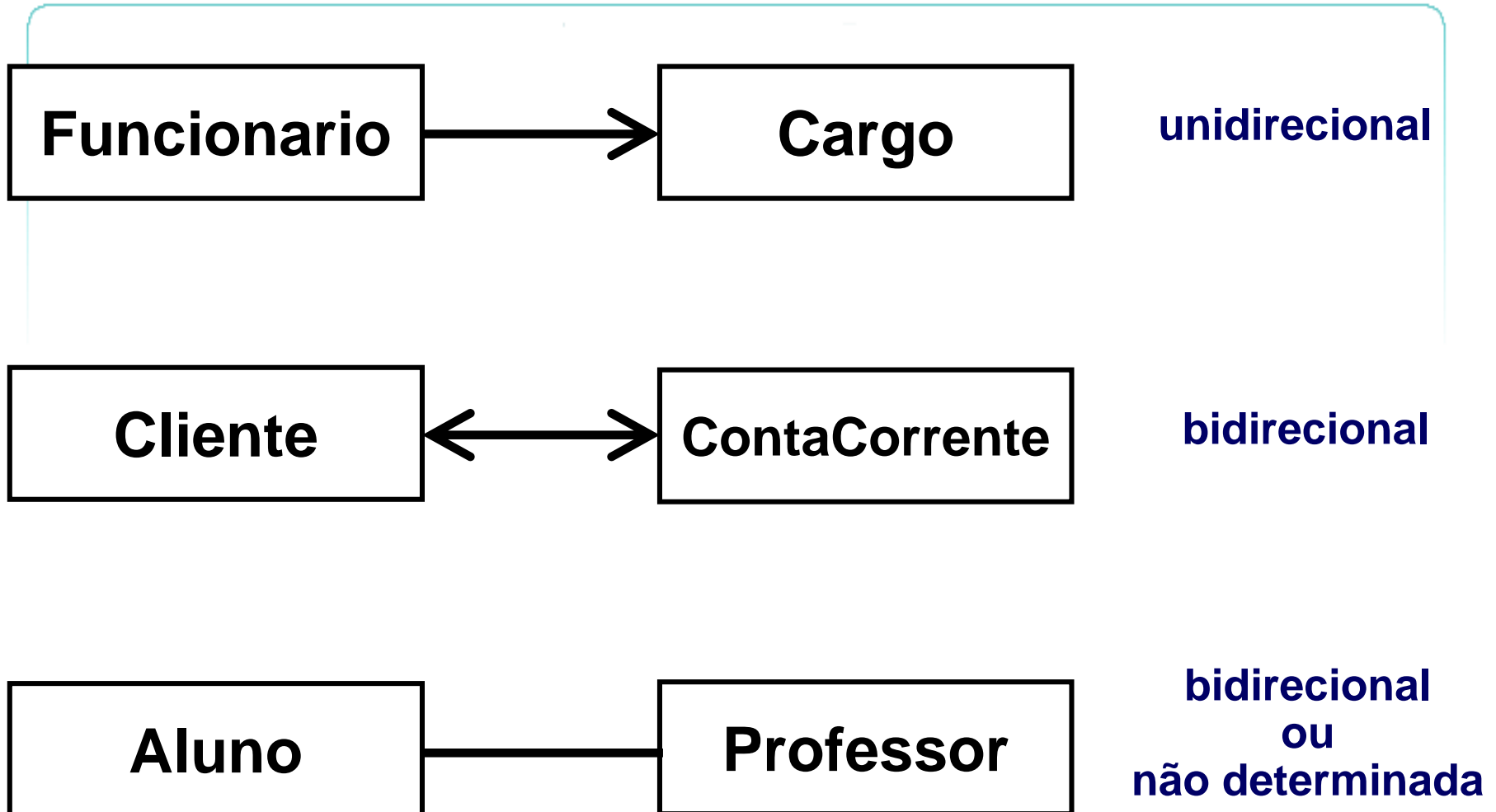
    def __init__(self, cargo: Cargo):
        self.__cargo = cargo
```

```
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**



Nome de Associação é
pouco utilizado

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

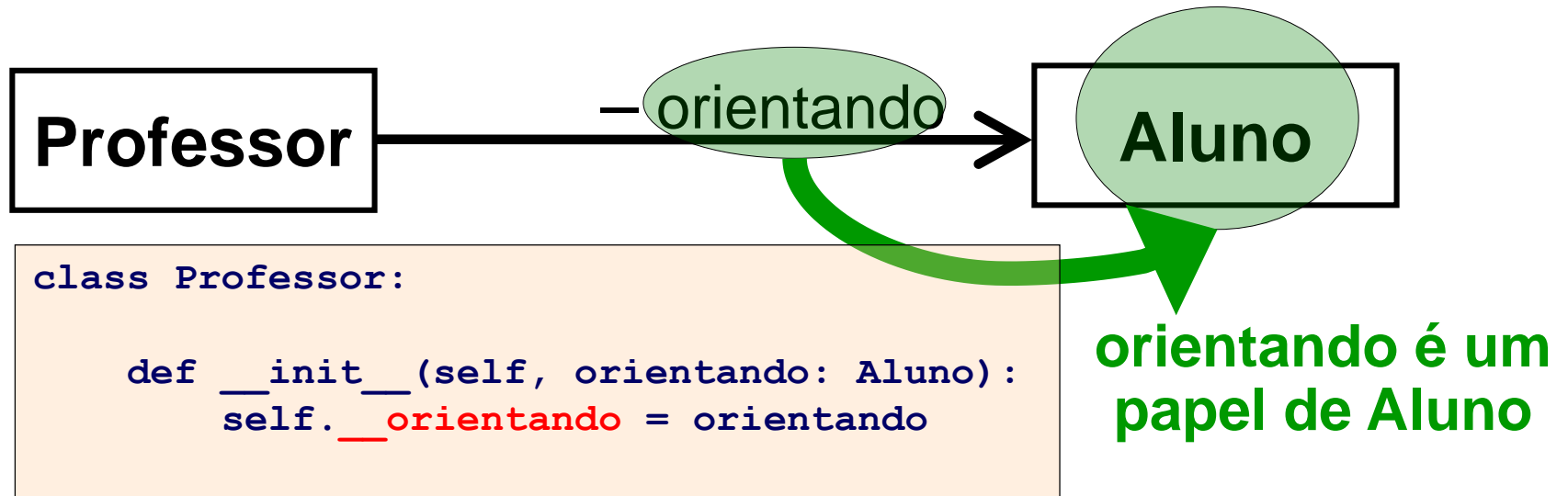


```
class Professor:

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

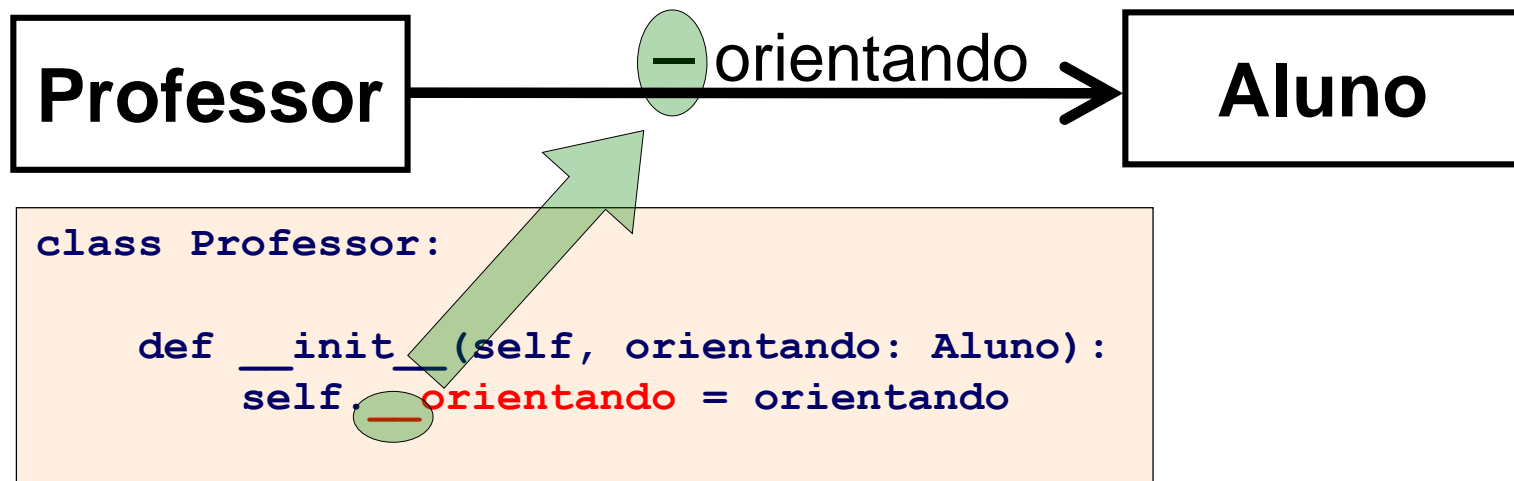
Definindo papéis (*rolenames*)

- 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



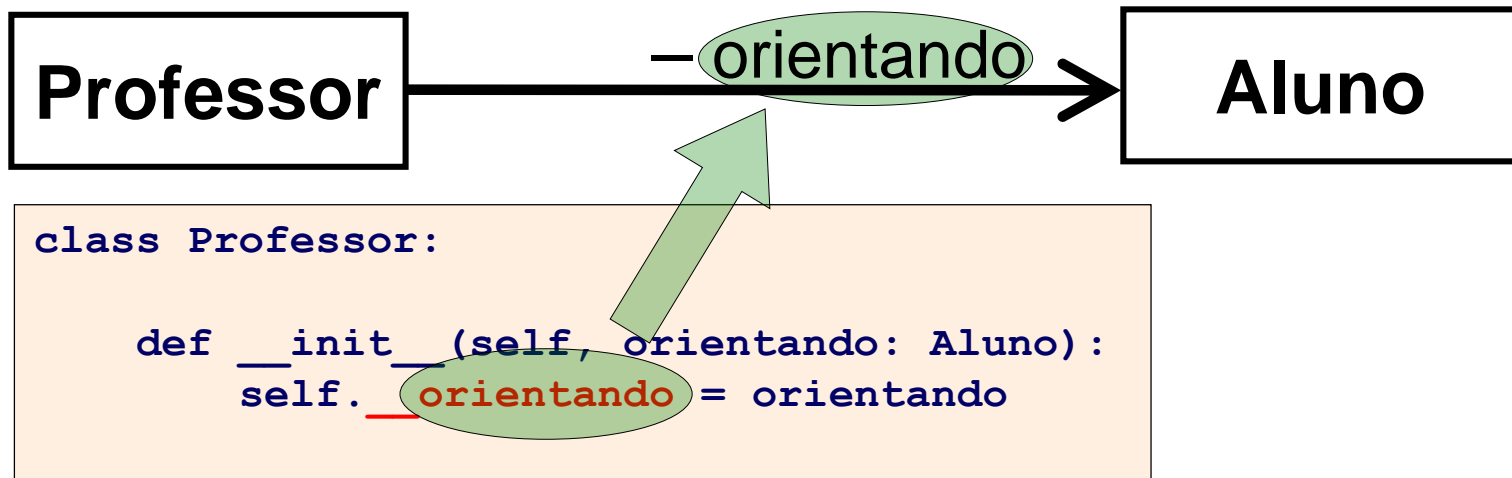
Definindo papéis (*rolenames*)

- 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



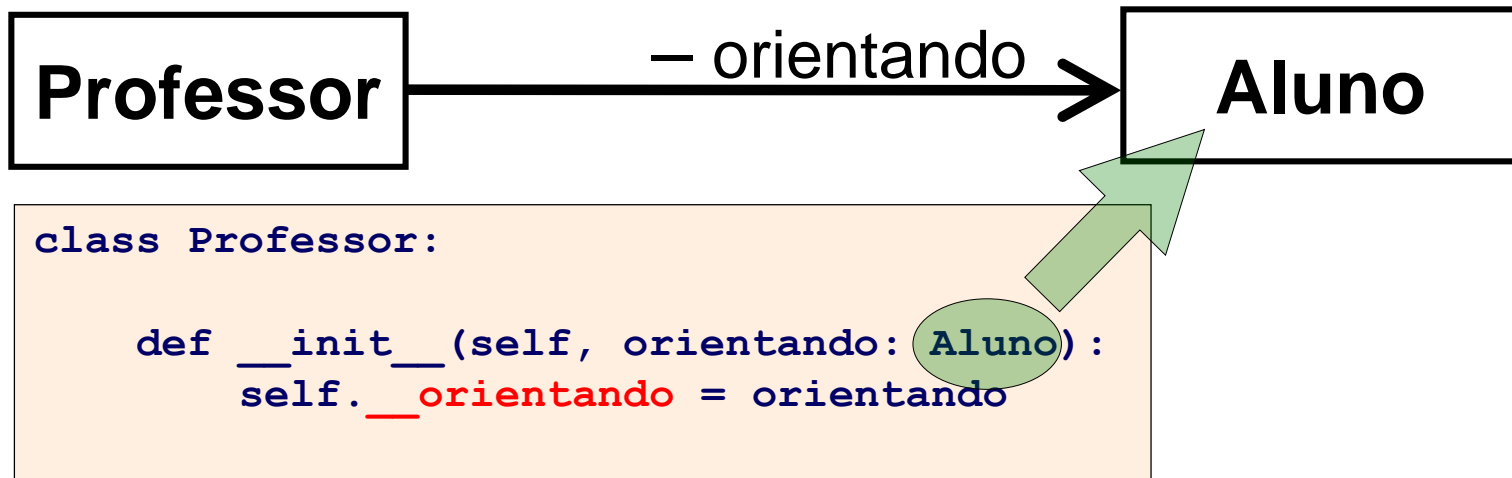
Definindo papéis (*rolenames*)

- 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



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



Definindo papéis (*rolenames*)

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

- Um **Professor** tem uma associação da classe **Aluno**. O **Professor** está apontando

**Python não obriga
parâmetro a ser
do tipo Aluno**

