

# **Desenvolvimento de Sistemas Orientados a Objetos I**

## **Relações entre Objetos – Parte II**

---

Jean Carlo Rossa Hauck, Dr.

[jean.hauck@ufsc.br](mailto: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

# 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.__fornecedores = []
```

**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:

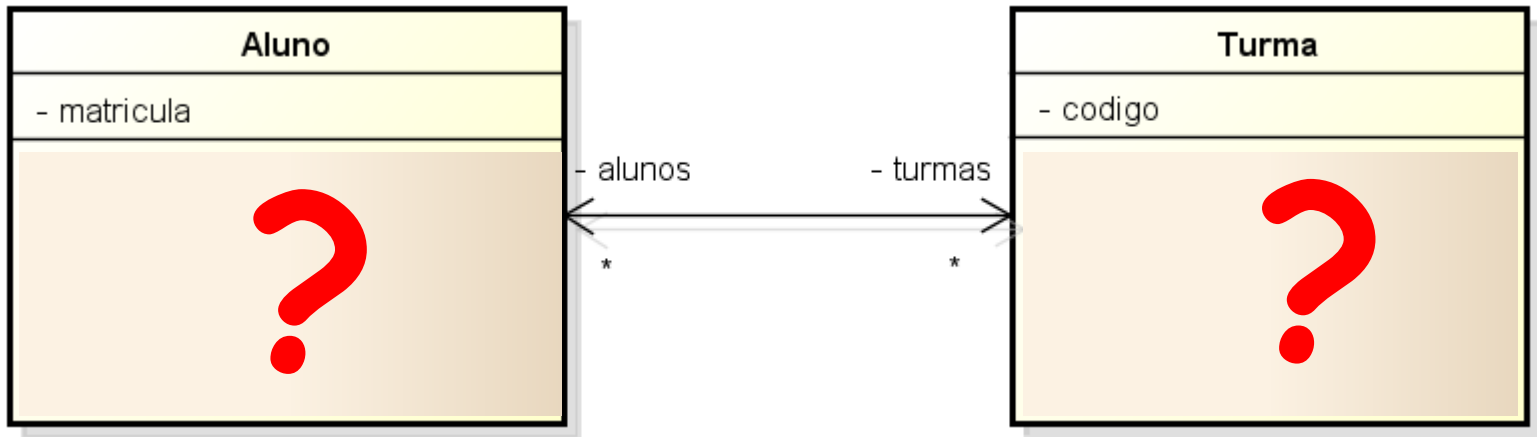
### ❑ Associações bidirecionais...

- ❑ ... 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



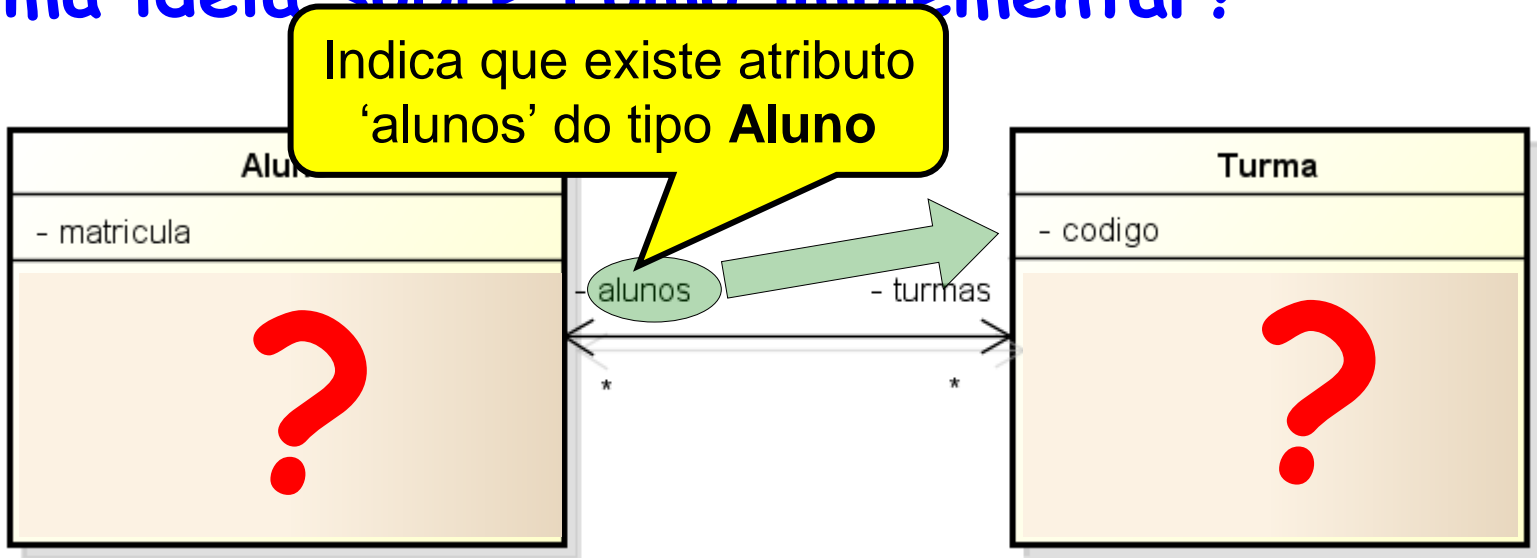
# Associações bidirecionais

Alguma ideia sobre como implementar?



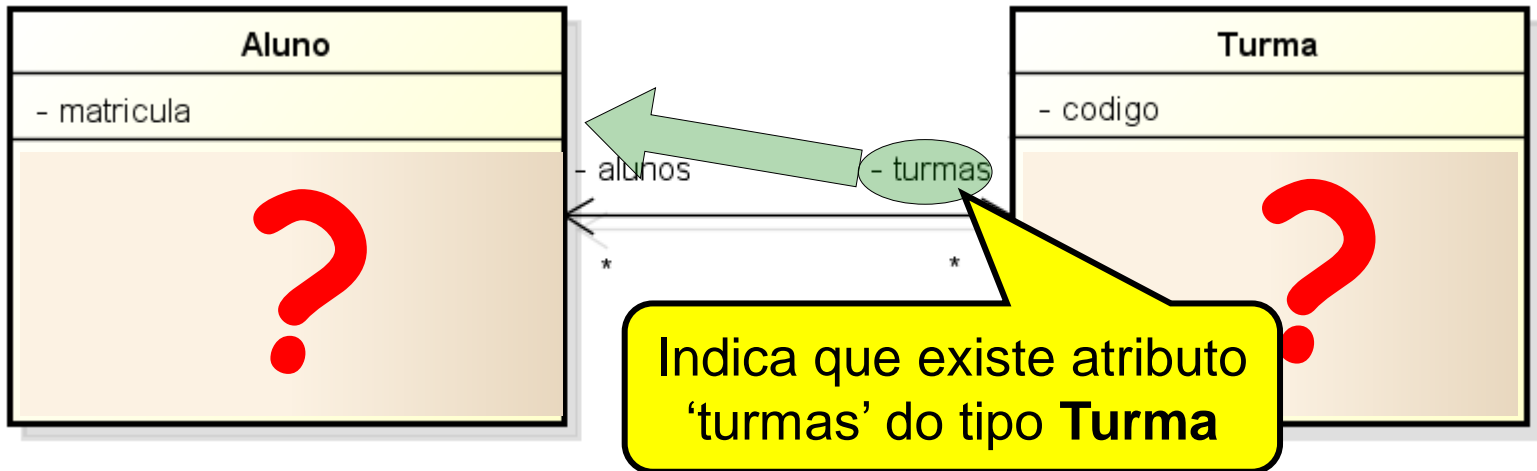
# Associações bidirecionais

Alguma ideia sobre como implementar?



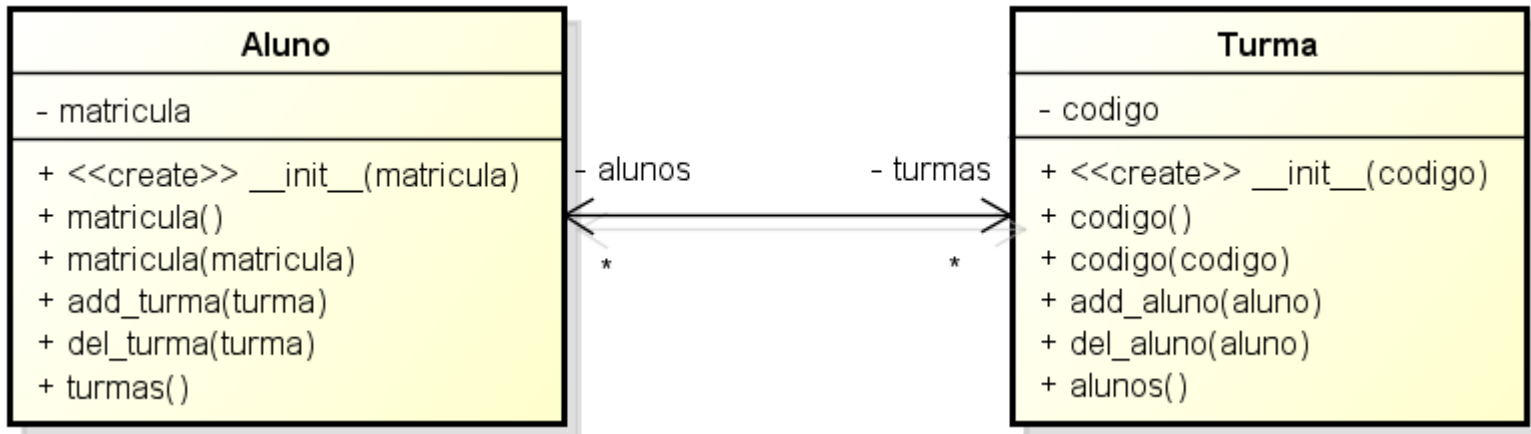
# Associações bidirecionais

Alguma ideia sobre como implementar?



# Associações bidirecionais

E agora?



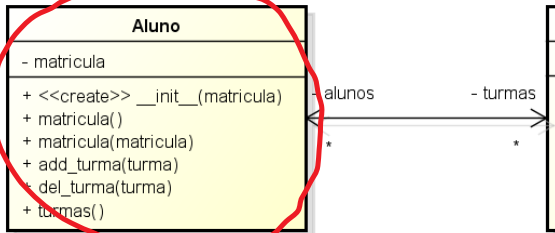
# Associações bidirecionais

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)
```

