Espace Malraux - Place Saint Clair
BP 86
14203 Hérouville Saint-Clair Cedex

le cnam Normandie

NFP121 : Programmation avancée

Enseignant : Philippe BRUTUS

Année universitaire 2020-2021

Sujet d'examen de première session

Date : lundi 21 juin 2021

Horaires : 9h - 12h (durée : 3h)

Préférences d'impression : recto-verso, couleur

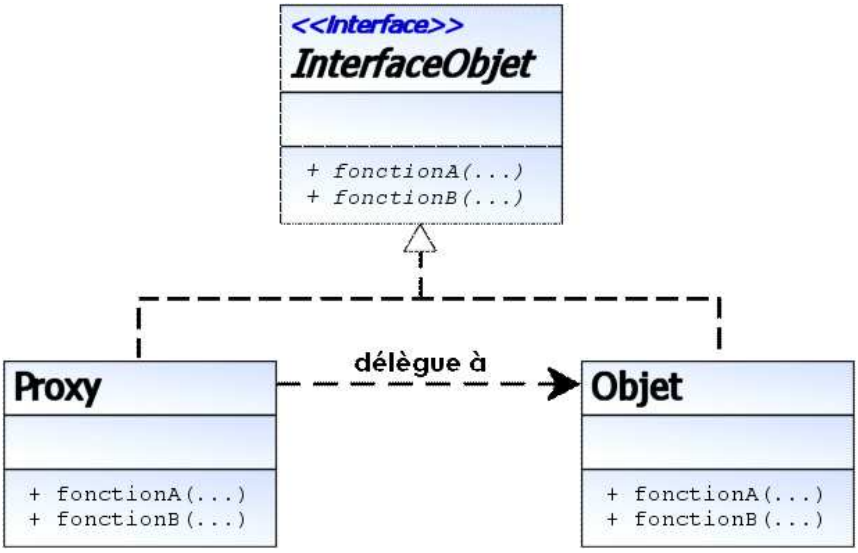
Modalités pratiques

Éléments :	**autorisés**	**non autorisés**
Supports de cours (polycopiés)		x
Documents manuscrits (prise de notes)		x
Calculatrice		x
Autres consignes / remarques :		
- Tout appareil communicant ou de stockage de données (téléphone, tablette, ordinateur …) doit être éteint et rangé. - Rendre le sujet avec la copie		
Barème de notation :		
- Exercice 1 : 7 points - Exercice 2 : 6 points - Exercice 3 : 7 points		

Exercice 1 : proxy (7 points)

Dans une application de gestion de stock, on souhaite associer une illustration graphique à chaque catégorie d'articles. Chaque image est disponible sous forme d'un fichier qu'il faut charger pour l'afficher. On souhaite éviter de charger plusieurs fois chaque image s'il y a plusieurs articles de la même catégorie et donc éviter de recharger une image à la création d'un nouvel article dans une catégorie donnée. On souhaite aussi éviter de charger l'image d'une catégorie pour rien (s'il n'y a pas d'article de cette catégorie).

Pour y arriver, on met en œuvre le patron de conception proxy.



Un proxy est un mandataire qui agit comme intermédiaire. Les opérations réalisées par mandataire sont spécifiées dans une interface qui est implantée dans la classe voulue et dans la classe mandataire associée.

Au lieu d'appeler fonctionA sur l'objet voulu, on crée un mandataire (proxy) à qui on donne procuration pour agir sur l'objet voulu et on demande au mandataire de réaliser fonctionA. Selon le cas, le mandataire appelle fonctionA sur l'objet voulu ou ne le fait pas (contrôle d'accès, traitement déjà effectué et dont il a conservé le résultat …).

On se propose de définir une interface Image et deux implantations RealImage et ProxyImage

```
interface Image {
    public void displayImage();
}
class RealImage implements Image {
    private String filename;
    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }
    private void loadImageFromDisk() {
        // Opération potentiellement coûteuse en temps et/ou espace
    }
```

```
    public void displayImage() {
        // Opération qui sera réalisée à la demande du mandataire
    }
}
class ProxyImage implements Image {
    ...
    public ProxyImage(String filename) {
        ...
    }
    public void displayImage() {
        ...
    }
}
```

a) Donner le code des variables (d'instance et/ou de classe) de la classe ProxyImage.

b) Donner le code du constructeur de la classe ProxyImage.

