TP n° 3
Licence Informatique (L2)
« Programmation objet avancée »

F. BERTRAND

Année universitaire 2020–2021

Éléments de correction

1 Système de fichiers

Dans ce TP nous allons reprendre l'exemple vu en cours concernant la représentation d'un système de fichiers. Soit une représentation très simplifiée d'un système de fichiers dont le diagramme de classes est donné ci-dessous sur la Figure 1.

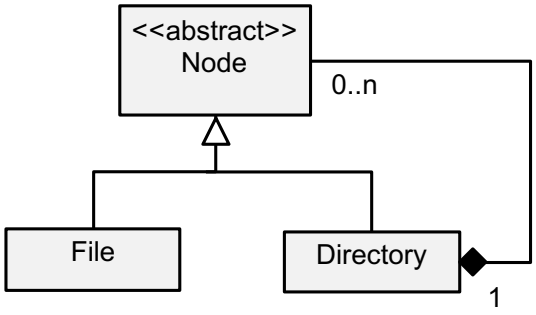

FIGURE 1 – Représentation (simplifiée) d'un système de fichiers

Éléments à noter :

— La classe `Node` factorise les éléments communs aux fichiers (`File`) et aux répertoires (`Directory`) comme la notion de nom mais elle pourrait contenir d'autres éléments comme les droits en écriture/lecture.
— Un répertoire peut contenir des fichiers et d'autres (sous-)répertoires alors qu'un fichier est un élément ne pouvant contenir aucun autre élément.

Le code Java donné ci-dessous décrit l'implémentation de ces trois classes :

```
1  import java.util.ArrayList;
2
3  abstract class Node
4  {
5
6      private String name;
7
8      public Node(String nom)
9      {
10         this.name = nom;
11     }
12
13     public String getName()
```

```
14      {
15          return this.name;
16      }
17
18      public void explore()
19      {
20          System.out.println(this.getName());
21          if (this.getClass() == Directory.class)
22          {
23              Directory rep = (Directory) this;
24              ArrayList<Node> content = rep.getChildren();
25              for (Node n : content)
26              {
27                  n.explore();
28              }
29          }
30      }
31  }
32
33  class File extends Node
34  {
35
36      public File(String name)
37      {
38          super(name);
39      }
40  }
41
42  class Directory extends Node
43  {
44
45      private ArrayList<Node> children;
46
47      public Directory(String name)
48      {
49          super(name);
50          this.children = new ArrayList<Node>();
51      }
52
53      public void addNode(Node n)
54      {
55          this.children.add(n);
56      }
57
58      public ArrayList<Node> getChildren()
59      {
60          return this.children;
61      }
62  }
63
64  public class TestFileSystem
65  {
66      public static void main(String[] args)
67      {
68          Directory r1 = new Directory("A");
69          Directory r2 = new Directory("B");
70          File f1 = new File("f1");
71          File f2 = new File("f2");
72          r1.addNode(r2);
73          r2.addNode(f2);
74          r1.addNode(f1);
75          r1.explore();
76      }
77  }
```

Travail à réaliser :

— Tout d'abord créer une classe par fichier (bonne habitude à prendre...);

— Créer une classe `FileSystem` représentant un système de fichiers ayant une racine unique de type `Directory` qui sera créée automatique lors de l'instanciation de la classe `FileSystem`;

— Modifier ce code pour retirer le test de la ligne 24 en utilisant la notion de polymorphisme. L'idée générale (expliquée dans le cours) est de permettre à chaque entité (fichier ou répertoire) de posséder des méthodes `getChildren` et `addNode` mais de se comporter de manière différente selon la classe où cette méthode sera implémentée.

— Sur le même modèle (déclaration comme méthode abstraite dans `Node` et implémen-

tation dans les sous-classes) implémenter une méthode `getSize` qui :
  - pour un fichier retournera sa taille (attribut à ajouter à la classe `File`);
  - pour un répertoire retournera la taille des éléments qu'il contient (ici on simplifie car dans la réalité un répertoire possède une taille même si celui-ci ne contient aucun élément).
- Mettre en œuvre la notion de parent (qui ne pourra être que de type `Directory`) de telle manière que toute entité, fichier ou répertoire, ait un parent. Le test présent dans la classe de test présentée ci-dessous montre un exemple de ce qu'on souhaite implémenter.
- Pour terminer, question optionnelle, créer une classe `SymbolicLink` représentant un lien symbolique.