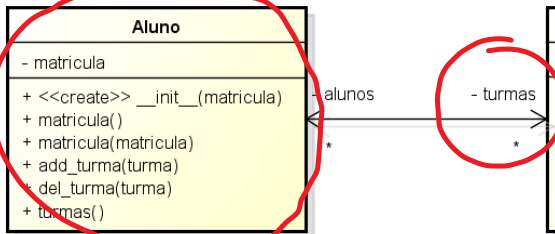
```
def del_turma(self, turma: Turma):
```

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

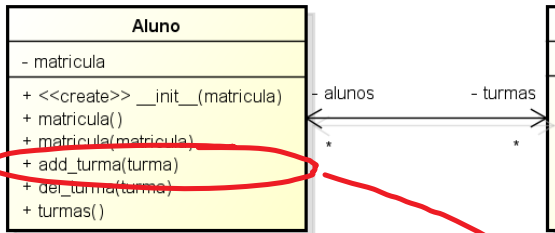
```
def del_turma(self, turma: Turma):
```

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

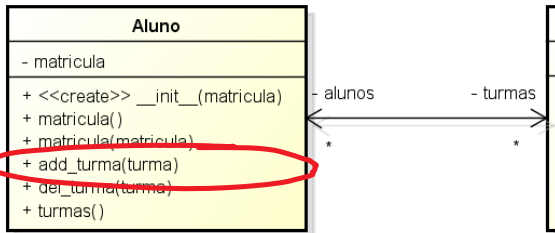
```
def add_turma(self, turma: Turma):
    if (turma is not None) and (isinstance(turma, Turma)):
        if turma not in self.__turmas:
            self.__turmas.append(turma)
        if self not in turma.alunos:
            turma.alunos.append(self)
```