```
class Professor:
```

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

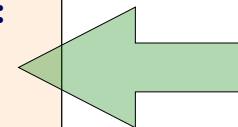
Definindo papéis (*rolenames*)

- 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**

orientando

Aluno

```
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

Professor

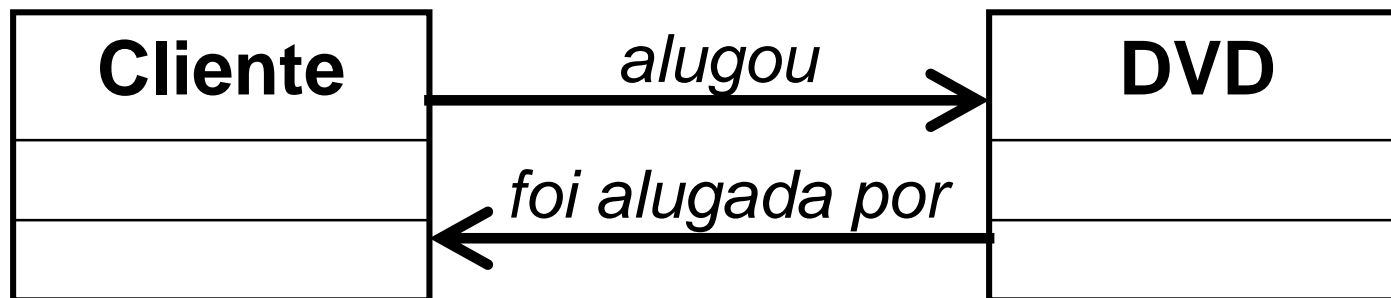
Esta abordagem será discutida quando tratarmos do princípio: **EAFP** (*Easier to ask for forgiveness than permission*)*

```
class Professor:
```

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

Múltiplas associações

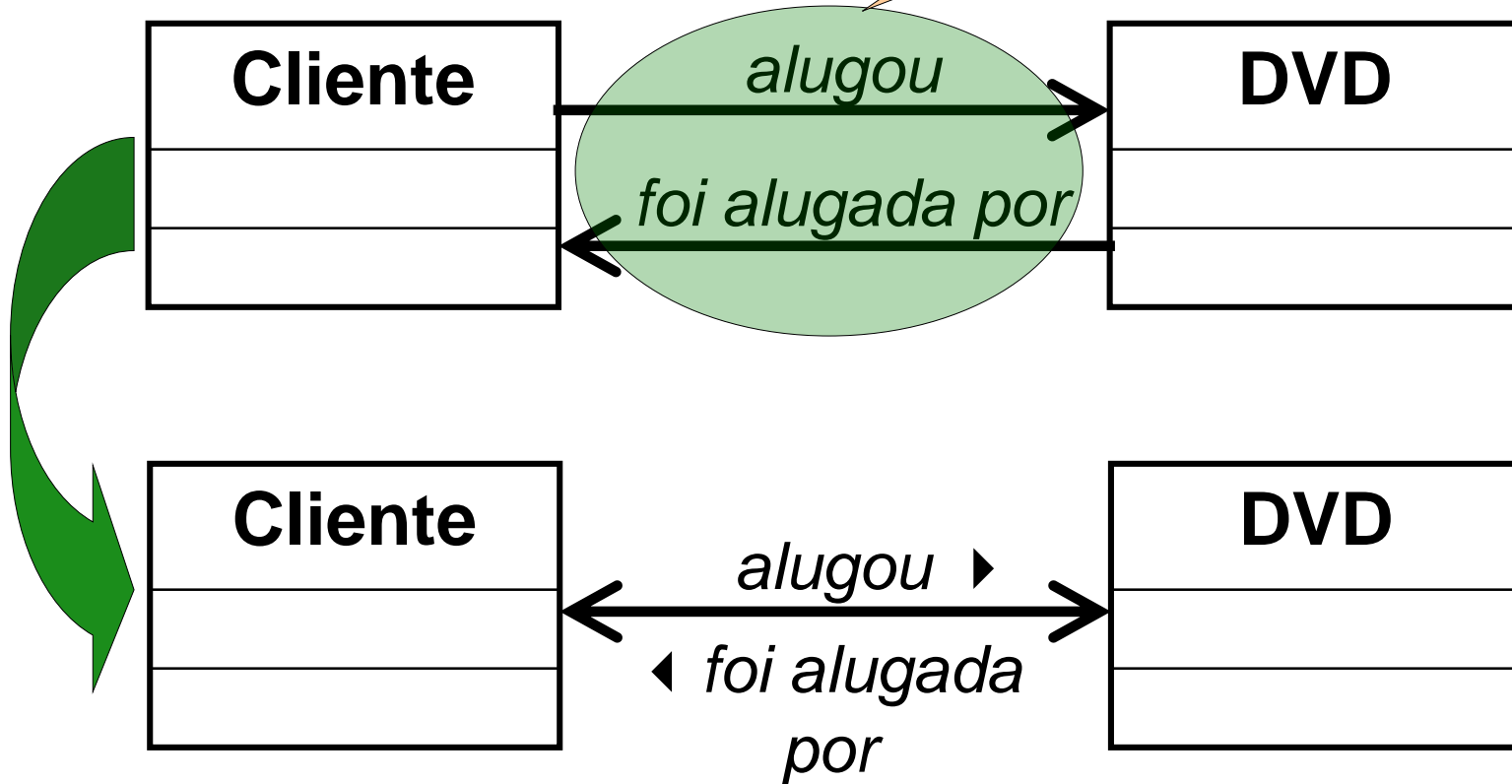
- Podem existir várias associações entre duas classes
- Se há mais que uma associação entre duas classes, então elas precisam ser nomeadas
- **Cuidado**: não mapear por mensagens



São duas associações diferentes?

Entendendo a semântica da associação

Errado!



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 DVD
 - Para cada instância de DVD, 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



Associações bidirecionais

- Como traduzir uma associação vários para vários?



Associações bidirecionais

- Como traduzir uma associação **vários para vários**?



```
class Cliente:

    def __init__(self):
        self.__contas = []
```

```
class ContaCorrente:

    def __init__(self):
        self.__clientes = []
```

Associações bidirecionais

- Como traduzir uma associação **vários para vários**?



```
class Cliente:

    def __init__(self):
        self.__contas = []
```

```
class ContaCorrente:

    def __init__(self):
        self.__clientes = []
```

Quais as dificuldades decorrentes desta implementação?

Associações bidirecionais

- Como traduzir uma associação vários para vários?

Cliente

Fornecedor

CONSISTÊNCIA!

```
class Cliente
```

```
def __init__(self):  
    self.__con = []
```

```
def __init__(self):  
    self.__clientes = []
```

**Quais as dificuldades decorrentes
desta implementação?**

Associações bidirecionais

- Como traduzir uma associação vários para vários?

Como garantir que a lista de contas do cliente está **consistente** com a lista de clientes da conta?

*

ContaCorrente

```
class Cliente:  
  
    def __init__(self):  
        self.__contas = []
```

```
class ContaCorrente:  
  
    def __init__(self):  
        self.__clientes = []
```

Quais as dificuldades decorrentes desta implementação?

Associações bidirecionais

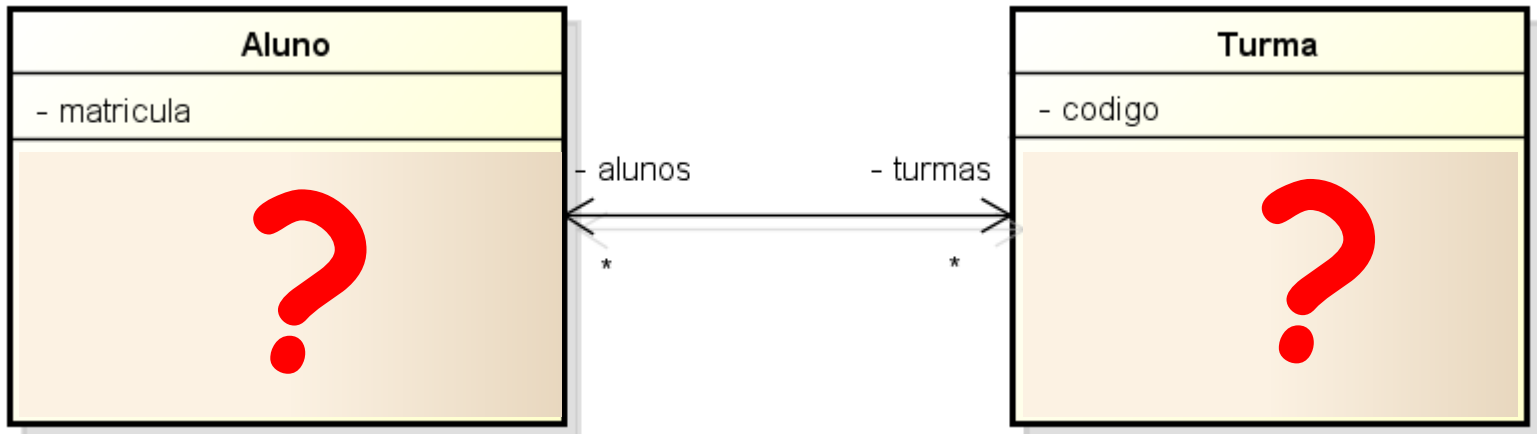
Para considerar:

❑ Associações bidirecionais...

- ❑ ... aumentam o **acoplamento** (dependência entre classes), reduzindo a reusabilidade
- ❑ ... aumentam a **complexidade** da implementação, pois exigem que o sincronismo seja mantido nos dois lados da associação
- ❑ ... quando definidas como **vários para vários**, aumentam ainda mais a complexidade da implementação

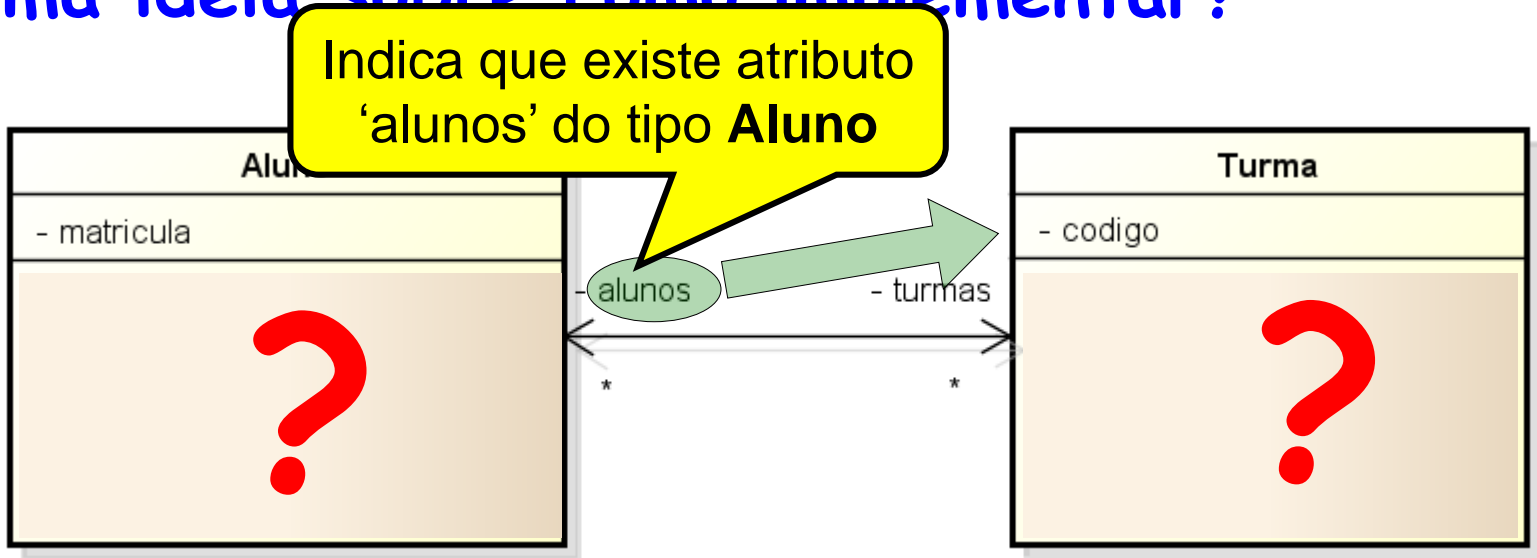
Associações bidirecionais

Alguma ideia sobre como implementar?



