

Desenvolvimento de Sistemas Orientados a Objetos I

Classes Abstratas e Interfaces

Jean Carlo Rossa Hauck, Dr.

jean.hauck@ufsc.br

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

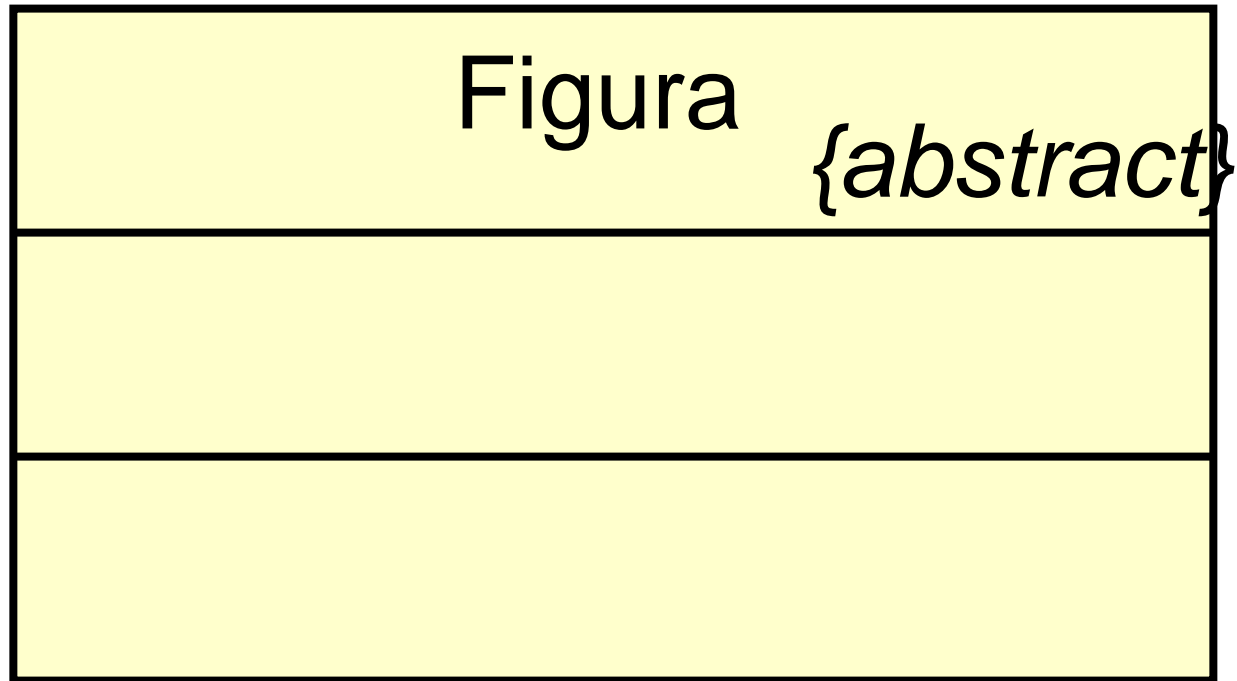
Conteúdo Programático

- Conceitos e mecanismos da programação orientada a objetos
 - Herança e polimorfismo
 - Classes abstratas

Classe abstrata

- Classe que não irá ter objetos instanciados
 - Quando não faz sentido criar objetos diretamente (por exemplo, objetos da classe Pessoa)
- Pode ser uma classe completa, incluindo atributos, operações e métodos
- Oferece a base para uma hierarquia de classes
 - Oferece um conjunto de operações e métodos comuns a todas as subclasses
- Interessante para uso de polimorfismo