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

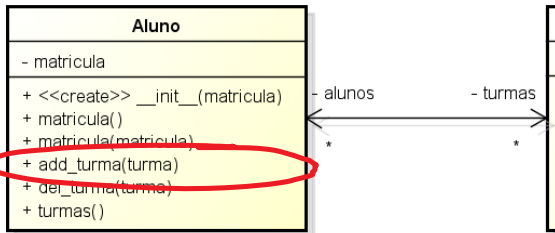
```
    if (turma is not None) and (isinstance(turma, Turma)):
        if turma in self.__turmas:
            self.__turmas.remove(turma)
        if self in turma.alunos:
            turma.alunos.remove(self)
```

```
@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

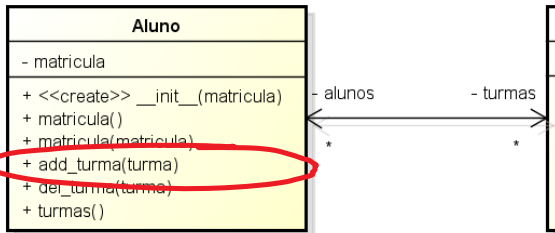
    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula

    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 in self.__turmas:
                self.__turmas.remove(turma)
            if self 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

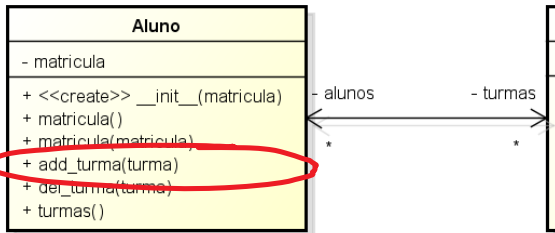
    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula

    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 in self.__turmas:
                self.__turmas.remove(turma)
            if self 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 in self.__turmas:
                self.__turmas.remove(turma)
            if self 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

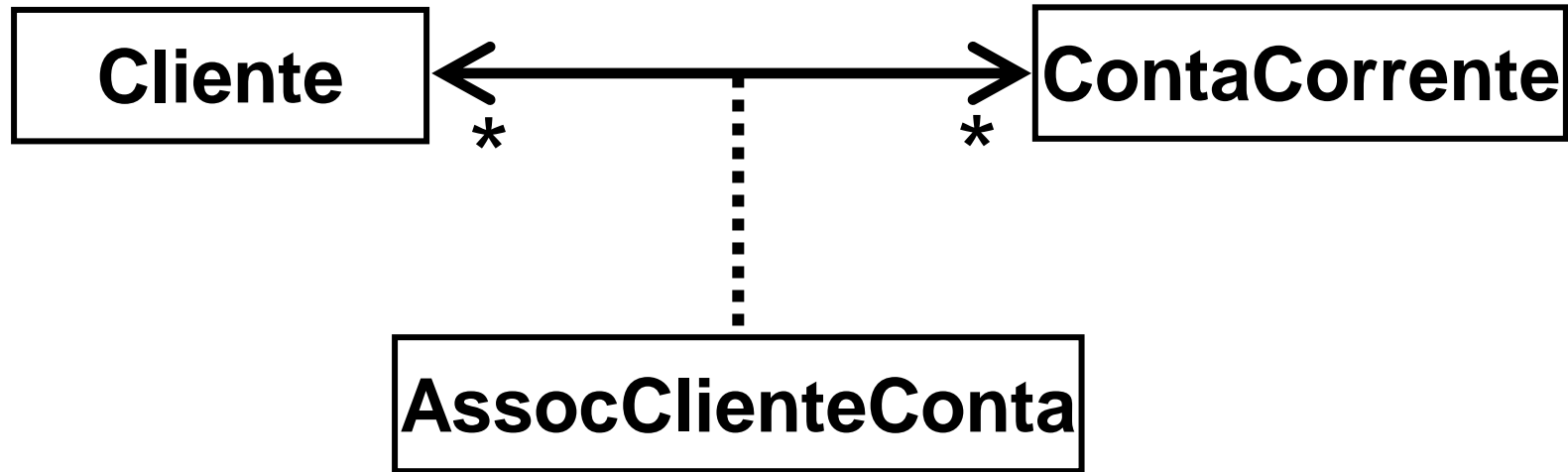
    def add_turma(self, turma: Turma):
        self.__turmas.append(turma)

    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:**



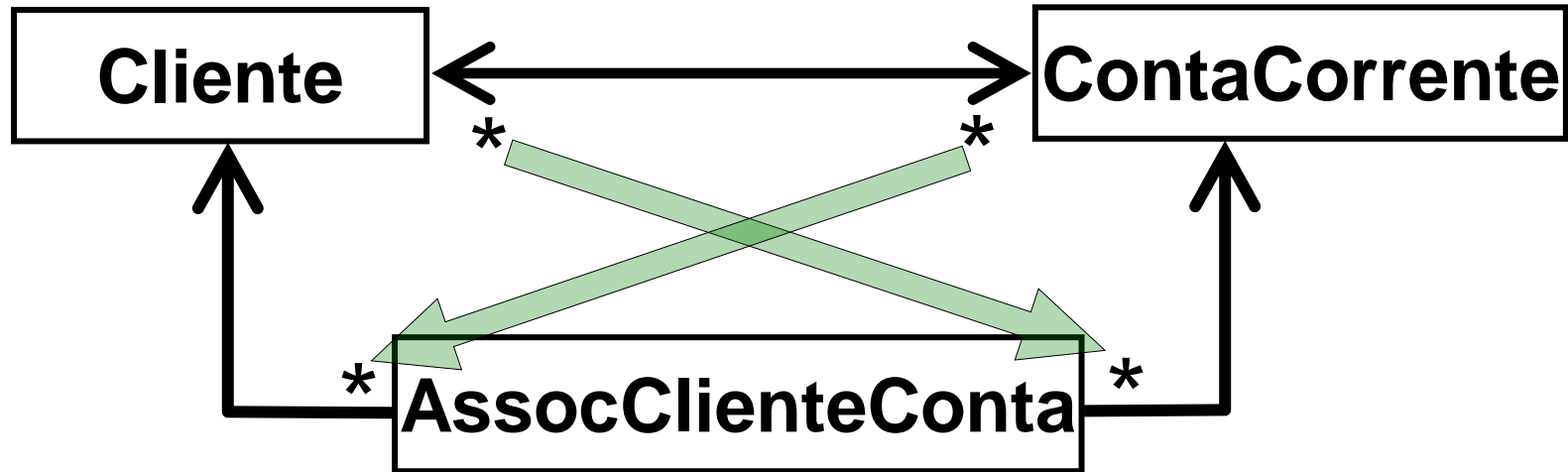
- Cliente e ContaCorrente passaram a ser independentes
- A nova classe irá implementar a associação

# 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
```