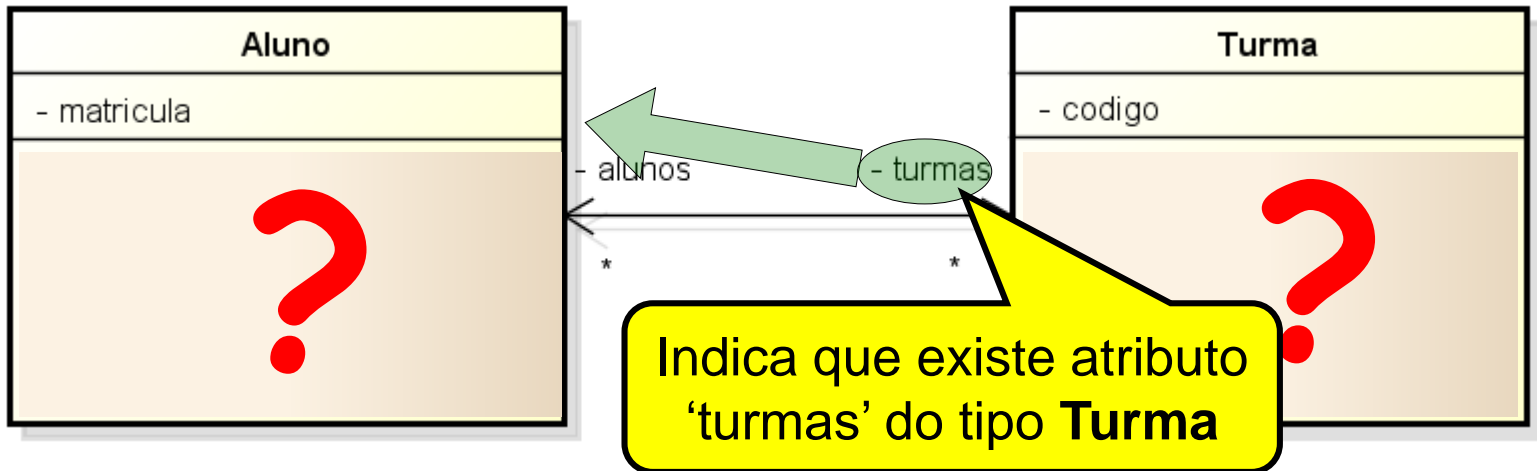
Associações bidirecionais

Alguma ideia sobre como implementar?



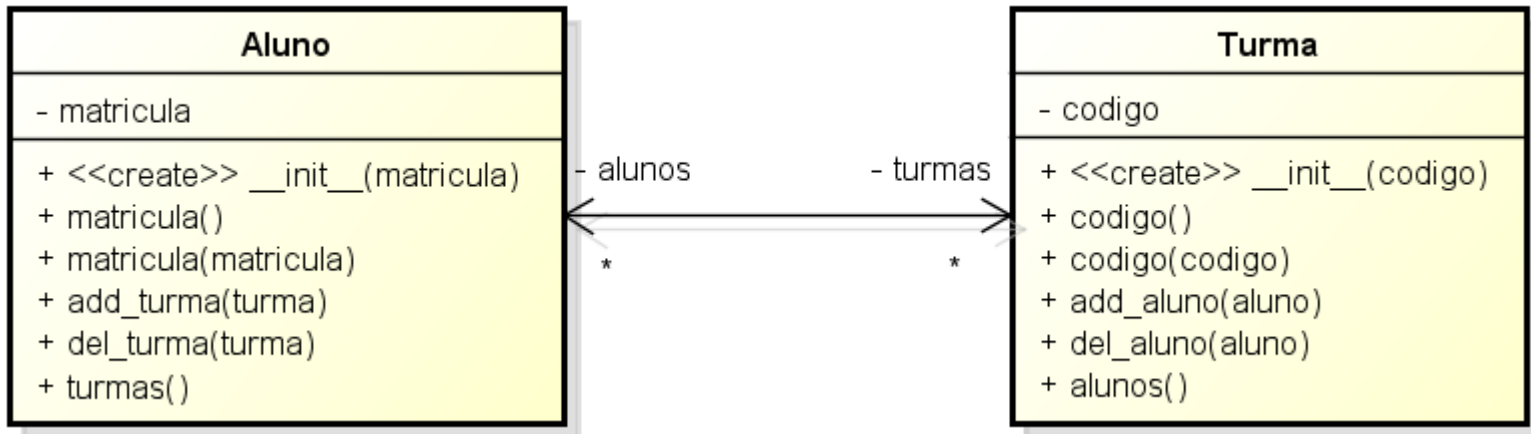
Associações bidirecionais

Alguma ideia sobre como implementar?



Associações bidirecionais

E agora?



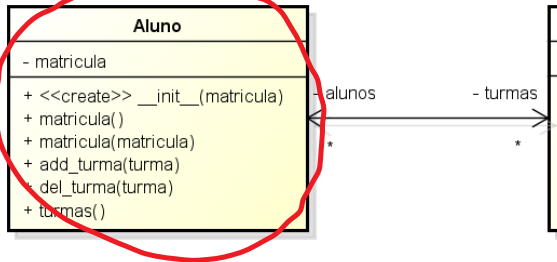
Associações bidirecionais

E agora?



CONSISTÊNCIA!

Associações bidirecionais



```
class Aluno:
```

```
def __init__(self, matricula: str):
    self.__matricula = matricula
    self.__turmas = []
```

```
@property
```

```
def matricula(self):
    return self.__matricula
```

```
@matricula.setter
```

```
def matricula(self, matricula: str):
    self.__matricula = matricula
```

```
def add_turma(self, turma: Turma):
```

```
    if (turma is not None) and (isinstance(turma, Turma)):
        if turma not in self.__turmas:
            self.__turmas.append(turma)
        if self not in turma.alunos:
            turma.alunos.append(self)
```

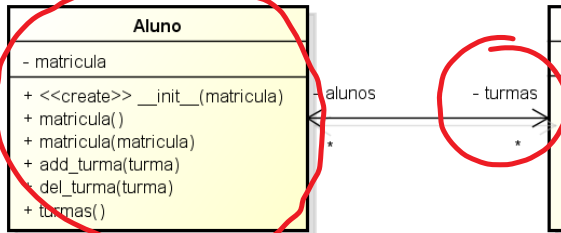
```
def del_turma(self, turma: Turma):
```

```
    if (turma is not None) and (isinstance(turma, Turma)):
        if turma not in self.__turmas:
            self.__turmas.remove(turma)
        if self not in turma.alunos:
            turma.alunos.remove(self)
```

```
@property
```

```
def turmas(self):
    return self.__turmas
```

Associações bidirecionais



```
class Aluno:
```

```
def __init__(self, matricula: str):
    self.__matricula = matricula
    self.__turmas = []
```

```
@property
```

```
def matricula(self):
    return self.__matricula
```

```
@matricula.setter
```

```
def matricula(self, matricula: str):
    self.__matricula = matricula
```

```
def add_turma(self, turma: Turma):
```

```
    if (turma is not None) and (isinstance(turma, Turma)):
        if turma not in self.__turmas:
            self.__turmas.append(turma)
        if self not in turma.alunos:
            turma.alunos.append(self)
```

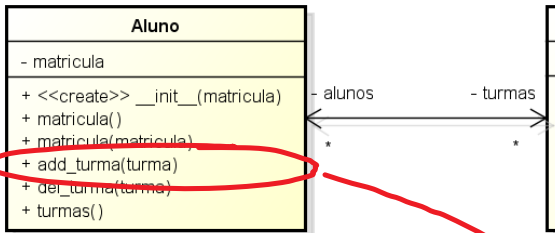
```
def del_turma(self, turma: Turma):
```

```
    if (turma is not None) and (isinstance(turma, Turma)):
        if turma not in self.__turmas:
            self.__turmas.remove(turma)
        if self not in turma.alunos:
            turma.alunos.remove(self)
```

```
@property
```

```
def turmas(self):
    return self.__turmas
```


Associações bidirecionais



```
class Aluno:
```

```
def __init__(self, matricula: str):
    self.__matricula = matricula
    self.__turmas = []
```

```
@property
def matricula(self):
    return self.__matricula
```

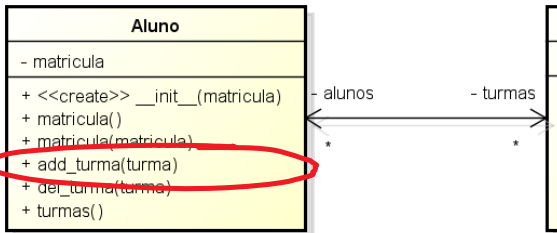
```
@matricula.setter
def matricula(self, matricula: str):
    self.__matricula = matricula
```

```
def add_turma(self, turma: Turma):
    if (turma is not None) and (isinstance(turma, Turma)):
        if turma not in self.__turmas:
            self.__turmas.append(turma)
        if self not in turma.alunos:
            turma.alunos.append(self)
```

```
def del_turma(self, turma: Turma):
    if (turma is not None) and (isinstance(turma, Turma)):
        if turma not in self.__turmas:
            self.__turmas.remove(turma)
        if self not in turma.alunos:
            turma.alunos.remove(self)
```

```
@property
def turmas(self):
    return self.__turmas
```

Associações bidirecionais



```

class Aluno:

    def __init__(self, matricula: str):
        self.__matricula = matricula
        self.__turmas = []

    @property
    def matricula(self):
        return self.__matricula

    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula

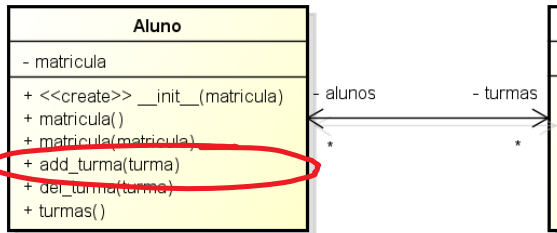
    def add_turma(self, turma: Turma):
        if (turma is not None) and (isinstance(turma, Turma)):
            if turma not in self.__turmas:
                self.__turmas.append(turma)
            if self not in turma.alunos:
                turma.alunos.append(self)

    def del_turma(self, turma: Turma):
        if (turma is not None) and (isinstance(turma, Turma)):
            if turma not in self.__turmas:
                self.__turmas.remove(turma)
            if self not in turma.alunos:
                turma.alunos.remove(self)

    @property
    def turmas(self):
        return self.__turmas
  
```

Validação da Classe

Associações bidirecionais



Verificar se aluno
já tem a turma

```

class Aluno:

    def __init__(self, matricula: str):
        self.__matricula = matricula
        self.__turmas = []

    @property
    def matricula(self):
        return self.__matricula

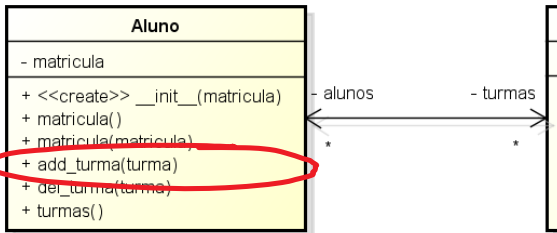
    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula

    def add_turma(self, turma: Turma):
        if (turma is not None) and (isinstance(turma, Turma)):
            if turma not in self.__turmas:
                self.__turmas.append(turma)
            if self not in turma.alunos:
                turma.alunos.append(self)

    def del_turma(self, turma: Turma):
        if (turma is not None) and (isinstance(turma, Turma)):
            if turma not in self.__turmas:
                self.__turmas.remove(turma)
            if self not in turma.alunos:
                turma.alunos.remove(self)

    @property
    def turmas(self):
        return self.__turmas
  
```

Associações bidirecionais



Garantir a
CONSISTÊNCIA
Com a Turma

```

class Aluno:

    def __init__(self, matricula: str):
        self.__matricula = matricula
        self.__turmas = []

    @property
    def matricula(self):
        return self.__matricula

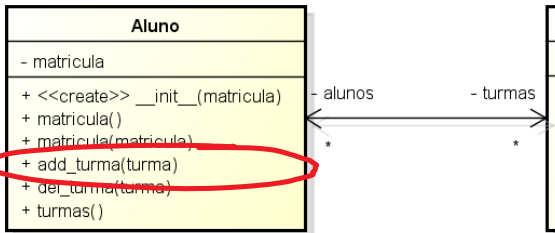
    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula

    def add_turma(self, turma: Turma):
        if (turma is not None) and (isinstance(turma, Turma)):
            if turma not in self.__turmas:
                self.__turmas.append(turma)
            if self not in turma.alunos:
                turma.alunos.append(self)

    def del_turma(self, turma: Turma):
        if (turma is not None) and (isinstance(turma, Turma)):
            if turma not in self.__turmas:
                self.__turmas.remove(turma)
            if self not in turma.alunos:
                turma.alunos.remove(self)

    @property
    def turmas(self):
        return self.__turmas
  