La nouvelle version de la classe de test :

```
public class TestFileSystem {
    public static void main(String[] args) {
        Directory dirA = new Directory("A");
        Directory dirB = new Directory("B");
        FileSystem fs = new FileSystem();
        fs.getRoot().addNode(dirA);
        File f1 = new File("f1",120);
        File f2 = new File("f2", 340);
        dirA.addNode(dirB);
        dirB.addNode(f2);
        dirA.addNode(f1);
        if (fs.getSize() != 460) {
            throw new Error("Calcul taille repertoire incorrect");
        }
        if (f1.getParent() != dirA || f2.getParent() != dirB || dirB.getParent() != dirA) {
            throw new Error("Mauvaise gestion du lien parent");
        }
        dirA.explore();
    }
}
```

Notes

L'idée ici est, comme indiquée dans l'énoncé, de :

- *montrer l'application du polymorphisme au travers de différentes méthodes comme `getSize()` et `getParent()` pour vérifier si cette notion a bien été appréhendée ;*
- *déclarer abstraites dans `Node` toutes les méthodes devant être implémentées par ses sous-classes ;*
- *montrer, avec la classe `SymbolicLink`, un exemple d'utilisation de la notion de délégation (un objet qui reçoit un appel de méthode le fait suivre à un autre objet) .*

Le code des classes :

```
1  /* Représente un système de fichiers qui a une racine unique constituée
2   * d'un répertoire.
3   */
4
5  public class FileSystem {
6      // un systeme de fichiers a une racine unique
7      private Directory root;
8
9      public FileSystem() {
10         this.root = new Directory("");
11     }
12
13     public Directory getRoot() {
14         return root;
15     }
16
17     public int getSize() {
18         return this.root.getSize();
```

```
    }
}
```

```
import java.util.ArrayList;

/* Représente une classe factorisant les éléments communs à l'ensemble
 * des classes représentant les objets constituant un système de fichiers.
 */
public abstract class Node {

    private String name;
    private Directory parent;

    public Node(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public Directory getParent() {
        return this.parent;
    }

    public void setParent(Directory parent) {
        this.parent = parent;
    }

    public abstract ArrayList<Node> getChildren();

    public abstract void addNode(Node n);

    public abstract int getSize();

    public void explore() {
        System.out.println(this.getName());
        for (Node n : this.getChildren()) {
            n.explore();
        }
    }
}
```

```
import java.util.ArrayList;

/* Représente la notion (simplifiée) de fichier dans un système de fichiers.
 */
public class File extends Node {
    private int size;

    public File(String name, int size) {
        super(name);
        this.size = size;
    }

    @Override
    public int getSize() {
        return this.size;
    }

    @Override
    public ArrayList<Node> getChildren() {
        return new ArrayList<Node>();
    }

    @Override
    public void addNode(Node n) {
        throw new UnsupportedOperationException(n.toString());
    }
}
```

```
import java.util.ArrayList;

/* Représente la notion (simplifiée) de répertoire dans un système de fichiers.
 */
class Directory extends Node {

    private ArrayList<Node> children;
```

```
8
9       public Directory(String name) {
10          super(name);
11          this.children = new ArrayList<Node>();
12          this.setParent(null);
13      }
14
15      public void addNode(Node n) {
16          // on ne peut pas ajouter un répertoire à lui-même
17          // ni l'ajouter 2 fois...
18          // ces tests sont cependant trop élémentaires pour s'assurer
19          // qu'on ne crée pas de cycles dans le système de fichiers
20          if (n != this && !this.children.contains(n)) {
21              this.children.add(n);
22              n.setParent(this);
23          }
24      }
25
26      @Override
27      public ArrayList<Node> getChildren() {
28          return this.children;
29      }
30
31      @Override
32      public int getSize() {
33          int sizeSum = 0;
34          for (Node n : this.getChildren()) {
35              sizeSum += n.getSize();
36          }
37          return sizeSum;
38      }
39
40  }
```

```
1   import java.util.ArrayList;
2
3   /* Représente un lien symbolique dans un système de fichiers.
4    * Cette classe utilise un mécanisme connu en programmation objet dit
5    * de délégation : les appels arrivant à un objet de cette classe sont
6    * "re-routés" (via le polymorphisme) à l'objet référencé par le lien.
7    */
8   public class SymbolicLink extends Node {
9       private Node link;
10
11      public SymbolicLink(String name, Node link) {
12          super(name);
13          this.link = link;
14      }
15
16      @Override
17      public ArrayList<Node> getChildren() {
18          return this.link.getChildren();
19      }
20
21      @Override
22      public int getSize() {
23          return this.link.getSize();
24      }
25
26      @Override
27      public void addNode(Node n) {
28          this.link.addNode(n);
29      }
30  }
```