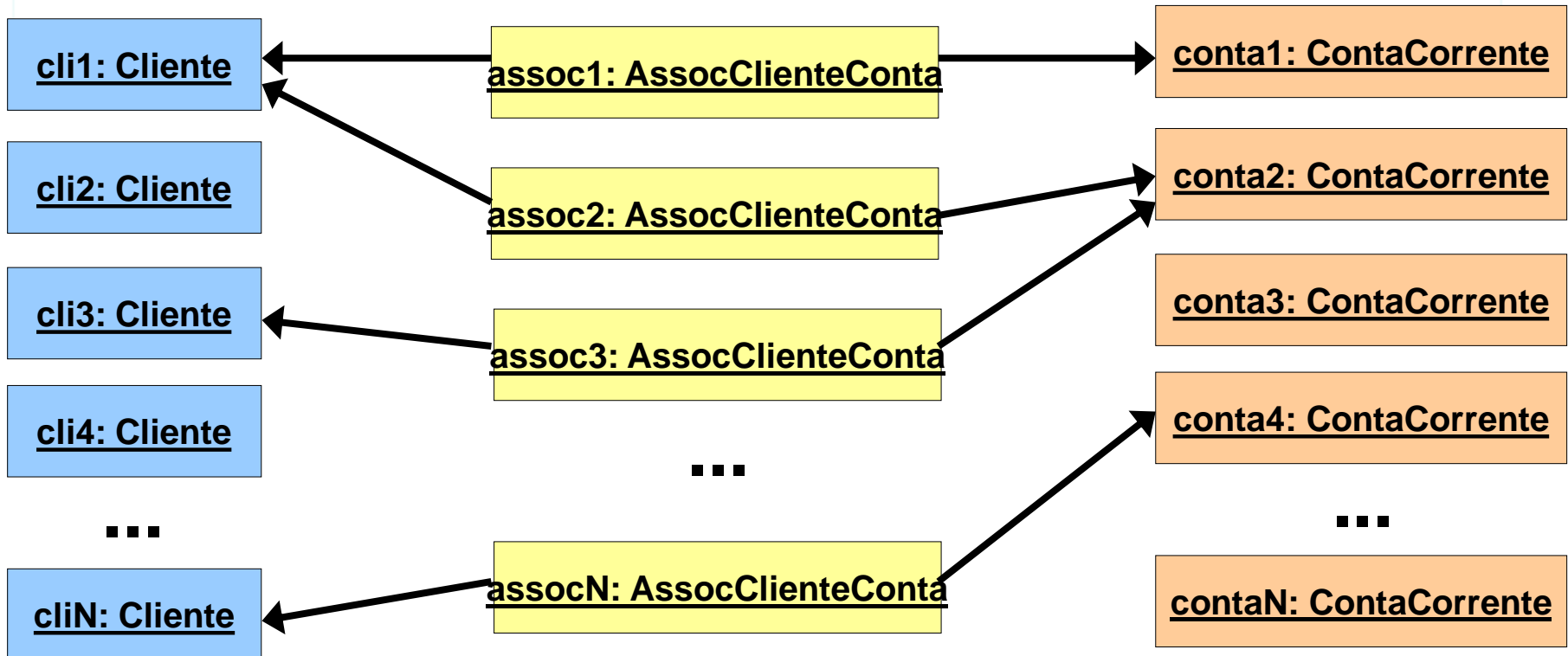
```
    @property  
    def conta(self):  
        return self.__conta
```

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

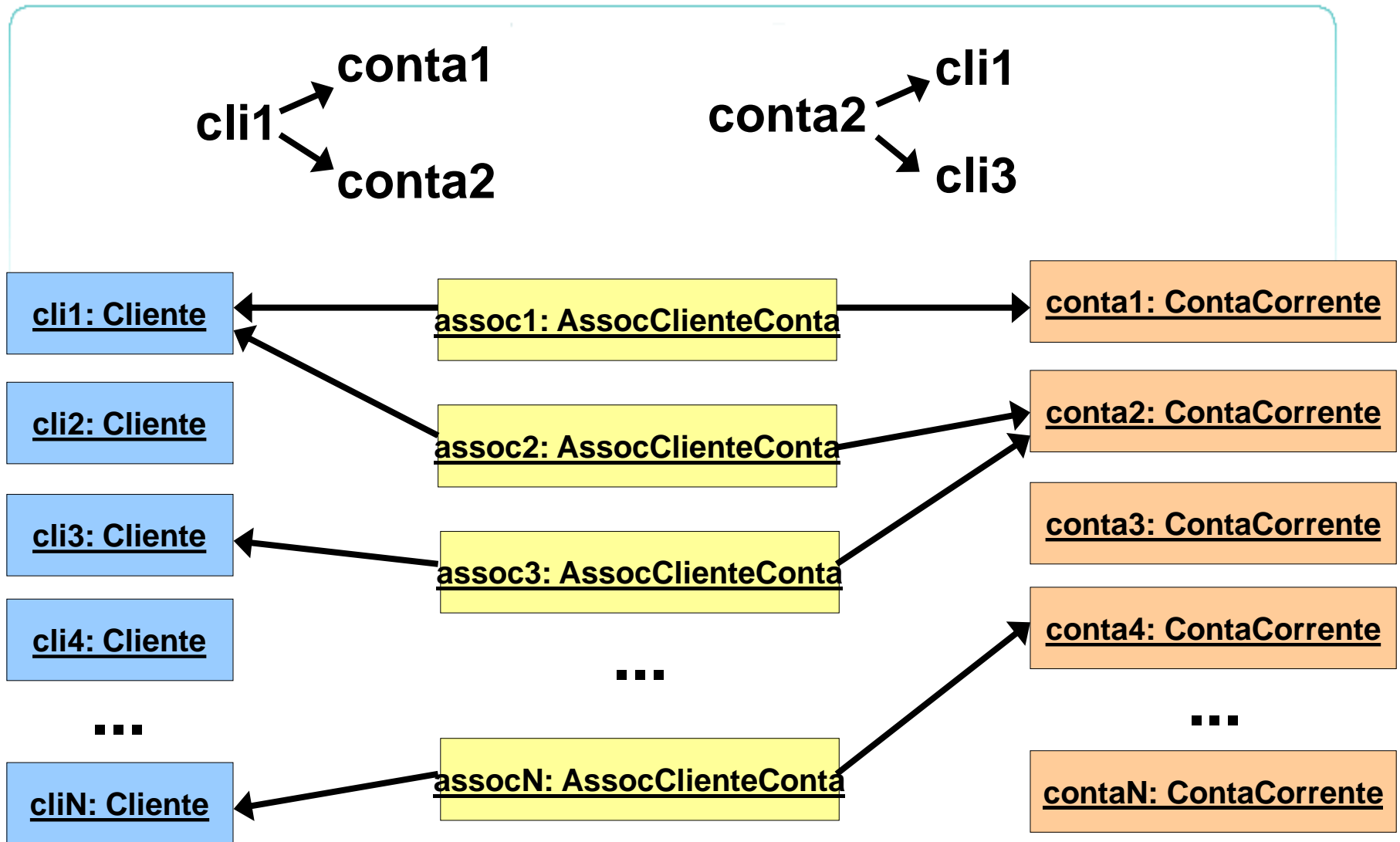
Assim é controlada a  
associação  
vários-para-vários

