



Interfaces

Mário Meireles Teixeira
mario@deinf.ufma.br



Principais conceitos

- Classes abstratas
- Interfaces
- Herança múltipla



Simulações (1)

- Programas são normalmente utilizados para simular atividades do mundo real, tais como:
 - tráfego de uma cidade;
 - previsão do tempo;
 - processos nucleares;
 - flutuações do mercado de ações;
 - mudanças ambientais.

3



Simulações (2)

- Frequentemente, são apenas simulações parciais, baseadas em um modelo do mundo real
- Em geral, elas envolvem simplificações:
 - Muitos detalhes têm o potencial de fornecer mais precisão
 - Porém, na maioria das vezes, mais detalhes exigem mais recursos:
 - Capacidade de processamento
 - Tempo de simulação

4



Benefícios das simulações

- Viabilizam previsões
 - Meteorologia, mercado de ações
- Permitem experimentação
 - Mais segura, mais barata, mais rápida, simulação de cenários incomuns
- Por exemplo:
 - 'Como a vida selvagem seria afetada se dividirmos esse parque nacional com uma estrada?'
 - 'Quanto funcionários devo contratar para reduzir pela metade o tempo de espera dos clientes em fila?'
 - 'Qual o efeito de colocar mais uma CPU no meu computador?'

5

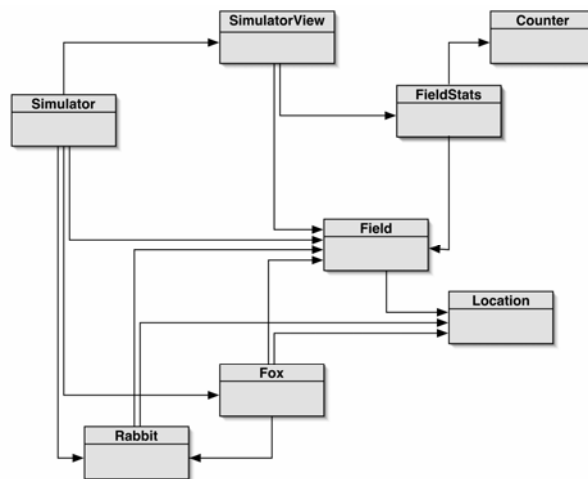


Simulações predador/presa

- Frequentemente, há um equilíbrio tênue entre as espécies:
 - Muitas presas significam muita comida
 - Muita comida estimula um número maior de predadores
 - Mais predadores comem mais presas
 - Menos presas significam menos comida
 - Menos comida significa ...

6

O projeto "raposas-e-coelhos"



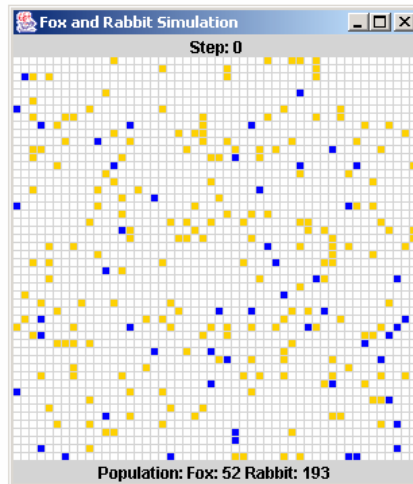
7

Principais classes

- **Fox**
 - Modelo simples de um tipo de predador
- **Rabbit**
 - Modelo simples de um tipo de presa
- **Simulator**
 - Gerencia a tarefa geral de simulação
 - Armazena uma coleção de raposas e coelhos
- **Field**
 - Representa um campo em 2D
- **Location**
 - Representa uma posição em 2D
- **SimulatorView, FieldStats, Counter**
 - Apresenta uma visualização do campo/simulação
 - Armazenam estatísticas sobre a simulação

8

Exemplo de visualização



9

Estado de um coelho

```
public class Rabbit
{
    Campos estáticos omitidos

    // Características individuais

    // A idade do coelho
    private int age;
    // Se o coelho está ou não vivo
    private boolean alive;
    // A posição do coelho
    private Location location;

    Método omitido
}
```

10



Comportamento de um coelho

- Gerenciado a partir do método **run**
 - Coelhos só sabem correr
- Idade incrementada em cada etapa da simulação
 - Um coelho poderia morrer nesse ponto
- Coelhos com idade suficiente poderiam procriar em cada fase
 - Novos coelhos poderiam nascer nesse ponto

11



Simplificações de coelho

- Coelhos não têm sexos diferentes
 - Na verdade, todos são fêmeas
- O mesmo coelho poderia procriar em cada fase
- Todos os coelhos morrem com a mesma idade
- Outros?

