

Descritores & metaclasses

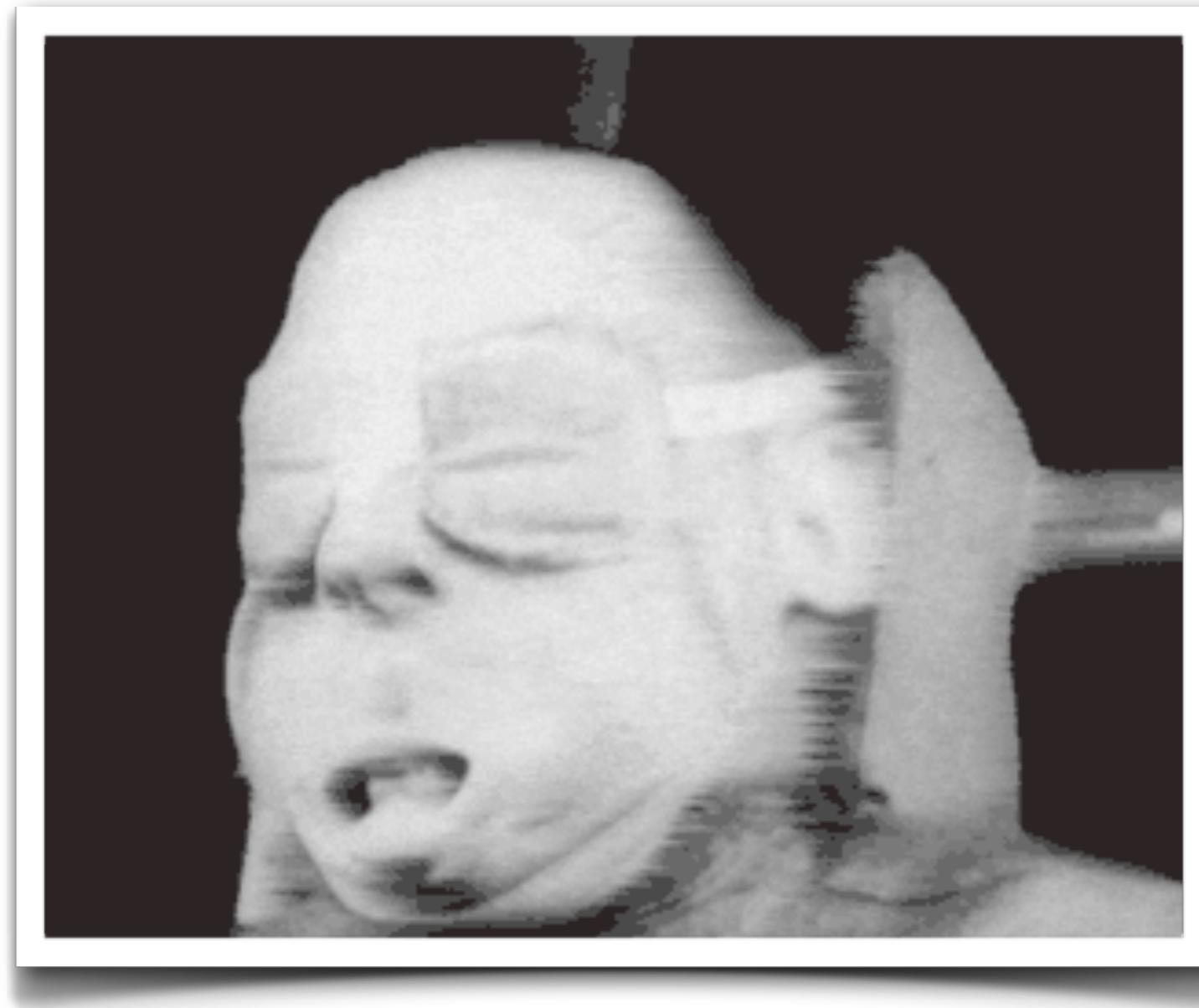
novembro/2013



Luciano Ramalho
ramalho@python.pro.br

@pythonprobr

Ritmo desta aula



Recomendação:
manter os olhos abertos

Pré-requisitos

- Para acompanhar os slides a seguir, é preciso saber como funciona o básico de orientação a objetos em Python. Especificamente:
 - contraste entre atributos de classe e de instância
 - herança de atributos de classe (métodos e campos)
 - atributos protegidos: como e quando usar
 - como e quando usar a função **super**

Roteiro da aula

- A partir de um cenário inicial, implementamos uma classe muito simples
- A partir daí, evoluímos a implementação em 6 etapas para controlar o acesso aos campos das instâncias, usando getters/setters, propriedades, descritores e finalmente uma metaclasses para abstrair toda a complexidade

O cenário

- Comércio de alimentos a granel
- Um pedido tem vários **itens**
- Cada **item** tem descrição, peso (kg), preço unitário (p/ kg) e sub-total

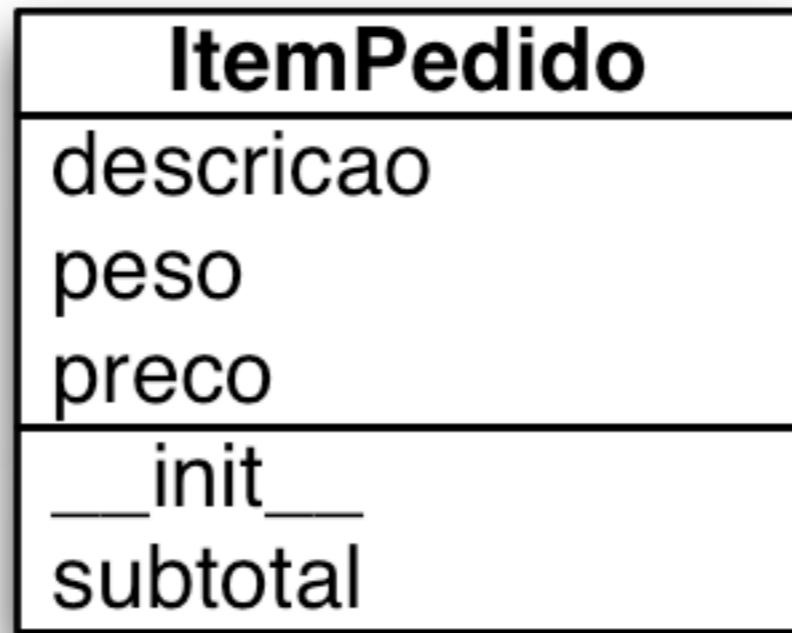
The image shows a blank invoice template from Formville.com. It includes fields for 'Your Company Name' (with sub-fields for Address, City, State, ZIP, and Phone Number), 'Bill To' (with a placeholder for a company name), and dates (Invoice Date, Invoice No., Date Due). Below these are sections for 'Description', 'Price', 'Quantity', and 'Extension'. At the bottom, there are fields for Subtotal, Tax, and Total Due.

Find Free Forms at Formville.com



1

1 mais simples, impossível

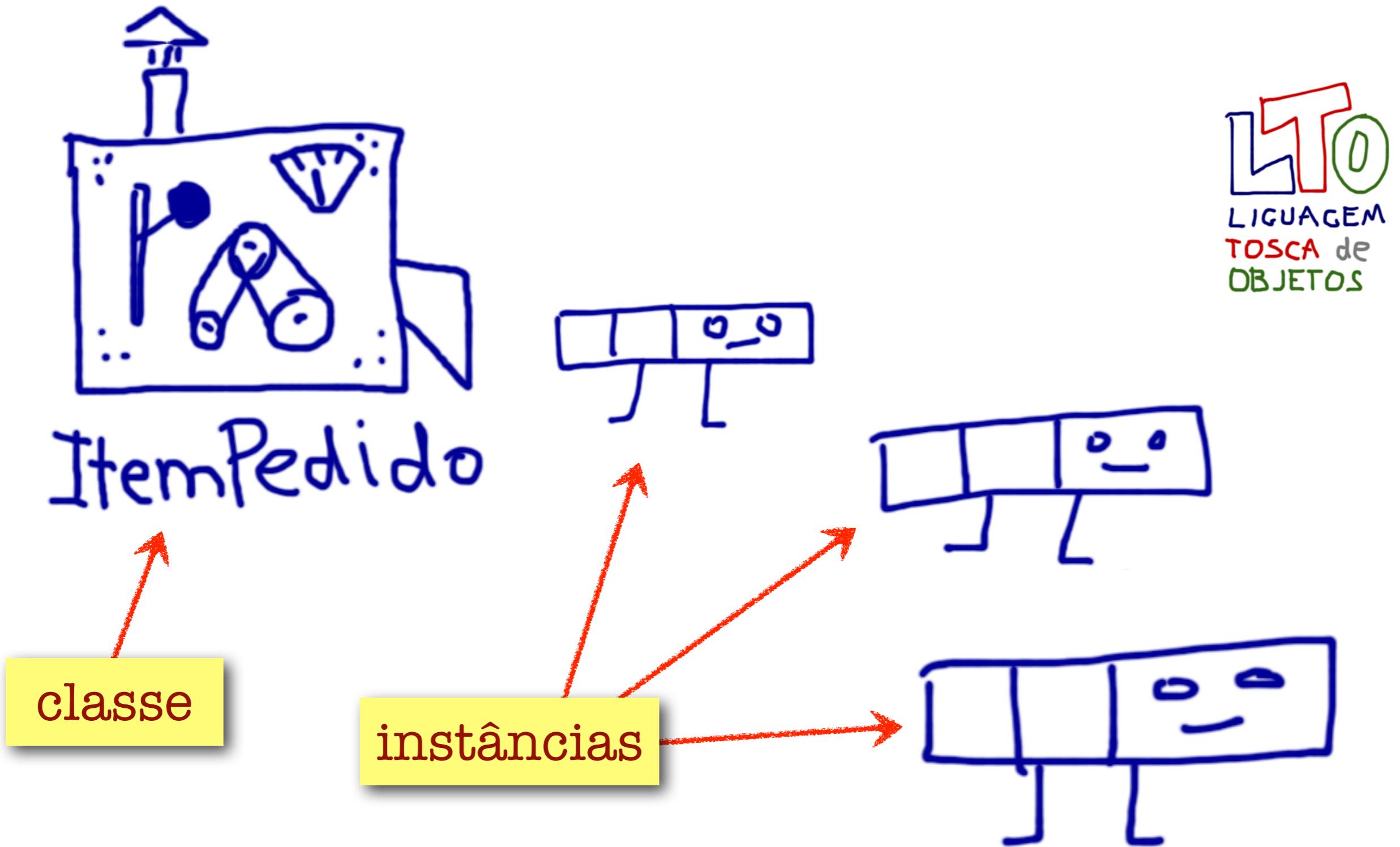


```
class ItemPedido(object):

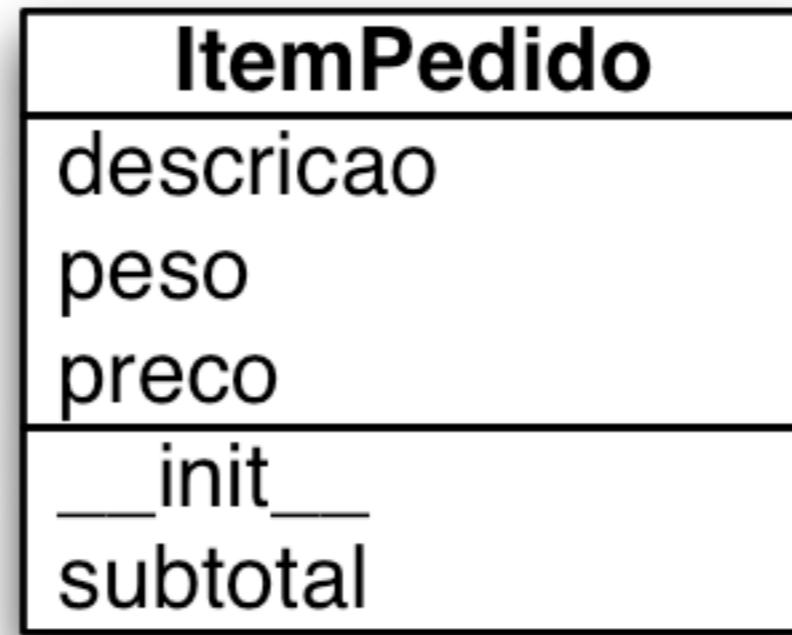
    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco
```

1 a classe produz instâncias



1 mais simples, impossível



```
class ItemPedido(object):  
  
    def __init__(self, descricao, peso, preco):  
        self.descricao = descricao  
        self.peso = peso  
        self.preco = preco  
  
    def subtotal(self):  
        return self.peso * self.preco
```

o método inicializador é conhecido como “dunder init”

1 porém, simples demais

```
>>> ervilha = ItemPedido('ervilha partida', .5, 7.95)
>>> ervilha.descricao, ervilha.peso, ervilha.preco
('ervilha partida', .5, 7.95)
>>> ervilha.peso = -10
>>> ervilha.subtotal()
-79.5
```

isso vai dar
problema na
hora de cobrar...



1 porém, simples demais

```
>>> ervilha = ItemPedido('ervilha partida', .5, 7.95)
>>> ervilha.descricao, ervilha.peso, ervilha.preco
('ervilha partida', .5, 7.95)
>>> ervilha.peso = -10
>>> ervilha.subtotal()
-79.5
```

isso vai dar
problema na
hora de cobrar...

“We found that customers could
order a **negative quantity** of books!
And we would credit their credit
card with the price...” *Jeff Bezos*



Jeff Bezos of Amazon: Birth of a Salesman
WSJ.com - <http://j.mp/VZ5not>

1 porém, simples demais

```
>>> ervilha = ItemPedido('ervilha partida', .5, 7.95)
>>> ervilha.descricao, ervilha.peso, ervilha.preco
('ervilha partida', .5, 7.95)
>>> ervilha.peso = -10
>>> ervilha.subtotal()
-79.5
```

isso vai dar
problema na
hora de cobrar...

“Descobrimos que os clientes conseguiam encomendar uma **quantidade negativa** de livros! E nós creditávamos o valor em seus cartões...” *Jeff Bezos*



Jeff Bezos of Amazon: Birth of a Salesman
WSJ.com - <http://j.mp/VZ5not>



@pythonprobr

2 validação com property

```
>>> ervilha = ItemPedido('ervilha partida', .5, 7.95)
>>> ervilha.descricao, ervilha.peso, ervilha.preco
('ervilha partida', .5, 7.95)
>>> ervilha.peso = -10
Traceback (most recent call last):
...
ValueError: valor deve ser > 0
```

parece uma
violação de
encapsulamento

mas a lógica do negócio
está preservada:
peso agora é uma **property**

ItemPedido
descricao
peso {property}
preco
__init__
subtotal

2 implementar property

```
class ItemPedido(object):

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco

    @property
    def peso(self):
        return self.__peso

    @peso.setter
    def peso(self, valor):
        if valor > 0:
            self.__peso = valor
        else:
            raise ValueError('valor deve ser > 0')
```

ItemPedido
descricao
__peso
preco
__init__
subtotal
peso {prop. get}
peso {prop. set}

2 implementar property

```
class ItemPedido(object):

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco

    @property
    def peso(self):
        return self.__peso

    @peso.setter
    def peso(self, valor):
        if valor > 0:
            self.__peso = valor
        else:
            raise ValueError('valor deve ser > 0')
```

