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

le cnam Normandie

NFP121 : Programmation avancée

Enseignant : Philippe BRUTUS

Année universitaire 2022-2023

Sujet d'examen de première session

Date : vendredi 23 juin 2023

Horaires : 9h15 - 11h15 (durée : 2h00)
Avec tiers-temps : 9h15 - 11h55 (durée : 2h40)

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.
- Lisez attentivement le sujet.
- Si vous faîtes une hypothèse par rapport au sujet, indiquez-le explicitement.

Barème de notation :

- Exercice 1 : 5 points
  a) 4 points
  b) 1 point
- Exercice 2 : 10 points
  a) 4 points
  b) 4 points
  c) 2 points
- Exercice 3 : 5 points
  a) 1,5 points
  b) 1,5 points
  c) 2 points

Une partie du code source d'une application de gestion de choses à faire (ou pense-bête) est donnée en annexes 1, 2, 3 et 4. Dans cette première version, quand une chose est terminée, on la retire du pense-bête.

On développe une version 2 dans laquelle la classe ChoseAFaire définit une variable d'instance booléenne **`fini`** pour distinguer les choses à faire des choses faites. Le code source de cette version est donné en annexe 5.

1) Itérateur

On souhaite pouvoir considérer tantôt toutes les choses du pense-bête, tantôt seulement les choses non terminées. A cet effet, on ajoute à la classe PenseBete les caractéristiques suivantes :

```
private boolean tout = true;
private class IterateurAFaire implements Iterator<ChoseAFaire> {
      ... //variables d'instance
      public IterateurAFaire(Iterator<ChoseAFaire> it) { ... }
      public boolean hasNext() { ... }
      public ChoseAFaire next() { ...    }
}
public void seulementLesChosesPasFaites() { tout = false; }
public void toutesLesChoses() { tout = true; }
```

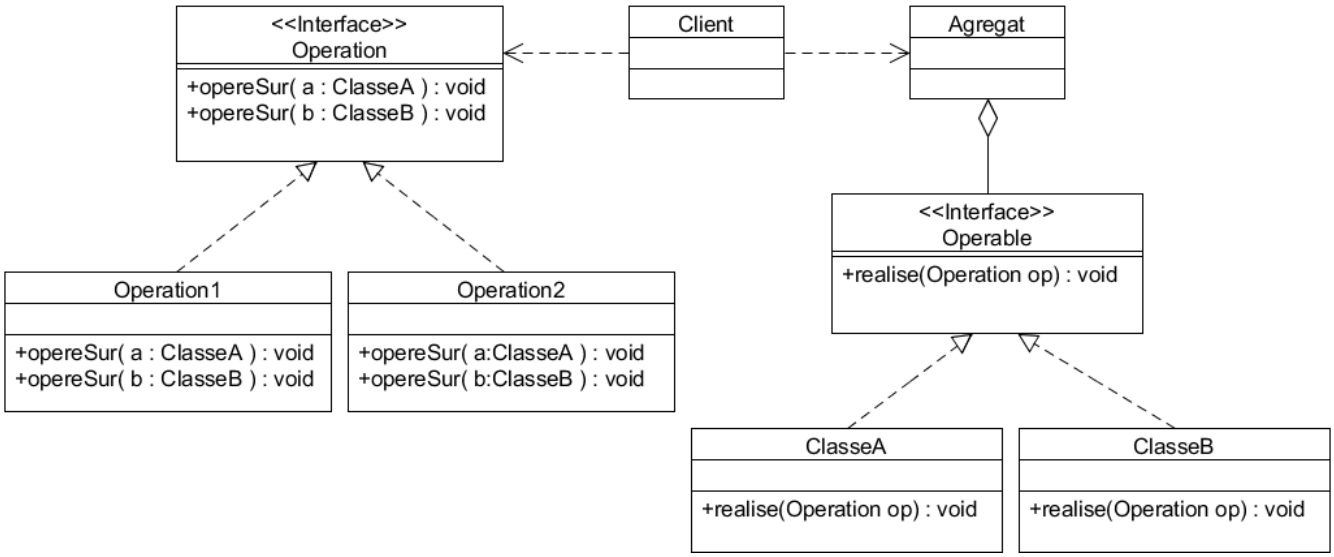
a) Donner le code complété de la classe imbriquée IterateurAFaire

b) Donner la nouvelle version de la méthode iterator de la classe PenseBete

2) Visiteur

Dans la version 2 de l'application pense-bête, on souhaite pouvoir lire les fichiers de la nouvelle version (dans laquelle un booléen complète la chaîne de caractères) mais aussi les fichiers de la version 1.