2 Composants informatiques

On souhaite représenter, sous forme de classes, des composants intervenant comme éléments d'un PC. Nous avons les trois composants suivants :

— le disque dur possédant les informations suivantes : identifiant, prix, consommation, capacité ;

— l'alimentation possédant les informations suivantes : identifiant, prix, puissance ;

— le processeur possédant les informations suivantes : identifiant, prix, consommation, fréquence, type de connecteur (*socket*) ;
— une barrette mémoire possédant les informations suivantes : identifiant, prix, fréquence, consommation.

Travail à réaliser :

1. Mettre en œuvre une hiérarchie de classes ayant comme racine la classe `Composant` regroupant les attributs communs à l'ensemble des classes.
2. Puis créer une classe `PC` qui possédera une liste de composants.
3. En considérant que chaque composant à une méthode `donneEnergie` qui retourne soit une valeur négative si le composant consomme de l'énergie, soit une valeur positive si le composant en fourni (cas de l'alimentation), vérifier lors de la constitution du PC (de préférence avant l'appel du constructeur) que le bilan énergétique des composants est positif (ou nul à la rigueur).
4. Créer une classe `TestAssemblagePC` vérifiant l'assemblage correct et incorrect d'un PC.

Notes

Le diagramme de classes représentant la modélisation du problème :

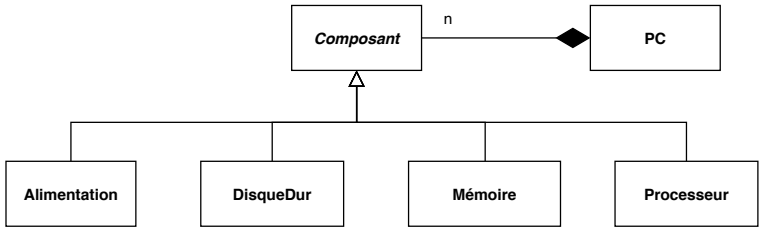

FIGURE 2 – Hiérarchie d'héritage

L'idée ici est de faire de la classe `Composant` une classe abstraite, factorisant tous les attributs communs entre les différents types de composant et de faire de la méthode `donneEnergie` une méthode abstraite dans `Composant`, forçant ainsi ses sous-classes à l'implémenter...