Classe abstrata na UML



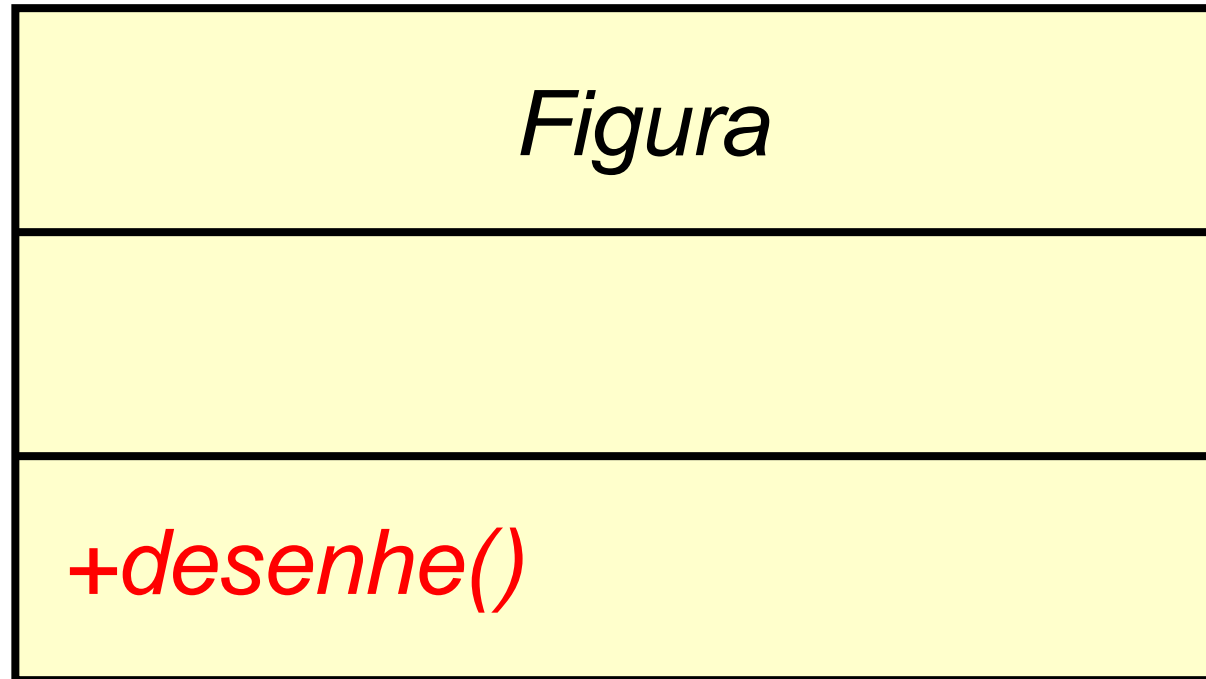
Classe abstrata na UML

Figura

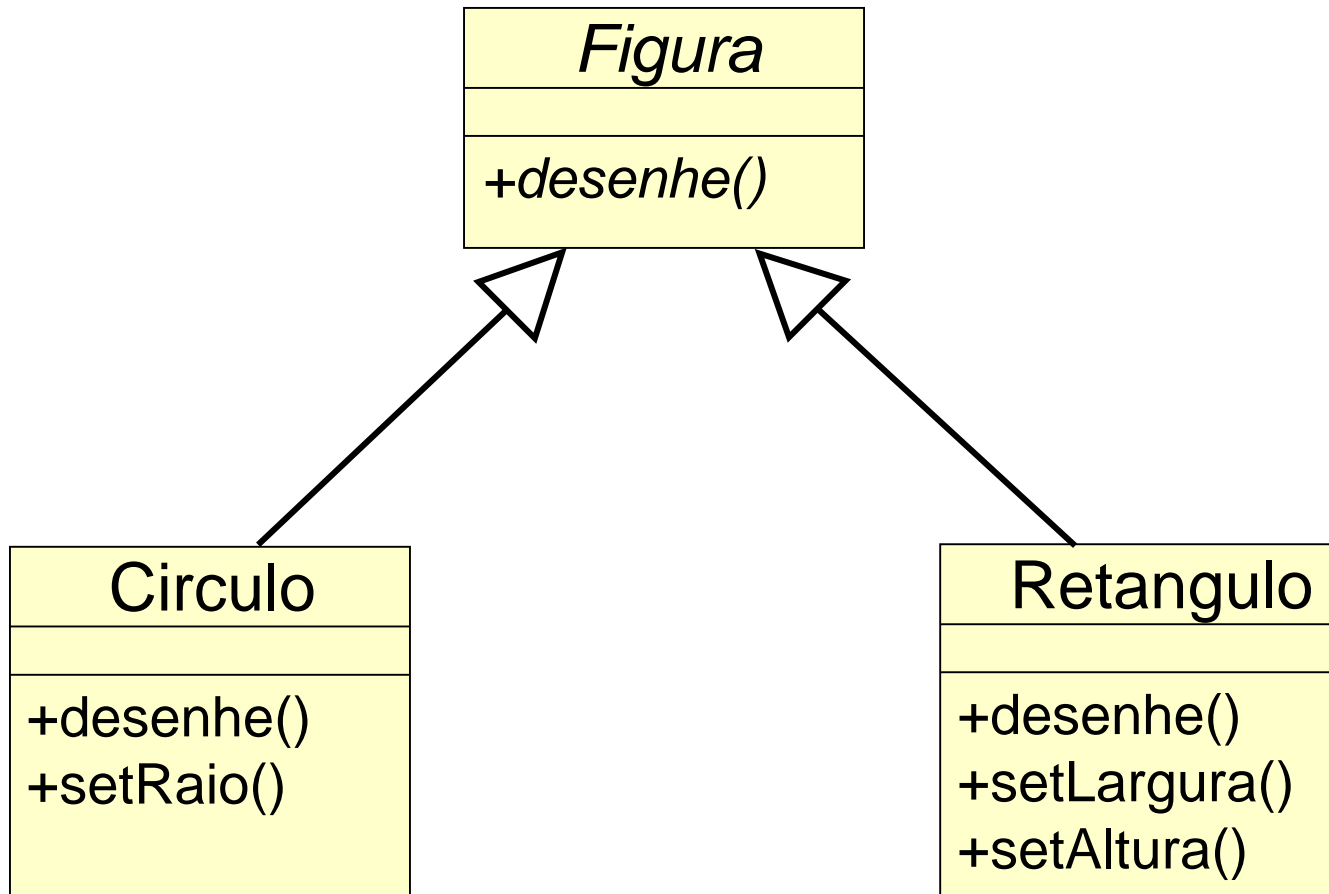
Operação abstrata

- ❑ Operação que não possui um método (somente o nome da operação, sem o corpo do método)
- ❑ A implementação da operação é delegada para as subclasses
- ❑ Usada com polimorfismo (por exemplo, padrão de projeto **template**)

