ECE PARIS
ÉCOLE D'INGÉNIEURS

POO Java
ING3 2018-2019
CORRIGE Examen DS 2

Durée : 1h **AUCUN DOCUMENT - AUCUNE CALCULATRICE**

Lisez attentivement chaque partie du sujet avant de vous lancer dans vos réponses.
Commentez brièvement votre code et importez les librairies nécessaires avant chaque classe.

Des extraits de la documentation Javadoc sont fournis en **ANNEXE**

Présentation du sujet : (source : examen du CNAM 2008)

L'objectif de ces exercices est de définir des tâches hiérarchiques et de les exploiter de différentes manières. L'architecture des tâches est donné par le diagramme des classes 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 ou supprimer une sous-tâche à une tâche complexe, de trier une tâche complexe par coût de sous-tâches ou de sérialiser une tâche complexe.

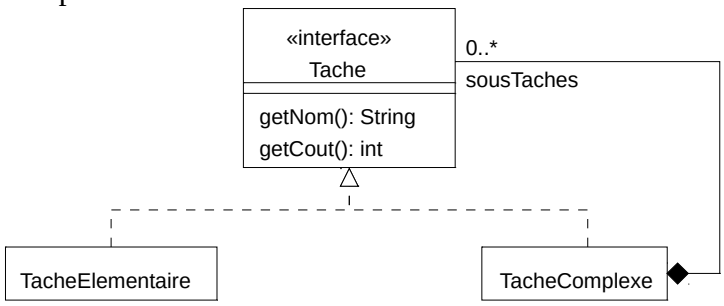

FIG. 1 – Diagramme de classes

Exercice 1 : Compréhension de l'architecture des tâches (6 points). Durée max : 15 min

1.1 Pourquoi Tache est-elle une interface ? Expliquer les relations utilisées dans ce diagramme de classes. (**2 points**)

C'est une interface car elle ne contient que des prototypes de méthodes. (1 point)
Les classe TacheElementaire et TacheComplexe implémentent l'interface Tache. TacheComplexe a une relation de composition avec Tache. (1 point)

1.2 Le listing 1 ci-dessous donne le code de l'interface Tache. Écrire en Java la classe TacheElementaire qui implémente les interfaces Tache et Comparable, en y définissant les attributs privés, le constructeur paramétré et la méthode `compareTo` pour comparer le coût de 2 tâches par ordre croissant. (**4 points**)

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       import java.util.Collections;
2
3       public class TacheElementaire implements Tache, Comparable {
4       private String nom;
5       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          public int compareTo(Tache t) {
19          return this.getCout()-t.getCout() ;
20          }
21
22      }
```

Exercice 2 : Définition et sérialisation d'une tâche complexe (8 points). Durée max : 25 min

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. Le coût d'une tâche complexe est la somme des coûts des tâches qui la composent.

2.1 Écrire en Java la classe TacheComplexe qui implémente les interfaces Tache et `java.io.Serializable` (0,5 pt). Elle contient les caractéristiques suivantes : (**5 points**)

- Des attributs privés : une `ArrayList` de tâches et le nom (0,5 pt)
- Un constructeur paramétré (0,5 pt)
- Une méthode `ajouter` qui ajoute une tâche et une méthode `supprimer` qui la supprime (1 pt)
- Une méthode `trier` qui trie les tâches par ordre croissant de coût (0,5 pt)
- une méthode `serialiser` qui sérialise la tâche complexe dans un fichier dont le nom est en paramètre, ferme le fichier et propage l'exception `IOException` (1 pt)
- La méthode `getCout` retourne le coût total des tâches (0,5 pt)

- La méthode `getTaches` retourne les tâches (0,5 pt)

Solution :

```
1  import java.util.Collections;
2  import java.util.ArrayList;
3  import java.io.*;
4
5      public class TacheComplexe implements Tache, Serializable {
6      private ArrayList<Tache> sousTaches;
7      private String nom;
8
9          public TacheComplexe(String nom) {
10         this.nom = nom;
11         this.sousTaches = new ArrayList<Tache>();
12         }
13
14         public void ajouter(Tache tache) {
15         this.sousTaches.add(tache);
16         }
17
18         public void supprimer(Tache tache) {
19         this.sousTaches.remove(tache);
20         }
21
22         public String getNom() {
23         return this.nom;
24         }
25
26        public int getCout() {
27        int result = 0;
28        for (Tache t : sousTaches) {
29        result += t.getCout();
30        }
31        return result;
32        }
33
34         public ArrayList<Tache> getTaches() {
35         return sousTaches ;
36         }
37
38         public void trier(){
39         Collections.sort(soustaches) ;
40         }
41
42         public void serialiser(String nomfic)throws IOException{
43         FileOutputStream fos = new FileOutputStream(nomfic);
44         ObjectOutputStream oos = new ObjectOutputStream(fos);
45         oox.writeObject(this);
46         }
47 }
```

2.2 Ecrire la classe TestTache avec le main qui effectue les traitements suivants : (**3 points**)

- Instancie une tâche complexe de nom « A »
- Instancie 2 tâches élémentaires : la première de nom « A1 » et de coût 20, la seconde de nom « A2 » et de coût 10 (0,5 pt)
- Ajoute les 2 tâches élémentaires à la tâche complexe (0,5 pt)
- Affiche le coût total de la tâche complexe et le nom et coût de ses tâches triées (1 pt)
- Sérialise la tâche complexe dans un fichier nommé « Tache.ser » et affiche un message d'erreur en cas d'exception `IOException` (1 pt)

