

Luciano Ramalho  
ramalho@python.pro.br

@pythonprobr



outubro/2013

# Objetos Pythonicos

Orientação a objetos e padrões de projeto em Python

# Aula 2

Introdução à Orientação Objetos em Python  
(continuação)

# Objetivos desta aula

- Continuar apresentando os conceitos fundamentais de orientação a objetos, utilizando a terminologia e as práticas da comunidade Python
- Apresentar a metodologia de TDD (Test Driven Design) usando Doctests

# Conceito: “classe”

- Uma categoria, ou tipo, de objeto
  - Uma idéia abstrata, uma forma platônica
- Exemplo: classe “Cão”:
  - Eu digo: “Ontem eu comprei um cão”
    - Você não sabe exatamente qual cão, mas sabe:
      - é um mamífero, quadrúpede, carnívoro
      - pode ser domesticado (normalmente)
      - cabe em um automóvel

# Exemplar de cão: instância da classe Cao



```
>>> rex = Cao()
```

instanciação

@pythonprobr

instanciação

# Classe Cao

```
class Mamifero(object):  
    """lição de casa: implementar"""  
  
class Cao(Mamifero):  
    qt_patas = 4  
    carnivoro = True  
    nervoso = False  
    def __init__(self, nome):  
        self.nome = nome  
    def latir(self, vezes=1):  
        # quando nervoso, late o dobro  
        vezes = vezes + (self.nervoso * vezes)  
        print self.nome + ':' + ' Au!' * vezes  
    def __str__(self):  
        return self.nome  
    def __repr__(self):  
        return 'Cao({0!r})'.format(self.nome)
```

```
>>> rex = Cao('Rex')  
>>> rex  
Cao('Rex')  
>>> print rex  
Rex  
>>> rex.qt_patas  
4  
>>> rex.latir()  
Rex: Au!  
>>> rex.latir(2)  
Rex: Au! Au!  
>>> rex.nervoso = True  
>>> rex.latir(3)  
Rex: Au! Au! Au! Au! Au! Au!  
>>> rex.qt_patas = 3  
>>> fido = Cao('Fido ')  
>>> fido.qt_patas  
4  
>>> rex.peso  
Traceback...  
...  
AttributeError...
```

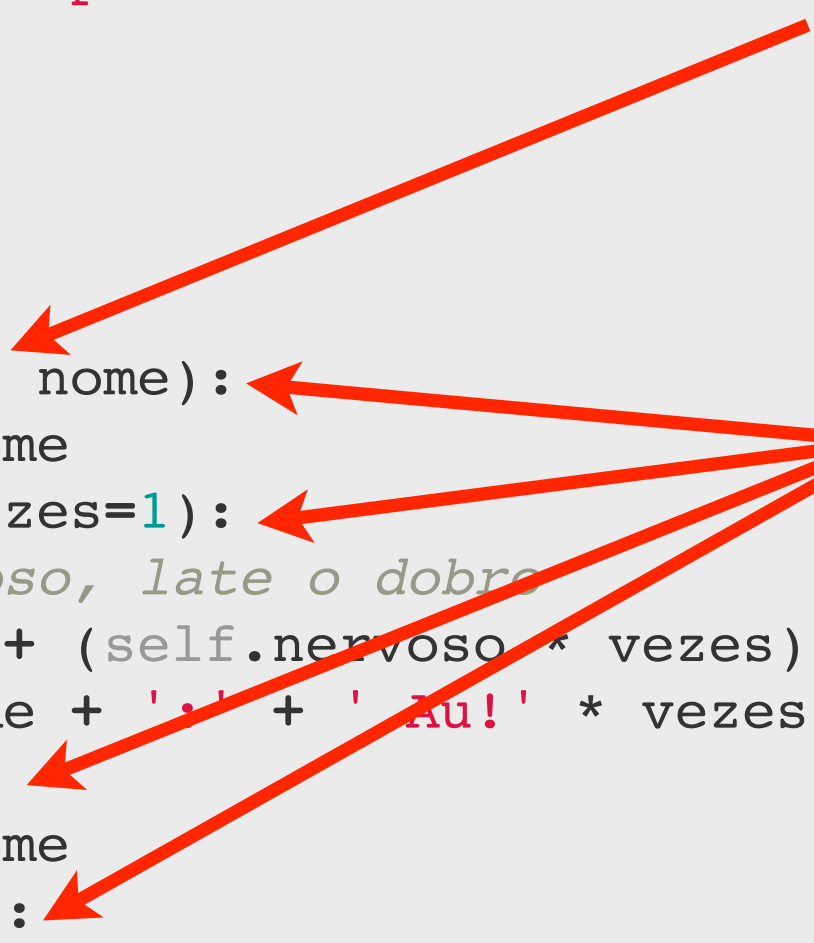
# Como atributos são acessados

- Ao buscar **`o.a`** (atributo **`a`** do objeto **`o`** da classe **`C`**), o interpretador Python faz o seguinte:
- 1) acessa atributo **`a`** da instancia **`o`**; caso não exista...
- 2) acessa atributo **`a`** da classe **`C`** de **`o`** (`type(o)` ou `o.__class__`); caso não exista...
- 3) busca o atributo **`a`** nas superclasses de **`C`**, conforme a MRO (method resolution order)



# Classe Cao em Python

```
class Mamifero(object):  
    """lição de casa: implementar"""  
  
class Cao(Mamifero):  
    qt_patas = 4  
    carnivoro = True  
    nervoso = False  
    def __init__(self, nome):  
        self.nome = nome  
    def latir(self, vezes=1):  
        # quando nervoso, late o dobro  
        vezes = vezes + (self.nervoso * vezes)  
        print self.nome + '.' + 'Au!' * vezes  
    def __str__(self):  
        return self.nome  
    def __repr__(self):  
        return 'Cao({0!r})'.format(self.nome)
```



- `__init__` é o construtor, ou melhor, o inicializador

- `self` é o 1º parâmetro formal em todos os métodos de instância



# Classe Cao

```
class Mamifero(object):  
    """lição de casa: implementar"""  
  
class Cao(Mamifero):  
    qt_patas = 4  
    carnivoro = True  
    nervoso = False  
    def __init__(self, nome):  
        self.nome = nome  
    def latir(self, vezes=1):  
        # quando nervoso, late o dobro  
        vezes = vezes + (self.nervoso * vezes)  
        print self.nome + ':' + ' Au!' * vezes  
    def __str__(self):  
        return self.nome  
    def __repr__(self):  
        return 'Cao({0!r})'.format(self.nome)
```

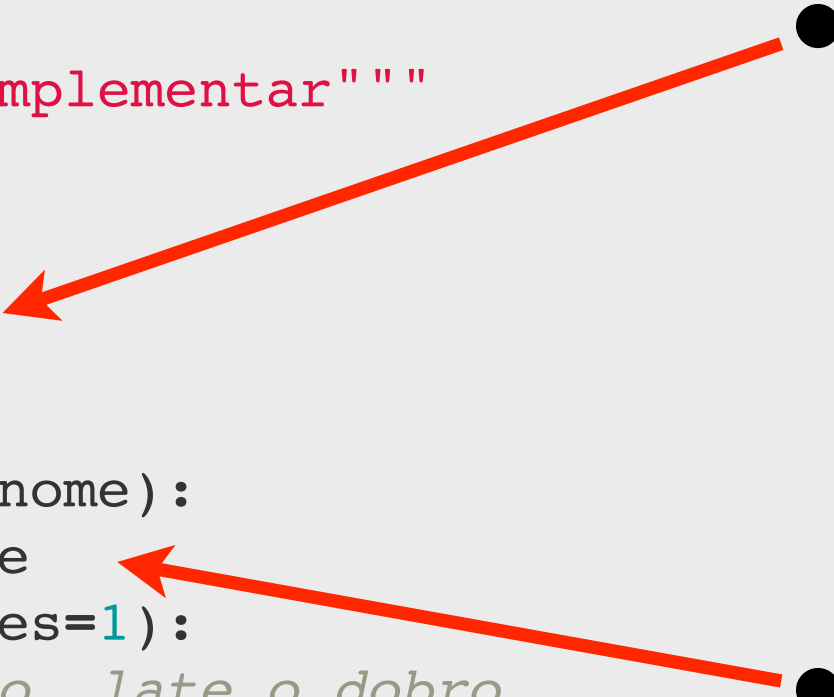
invocação

```
>>> rex = Cao('Rex')  
>>> rex  
Cao('Rex')  
>>> print rex  
Rex  
>>> rex.qt_patas  
4  
>>> rex.latir()  
Rex: Au!  
>>> rex.latir(2)  
Rex: Au! Au!
```

na invocação do método, a instância é passada automaticamente na posição do self

# Classe Cao em Python

```
class Mamifero(object):  
    """lição de casa: implementar"""  
  
class Cao(Mamifero):  
    qt_patas = 4  
    carnivoros = True  
    nervoso = False  
    def __init__(self, nome):  
        self.nome = nome  
    def latir(self, vezes=1):  
        # quando nervoso, late o dobro  
        vezes = vezes + (self.nervoso * vezes)  
        print self.nome + ':' + ' Au!' * vezes  
    def __str__(self):  
        return self.nome  
    def __repr__(self):  
        return 'Cao({0!r})'.format(self.nome)
```



