MAC0321 – Trabalho em Grupo – Batalha Pokémon

Marcelo Finger*
Alfredo Goldman Vel Lejbman
Thiago Kenji Okada

29 de abril de 2019

Instruções

Esse trabalho pode ser feito em duplas e deve ser feito durante o horário da aula. Vocês terão duas aulas para resolver os dois exercícios descritos, porém caso a dupla ache necessário é possível fazer parte do exercício em casa também.

A entrega **não** será feita pela Graúna. Os alunos terão que criar uma conta no GitHub[1] ou BitBucket[2] (pode ser apenas uma conta por dupla, mas preferencialmente cada membro da dupla cria uma conta e os dois têm acesso de escrita no repositório) e colocam todo o código nesse repositório. Esse repositório deve ser público, pois no final iremos baixar o código dos repositórios para correção. **Recomendação:** não deixem para *commitar* apenas quando tudo estiver pronto! Façam *commits* constantes, isso evita que vocês percam parte do trabalho. No Git, além de commitar vocês DEVEM dar um **push** nos commits para que as atualizações vão para o repositório remoto.

Introdução

Observe a forma utilizada por Bruce Eckel[3] para escrever um esqueleto de programa que controla uma *Greenhouse* (estufa para plantas).

Primeiro foi criada uma classe abstrata que serve de base para os eventos, esta base é responsável pela definição de quais métodos devem estar disponíveis para os eventos:

```
1  package c07.controller;
2
3  abstract public class Event {
4    private long evtTime;
5    public Event(long eventTime) {
6      evtTime = eventTime;
7    }
8    public boolean ready() {
9      return System.currentTimeMillis() >= evtTime;
10   }
11   abstract public void action();
12   abstract public String description();
13 } ///:~
```

*Esse mané não tem ideia do que seja um Pokémon
[1] `https://github.com/`
[2] `https://bitbucket.org/`
[3] Thinking in Java, disponível na internet no endereço: `www.bruceeckel.com`

Vemos claramente que nesta classe existem 4 métodos, sendo dois deles abstratos (`action` e `description`). No método `Event` (construtor), o instante a partir do qual o evento pode ser disparado é determinado. O método `ready` apenas retorna verdadeiro caso o objeto evento em questão está "pronto".

A partir desta classe `Event`, uma classe para controlar vários objetos do tipo Event é criada:

```
1  package c07.controller;
2
3  // This is just a way to hold Event objects.
4  class EventSet {
5    private Event[] events = new Event[100];
6    private int index = 0;
7    private int next = 0;
8    public void add(Event e) {
9      if(index >= events.length)
10       return; // (In real life, throw exception)
11     events[index++] = e;
12   }
13   public Event getNext() {
14     boolean looped = false;
15     int start = next;
16     do {
17       next = (next + 1) % events.length;
18       // See if it has looped to the beginning:
19       if(start == next) looped = true;
20       // If it loops past start, the list
21       // is empty:
22       if((next == (start + 1) % events.length)
23          && looped)
24         return null;
25     } while(events[next] == null);
26     return events[next];
27   }
28   public void removeCurrent() {
29     events[next] = null;
30   }
31 }
32
33 public class Controller {
34   private EventSet es = new EventSet();
35   public void addEvent(Event c) { es.add(c); }
36   public void run() {
37     Event e;
38     while((e = es.getNext()) != null) {
39       if(e.ready()) {
40         e.action();
41         System.out.println(e.description());
42         es.removeCurrent();
43       }
44     }
45   }
46 } ///:~
```

Na classe `EventSet` vemos que existem três membros privados: `index` que corresponde à quantidade de eventos no vetor de eventos `Event`, e `next` que corresponde ao evento em questão. Existem também métodos para adicionar um evento ao vetor, pegar o próximo evento (não nulo) do vetor e remover o evento em questão.

A classe `Controler` gerencia um elemento da classe `EventSet`. Seus métodos permitem a adição de um

elemento, e a varredura dos eventos, enquanto existirem eventos disponíveis.

