Université Paris Diderot **Programmation Orientée Objet**
L2, EIDD et M1 linguistiques **Année 2015-2016**

TD n° 11

Classes internes et pattern MVC

1 Listes chaînées & classes internes

Le but de cette section va être de construire une structure de *liste chaînée*. Une liste chaînée est, comme son nom l'indique, constituée de maillons. Chaque maillon contient une valeur et un lien vers le maillon suivant, et le lien du dernier maillon est vide.

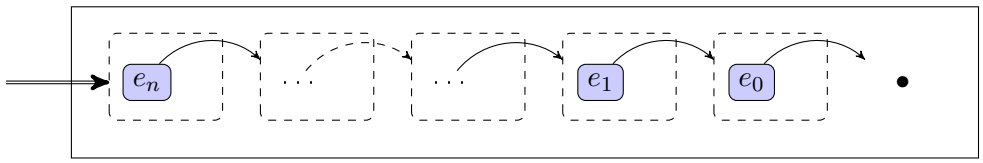

FIGURE 1 – Une liste chaînée et ses maillons

Cette structure permet facilement l'ajout d'un élément, la suppression de l'élément de tête (`pop`) et le parcours de la liste. En revanche, pour accéder au $n+1^{\text{ème}}$ élément, il faut nécessairement passer par les n premiers.

Le but de cette section sera donc de créer une classe `ListeChainee<E>` qui implémentera l'interface suivante :

```
1 public interface Liste<E> extends Iterable<E>{
2     public E head();
3     public void add(E e);
4     public void pop();
5     public void remove(E e);
6     publid int size();
7     public boolean contains(E e);
8     public boolean isEmpty();
9 }
```

en utilisant la structure de liste chaînée. Pour ce faire, on créera une classe interne que l'on appellera `Maillon`. Comme on ne souhaite pas que l'utilisateur ait accès à l'implémentation pratique, cette classe sera définie comme privée. De même, on implémentera l'interface `Iterable<E>` à l'aide d'une autre classe interne privée `Parcours`.

Exercice 1 Quels attributs/méthodes contiendra la classe `Maillon` ? Donnez la modélisation UML des classes.

Exercice 2 Écrire le code de `ListeChainee` et `Maillon` nécessaire à la définition de la méthode `void add(E e)`.

Exercice 3 Écrire les méthode `void pop()` et `isEmpty()`.

Exercice 4 Écrire la méthode `boolean contains(E e)`.

Exercice 5 Écrire la méthode `void remove(E e)`. Lorsque l'élément `e` n'appartient pas à la liste, celle-ci lèvera une exception `NotInListException`, que l'on aura définie comme héritière de `RuntimeException`.

Exercice 6 Redéfinir la méthode `toString()` afin qu'elle affiche la liste sous la forme : `"5-4-3-2-1"`.

Exercice 7 Afin d'implémenter l'interface `Iterable<E>`, créer une classe interne `Parcours` (qui implémentera elle-même l'interface `Iterator<E>`).

Exercice 8 Écrire un programme pour tester les différentes méthodes.

Exercice 9 [Bonus] Serait-il compliqué, avec notre structure telle qu'elle est définie, d'écrire la méthode `remove()` de l'itérateur ? Le cas échéant, quelles modifications pourriez-vous apporter pour pouvoir le faire ?

2 Pattern Modèle-Vue-Contrôleur

Utilisé pour la conception d'interface graphique, le pattern *Modèle-Vue-Contrôleur* a pour but de séparer trois entités :

- le modèle (les données),
- la vue utilisateur, qui présente les données du modèle à l'utilisateur,
- le contrôleur, qui, en fonction des événements qu'il reçoit, modifie les données du modèle.