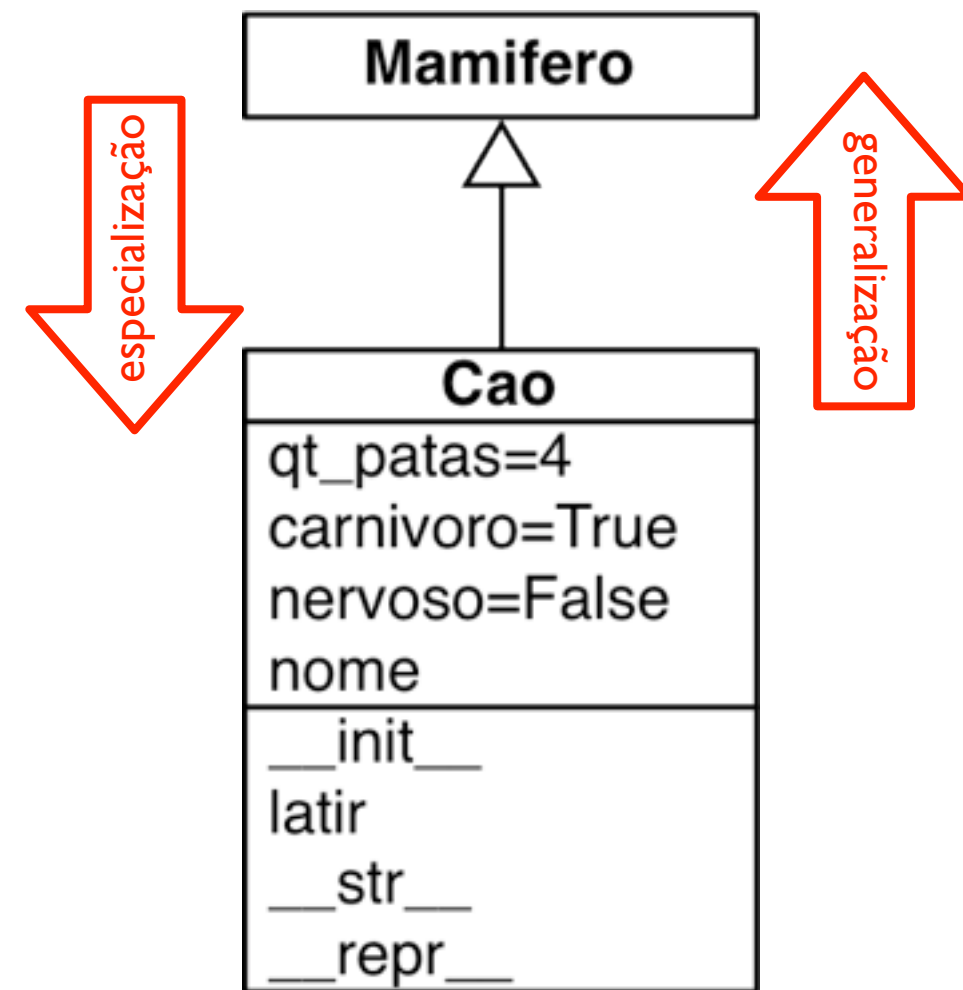
- atributos de dados na classe funcionam como valores default para os atributos das instâncias

- atributos da instância só podem ser acessados via self

# Mamifero: superclasse de Cao

```
class Mamifero(object):  
    """lição de casa: implementar"""  
  
class Cao(Mamifero):  
    qt_patas = 4  
    carnivoro = True  
    nervoso = False  
    def __init__(self, nome):  
        self.nome = nome  
    def latir(self, vezes=1):  
        # quando nervoso, late o dobro  
        vezes = vezes + (self.nervoso * vezes)  
        print self.nome + ':' + ' Au!' * vezes  
    def __str__(self):  
        return self.nome  
    def __repr__(self):  
        return 'Cao({0!r})'.format(self.nome)
```

## UML diagrama de classe



# Subclasses de Cao

- Continuação de cao.py

```
class Pequines(Cao):  
    nervoso = True  
  
class Mastiff(Cao):  
    def latir(self, vezes=1):  
        # o mastiff não muda seu latido  
        print self.nome + ':' + ' Wuff!'  
  
class SaoBernardo(Cao):  
    def __init__(self, nome):  
        Cao.__init__(self, nome)  
        self.doses = 10  
    def servir(self):  
        if self.doses == 0:  
            raise ValueError('Acabou o conhaque!')  
        self.doses -= 1  
        msg = '{0} serve o conhaque (restam {1} doses)'  
        print msg.format(self.nome, self.doses)
```



Diz a lenda que o cão São Bernardo leva um pequeno barril de conhaque para resgatar viajantes perdidos na neve.

# Subclasses de Cao

```
class Pequines(Cao):  
    nervoso = True
```

```
class Mastiff(Cao):  
    def latir(self, vezes=1):  
        # o mastiff não muda seu latido quando nervoso  
        print self.nome + ':' + ' Wuff!' * vezes
```

```
class SaoBernardo(Cao):  
    def __init__(self, nome):  
        Cao.__init__(self, nome)  
        self.doses = 10  
    def servir(self):  
        if self.doses == 0:  
            raise ValueError('Acabou o conhaque!')  
        self.doses -= 1  
        msg = '{0} serve o conhaque (restam {1} doses)'  
        print msg.format(self.nome, self.doses)
```

```
>>> sansao = SaoBernardo('Sansao')  
>>> sansao.servir()  
Sansao serve o conhaque (restam 9 doses)  
>>> sansao.doses = 1  
>>> sansao.servir()  
Sansao serve o conhaque (restam 0 doses)  
>>> sansao.servir()  
Traceback (most recent call last):  
    ...  
ValueError: Acabou o conhaque!
```



