Université Paris Diderot **Programmation Orientée Objet**
L2, EIDD et M1 linguistiques **Année 2020-2021**

TP n° 7

Classes abstraites et interfaces : un gestionnaire de fichiers

Un système de fichiers

On se propose dans ce TP d'implémenter un mini shell en Java qui permettra de manipuler une arborescence de fichiers et de dossiers. Pour ne pas risquer de supprimer vos fichiers réels, cette arborescence de fichiers et dossiers existera seulement dans la mémoire vive du programme Java ; on ne va pas la lire de ou écrire sur votre disque local. On utilisera / comme séparateur de nom de 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   }
```

Exercice 1 Fichiers texte

1. Écrire une classe `FichierTexte` dérivant de `Element`. Un fichier texte possède un attribut `contenu` de type `String`.
2. Écrire une interface `Affichable` pour les éléments qui peuvent être affichés : ils possèderont une méthode `public void afficher()` affichant leur contenu à l'écran.
3. Implémenter l'interface `Affichable` pour `FichierTexte`.
4. Écrire une interface `Editable` pour les éléments qui peuvent être édités avec un éditeur de texte : ils possèderont une méthode `public void editer(Scanner sc, boolean echo)` qui permet à l'utilisateur de donner le contenu du fichier texte.

   Implementez `Editable` pour `FichierTexte`. Utilisez le `Scanner` donné en paramètre pour lire ligne par ligne l'entrée de l'utilisateur.

   Si une ligne contient seulement un point `"."`, on intérpretera cela comme fin de l'édition. Ce point n'appartient plus au contenu du fichier.

   La valeur booléenne `echo` indique si la méthode doit afficher le contenu entré par l'utilisateur. Si `echo` est `true`, après chaque ligne lu, le programme doit afficher cette même ligne avec `System.out.println`.

   Ceci est utile si le scanner donné en paramètre ne lit pas depuis la console mais depuis un fichier – dans ce cas on veut quand même voir qu'est-ce qui était lu.

Exercice 2 Lire des dossiers

On veut maintenant pouvoir manipuler des dossiers : les créer et les parcourir, en introduisant une nouvelle classe. On fait d'abord une distinction entre les `Element`s (les fichiers et les dossiers qui sont ici sans nom) et les `Entrée` qui sont des objets qui servent à décorer des éléments avec des informations utiles.

Une `Entrée` se définit par :
- un élément de type `Element`,
- le nom donné à l'élément,
- et le dossier parent qui contient l'entrée.

