IUT de Paris Descartes – Base de la Programmation Objet

Travaux Dirigés – Sujet n°5

Objectifs

- Choisir une structure de donnée adaptée à un problème
- Être sensibilisé au problème de la redondance de données

Une animation

On veut réaliser une animation simple dans laquelle des éléments (représentés par des caractères) se déplacent de case en case sur un terrain de taille finie.

Les éléments ont une position et une direction initiale. A chaque pas de l'animation, ils changent de position en fonction de la direction. Ce déplacement n'est réalisé que s'il ne provoque pas de sortie du terrain. Dans le cas contraire, la direction est changée de manière à simuler un rebond sur un mur. Les collisions entre les éléments ne sont pas prises en compte et en conséquence plusieurs éléments peuvent être simultanément présents à un même endroit.

L'objectif de la séance est de concevoir la structure de l'application. Plusieurs solutions sont envisageables. On part sur la base de deux classes : une pour les éléments et une pour le terrain. Une ébauche de ces deux classes est donnée en annexe.

1. La classe `Terrain` doit stocker les éléments qui y sont présents. Proposez une structure de données.

Solution: Commençons par une explication des squelettes de classes du sujet. Un élément a connaissance de son identificateur (un caractère) et de sa direction (représentée par 2 entiers donnant le déplacement relatif). Des méthodes très élémentaires sont prévues.

Il semble naturel que le terrain ait la connaissance de sa taille (définie par 2 entiers - `hauteur` et `largeur` ; les coordonnées valides sont comprises entre $(1, 1)$ et $(\texttt{largeur}, \texttt{hauteur})$). Des méthodes permettant la gestion du terrain ont été prototypées. Les deux méthodes principales sont `simule` et `toString`.

Première structuration envisagée (c'est généralement celle que proposent les étudiants) :

Admettons que le terrain connaisse le contenu de chaque position possible. Une position peut contenir un ensemble d'éléments potentiellement vide. Admettons que nous disposions d'une classe `Ensemble` disposant des méthodes ad hoc. Le terrain peut être représenté par un tableau à deux dimensions d'`Ensemble`. Il est important de noter que cette solution implique que les éléments ne stockent pas leurs propres coordonnées car les indices du tableau donnent la position des éléments.

Est-ce que cette répartition des données est satisfaisante ? Pour cela, étudions les différents traitements que nous devons mettre en place dans la classe `Terrain` et plus précisément les méthodes `toString` et `simule`.

Constituer la chaîne de caractère dans la méthode `toString` est aisé, il suffit de parcourir le tableau et si une position est vide (une méthode de la classe `Ensemble`) on ajoute un espace et dans le cas contraire on ajoute l'identifiant d'un des éléments s'y trouvant (la classe `Ensemble` doit offrir une méthode rendant un élément présent).

La simulation consiste à réaliser itérativement un déplacement pour chacun des éléments présents sur le terrain. Il s'agit donc de parcourir le tableau et les ensembles qui y sont contenu pour faire des déplacements élémentaires. Le déplacement d'un élément implique des données du terrain (les coordonnées de l'élément y sont stockées ainsi que la dimension du terrain) et des données de l'élément (sa direction). De plus le déplacement va entraîner une modification de ces données. Cela implique de prévoir une ou plusieurs méthodes supplémentaires dans la classe `Elément`.

Cette solution est complexe car elle impose l'introduction d'une classe `Ensemble`.

Deuxième structuration envisagée :

Admettons que le terrain stocke uniquement les éléments participant à l'animation (une liste d'éléments) et qu'un élément ait la gestion de sa position courante (deux entiers).

La constitution de la chaîne retournée par la méthode `toString` de la classe `Terrain` nécessite de déterminer pour chaque coordonnée possible si au moins un élément occupe cette position et, si c'est le cas, d'afficher l'identifiant d'un de ces éléments. Dans le cas contraire un espace sera affiché. Cela implique que la classe `Elément` propose une méthode renseignant sur sa position courante (un accesseur). L'algorithme peut consister à initialiser un tableau à 2 dimensions de caractère pré-rempli avec des espaces et de pseudo-afficher chaque élément dans ce tableau de `char` par accès direct puis construire la chaîne à retourner à partir de ce tableau. Une seconde solution, celle proposée dans la correction de l'exercice 4, est plus simple à mettre en place même si elle n'est pas la plus efficace.

Le déplacement d'un élément nécessite de connaître sa position, sa direction et la dimension du terrain. Là encore, les données sont stockées dans l'élément mais aussi dans le terrain. Lorsque le terrain provoque le déplacement d'un élément (dans la méthode `simule`) la dimension du terrain peut être passée en paramètre à la méthode de la classe `Elément` réalisant le déplacement proprement dit. Ainsi la classe `Elément` dispose de toutes les informations nécessaires et les données mises à jour sont cantonnées dans la classe `Elément`.