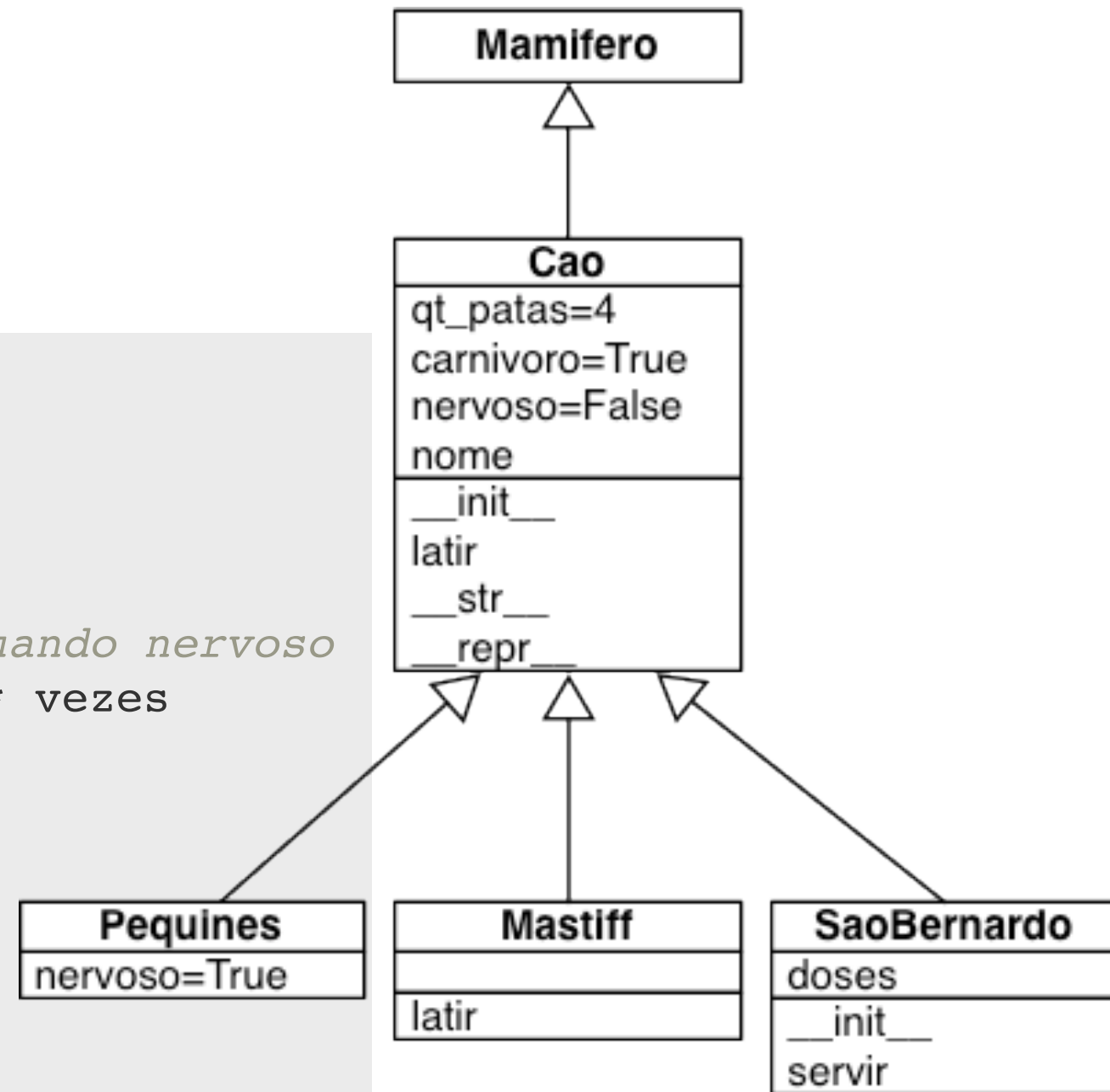
# Subclasses de Cao

- Continuação de cao.py

```
class Pequines(Cao):
    nervoso = True

class Mastiff(Cao):
    def latir(self, vezes=1):
        # o mastiff não muda seu latido quando nervoso
        print self.nome + ':' + ' Wuff!' * vezes

class SaoBernardo(Cao):
    def __init__(self, nome):
        Cao.__init__(self, nome)
        self.doses = 10
    def servir(self):
        if self.doses == 0:
            raise ValueError('Acabou o conhaque!')
        self.doses -= 1
        msg = '{0} serve o conhaque (restam {1} doses)'
        print msg.format(self.nome, self.doses)
```



# Herança múltipla

- Refatoração de cao.py para cao2.py
- Reutilizar o latido do mastiff em outros cães grandes

mixin

herança múltipla

herança múltipla

```
class Cao(Mamifero):
    qt_patas = 4
    carnivoros = True
    nervoso = False
    def __init__(self, nome):
        self.nome = nome
    def latir(self, vezes=1):
        # quando nervoso, late o dobro
        vezes = vezes + (self.nervoso * vezes)
        print self.nome + ':' + ' Au!' * vezes
    def __str__(self):
        return self.nome
    def __repr__(self):
        return 'Cao({0!r})'.format(self.nome)

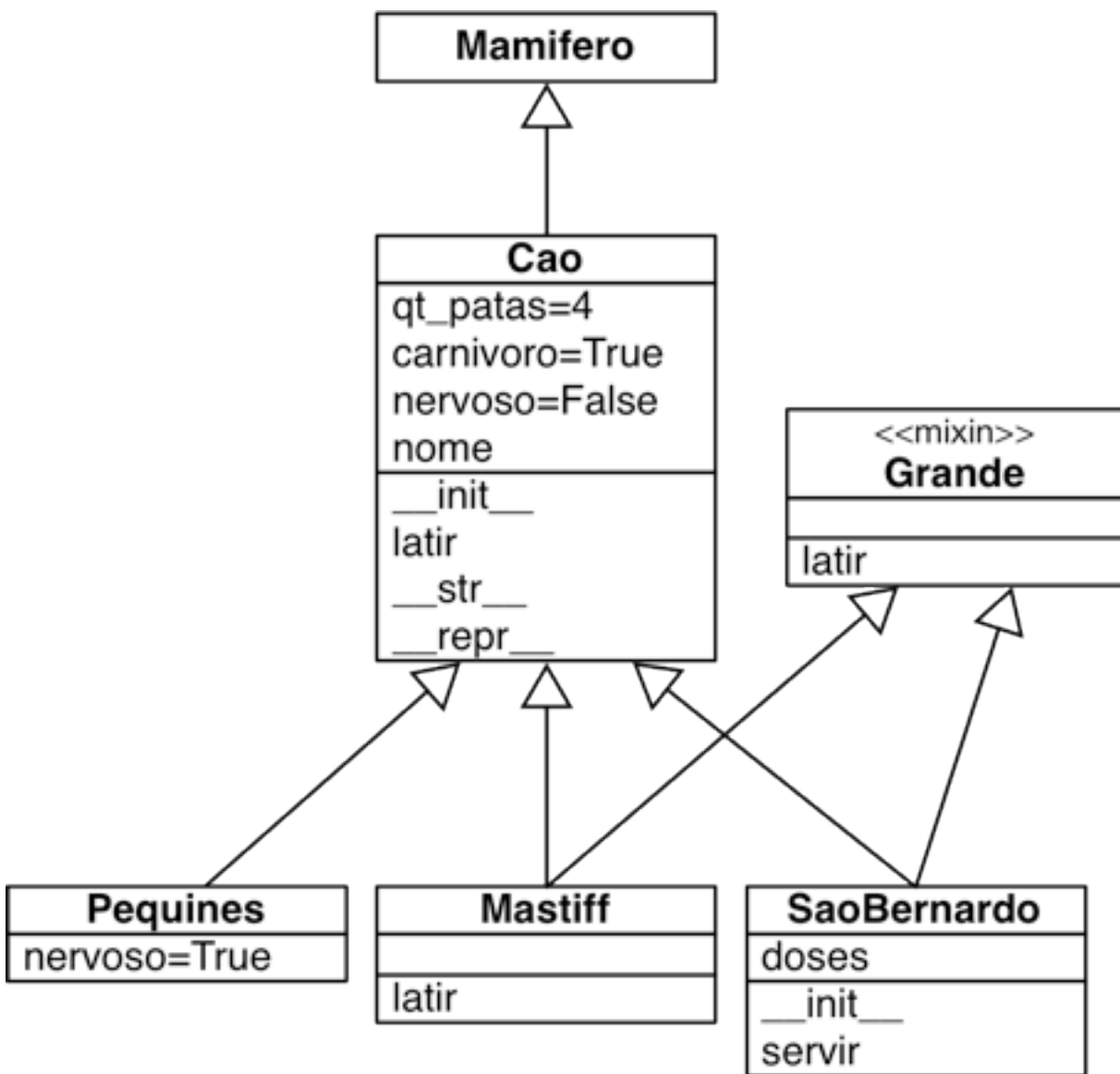
class Grande(object):
    """ Mixin: muda o latido """
    def latir(self, vezes=1):
        # faz de conta que cães grandes não mudam
        # seu latido quando nervosos
        print self.nome + ':' + ' Wuff!' * vezes

class Mastiff(Grande, Cao):
    """ O mastiff é o maior cão que existe """

class SaoBernardo(Grande, Cao):
    def __init__(self, nome):
        Cao.__init__(self, nome)
        self.doses = 10
    def servir(self):
        if self.doses == 0:
            raise ValueError('Acabou o conhaque!')
        self.doses -= 1
        msg = '{0} serve o conhaque (restam {1} doses)'
        print msg.format(self.nome, self.doses)
```