ItemPedido
descricao
__peso
preco
__init__
subtotal
peso {prop. get}
peso {prop. set}

atributo
protegido

2 implementar property

```
class ItemPedido(object):

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco

    @property
    def peso(self):
        return self.__peso

    @peso.setter
    def peso(self, valor):
        if valor > 0:
            self.__peso = valor
        else:
            raise ValueError('valor deve ser > 0')
```

no `__init__` a
`property` já
está em uso

2 implementar property

```
class ItemPedido(object):

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco

    @property
    def peso(self):
        return self.__peso

    @peso.setter
    def peso(self, valor):
        if valor > 0:
            self.__peso = valor
        else:
            raise ValueError('valor deve ser > 0')
```

o atributo
protegido
peso só é
acessado nos
métodos da
property

2 implementar property

```
class ItemPedido(object):

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco

@property
def peso(self):
    return self.__peso

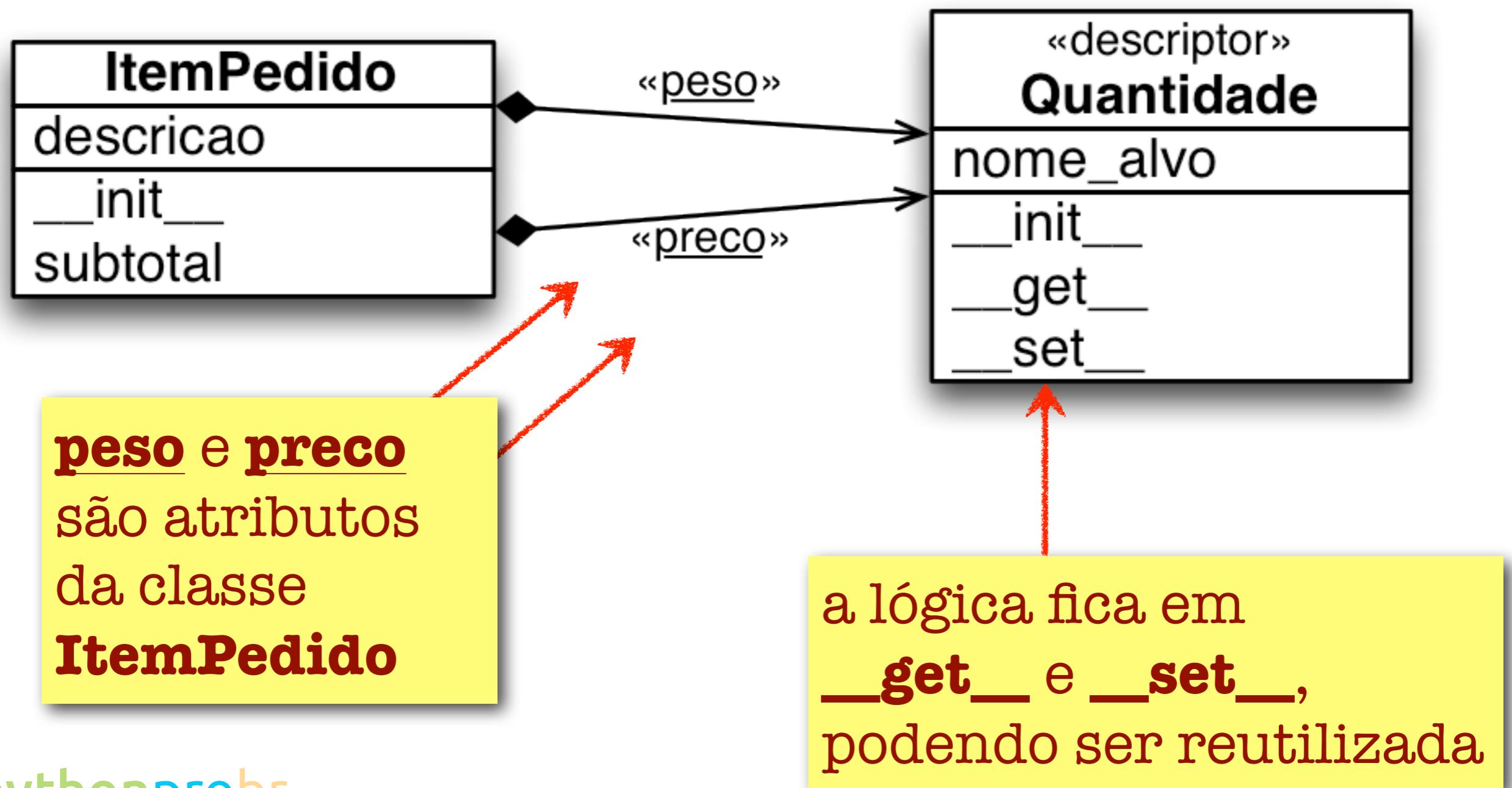
@peso.setter
def peso(self, valor):
    if valor > 0:
        self.__peso = valor
    else:
        raise ValueError('valor deve ser > 0')
```

e se quisermos
a mesma lógica
para o **preco**?

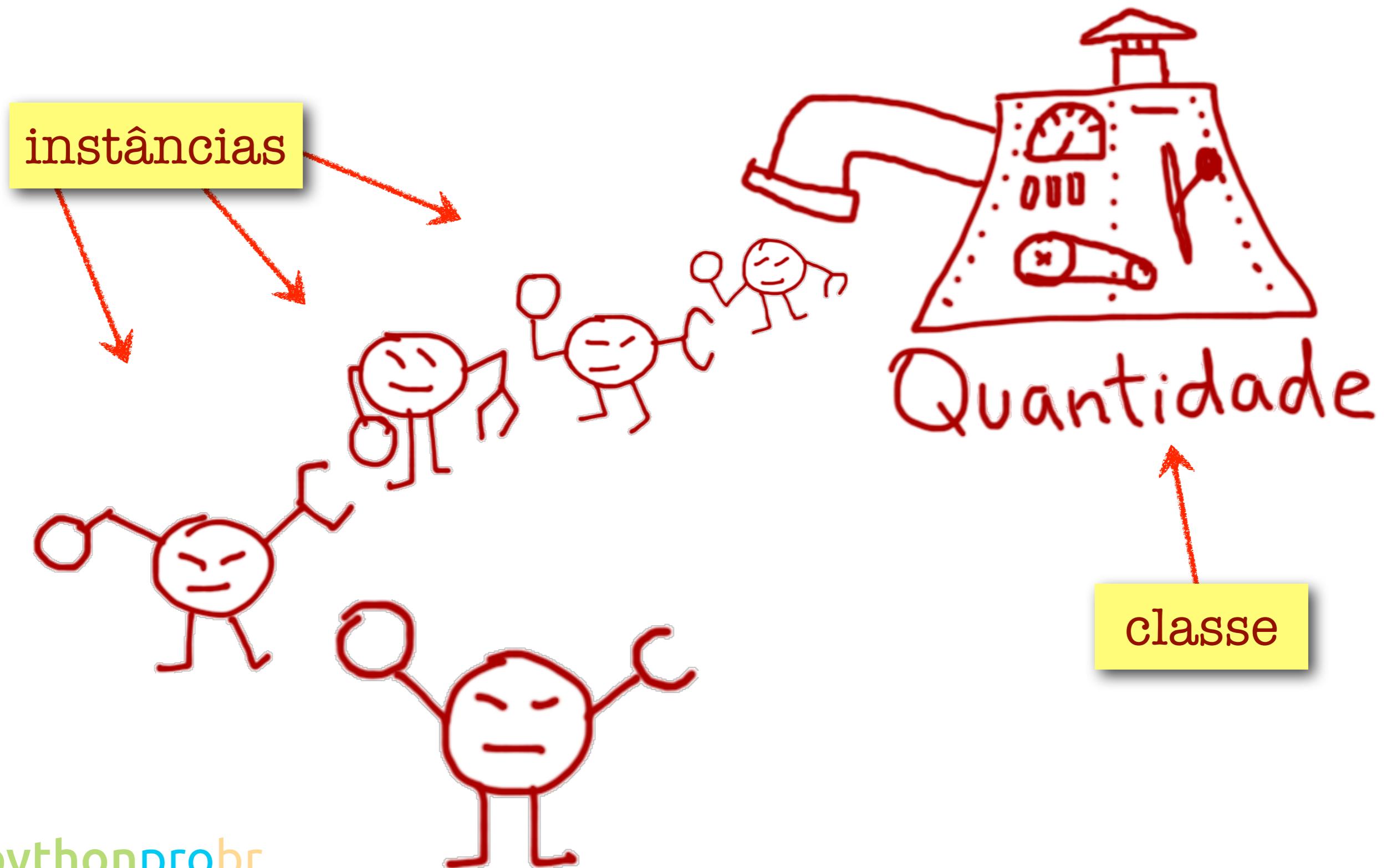
teremos que
duplicar tudo
isso?



3 validação com descriptor



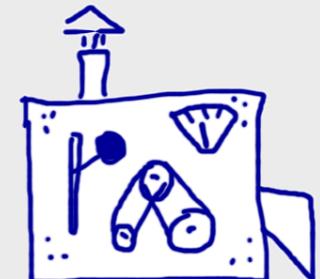
③ validação com descriptor



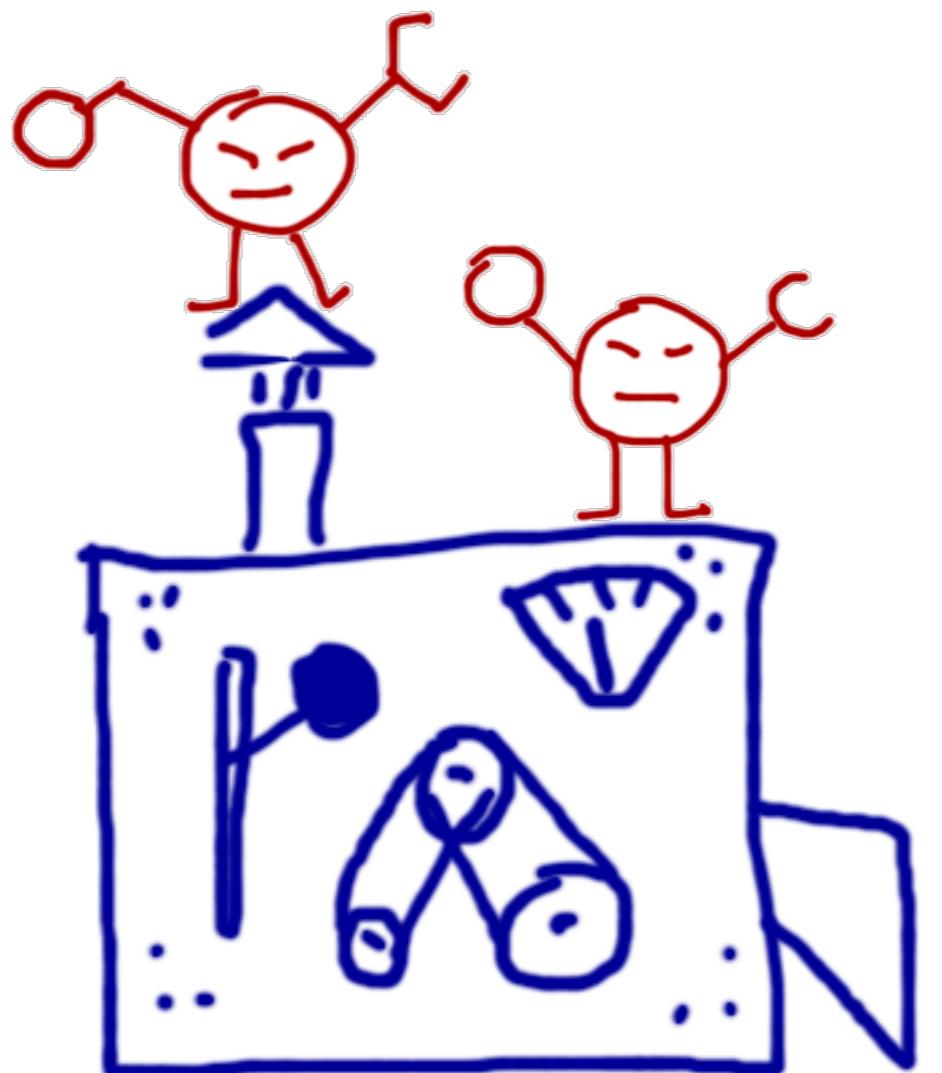
3

implementação do descriptor

```
class Quantidade(object):  
  
    def __init__(self):  
        prefixo = self.__class__.__name__ Quantidade  
        chave = id(self)  
        self.nome_alvo = '%s_%s' % (prefixo, chave)  
  
    def __get__(self, instance, owner):  
        return getattr(instance, self.nome_alvo)  
  
    def __set__(self, instance, value):  
        if value > 0:  
            setattr(instance, self.nome_alvo, value)  
        else:  
            raise ValueError('valor deve ser > 0')  
  
class ItemPedido(object):  
    peso = Quantidade()  
    preco = Quantidade()  
  
    def __init__(self, descricao, peso, preco):  
        self.descricao = descricao  
        self.peso = peso  
        self.preco = preco  
  
    def subtotal(self):  
        return self.peso * self.preco
```



③ uso do descriptor



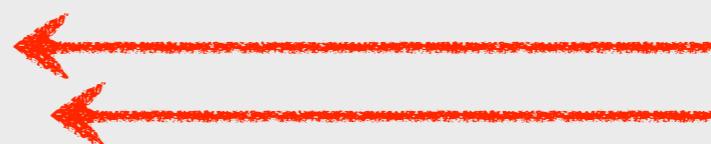
a classe
ItemPedido
tem duas
instâncias de
Quantidade
associadas a ela

3 uso do descriptor

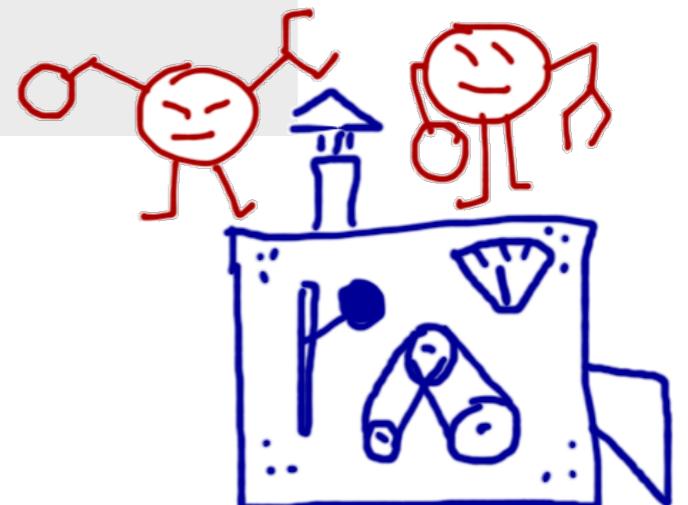
```
class ItemPedido(object):
    peso = Quantidade()
    preco = Quantidade()

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco
```



a classe
ItemPedido
tem duas
instâncias de
Quantidade
associadas a ela



3 uso do descriptor

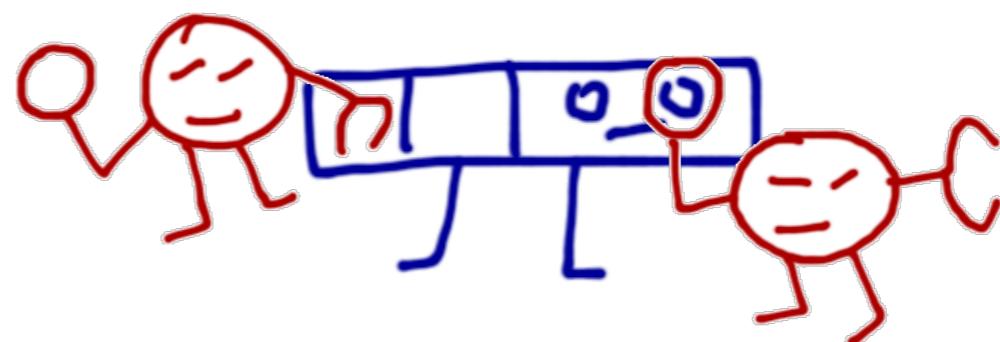
```
class ItemPedido(object):
    peso = Quantidade()
    preco = Quantidade()

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco
```



cada instância
da classe
Quantidade
controla um
atributo de
ItemPedido



3 implementar o descriptor

```
class Quantidade(object):  
  
    def __init__(self):  
        prefixo = self.__class__.__name__  
        chave = id(self)  
        self.nome_alvo = '%s_%s' % (prefixo, chave)  
  
    def __get__(self, instance, owner):  
        return getattr(instance, self.nome_alvo)  
  
    def __set__(self, instance, value):  
        if value > 0:  
            setattr(instance, self.nome_alvo, value)  
        else:  
            raise ValueError('valor deve ser > 0')
```



uma classe
com método
get é um
descriptor

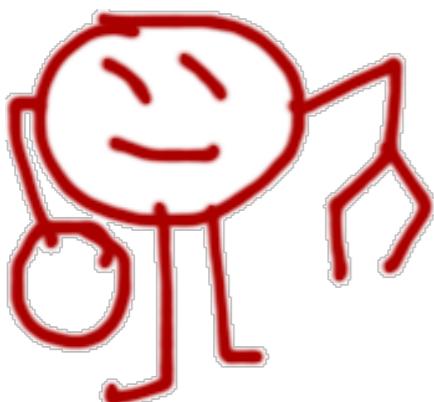
3 implementar o descriptor

```
class Quantidade(object):

    def __init__(self):
        prefixo = self.__class__.__name__
        chave = id(self)
        self.nome_alvo = '%s_%s' % (prefixo, chave)

    def __get__(self, instance, owner):
        return getattr(instance, self.nome_alvo)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.nome_alvo, value)
        else:
            raise ValueError('valor deve ser > 0')
```



self é a instância
do descriptor (associada
ao preço ou ao peso)