```

Associações bidirecionais



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

```
class Aluno:

    def __init__(self, matricula: str):
        self.__matricula = matricula
        self.__turmas = []

    @property
    def matricula(self):
        return self.__matricula

    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula

    def add_turma(self, turma: Turma):
        if (turma is not None) and (isinstance(turma, Turma)):
            if turma not in self.__turmas:
                self.__turmas.append(turma)
            if self not in turma.alunos:
                turma.alunos.append(self)

    def del_turma(self, turma: Turma):
        if (turma is not None) and (isinstance(turma, Turma)):
            if turma not in self.__turmas:
                self.__turmas.remove(turma)
            if self not in turma.alunos:
                turma.alunos.remove(self)

    @property
    def turmas(self):
        return self.__turmas
```

Associações bidirecionais

Aluno
- matricula
+ <<create>> __init__(matricula)
+ matricula()
+ matricula(matricula)
+ add_turma(turma)
+ del_turma(turma)
+ turmas()

- alunos

- turmas

Turma
- codigo
+ <<create>> __init__(codigo)
+ codigo()
+ codigo(codigo)
+ add_aluno(aluno)
+ del_aluno(aluno)
+ alunos()

Mesma lógica deve ser implementada do lado da Turma

```
aluno:
    def __init__(self, matricula: str):
        self.__matricula = matricula
        self.__turmas = []

    @property
    def matricula(self):
        return self.__matricula

    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula

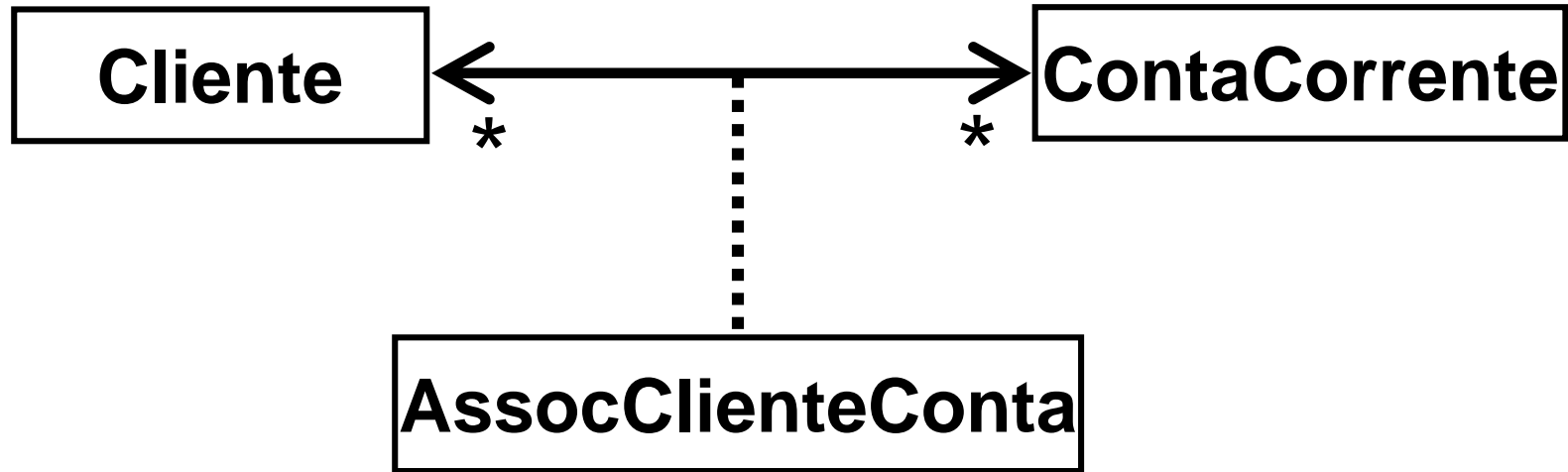
    def add_turma(self, turma: Turma):
        self.__turmas.append(turma)

    def del_turma(self, turma: Turma):
        self.__turmas.remove(turma)

    @property
    def turmas(self):
        return self.__turmas
```

Classes Associativas: visão lógica

Outra forma de resolver vários-para-vários:



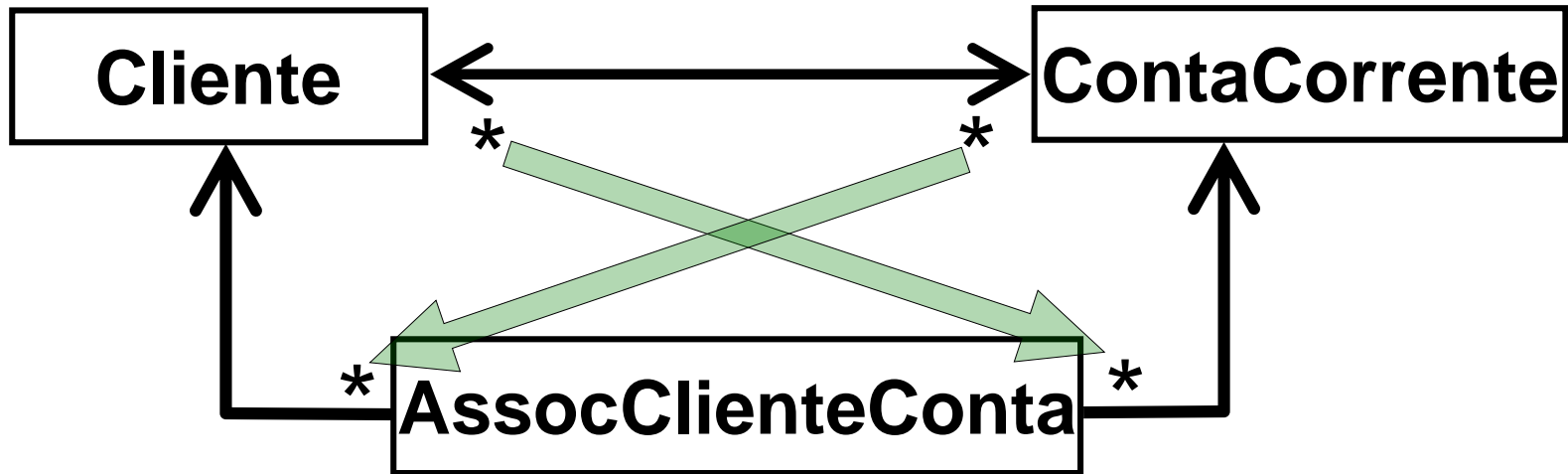
- Cliente e ContaCorrente passaram a ser independentes
- A nova classe irá implementar a associação

Classes Associativas: visão física



- ❑ A associação irá associar um objeto da classe **Cliente** com um objeto da classe **ContaCorrente**

Classes Associativas: transformação



- ❑ A associação irá associar um objeto da classe **Cliente** com um objeto da classe **ContaCorrente**

Implementação da classe associativa

```
class AssocClienteConta:

    def __init__(self, cliente: Cliente, conta: ContaCorrente):
        if isinstance(cliente, Cliente) and isinstance(conta, ContaCorrente):
            self.__cliente = cliente
            self.__conta = conta

    @property
    def cliente(self):
        return self.__cliente

    @cliente.setter
    def cliente(self, cliente):
        if isinstance(cliente, Cliente):
            self.__cliente = cliente

    @property
    def conta(self):
        return self.__conta

    @conta.setter
    def conta(self, conta):
        if isinstance(conta, ContaCorrente):
            self.__conta = conta
```

Implementação da classe associativa

```
class AssocClienteConta:
```

```
    def __init__(self, cliente: Cliente, conta: ContaCorrente):  
        if isinstance(cliente, Cliente) and isinstance(conta, ContaCorrente):  
            self.__cliente = cliente  
            self.__conta = conta
```

```
    @property  
    def cliente(self):  
        return self.__cliente
```

```
    @cliente.setter  
    def cliente(self, cliente):  
        if isinstance(cliente, Cliente):  
            self.__cliente = cliente
```

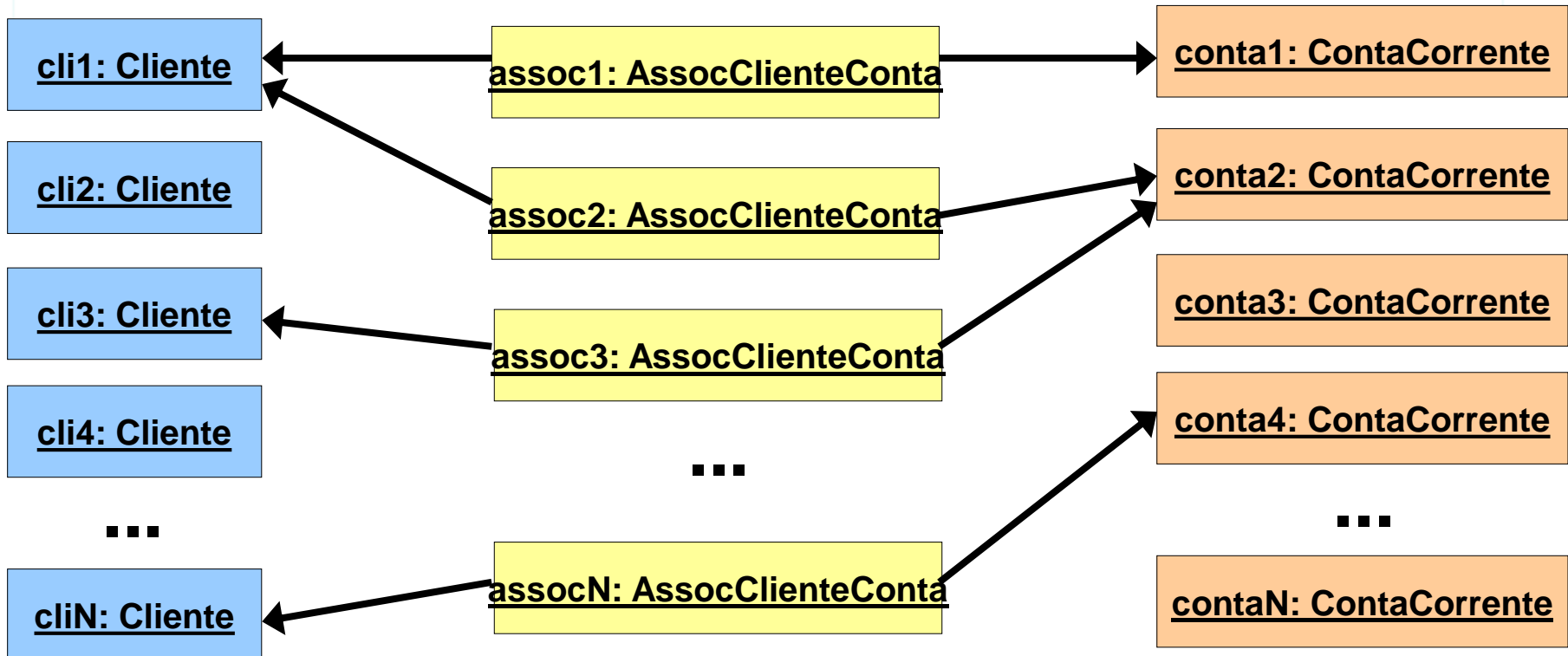
```
    @property  
    def conta(self):  
        return self.__conta
```

```
    @conta.setter  
    def conta(self, conta):  
        if isinstance(conta, ContaCorrente):  
            self.__conta = conta
```