# Implementação da classe associativa

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

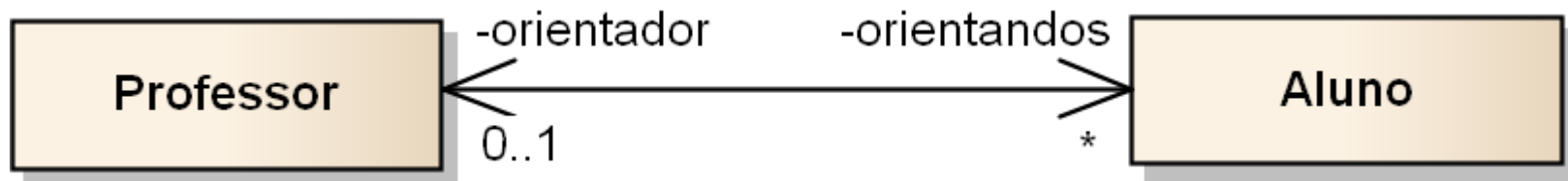


# Implementação da classe associativa



## Outro exemplo de Classe Associativa

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

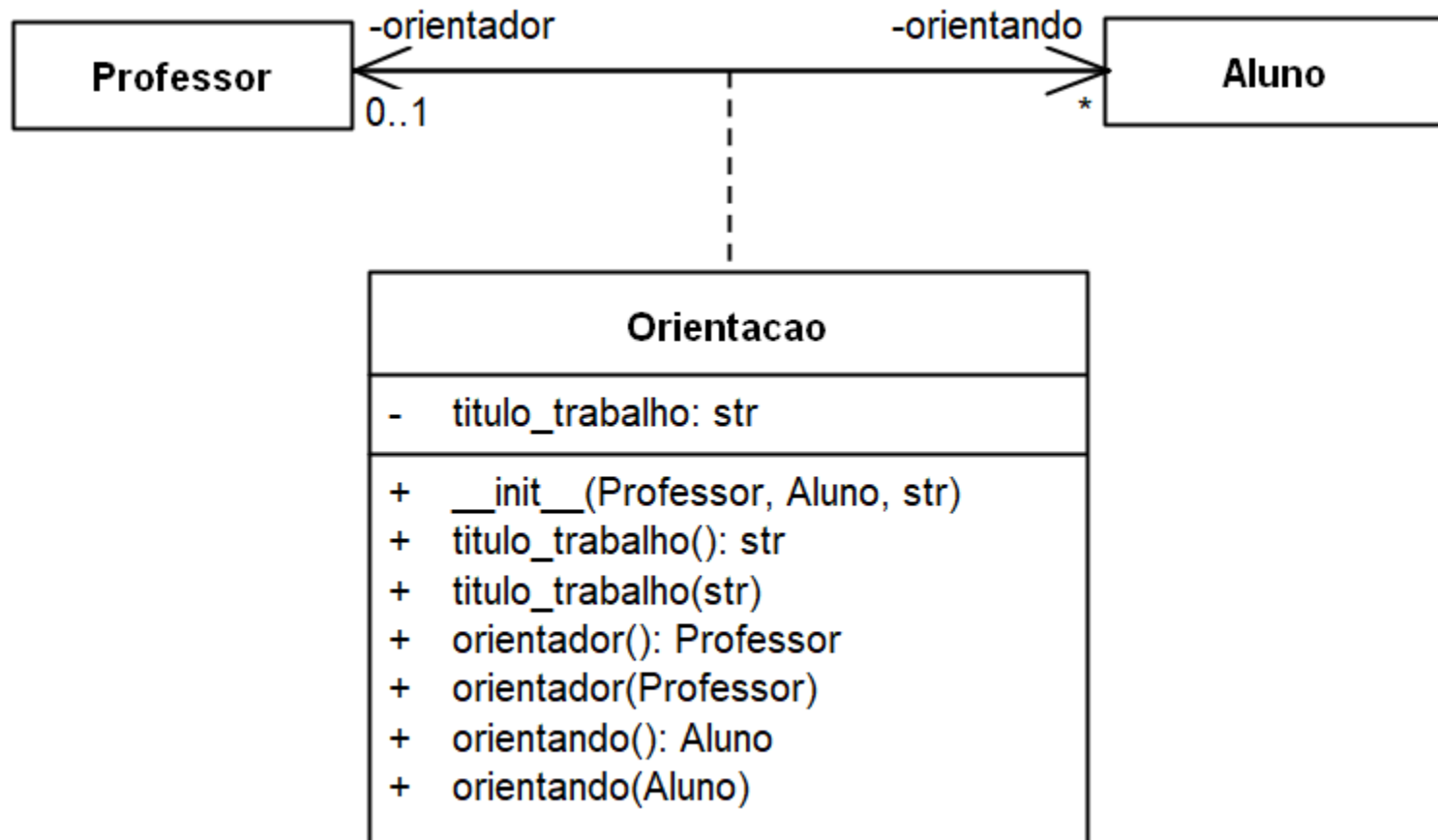


## Outro exemplo de Classe Associativa

- 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