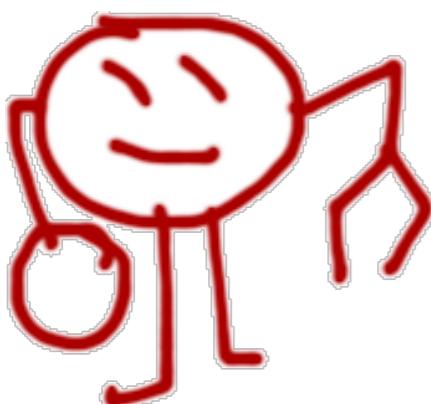
3 implementar o descriptor

```
class Quantidade(object):

    def __init__(self):
        prefixo = self.__class__.__name__
        chave = id(self)
        self.nome_alvo = '%s_%s' % (prefixo, chave)

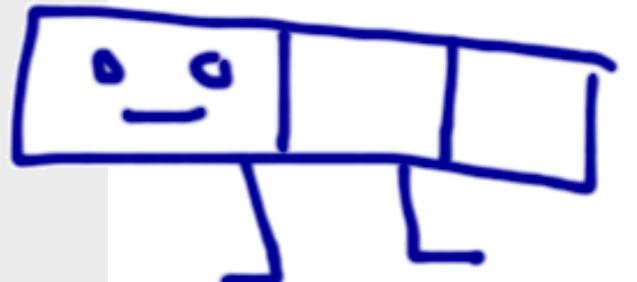
    def __get__(self, instance, owner):
        return getattr(instance, self.nome_alvo)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.nome_alvo, value)
        else:
            raise ValueError('valor deve ser maior que zero')
```



self é a instância do descriptor (associada ao preço ou ao peso)

instance é a instância de **ItemPedido** que está sendo acessada



3 implementar o descriptor

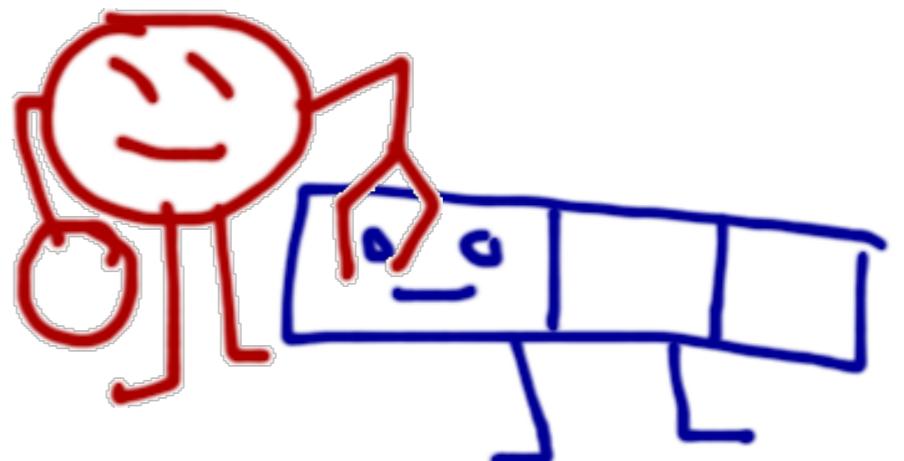
```
class Quantidade(object):

    def __init__(self):
        prefixo = self.__class__.__name__
        chave = id(self)
        self.nome_alvo = '%s_%s' % (prefixo, chave)

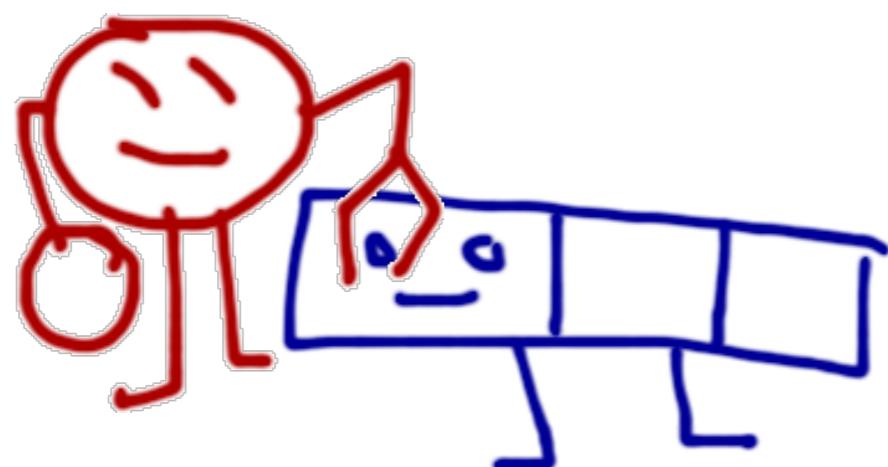
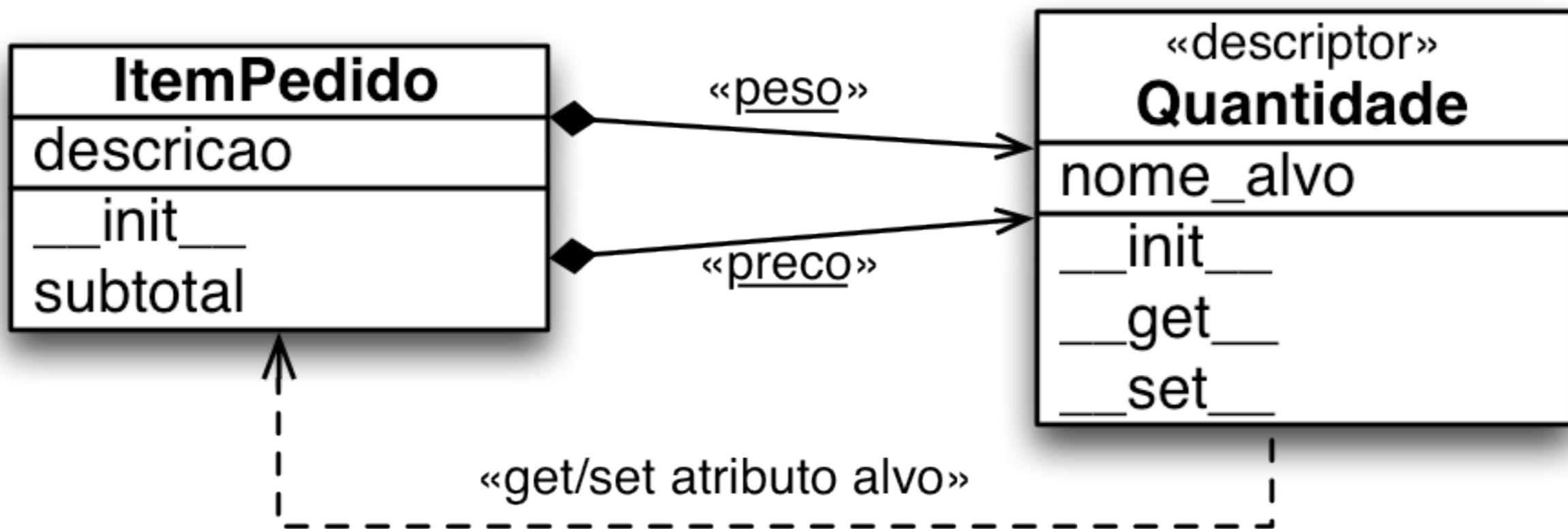
    def __get__(self, instance, owner):
        return getattr(instance, self.nome_alvo)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.nome_alvo, value)
        else:
            raise ValueError('valor deve ser > 0')
```

nome_alvo é
o nome do
atributo da
instância de
ItemPedido
que este
descritor
(self)
controla



3 implementar o descriptor



get e **set**
manipulam o atributo-alvo
no objeto **ItemPedido**

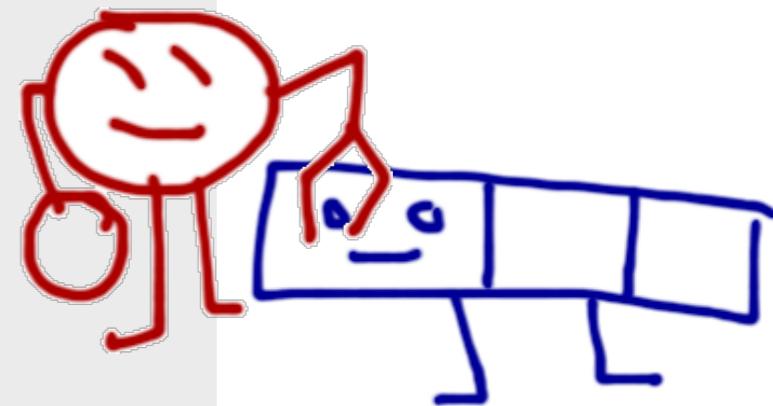
3 implementar o descriptor

```
class Quantidade(object):

    def __init__(self):
        prefixo = self.__class__.__name__
        chave = id(self)
        self.nome_alvo = '%s_%s' % (prefixo, chave)

    def __get__(self, instance, owner):
        return getattr(instance, self.nome_alvo)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.nome_alvo, value)
        else:
            raise ValueError('valor deve ser > 0')
```



__get__ e **__set__** usam
getattr e **setattr** para manipular o
atributo-alvo na instância de **ItemPedido**

3 inicialização do descriptor

```
class ItemPedido(object):
    peso = Quantidade()
    preco = Quantidade()

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco
```

quando um descriptor é instanciado, o atributo ao qual ele será vinculado ainda não existe!

exemplo: o atributo **preco** só passa a existir após a atribuição

3 implementar o descriptor

```
class Quantidade(object):

    def __init__(self):
        prefixo = self.__class__.__name__
        chave = id(self)
        self.nome_alvo = '%s_%s' % (prefixo, chave)

    def __get__(self, instance, owner):
        return getattr(instance, self.nome_alvo)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.nome_alvo, value)
        else:
            raise ValueError('valor deve ser maior que zero')

    def __delete__(self, instance):
        delattr(instance, self.nome_alvo)
```

temos que inventar um nome para o atributo-alvo onde será armazenado o valor na instância de **ItemPedido**

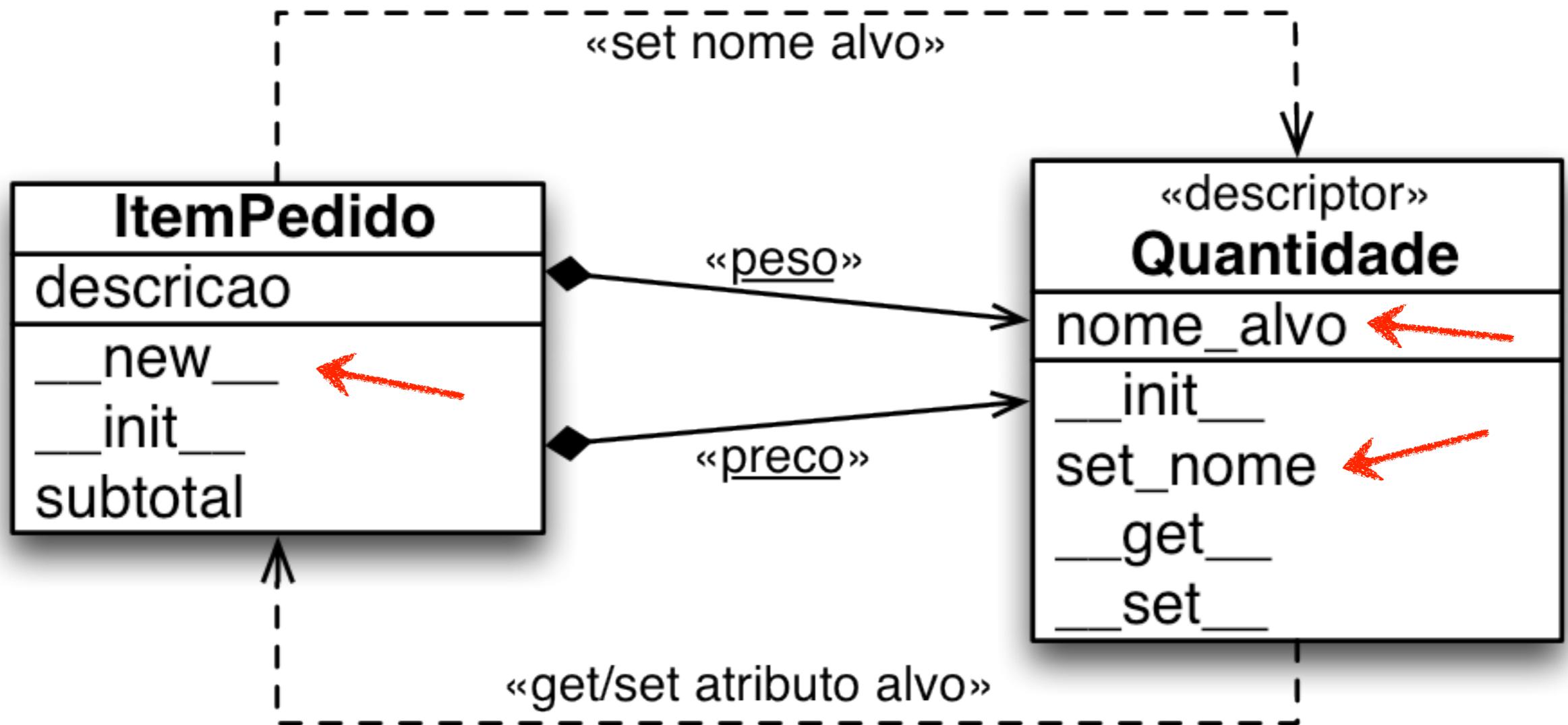
3 implementar o descriptor

```
>>> ervilha = ItemPedido('ervilha partida', .5, 3.95)
>>> ervilha.descricao, ervilha.peso, ervilha.preco
('ervilha partida', .5, 3.95)
>>> dir(ervilha)
['Quantidade_4299545872', 'Quantidade_4299546064',
 '__class__', '__delattr__', '__dict__', '__doc__',
 '__format__', '__getattribute__', '__hash__',
 '__init__', '__module__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', 'descricao', 'peso', 'preco',
 'subtotal']
```

nesta implementação, os nomes dos atributos-alvo não são descritivos, dificultando a depuração



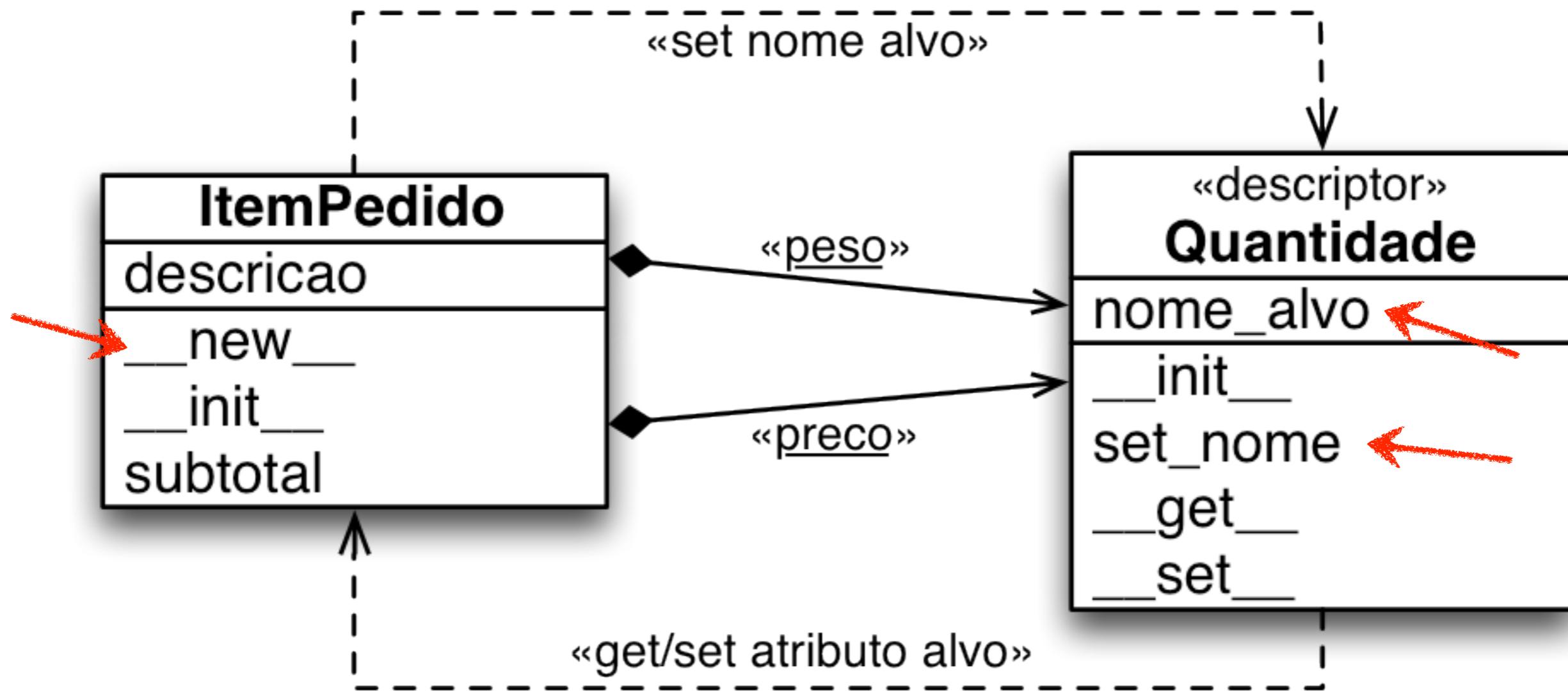
4 usar nomes descritivos



ItemPedido.__new__ invoca
«**quantidade**».set_nome
para redefinir o **nome_alvo**

