Programando em Java

Gestor de agenda de campanha eleitoral

Mestrado Integrado em Engenharia Informática FCT UNL

http://ctp.di.fct.unl.pt/miei/ip/

Corpo Docente 2020/2021

António Ravara, Artur Miguel Dias, Bernardo Toninho,
Ema Vieira, Inês Fernandes, Margarida Mamede,
Miguel Monteiro, Rui Nóbrega

Gestor de agenda de campanha eleitoral (adaptado de exame 2015/16)

Pretende-se gerir as agendas de candidatos num processo eleitoral.

Cada candidato é univocamente identificado pelo seu nome e pode ter eventos em cada um dos 15 dias de campanha, num dos 3 períodos possíveis - manhã, tarde ou serão - em dada localidade.

Deve ser possível:
- juntar candidatos à campanha (sem limitar o seu número)
- juntar eventos em dado local aos candidatos, desde que o *slot* (par dia e período) não esteja ocupado
- conhecer todos os candidatos e a sua agenda eleitoral completa
- conhecer todos os eventos programados para dada localidade.

Gestor de agenda de campanha eleitoral (adaptado de exame 2015/16)

Que entidades estão envolvidas nesta realidade a modelar?

Cada candidato é *univocamente* identificado pelo seu nome e pode ter eventos

Um Evento ocorre num dos 15 dias de campanha, num dos 3 períodos possíveis - manhã, tarde ou serão - em dada localidade

Entidades:

- candidato - ?
- nome - ?
- evento - ?
- dia - ?
- período - ?
- local - ?

Gestor de agenda de campanha eleitoral (adaptado de exame 2015/16)

Que entidades estão envolvidas nesta realidade a modelar?

Cada candidato é *univocamente* identificado pelo seu nome e pode ter eventos

Um Evento ocorre num dos 15 dias de campanha, num dos 3 períodos possíveis - manhã, tarde ou serão - em dada localidade

Entidades:

- candidato - classe
  - nome - String
  - eventos - vector de classe
- evento - classe
  - dia - int entre 1 e 15
  - período - int entre 1 e 3
  - local - String

Gestor de agenda de campanha eleitoral (adaptado de exame 2015/16)

Comecemos por programar os eventos

- evento - classe
  - dia - int entre 1 e 15
  - período - int entre 1 e 3 (manhã, tarde ou serão)
  - local - String

Constantes úteis

```
public static final int NUM_DAYS = 15;
public static final int MORNING = 1;
public static final int AFTERNOON = 2;
public static final int EVENING = 3;
```

Classe Event

Constantes úteis

```
public static final int NUM_DAYS = 15;
public static final int MORNING = 1;
public static final int AFTERNOON = 2;
public static final int EVENING = 3;
```

Variáveis de instância

```
private int day; // day of the event
private int period; // day period of the event
private String place; // location of the event
```

Construtor - inicializa as variáveis de instância

```
public Event(int day, int period, String place) {...}
// @pre ???
```

Classe Event

```
public static final int NUM_DAYS = 15;
public static final int MORNING = 1;
public static final int AFTERNOON = 2;
public static final int EVENING = 3;

private int day; // day of the event
private int period; // day period of the event
private String place; // location of the event

/** @pre - deve-se restringir os valores dos argumentos para
 *  day - entre 1 e NUM_DAYS
 *  period - MORNING, AFTERNOON ou EVENING
 *  place - não deve ser null
 * /
public Event(int day, int period, String place) {...}
```

Classe Event

```
/** @pre - deve-se restringir os valores dos argumentos para
 *  day - entre 1 e NUM_DAYS
 *  period - MORNING, AFTERNOON ou EVENING
 *  place - não deve ser null
 * /
public Event(int day, int period, String place) {...}
```

Como a condição sobre o dia e o período é elaborada e será usada várias vezes (também fora da classe), porque não codificá-la num *método "de classe"* ?

```
public static boolean isValidSlot(int day, int period) {
    return 1 <= day && day <= NUM_DAYS &&
  (period == MORNING || period == AFTERNOON || period == EVENING)
}
```