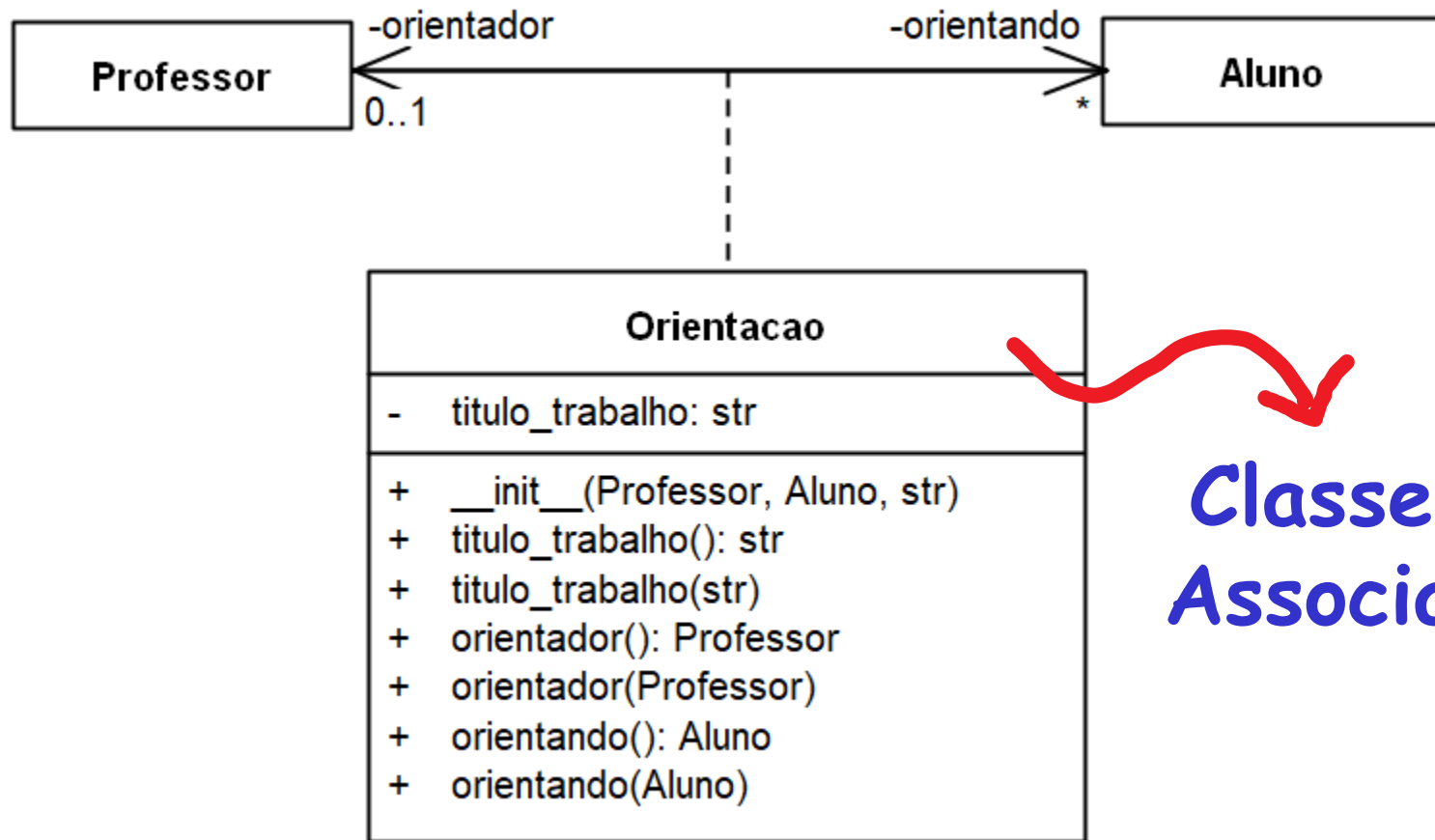
# Outro exemplo de Classe Associativa

Que tal considerarmos a associação bidirecional como uma classe?



## Outro exemplo de Classe Associativa

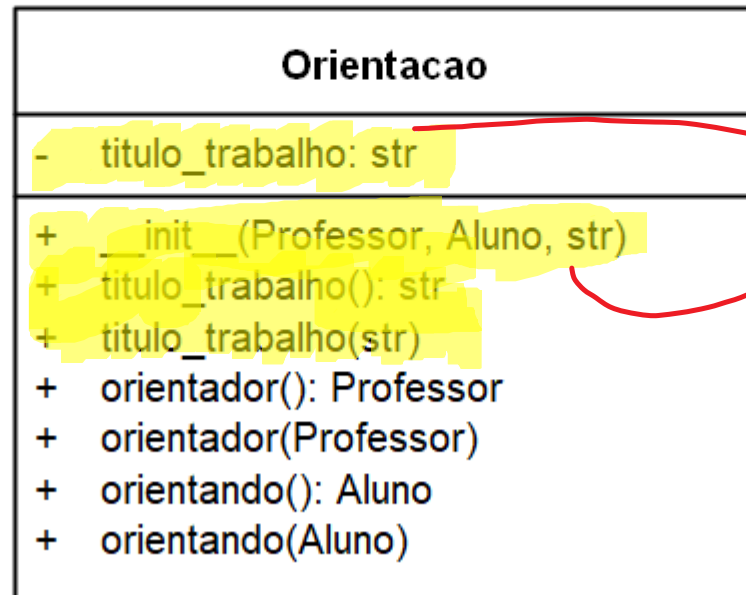
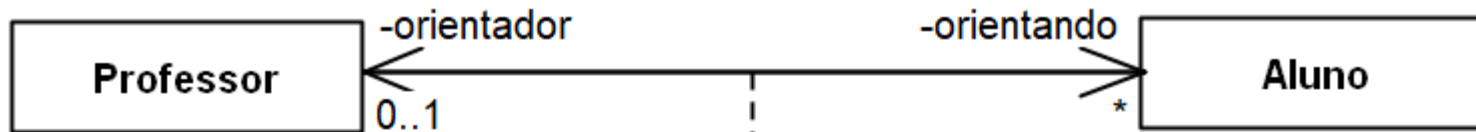
Que tal considerarmos a associação bidirecional como uma classe?