```
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          tA.trier() ;
9          for (Tache t : tA.getTaches() {
7          System.out.println("Nom et Cout = " + t.getNom() + t.getCout());
8
9          try {
10         tA.serialiser("Tache.ser") ;
11         }
12         catch(IOException) {
13         System.out.println("Erreur de sérialisation de la tâche") ;
14         }
15
16         }
17         }
```

Exercice 3 : Interface graphique d'une tâche complexe (6 points). Durée max : 20 min

Nous définissons maintenant une interface graphique en Swing qui permet d'ajouter de nouvelles sous-tâches à une tâche complexe. Le code partiel de cette classe TacheComplexeSwing est donné au listing 2 ci-dessous.

Listing 2 – La classe TacheComplexeSwing

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4
5 public class TacheComplexeSwing extends JFrame implements ActionListener{
6
```

```
7       private TacheComplexe tache;
8       final private JTextField valeurNom = new JTextField(10);
9       final private JTextField valeurCout = new JTextField(10);
10      final private JButton Ajouter = new JButton("Ajouter");
11      final private JButton Quitter = new JButton("Quitter");
12
13
14          public TacheComplexeSwing(TacheComplexe tache) {
15          super("Nouvelle tâche");
16          this.tache = tache;
17          Container c = this.getContentPane();
18          c.setLayout(new BorderLayout());
19          JPanel informations = new JPanel(new GridLayout(2,2));
20          informations.add(new JLabel("Nom : ", SwingConstants.RIGHT));
21          informations.add(valeurNom);
22          informations.add(new JLabel("Coût : ", SwingConstants.RIGHT));
23          informations.add(valeurCout);
24          c.add(informations, BorderLayout.CENTER);
24
25          JPanel boutons = new JPanel(new FlowLayout());
26          boutons.add(Ajouter);
27          boutons.add(Quitter);
28          c.add(boutons, BorderLayout.SOUTH);
29
30
31           this.pack();
32          this.setVisible(true);
33      }
34
35       public static void main(String[] args) {
36           new TacheComplexeSwing(new TacheComplexe("Test TacheComplexeSwing"));
37       }
38  }
```

3.1 Dessiner la fenêtre (et les composants graphiques qu'elle contient) telle qu'elle est affichée quand cette classe est exécutée. **(2 points)**

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 : voir méthode `ajouter` de la question **2.1**. Le nom et le coût de cette sous-tâche sont 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. **(4 points)**

```
Quitter.addActionListener(this) ;
Ajouter.addActionListener(this) ;


public void actionPerformed(ActionEvent ev) {
try {
     Object o = ev.getSource();
     If (o == Ajouter) {
           String nom = valeurNom.getText();
           int cout = Integer.parseInt(valeurCout.getText());
           tache.ajouter(new TacheElementaire(nom, cout));
           System.out.println("cout total = " + tache.getCout());
     }
     else {
           System.out.println("Appui sur Quitter...");
         fenetre.dispose();
     }
}
catch (NumberFormatException e) {
     valeurCout.setBackground(Color.RED);
}
}
```

ANNEXE : extraits de la documentation API

java.lang

Interface Comparable<T>

Method Summary	
int	**compareTo**(T o) Compares this object with the specified object for order.

java.util

Class ArrayList<E>

Constructor Summary	
ArrayList() Constructs an empty list with an initial capacity of ten.	
ArrayList(Collection<? extends E> c) Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.	

Method Summary	
boolean	**add**(E e) Appends the specified element to the end of this list.
boolean	**remove(Object o)** Removes the first occurrence of the specified element from this list, if it is present.
E	**get(int index)** Returns the element (object) at the specified position in this list.
int	**size()** Returns the number of elements (objects) in this list.

Class Collections

Method Summary	
static <T extends Comparable<? super T>> void	sort(List<T> list) Sorts the specified list into ascending order, according to the natural ordering of its elements.

javax.swing

Class JButton

Method Summary	
void	**addActionListener**(ActionListener l) Adds an ActionListener to the button.

Class JTextField

Method Summary	
String	**getText**() Returns the text contained in this `TextComponent`.
void	**setBackground(Color bg)** Sets the background color of this component.

Class JFrame

Method Summary	
void	**dispose()** Releases all of the native screen resources used by this `Window`, its subcomponents, and all of its owned children.

java.awt S

Class Container

Method Summary	
Component	**add**(Component comp) Appends the specified component to the end of this container.

java.awt.event

Interface ActionListener

Method Summary

void	**actionPerformed**(ActionEvent e) Invoked when an action occurs.

java.io

Class FileOutputStream

Constructor Summary	
FileOutputStream(String name) Creates an output file stream to write to the file with the specified name.	

Class ObjectOutputStream

Constructor Summary	
	ObjectOutputStream(OutputStream out) Creates an ObjectOutputStream that writes to the specified OutputStream.
Method Summary	
void	**close**() Closes the output stream.
void	**writeObject**(Object obj) Write the specified object to the ObjectOutputStream.