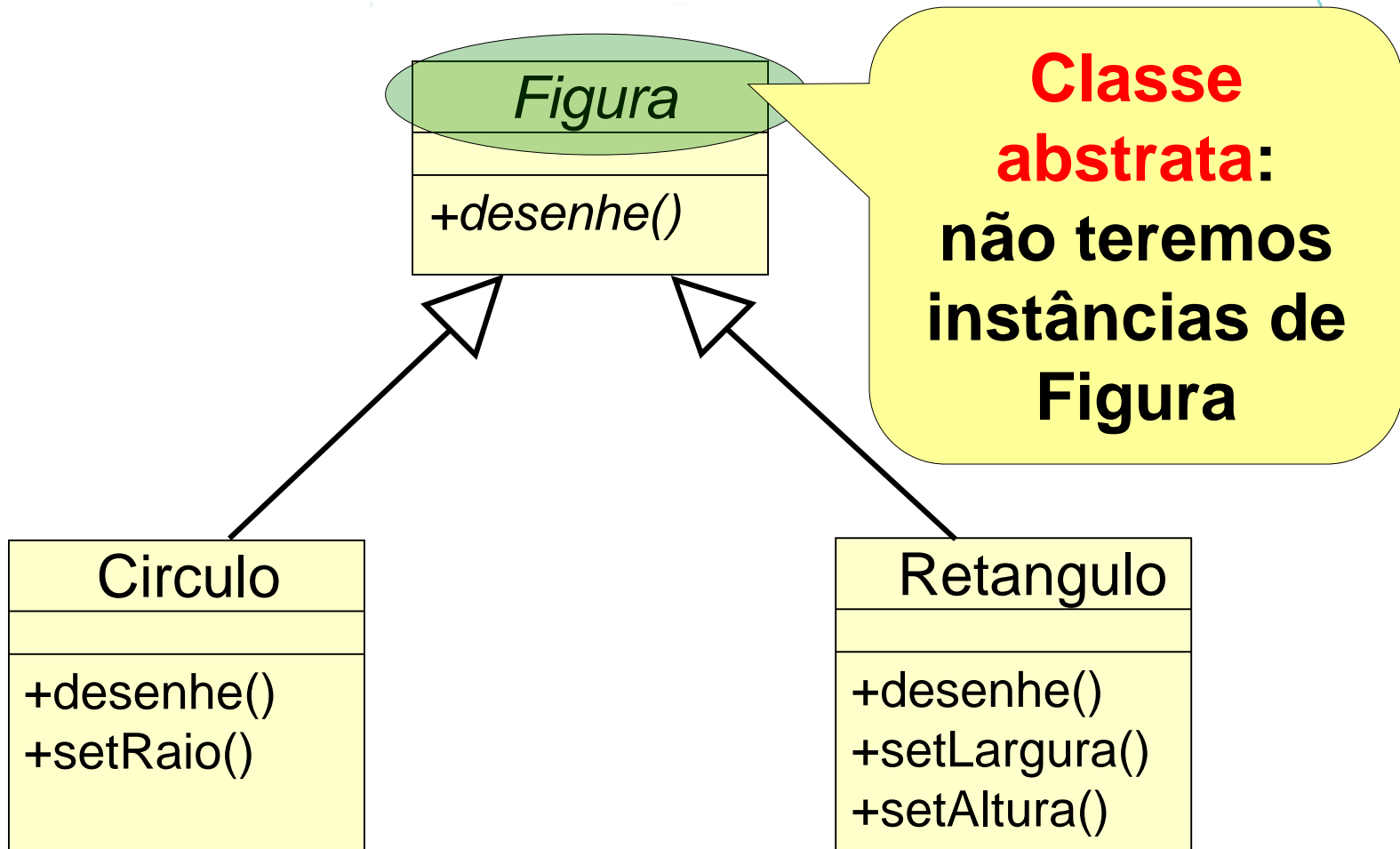
Operação abstrata na UML



Classes e operações abstratas: exemplo

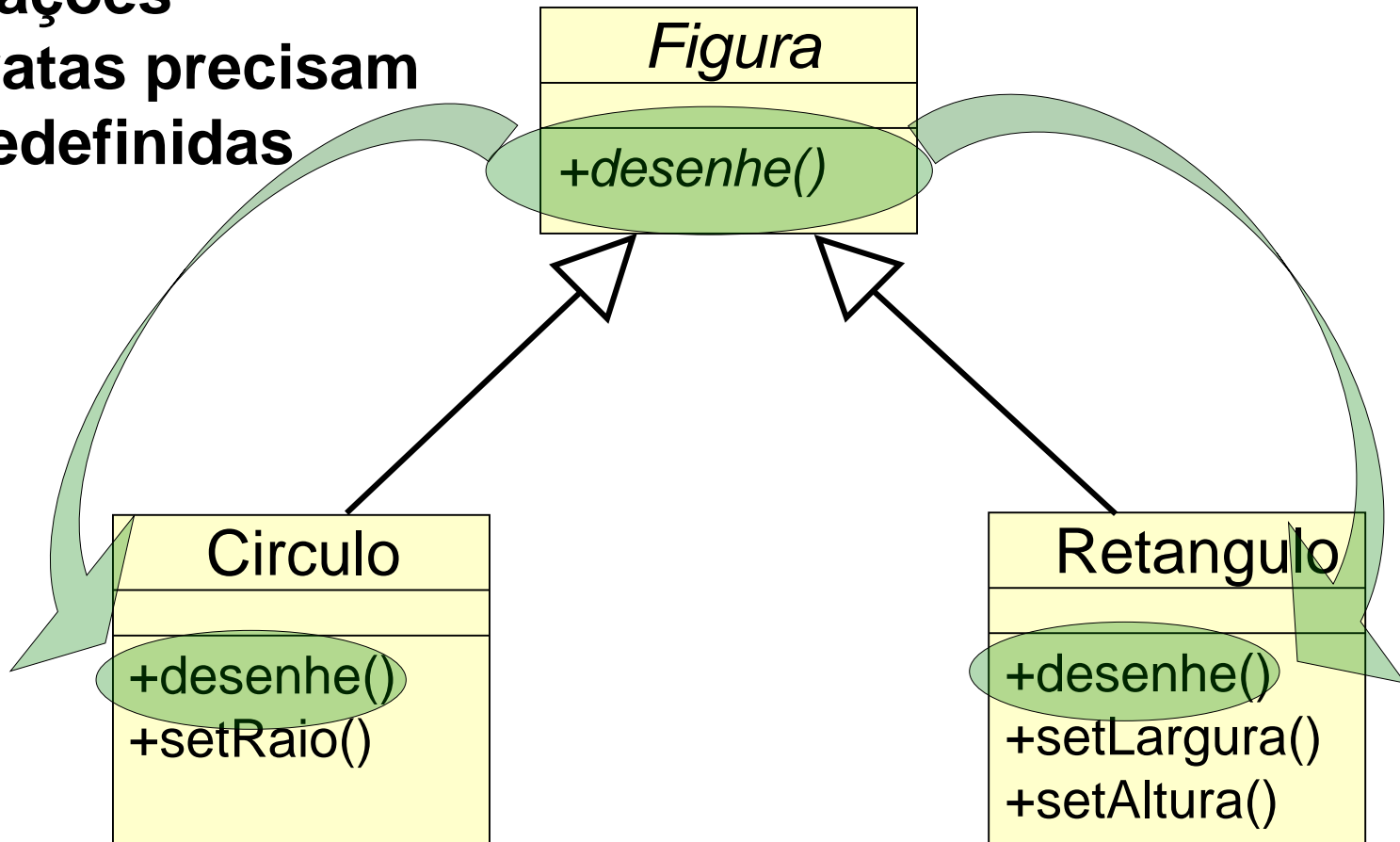


Classes e operações abstratas: exemplo



Classes e operações abstratas: exemplo

Operações
abstratas precisam
ser redefinidas



A classe Figura

```
from abc import ABC, abstractmethod

class Figura(ABC):
    @abstractmethod
    def __init__(self):
        pass

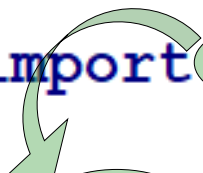
    @abstractmethod
    def desenhe(self):
        pass
```

A classe Figura

```
from abc import ABC, abstractmethod

class Figura(ABC):
    @abstractmethod
    def __init__(self):
        pass

    @abstractmethod
    def desenha(self):
        pass
```

A green curved arrow points from the `ABC` in the `from abc import ABC, abstractmethod` line to the `(ABC)` in the `class Figura(ABC):` line, indicating inheritance.

**Herança de
ABC indica
que a Classe é
abstrata**

A classe Figura

```
from abc import ABC, abstractmethod
```

```
class Figura(ABC):
```

```
    @abstractmethod
```

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

```
    @abstractmethod
```

```
    def desenha(self):  
        pass
```

@abstractmethod
no construtor
impede **instanciar**
a classe

A classe Figura

```
from abc import ABC, abstractmethod
```

```
class Figura(ABC):  
    @abstractmethod  
    def __init__(self):  
        pass
```

```
    @abstractmethod  
    def desenha(self):  
        pass
```

**Indica que o
método é
abstrato**

A classe Circulo

```
class Circulo(Figura):  
    def __init__(self, raio: int):  
        super().__init__()  
        self.__raio = raio  
  
    @property  