12

Estado de uma raposa

```
public class Fox
{
    Campos estáticos omitidos

    // A idade da raposa
    private int age;
    // Se a raposa está ou não viva
    private boolean alive;
    // A posição da raposa
    private Location location;
    // O nível de alimentos da raposa, que aumenta
    // comendo coelhos
    private int foodLevel;

    Métodos omitidos
}
```

13

Comportamento de uma raposa

- Gerenciado a partir do método **hunt**
 - Raposas só sabem caçar
- Raposas também envelhecem e procriam
- Elas têm fome
- Elas caçam em locais adjacentes a sua posição no campo

14



Configuração de uma raposa

- Simplificações semelhantes a coelhos
- Caçar e comer poderiam ser modelados de diferentes maneiras.
 - O nível de alimentos deve ser cumulativo?
 - Uma raposa fica saciada?
 - Qual é a probabilidade de uma raposa faminta caçar?
- As simplificações são aceitáveis?

15



A classe `simulator`

- Três componentes-chave:
 - Configuração da simulação no construtor
 - O método `populate`:
 - A cada animal é dada uma idade inicial aleatória
 - Os animais são espalhados aleatoriamente pelo campo
 - O método `simulateOneStep`:
 - Itera pela população de animais
 - Dois objetos `Field` são utilizados: `field` e `updatedField`

16

Simulação de um passo

```
for(Iterator iter = animals.iterator();
    iter.hasNext(); ) {
    Object animal = iter.next();
    if(animal instanceof Rabbit) {
        Rabbit rabbit = (Rabbit)animal;
        if(rabbit.isAlive())
            rabbit.run(updatedField, newAnimals);
        else
            iter.remove();
    }
    else if(animal instanceof Fox){
        Fox fox = (Fox)animal;
        if(fox.isAlive())
            fox.hunt(field, updatedField, newAnimals);
        else
            iter.remove();
    }
}
```

Retorna true se tipo dinâmico de animal for Fox ou subclasse de Fox

17

Espaço para aprimoramento

- **Fox** e **Rabbit** têm grandes semelhanças, mas não têm uma superclasse comum
- **Simulator** está fortemente acoplada a classes específicas
 - Ela 'conhece' bastante o comportamento das raposas e dos coelhos

18

A superclasse **Animal**

- Colocar campos comuns em **Animal**:
 - **age**, **alive**, **location**
- Método genérico em **Animal**:
 - **run** e **hunt** tornam-se **act**
- **Simulator** agora pode ser significativamente desacoplada:
 - Possui um método que permite que um animal atue, redefinido em cada subclasse (polimorfismo)

19

Iteração (genérica) aprimorada

```
public class Simulator
{
    public void simulateOneStep()
    {
        ...
        for(Iterator iter = animals.iterator();
            iter.hasNext(); )
        {
            Animal animal = (Animal)iter.next();
            if(animal.isAlive()) {
                animal.act(field, updatedField, newAnimals);
            }
            else {
                iter.remove();
            }
            ...
        }
    }
}
```

20

O método `act` em `Animal`

- Verificação de tipo estático pelo compilador requer um método `act` em `Animal`, pois não há nenhuma implementação óbvia compartilhada

- Define `act` como abstrato:

```
abstract public void act(Field currentField,  
                          Field updatedField,  
                          List newAnimals);
```

- O tipo dinâmico da variável determina se de fato a ação será executada para raposas ou coelhos

21

A classe `Animal`

```
public abstract class Animal  
{  
    campos omitidos  
  
    /**  
     * Faz com que esse animal atue - isto é: faz  
     * com que ele realize seja lá o que ele queira/  
     * precise realizar.  
     */  
    abstract public void act(Field currentField,  
                             Field updatedField,  
                             List newAnimals);  
  
    outros métodos omitidos  
}
```