# Herança múltipla



```
class Cao(Mamifero):
    qt_patas = 4
    carnivoro = True
    nervoso = False
    def __init__(self, nome):
        self.nome = nome
    def latir(self, vezes=1):
        # quando nervoso, late o dobro
        vezes = vezes + (self.nervoso * vezes)
        print self.nome + ':' + ' Au!' * vezes
    def __str__(self):
        return self.nome
    def __repr__(self):
        return 'Cao({0!r})'.format(self.nome)

class Grande(object):
    """ Mixin: muda o latido """
    def latir(self, vezes=1):
        # faz de conta que cães grandes não mudam
        # seu latido quando nervosos
        print self.nome + ':' + ' Wuff!' * vezes

class Mastiff(Grande, Cao):
    """ O mastiff é o maior cão que existe """

class SaoBernardo(Grande, Cao):
    def __init__(self, nome):
        Cao.__init__(self, nome)
        self.doses = 10
    def servir(self):
        if self.doses == 0:
            raise ValueError('Acabou o conhaque!')
        self.doses -= 1
        msg = '{0} serve o conhaque (restam {1} doses)'
        print msg.format(self.nome, self.doses)
```

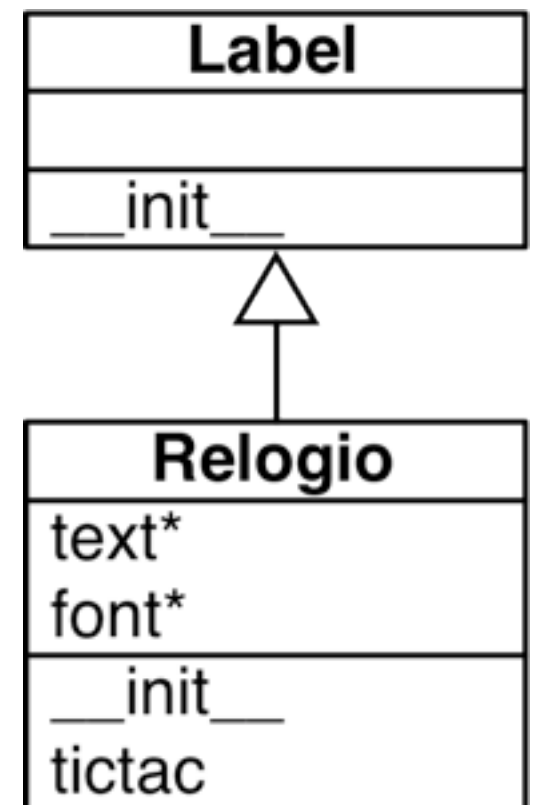
# Relógio com classe

```
import Tkinter
from time import strftime

class Relogio(Tkinter.Label):
    def __init__(self):
        Tkinter.Label.__init__(self)
        self.pack()
        self['text'] = strftime('%H:%M:%S')
        self['font'] = 'Helvetica 120 bold'
        self.tictac()

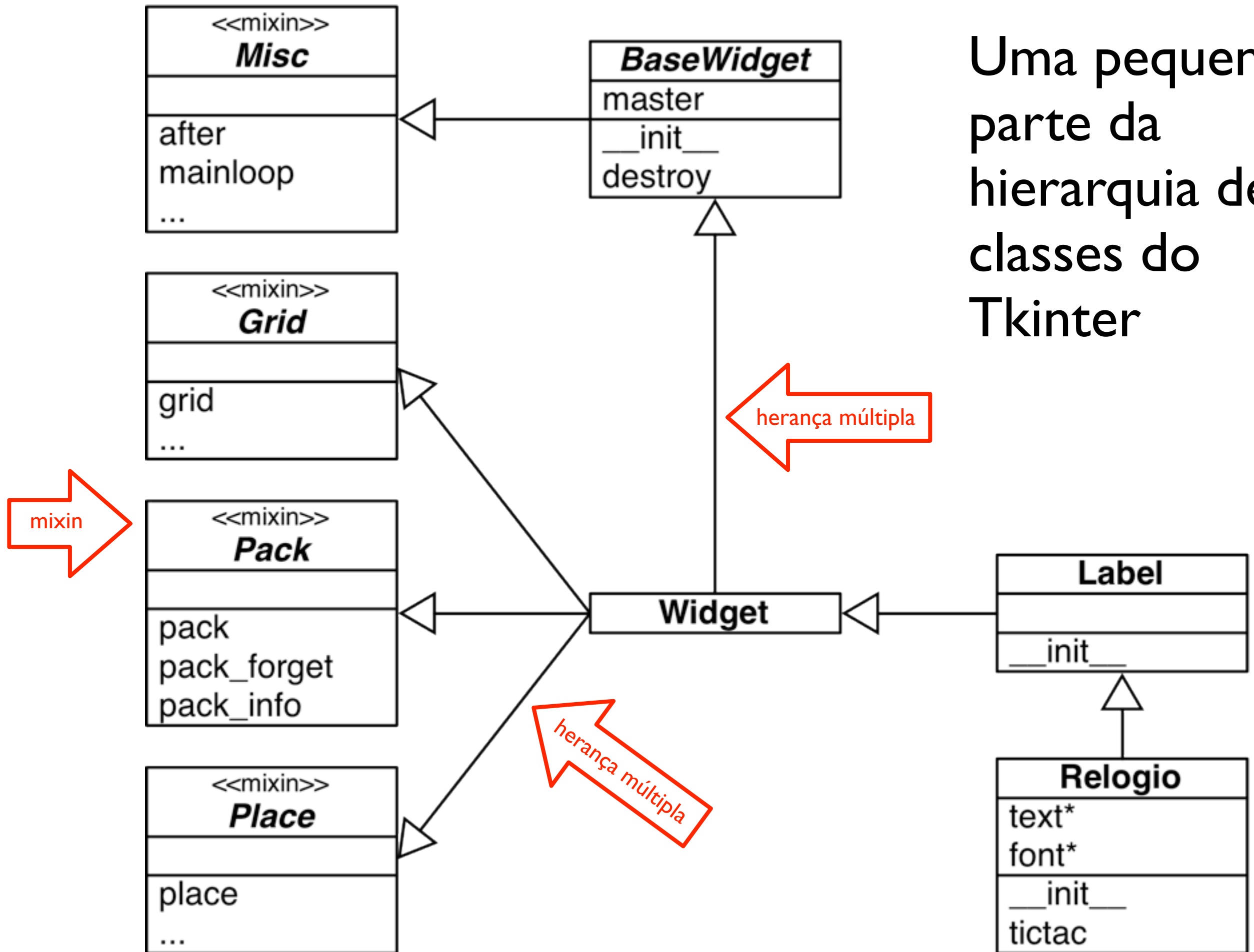
    def tictac(self):
        agora = strftime('%H:%M:%S')
        if agora != self['text']:
            self['text'] = agora
            self.after(100, self.tictac)

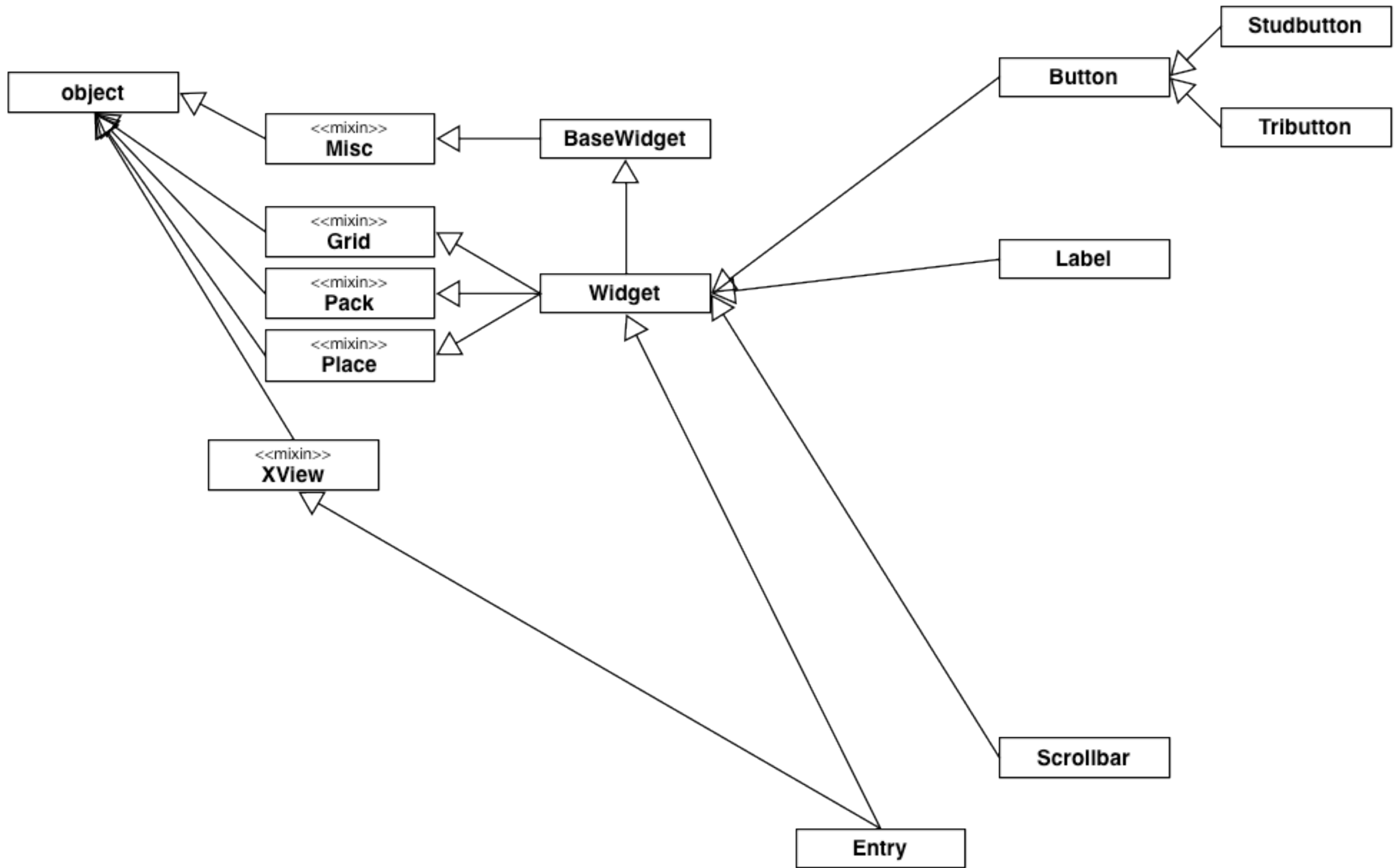
rel = Relogio()
rel.mainloop()
```