4 usar nomes descritivos

ItemPedido.__new__ invoca «**quantidade».set_nome**



«quantidade».set_nome
redefine o **nome_alvo**

4

usar nomes descritivos

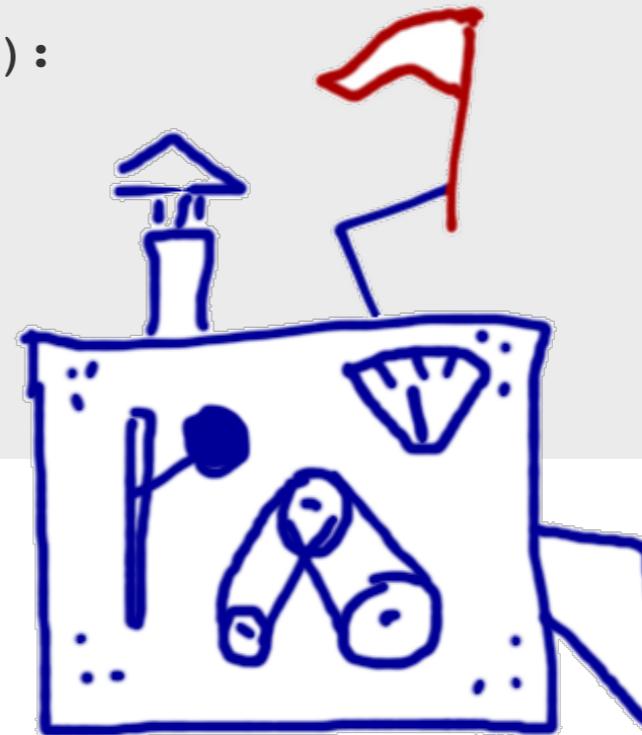
```
class ItemPedido(object):
    peso = Quantidade()
    preco = Quantidade()

    def __new__(cls, *args, **kwargs):
        for chave, atr in cls.__dict__.items():
            if hasattr(atr, 'set_nome'):
                atr.set_nome('__' + cls.__name__, chave)
        return super(ItemPedido, cls).__new__(cls, *args, **kwargs)

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco
```

ItemPedido.__new__
invoca
«quantidade».set_nome



ItemPedido

4 usar nomes descritivos

```
class Quantidade(object):  
  
    def __init__(self):  
        self.set_nome(self.__class__.__name__, id(self))  
  
    def set_nome(self, prefix, key):  
        self.nome_alvo = '%s_%s' % (prefix, key)  
  
    def __get__(self, instance, owner):  
        return getattr(instance, self.nome_alvo)  
  
    def __set__(self, instance, value):  
        if value > 0:  
            setattr(instance, self.nome_alvo, value)  
        else:  
            raise ValueError('valor deve ser > 0')
```

«quantidade».set_nome
redefine o nome_alvo



4

usar nomes descritivos

```
class ItemPedido(object):
    peso = Quantidade()
    preco = Quantidade()

    def __new__(cls, *args, **kwargs):
        for chave, atr in cls.__dict__.items():
            if hasattr(atr, 'set_nome'):
                atr.set_nome('__' + cls.__name__, chave)
        return super(ItemPedido, cls).__new__(cls, *args, **kwargs)

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco
```

Quando `__init__` executa, o descritor já está configurado com um nome de atributo-alvo descritivo

4 usar nomes descritivos

ItemPedido.__new__ invoca «**quantidade**».set_nome



ItemPedido

4 usar nomes descritivos



ItemPedido

@pythonprobr

«quantidade».set_nome
redefine o **nome_alvo**

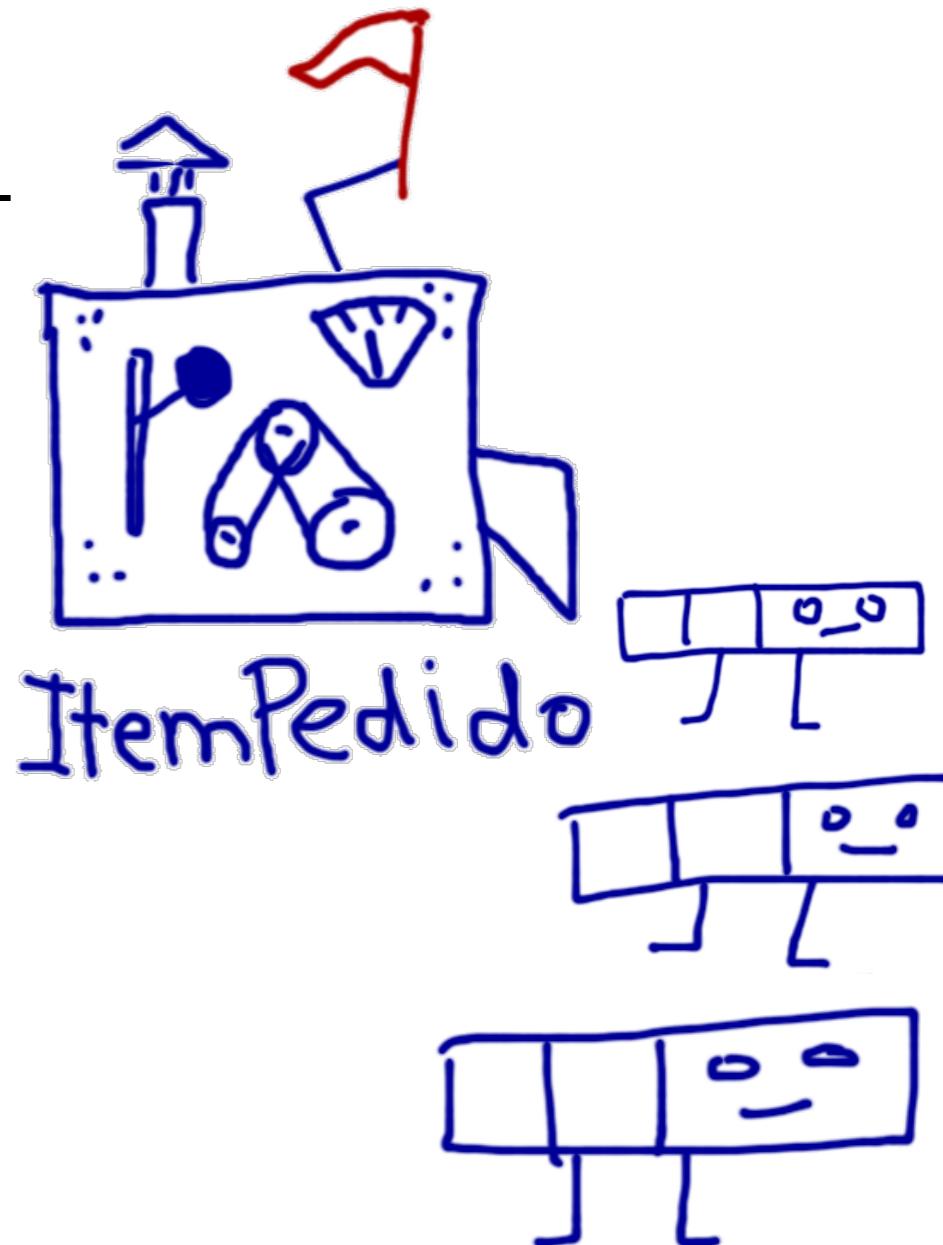
4 nomes descritivos

```
>>> ervilha = ItemPedido('ervilha partida', .5, 3.95)
>>> ervilha.descricao, ervilha.peso, ervilha.preco
('ervilha partida', 0.5, 3.95)
>>> dir(ervilha)
['__ItemPedido__peso', '__ItemPedido__preco',
 '__class__', '__delattr__', '__dict__', '__doc__',
 '__format__', '__getattribute__', '__hash__',
 '__init__', '__module__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', 'descricao', 'peso', 'preco',
 'subtotal']
```

nesta implementação e nas próximas, os nomes dos atributos-alvo seguem a convenção de atributos protegidos de Python

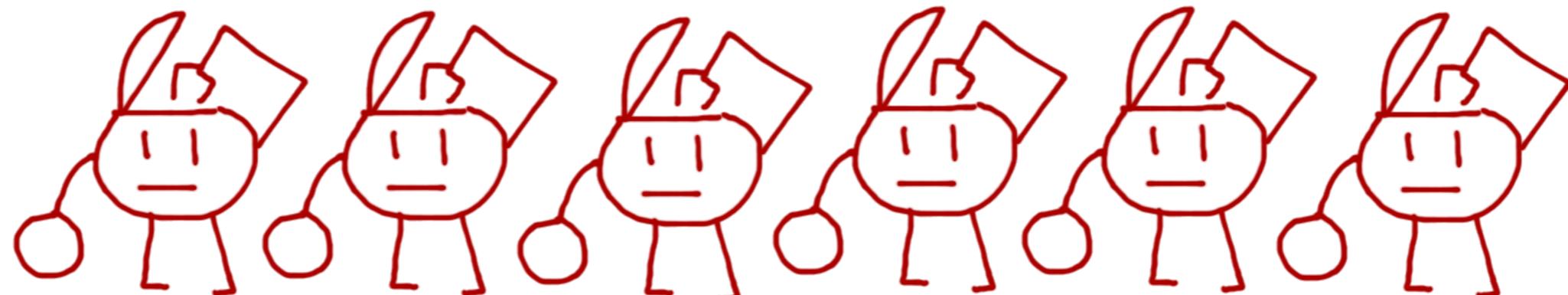
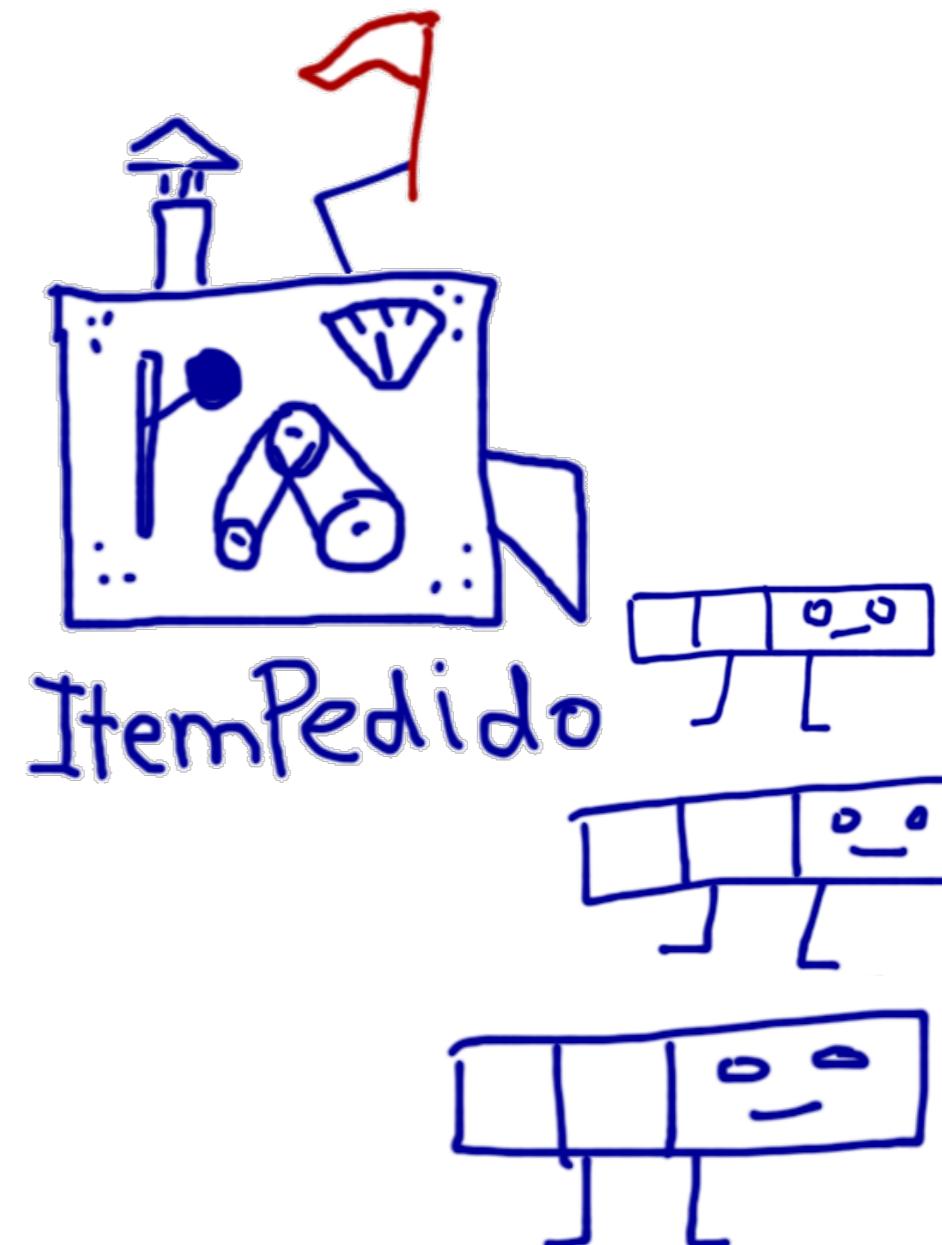
4 funciona, mas custa caro

- **ItemPedido** aciona `__new__` para construir cada nova instância
- Porém a associação dos descritores é com a classe **ItemPedido**: o nome do atributo-alvo nunca vai mudar, uma vez definido corretamente