    def raio(self):  
        return self.__raio  
  
    @raio.setter  
    def raio(self, raio):  
        self.__raio = raio  
  
    def desenhe(self):  
        return "circulo de raio {0:d}".format(self.__raio)
```

A classe Circulo

```
class Circulo(Figura):  
    def __init__(self, raio: int):  
        super().__init__()  
        self.__raio = raio  
  
    @property  
    def raio(self):  
        return self.__raio  
  
    @raio.setter  
    def raio(self, raio):  
        self.__raio = raio  
  
    def desenhe(self):  
        return "circulo de raio {0:d}".format(self.__raio)
```

**Implementação
obrigatória dos
métodos que eram
abstratos na
classe-pai**

A classe Retangulo

```
class Retangulo(Figura):
    def __init__(self, lado1=0, lado2=0):
        super().__init__()
        self.__lado1 = lado1
        self.__lado2 = lado2

    @property
    def lado1(self):
        return self.__lado1

    @lado1.setter
    def lado1(self, lado1):
        self.__lado1 = lado1

    ...

    def desenha(self):
        return "retangulo com lados {0:d} e {1:d}".\
            format(self.__lado1, self.__lado2)
```

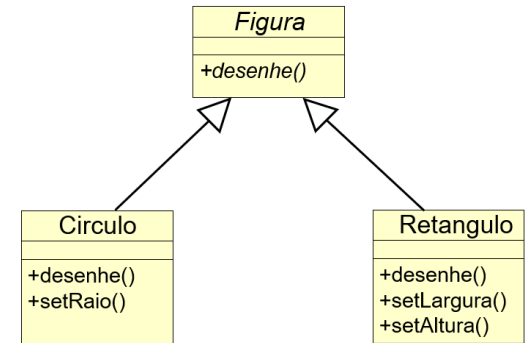
A classe Retangulo

```
class Retangulo(Figura):  
    def __init__(self, lado1=0, lado2=0):  
        super().__init__()  
        self.__lado1 = lado1  
        self.__lado2 = lado2  
  
    @property  
    def lado1(self):  
        return self.__lado1  
  
    @lado1.setter  
    def lado1(self, lado1):  
        self.__lado1 = lado1  
  
    ...  
  
    def desenha(self):  
        return "retangulo com lados {0:d} e {1:d}".\n            format(self.__lado1, self.__lado2)
```