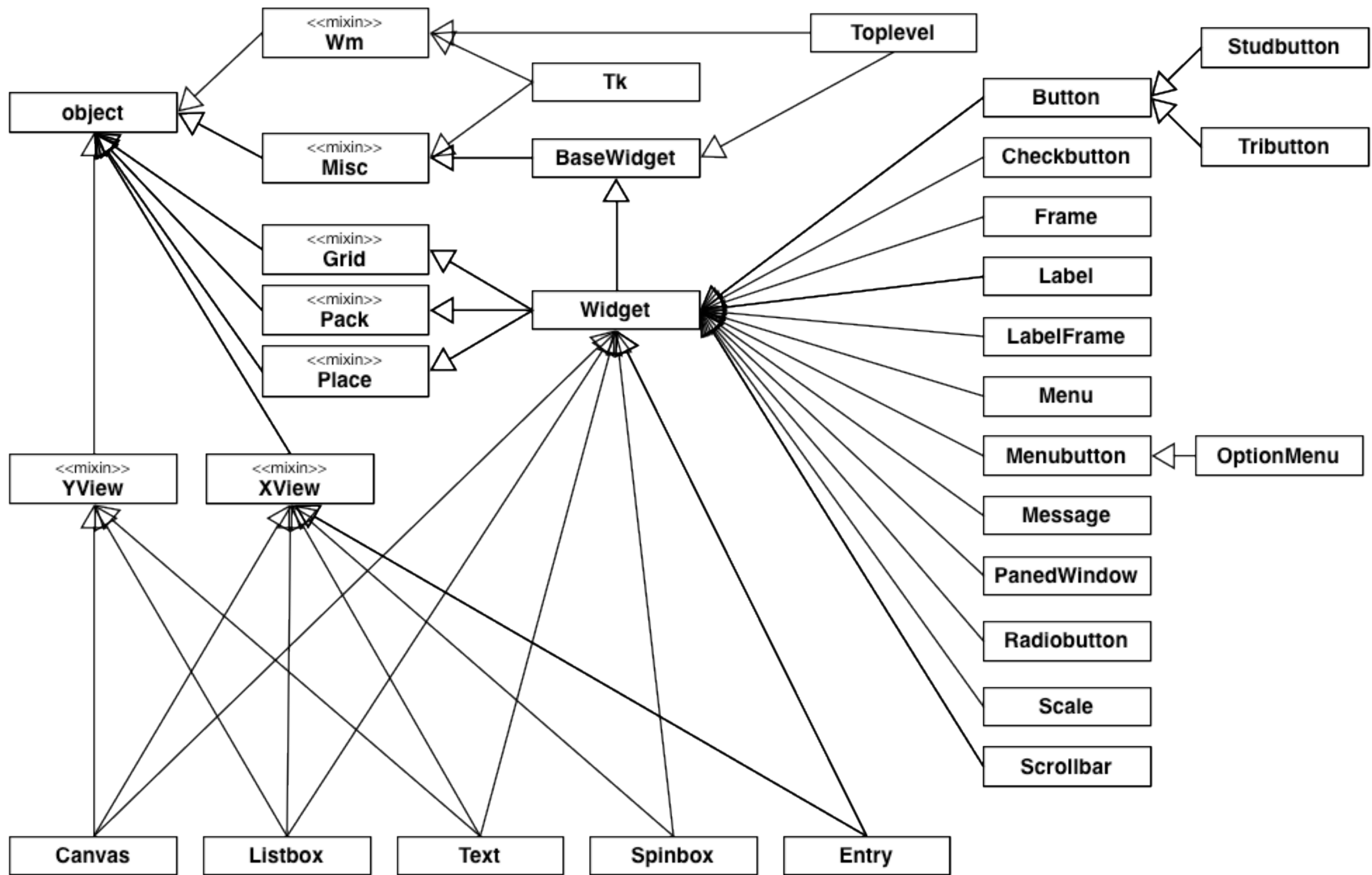
Fonte: <https://github.com/pythonprobr/oopy>