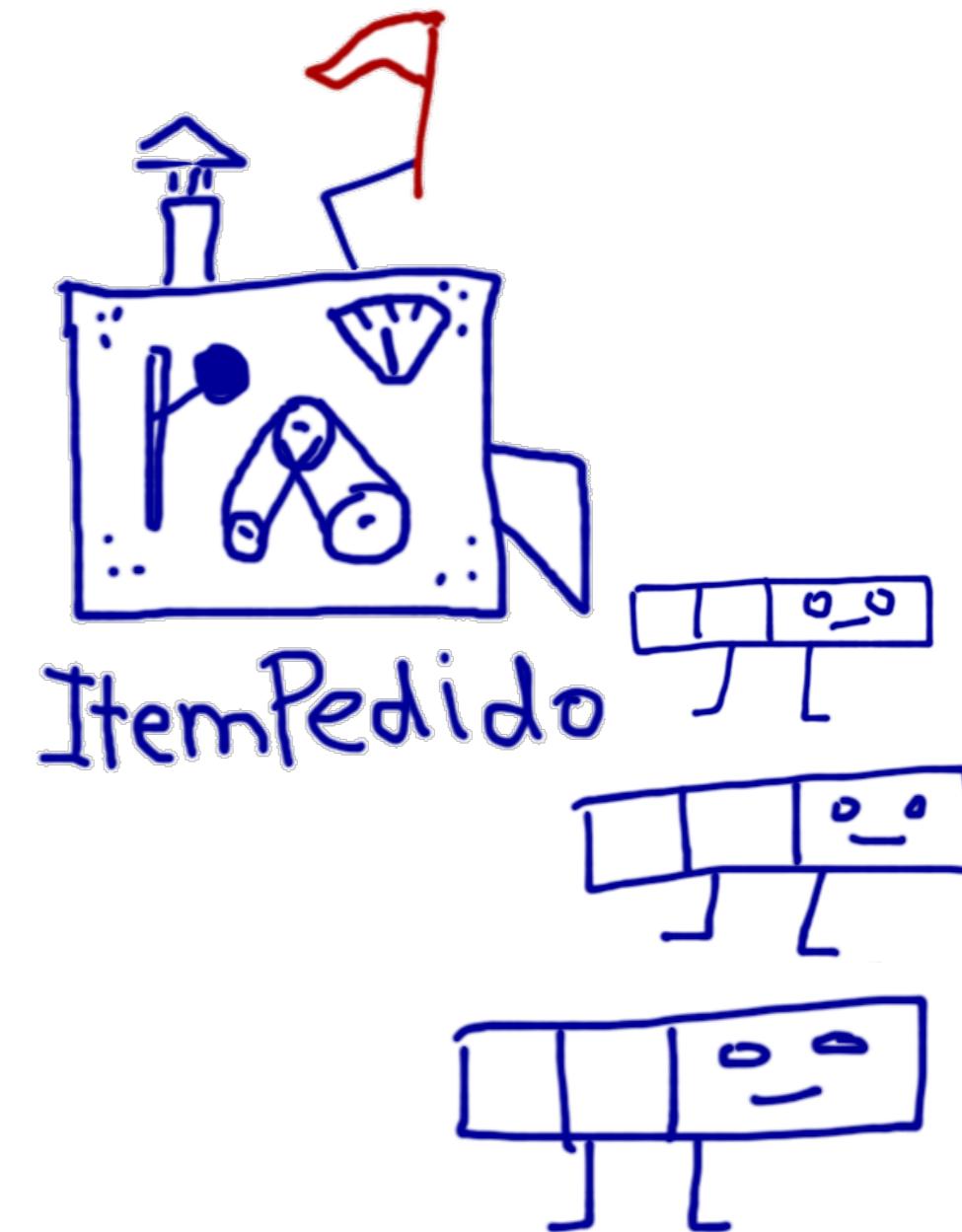
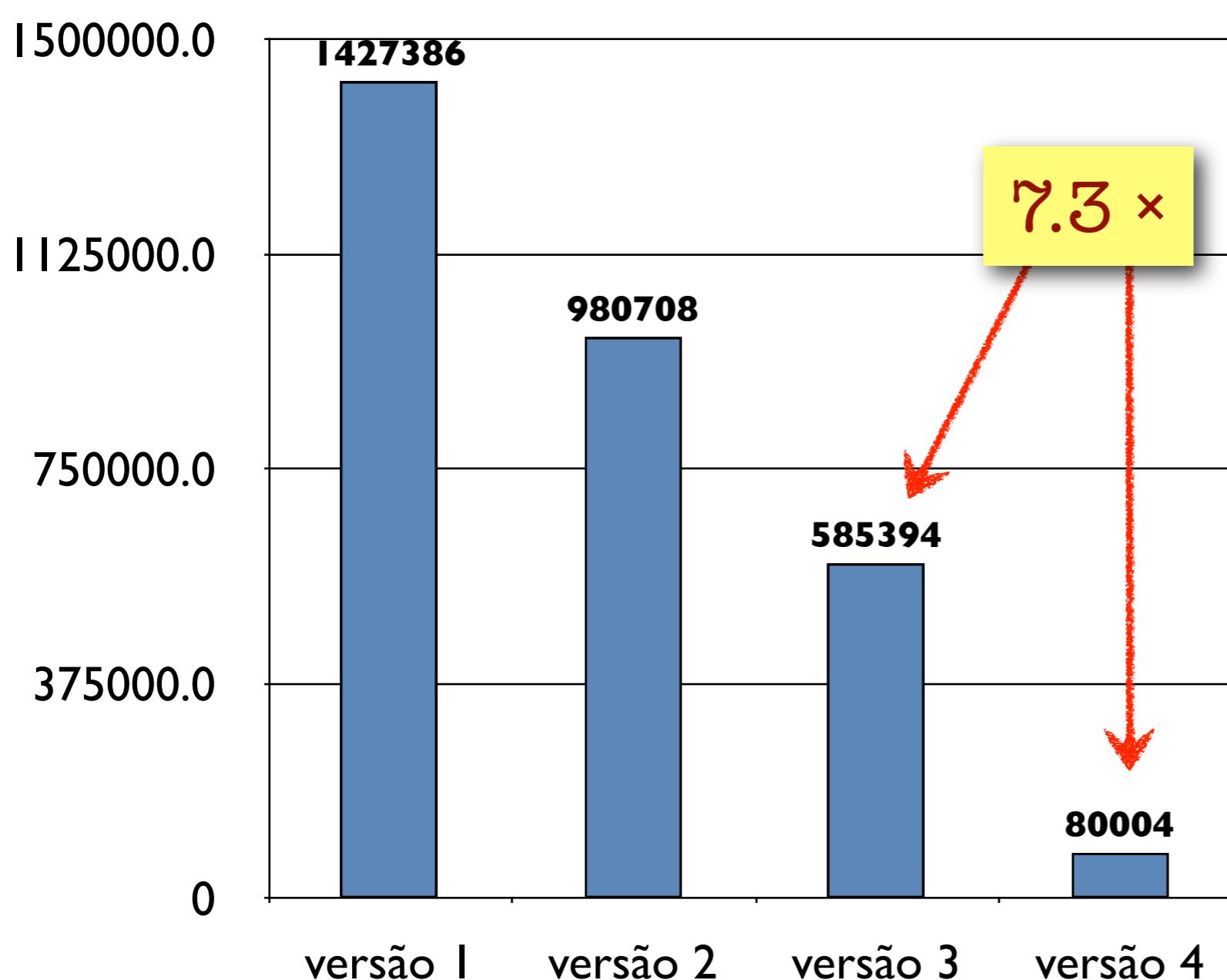


4 funciona, mas custa caro

- Isso significa que para cada nova instância de **ItemPedido** que é criada, **«quantidade».set_nome** é invocado duas vezes
- Mas o nome do atributo-alvo não tem porque mudar na vida de uma **«quantidade»**



4 funciona, mas custa caro



Número de instâncias de **ItemPedido** criadas por segundo (MacBook Pro 2011, Intel Core i7)

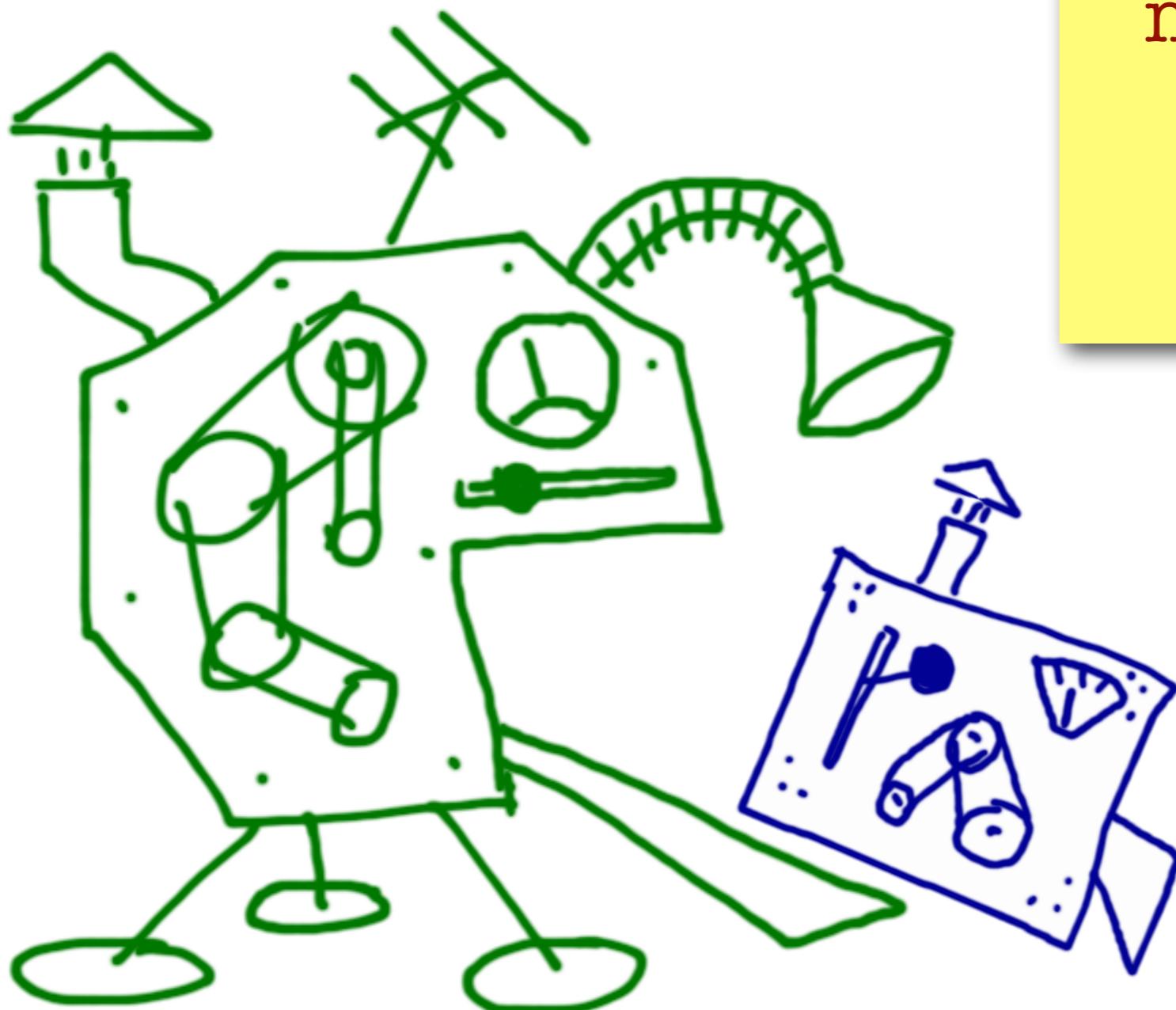


5 como evitar trabalho inútil

- **ItemPedido.__new__** resolveu mas de modo ineficiente.
- Cada «**quantidade**» deve receber o nome do seu atributo-alvo apenas uma vez, quando a própria classe **ItemPedido** for criada
- Para isso precisamos de uma...

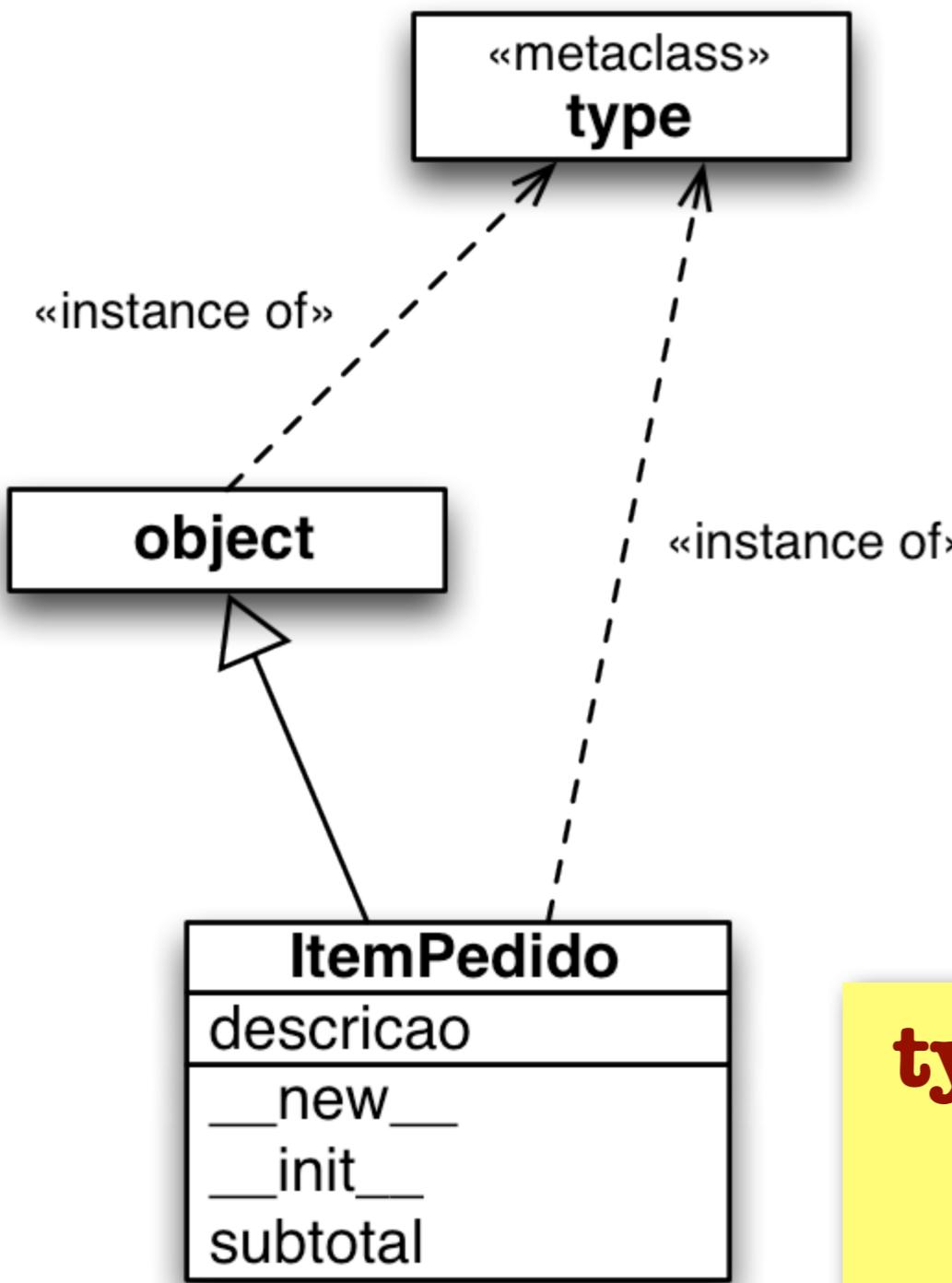
METACLASSE

5 metaclasses criam classes!



metaclasses são
classes cujas
instâncias são
classes

5 metaclasses criam classes!



metaclasses são classes cujas instâncias são classes

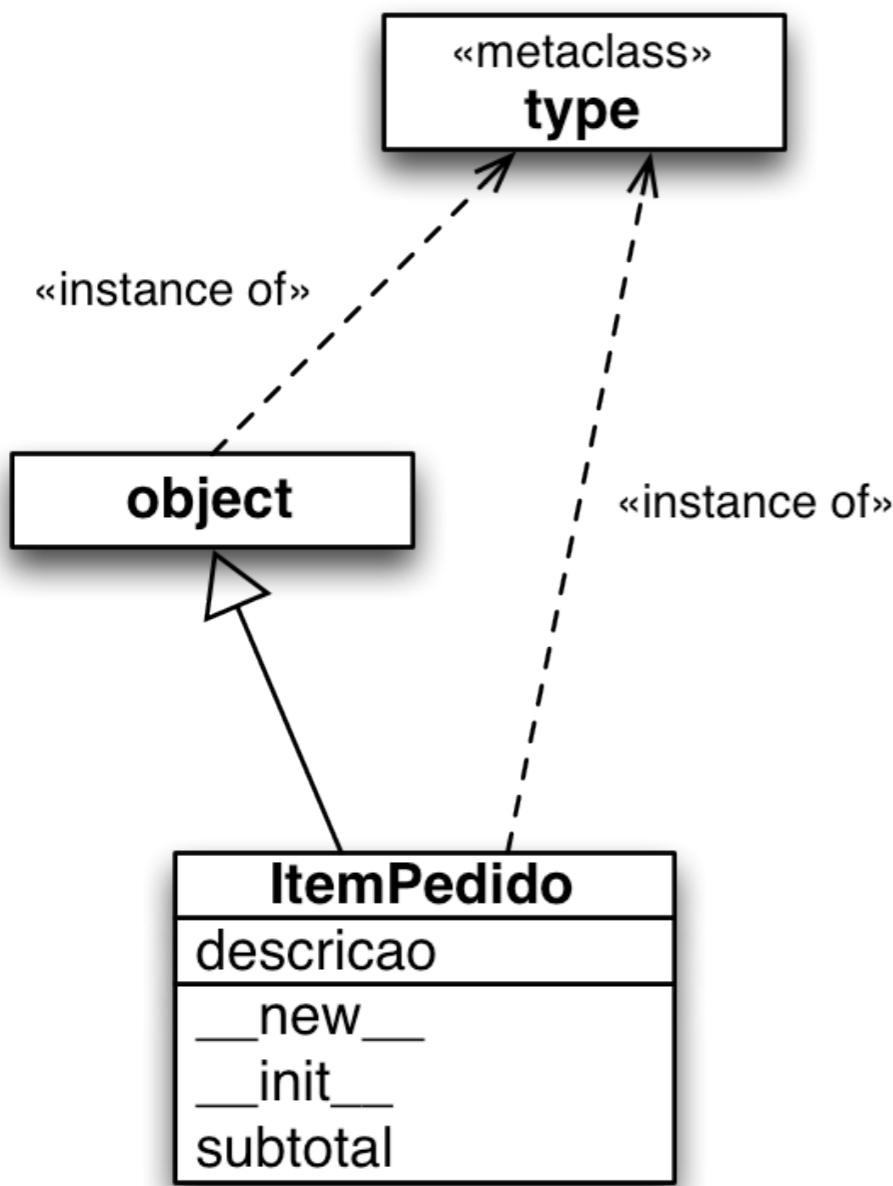
ItemPedido é uma instância de **type**

type é a metaclasses default em Python: a classe que normalmente constroi outras classes

