Examen (avec document)
Corrigé

Préambule : Répondre de manière concise et précise aux questions. Ne pas mettre de commentaires de documentation sauf s'ils sont nécessaires à la compréhension.

Barème indicatif :

exercice	1	2	3	4	5
points	3	4	5	5	3

L'objectif de ces exercices est de définir des tâches hiérarchiques et de les exploiter de différentes manières.

Exercice 1 : Compréhension de l'architecture des tâches

L'architecture des tâches est donnée à la figure 1 où le détail des classes TacheElementaire et TacheComplexe n'est pas donné. Une tâche est caractérisée par un nom et un coût. Une tâche est soit une tâche élémentaire, soit une tâche complexe qui est alors composée de sous-tâches. Il est ainsi possible d'ajouter une sous-tâche à une tâche complexe, `ajouter(Tache)` ou de supprimer une sous-tâche, `supprimer(Tache)`. Le coût d'une tâche complexe est la somme des coûts des tâches qui la composent.

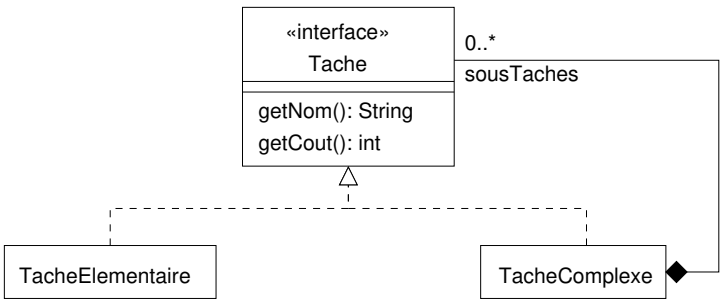

FIG. 1 – Architecture des tâches hiérarchiques

1.1 Indiquer le ou les patrons de conception utilisés dans cette architecture.

Solution : Composite : La tâche complexe est le composite, Tache est le composant et TacheElementaire est la feuille. Notons que les objets ajouter et supprimer ne sont ici que sur le composite et pas sur le composant.

1.2 Le listing 1 donne le code de l'interface Tache.

Écrire en Java la classe TacheElementaire qui est une réalisation de l'interface Tache.

Listing 1 – L'interface Tache

```
1  public interface Tache {
2      /**  Obtenir le nom de la tâche. */
3      String getNom();
4
5      /** Obtenir le coût de la tâche. */
6      int getCout();
7  }
```

Solution :

```
1  public class TacheElementaire implements Tache {
2      private String nom;
3      private int cout;
4
5      public TacheElementaire(String nom, int cout) {
6          this.nom = nom;
7          this.cout = cout;
8      }
9
10     public String getNom() {
11         return this.nom;
12     }
13
14     public int getCout() {
15         return this.cout;
16     }
17
18 }
```

Exercice 2 : Définition d'une tâche complexe

Nous nous intéressons maintenant à la classe TacheComplexe, en particulier à sa relation avec l'interface Tache. Une tâche complexe est composée d'un nombre quelconque de tâches. On décide d'utiliser l'interface `java.util.Collection` pour stocker les sous-tâches. On l'utilisera bien entendu dans sa version générique.

Comme on souhaite pouvoir parcourir toutes les sous-tâches d'une tâche complexe, la classe TacheComplexe réalise l'interface `java.lang.Iterable`.

2.1 Indiquer quel est le principal intérêt de la généricité.

Solution : Le contrôle de type réalisé par le compilateur.

2.2 Indiquer quel est le coût de la tâche `tA` construite comme indiqué dans le listing 2.

Listing 2 – La classe TestTache1

```
1  public class TestTache1 {
2      public static void main(String[] args) {
3          TacheComplexe tA = new TacheComplexe("A");
4          tA.ajouter(new TacheElementaire("A1", 10));
5          tA.ajouter(new TacheElementaire("A2", 20));
6
7          System.out.println("Cout␣de␣tA␣=␣" + tA.getCout());
8      }
9  }
```

Solution : 30

2.3 Écrire en Java la classe TacheComplexe.

Solution :

```java
import java.util.Collection;
import java.util.ArrayList;
import java.util.Iterator;

public class TacheComplexe implements Tache, Iterable<Tache> {
    private Collection<Tache> sousTaches;
    private String nom;

    public TacheComplexe(String nom) {
        this.nom = nom;
        this.sousTaches = new ArrayList<Tache>();
    }

    public void ajouter(Tache tache) {
        this.sousTaches.add(tache);
    }

    public void supprimer(Tache tache) {
        this.sousTaches.remove(tache);
    }

    public String getNom() {
        return this.nom;
    }

    public int getCout() {
        int result = 0;
        for (Tache t : sousTaches) {
            result += t.getCout();
        }
        return result;
    }

    public Iterator<Tache> iterator() {
        return this.sousTaches.iterator();
    }

}
```

Exercice 3 : Interface graphique pour définir une tâche complexe

Nous définissons maintenant une interface graphique en Swing minimale qui permet d'ajouter de nouvelles sous-tâches à une tâche complexe. Le code partiel de cette classe est donné au listing 3.