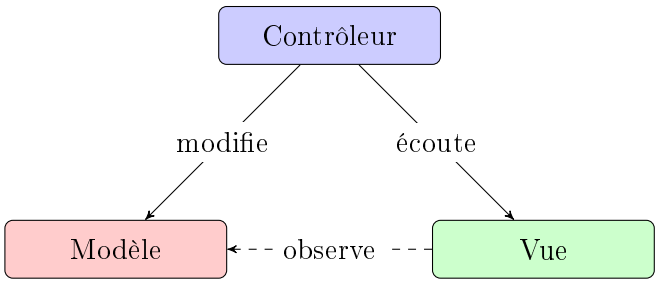

FIGURE 2 – Notre pattern MVC simplifié

Dans cet exercice, nous allons essayer de le mettre en œuvre dans un cas très simple (cf. Figure 2). On veut construire une interface graphique contenant :

- un panneau avec le dessin éventuel d'un disque,
- deux boutons, *Tracer* et *Effacer*, qui permettent de tracer/effacer le disque.

En pratique, notre classe `Modele` étendra la classe `Observable`, et sera observée par la classe `Vue` (qui implémentera donc `Observer`).

Exercice 10 Que contiendra le modèle ? Écrire une classe `Modele` avec les attributs et méthodes appropriés. On pensera bien à avertir les observateurs des changements avec les méthodes `setChanged()` et `notifyObservers()`.

On suppose avoir à disposition la classe `Ardoise` permettant le dessin d'un disque :

```
1  public class Ardoise extends JPanel {
2    private boolean possedeDisque = true;
3    public void setPossedeDisque(boolean possedeDisque) {
4      this.possedeDisque = possedeDisque;
5    }
6
7    public void dessiner(Graphics g) {
8      g.setColor(Color.RED);
9      g.fillOval(60, 35, 80, 80);
10   }
11
12   public void paintComponent(Graphics g) {
13     super.paintComponent(g);
14     if (possedeDisque) dessinerDisque(g);
15   }
16 }
```

Exercice 11 Compléter le code suivant pour ajouter les boutons *Tracer* et *Effacer* à la vue. On pourra se servir de la classe `JButton` et de la méthode

```
1  public class Vue extends JFrame implements Observer {
2      Ardoise ardoise = new Ardoise();
3      // À compléter
4
5       public Vue() {
6          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7          setLocation(200,200);
8          setTitle("Un disque");
9
10         add(ardoise, BorderLayout.CENTER);
11         // À compléter
12
13         pack();
14         setVisible(true);
15     }
16 }
```

Exercice 12 Afin d'implémenter l'interface `Observer`, définir une méthode `void update(Observable o, Object arg)`, qui sera utilisée pour mettre à jour l'ardoise. Pour mettre à jour l'affichage d'un objet de type `Component`, on utilise la méthode `repaint()`, qui fait elle-même appel à `paintComponent`. On en profitera aussi pour remplacer le titre de la fenêtre par `"Rien"` lorsqu'il n'y a plus de disque.

On va maintenant créer le contrôleur, qui devra implémenter l'interface `ActionListener` contenant une seule méthode :

```
1  void actionPerformed(ActionEvent e);
```

Le contrôleur va donc écouter les événements en provenance de la vue (dans notre cas des deux boutons), et mettre à jour au besoin le modèle. C'est aussi lui qui mettra en relation le modèle et la vue (en ajoutant la vue aux observateurs du modèle.

Exercice 13 En utilisant les méthodes `addObserver(Observer o)` et `addActionListener(ActionListener l)`, donner le code du constructeur de la classe `Controleur`, qui prendra en paramètres une vue et un modèle.

Exercice 14 En s'aidant de la méthode `Object getSource()` de `ActionEvent` (qui permet de déterminer la source d'un événement), écrire le code de la méthode `actionPerformed(ActionEvent e)` qui mettra à jour le modèle lors de la réception d'un événement. Y a-t-il besoin de changer la vue ?

Exercice 15 [ESSENTIEL] Cet exercice ayant pour principal intérêt d'être concrètement mis en œuvre, une fois n'est pas coutume, **vous prendrez donc le temps chez vous** (oui oui, vous avez bien lu), de tester ce que vous venez de définir. L'exécution de la classe suivante :

```
1 | public class MVC {
2 |   public static void main(String[] arg) {
3 |       Controleur controleur = new Controleur(new Modele(), new Vue());
4 |
5 |   }
6 | }
```

doit normalement créer la fenêtre voulue. Vous trouverez sur DidEL un fichier `MVC.java` où vous n'aurez plus qu'à compléter avec les réponses des questions précédentes.