```
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  }
```

Et on définira un `Dossier` comme une extension de la classe `Element` qui contient une collection d'`Entree`s. Remarquez la référence croisée : un dossier contient des entrées, et chaque entrée sait quel est le dossier qui la contient.

1. Écrire la classe `Dossier` qui contient une `LinkedList<Entree>` et un champ référençant le dossier parent. Écrire un constructeur qui prend le dossier parent et initialise un dossier vide.

2. Créer une méthode `toString` dans `Entree` qui renvoie le nom de l'entrée et le type de l'élément correspondant, dans le format suivant : `"nom (type)"`, où `"type"` est
   — `"entrée vide"` pour une entrée qui n'a pas d'élément,
   — `"dossier"` pour un dossier,
   — `"texte"` pour un fichier texte.
   Implémenter des méthodes `getNom()` et `getElement()`.
3. Créer, dans `Entree`, une méthode `public void supprimer()` qui supprime l'entrée du dossier qui la contient, ainsi qu'une méthode `public void remplacer(Element e)` qui remplace l'élément contenu dans une entrée par un nouvel élément. On va faire en sorte que quand on insère un dossier, son champ `parent` soit actualisé. Pour cela, on pourra utiliser `instanceof`.
4. Créer, dans la classe `Dossier`, une méthode permettant de chercher une entrée dans un dossier : `public Entree getEntree(String nom)` renvoie l'entrée de nom `nom` du dossier, ou `null` sinon.
5. Par défaut un dossier possède toujours des entrées spéciales : `.` qui pointe sur lui-même, et `..` qui pointe sur le dossier parent (s'il en a un).
   Les entrées `.` et `..` ne peuvent pas être modifiées : définir une classe `EntreeSpeciale` héritant de `Entree` affichant un message d'erreur plutôt que d'effectuer les opérations de modification.
6. Implémenter l'interface `Affichable` pour les dossiers : on affichera la liste des entrées du dossier (`.` et `..` incluses).
7. Pour ajouter une entrée dans un dossier, la méthode générale consiste à commencer par créer une entrée dont l'élément est `null`, puis plus tard on remplira le champ élément en utilisant `remplacer`.
   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.

Exercice 3 Création du shell

On va maintenant créer l'interpréteur de commandes et écrire différentes commandes.

1. Écrire une classe `Shell` qui aura comme attributs un dossier racine et un dossier courant (tous deux du type `Dossier`). Lorsque l'on lance un `Shell`, les deux pointent évidemment vers le même dossier au début (donc le constructeur prend un dossier en paramètre).
2. Écrire une classe abstraite `CommandeShell` qui contient comme attributs un dossier racine, un dossier courant, et un `String[]` de paramètres.
   — Écrire un constructeur qui reçoit les valeurs pour ces attributs.
   — Déclarer une fonction abstraite publique `executer()` qui renvoie un objet de type `Dossier`. (C'est une préparation pour la commande `cd`, qui peut changer le dossier courant.)
   — Écrire une méthode `public static void aide()` qui ne fait rien pour le moment, donc son corps est juste `return`. Les commandes concrètes vont la remplacer avec une affichage d'un manuel de la commande.
   (Remarque : on a envie que la méthode soit statique et abstraite, mais Java ne le permet pas.)
   — Écrire une méthode `protected static void erreurParam()` qui affiche `"Pas un bon nombre de paramètres."` et qui appelle la méthode `aide()` après.
   — Définir une méthode `protected Entree acceder(String chemin, boolean creer)` qui renvoie l'entrée correspondant à un chemin d'accès, ou affiche une erreur et renvoie `null`.

On vérifiera si le chemin commence par `/` (chemin absolu) ou non (chemin relatif au dossier courant).

On pourra utiliser un `Scanner` sur lequel on a appelé `useDelimiter("/")` pour découper le chemin.

Le constructeur sert à créer une commande avec toutes les informations dont elle a besoin. Puis, `executer()` éxecute la commande en utilisant ces informations à l'intérieur, et aussi en utilisant la fonction `acceder()` pour interpréter les chemins donnés en paramètres.

Si une commande reçoit un nombre de paramètres avec lequel elle ne peut pas travailler, elle affiche une erreur en utilisant `erreurParam()`.

3. Implementez les commandes suivantes en héritant de `CommandeShell`. Pour toutes ces commandes, il faut implémenter un constructeur (qui utilise `super`), et les méthodes `executer()` et `aide()`.

   La méthode `aide()` doit afficher une seule ligne par commande, et l'affichage concrete par commande est comme suit :

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

   C'est donc seulement un manuel minimal, qui indique le nombre de paramètres et si un paramètre est optionel (avec `[]`).

   (a) `CommandeMkdir`. Dans la console, `mkdir nom` crée un dossier `nom` dans le dossier courant.

   (b) `CommandeCat`. Dans la console, `cat nom` affiche un élément `Affichable` du nom `nom`, donc soit le contenu du fichier texte `nom`, soit la liste d'entrées du dossier `nom`.

   (c) `CommandeLs`. Dans la console, `ls` affiche
   — le dossier courant si on ne lui donne pas de paramètres,
   — la liste d'entrées d'un dossier si on donne le chemin d'un dossier en paramètre,
   — l'entrée d'un fichier texte si on donne le chemin d'un fichier texte en paramètre.

   (d) `CommandeCd`. Dans la console, `cd` change le dossier courant. On change dans
   — le dossier racine si `cd` est utilisé sans paramètres, ou
   — dans le dossier indiqué en paramètre.

   La commande `cd` est la seule commande dans ce sujet qui change le dossier courant. Elle renvoie le nouveau dossier courant, et le shell (on va l'écrire plus tard) doit utiliser cette valeur pour mettre à jour son dossier courant.

   (e) `CommandeEd`. Dans la console, `ed nom` écrit un fichier texte avec le contenu indiqué par l'utilisateur (voir Exercice 1). Comme on a définit la méthode `editer(Scanner sc, boolean echo)` avec un scanner et un booléen en paramètre, le constructeur d'une commande `CommandeEd` va recevoir un scanner et un booléen en paramètre, et la commande transmettra ceux à la méthode `editer` du fichier.

   (f) `CommandeMv`. Dans la console, `mv source destination` déplace une entrée `source` vers le chemin `destination`.

Si `destination` est un dossier, on déplacera `source` en gardant son nom dans le dossier `destination`.

On n'écrasera pas un fichier déjà existant.

Si `source` est un dossier et `destination` se trouve à l'intérieur de `source`, on affiche le message d'erreur `Pas possible de déplacer un dossier dans lui-même.` Pour cela, on pourrait implémenter dans `Dossier` une fonction `public boolean estEnfantDe(Dossier o)`.

(g) `CommandeRm`. Dans la console, `rm` supprime une entrée, n'importe si c'est un dossier ou un fichier texte. Les entrées speciales ne peuvent pas être supprimés.

4. Dans la classe `Shell`, implémenter une méthode `public void interagir(InputStream in)` qui lit, en boucle, des commandes (au clavier ou d'un fichier) et les découpe en mots.

   Une commande est terminé par un retour à la ligne, comme on le connaît de la console.

   On utilisera un `Scanner` pour récupérer les lignes, et un deuxième `Scanner` pour découper la ligne en mots.

   Pour le premier scanner : il est possible de construire un scanner à partir d'un `InputStream` :

   ```
   Scanner sc = new Scanner(in);
   ```

   Quand la commande est découpé en mots et donc dans ses paramètres individuels, on peut construire un objet de la commande correspondante, et l'éxecuter.

   Pensez aux points suivants :

   (a) on veut permettre deux commandes ad-hoc qui n'étaient pas définis ci-dessus :
   — `quit` va terminer le programme,
   — `help` va afficher tous les manuels des commandes (y inclus `quit` et `help`)

   (b) la commande `cd` peut changer le dossier courant, donc le shell doit utiliser la valeur renvoyé par cette commande (pour les autres commandes, le shell peut ignorer la valeur renvoyé).

   (c) la commande `ed` a besoin d'un scanner et un booléen. Le shell doit les fournir (au constructeur de la commande). On donne le scanner `sc` défini ci-dessus, et comme booléen on donne `(in != System.in)`.

5. Écrire une classe `TerminalEmulator.java` comme suit :

   ```
   import java.util.*;