**Classe de  
Associação**

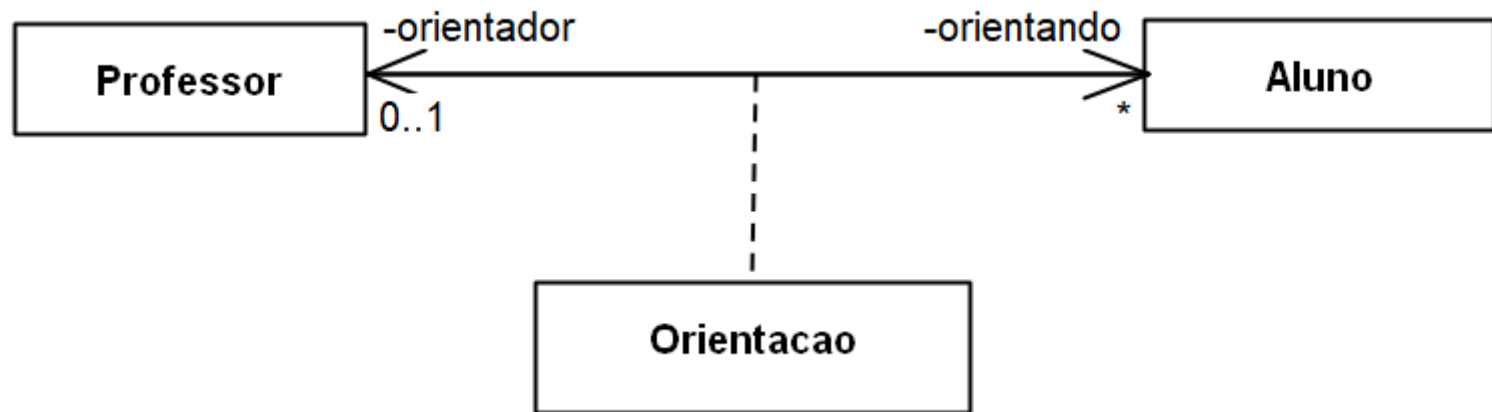


# Outro exemplo de Classe Associativa

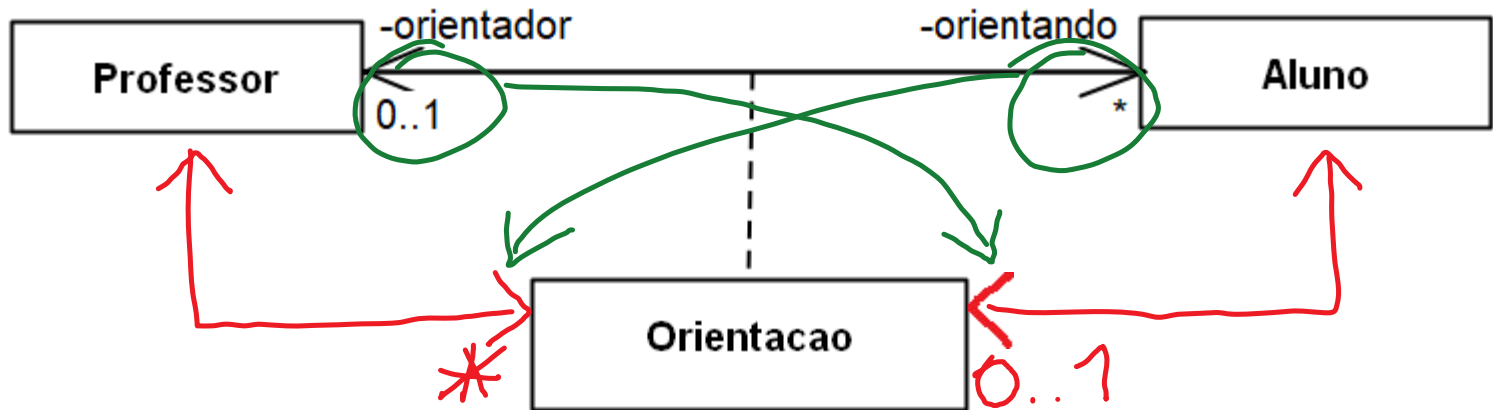


Atributos e operações específicas de Orientação

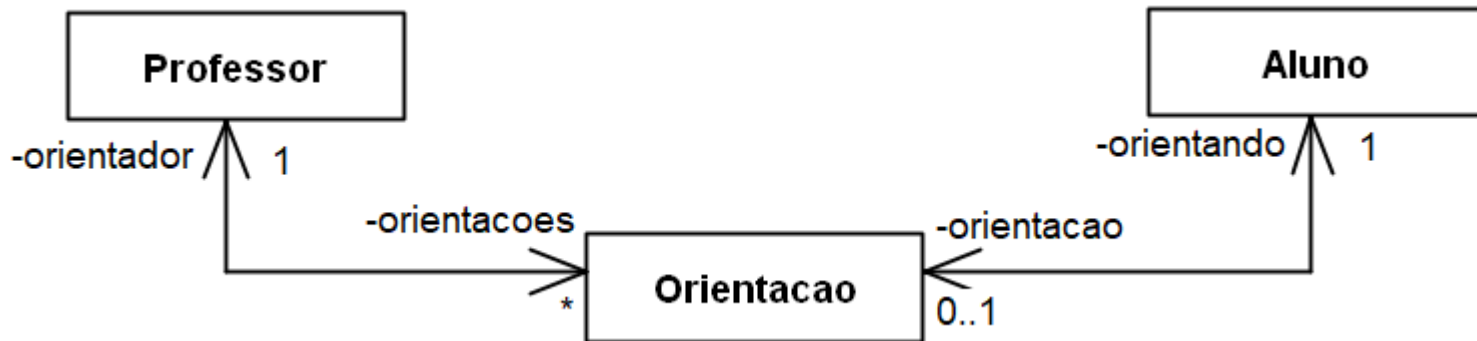
# Como implementar?



# 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: Professor):
        self.__orientador = orientador

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

    @orientando.setter
    def orientando(self, orientando: Aluno):
        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: Professor):  
    self.__orientador = orientador
```

```
@property
```

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

```
@orientando.setter
```

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

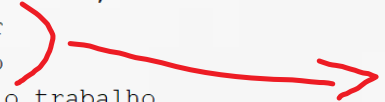
```
@property
```

```
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)...

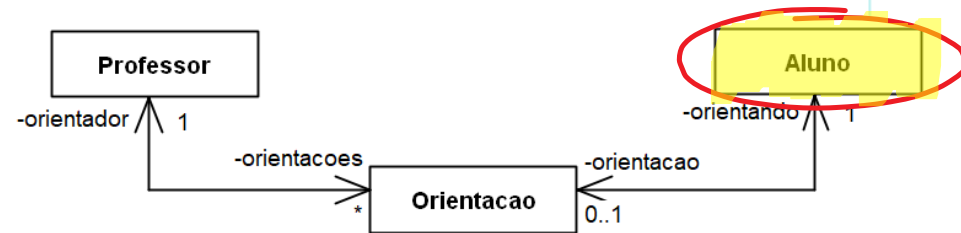
```
class Aluno:

    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

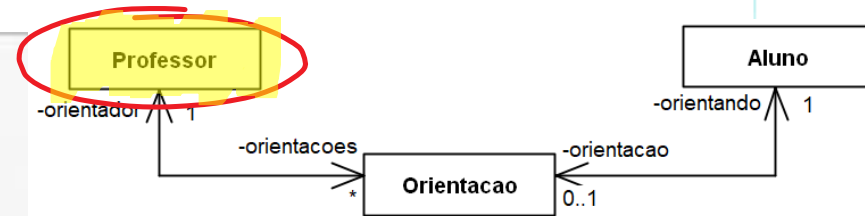
## No lado Professor (vários orientandos)...