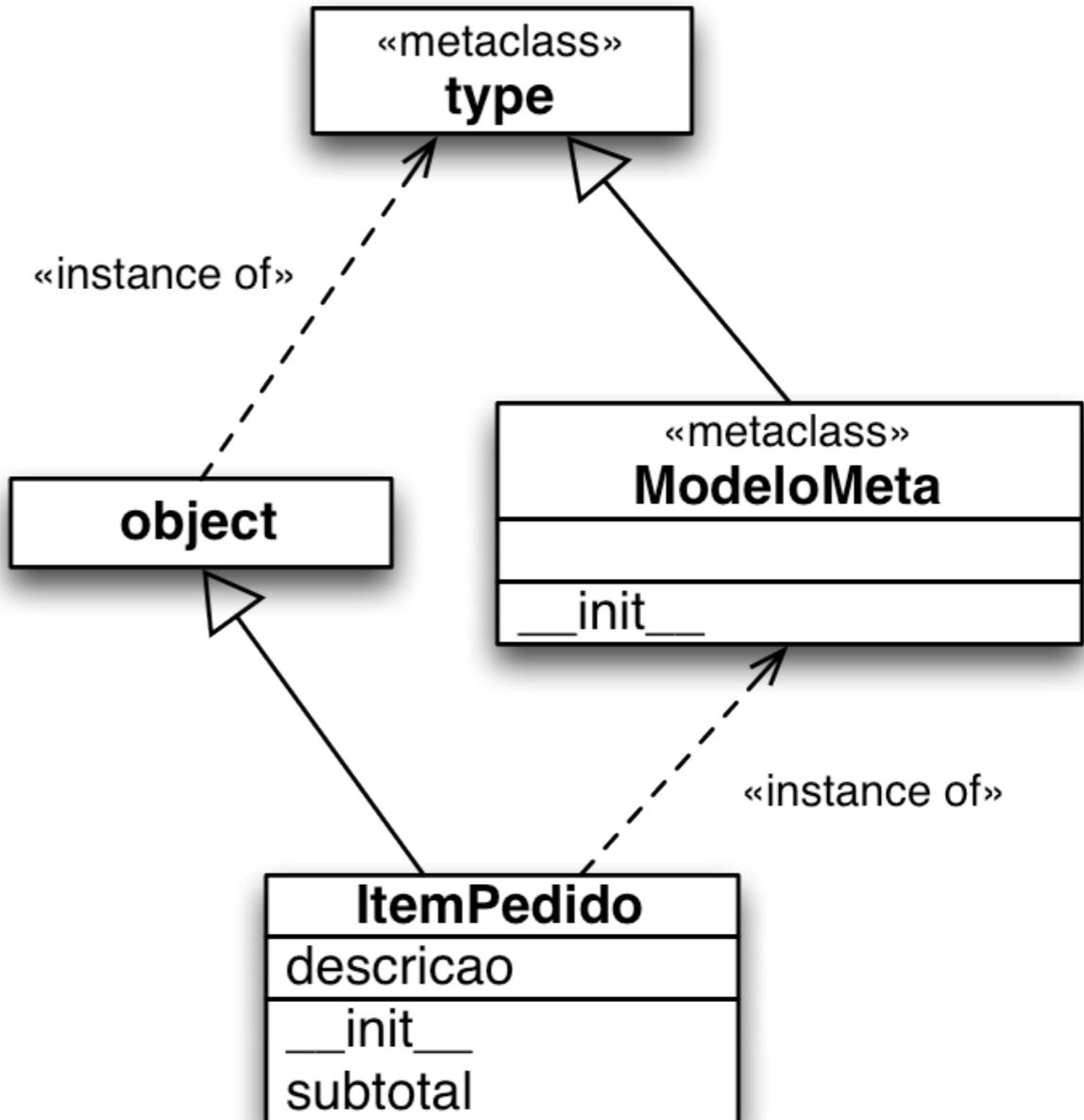
5

nossa metaclasses

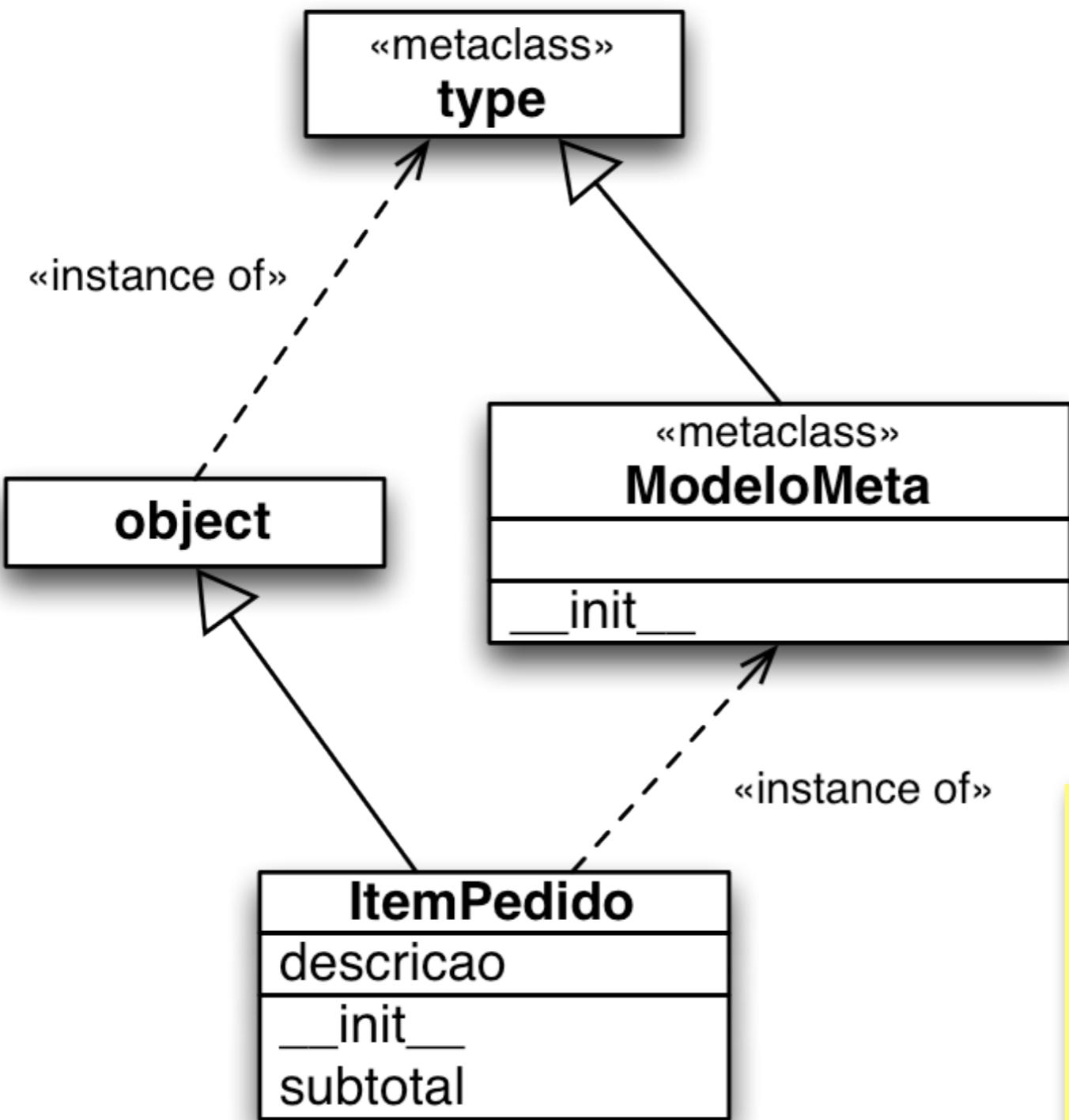
antes



depois



5 nossa metaclasses



ModeloMeta é a metaclasses que vai construir a classe **ItemPedido**

ModeloMeta.__init__ fará apenas uma vez o que antes era feito em **ItemPedido.__new__** a cada nova instância

5 nossa metaclasses

```
class ModeloMeta(type):

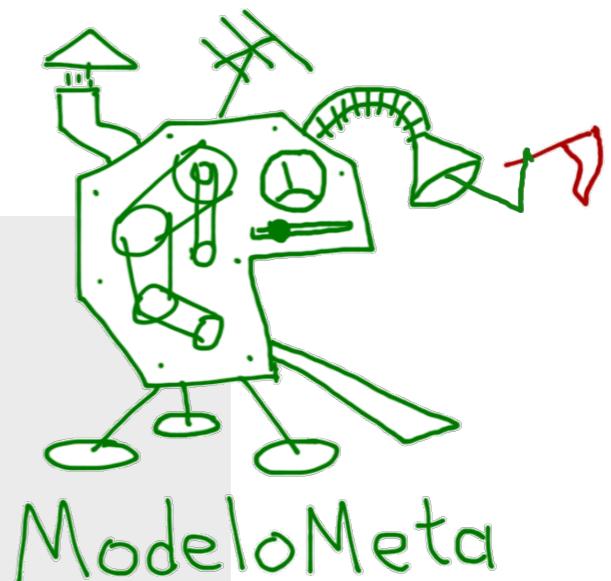
    def __init__(cls, nome, bases, dic):
        super(ModeloMeta, cls).__init__(nome, bases, dic)
        for chave, atr in dic.items():
            if hasattr(atr, 'set_nome'):
                atr.set_nome('__' + nome, chave)

class ItemPedido(object):
    __metaclass__ = ModeloMeta

    peso = Quantidade()
    preco = Quantidade()

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco
```



Assim dizemos que a classe **ItemPedido** herda de **object**, mas é uma instância de (é construída por) **ModeloMeta**



5

nossa metaclasses

```
class ModeloMeta(type):

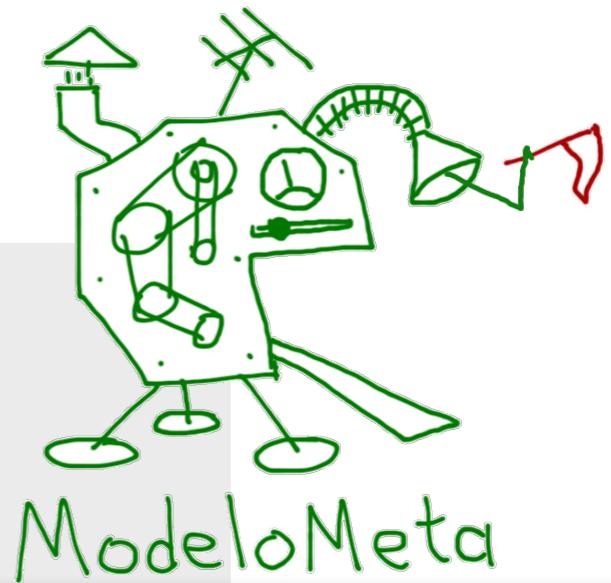
    def __init__(cls, nome, bases, dic):
        super(ModeloMeta, cls).__init__(nome, bases, dic)
        for chave, atr in dic.items():
            if hasattr(atr, 'set_nome'):
                atr.set_nome('__' + nome, chave)

class ItemPedido(object, metaclass=ModeloMeta): ←

    peso = Quantidade()
    preco = Quantidade()

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco
```



Somente em Python 3:
classe **ItemPedido**
herda de **object**, mas é
uma instância de
(construída por)
ModeloMeta



ItemPedido

5 nossa metaclasses

```
class ModeloMeta(type):

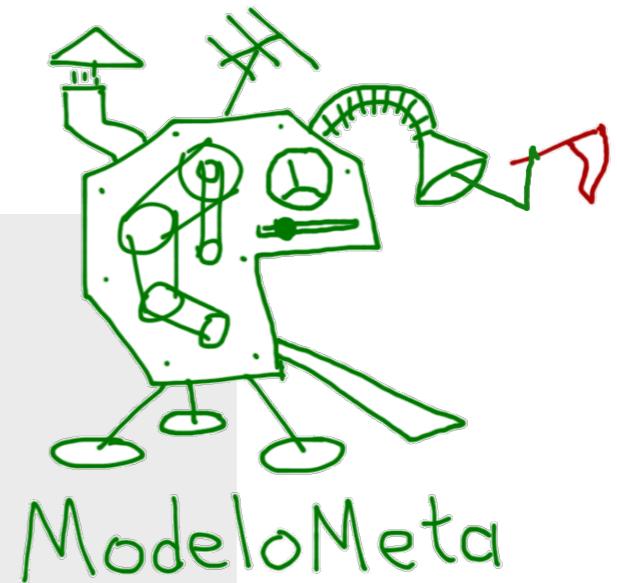
    def __init__(cls, nome, bases, dic):
        super(ModeloMeta, cls).__init__(nome, bases, dic)
        for chave, atr in dic.items():
            if hasattr(atr, 'set_nome'):
                atr.set_nome('__' + nome, chave)

class ItemPedido(object):
    __metaclass__ = ModeloMeta

    peso = Quantidade()
    preco = Quantidade()

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco

    def subtotal(self):
        return self.peso * self.preco
```

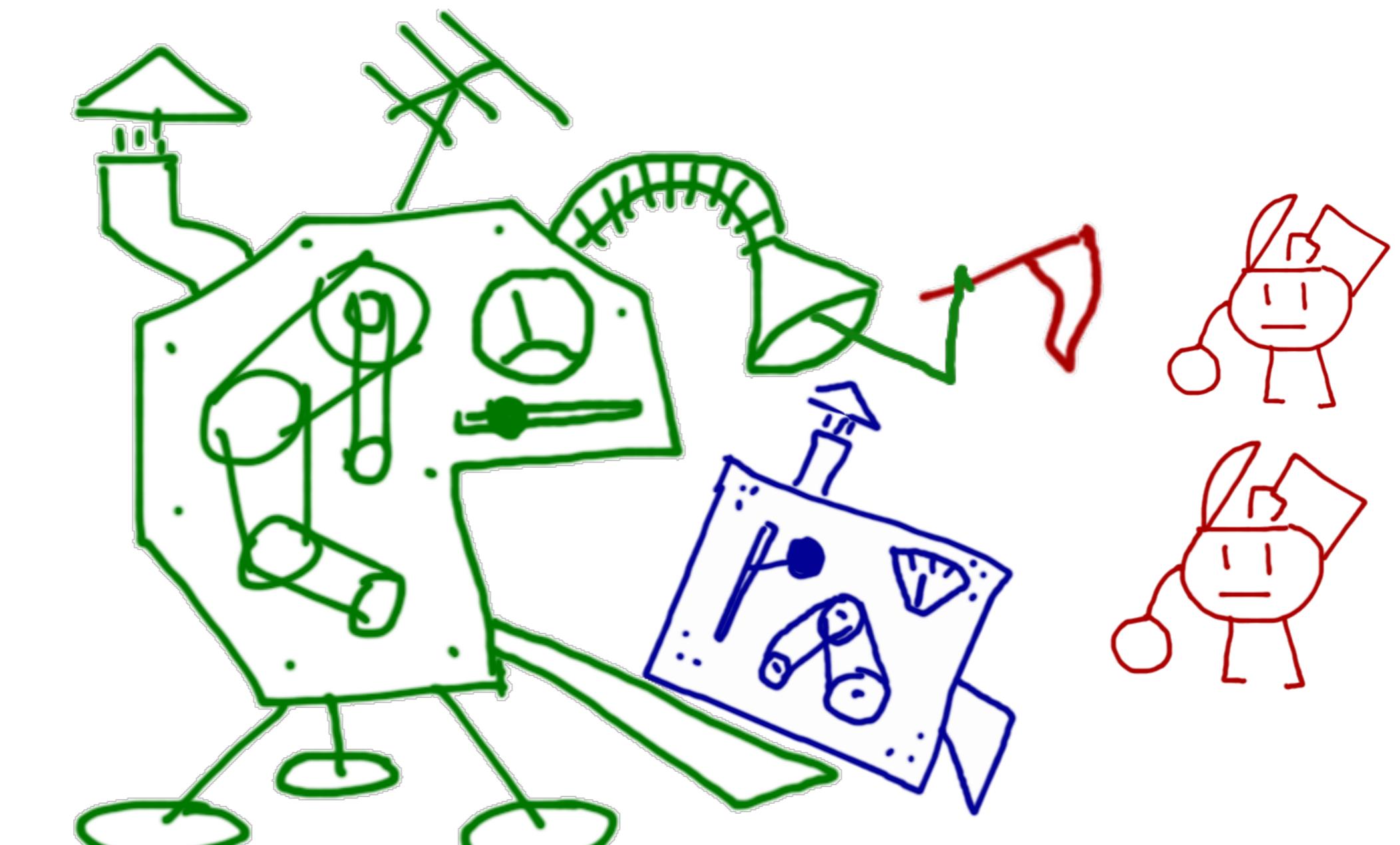


Este `__init__` invoca «`quantidade`».set_nome, para cada descritor, uma vez só, na inicialização da classe **ItemPedido**