Uma pequena  
parte da  
hierarquia de  
classes do  
Tkinter



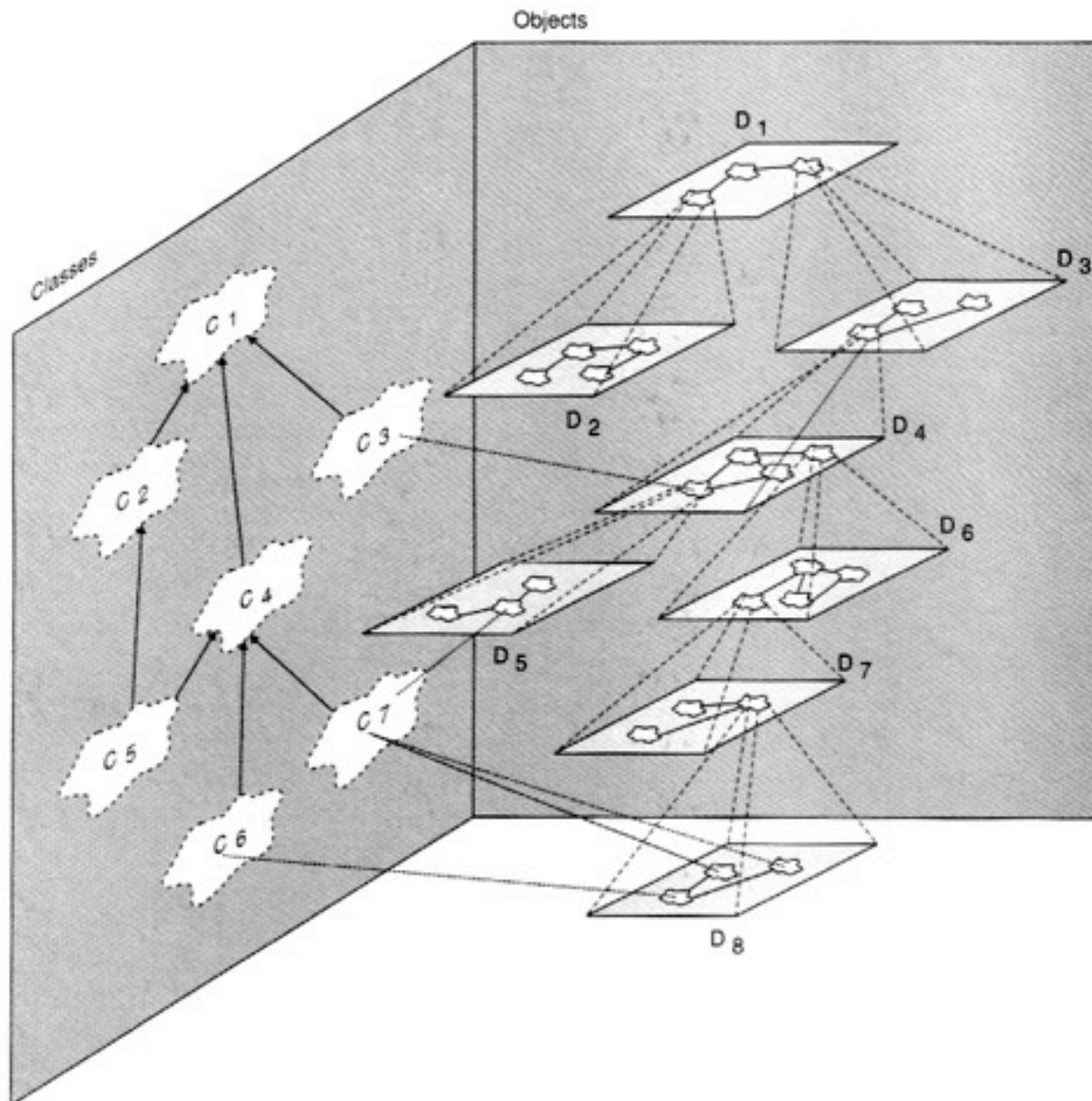


Um pouco mais da hierarquia de classes do Tkinter



Hierarquia de classes dos objetos gráficos doTkinter

# As duas hierarquias de um sistema OO



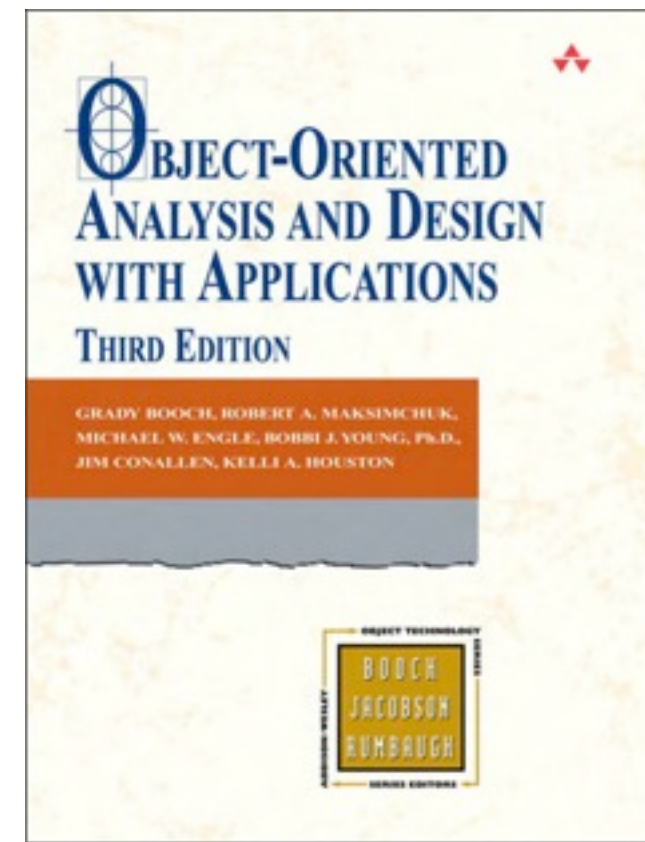
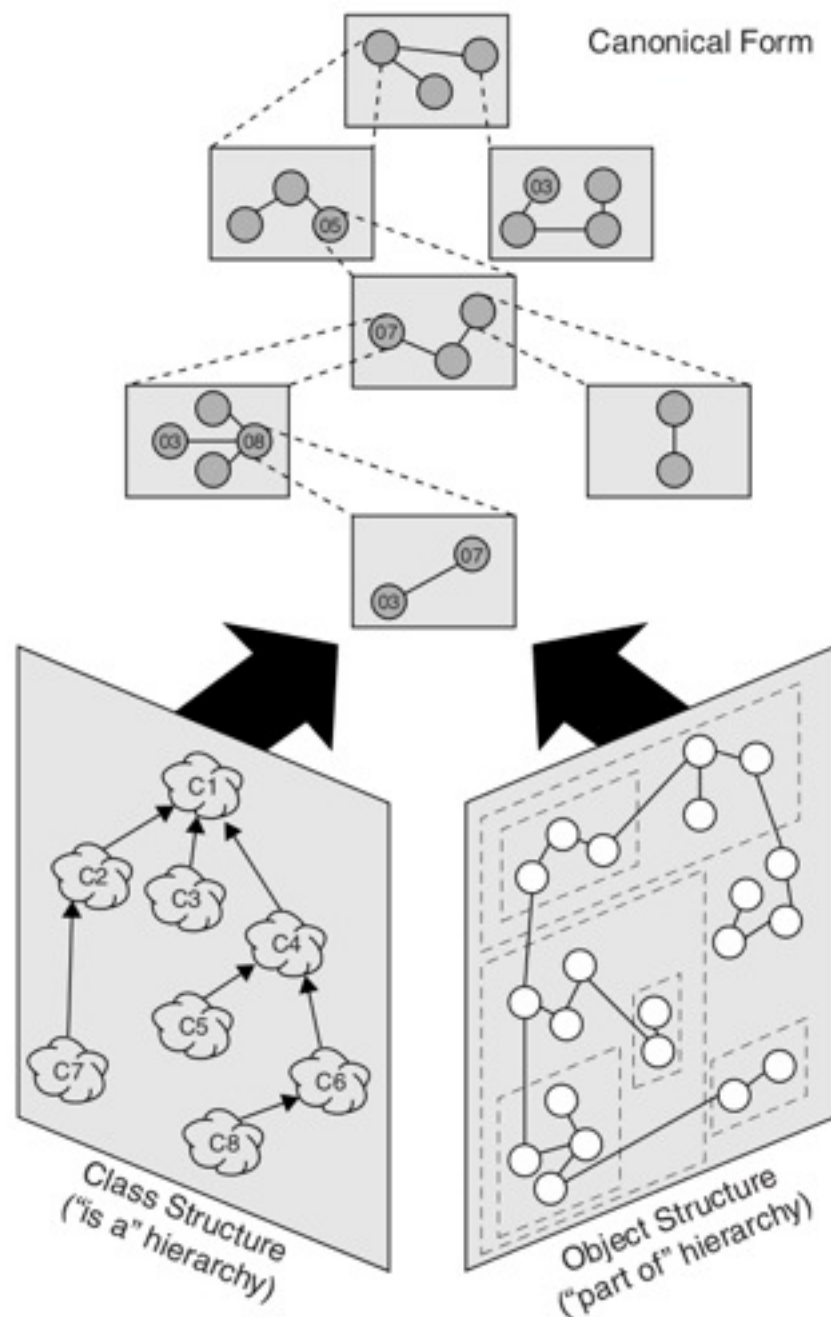
**Figure 1-1**  
The Canonical Form of a Complex System

- Hierarquia de classes
  - is-a: é um
- Hierarquia de objetos
  - part-of: parte de

Object-oriented Analysis  
and Design with Applications  
2ed. - Grady Booch



# As duas hierarquias de um sistema OO



Object-oriented Analysis  
and Design with Applications  
3ed. - Booch et. al.

