Université Paris Diderot **Programmation Orientée Objet**
L2, **Année 2022-2023**

TP n° 7

Classes abstraites et interfaces : un gestionnaire de fichiers

Un système de fichiers

On se propose dans ce TP d'implémenter des commandes texte pour manipuler une arborescence (théorique) de fichiers et de dossiers.

On utilisera / comme séparateur de nom pour repérer les dossiers. L'exécution du programme dans un terminal donnera quelque chose comme suit :

```
$ ls
. (dossier)
$ mkdir test
$ ls
. (dossier)
test (dossier)
$ ls test
. (dossier)
.. (dossier)
$ ed test/a
Entrez le texte du fichier (terminez par une ligne contenant seulement un point)
Bonjour !
.
$ cp test test2
$ ed test2/a
Entrez le texte du fichier (terminez par une ligne contenant seulement un point)
Bonsoir !
.
$ cd test
$ cat a
Bonjour !
$ cat ../test2/a
Bonsoir !
```

Les fichiers et les dossiers hériteront d'une classe abstraite `Element` définie comme suit :

```
1 abstract class Element {
2     public abstract String getType();
3
4     public String toString() {
5         return "fichier de type " + getType();
```

```
6       }
7   }
```

Fichiers texte

1. Écrire une classe `FichierTexte` dérivant de `Element`. Un fichier texte possède un attribut `contenu` de type `String`. Le type de `FichierTexte` est "texte". Remarquez que le nom du fichier n'apparaît pas ici, nous en parlerons un peu plus tard.
2. Écrire une interface `Affichable` destinée à décrire les éléments qui peuvent être affichés : ils possèderont une méthode `public void afficher()` montrant leur contenu à l'écran.
3. Implémenter l'interface `Affichable` pour `FichierTexte`.
4. Écrire une interface `Editable` qui permettra de caractériser les éléments qui peuvent être édités : ils posséderont une méthode `public void editer(Scanner sc, boolean echo)` qui permettra en particulier à l'utilisateur de fixer l'attribut "contenu" du fichier texte.

   Le scanner est transmis en argument ainsi on pourra utiliser la même méthode pour lire des données au clavier ou bien à partir d'un autre flux (fichier réel).

   Faites que `FichierTexte` implémente `Editable` :

   Vous utiliserez le `Scanner` pour lire ligne par ligne son entrée : si une ligne ne contient seulement qu'un point `"."`, cela marquera la fin de l'édition. Ce point final n'appartient pas au contenu du fichier.

   La valeur booléenne `echo` indique si la méthode va afficher sur la console les données utilisées : si `echo` est `true`, à chaque traitement de ligne un affichage de cette même ligne est produit (avec `System.out.println`).

   Ceci est utile si le scanner donné en paramètre ne lit pas depuis le clavier mais depuis une autre source – dans ce cas on peut vouloir tout de même voir ce qui a été lu.

Entrées et Dossiers

Du point de vue de la conception on considère qu'un `Element` (et en particulier un fichier ou un dossier) est anonyme. Son nom peut être vu comme une simple décoration qui lui est associée. On réalise toutes les associations qu'on désire avoir en définissant une classe qui encapsule l'élément : on y ajoute les propriétés qui nous intéressent en déclarant un attribut pour chacune d'entre elle.

Concrètement, voilà comment les `Entree` décorent les `Element` :

```
1  class Entree {
2      private Element element;
3      private String nom;
4      private Dossier parent;
5      public Entree(Dossier p, String n, Element e) { ... }
6      ...
7  }
```

On y reconnaît :

— l'encapsulation d'un élément,

— le nom qui lui est associé ,
— et on ajoute une décoration avec la mention du dossier parent.

La classe `Dossier` est définie en parallèle d'`Entree` : il s'agit d'une extension de la classe `Element` qui contient une collection d'`Entree`s. (Remarquez la référence croisée qu'il faudra arriver à maintenir : un dossier stocke des entrées, et chaque entrée sait quel est le dossier qui la contient)

Nous allons implémenter ces deux classes progressivement :

1. Écrire la classe `Dossier` qui contient une `List<Entree>` et un champ référençant son dossier parent (sous forme d'une `Entree`). Écrire un constructeur de dossier vide, il ne prend en argument que le dossier déclaré parent.
2. Créer une méthode `toString` dans `Entree` qui retourne une chaîne de la forme `"nom (type)"` : nom est celui de l'entrée et le type est celui de l'élément encapsulé :
   — `"texte"` pour un fichier texte.
   — `"dossier"` pour un dossier,
   — `"entrée vide"` si l'entrée n'a pas été encore été affectée,
   Ajoutez des méthodes `getNom()` et `getElement()`.
3. Créer, dans `Entree`, une méthode `public void supprimer()`. Pour cela il faudra aller retirer cette entrée du dossier parent qui la contient. Vous écrirez une méthode intermédiaire destinée à déléguer une partie du travail au dossier, et mettrez à jour les attributs.
4. Ecrivez une méthode `public void remplacer(Element e)` qui remplace l'élément encapsulée par un nouvel élément. Remarquez que si les éléments concernés sont des dossiers les champs parents doivent être mis à jour. (Vous pouvez ici utiliser `instanceof`)
5. Ecrivez une méthode `ajouter(Element e)` : la méthode consiste d'abord à créer une entrée dont l'élément encapsulé est `null`, puis d'utiliser `remplacer`.
6. Implémenter l'interface `Affichable` pour les dossiers : on affichera la liste des entrées du dossier (`.` et `..` incluses).
7. Ecrivez, dans la classe `Dossier`, une méthode permettant de chercher, par son nom, une entrée dans un dossier : `public Entree getEntree(String nom)` . Vous regarderez seulement dans la liste stockée (pas plus profondément). Que retournez vous s'il n'y a pas d'entrée ayant ce nom ?
8. Ajoutez un paramètre `boolean creer` à la méthode `public Entree getEntree(String nom)` pour qu'elle crée éventuellement l'entrée dans le cas où elle n'existe pas encore. Cette méthode peut elle être publique ?
9. En réalité, par défaut un dossier n'est jamais vide, il possède toujours deux entrées que vous connaissez : `.` qui pointe sur lui-même, et `..` qui pointe sur le dossier parent (s'il en a un).
   Les entrées `.` et `..` ne peuvent pas répondre aux méthodes publiques `supprimer` et `remplacer` de la même façon que les autres `Entree`. C'est pourquoi on défini une classe `EntreeSpeciale` héritant de `Entree` affichant un message d'erreur plutôt que d'effectuer ces deux opérations.

Préparation à l'écriture de quelques commandes

Ayant développé ces classes modélisant les fichiers et répertoires, on peut maintenant vouloir les manipuler.

Définissez une classe `Shell` qui aura comme attribut un dossier racine représentant l'arborescence de fichier. Vous lui ajouterez un attribut technique qui mémorisera le dossier courant. A la création un Shell se verra proposer un dossier, il vous faudra remonter à sa racine pour définir le shell.

Ecrivez les méthodes qui permettront de réaliser les commandes suivantes.

```
cat <name>
cd [<foldername>]
ls [<name>]
mkdir <foldername>
mv <src> <dst>
rm <name>
ed <filename>
cp <src> <dst>
```

Bonus

Implémentez un parseur qui permette d'entrer ces commandes en intéragissant avec le clavier.