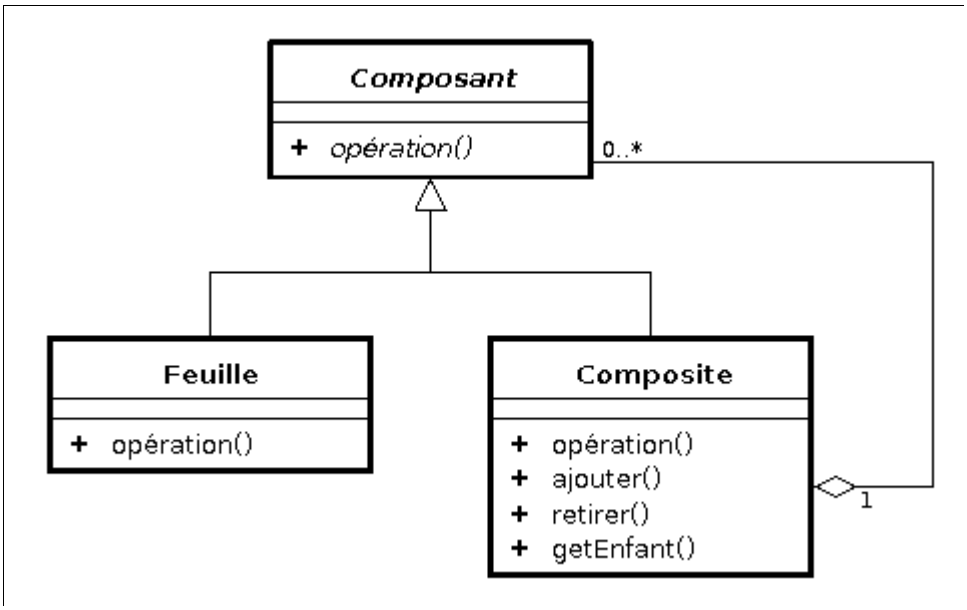
c) Donner le code de la méthode displayImage() de la classe ProxyImage qui ne charge le fichier image que si cela n'a pas déjà été fait.

d) Proposer une solution pour gérer par des proxy les images des différentes catégories.

e) Donner le code de création d'une image ainsi que le code pour afficher une image.

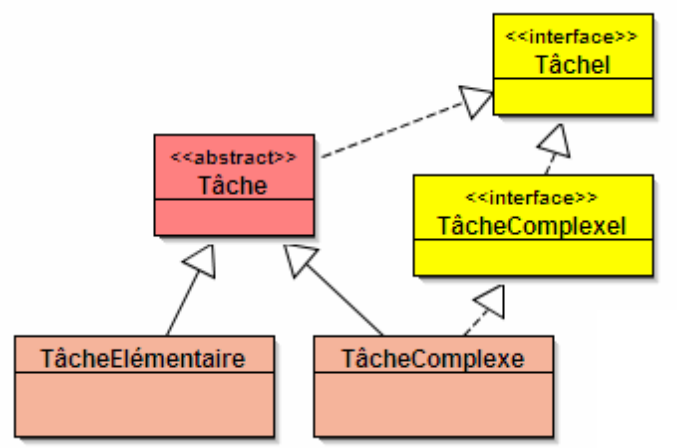
Exercice 2 : composite (6 points)

Ce patron de conception représente une structure de données récursive quelconque au moyen de 3 classes.



On souhaite écrire un programme qui exploite ce patron de conception pour représenter des tâches.

On définira à cet effet les interfaces TâcheI et TâcheComplexeI, la classe abstraite Tâche et les classes (instanciables) TâcheElémentaire et TâcheComplexe.



L'interface TâcheI sera définie comme suit :

```
public interface TâcheI{
     String nom(); // fournit le nom
     int coût(); // fournit le coût
}
```

L'interface TâcheComplexeI sera définie comme suit :

```
public interface TâcheComplexeI extends TâcheI{
     void ajouter(TâcheI tâche); // ajout d'une tâche
}
```

La classe abstraite Tâche implante une méthode et laisse la responsabilité à ses classes dérivées d'implanter le coût :

```
public abstract class Tâche implements TâcheI{
     private String nom;
     public Tâche(String nom) {
          this.nom = nom;
     }
     public String nom() {
          return this.nom;
     }
     public abstract int coût();
}
```

a) Ecrire une implantation complète de la classe TâcheElémentaire

Une exception de type RuntimeException sera levée si le coût transmis lors de la création d'une tâche élémentaire est négatif.

b) Ecrire une implantation complète de la classe TâcheComplexe

Exercice 3 : visiteur (7 points)

Une application de liste de choses à faire utilise les classes AFaire et ListeAFaire.

Pour l'enregistrement et le chargement de listes de choses à faire, cette application utilise le patron de conception « Visiteur » en définissant les classes Serialisation, Enregistrement et Chargement.

