Computação II – Orientação a Objetos

Fabio Mascarenhas - 2016.2

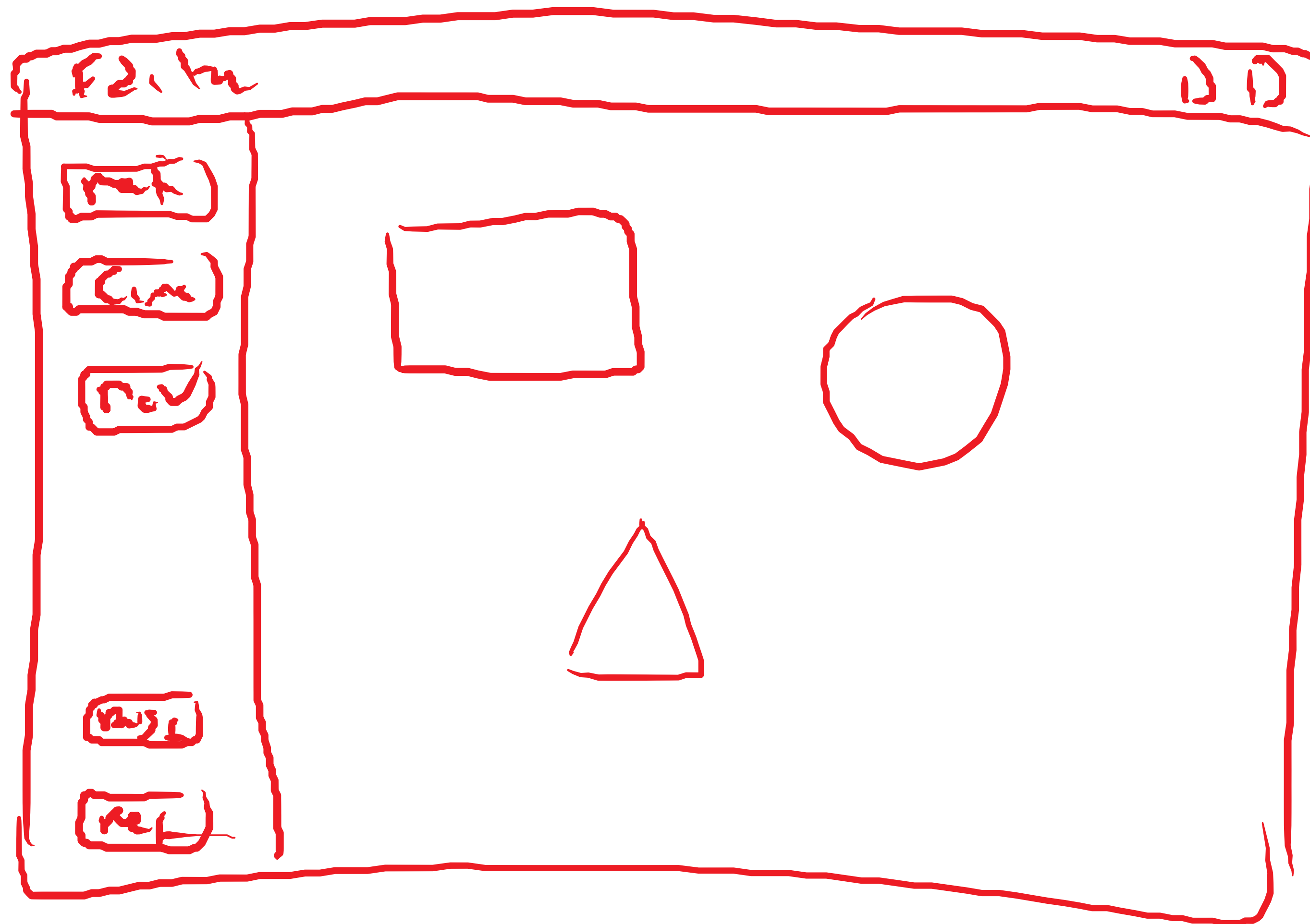
http://www.dcc.ufrj.br/~fabiom/java

Editor gráfico

- Vamos fazer um programa simples para desenho e manipulação de figuras geométricas: um “nano-sketchpad”

- 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



Canvas

- O canvas é uma área de desenho com uma borda
- Assim como botões despacham cliques para objetos ação, o canvas despacha eventos para um *observador*
- Esse observador desenha no canvas, e é avisado quando o mouse é arrastado no mesmo
- O canvas avisa o observador que ele deve desenhar nele com o método `desenhar`, e avisa sobre um arrasto do mouse com os métodos `aperto`, `arrasto` e `solta`

O modelo do Editor

- O modelo do Editor Gráfico é formado por uma classe principal, mais instâncias de classes que dão o *modo de edição*, e de instâncias de classes que representam figuras

- O modo de edição diz o que acontece quando o usuário começa uma ação, quando continua essa ação, e quando termina essa ação

- Vamos começar com três modos simples, para desenhar um retângulo, para mover uma figura na tela, e para apagar uma figura

- As figuras são representadas por uma classe abstrata, com métodos para descobrir se um ponto está dentro da figura, para mover a figura e para desenhá-la

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

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, *mesmo que este esteja sendo chamado por uma referência para a superclasse*

Herança de classes concretas e `Object`

- A superclasse que passamos para a cláusula `extends` não precisa ser uma classe abstrata
- Pode ser uma classe qualquer, assim podemos criar novas versões de uma determinada classe, mas que reaproveitam parte do seu código
- Quando não damos uma cláusula `extends`, implicitamente estamos herdando de uma classe embutida chamada `Object`
- `Object` tem um construtor sem parâmetros, por isso nunca percebemos que estávamos herdando dela!

toString, equals, hashCode

- `Object` tem alguns métodos que são úteis quando redefinidos
- O método `toString` é chamado toda vez que Java precisa converter um objeto para uma string (na concatenação com uma string, por exemplo)
- O método `equals` compara um objeto com outro, e devemos redefini-lo quando queremos comparar objetos por estrutura e não por referência
- O método `hashCode` é usado pela coleção `HashMap` como um "resumo" da estrutura de um objeto; se redefinimos `equals`, precisamos redefinir `hashCode` para dois objetos "iguais" terem o mesmo `hashCode`, ou quebramos `HashMap` com esses objetos como chaves

Redefinição com extensão

- Quando redefinimos um método em uma subclasse, podemos chamar o método que está sendo redefinido

- É como se estivéssemos estendendo o método, fazendo tudo o que ele faz, mais algum comportamento extra

- Para chamar o método que está sendo redefinido também usamos `super`, mas em uma chamada de método:

```
public String toString() {
  return "Meu objeto (" + super.toString() + ")";
}
```