**Implementação
obrigatória dos
métodos que eram
abstratos na
classe-pai**

Usando Polimorfismo com as classes

```
figuras = []  
  
retangulo = Retangulo(1, 2)  
  
circulo = Circulo(2)  
  
figuras.append(retangulo)  
figuras.append(circulo)  
  
for figura in figuras:  
    print(figura.desenhe())
```



Usando Polimorfismo com as classes

```
figuras = []
```

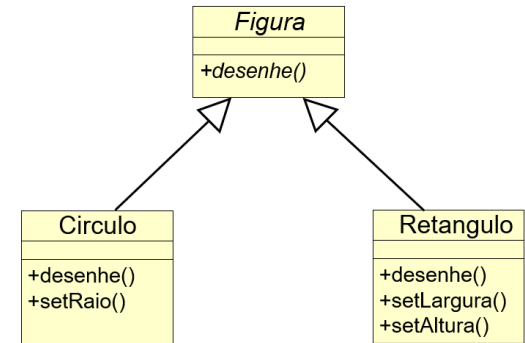
```
retangulo = Retangulo(1, 2)
```

```
circulo = Circulo(2)
```

```
figuras.append(retangulo)
```

```
figuras.append(circulo)
```

```
for figura in figuras:  
    print(figura.desenhe())
```



**Polimorfismo:
garantido pela
herança de
Figura**

Polimorfismo

Princípio pelo qual, objetos de duas ou mais **classes derivadas** de uma mesma **superclasse** podem invocar **operações que têm a mesma assinatura mas comportamentos distintos**, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse

Polimorfismo – Riscos!

```
figuras = []
```

```
retangulo = Retangulo(1, 2)
```

```
circulo = Circulo(2)
```

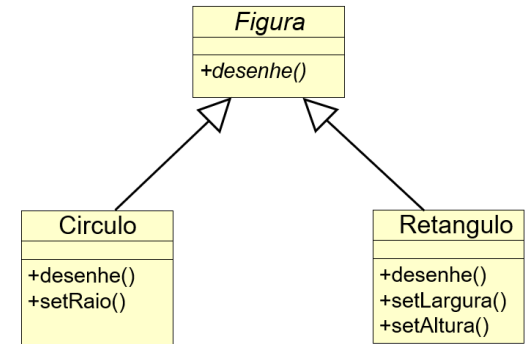
```
pessoa = Pessoa("Jean")
```

```
figuras.append(retangulo)
```

```
figuras.append(circulo)
```

```
figuras.append(pessoa)
```

```
for figura in figuras:  
    print(figura.desenhe())
```



Polimorfismo – Riscos!

```
figuras = []
```

```
retangulo = Retangulo(1, 2)
```

```
circulo = Circulo(2)
```

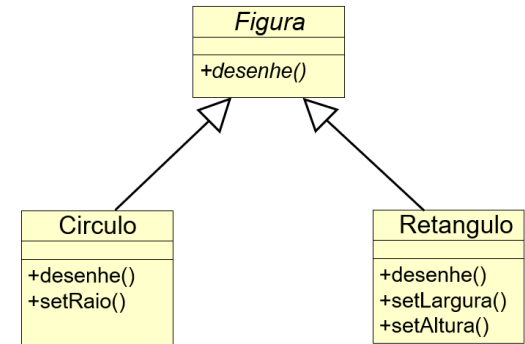
```
pessoa = Pessoa("Jean")
```

```
figuras.append(retangulo)
```

```
figuras.append(circulo)
```

```
figuras.append(pessoa)
```

```
for figura in figuras:  
    print(figura.desenhe())
```



O que acontece aqui?

Polimorfismo – Riscos!

```
figuras = []
```

```
retangulo = Retangulo(1, 2)
```

```
circulo = Circulo(2)
```

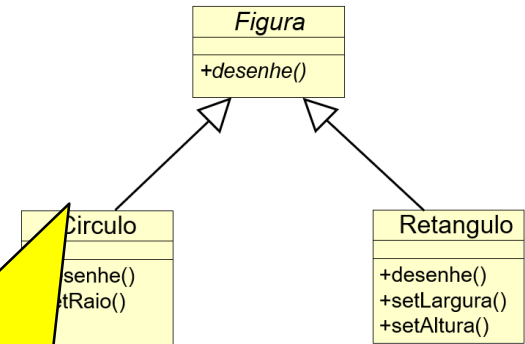
```
pessoa = Pessoa()
```

```
figuras.append(pessoa)
```

```
figuras.append(pessoa)
```

```
figuras.append(pessoa)
```

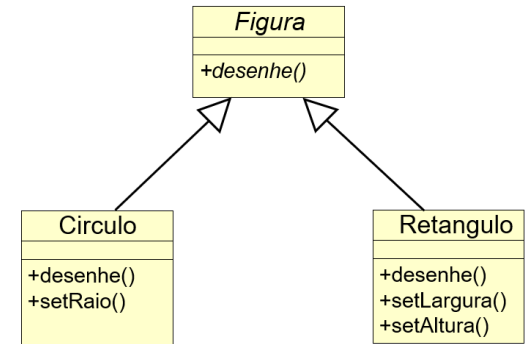
```
for figura in figuras:  
    print(figura.desenhe())
```