3.1 Dessiner la fenêtre (et les composants graphiques qu'elle contient) telle qu'elle est affichée quand cette classe est exécutée.

Solution :

3.2 Compléter cette classe pour que les boutons `Ajouter` et `Quitter` deviennent actifs. Le bouton `Quitter` ferme la fenêtre. Le bouton `Ajouter` ajoute une nouvelle sous-tâche à la tâche complexe passée en paramètre du constructeur de cette classe. Le nom et le coût de cette sous-tâche sont, bien entendu, saisis par l'utilisateur dans les zones de saisie prévues, `valeurNom` et `valeurCout`. Dans le cas où l'utilisateur saisit une information qui n'est pas un entier pour saisir le coût, on signalera l'erreur en mettant la couleur de fond de la zone de saisie correspondante en rouge (`setBackground(Color.RED)`). On rappelle que la méthode `Integer.parseInt(String)` renvoie l'entier correspondant à la chaîne de caractère passé en paramètre. Cette méthode lève l'exception `NumberFormatException` si la chaîne ne correspond pas à un entier.
Remarque : On pourra mettre des numéros sur le listing 3 pour indiquer où du code doit être ajouté, le code correspondant étant écrit sur la copie, précédé du même numéro.

Listing 3 – La classe TacheComplexeSwing

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  public class TacheComplexeSwing {
6
7      private TacheComplexe tache;
8      final JFrame fenetre = new JFrame("Nouvelle␣tâche");
9      final private JTextField valeurNom = new JTextField(10);
10     final private JTextField valeurCout = new JTextField(10);
11     final private JButton boutonAjouter = new JButton("Ajouter");
12     final private JButton boutonQuitter = new JButton("Quitter");
13
14     public TacheComplexeSwing(TacheComplexe tache) {
15         this.tache = tache;
16         Container c = fenetre.getContentPane();
17         c.setLayout(new BorderLayout());
18         JPanel informations = new JPanel(new GridLayout(2,2));
19         informations.add(new JLabel("Nom␣:␣", SwingConstants.RIGHT));
20         informations.add(valeurNom);
21         informations.add(new JLabel("Coût␣:␣", SwingConstants.RIGHT));
22         informations.add(valeurCout);
23         c.add(informations, BorderLayout.CENTER);
24
25         JPanel boutons = new JPanel(new FlowLayout());
26         boutons.add(boutonAjouter);
27         boutons.add(boutonQuitter);
28         c.add(boutons, BorderLayout.SOUTH);
29
30
31         boutonQuitter.addActionListener(new ActionQuitter());
32         boutonAjouter.addActionListener(new ActionAjouter());
```

```
33
34
35          fenetre.pack();
36          fenetre.setVisible(true);
37
38      }
39
40
41      private class ActionAjouter implements ActionListener {
42
43          public void actionPerformed(ActionEvent ev) {
44              try {
45                  String nom = valeurNom.getText();
46                  int cout = Integer.parseInt(valeurCout.getText());
47                  tache.ajouter(new TacheElementaire(nom, cout));
48                  System.out.println("cout total = " + tache.getCout());
49              } catch (NumberFormatException e) {
50                  valeurCout.setBackground(Color.RED);
51              }
52          }
53
54      }
55
56      private class ActionQuitter implements ActionListener {
57
58          public void actionPerformed(ActionEvent ev) {
59              System.out.println("Appui sur Quitter...");
60              fenetre.dispose();
61          }
62
63      }
64
65      public static void main(String[] args) {
66          new TacheComplexeSwing(new TacheComplexe("Test TacheComplexeSwing"));
67      }
68
69
70  }
```

Exercice 4 : Sauvegarde d'une tâche en XML

Intéressons nous maintenant à la sauvegarde d'une tâche sous la forme d'un fichier XML dont la DTD est donnée au listing 4

Listing 4 – La DTD pour représenter une tâche

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!ELEMENT taches (tache*)>
4  <!ELEMENT tache (attribut*, tache*)>
5  <!ELEMENT attribut EMPTY>
6  <!ATTLIST attribut
7                  nom    CDATA #REQUIRED
8                  valeur CDATA #REQUIRED>
```

4.1 Donner le contenu du fichier XML qui correspond à la tâche `tA` construite dans le listing 2. Ne pas donner l'entête de ce fichier XML.