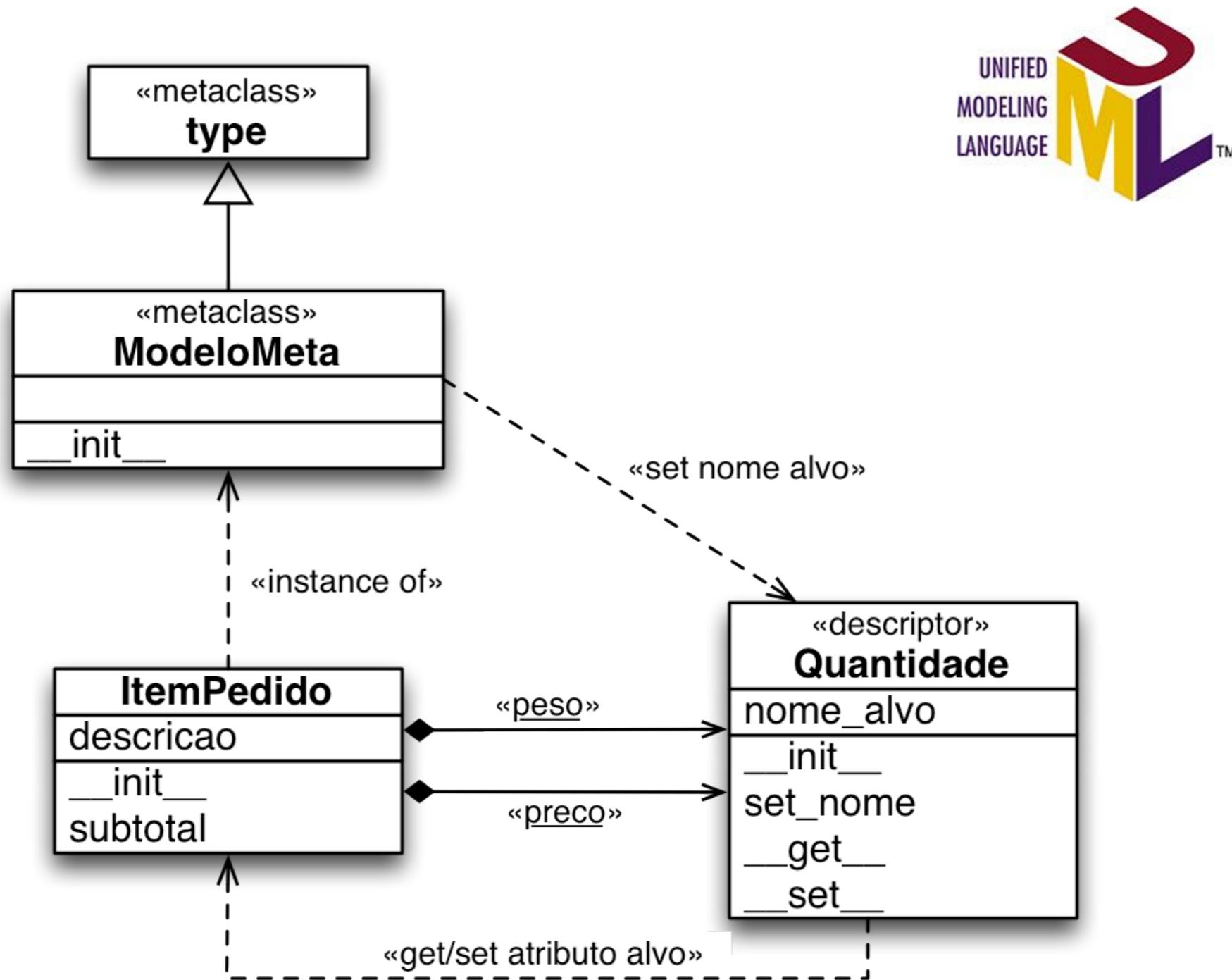
ItemPedido

5 nossa metaclass em ação

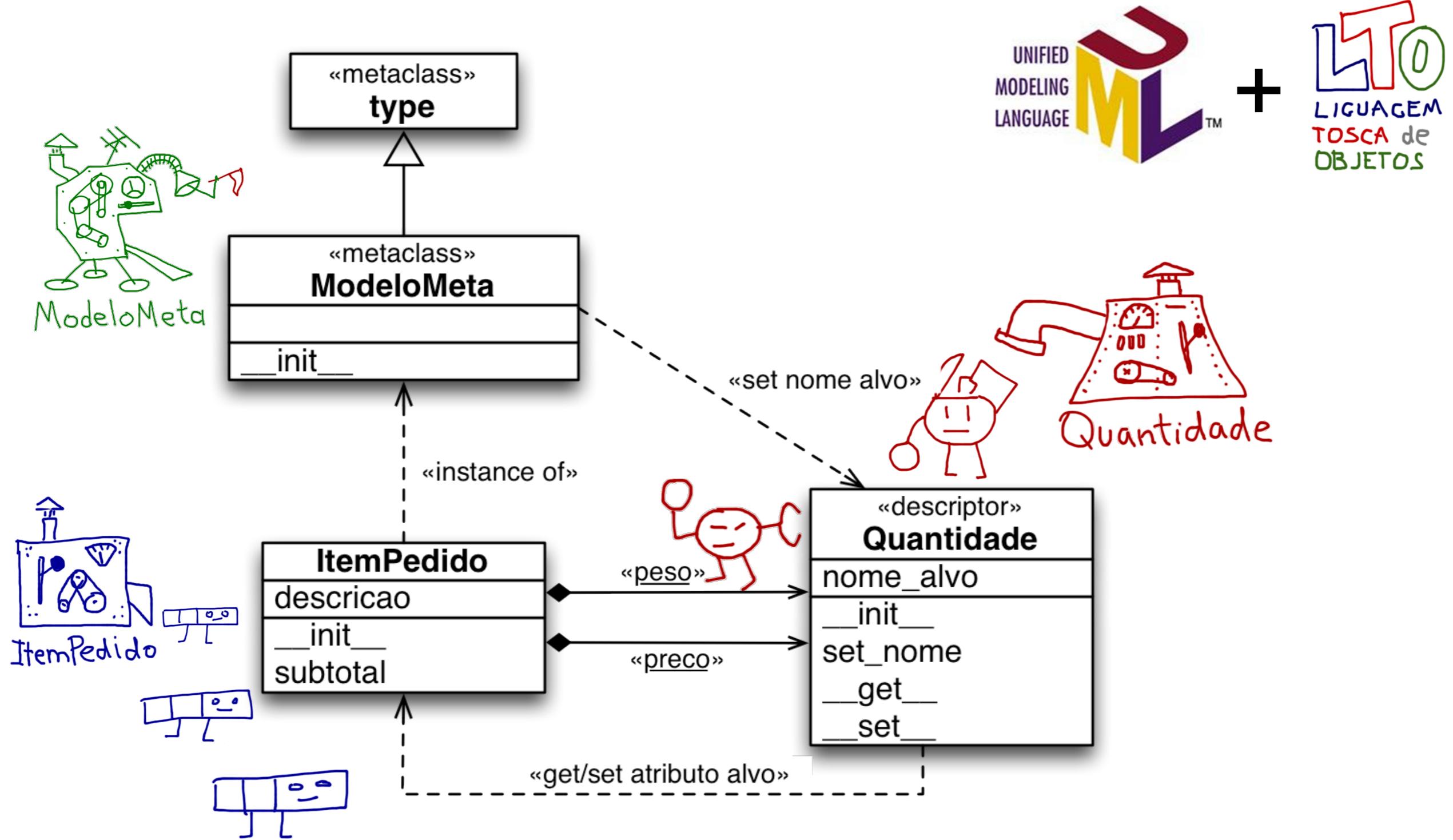


ModeloMeta

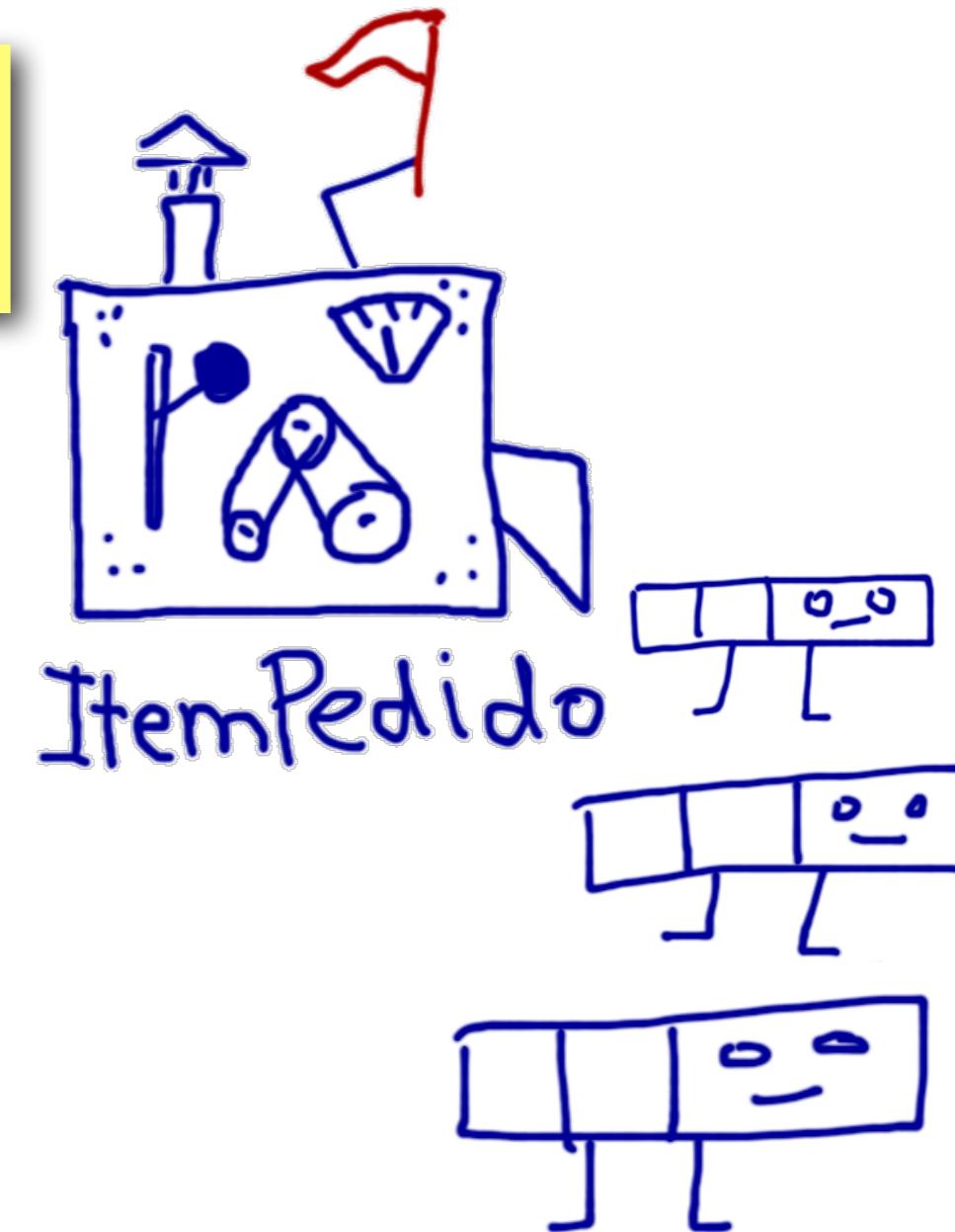
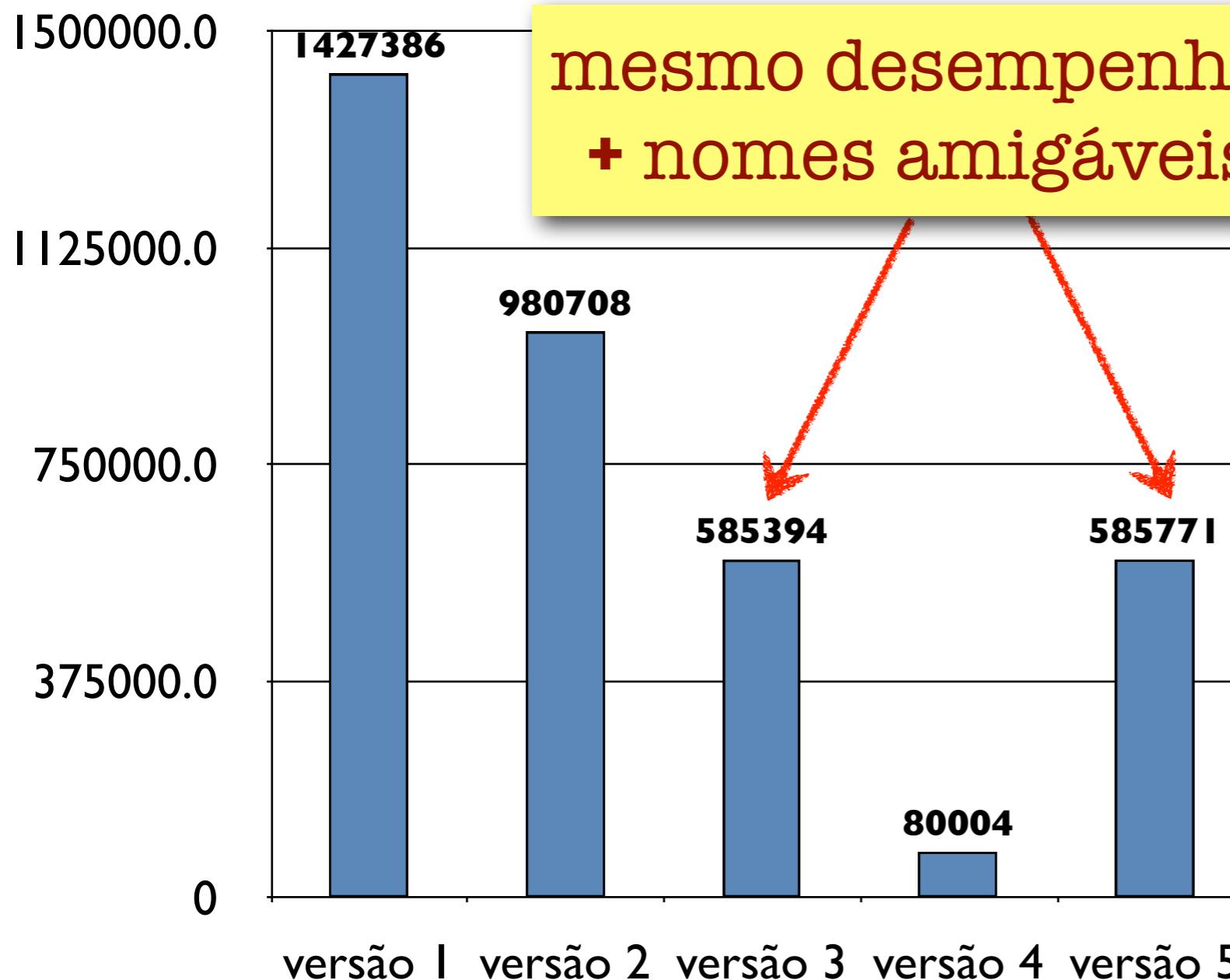
5 nossa metaclass em ação



