Computação II (MAB 225)

Fabio Mascarenhas - 2015.1

http://www.dcc.ufrj.br/~fabiom/pythonoo

Observadores ou listeners

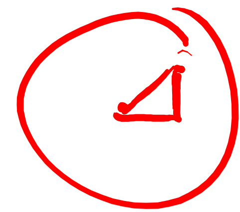
- Usamos o padrão observador quando queremos desacoplar objetos que *emitem* eventos dos objetos que os consomem

- A fonte ou sujeito dos eventos pode implementar uma interface que permite aos observadores se cadastrar para receber eventos, e se descadastrar, mas também é comum ter sujeitos com um único observador

- Os observadores ou *listeners* implementam uma interface que permite que sejam notificados caso algum evento ocorra

- É um padrão muito usado em interfaces gráficas

Exemplos de observadores



- Já usamos muitos observadores:
  - A interface `Jogo` é em parte um observador, assim como as interfaces `App` e `Componente`
  - A interface Acao para conectar um botão ao que fazer quando é clicado
  - A classe `ObservadorSlider` para receber mudanças no valor de um slider
  - A classe `ObservadorCanvas` para receber eventos do canvas
- As classes que formam o *controlador* de nosso editor gráfico todas são observadores

Undo/redo e o padrão Comando

- Para implementar a funcionalidade de desfazer/refazer do editor de figuras, vamos usar o padrão *Comando*
- Um comando é um objeto que representa uma ação da aplicação; para uma aplicação típica, qualquer coisa que podemos desfazer
- As instâncias de comando encapsulam toda a informação necessária para desfazer a ação, ou refazê-la
- Com isso, implementar desfazer/refazer no modelo é só uma questão de manter uma *pilha* de ações feitas e outra pilha de ações desfeitas

Aplicações sem modelo

- Para uma aplicação simples como essa, poderíamos ter dispensado o modelo, como fizemos com o Breakout

- Ainda valeria a pena organizar a aplicação em componentes gráficos, figuras, comandos, estados e ações, mas o "modelo" estaria fundido ao "controlador"

- Usamos o MVC quando queremos trocar complexidade por flexibilidade: quando bem arquitetado, o modelo pode ser reaproveitado

- Vamos ver isso na prática usando nossos modelos em uma aplicação com uma camada visão-controlador radicalmente diferente

Classes abstratas

- Até agora, usamos *interfaces* toda vez que queríamos representar algum conceito abstrato em nosso programa, não importa a forma como ele era implementado

- Programar com interfaces é flexível, mas a restrição de só podermos ter assinaturas de métodos em uma interface às vezes é inconveniente, e pode levar a duplicação de código

- Para contornar isso, temos um segundo mecanismo para representar objetos abstratos: as *classes abstratas*

Classes abstratas - sintaxe

- Uma classe abstrata é como qualquer outra classe, exceto que o corpo de alguns de seus métodos dispara um *erro de método não implementado*:

```
class FiguraPt:
    def mover(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

    def desenhar(self, canvas):
        raise NotImplementedError()

    def dentro(self, x, y):
        raise NotImplementedError()
```

método abstrato

Classes abstratas – métodos abstratos

- Uma classe abstrata pode ter campos e métodos, como qualquer outra classe, mas não devemos instanciar uma classe abstrata, mesmo se ela tiver um construtor `__init__`

- Os métodos que declaramos como `raise NotImplementedError()` são *métodos abstratos*

- A ideia é que os métodos abstratos são métodos que vão ser definidos depois, nas classes concretas que *derivam* da classe abstrata

- Se a classe abstrata não tiver um construtor, a classe concreta também deve fornecer um, e inicializar os campos pedidos pela classe abstrata, assim como seus campos em particular

Usando classes abstratas - herança

- Se não devemos instanciar uma classe abstrata diretamente, para ter instâncias dela criamos uma classe concreta que *deriva* ou *herda* da classe abstrata

deriva de
herda de

```
class Retangulo(FiguraPt):
    def __init__(self, x, y, largura, altura, cor):
        self.x = x
        self.y = y
        self.largura = largura
        self.altura = altura
        self.cor = cor

    def desenhar(self, canvas):
        canvas.retangulo(self.x, self.y, self.largura, self.altura, self.cor)

    def dentro(self, x, y):
        return (x >= self.x and x <= self.x + self.largura and
                y >= self.y and y <= self.y + self.altura)
```

Herança

- A relação de herança, como a implementação de uma interface, também é uma relação “é-um”

- Uma instância de `Retangulo` *é uma* instância de `FiguraPt`

- As relações “é um” são transitivas: uma instância de `Retangulo` é uma instância de `FiguraPt` (via herança), e uma instância de `FiguraPt` é uma instância da interface `Figura` (por implementação de seus métodos), então uma instância de `Retangulo` é uma instância de `Figura`

- A diferença da herança é que nela a classe herda a *forma* da outra classe, incluindo todos os seus campos, seus métodos, e seu construtor

Redefinição de métodos

- A subclasse não está restrita a só fornecer construtores e implementações para métodos abstratos
- Ela também pode *redefinir* métodos, dando uma outra implementação para eles
- Uma redefinição tem a mesma assinatura do método que está sendo redefinido
- Instâncias da subclasse usam sempre a nova implementação do método
- Dentro da redefinição de um método, podemos chamar sua definição original com <nome da superclasse>.<nome do método>(self, <outros argumentos>)

FiguraPt e Retangulo, com redefinição

```
class FiguraPt:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.cor = cor

    def mover(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

    def desenhar(self, canvas):
        raise NotImplementedError()

    def dentro(self, x, y):
        raise NotImplementedError()
```

```
class Retangulo(FiguraPt):
    def __init__(self, x, y, largura, altura, cor):
        FiguraPt.__init__(self, x, y)
        self.largura = largura
        self.altura = altura

    def desenhar(self, canvas):
        canvas.retangulo(self.x, self.y,
                         self.largura,
                         self.altura, self.cor)

    def dentro(self, x, y):
        return (x >= self.x and
                x <= self.x + self.largura and
                y >= self.y and
                y <= self.y + self.altura)
```