```
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

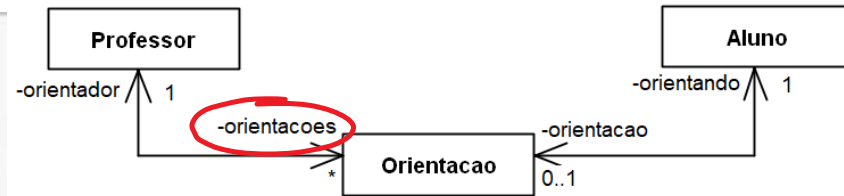
## No lado Professor (vários orientandos)...

```
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

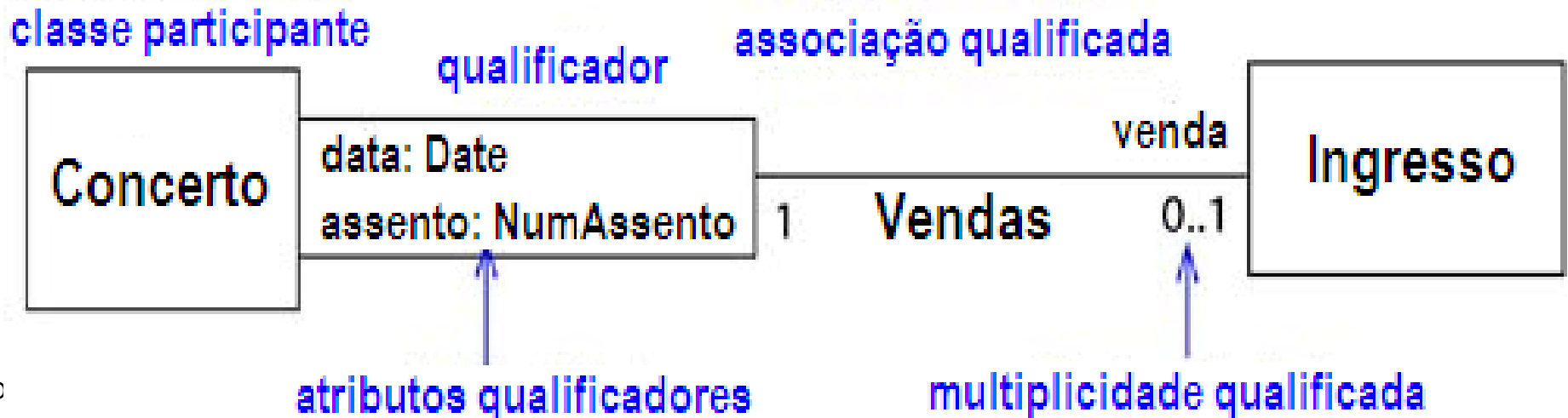
    @matricula.setter
    def matricula(self, matricula: str):
        self.__matricula = matricula

    ...
```

**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

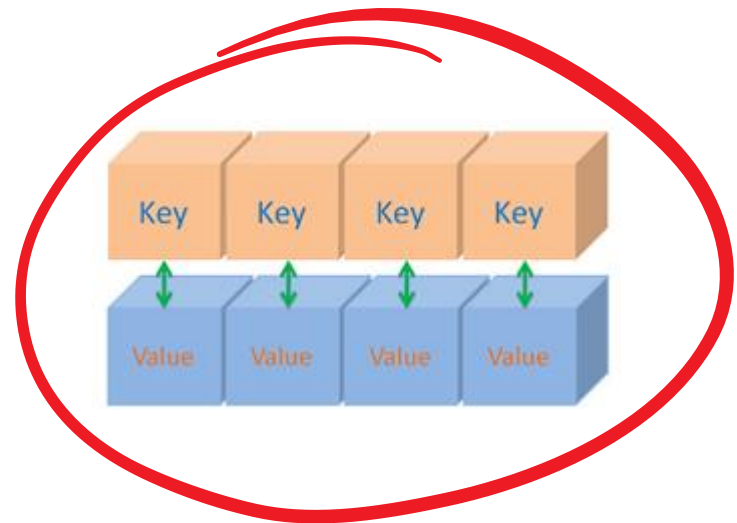
- É o equivalente na UML para o conceito na programação conhecido como **dicionários**, **hashmaps** ou **arrays associativos**
- No exemplo abaixo, o qualificador informa que na conexão com um **Pedido**, poderá haver um **ItemPedido** para cada instância de **Produto**



# 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]
```



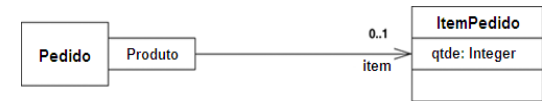
produto é  
a chave e  
**item\_pedido**  
é o valor

**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]
```

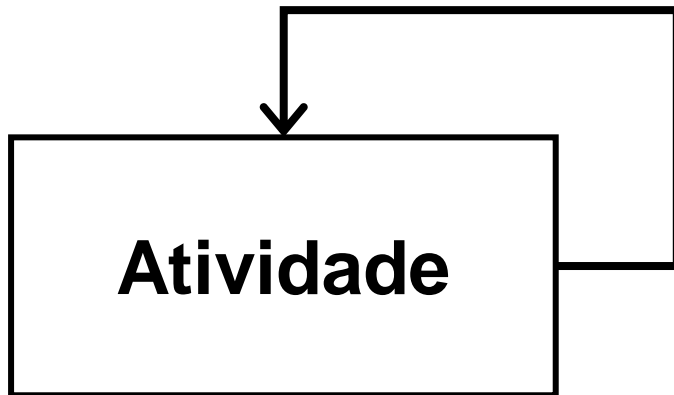


- ❑ Assim, todo acesso a um dado ItemPedido requer um Produto como um parâmetro, numa estrutura de dados baseada em chave/valor
- ❑ A multiplicidade no contexto do qualificador:  
um Pedido pode ter vários ItemPedido, mas apenas 0 ou 1 por Produto

# 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
```

# 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/Cummings 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.





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

### ***Você pode:***

- copiar, distribuir, exhibir 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.