**AttributeError: 'Pessoa' object
has no attribute 'desenhe'**

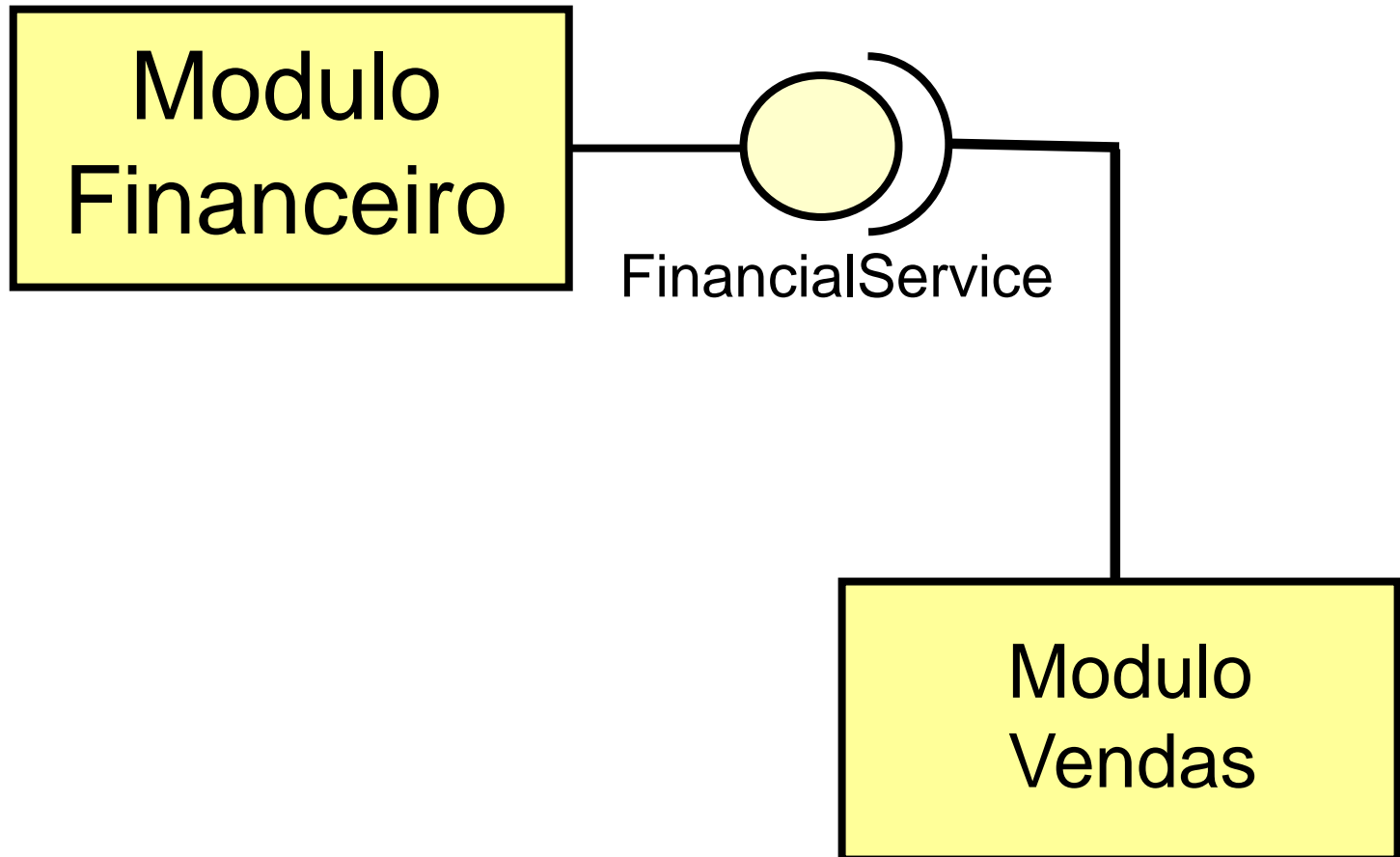
Polimorfismo – Riscos!

```
figuras = []  
  
retangulo = Retangulo(1, 2)  
  
circulo = Circulo(2)  
  
pessoa = Pessoa("Jean")  
  
figuras.append(retangulo)  
figuras.append(circulo)  
figuras.append(pessoa)  
  
for figura in figuras:  
    print(figura.desenhe())
```