Classe Event

Como a agenda deve estar ordenada por dias (de 1 a 15) e cada dia ordenado por período, precisamos de definir critérios para dado evento ser *igual* a e ser *menor* que outro.

```
public boolean equals(Event other) {
    return day == other.getDay() && period == other.getPeriod()
}
```

Dois eventos são iguais (do ponto de vista da ordenação da agenda) se ocorrem no mesmo dia e período

Classe Event

Como a agenda deve estar ordenada por dias (de 1 a 15) e cada dia ordenado por período, precisamos de definir critérios para dado evento ser *igual* a e ser *menor* que outro.

```
public boolean lessThan(Event other) {
    return day < other.getDay() ||
  (day == other.getDay() && period < other.getPeriod())
}
```

Como `true` é elemento absorvente da disjunção, se `day < other.getDay()` então toda a expressão dá `true`

Como `false` é elemento neutro da disjunção, se `!(day < other.getDay())` então toda a expressão dá o que der

```
day == other.getDay() && period < other.getPeriod()
```

Gestor de agenda de campanha eleitoral (adaptado de exame 2015/16)

Vamos agora programar os candidatos:

cada candidato é *univocamente* identificado pelo seu nome e pode ter eventos

- candidato - classe
  - nome - String
  - eventos - vector de classe

Constante útil - número máximo de eventos (dias x periodos = 15*3)

```
private static final int MAX_SIZE = 45;
```

Variáveis de instância:

```
private String name;
private int count;
private Event[] events;
```

Classe `Candidate`

Funcionalidade central:

adicionar eventos

```
public void addEvent(Event ev) {
    insertSort(ev);
}
```

Requisito que `ev` deve respeitar:

o evento decorre num *slot* (dia e período) disponível

Pré-condição de `addEvent`:

`isAvail(ev.getDay(),ev.getPeriod())`

para que `ev.getDay()` não dê erro, é necessário que `ev != null`

Classe `Candidate`

Como determinar se um *slot* está disponível?

não há nenhum outro evento no vector a decorrer nesse *slot*

```
public boolean isAvail(int day, int period) {
    return searchEvent(day,period) == -1;
}
```

Requisito que o *slot* deve respeitar:

deve ser válido - dia entre 1 e 15 e período entre 1 e 3

Pré-condição de `isAvail`:

Event.`isValidSlot(day,period)`

Gestor de agenda de campanha eleitoral (adaptado de exame 2015/16)

Vamos finalmente programar o *sistema* (classe `Campaign`):

a aplicação deve permitir **adicionar tantos candidatos quantos se desejar**, adicionar eventos aos candidatos, listar todos os candidatos, todos os eventos de dado candidato e todos os eventos em dada localidade

Adicionar candidatos - só temos que garantir que os seus nomes são únicos

```
// @pre name != null && !exists(name)
public void addCandidate(String name) {
    if (isFull()) grow();
    candidates[count++] = new Candidate(name);
}
```

assumindo que a classe tem as variáveis de instância `candidates` e `count` e os métodos `exists`, `isFull` e `grow`

Gestor de agenda de campanha eleitoral (adaptado de exame 2015/16)

Vamos finalmente programar o *sistema* (classe `Campaign`):

a aplicação deve permitir adicionar tantos candidatos quantos se desejar, **adicionar eventos aos candidatos**, ...

Adicionar **eventos aos candidatos** - temos que garantir que:

- o candidato existe
- o *slot* é válido
- o *slot* está disponível na agenda do candidato

```
// @pre name != null && exists(name) && place != null &&
    Event.isValidSlot(day,period) &&
    getCandidate(name).isAvail(day,period)
public void addEvent(String name, int day, int period, String place)
{ getCandidate(name).addEvent(new Event(day,period,place)); }
```

assumindo que o método `getCandidate` existe