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

Vimos como juntar:

- candidatos à campanha
- eventos à campanha de dado candidato

Vamos ver como listar:

- todos os candidatos
- todos os eventos que decorrem em dada localidade

Uma vez que a classe `Main` não tem acesso à memória privada da classe `Campaign`, vamos precisar de *iteradores*

> Um iterador é um objecto que percorre (com algum critério) um conjunto de outros objectos

Iteradores

Precisamos de iteradores para apresentar:

- todos os candidatos, por ordem alfabética - o iterador deve ordenar
- todos os eventos em dada localidade - o iterador deve filtrar - por ordem crescentes de dia, para cada dia por ordem crescente de período, e em cada período por ordem alfabética de candidato

A classe `Campaign` tem como variável de instância um vector de candidatos
A classe `Candidate` tem como variável de instância um vector de eventos

Para passar à classe `Main` a informação requerida, a classe `Campaign` precisa de método que devolvam iteradores de:

1. candidatos (standard; devolvido por método na classe `Campaign`)
2. eventos dum candidato (standard devolvido por método na classe `Candidate`)
3. eventos num local - filtra eventos no local de cada candidato e agrega todos

Iterador de eventos num local

Construtor tem que receber *vector de eventos*, *número de eventos* no vector e *local*

```
// @pre events != null && count >= 0 && place != null
public IteratorEventsPlace(Event[] events, int count, String place)
```

No construtor, coloca-se `current` a "apontar" para o 1º evento que decorre no local

```
this.events = events;
this.count = count;
current = 0;
this.place = place;
moveToNextEv();
```

Iterador de eventos num local

O método privado `moveToNextEv` avança para o evento em causa:

```
while (current < count && !events[current].getPlace().equals(place)) current++;
```

O método `next` faz o mesmo:

```
// @pre hasNext()
public Event next() {
    Event toReturn = events[current++];
    moveToNextEv();
    return toReturn;
}
```

Este iterador tem que ser chamado para cada candidato, de forma a se construir um vector com todos os eventos no local. Depois é só iterar sobre este vector.

Gestor de agenda de campanha eleitoral

Vamos ver agora a classe `Main`

O único método público é o `main`, que precisa:

1. dum objecto `Scanner` para ler o input do utilizador (para uma variável `String`)
2. dum objecto `Campaign` para guardar os dados fornecidos e executar comandos
3. dum ciclo para pedir comandos ao utilizador e executá-los

Método `main`

Vamos ver agora a classe `Main`

O único método público é o `main`, que precisa:

1. dum objecto `Scanner` para ler o input do utilizador (para uma variável `String`)
2. dum objecto `Campaign` para guardar os dados fornecidos e executar comandos
3. dum ciclo para pedir comandos ao utilizador e executá-los

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in); String option;
    Campaign camp = new Campaign();
    do {
        printMenu(); option = in.nextLine();
        executeOption(in,option,camp);
    } while (!option.equals(EXIT));
    in.close();
}
```

Método executeOption

Processa cada comando num método

```
private static void executeOption(Scanner in, String option,
Campaign camp) {
    switch (option) {
        case ADD_CAND: processAC(in,camp); break;
        case ADD_EVENT: processAE(in,camp); break;
        case LIST_CAND: processLC(camp); break;
        case LIST_CAND_EVENTS: processLEC(in,camp); break;
        case LIST_EVENTS_PLACE: processLEL(in,camp); break;
        case EXIT: System.out.println("Terminou o uso da aplicação");
                   break;
        default: System.out.println("Comando inexistente");
    }
}
```

Método processAC (Add Candidate)

Pede nome; se ainda não existe cria candidato e insere na colecção da Campaign

```
private static void processAC(Scanner in, Campaign camp) {
    System.out.print("Nome: ");
    String name = in.nextLine();
    if (!camp.hasCandidates() || !camp.exists(name))
       camp.addCandidate(name);
    else
       System.out.println("Nome já existe");
    }
}
```

Se não há candidatos não se perde tempo a fazer pesquisa;
se há, só insere se não existir ainda candidato com o mesmo nome

Método processAC (Add Candidate)

Como a colecção de candidatos está ordenada, tanto para *procurar se já existe o nome* como para *inserir na posição correcta* é preciso fazer pesquisa binária no vector de candidatos…

Como evitar pesquisar duas vezes?

O método `addCandidate` da classe `Campaign` devolve código a indicar se inseriu ou não (porque o nome existia).