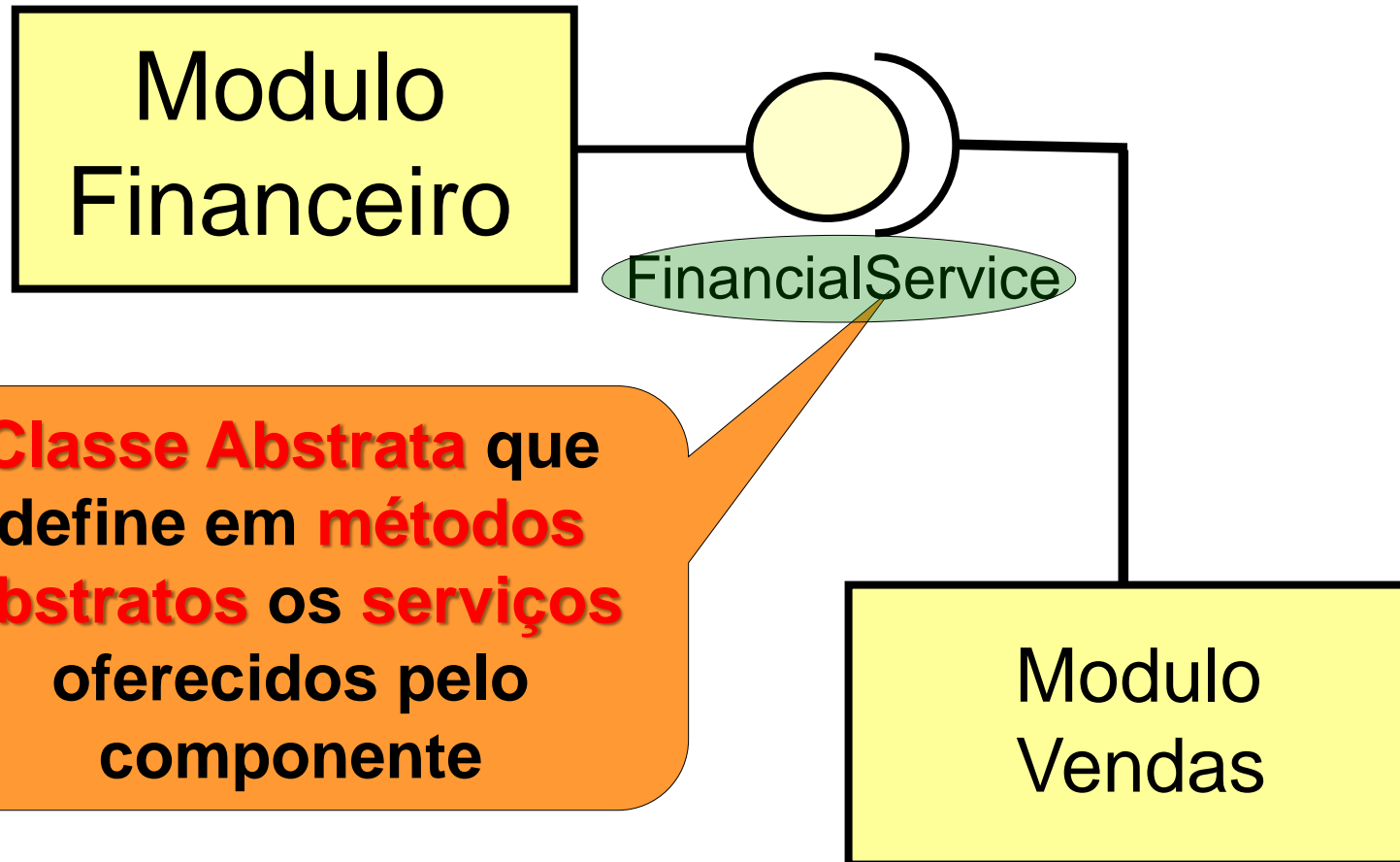


**Não herda de
Figura e não
implementa
“desenhe()”**

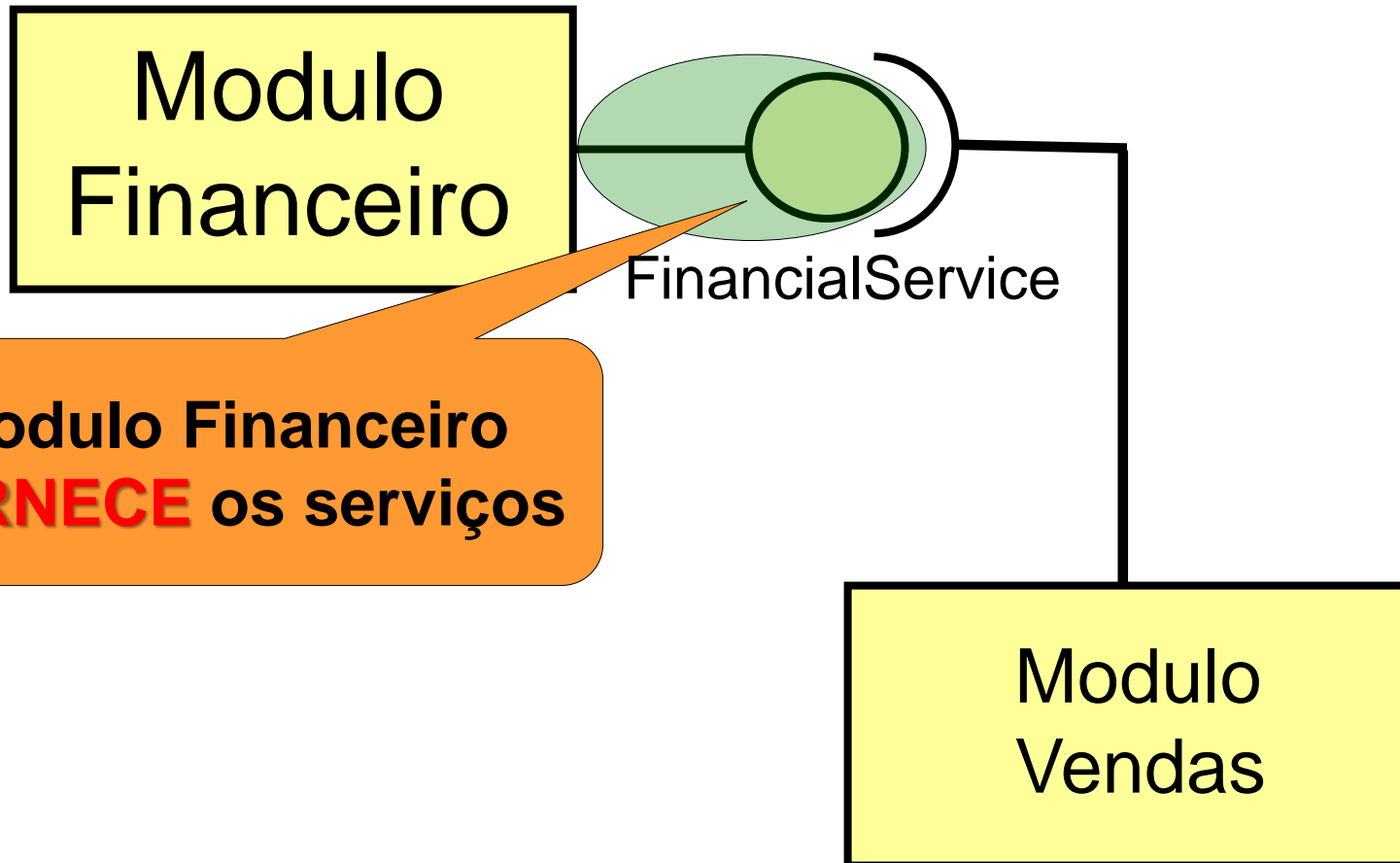
Classe abstrata - Interface de Componente