5 nossa metaclass em ação



5 desempenho melhor

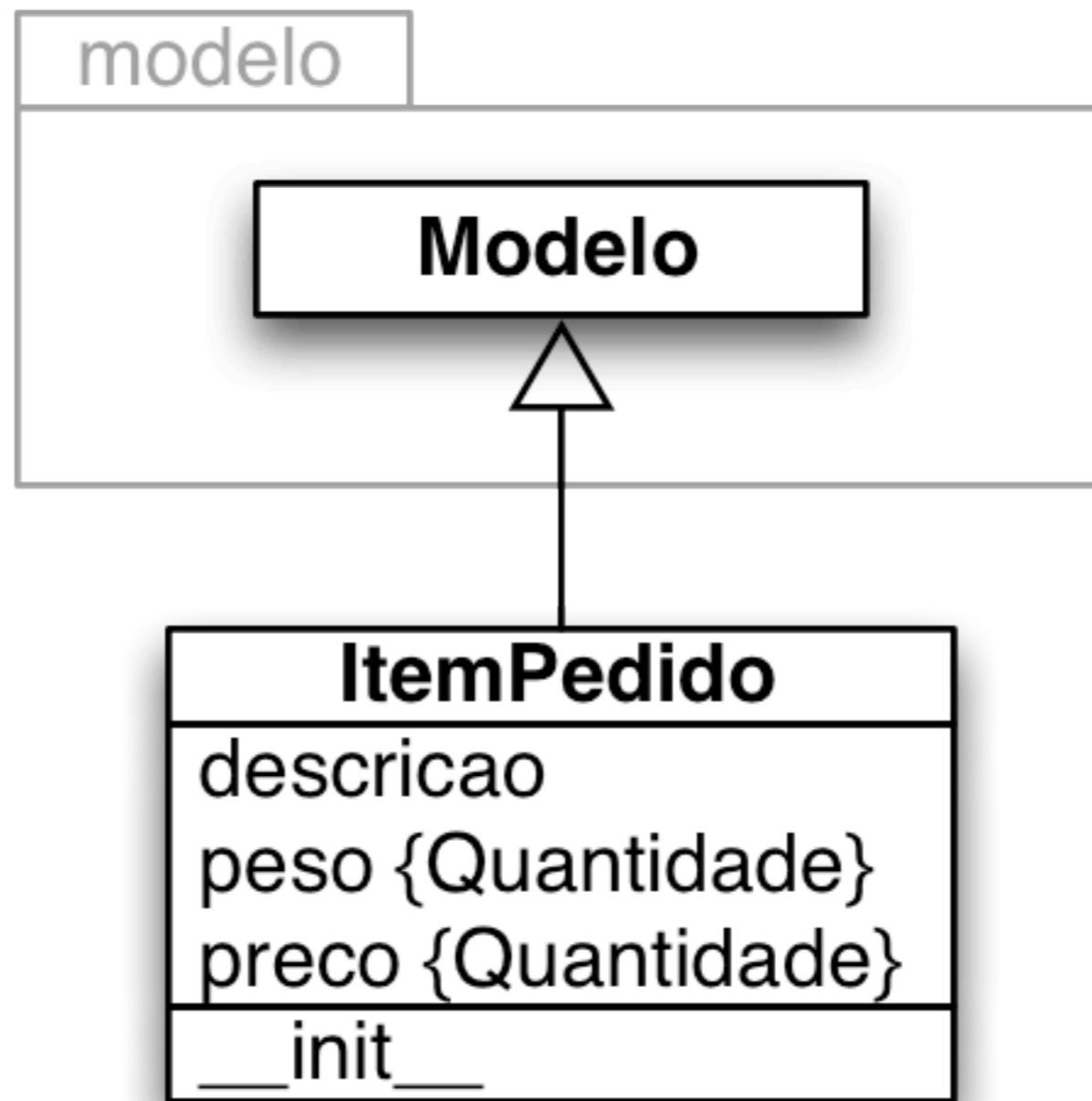


Número de instâncias de **ItemPedido** criadas por segundo (MacBook Pro 2011, Intel Core i7)

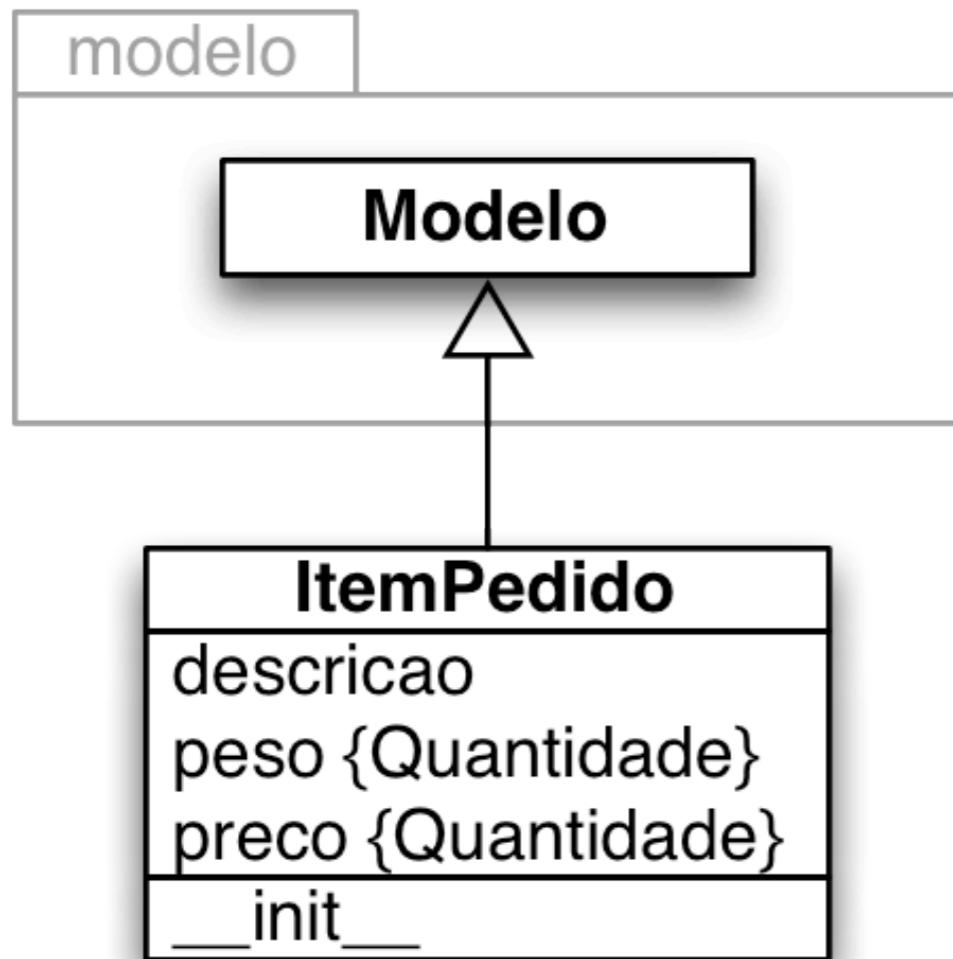


@pythonprobr

6 simplicidade aparente



6 o poder da abstração



```
from modelo import Modelo, Quantidade

class ItemPedido(Modelo):

    peso = Quantidade()
    preco = Quantidade()

    def __init__(self, descricao, peso, preco):
        self.descricao = descricao
        self.peso = peso
        self.preco = preco
```

6 módulo modelo.py

```
class Quantidade(object):

    def __init__(self):
        self.set_nome(self.__class__.__name__, id(self))

    def set_nome(self, prefix, key):
        self.nome_alvo = '%s_%s' % (prefix, key)

    def __get__(self, instance, owner):
        return getattr(instance, self.nome_alvo)

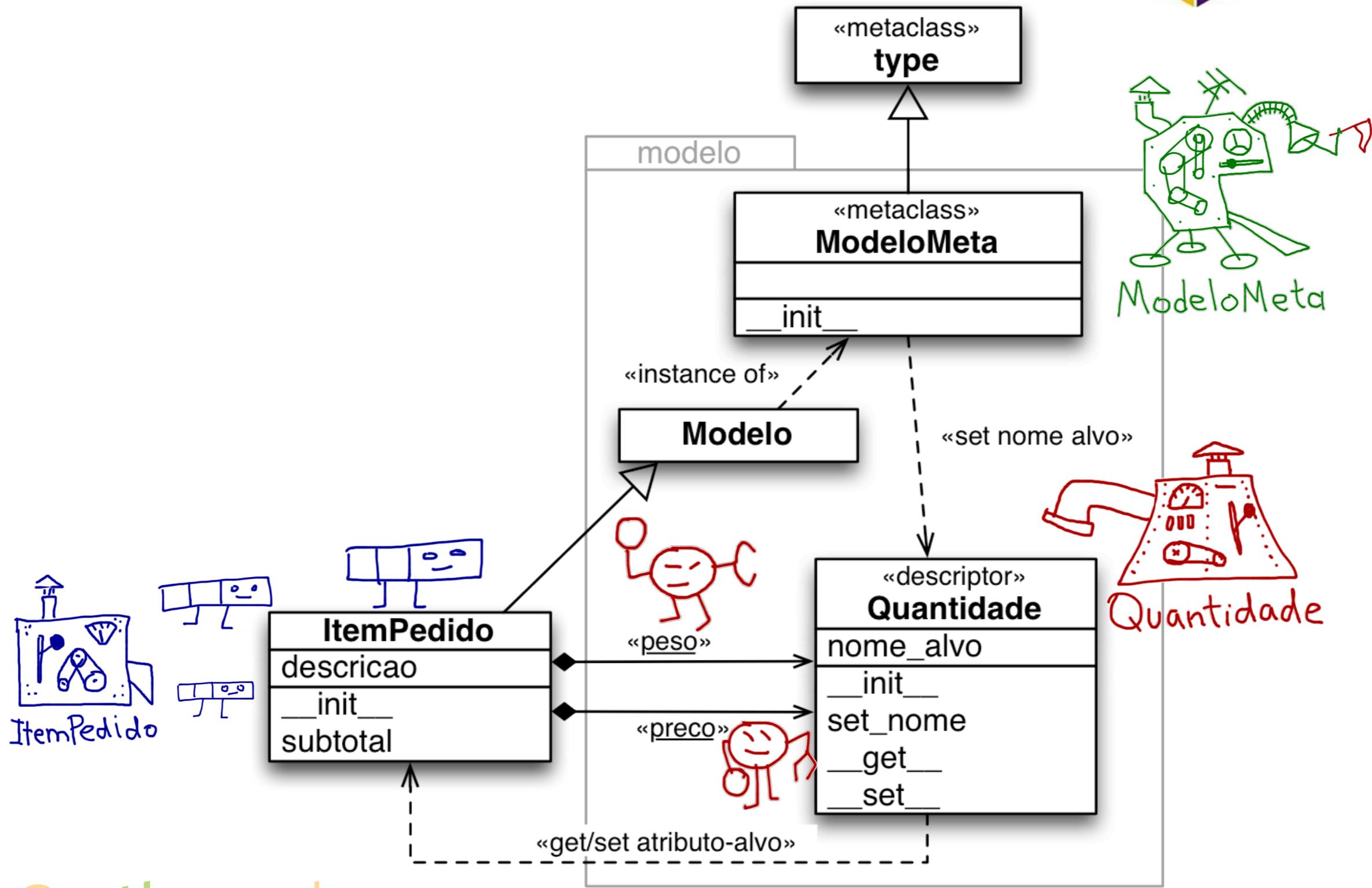
    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.nome_alvo, value)
        else:
            raise ValueError('valor deve ser > 0')

class ModeloMeta(type):

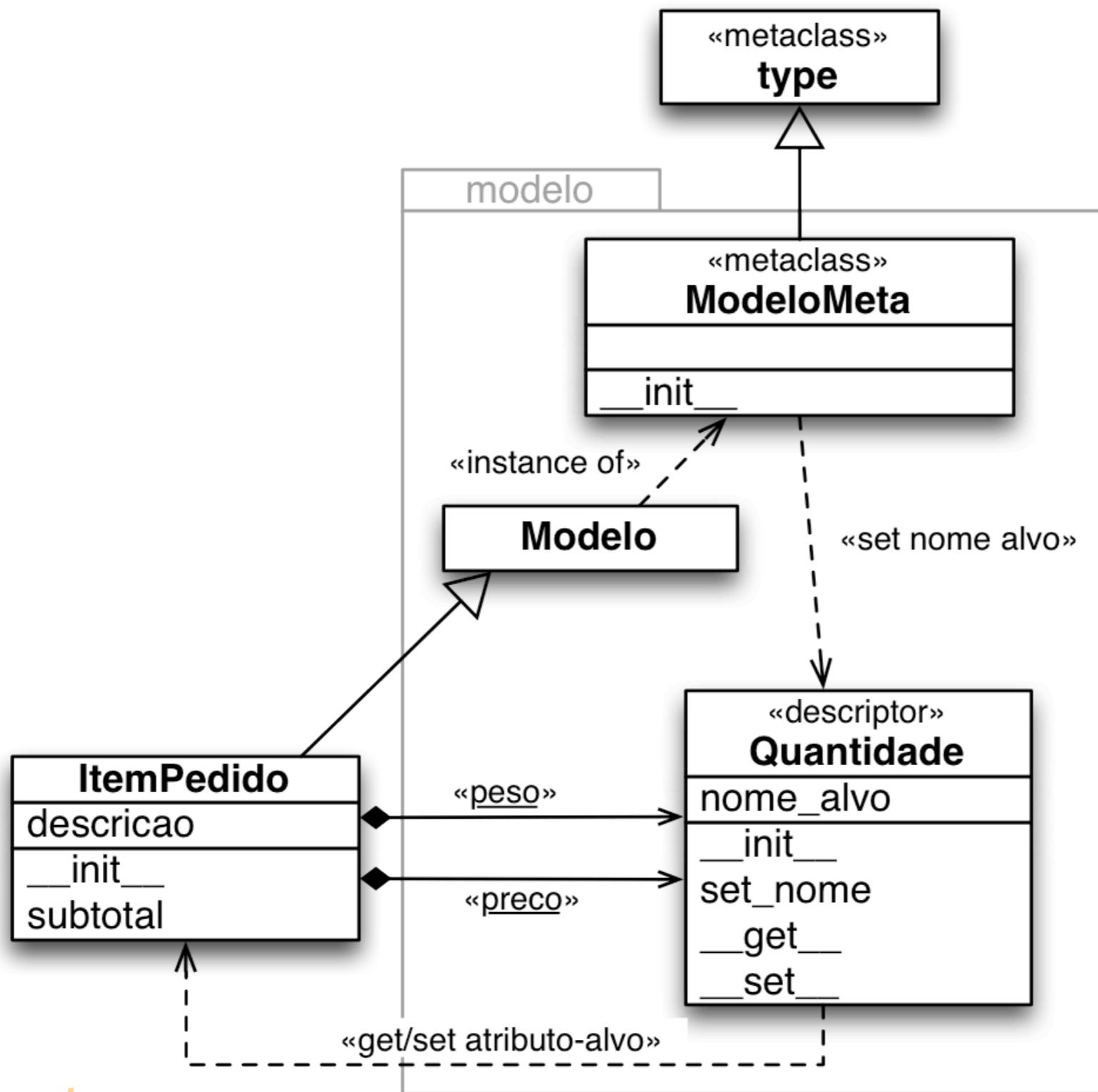
    def __init__(cls, nome, bases, dic):
        super(ModeloMeta, cls).__init__(nome, bases, dic)
        for chave, atr in dic.items():
            if hasattr(atr, 'set_nome'):
                atr.set_nome('__' + nome, chave)

class Modelo(object):
    __metaclass__ = ModeloMeta
```

6 esquema final



6 esquema final



Concluindo

- Uma metaclasses serve para customizar o processo de criação ou inicialização de novas classes
- Muitos ORMs (Django, SQLAlchemy) utilizam metaclasses para construir as classes de modelo
- Alguns casos de uso de metaclasses podem ser resolvidos de modo mais simples usando decoradores de classes em Python 3

Veremos os passos 5 e 6 deste exemplo com um decorador de classe