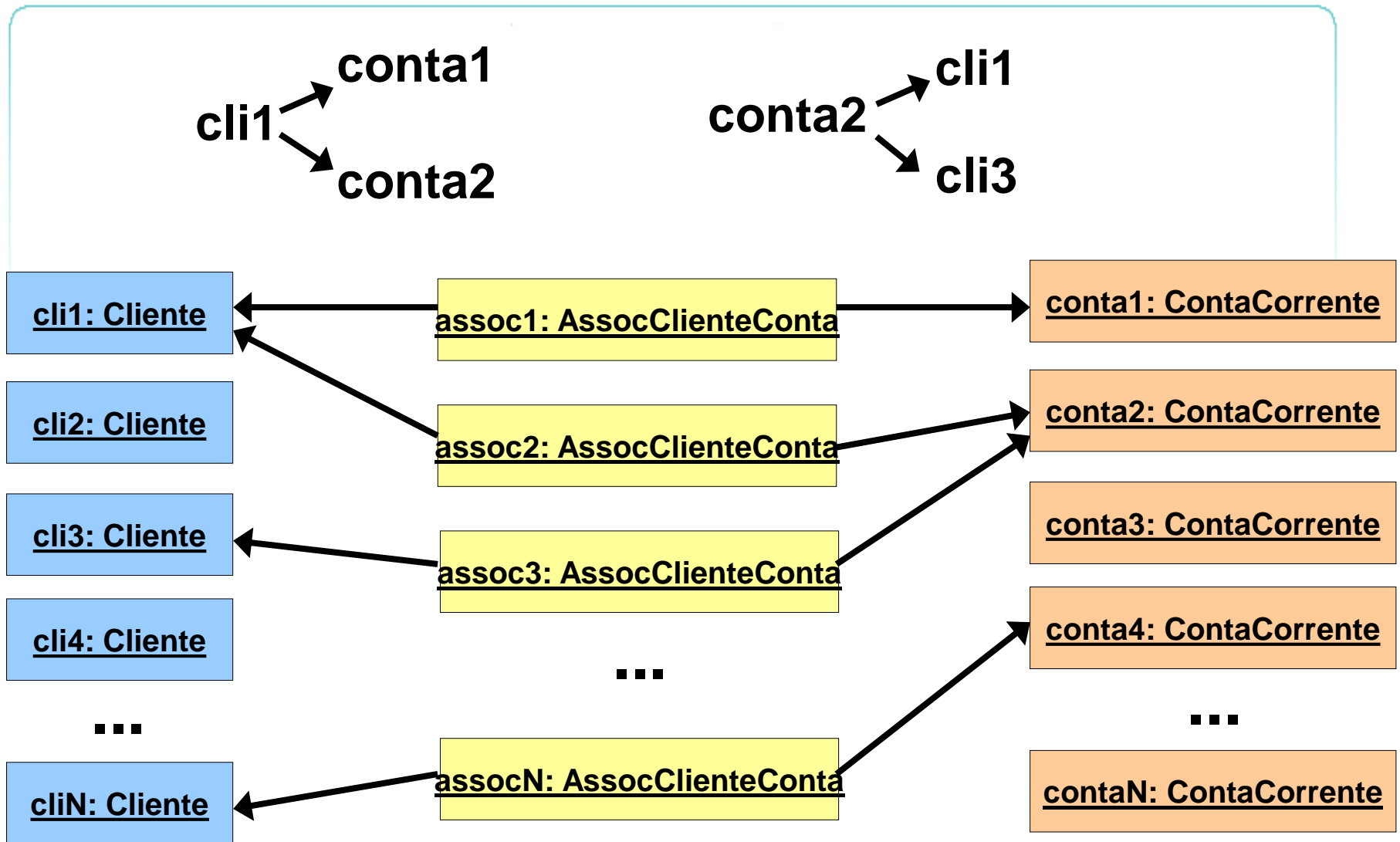
Assim é controlada a
associação
vários-para-vários

Implementação da classe associativa

- Para controlar as várias associações entre Cliente e ContaCorrente, pode-se implementar uma classe Broker para a associação

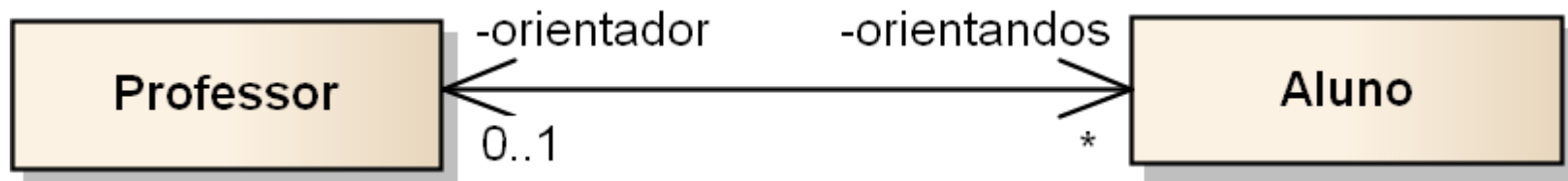


Implementação da classe associativa



Outro exemplo de Classe Associativa

Ainda no exemplo de Aluno ... agora vai fazer o TCC ...

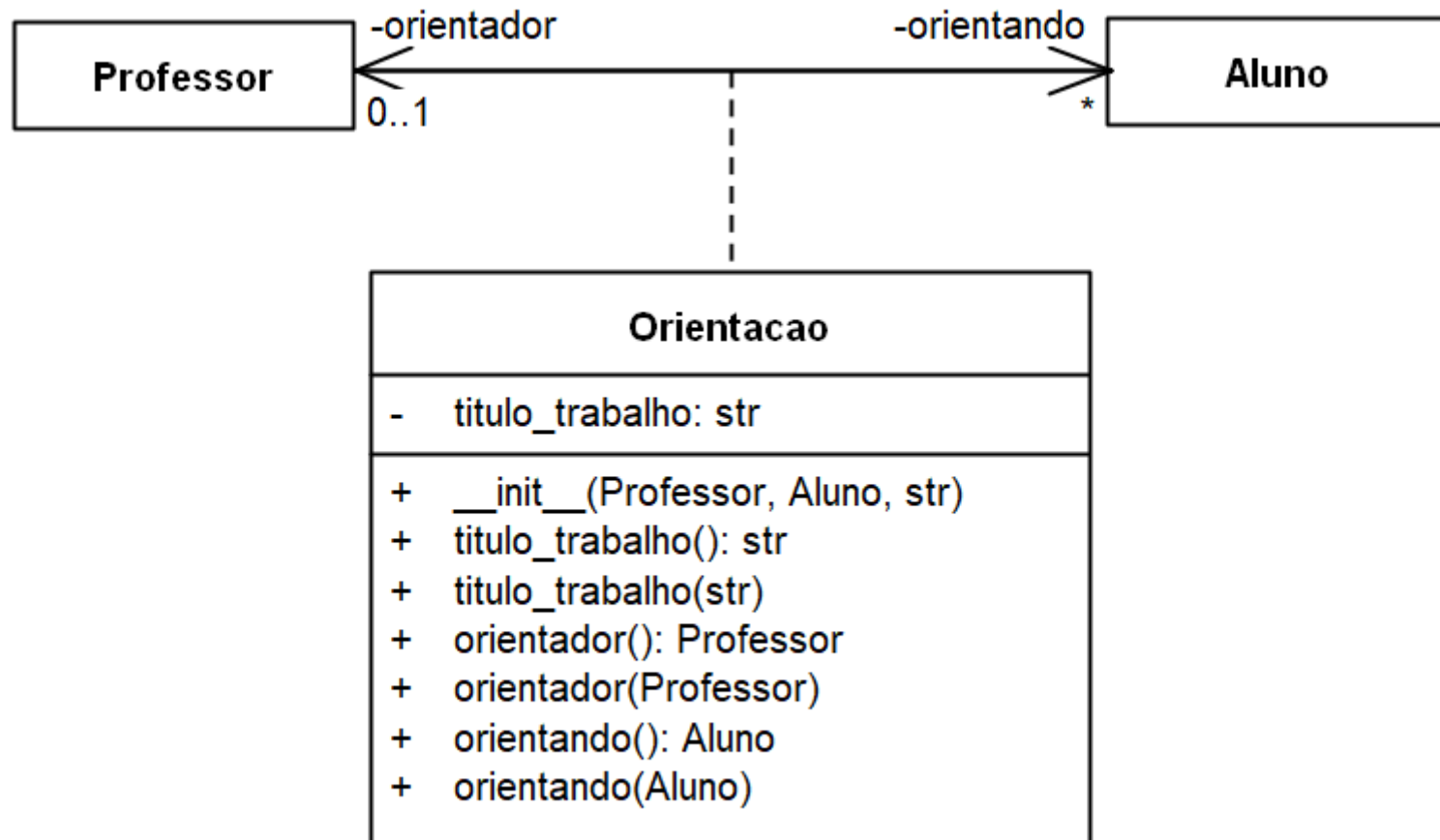


Outro exemplo de Classe Associativa

- E se precisarmos de **informações adicionais** que deveriam estar **na associação**?
 - Por exemplo, se for necessário armazenar o **título do trabalho** de conclusão de curso (TCC)
 - Note que o TCC não é uma informação do aluno e nem do professor, mas da associação de orientação entre Professor e Aluno

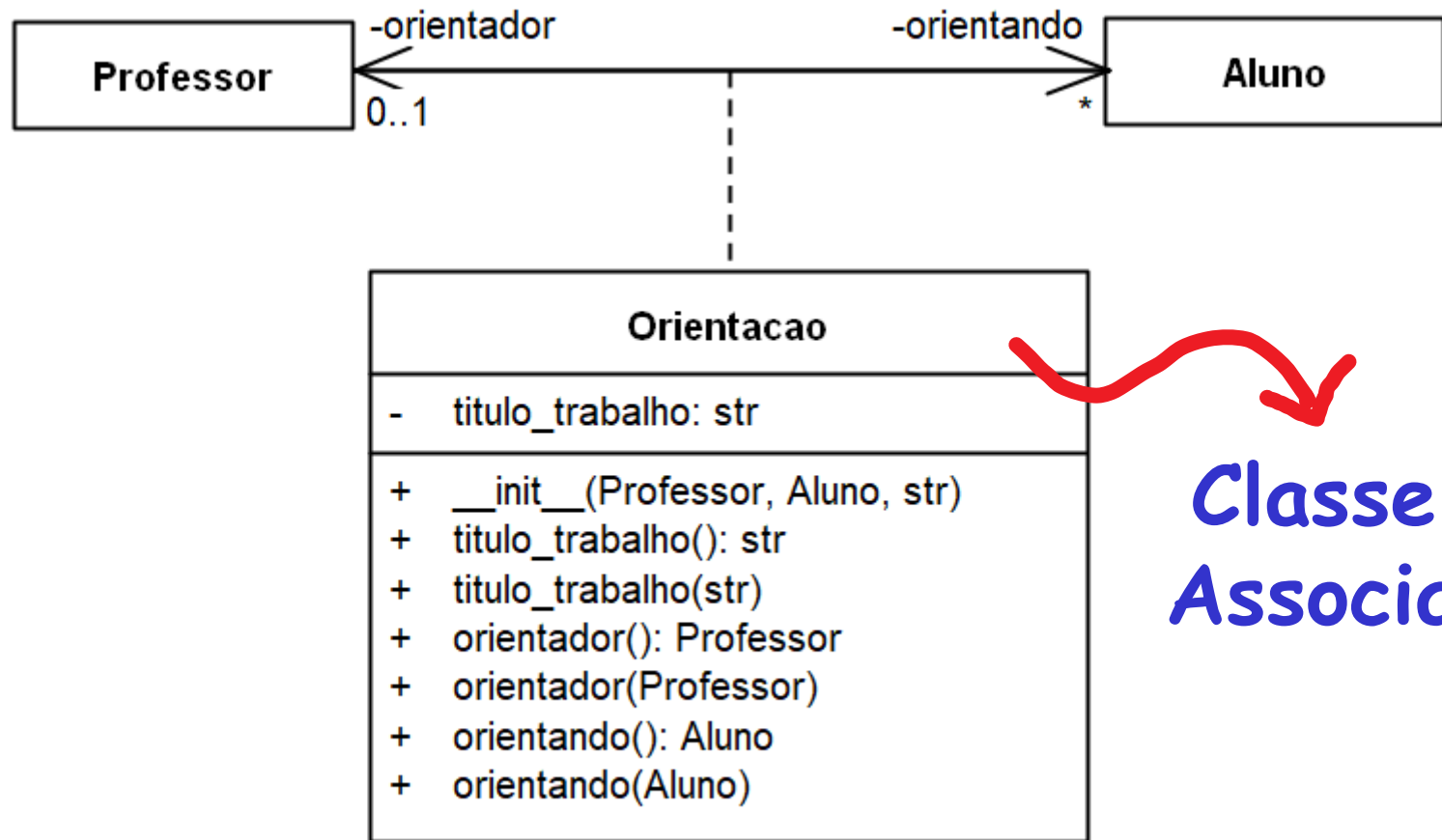
Outro exemplo de Classe Associativa

Que tal considerarmos a associação bidirecional como uma classe?