Finalmente na classe `GreenhouseControls` serão criados eventos para controlar uma estufa.

```
package c07.controller;

public class GreenhouseControls extends Controller {
  private boolean light = false;
  private boolean water = false;
  private String thermostat = "Day";
  private class LightOn extends Event {
    public LightOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn on the light.
      light = true;
    }
    public String description() {
      return "Light is on";
    }
  }
  private class LightOff extends Event {
    public LightOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn off the light.
      light = false;
    }
    public String description() {
      return "Light is off";
    }
  }
  private class WaterOn extends Event {
    public WaterOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      water = true;
    }
    public String description() {
      return "Greenhouse water is on";
    }
  }
  private class WaterOff extends Event {
    public WaterOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      water = false;
    }
    public String description() {
      return "Greenhouse water is off";
```

```
    }
  }
  private class ThermostatNight extends Event {
    public ThermostatNight(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      thermostat = "Night";
    }
    public String description() {
      return "Thermostat on night setting";
    }
  }
  private class ThermostatDay extends Event {
    public ThermostatDay(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      thermostat = "Day";
    }
    public String description() {
      return "Thermostat on day setting";
    }
  }
  // An example of an action() that inserts a
  // new one of itself into the event list:
  private int rings;
  private class Bell extends Event {
    public Bell(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Ring bell every 2 seconds, rings times:
      System.out.println("Bing!");
      if(--rings > 0)
        addEvent(new Bell(
          System.currentTimeMillis() + 2000));
    }
    public String description() {
      return "Ring bell";
    }
  }
  private class Restart extends Event {
    public Restart(long eventTime) {
      super(eventTime);
    }
    public void action() {
      long tm = System.currentTimeMillis();
      // Instead of hard-wiring, you could parse
      // configuration information from a text
      // file here:
      rings = 5;
      addEvent(new ThermostatNight(tm));
      addEvent(new LightOn(tm + 1000));
      addEvent(new LightOff(tm + 2000));
```

```
      addEvent(new WaterOn(tm + 3000));
      addEvent(new WaterOff(tm + 8000));
      addEvent(new Bell(tm + 9000));
      addEvent(new ThermostatDay(tm + 10000));
      // Can even add a Restart object!
      addEvent(new Restart(tm + 20000));
    }
    public String description() {
      return "Restarting system";
    }
  }
  public static void main(String[] args) {
    GreenhouseControls gc =
      new GreenhouseControls();
    long tm = System.currentTimeMillis();
    gc.addEvent(gc.new Restart(tm));
    gc.run();
  }
} ///:~
```

Exercício 1

Vocês devem agora escrever o código que simule uma batalha do jogo *Pokémon Red/Blue/Yellow*, com as seguintes características:

- Vocês irão simular uma batalha entre dois jogadores no modo Versus[4]. Cada jogador pode ter até 6 *Pokémons*. Cada *Pokémon* tem seu nome e quantidade de HP, além de até 4 ataques diferentes.
- Cada ação do treinador será um evento; atacar (usando uma habilidade do *Pokémon* atual), trocar o *Pokémon* ativo, usar um item e fugir da batalha.
- Para simplificar a modelagem, não é necessário implementar status especiais, PPs ou modificadores de dano. As habilidades devem apenas causar dano e ter uma prioridade, que deve ser respeitada (se a prioridade de um ataque for menor que a da outra, ela deve ser executada primeiro). Itens devem ser apenas de cura, e não precisam ter limite. Entre os tipos de evento a prioridade é fugir da batalha > trocar o *Pokémon* ativo > usar um item > atacar. O critério de desempate é o primeiro jogador.
- A batalha *Pokémon* termina quando um dos treinadores não tiver mais *Pokémons* sobrando ou fugir da batalha.

Além da classe *Events*, vocês tem que desenvolver classes que modelem *Pokémons*, Treinadores e etc. Respeitem a Orientação a Objetos ;).

Crédito extra: implemente um sistema de fraqueza baseado no tipo do *Pokémon*. Para simplificar, o tipo do ataque do *Pokémon* é baseado no tipo básico do mesmo. Você pode usar o *Type chart* do site Bulbapedia[5] para isso, ou invente seu próprio.

[4] Não me perguntem o que é isso, eu não jogo joguinho!!! Recomendo dar uma olhada no joguinho para ver do que se trata, começando por aqui: `https://en.wikipedia.org/wiki/Pok%C3%A9mon_Red_and_Blue` .

[5] `http://bulbapedia.bulbagarden.net/wiki/Type`

Exercício 2

Baseado no código de batalha *Pokémon* desenvolvido na última aula, será necessário agora desenvolver o mapa do jogo. Para isso queremos as seguintes características:

- O treinador pode andar sobre um chão comum ou um gramado. Se estiver andando no gramado ele tem uma chance (a ser definida por vocês) de entrar em uma batalha com um *Pokémon* selvagem. Isso deverá ser modelado como eventos no programa de vocês.
- Uma batalha de *Pokémon* selvagem pode ser modelada como uma batalha contra um treinador com um *Pokémon* apenas.
- Contra *Pokémons* selvagens é necessário que o adversário ataque sozinho. Não é necessário desenvolver nada muito sofisticado, atacar de forma aleatória basta ;).

Crédito extra: permita que o treinador capture um *Pokémon* selvagem por meio do uso de Pokébolas. A chance de captura de um *Pokémon* selvagem deve aumentar quanto mais fraco o mesmo estiver. Você pode usar a fórmula disponível no site Bulbapedia[6] para isso, ou inventar a sua própria fórmula (porém a chance não deve ser fixa ou simplesmente aleatória).

[6] `http://bulbapedia.bulbagarden.net/wiki/Catch_rate`