Solution :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taches SYSTEM "taches.dtd">

<taches>
  <tache>
    <attribut nom="nom" valeur="A" />
    <attribut nom="cout" valeur="30" />
    <tache>
      <attribut nom="nom" valeur="A1" />
      <attribut nom="cout" valeur="10" />
    </tache>
    <tache>
      <attribut nom="nom" valeur="A2" />
      <attribut nom="cout" valeur="20" />
    </tache>
  </tache>
</taches>
```

4.2 Compléter la classe TacheJDom pour engendrer le fichier XML représentant une tâche. Ce fichier devra être valide par rapport à la DTD du listing 4. On ne modifiera aucune des interfaces ou classes Tache, TacheElementaire et TacheComplexe.

Listing 5 – La classe TacheJDom

```java
import org.jdom.*;
import org.jdom.output.*;
import java.io.OutputStream;
import java.io.PrintStream;
import java.io.IOException;
import java.io.FileWriter;

public class TacheJDom {
    private XMLOutputter sortie =
            new XMLOutputter(Format.getPrettyFormat());

    public void printToXML(Tache t, String nomFichier) throws IOException {
        Element racine = new Element("taches");
        racine.addContent(this.getElement(t));
        Document d = new Document(racine, new DocType("taches", "taches.dtd"));
        sortie.output(d, new FileWriter(nomFichier));
    }


    public Element ajouterAttibut(Element e, String nom, String valeur) {
        return e.addContent(
                    new Element("attribut")
                        .setAttribute("nom", nom)
                        .setAttribute("valeur", valeur));
    }


```

```
28      /** Construire l'élément correspondant à la tâche complexe tc. */
29      public Element getElement(TacheComplexe tc) {
30
31          Element elt = new Element("tache");
32          this.ajouterAttibut(elt, "nom", tc.getNom());
33          this.ajouterAttibut(elt, "cout", "" + tc.getCout());
34          for (Tache t: tc) {
35              elt.addContent(getElement(t));
36          }
37          return elt;
38
39      }
40
41      /** Construire l'élément correspondant à la tâche élémentaire te. */
42      public Element getElement(TacheElementaire te) {
43
44          Element elt = new Element("tache");
45          this.ajouterAttibut(elt, "nom", te.getNom());
46          this.ajouterAttibut(elt, "cout", "" + te.getCout());
47          return elt;
48
49      }
50
51      /** Construire l'élément correspondant à la tâche t. */
52      public Element getElement(Tache t) {
53
54          if (t instanceof TacheComplexe) {
55              return getElement((TacheComplexe) t);
56          } else {
57              return getElement((TacheElementaire) t);
58          }
59
60      }
61
62  }
```

4.3 Indiquer les patrons de conception qui auraient pu être utilisés pour engendrer le fichier XML. Il est possible de modifier les interfaces ou classes Tache, TacheElementaire et TacheComplexe.

Solution : Le visiteur. Le visiteur permettrait alors de visiter les classes TacheElementaire et TacheComplexe.

Une autre solution consisterait à définir la méthode getElement() comme abstraite sur Tache et, bien sûr, redéfinie sur les sous-classes.

Exercice 5 : Utilisation de SaX

Étant donné, un fichier XML valide vis-à-vis de la DTD du listing 4, écrire un gestionnaire SaX (ContentHandler ou DefaultHandler) qui affiche le niveau d'imbrication de la tâche la plus interne. Appliqué sur le fichier correspondant à la tâche tA du listing 2, le résultat doit être 2.

Indication : On pourra compter le nombre de balises ouvrantes `<tache>` non fermées.

Solution :

```
1  import org.xml.sax.*;
```

```
2  import org.xml.sax.helpers.*;
3
4  class ProfondeurMaxHandler extends DefaultHandler {
5
6      private int level, max;
7
8      public void startDocument() {
9          this.level = 0;
10         this.max = 0;
11     }
12
13     public void startElement(String uriEspaceNom, String nom,
14                 String nomQualifie, Attributes attributs) throws SAXException {
15         if (nomQualifie.equals("tache")) {
16             this.level++;
17             if (this.level > this.max) {
18                 this.max = this.level;
19             }
20         }
21     }
22
23     public void endElement(String uriEspaceNom, String nom,
24                 String nomQualifie) throws SAXException {
25         if (nomQualifie.equals("tache")) {
26             this.level--;
27         }
28     }
29
30     public int getProfondeurMax() {
31         return this.max;
32     }
33
34 }
```