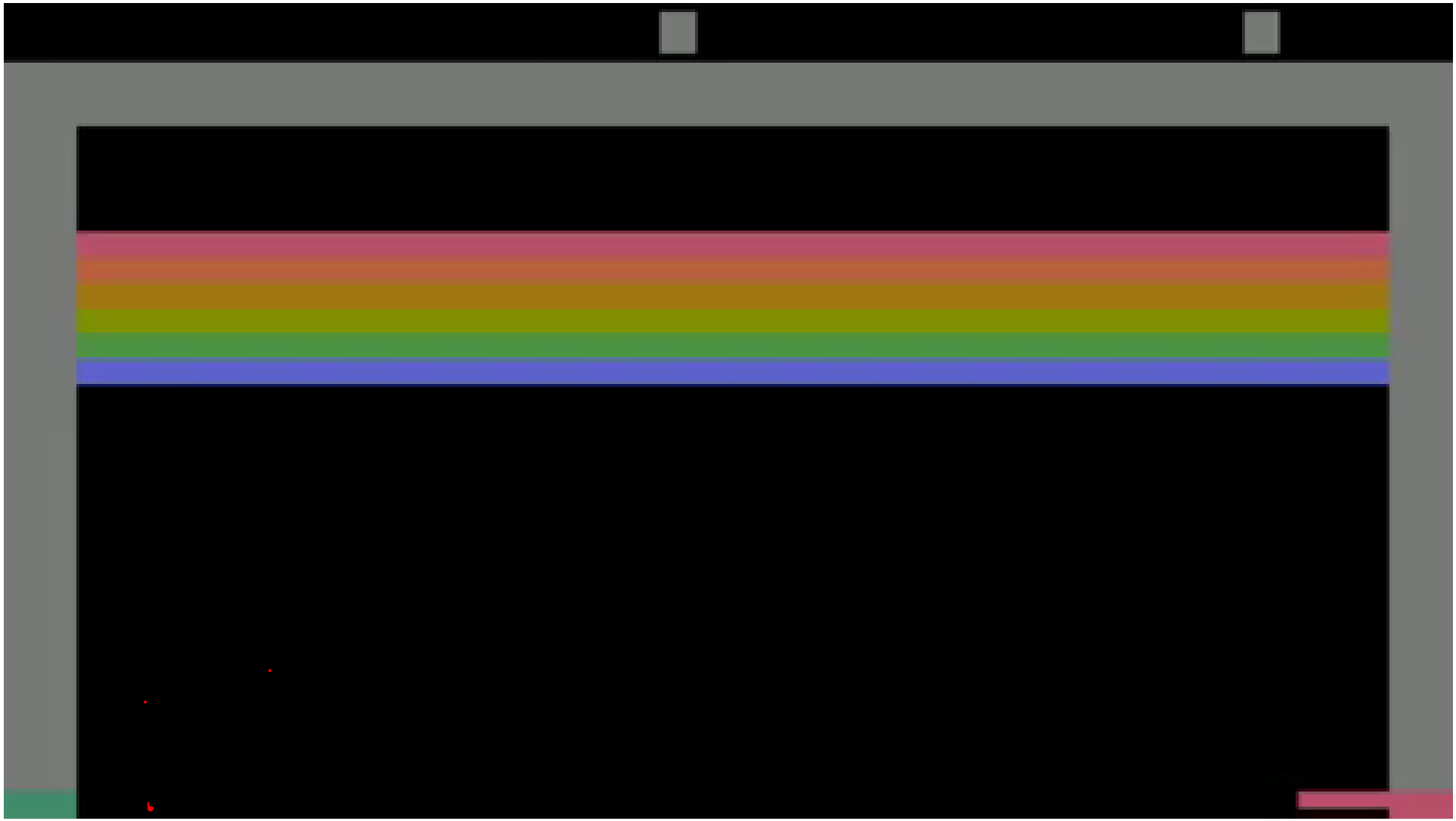
Computação II (MAB 225)

Fabio Mascarenhas - 2015.1

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

Breakout



Múltiplos tijolos

- Os tijolos do Breakout só variam na posição e na cor, mas e se quiséssemos ter tijolos com *comportamento* diferente?
  - Tijolos que quando destruídos dão mais pontos
  - Tijolos que precisam de mais de um “hit” para serem destruídos
  - Tijolos que aceleram ou retardam a bola
  - Tijolos que precisam ser acertados em um canto específico para serem destruídos
  - ...