@pythonprobr

Figure 1-2 The Canonical Form of a Complex System



# Timer



- Exemplo simples de composição

```
from Tkinter import Frame, Label, Button

class Timer(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.inicio = self.agora = 15
        self.pendente = None # alarme pendente
        self.grid()
        self.mostrador = Label(self, width=2, anchor='e',
                                font='Helvetica 120 bold',)
        self.mostrador.grid(column=0, row=0, sticky='nswe')
        self.bt_start = Button(self, text='Start', command=self.start)
        self.bt_start.grid(column=0, row=1, sticky='we')
        self.atualizar_mostrador()

    def atualizar_mostrador(self):
        self.mostrador['text'] = str(self.agora)

    def start(self):
        if self.pendente:
            self.after_cancel(self.pendente)
        self.agora = self.inicio
        self.atualizar_mostrador()
        self.pendente = self.after(1000, self.tictac)

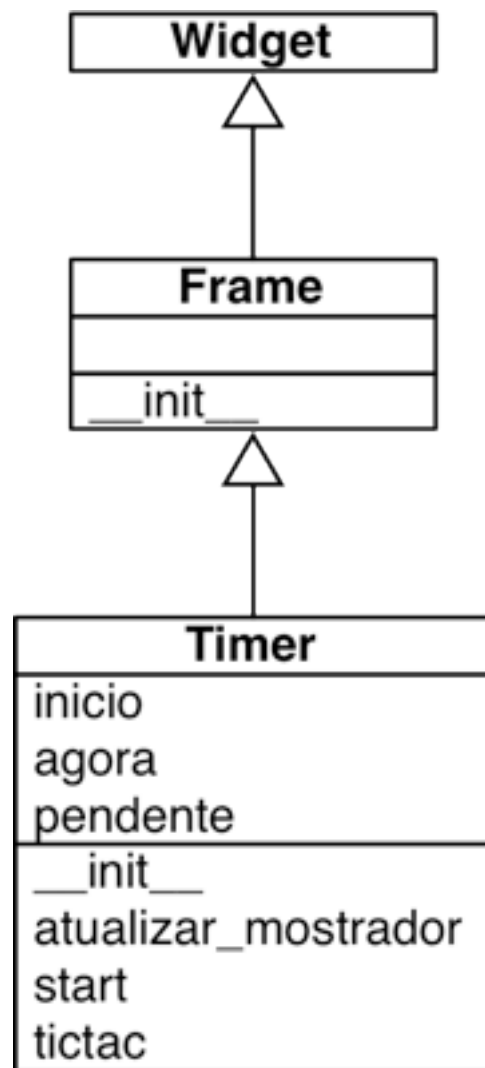
    def tictac(self):
        self.agora -= 1
        self.atualizar_mostrador()
        if self.agora > 0:
            self.pendente = self.after(1000, self.tictac)

timer = Timer()
timer.mainloop()
```



[oopy/exemplos/timer.py](https://github.com/honprobr/oopy/blob/master/exemplos/timer.py):honprobr

# Timer



```
from Tkinter import Frame, Label, Button
```

```
class Timer(Frame):
```

```
    def __init__(self):
```

```
        Frame.__init__(self)
```

```
        self.inicio = self.agora = 15
```

```
        self.pendente = None # alarme pendente
```

```
        self.grid()
```

```
        self.mostrador = Label(self, width=2, anchor='e',  
                                font='Helvetica 120 bold',)
```

```
        self.mostrador.grid(column=0, row=0, sticky='nswe')
```

```
        self.bt_start = Button(self, text='Start', command=self.start)
```

```
        self.bt_start.grid(column=0, row=1, sticky='we')
```

```
        self.atualizar_mostrador()
```

```
    def atualizar_mostrador(self):
```

```
        self.mostrador['text'] = str(self.agora)
```

```
    def start(self):
```

```
        if self.pendente:
```

```
            self.after_cancel(self.pendente)
```

```
        self.agora = self.inicio
```

```
        self.atualizar_mostrador()
```

```
        self.pendente = self.after(1000, self.tictac)
```

```
    def tictac(self):
```

```
        self.agora -= 1
```

```
        self.atualizar_mostrador()
```

```
        if self.agora > 0:
```

```
            self.pendente = self.after(1000, self.tictac)
```

```
timer = Timer()
```

```
timer.mainloop()
```



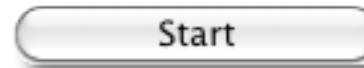
[oopy/exemplos/timer.py](https://oopy/exemplos/timer.py):honprobr

# Composição

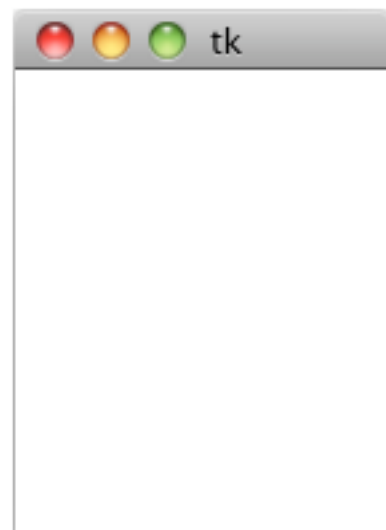
- Arranjo de partes de um sistema
- componentes, sub-componentes...



```
mostrador = Label()
```



```
bt_start = Button()
```



```
timer = Timer()
```

15

Start

instanciação

instanciação

instanciação

```
from Tkinter import Frame, Label, Button

class Timer(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.inicio = self.agora = 15
        self.pendente = None # alarme pendente
        self.grid()
        self.mostrador = Label(self, width=2, anchor='e',
                                font='Helvetica 120 bold',)
        self.mostrador.grid(column=0, row=0, sticky='nswe')
        self.bt_start = Button(self, text='Start', command=self.start)
        self.bt_start.grid(column=0, row=1, sticky='we')
        self.atualizar_mostrador()

    def atualizar_mostrador(self):
        self.mostrador['text'] = str(self.agora)

    def start(self):
        if self.pendente:
            self.after_cancel(self.pendente)
        self.agora = self.inicio
        self.atualizar_mostrador()
        self.pendente = self.after(1000, self.tictac)

    def tictac(self):
        self.agora -= 1
        self.atualizar_mostrador()
        if self.agora > 0:
            self.pendente = self.after(1000, self.tictac)

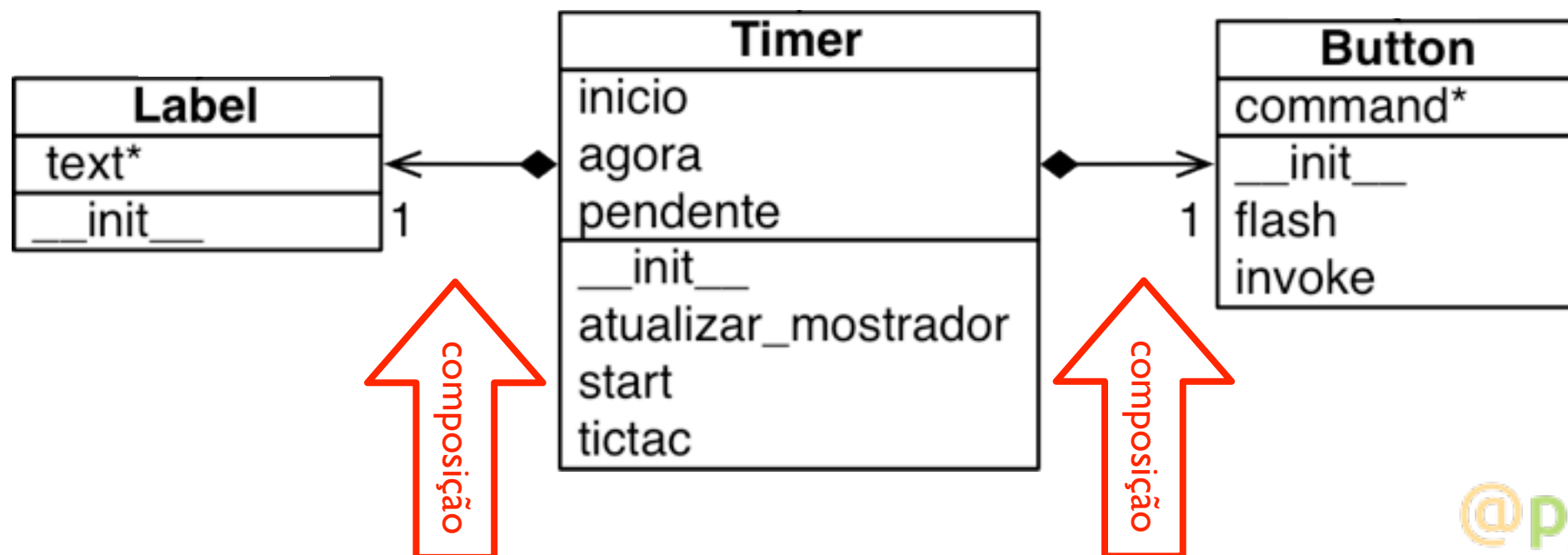
timer = Timer()
timer.mainloop()
```