   class TerminalEmulator {
     public static void main(String[] args) {
       Dossier racine = new Dossier(null);
       Shell s = new Shell(racine);
       s.interagir(System.in);
     }
   }
   ```

   Elle crée un dossier racine, et un shell, et commence une interaction avec le shell via la console (donc on lit de `System.in`). Testez votre programme un peu dans ce mode interactif.

6. Utilisez la classe `Test.java` et le fichier `test00input` fourni avec ce TP pour tester votre programme automatiquement :

```
java Test test00input
```

Pour savoir si votre programme satisfait ce test, comparez la sortie avec le fichier `test00output`.

Pour faire cette comparaison automatiquement, vous pourriez essayer le suivant : Si vous avez une console compatible POSIX (sous Linux ou MacOS c'est fourni par défaut, sous Windows vous pouvez l'installier avec Cygwin ou Windows Subsystem for Linux), vous pouvez faire le suivant :

```
java Test test00input > output
diff output test00output
```

Le caractère `>` redirige la sortie d'une commande vers un fichier. La commande `diff` compare deux fichiers. Si `diff` n'affiche rien, les deux fichiers sont égaux.

Ce cas de test ne teste pas toute les possibles failles de ce sujet, mais certaines fonctionalités de base. Les enseignants testent votre programme sur des cas de tests possiblement plus larges.

En tout cas, ce cas de test peut servir pour clarifier toute amiguité concernant l'affichage souhaite du programme.

Exercice 4 Copie (Optionnel)

On va ajouter une commande `cp` pour copier des fichiers et des dossiers.

On va implémenter cela à l'aide de la méthode `clone()`. On veut signaler dans la classe `Element` que tous les éléments sont clonables : on fait implémenter `Cloneable` à `Element` et on redéfinit `clone()` de la façon suivante pour garantir qu'elle ne renvoie pas d'exception (ainsi, on n'a pas à mettre de clause `throws` dans sa signature) :

```
1  abstract class Element implements Cloneable {
2      (...)
3      public Element clone() {
4          try {
5              return (Element) super.clone();
6          } catch (CloneNotSupportedException e) {
7              throw new InternalError();
8          }
9      }
10 }
```

1. Implémenter l'opération `cp source destination` dans le shell à l'aide de cette méthode `clone()`.
2. Pourquoi cette implémentation n'est pas satisfaisante ? Donner une séquence de commandes qui le montre.
3. Redéfinir la méthode `clone()` dans `Dossier` pour que la copie soit correcte. On pourra définir sur la classe `Entree` une méthode
   `public Entree clone(Dossier contenant)` qui crée une copie de l'entrée dans le nouveau dossier `contenant`.
4. Testez votre implémentation avec le cas de test `test01`.