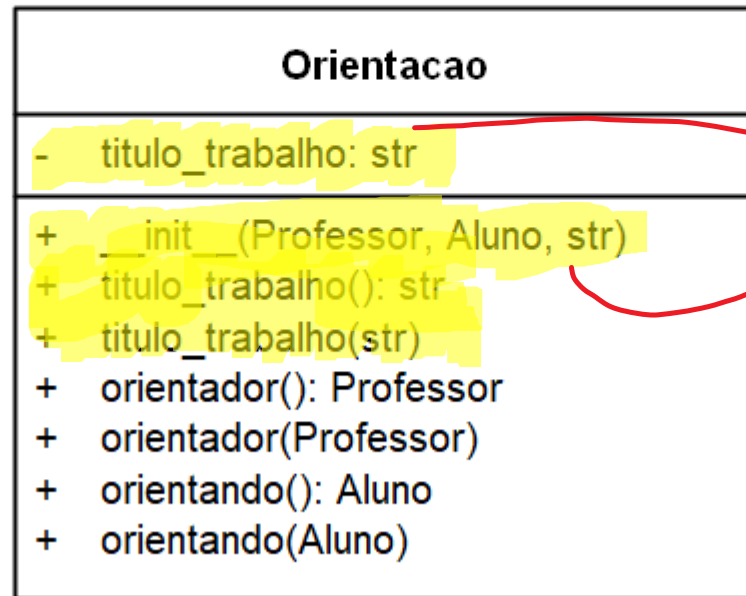
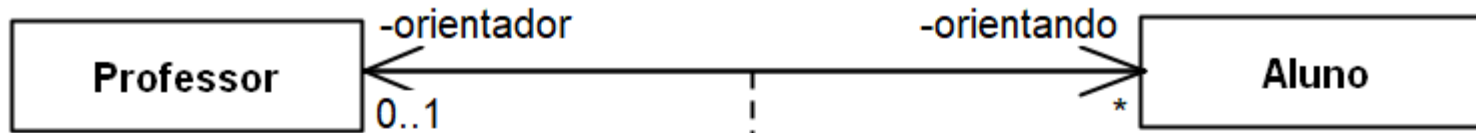
Outro exemplo de Classe Associativa

Que tal considerarmos a associação bidirecional como uma classe?



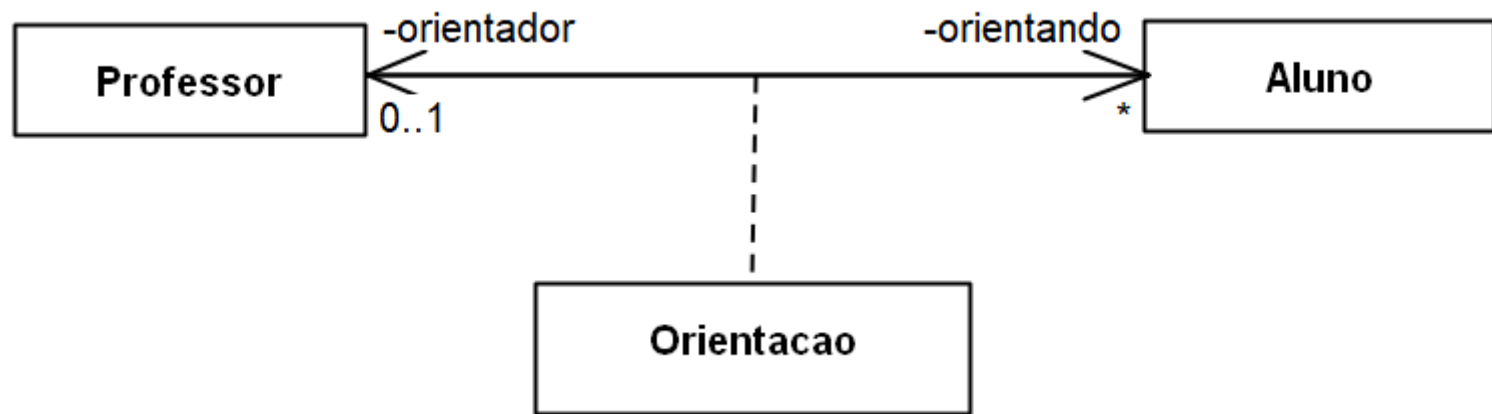
**Classe de
Associação**

Outro exemplo de Classe Associativa

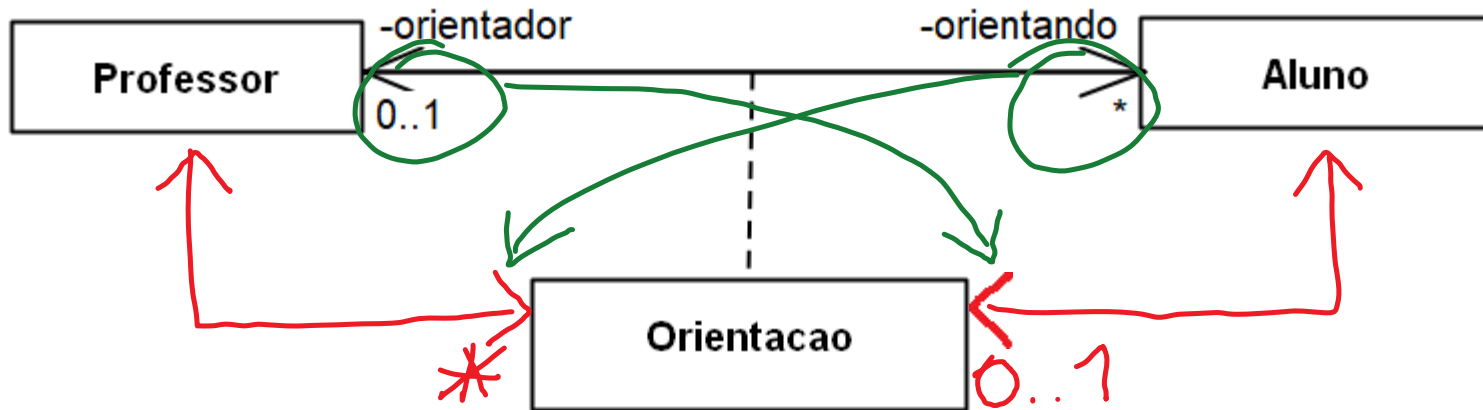


Atributos e operações específicas de Orientação

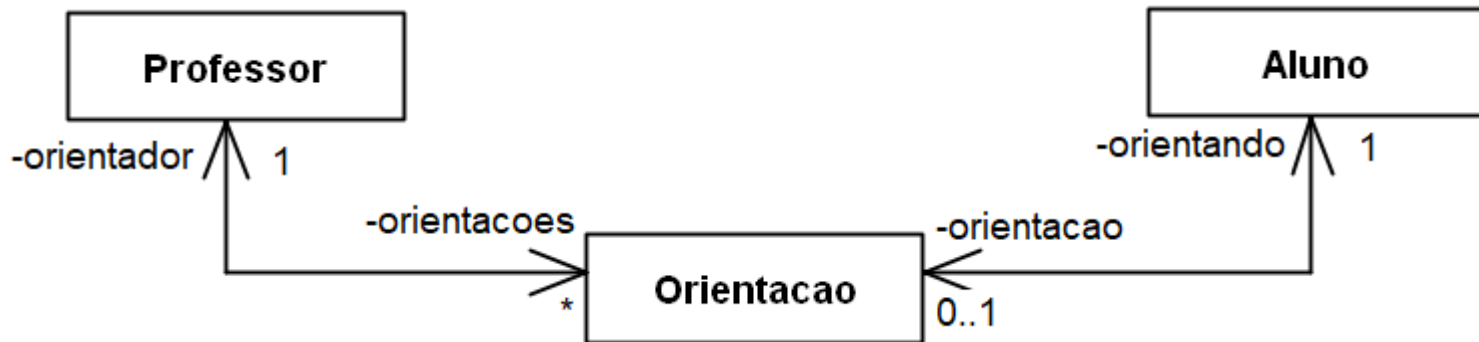
Como implementar?



Como fica fisicamente...



Visão física (implementação)



Classe Associativa: Orientacao

```
class Orientacao:

    def __init__(self, orientador: Professor, orientando: Aluno, titulo_trabalho: str):
        self.__orientador = orientador
        self.__orientando = orientando
        self.__titulo_trabalho = titulo_trabalho

    @property
    def orientador(self):
        return self.__orientador

    @orientador.setter
    def orientador(self, orientador: str):
        self.__orientador = orientador

    @property
    def orientando(self):
        return self.__orientando

    @orientando.setter
    def orientando(self, orientando: str):
        self.__orientando = orientando

    @property
    def titulo_trabalho(self):
        return self.__titulo_trabalho

    @titulo_trabalho.setter
    def titulo_trabalho(self, titulo_trabalho: str):
        self.__titulo_trabalho = titulo_trabalho
```

Orientacao

- titulo_trabalho: str
+ __init__(Professor, Aluno, str)
+ titulo_trabalho(): str
+ titulo_trabalho(str)
+ orientador(): Professor
+ orientador(Professor)
+ orientando(): Aluno
+ orientando(Aluno)

Classe Associativa: Orientacao

```
class Orientacao:
```

```
    def __init__(self, orientador: Professor, orientando: Aluno, titulo_trabalho: str):  
        self.__orientador = orientador  
        self.__orientando = orientando  
        self.__titulo_trabalho = titulo_trabalho
```

```
@property
```

```
def orientador(self):  
    return self.__orientador
```

```
@orientador.setter
```

```
def orientador(self, orientador: str):  
    self.__orientador = orientador
```

```
@property
```

```
def orientando(self):  
    return self.__orientando
```

```
@orientando.setter
```

```
def orientando(self, orientando: str):  
    self.__orientando = orientando
```

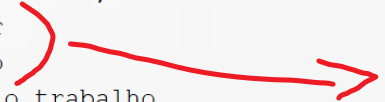
```
@property
```

```
def titulo_trabalho(self):  
    return self.__titulo_trabalho
```

```
@titulo_trabalho.setter
```

```
def titulo_trabalho(self, titulo_trabalho: str):  
    self.__titulo_trabalho = titulo_trabalho
```

**Ligação das duas
instâncias associadas**



Classe Associativa: No lado do Aluno

No lado Aluno (0 ou 1 professor)...

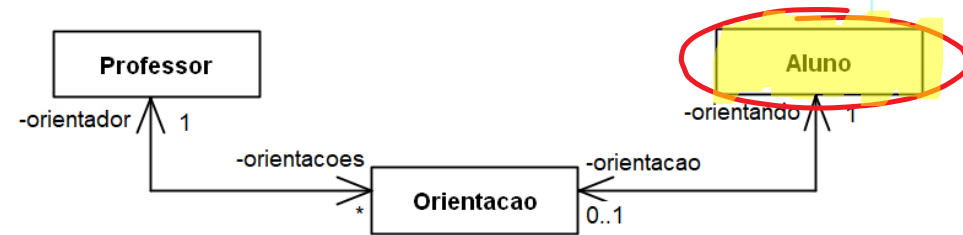
```
class Aluno:

    def __init__(self, matricula: str):
        self.__matricula = matricula
        self.__turmas = []
        self.__orientacao = None

    @property
    def matricula(self):
        return self.__matricula

    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula

    @property
    def orientacao(self):
        return self.__orientacao
```



Classe Associativa: No lado do Aluno

No lado Aluno (0 ou 1 professor)...

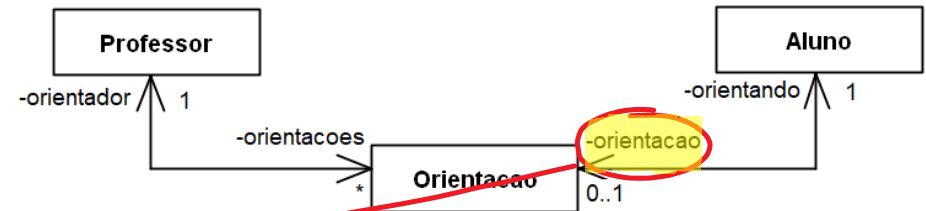
```
class Aluno:

    def __init__(self, matricula: str):
        self.__matricula = matricula
        self.__turmas = []
        self.__orientacao = None

    @property
    def matricula(self):
        return self.__matricula

    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula

    @property
    def orientacao(self):
        return self.__orientacao
```



Classe Associativa: No lado do Aluno

No lado Aluno (0 ou 1 professor)...

```
class Aluno:

    def __init__(self, matricula: str):
        self.__matricula = matricula
        self.__turmas = []
        self.__orientacao = None

    @property
    def matricula(self):
        return self.__matricula

    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula

    @property
    def orientacao(self):
        return self.__orientacao
```



Note que agora, o objeto é uma "Orientacao"

Classe Associativa: No lado do Aluno

```
@orientacao.setter
def orientacao(self, nova_orientacao: Orientacao):
    if nova_orientacao is None:
        if (self.__orientacao is not None) and (isinstance(self.__orientacao, Orientacao)):
            self.__orientacao.orientador.del_orientacao_by_orientando(self)
            self.__orientacao = None
    elif isinstance(nova_orientacao, Orientacao):
        if nova_orientacao.orientando != self:
            raise Exception('Associacao incorreta de Orientando, orientador: {0!s}, orientando {1!s}'.
                            format(nova_orientacao.orientador.matricula, nova_orientacao.orientando.matricula))
        else:
            novo_orientador = nova_orientacao.orientador
            orientacao_lado_professor = novo_orientador.find_orientacao_by_orientando(self)
            if self.__orientacao is None:
                self.__orientacao = nova_orientacao
                if orientacao_lado_professor != nova_orientacao:
                    novo_orientador.del_orientacao_by_orientando(orientacao_lado_professor)
                    novo_orientador.add_orientacao(nova_orientacao)
            else:
                if self.__orientacao.orientador == novo_orientador:
                    raise Exception('Duplicacao de Orientacao, orientador: {0!s}, orientando {1!s}'.
                                    format(nova_orientacao.orientador.matricula,
                                            nova_orientacao.orientando.matricula))
                else:
                    self.__orientacao.orientador.del_orientacao_by_orientando(self)
                    self.__orientacao = nova_orientacao
                    novo_orientador.add_orientacao(nova_orientacao)
```

