Computação II – Orientação a Objetos

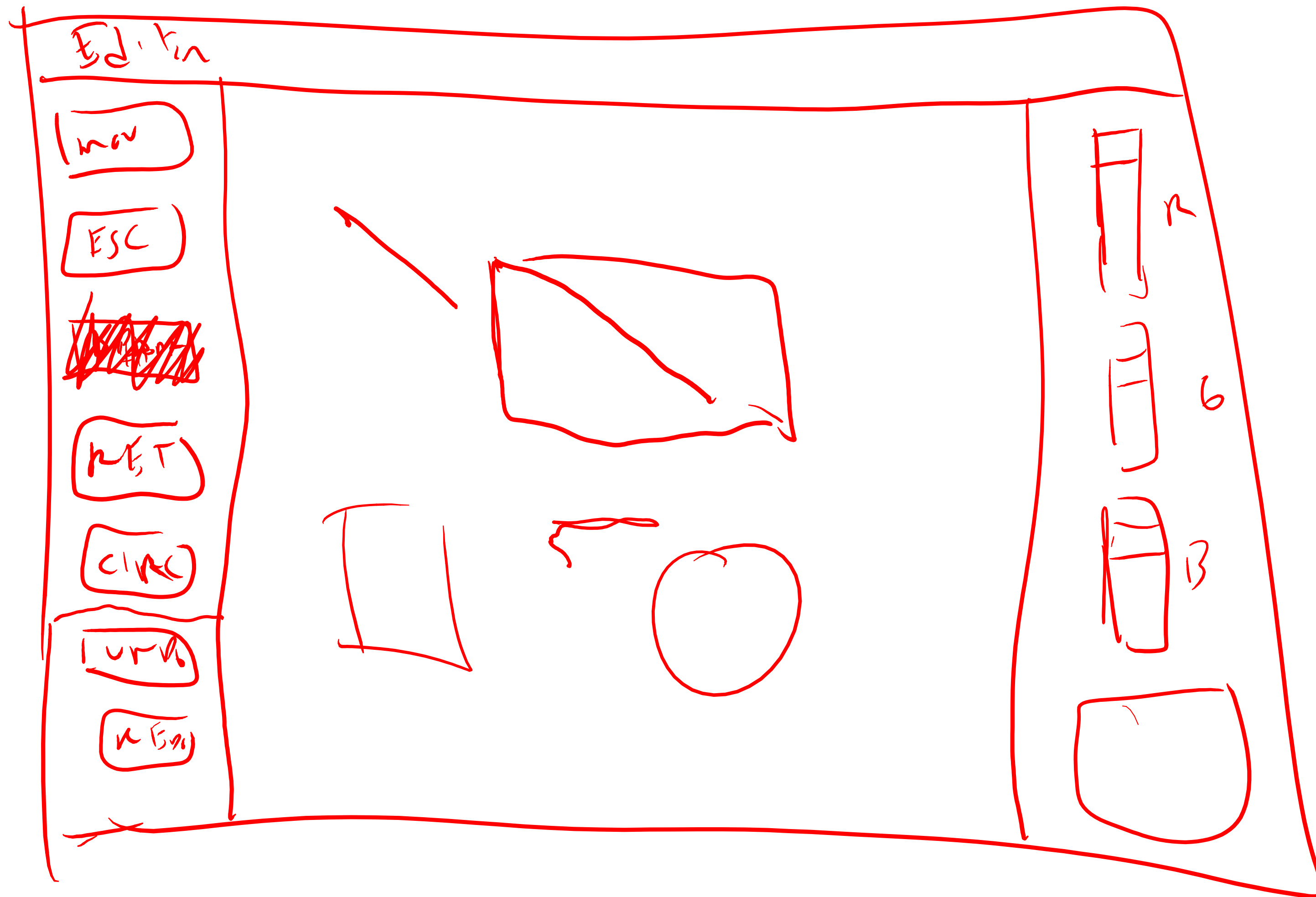
Fabio Mascarenhas - 2014.1

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

Editor Gráfico

- 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 um programa simples para desenho e manipulação de figuras geométricas: um “nano-sketchpad” (lembrem da primeira aula!)
- Nosso editor vai ter botões de comando, as figuras vão poder ser desenhadas e manipuladas usando o mouse, e vamos ter undo e redo (desfazer e refazer) de vários níveis!

Editor gráfico – 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

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 descadastras
- 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` - sujeito é o Motor

  - As interfaces `*Listener` usadas em `Motor` para despachar os eventos para a aplicação | sujeito é o canvas Swing |

  - A interface Acao para conectar um botão ao que fazer quando é clicado - sujeito é o botão

  - A interface `Toggle` para um botão se conectar a uma chave ligado/desliado - sujeito é o modelo

  - A interface `ObservadorCanvas` para receber eventos do componente canvas - sujeito é o canvas

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

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, Java oferece um segundo mecanismo para representar objetos abstratos: as *classes abstratas*

Classes abstratas - sintaxe

- Uma classe abstrata é declarada com a palavra-chave `abstract` acompanhando `class`:

```
abstract class FiguraPt {
   int x;
   int y;

   public void mover(int dx, int dy) {
      x += dx;
      y += dy;
   }

   public abstract void desenhar(Canvas c);
   public abstract void escala(double f);
   public abstract boolean dentro(int x, int y);
}
```

ponto de controle

Classes abstratas – métodos abstratos

- Uma classe abstrata pode ter campos e métodos, como qualquer outra classe, mas não podemos instanciar uma classe abstrata, não importa se ela tem construtores públicos

- Em troca, podemos declarar métodos sem corpo, do mesmo modo que em uma interface, basta declará-los como `abstract` também

- Podemos até implementar uma interface, e deixar métodos em aberto, sem declará-los! É como se os declarássemos `abstract` também.

```
abstract class FiguraPt implements Figura {
  int x, y;

  public void mover(int dx, int dy) {
    x += dx; y += dy;
  }
}
```

Usando classes abstratas - herança

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

```
class Retangulo extends FiguraPt {
  int largura;
  int altura;

  public Retangulo(int x, int y, int largura, int altura) {
    this.x = x; this.y = y; this.largura = largura; this.altura = altura;
  }

  public boolean dentro(int x, int y) {
    return (x >= this.x)&&(x <= this.x + largura)&&(y >= this.y)&&(y <= this.y + altura);
  }

  public void escala(double f) {
    altura *= f;
    largura *= f;
  }

  public void desenhar(Canvas c) {
    c.retangulo(x, y, largura, altura, 1, 1, 1);
  }
}
```

Herança

- A relação de herança (`extends`), como `implements`, também é uma relação "é-um" → é-um = subtipagem

- 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 de `Figura` (via implements), então uma instância de `Retangulo` é uma instância de `Figura`

- A diferença da herança é que nela a classe também herda a *forma* da outra, e quaisquer implementações de suas operações, não apenas as assinaturas

Herança - restrições

- Uma classe concreta precisa fornecer implementações para todos os métodos abstratos que ela herdou
    - Como uma classe abstrata pode ela própria ter herdado métodos, essa obrigação é transitiva
- Uma classe pode implementar quantas interfaces ela quiser, mas só pode herdar de uma única outra classe
- Construtores não são herados diretamente