C'est la solution que nous retiendrons.

2. Complétez la définition des classes en précisant les données et les prototypes des méthodes en fonction des choix réalisés ci-dessus. Faites en sorte qu'aucune donnée ne soit redondante. Notez bien que les prototypes du constructeur de la classe `Elément` et la méthode `Terrain.ajoute` peuvent être complétés si nécessaire.

Solution:

```
public class Elément {
  private char id;
  private int dx, dy;
  private int x, y;   // coordonnée de l'élément

  public Elément(char id, int x, int y, int dx, int dy) {
    ...
  }

  public void bouge(int largeur, int hauteur) {
    ...
  }

  public boolean occupe(int x, int y) {
    ...
  }

  public char getId() {
    ...
  }
```

```
}

import java.util.LinkedList;

public class Terrain {
  private LinkedList<Elément> éléments;    // les éléments présents
  private int largeur, hauteur;

  public Terrain(int largeur, int hauteur) {
    ...
  }

  public void ajoute(Elément e) {
    ...
  }

  public void simule(int durée) {
    ...
  }

  public String toString() {
    ...
  }
}
```

La position initiale d'un élément est maintenant transmise en paramètre au constructeur de la classe `Elément` et la méthode de déplacement est paramétrée par la dimension du terrain.

3. Écrivez un court programme réalisant une animation avec deux éléments.

Solution: La méthode statique `pause` limite la vitesse de l'animation. Elle peut être ommise de la solution présentée en séance.

```
public class Appli {
  public static void pause(long millis) {
    try {
      Thread.sleep(millis);
    } catch (InterruptedException e) {
      assert(false);
    }
  }

  public static void main(String[] args) {
    final int DUREE = 100;
    final long PAUSE = 100;
    Terrain t = new Terrain(10, 10);
    t.ajoute(new Elément('a', 2, 2, -1, 1));
    t.ajoute(new Elément('b', 5, 7, 1, -1));
    for (int i = 0; i < DUREE; ++i) {
      t.simule(1);
      System.out.println(t.toString());
      pause(PAUSE);
    }
  }
```

```
}
```

4. Implémentez les méthodes de la classe `Terrain`.

Solution: Cette question n'est pas essentielle pour le TD.

```
public class Terrain {
  private LinkedList<Elément> éléments;
  private int largeur, hauteur;

  public Terrain(int largeur, int hauteur) {
    assert(hauteur > 0 && largeur > 0);
    éléments = new LinkedList<Elément>();
    this.largeur = largeur;
    this.hauteur = hauteur;
  }

  public void ajoute(Elément e) {
    éléments.add(e);
  }

  public void simule(int durée) {
    for (; durée > 0; --durée)
      for (Elément e : éléments)
        e.bouge(largeur, hauteur);
  }

  private Elément premierSur(int x, int y) {
    for (Elément e : éléments)
      if (e.occupe(x, y))
        return e;
    return null;
  }

  public String toString() {
    StringBuilder sb = new StringBuilder();
    for (int x = 1; x <= largeur + 2; ++x)
      sb.append('-');
    sb.append(System.lineSeparator());

    for (int y = 1; y <= hauteur; ++y) {
      sb.append('|');
      for (int x = 1; x <= largeur; ++x) {
        Elément e = premierSur(x, y);
        if (e == null)
          sb.append('␣');
        else
          sb.append(e.getId());
      }
      sb.append('|' + System.lineSeparator());
    }
    for (int x = 1; x <= largeur + 2; ++x)
      sb.append('-');
```

```
    sb.append(System.lineSeparator());
    return sb.toString();
  }
}
```

Annexe

```
public class Elément {
  private char id;     // identificateur de l'élément
  private int dx, dy; // déplacement relatif
  ...                 // à compléter

  public Elément(char id, int dx, int dy, ...) { // à compléter
    this.id = id;
    this.dx = dx;
    this.dy = dy;
    ...               // à compléter
  }

  public char getId() {
    return id;
  }
  ... // à compléter
}
```

```
public class Terrain {
  private int largeur, hauteur;
  // à compléter

  // construit un terrain vide
  public Terrain(int largeur, int hauteur) {
    assert(hauteur > 0 && largeur > 0);
    this.largeur = largeur;
    this.hauteur = hauteur;
    ... // à compléter
  }

  // ajoute un élément au terrain
  public void ajoute(Elément e, ...) { // à compléter
    ... // à compléter
  }

  // simule d déplacements des éléments du terrain
  public void simule(int d) {
    ... // à compléter
  }

  // retourne une représentation textuelle du terrain
  public String toString() {
    ... // à compléter
  }
}
```