Classe Associativa: No lado do Aluno

```
@orientacao.setter
def orientacao(self, nova_orientacao: Orientacao):
    if nova_orientacao is None:
        if (self.__orientacao is not None) and (isinstance(self.__orientacao, Orientacao)):
            self.__orientacao.orientador.del_orientacao_by_orientando(self)
            self.__orientacao = None
    elif isinstance(nova_orientacao, Orientacao):
        if nova_orientacao.orientando != self:
            raise Exception('Associacao incorreta de Orientando, orientador: {0!s}, orientando {1!s}'.
                             format(nova_orientacao.orientador.matricula, nova_orientacao.orientando.matricula))
        else:
            novo_orientador = nova_orientacao.orientador
            orientacao_lado_professor = novo_orientador.find_orientacao_by_orientando(self)
            if self.__orientacao is None:
                self.__orientacao = nova_orientacao
                if orientacao_lado_professor != nova_orientacao:
                    novo_orientador.del_orientacao_by_orientando(orientacao_lado_professor)
                    novo_orientador.add_orientacao(nova_orientacao)
            else:
                if self.__orientacao.orientador == novo_orientador:
                    raise Exception('Duplicacao de Orientacao, orientador: {0!s}, orientando {1!s}'.
                                     format(nova_orientacao.orientador.matricula,
                                             nova_orientacao.orientando.matricula))
                else:
                    self.__orientacao.orientador.del_orientacao_by_orientando(self)
                    self.__orientacao = nova_orientacao
                    novo_orientador.add_orientacao(nova_orientacao)
```

Removendo a
associação

Classe Associativa: No lado do Aluno

```
@orientacao.setter
def orientacao(self, nova_orientacao: Orientacao):
    if nova_orientacao is None:
        if (self.__orientacao is not None) and (isinstance(self.__orientacao, Orientacao)):
            self.__orientacao.orientador.del_orientacao_by_orientando(self)
            self.__orientacao = None
    elif isinstance(nova_orientacao, Orientacao):
        if nova_orientacao.orientando != self:
            raise Exception('Associacao incorreta de Orientando, orientador: {0!s}, orientando {1!s}'.
                             format(nova_orientacao.orientador.matricula, nova_orientacao.orientando.matricula))
        else:
            novo_orientador = nova_orientacao.orientador
            orientacao_lado_professor = novo_orientador.find_orientacao_by_orientando(self)
            if self.__orientacao is None:
                self.__orientacao = nova_orientacao
                if orientacao_lado_professor != nova_orientacao:
                    novo_orientador.del_orientacao_by_orientando(orientacao_lado_professor)
                    novo_orientador.add_orientacao(nova_orientacao)
            else:
                if self.__orientacao.orientador == novo_orientador:
                    raise Exception('Duplicacao de Orientacao, orientador: {0!s}, orientando {1!s}'.
                                     format(nova_orientacao.orientador.matricula,
                                             nova_orientacao.orientando.matricula))
                else:
                    self.__orientacao.orientador.del_orientacao_by_orientando(self)
                    self.__orientacao = nova_orientacao
                    novo_orientador.add_orientacao(nova_orientacao)
```

O aluno ainda
não tinha um
orientador

Classe Associativa: No lado do Aluno

```
@orientacao.setter
def orientacao(self, nova_orientacao: Orientacao):
    if nova_orientacao is None:
        if (self.__orientacao is not None) and (isinstance(self.__orientacao, Orientacao)):
            self.__orientacao.orientador.del_orientacao_by_orientando(self)
            self.__orientacao = None
    elif isinstance(nova_orientacao, Orientacao):
        if nova_orientacao.orientando != self:
            raise Exception('Associacao incorreta de Orientando, orientador: {0!s}, orientando {1!s}'.
                             format(nova_orientacao.orientador.matricula, nova_orientacao.orientando.matricula))
        else:
            novo_orientador = nova_orientacao.orientador
            orientacao_lado_professor = novo_orientador.find_orientacao_by_orientando(self)
            if self.__orientacao is None:
                self.__orientacao = nova_orientacao
                if orientacao_lado_professor != nova_orientacao:
                    novo_orientador.del_orientacao_by_orientando(orientacao_lado_professor)
                    novo_orientador.add_orientacao(nova_orientacao)
            else:
                if self.__orientacao.orientador == novo_orientador:
                    raise Exception('Duplicacao de Orientacao, orientador: {0!s}, orientando {1!s}'.
                                     format(nova_orientacao.orientador.matricula,
                                             nova_orientacao.orientando.matricula))
                else:
                    self.__orientacao.orientador.del_orientacao_by_orientando(self)
                    self.__orientacao = nova_orientacao
                    novo_orientador.add_orientacao(nova_orientacao)
```

O aluno já
tinha um
orientador

Classe Associativa: No lado do Aluno

```
@orientacao.setter
def orientacao(self, nova_orientacao: Orientacao):
    if nova_orientacao is None:
        if (self.__orientacao is not None) and (isinstance(self.__orientacao, Orientacao)):
            self.__orientacao.orientador.del_orientacao_by_orientando(self)
            self.__orientacao = None
        elif isinstance(nova_orientacao, Orientacao):
            if nova_orientacao.orientando != self:
                raise Exception('Associacao incorreta de Orientando, orientador: {0!s}, orientando {1!s}'
                                format(nova_orientacao.orientador.matricula, nova_orientacao.orientando.matricula))
            else:
                novo_orientador = nova_orientacao.orientador
                orientacao_lado_professor = novo_orientador.find_orientacao_by_orientando(self)
                if self.__orientacao is None:
                    self.__orientacao = nova_orientacao
                    if orientacao_lado_professor != nova_orientacao:
                        novo_orientador.del_orientacao_by_orientando(orientacao_lado_professor)
                        novo_orientador.add_orientacao(nova_orientacao)
                else:
                    if self.__orientacao.orientador == novo_orientador:
                        raise Exception('Duplicacao de Orientacao, orientador: {0!s}, orientando {1!s}'
                                        format(nova_orientacao.orientador.matricula,
                                              nova_orientacao.orientando.matricula))
                    else:
                        self.__orientacao.orientador.del_orientacao_by_orientando(self)
                        self.__orientacao = nova_orientacao
                        novo_orientador.add_orientacao(nova_orientacao)
```

Consistência

Classe Associativa: No lado do Aluno

Gerando exceção na
validação da associação

```
@orientacao.setter
def orientacao(self, nova_orientacao: Orientacao):
    if nova_orientacao is None:
        if (self.__orientacao is not None) and (isinstance(self.__orientacao, Orientacao)):
            self.__orientacao.orientador.del_orientacao_by_orientando(self)
            self.__orientacao = None
    elif isinstance(nova_orientacao, Orientacao):
        if nova_orientacao.orientando != self:
            raise Exception('Associacao incorreta de Orientando, orientador: {0!s}, orientando {1!s}'.
                             format(nova_orientacao.orientador.matricula, nova_orientacao.orientando.matricula))
        else:
            novo_orientador = nova_orientacao.orientador
            orientacao_lado_professor = novo_orientador.find_orientacao_by_orientando(self)
            if self.__orientacao is None:
                self.__orientacao = nova_orientacao
                if orientacao_lado_professor != nova_orientacao:
                    novo_orientador.del_orientacao_by_orientando(orientacao_lado_professor)
                    novo_orientador.add_orientacao(nova_orientacao)
            else:
                if self.__orientacao.orientador == novo_orientador:
                    raise Exception('Duplicacao de Orientacao, orientador: {0!s}, orientando {1!s}'.
                                     format(nova_orientacao.orientador.matricula,
                                             nova_orientacao.orientando.matricula))
                else:
                    self.__orientacao.orientador.del_orientacao_by_orientando(self)
                    self.__orientacao = nova_orientacao
                    novo_orientador.add_orientacao(nova_orientacao)
```


Classe Associativa: No lado do Professor

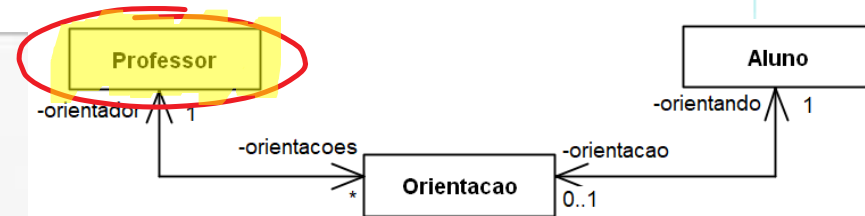
No lado Professor (vários orientandos)...

```
class Professor:

    def __init__(self, matricula: str):
        self.__matricula = matricula
        self.__orientacoes = []

    @property
    def matricula(self):
        return self.__matricula

    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula
```



Classe Associativa: No lado do Professor

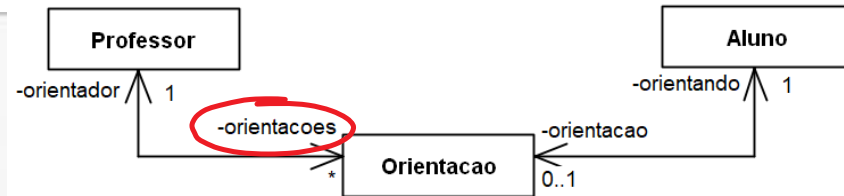
No lado Professor (vários orientandos)...

```
class Professor:

    def __init__(self, matricula: str):
        self._matricula = matricula
        self._orientacoes = []

    @property
    def matricula(self):
        return self._matricula

    @matricula.setter
    def matricula(self, matricula: str):
        self._matricula = matricula
```



Note que agora, a lista é de objetos "Orientacao"

Classe Associativa: No lado do Professor

No lado Professor (vários orientandos)...

```
class Professor:

    def __init__(self, matricula: str):
        self.__matricula = matricula
        self.__orientacoes = []

    @property
    def matricula(self):
        return self.__matricula

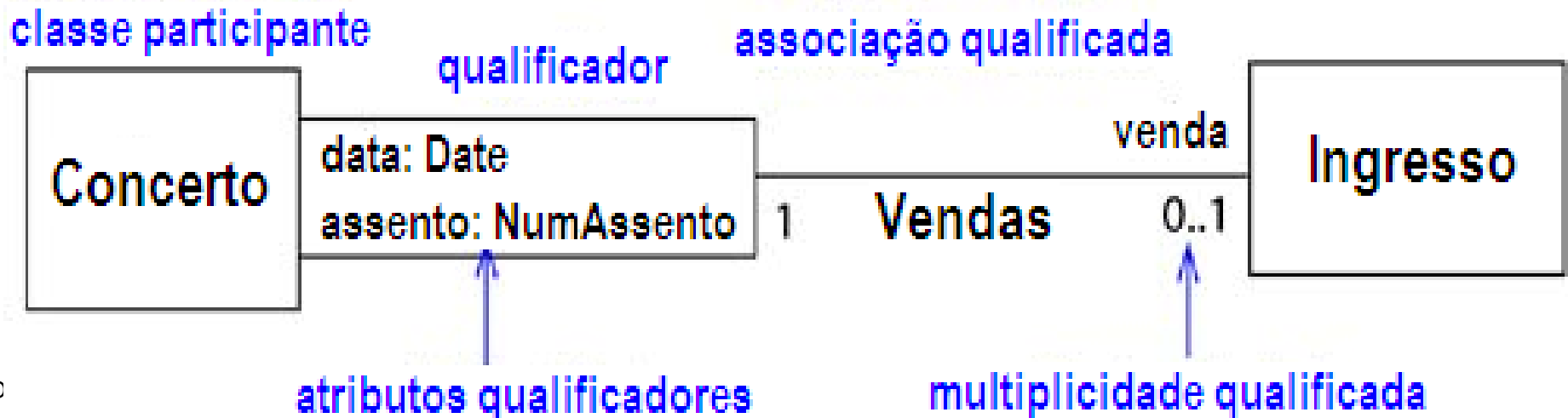
    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula

    ...
```

add_orientação e del_orientação
seguirão mesma lógica já
implementada em outras classes

Associação Qualificada

- Se o valor de um atributo da associação é único dentro de um conjunto de objetos relacionados, então ele é um **qualificador**
- Um qualificador é um valor que seleciona um único objeto de um conjunto de objetos relacionados através de uma associação; qualificadores permitem a modelagem de índices



Associação Qualificada

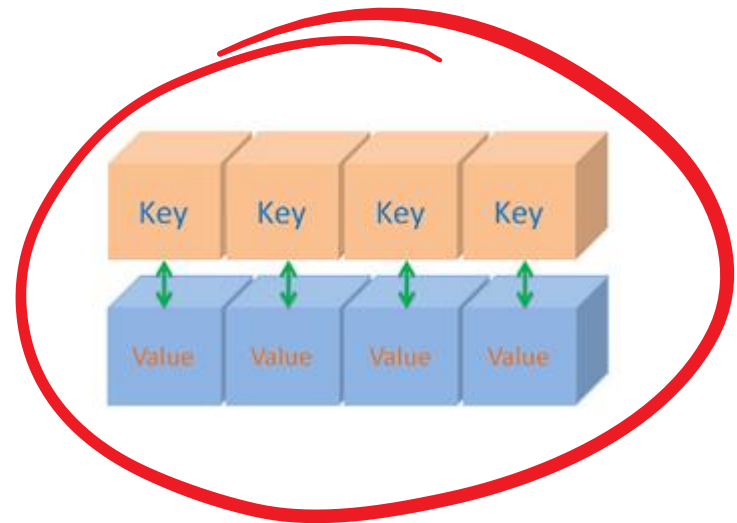
- É o equivalente na UML para o conceito na programação conhecido como **dicionários**, **hashmaps** ou **arrays associativos**
- No exemplo abaixo, o qualificador informa que na conexão com um **Pedido**, poderá haver um **ItemPedido** para cada instância de **Produto**



Associação qualificada

Implementação na forma de **Dicionários**

Um **dicionário** é composto por um conjunto de associações de um objeto chave para um objeto valor.