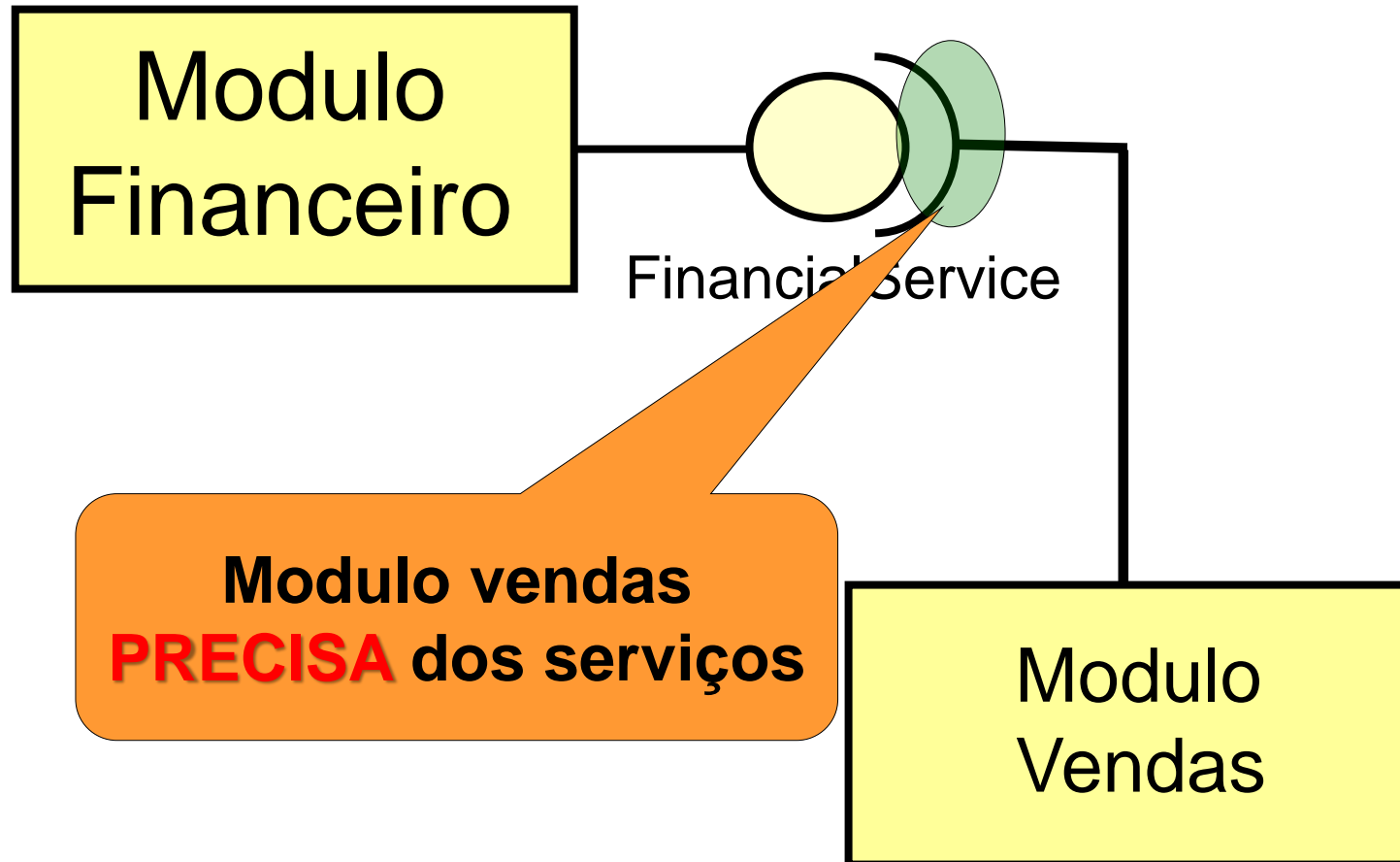
Classe abstrata - Interface de Componente



Classe abstrata - Interface de Componente



Classe abstrata - Interface de Componente



Porque utilizar Classes Abstratas

- Reduz o acoplamento entre classes, aumentando a sua reusabilidade
- Permite que componentes possam ter diferentes interfaces de acordo com as necessidades dos seus usuários
- Ajuda a esconder a complexidade da arquitetura interna de componentes

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.