Les classes :

```
1  /* Représente un composant d'un PC */
2  public abstract class Composant {
3      private String identifiant;
4      private float prix;
5
6      public Composant(String identifiant,
7                       float prix) {
8          this.identifiant = identifiant;
9          this.prix = prix;
10     }
11
12     public abstract int donneEnergie();
13 }
```

```java
/* Représente une alimentation d'un PC */
public class Alimentation extends Composant {
    private int puissance;

    public Alimentation(String identifiant,
                        float prix,
                        int puissance) {
        super(identifiant, prix);
        this.puissance = puissance;
    }

    @Override
    /* Elle produit de l'energie donc l'energie est positive... */
    public int donneEnergie() {
        return this.puissance;
    }
}
```

```java
/* Représente un disque dur de PC */
public class DisqueDur extends Composant {
    private int consommation;
    private int capaciteEnGo;

    public DisqueDur(String identifiant,
                     float prix,
                     int consommation,
                     int capaciteEnGo) {
        super(identifiant, prix);
        this.consommation = consommation;
        this.capaciteEnGo = capaciteEnGo;
    }

    @Override
    /* Il consomme donc l'energie est negative... */
    public int donneEnergie() {
        return - this.consommation;
    }
}
```

```java
/* Représente une barrette mémoire d'un PC */
public class Memoire extends Composant {
    private float frequenceEnMHz;
    private int consommation;

    public Memoire(String identifiant, float prix, float frequenceEnMHz, int consommation) {
        super(identifiant, prix);
        this.frequenceEnMHz = frequenceEnMHz;
        this.consommation = consommation;
    }

    @Override
    /* Elle consomme donc l'energie est negative... */
    public int donneEnergie() {
        return - this.consommation;
    }
}
```

```java
/* Représente un microprocesseur de PC */
public class Processeur extends Composant {
    private float frequenceEnGHz;
    private int consommation;
    private ConnecteurProcesseur connecteur;

    public Processeur(String identifiant,
                      float prix,
                      int consommation,
                      float frequenceEnGHz,
                      ConnecteurProcesseur connecteur) {
        super(identifiant, prix);
        this.frequenceEnGHz = frequenceEnGHz;
        this.consommation = consommation;
        this.connecteur = connecteur;
    }

    @Override
    /* Il consomme donc l'energie est negative... */
    public int donneEnergie() {
        return - this.consommation;
    }
}
```

```java
/* Représente les différentes valeurs possibles pour un connecteur
 * de processeur.
 */
public enum ConnecteurProcesseur
{
        TR4, LGA_1151, LGA_2066
}
```

```java
import java.util.ArrayList;
import java.util.Arrays;

public class PC {
    private ArrayList<Composant> composants;

    // Constructeur en privé pour obliger à passer par la
    // méthode de classe createPC
    private PC(Composant... elts) {
        this.composants = new ArrayList<>();
        this.composants.addAll(Arrays.asList(elts));
    }

    /* Permet d'assembler un PC à partir d'un ensemble de composants.
     * Utilisation comme paramètres d'un nombre variable d'arguments qui
     * se manipule comme un tableau.
     * Ce n'est pas obligatoire nous aurions pu utiliser une liste...
     * L'avantage de ce mécanisme (connu sous le nom de Fabrique) c'est de n'appeler le
     * constructeur que si les paramètres sont OK.
     * En effet si on s'aperçoit qu'il y a un problème avec les paramètres lorsque
     * l'exécution se situe dans le constructeur, la mémoire a malheureusement
     * déjà été allouée.
     */
    public static PC createPC(Composant... elts) {
        int bilanEnergie = 0;
        for(Composant c : elts) {
            bilanEnergie += c.donneEnergie();
        }
        if (bilanEnergie >= 0) {
            return new PC(elts);
        } else {
            return null;
        }
    }
}
```

```
public class TestAssemblagePC {
    public static void main(String[] args) {
        Alimentation alim1 = new Alimentation("PW125", 89, 150);
        DisqueDur hd1 = new DisqueDur("BRC3G", 78, 10, 3);
        Memoire ram1 = new Memoire("CRC188", 3, 2500, 5);
        Processeur cpu1 = new Processeur("ITLi9", 250, 120, 3.4f, ConnecteurProcesseur.LGA_2066);
        PC monPC = PC.createPC(alim1, hd1, ram1, ram1, cpu1);
        if (monPC == null) {
            System.out.println("Probleme car l'alimentation est assez puissante");
        }
        Alimentation alim2 = new Alimentation("PW125", 89, 100);
        PC unAutrePCmalConstruit = PC.createPC(alim2, hd1, ram1, ram1, cpu1);
        if (unAutrePCmalConstruit != null) {
            System.out.println("Probleme car l'alimentation n'est pas assez puissante");
        }
    }
}
```