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 noms 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 :

```
abstract class Element {
    public abstract String getType();

    public String toString() {
```

```
        return "fichier de type " + getType();
    }
}
```

Fichiers texte

1. Écrivez 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. Écrivez une interface `Affichable` destinée aux élèmentspouvant être affichés : ils devront posséder une méthode `public void afficher()` montrant leur contenu à l'écran.
3. Implémentez maintenant l'interface `Affichable` dans `FichierTexte`.
4. Écrivez une interface `Editable` destinée aux élèments pouvant être édités : ils devront posséde une méthode `public void editer(Scanner sc, boolean echo)` qui permettra à l'utilisateur de modifier l'attribut `contenu` du fichier texte.

   Remarquer que le scanner est transmis en argument. Ainsi, on pourra utiliser la même méthode pour éditer un `Element` à partir des données saisies au clavier ou bien à partir d'un autre flux (fichier réel).
5. Faites que `FichierTexte` implémente `Editable` :

   Vous utiliserez le `Scanner` pour lire ligne par ligne son entrée. Une ligne contenant qu'un point `"."` marquera la fin de l'édition. Ce point final n'appartient pas au contenu du fichier.

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

   Cela peut être utile lorsque le scanner lit depuis une autre source que le clavier. On peut alors 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 lui étant associée. Les associations sont définies à l'aide d'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 entrées décorent les élèments :

```
class Entree {
    private Element element;
    private String nom;
    private Dossier parent;
    public Entree(Dossier p, String n, Element e) { ... }
    ...
}
```

On y reconnaît :
- l'encapsulation d'un élément,
- la décoration de nom qui lui est associé ,
- la décoration mentionnant le 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. Écrivez un classe `Dossier` possédant deux attributs :
   - une liste d'`Entree`s qui correspond à son contenu
   - une `Entree` qui correspond à son dossier parent

   Écrivez un constructeur de dossier vide. Il ne prend en argument que le dossier déclaré parent.
2. Dans `Entree`, ajoutez des méthodes `getNom()` et `getElement()`.
3. Créez une méthode `String toString()` dans `Entree` qui retourne une chaîne de la forme `"nom (type)"` où `nom` est celui de l'entrée et `type` est le type 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 (l'attribut `element` égal à `null`).
4. Dans `Entree`, créez une méthode `public void supprimer()` permettant de supprimer l'entrée. 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.
5. Dans `Entree`, écrivez 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`.) Ajoutez aussi les méthodes setter nécessaires à la classe `Dossier`.
6. 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`. Définissez une classe `EntreeSpeciale` héritant de `Entree` affichant un message d'erreur plutôt que d'effectuer ces deux opérations.
7. Écrivez une méthode `ajouter(Element e, String nom)` dans la classe `Dossier` permettant d'ajouter une `Entree` à la liste d'entrées du dossier. La méthode consiste d'abord à créer une entrée dont l'élément encapsulé est `null`, puis d'utiliser `remplacer`.
8. Implémentez l'interface `Affichable` pour les dossiers. On affichera la liste des entrées du dossier. N'oubliez pas d'ajouter `.` et `..` à la liste d'entrées pendant la construction d'un objet `Dossier`.
9. Dans la classe `Dossier`, écrivez 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 ?

10. 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 ?

Préparation à l'écriture de quelques commandes

Maintenant que nous avons l'ensemble des classes modélisant les fichiers et répertoires, nous passerons au développement du shell permettant de les manipuler.

Le but de cette partie du TP est de réaliser les commandes suivantes.

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

Création du shell

1. Définissez une classe `Shell` qui aura comme attributs un dossier racine et un dossier courant (tous deux du type `Dossier`). On veut pouvoir détenir un `Shell` uniquement à partir d'un dossier courant. Il vous faudra alors pour initialiser correctement l'attribut `racine` remonter à sa racine.
2. Écrivez une classe abstraite `CommandeShell` possédant trois attributs :
   — un dossier racine,
   — un dossier courant, et
   — un `String[]` de paramètres.
3. Écrivez un constructeur qui reçoit les valeurs pour ces attributs.
4. Déclarez une fonction abstraite publique `executer()` qui renvoie un objet de type `Dossier`. (En anticipation de la commande `cd` permettant de changer le dossier courant.)
5. Écrivez une méthode `public static void aide()` qui ne fait rien pour le moment. Mettez simplement l'instruction `return` dans son corps. Nous y placerons plus tard les commandes affichant le manuel de la commande.

   (Remarque : on a envie que la méthode soit statique et abstraite, mais Java ne le permet pas.)
6. Écrivez une méthode `protected static void erreurParam()` qui affiche `"Pas un bon nombre de paramètres."` et qui appelle la méthode `aide()` après.
7. Définissez une méthode `protected Entree acceder(String chemin, boolean creer)` qui renvoie l'entrée correspondant à un chemin d'accès. On affiche une erreur et renvoie `null` si le chemin est invalide.

On vérifiera si le chemin commence par `/` (chemin absolu) ou non (chemin relatif au dossier courant). Le début du chemin sera donc soit le dossier `racine` ou le premier dossier dans le chemin donné.

On pourra utiliser un `Scanner` sur lequel on a appelé `useDelimiter("/")` pour découper le chemin. Pensez à utiliser la méthode `getEntree()` définie dans le premier exercice.

Le constructeur sert à créer une commande avec toutes les informations dont elle a besoin. Puis, `executer()` exécute 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()`.

8. Implémentez 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 optionnel (avec `[]`).

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

   (b) `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.

   (c) `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.

9. Dans la classe `Shell`, implémentez 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 la méthode `split()` de la classe `String` pour découper la ligne en mots.

   Pour le 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'exécuter.

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).

10. 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.

Bonus

Rajoutez des classes implémentant des commandes suivantes :

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

— `CommandeEd`. Dans la console, `ed nom` écrit un fichier texte avec le contenu indiqué par l'utilisateur (voir le premier exercice). 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.
La commande `ed` a besoin d'un scanner et un booléen. Dans la classe `Shell` on doit donc les fournir (au constructeur de la commande). On donne le scanner `sc` défini dans l'exercice précedent, et comme booléen on donne `(in != System.in)`.

— `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)`.

— `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.

— `CommandeCp`, Dans la console, `mv source destination` copie une entrée `source` vers le chemin `destination`.

On peut implementer cette opération 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) :

```
abstract class Element implements Cloneable {
    (...)
    public Element clone() {
        try {
            return (Element) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
}
```