Associação qualificada: implementação

```
class Pedido:
```

```
    def __init__(self, numero: int):
        self.__numero = numero
        self.__itens = {}
```

```
    @property
    def numero(self):
        return self.__numero
```

```
    @numero.setter
    def numero(self, numero: int):
        self.__numero = numero
```

```
    @property
    def itens(self):
        return self.__itens.values()
```

```
    def add_item(self, item_pedido: ItemPedido):
        if (item_pedido is not None) \
            and (isinstance(item_pedido, ItemPedido)) \
            and (isinstance(item_pedido.produto, Produto)):
            self.__itens[item_pedido.produto] = item_pedido
```

```
    def get_item(self, produto: Produto):
        if (produto is not None) and (isinstance(produto, Produto)):
            return self.__itens[produto]
```



Associação qualificada: implementação

```
class Pedido:
```

```
def __init__(self, numero: int):
    self.__numero = numero
    self.__itens = {}
```

```
@property
def numero(self):
    return self.__numero
```

```
@numero.setter
def numero(self, numero: int):
    self.__numero = numero
```

```
@property
def itens(self):
    return self.__itens.values()
```

```
def add_item(self, item_pedido: ItemPedido):
    if (item_pedido is not None) \
        and (isinstance(item_pedido, ItemPedido)) \
        and (isinstance(item_pedido.produto, Produto)):
        self.__itens[item_pedido.produto] = item_pedido
```

```
def get_item(self, produto: Produto):
    if (produto is not None) and (isinstance(produto, Produto)):
        return self.__itens[produto]
```



Dicionário,
representando a
Associação
Qualificada

Associação qualificada: implementação

```
class Pedido:
```

```
    def __init__(self, numero: int):
        self.__numero = numero
        self.__itens = {}
```

```
    @property
    def numero(self):
        return self.__numero
```

```
    @numero.setter
    def numero(self, numero: int):
        self.__numero = numero
```

```
    @property
    def itens(self):
        return self.__itens.values()
```

```
    def add_item(self, item_pedido: ItemPedido):
        if (item_pedido is not None) \
            and (isinstance(item_pedido, ItemPedido)) \
            and (isinstance(item_pedido.produto, Produto)):
```

```
    def get_item(self, produto: Produto):
        if (produto is not None) and (isinstance(produto, Produto)):
            return self.__itens[produto]
```



Garantindo
classes
esperadas

Associação qualificada: implementação

```
class Pedido:
```

```
def __init__(self, numero: int):
    self.__numero = numero
    self.__itens = {}
```

```
@property
def numero(self):
    return self.__numero
```

```
@numero.setter
def numero(self, numero: int):
    self.__numero = numero
```

```
@property
def itens(self):
    return self.__itens.values()
```

```
def add_item(self, item_pedido: ItemPedido):
    if (item_pedido is not None) \
        and (isinstance(item_pedido, ItemPedido)) \
        and (isinstance(item_pedido.produto, Produto)):
        self.__itens[item_pedido.produto] = item_pedido
```

```
def get_item(self, produto: Produto):
    if (produto is not None) and (isinstance(produto, Produto)):
        return self.__itens[produto]
```



produto é a chave e item_pedido é o valor

Associação qualificada: implementação

```
class Pedido:
```

```
    def __init__(self, numero: int):
        self.__numero = numero
        self.__itens = {}
```

```
    @property
    def numero(self):
        return self.__numero
```

```
    @numero.setter
    def numero(self, numero: int):
        self.__numero = numero
```

```
    @property
    def itens(self):
        return self.__itens.values()
```

```
    def add_item(self, item_pedido: ItemPedido):
        if (item_pedido is not None) \
            and (isinstance(item_pedido, ItemPedido)) \
            and (isinstance(item_pedido.produto, Produto)):
            self.__itens[item_pedido.produto] = item_pedido
```

```
    def get_item(self, produto: Produto):
        if (produto is not None) and (isinstance(produto, Produto)):
            return self.__itens[produto]
```



produto é
a chave e
item_pedido
é o valor

item_pedido

Associação qualificada: implementação

```
def add_item(self, item_pedido: ItemPedido):  
    if (item_pedido is not None) \  
        and (isinstance(item_pedido, ItemPedido)) \  
        and (isinstance(item_pedido.produto, Produto)):  
        self.__items[item_pedido.produto] = item_pedido  
  
def get_item(self, produto: Produto):  
    if (produto is not None) and (isinstance(produto, Produto)):  
        return self.__items[produto]
```

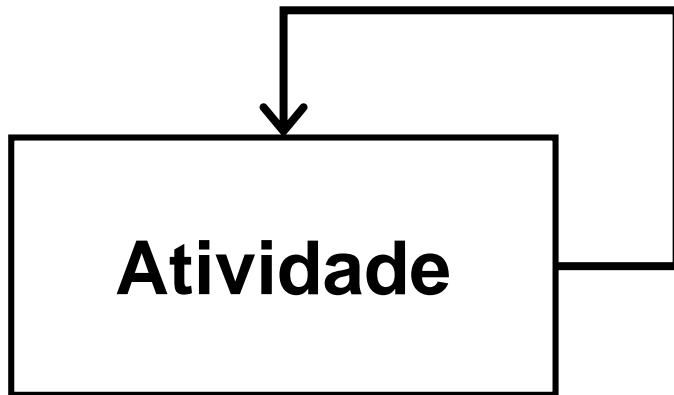


- ❑ Assim, todo acesso a um dado ItemPedido requer um Produto como um parâmetro, numa estrutura de dados baseada em chave/valor
- ❑ A multiplicidade no contexto do qualificador:
um Pedido pode ter vários ItemPedido, mas apenas 0 ou 1 por Produto

Associações reflexivas

- Uma associação reflexiva modela o relacionamento entre objetos da mesma classe
- A utilização de nomes de papel é bastante indicada

- **proxima_atividade**



```
class Atividade:
```

```
def __init__(self, proxima_atividade):
    if isinstance(proxima_atividade, Atividade):
        self.__proxima_atividade = proxima_atividade
```

Mais detalhes sobre: **AGREGAÇÃO E COMPOSIÇÃO**

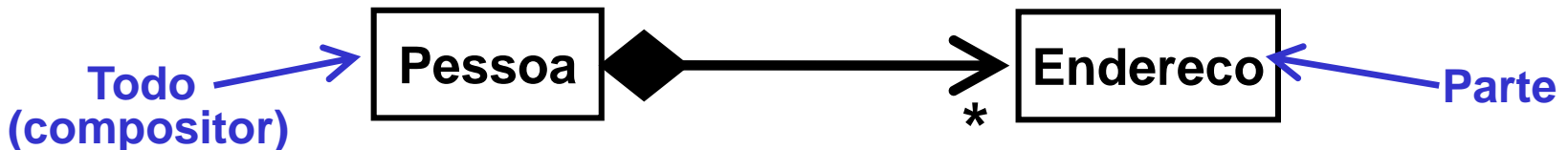


Agregação e Composição

- 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



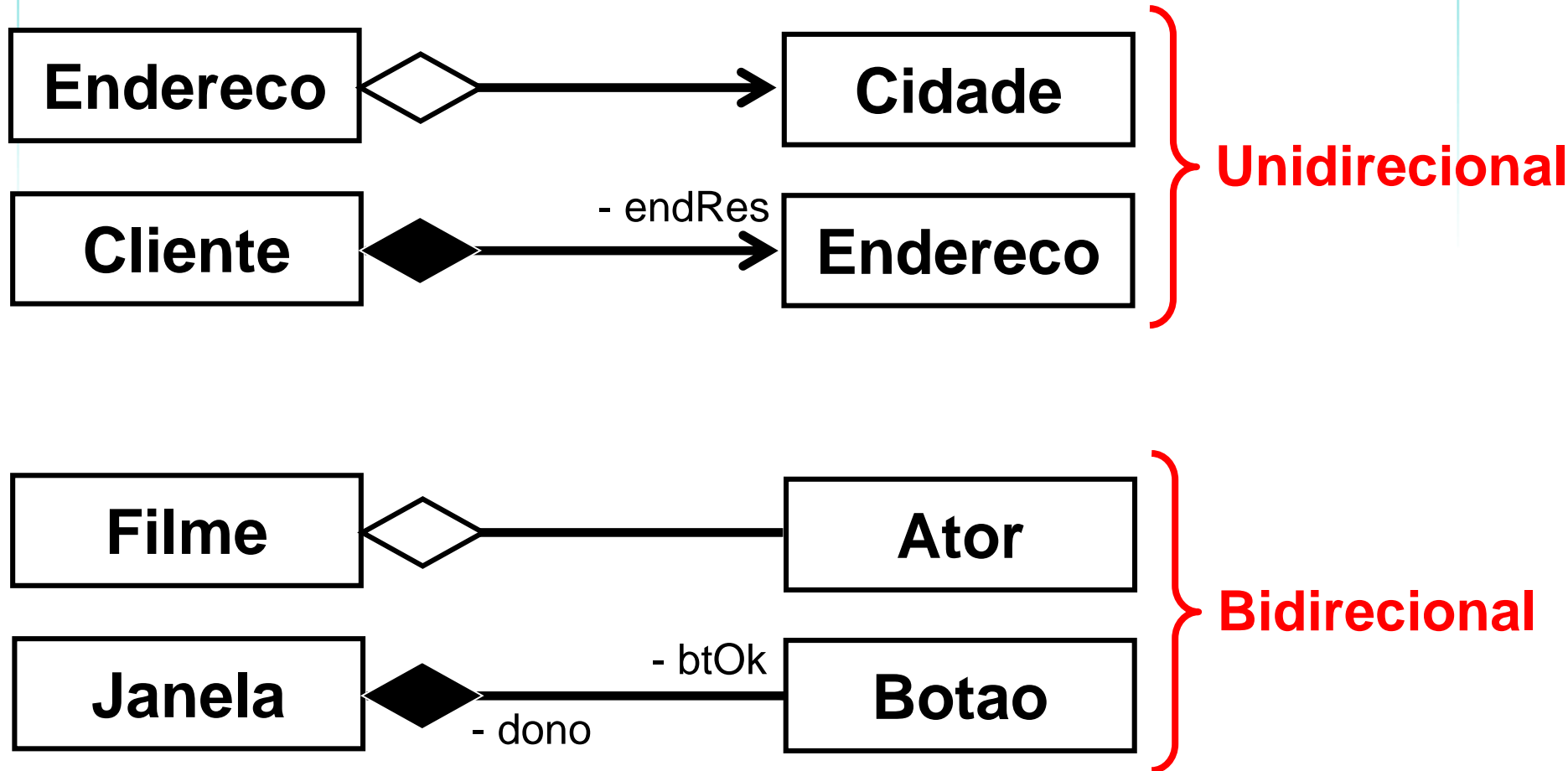
- Uma **composição** é uma forma mais forte de associação na qual o compositor tem responsabilidade exclusiva sobre gerenciar suas partes, assim como sua criação e destruição; sua notação é um losango preenchido no final da conexão, anexado à classe compositora



Agregação e Composição

- Na **agregação**, um objeto parte pode ser **compartilhado** (***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
```

Não está adicionando
um endereço!

```
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)
```

```
...
```

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