Voici le code de ces classes.

```
public abstract class Serialisation {
 abstract void opereSur(AFaire af) throws IOException;
}
public class Enregistrement extends Serialisation {
 private DataOutputStream dos;
 public Enregistrement(DataOutputStream dos) { this.dos = dos; }
 void opereSur(AFaire af) throws IOException {
       dos.writeUTF(af.getNom());
       dos.writeUTF(af.getDescription());
 }
}
public class Chargement extends Serialisation {
 private DataInputStream dis;
 public Chargement(DataInputStream dis) { this.dis = dis; }
 void opereSur(AFaire af) throws IOException {
       af.setNom(dis.readUTF());
       af.setDescription(dis.readUTF());
 }
}

public class AFaire {
 public static final int VERSION = 1;
 private String nom;
 private String description;
 public AFaire() {
       nom = null;
       description = null;
 }
 public AFaire(String nom, String description) {
       this.nom = nom;
       this.description = description;
 }
 public String getNom() { return nom; }
 public void setNom(String nom) { this.nom = nom; }
 public String getDescription() { return description; }
 public void setDescription(String description) {
       this.description = description;
 }
 public String toString() { return nom + " (" + description + ")"; }
 public void realise(Serialisation s) throws IOException {
       s.opereSur(this);
 }
}

public class ListeAFaire {
 private List<AFaire> chosesAFaire;
 public ListeAFaire() { chosesAFaire = new LinkedList<AFaire>(); }
 public void ajoute(AFaire af) { chosesAFaire.add(af); }
 public void affiche() {
       System.out.println(chosesAFaire.size() + " chose(s) à faire.");
       for (AFaire af : chosesAFaire) {
             System.out.println(af);
       }
 }
```

```
    public void enregistre(String nomFichier) throws IOException {
          DataOutputStream dos = new DataOutputStream(
                                              new FileOutputStream(nomFichier));
          Enregistrement enregistrement = new Enregistrement(dos);
          dos.writeInt(AFaire.VERSION);
          dos.writeInt(chosesAFaire.size());
          for (AFaire af : chosesAFaire) { af.realise(enregistrement); }
          dos.close();
    }
    public void charge(String nomFichier) throws IOException {
          DataInputStream dis = new DataInputStream(
                                             new FileInputStream(nomFichier));
          Chargement chargement = new Chargement(dis);
          dis.readInt(); //version ignorée ici
          int n = dis.readInt();
          for(int i = 0; i< n; i++) {
                AFaire af = new AFaire();
                af.realise(chargement);
                chosesAFaire.add(af);
          }
          dis.close();
    }
}
```

Une évolution de cette application introduit dans la classe AFaire un nouvel attribut

```
private int importance; //de 0 (négligeable) à 5 (essentiel)
```

La nouvelle version de l'application doit pouvoir enregistrer et charger des fichiers à son propre format (qui contient l'importance des choses à faire) mais aussi des fichiers de l'ancienne version.

a) Ecrire le code des nouvelles classes de visiteurs EnregistrementV2 et ChargementV2

Dans la nouvelle version de la classe ListeAFaire, la méthode enregistre prend maintenant un second argument entier qui indique le numéro de version du fichier.

b) En utilisant une fabrique dont on écrira le code, écrire le code de la nouvelle version de la méthode charge.

c) En utilisant une fabrique dont on écrira le code, écrire le code de la nouvelle version de la méthode enregistre.

java.util
Interface Collection<E>
All Superinterfaces:
Iterable<E>
All Known Subinterfaces:
..., List<E>, …
All Known Implementing Classes:
…, ArrayList, …, LinkedList, …, Vector

```
public interface Collection<E>
extends Iterable<E>
```

Method Summary

boolean	**add**(E e) Ensures that this collection contains the specified element (optional operation).
void	**clear**() Removes all of the elements from this collection (optional operation).
boolean	**contains**(Object o) Returns `true` if this collection contains the specified element.
boolean	**equals**(Object o) Compares the specified object with this collection for equality.
boolean	**isEmpty**() Returns `true` if this collection contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this collection.
boolean	**remove**(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
int	**size**() Returns the number of elements in this collection.

java.util
Interface List<E>
All Superinterfaces:
Collection<E>, Iterable<E>
All Known Implementing Classes:
…, ArrayList, …, LinkedList, …, Vector

```
public interface List<E>
extends Collection<E>
```

Method Summary

boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns true if this list contains the specified element.
boolean	**equals**(Object o) Compares the specified object with this list for equality.
E	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size**() Returns the number of elements in this list.