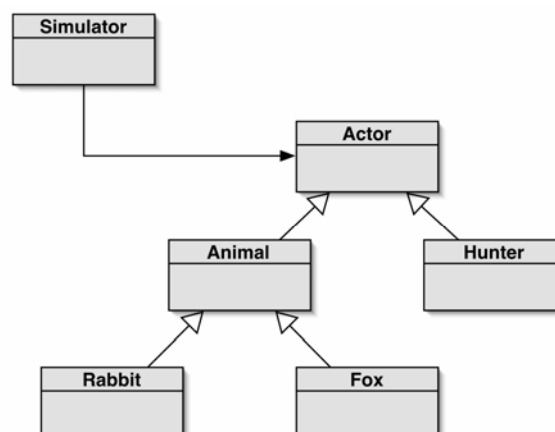
22

Classes Abstratas

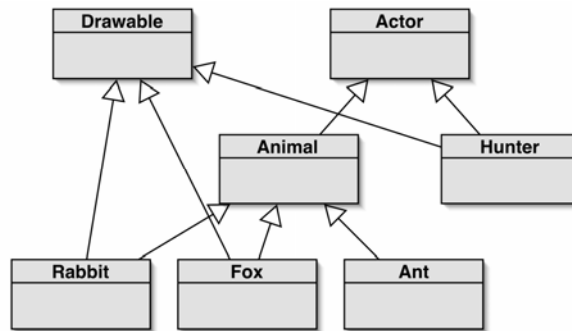
- Métodos abstratos têm **abstract** na assinatura e não têm nenhum corpo → tornam a classe abstrata
- Classes abstratas não podem ser instanciadas → não criam objetos
- Uma classe abstrata serve apenas como uma superclasse para outras classes
- Uma subclasse concreta de uma classe abstrata deve fornecer uma implementação para todos os métodos abstratos herdados. Caso contrário, ela mesma será abstrata

23

Mais abstração



24



```

for(Iterator it = actors.iterator(); it.hasNext()) {
  Actor actor = (actor) it.next();
  actor.act(...);
}
for(Iterator it = drawables.iterator(); it.hasNext()) {
  Drawable item = (drawable) it.next();
  item.draw(...);
}

```

Em vez de percorrer o campo inteiro, pode-se percorrer uma coleção separada de atores desenháveis

25

Herança múltipla

- Faz com que uma classe herde diretamente de múltiplas superclasses
- Cada linguagem tem suas próprias regras
 - Como resolver as definições de concorrência?
- O Java proíbe herança múltipla para classes
- Porém permite para interfaces, numa forma limitada

26

Uma interface actor

```
public interface Actor
{
    /**
     * Realiza o comportamento do ator.
     * Transfere o ator para updatedField se ele for
     * participar das etapas posteriores da simulação
     */
    void act(Field currentField, Location location,
             Field updatedField);
}
```

27

Classes 'implementam' interfaces

```
public class Fox extends Animal implements Drawable
{
    ...
}

public class Hunter implements Actor, Drawable
{
    ...
}
```

- Todos os métodos de uma interface são abstratos, nenhum corpo é permitido (implícito **abstract**)
- As interfaces não possuem construtores
- Todos os métodos são públicos
- Apenas declarações de campos constantes são permitidas (implícitos **public static final**)

28



Interfaces como tipos

- **Herança** tem as seguintes vantagens:
 - A subclasse herda o código da superclasse (reuso de código)
 - A subclasse é um subtipo da superclasse (polimorfismo de variáveis e métodos)
- **Interfaces** também definem um tipo, assim como uma classe:
 - Variáveis podem ser declaradas como sendo do tipo da interface (tipo estático)
 - Durante o programa, elas vão referenciar um objeto real, instanciado a partir de um dos subtipos da interface → polimorfismo

29

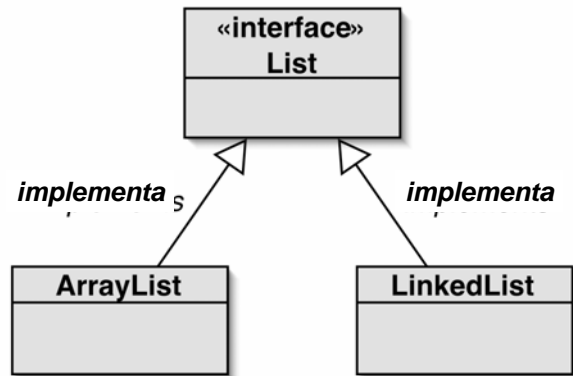


Interfaces como especificações

- Interfaces separam completamente a definição da funcionalidade da sua implementação
 - Embora parâmetros e tipos de retorno sejam obrigatórios
- Clientes interagem independentemente da implementação
 - Mas os clientes podem escolher implementações alternativas
- Um exemplo disso são as hierarquias de coleções em Java: **List**, **ArrayList** e **LinkedList**

30

Implementações alternativas



31

Visualização da simulação

- A lógica da simulação e sua visualização também são partes do sistema fracamente acopladas

```
public interface SimulatorView
{
    void setColor(Class c1, Color color);
    void isVisible(Field field);
    void showStatus(int step, Field field);
}

public class AnimatedView implements SimulatorView
{
    ...
}
```

32



Revisão (1)

- Herança pode fornecer implementação compartilhada
 - Classes concretas e abstratas
- Herança fornece informações sobre o tipo compartilhado
 - Classes e interfaces

33



Revisão (2)

- Métodos abstratos permitem a verificação do tipo estático sem exigir uma implementação
- Classes abstratas funcionam como superclasses incompletas
 - Nenhuma instância
- Classes abstratas suportam o polimorfismo

34



Interfaces

- Interfaces fornecem uma especificação sem uma implementação
 - Interfaces são totalmente abstratas
- Interfaces suportam polimorfismo
- Interfaces Java suportam herança múltipla