Interfaces

- Como ficaria a classe Tijolo que permitisse todas essas variações?
  - Campos que são usados por uma variante mas não por outra
  - Um campo “tag” que indica qual variante esse tijolo específico é
  - Métodos que inspecionam a tag para saber o que fazer
- Uma solução é ter diferentes classes para os diferentes tipos de Tijolo
- Um tijolo deixa de ser uma instância de uma classe específica para ser uma instância de qualquer classe que implementa determinada *interface*

Interfaces, cont.

- Uma *interface* é uma forma abstrata de descrever um objeto
- Ela determina um conjunto mínimo de campos e métodos que uma classe que implementa aquela interface deve ter → sem implementação
- A ideia é que, se estamos usando apenas esse conjunto mínimo, qualquer instância de qualquer classe que implementa a interface vai servir, e não apenas instâncias de uma classe específica
- Algumas linguagens possuem suporte sintático para interfaces, mas em Python elas são apenas uma convenção

Uma interface para tijolos

- Um tijolo precisa se desenhar, verificar se a bola está colidindo com ele, retornando um sinal dizendo se ele deve ser destruído ou não, e dizer quantos pontos ele vale

- Isso se traduz em nos métodos **`colisao(`*`self, bola`*`)`** e **`desenhar(`*`self, tela`*`)`** e no campo **`pontos`**

- Precisamos refatorar nossa classe `Tijolo` para respeitar essa interface, e agora estamos livres para criar outras classes, como `TijoloFurado` (um tijolo com com um buraco no meio por onde a bola pode passar) e `TijoloVidas` (um tijolo que precisa ser atingido um certo número de vezes até ser destruído)

A interface Jogo

- Nosso framework para jogos 2D já define uma interface que o Breakout está implementando
- Essa interface é composta dos campos `TITULO`, `ALTURA` e `LARGURA`, e dos métodos tique, tecla e desenhar
- Qualquer classe que implemente essa interface pode ser passada para `motor.rodar`
- Mais tarde vamos estender essa interface com outros métodos

Polimorfismo

- Polimorfismo é poder operar com objetos diferentes de maneira uniforme, mesmo que cada objeto implemente a operação de uma maneira particular; basta que a assinatura da operação seja a mesma para todos os objetos

- Em programas OO reais, é muito comum que todas as operações sejam chamadas em referências para as quais só vamos saber qual classe concreta o objeto vai ter em tempo de execução

- Vamos ver muitas aplicações diferentes de polimorfismo ao longo do curso

Múltiplas Fases

- Se o jogador conseguir quebrar todos os tijolos, podemos querer muda-lo para uma outra fase de jogo
    - Layout diferente dos tijolos, velocidade diferente da bola, largura diferente da raquete...
    - Uma fase é como se fosse um novo jogo
- Como podemos fazer isso sem mudar `motor`?

Estado

- Com o padrão *Estado*, podemos mudar o comportamento de um objeto enquanto o programa está rodando
- A ideia é fazer o objeto delegar seu comportamento para um objeto *estado*
- Trocamos o estado, trocarmos o comportamento
- Para isso, todos os estados implementam uma interface comum
- Em nosso exemplo, as fases serão os diferentes estados

A interface Fase

- Uma fase é como um jogo: precisa responder a eventos do teclado e à passagem do tempo e precisa se desenhar
- Podemos simplesmente reaproveitar os métodos `tique`, `tecla` e `desenhar`
- A classe principal do jogo passa apenas a coordenar a passagem de uma fase para a outra

```
class Breakout:
    def __init__(self):
        self.fases = [PrimeiraFase,
                      SegundaFase,
                      TerceiraFase]
        self.i = 0
        self.fase = self.fases[self.i](self)

    def proxima_fase(self):
        self.i = self.i + 1
        self.fase = self.fases[selft.i](self)

    def tique(self, dt, teclas):
        self.fase.tique(dt, teclas)

    def tecla(self, tecla):
        self.fase.tecla(tecla)

    def desenhar(self, tela):
        self.fase.desenhar(tela)
```

Interfaces e abstrações

- Interfaces são uma ferramenta poderosa de *abstração*: representar um conceito pelas suas características essenciais

- Com elas, podemos decompor nossos problemas em pequenas partes genéricas

- Vamos ver um exemplo prático de como mesmo uma interface simples pode ser combinada de maneiras poderosas: *funções reais de uma variável*

- A interface `Funcao` é dada por um único método `valor`, que recebe um número real `x` como parâmetro e retorna o valor da função em `x`