Computação II – Orientação a Objetos

Fabio Mascarenhas - 2016.1

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

Interfaces Gráficas

- Vamos usar nosso framework do Motor, com pequenas mudanças (para permitir interação com o mouse) para implementar não um jogo, mas uma aplicação com uma pequena interface gráfica

- Vamos fazer dois programas: uma calculadora de quatro funções, e um editor de figuras geométricas

- As duas aplicações vão ter vários *controles*: botões de comando, caixas de texto, áreas de desenho, sliders

- Vamos ter também undo e redo (desfazer e refazer) de vários níveis

Extensão de interfaces

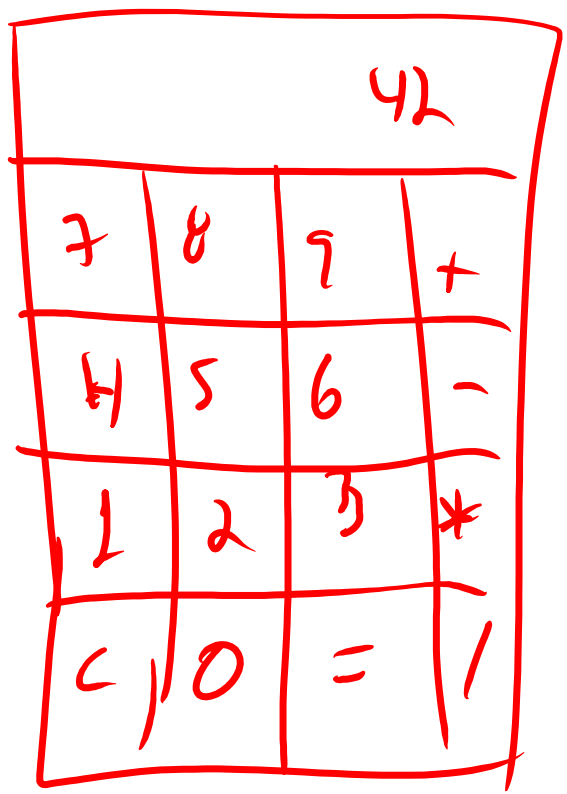
- Uma interface pode *extender* outra:

```
public interface App extends Jogo {
    void movimento(int x, int y);
    void arrasto(int x, int y);
    void clique(int x, int y);
    void aperto(int x, int y);
    void solta(int x, int y);
}
```

- Uma classe que implementa `App` precisa implementar todos os métodos de `Jogo`, mais os métodos específicos de `App`

- Uma instância de uma classe derivada de `App` pode ser tanto tratada como uma instância de `Jogo` quanto uma `App`

Calculadora – esboço da interface



Model-View-Controller (MVC)

- O MVC é o principal padrão para estruturação de aplicações com interfaces gráficas

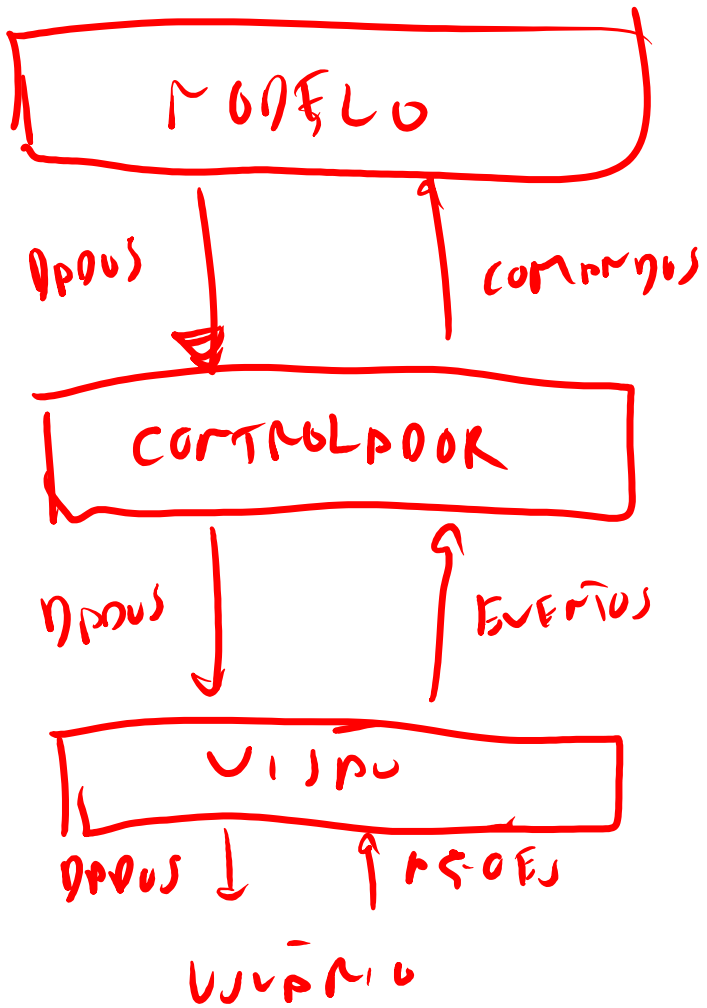
- Ele separa a aplicação em três grandes partes:

  - O *modelo* representa os dados da aplicação, e implementa a sua lógica interna de uma maneira o mais independente dos detalhes da interface quanto possível

  - A *visão* é a parte visível da interface, e é como o usuário enxerga os dados do modelo

  - O *controlador* faz a mediação entre o usuário, o modelo e a visão

MVC em um diagrama



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

Componentes

- A interface `Componente` é formada por quatro métodos `getX1()`, `getY1()`, `getX2()` e `getY2()`, que dão as *bordas* do componente

- Usamos essas coordenadas para saber se um clique do mouse está dentro do componente ou não

- Também temos um método `desenhar(Tela t)`, para o componente se desenhar

- Finalmente, temos quatro métodos, `clique`, `aperto`, `solta` e `arrasto`, que dão os eventos do mouse

- Cada um desses métodos recebe as coordenadas `x` e `y` do evento

Botões

- Um botão é nosso componente mais simples
- Quando um botão é clicado, ele deve executar uma *ação*: essa ação é instância de uma interface `Acao` bem simples, com um único método `executar` sem parâmetros
- O botão também monitora quando o mouse é apertado ou solto, para fornecer *feedback* visual ao usuário de que ele está sendo clicado
- O controlador despacha cliques do mouse para o botão apropriado

Referências a métodos

- As ações de cada botão são boas candidatas para classes anônimas, pois muitas vezes o código associado a um botão é bem específico para ele

- A versão 8 de Java introduziu formas mais convenientes de criar classes anônimas para interfaces com um único método, como `Acao`

- Uma dessas formas é a *referência a um método*: podemos usar um método com a mesma lista de parâmetros e tipo de retorno como base para essa classe:

`objeto::metodo`

```
new Acao() {
  public void executa() {
    objeto.metodo();
  }
}
```

- Também podemos fazer referências a funções (`Classe::funcao`), e até mesmo a construtores (`Classe::new`)