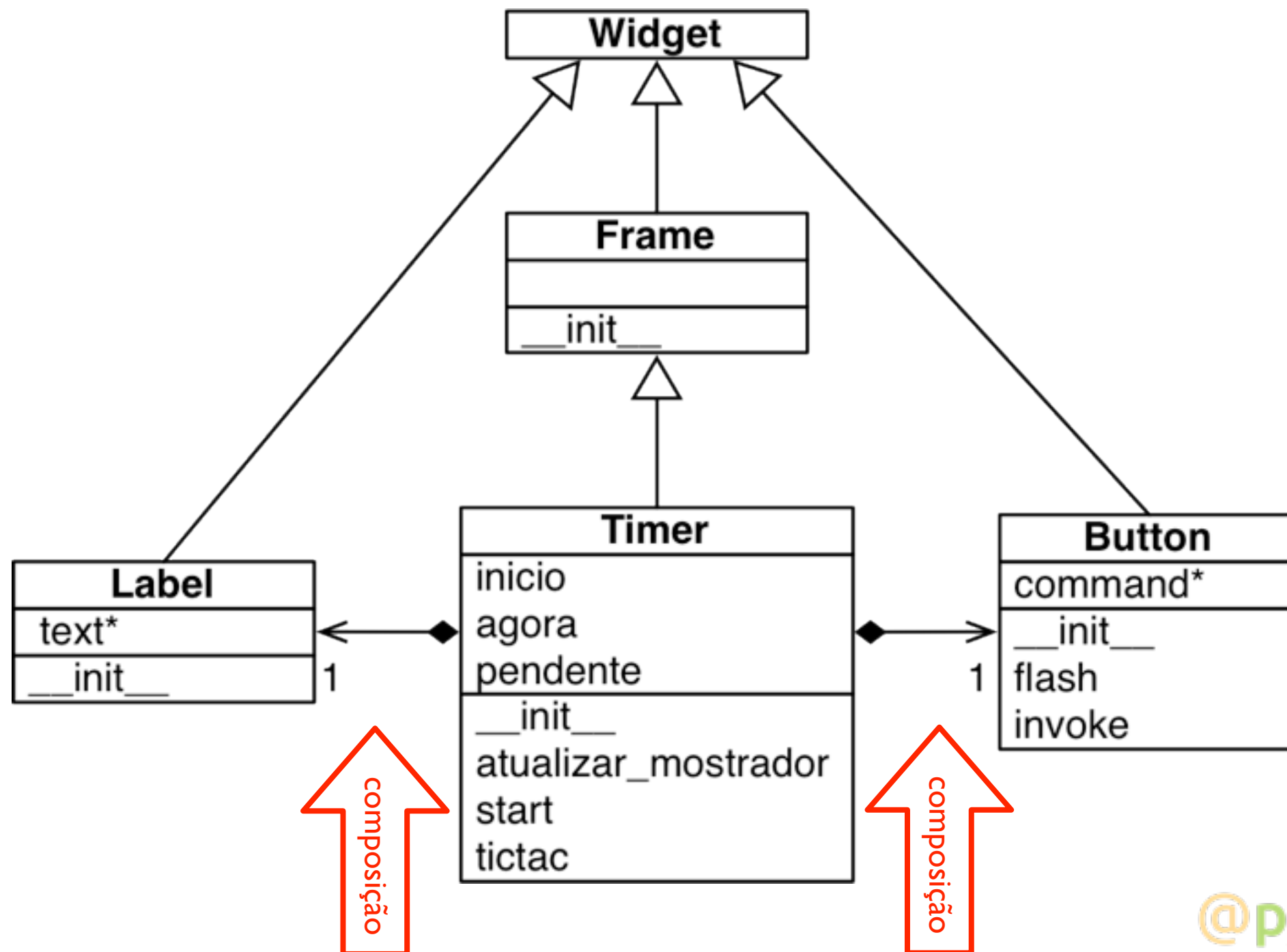
git

[oopy/exemplos/timer.py](https://oopy/exemplos/timer.py):honprobr

# Composição em UML



# Composição em UML



# Interface

- Interface é um conceito essencial em OO
- não depende de uma palavra reservada

A interface fornece uma separação entre a implementação de uma abstração e seus clientes. Ela limita os detalhes de implementação que os clientes podem ver. Também especifica a funcionalidade que as implementações devem prover.



# Interfaces e protocolos

- Em SmallTalk, as interfaces eram chamadas de “protocolos”.
- Não há verificação de interfaces na linguagem, mas algumas IDEs (“browsers”) permitem agrupar os métodos por protocolo para facilitar a leitura
- Um protocolo é uma interface informal, não declarada porém implementada por métodos concretos

# Interfaces em Python

- Conceitualmente, sempre existiram como protocolos
- Não havia maneira formal de especificar interfaces em Python até a versão 2.5
- usava-se termos como “uma sequência” ou “a file-like object”
- Agora temos ABC (Abstract Base Class)
- com herança múltipla, como em C++

# Exemplo: tómbola

- Sortear um a um todos os itens de uma coleção finita, sem repetir
- A mesma lógica é usada em sistemas para gerenciar banners online



# Interface da tómbola

- Carregar itens
- Misturar itens
- Sortear um item
- Indicar se há mais itens



# Projeto da tómbola

Tômbola
itens
carregar
carregada
misturar
sortear

- UML:  
diagrama de classe





# TDD: Test Driven Design

- Metodologia de desenvolvimento iterativa na qual, para cada funcionalidade nova, um teste é criado antes do código a ser implementado
- Esta inversão ajuda o programador a desenvolver com disciplina apenas uma funcionalidade de cada vez, mantendo o foco no teste que precisa passar
- Cada iteração de teste/implementação deve ser pequena e simples: “baby steps” (passinhos de bebê)

# Doctests

- Um dos módulos para fazer testes automatizados na biblioteca padrão de Python
  - o outro módulo é o unittest, da família xUnit
- Doctests foram criados para testar exemplos embutidos na documentação
- Usaremos doctests para especificar exercícios
- Exemplo: `$ python -m doctest cao_test.rst`



# TDD:

## demonstração ao vivo

- Implementação da classe Tombola, com testes feitos em Doctest

# Implementação da tômbola

Tômbola
itens
carregar
carregada
misturar
sortear

- Python 2.2 a 2.7

```
# coding: utf-8

import random

class Tombola(object):
    itens = None

    def carregar(self, itens):
        self.itens = list(itens)

    def carregada(self):
        return bool(self.itens)

    def misturar(self):
        random.shuffle(self.itens)

    def sortear(self):
        return self.itens.pop()
```

# Implementação da tômbola

Tômbola
itens
carregar
carregada
misturar
sortear

- Python 3.x

```
import random

class Tombola:
    itens = None

    def carregar(self, itens):
        self.itens = list(itens)

    def carregada(self):
        return bool(self.itens)

    def misturar(self):
        random.shuffle(self.itens)

    def sortear(self):
        return self.itens.pop()
```