```
private static void processAC(Scanner in, Campaign camp) {
    System.out.print("Nome: ");
    String name = in.nextLine();
    int pos = camp.addCandidate(name);
    if (pos == Campaign.POS_UNAVAILABLE)
        System.out.println("Nome já existe");
}
```

Método addCandidate da classe Campaign

Procura o nome;

- se o encontrar, devolve código de erro
  (atenção ao caso da 1ª inserção - pos == count)
- senão, devolve posição onde inserir

```
// @pre name != null
public int addCandidate(String name) {
    int pos = searchPos(name);
    if (pos < count && candidates[pos].getName().equals(name))
        return POS_UNAVAILABLE;
    else {
        if (isFull()) grow();
        insertAt(pos,new Candidate(name));
        count++;
        return pos;
    }
}
```

Método searchPos da classe Campaign

Pesquisa binária modificada: devolve posição a inserir (ou do nome, se existe)

```
// @return pos of candidate, if there is a candidate with name;
//         pos to insert, otherwise
// @pre name != null
private int searchPos(String name) {
    boolean found = false;
    int low = 0, high = count-1;
    int mid = 0;
    while(!found && low <= high) {
        mid = (low+high)/2;
        String aux = candidates[mid].getName();
        ...
    }
    if (found) return mid; else return low;
}
```

Método searchPos da classe Campaign

Pesquisa binária modificada: quando name não ocorre devolve posição que tem o menor nome maior que name e que à esquerda tem nome maior que name

```
private int searchPos(String name) {
    ...
   while(!found && low <= high) {
       int mid = (low+high)/2;
        String aux = candidates[mid].getName();
        if (aux.equals(name)) found = true;
        else if (aux.compareTo(name) > 0) high = mid-1;
             else low = mid+1;
    }
    if (found) return mid;
    else return low;
}
```

Quando o ciclo termina e low > high, low tem a posição desejada

Método processAE (Add Event)

Pede dia e período (*slot*); se é válido, pede nome do candidato e localidade

```
private static void processAE(Scanner in, Campaign camp) {
    System.out.print("Dia: ");
    int day = in.nextInt();in.nextLine();
    System.out.print("Período: ");
    int period = in.nextInt();in.nextLine();
    if (Event.isValidSlot(day,period)) {
        System.out.print("Nome: ");
        String name = in.nextLine();
        System.out.println("Localidade: ");
       ...
    }
    else System.out.println("Slot inválido");
}
```

Método processAE (Add Event)

Agora é preciso ver:

1. se há algum candidato com o nome dado - requer pesquisa na colecção de candidatos - e se sim,
2. se tem o *slot* disponível - requer pesquisa na colecção de eventos do candidato.

Mas para inserir o evento, será de novo necessário pesquisar (candidato e posição do evento)...

O melhor é fazer tudo de uma só vez: o método `addEvent` devolve código a indicar:

1. não há candidato - `NAME_UNKNOWN`
2. o *slot* está ocupado - `SLOT_BUSY`
3. a inserção do evento na agenda do candidato aconteceu - `SUCCESS`

Método processAE (Add Event)

O método addEvent devolve código (*constante da classe*) a indicar:

1. não há candidato - NAME_UNKNOWN
2. o *slot* está ocupado - SLOT_BUSY
3. a inserção do evento na agenda do candidato aconteceu - SUCCESS

```
private static void processAE(Scanner in, Campaign camp) { ...
    if (Event.isValidSlot(day,period)) { ...
        int res = camp.addEvent(day,period,name,in.nextLine());
        if (res == Campaign.NAME_UNKNOWN)
            System.out.println("Nome não existe");
        else
            if (res == Campaign.SLOT_BUSY)
                System.out.println("Slot ocupado");
    }
    else System.out.println("Slot inválido");
}
```

Localidade

Método addEvent da classe Campaign

```
/* @return NAME_UNKNOWN, if !exists(name);
 *         SLOT_BUSY, if exists(name) &&
 *                    !getCandidate(name).isAvail(day,period);
 *         SUCCESS, otherwise
 * @pre name != null && place != null &&
 * Event.isValidSlot(day,period)
 */
public int addEvent(int day, int period, String name, String place){
    Candidate cand = getCandidate(name);
    if (cand == null)
       return NAME_UNKNOWN;
    else
       return cand.addEvent(new Event(day,period,name,place));
}
```

Método getCandidate da classe Candidate

A ideia repete-se:

- com a pesquisa binária obtém-se a posição do candidato com o nome dado (se existir; se não existir, devolve-se `null`)
- devolve-se depois o candidato nessa posição

```
// @return null if !exists(name);
//         candidates[searchPos(name)] otherwise
// @pre name != null
public Candidate getCandidate(String name) {
    int pos = searchPos(name);
    if (pos < count && !candidates[pos].getName().equals(name))
        return null;
    else
        return candidates[pos];
 }
```

Método addEvent da classe Candidate

Como no caso do método addCandidate, se se pode inserir (neste caso, o *slot* está disponível), coloca-se na posição "correcta" (que mantém a ordenação)

```
// insert sort ordered by day and by period
// @pre ev != null
public int addEvent(Event ev) {
   int pos = searchEvent(ev);
   if (pos != SLOT_BUSY) {
       insertAt(pos,ev);
       count++;
   }
   return pos;
}
```