Pour ne pas surcharger la classe ChoseAFaire avec une autre version de chargeDepuis, ni d'ailleurs la classe PenseBete avec une autre version de chargeDepuis, on prend la décision de déporter cette opération dans une autre classe en utilisant le patron de conception visiteur.



Ce patron de conception permet de définir plusieurs opérations réalisables sur les éléments d'un agrégat (et éventuellement sur l'agrégat lui-même) sans les définir dans les classes des éléments (ni dans l'agrégat).

On définit à cet effet les interfaces suivantes :

```
public interface Operable {
	void realise(Operation op) throws Exception;
}
public interface Operation {
	void opereSur(PenseBete p) throws Exception;
	void opereSur(ChoseAFaire c) throws Exception;
}
```

On définit une classe ChargementV1 qui implante l'interface Operation et définit le chargement des fichiers de version 1. La variable d'instance **fini** des choses lues dans le fichier (dont la valeur est absente du fichier) sera initialisée à **false**.

On définit aussi une classe ChargementV2 qui implante l'interface Operation et définit le chargement des fichiers de version 2 et on supprime la méthode chargeDepuis des classes ChoseAFaire et PenseBete.

a) **Donner le code de la classe ChargementV1**

b) **Donner le code de la classe ChargementV2**

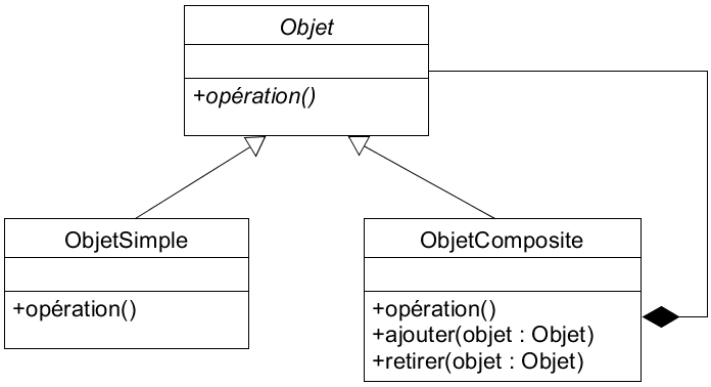
La méthode charge de la classe PenseBete doit maintenant réaliser un chargement de la version 1 ou de la version 2 en fonction du numéro de version lu au début du fichier.

c) **Donner le code de la version modifiée de la méthode charge de la classe PenseBete**

3) Composite

Dans la version 3 de l'application de gestion de tâches, on introduit une nouvelle sorte de chose à faire, ChoseAFaireComplexe, composée de choses à faire (simples ou complexes). Une chose à faire complexe est finie quand toutes les choses à faire qui la composent sont finies.

A cet effet, on met en œuvre le patron de conception Composite. Ce patron de conception représente une structure de données récursive quelconque au moyen de 3 classes.



La classe ChoseAFaire est donc modifiée comme en annexe 6.

Les classes ChoseAFaireSimple et ChoseAFaireComplexe, qui en héritent, sont définies pour compléter (voir annexes 7 et 8).

a) **Donner le code de la méthode fini() de la classe ChoseAFaireComplexe.**

b) **Donner le code de la méthode termine() de la classe ChoseAFaireComplexe.**

c) **Donner le code de la méthode toString() de la classe ChoseAFaireComplexe.**

Annexe 1 : Version 1 de la classe ChoseAFaire

```
public class ChoseAFaire {
	protected String nom;
	public ChoseAFaire() { nom = null; }
	public ChoseAFaire(String nom) { this.nom = nom; }
	public String designation() { return nom; }
	public void designePar(String nom) { this.nom = nom; }
	public String toString() { return nom; }
	public void chargeDepuis(DataInputStream s) throws IOException {
		nom = s.readUTF();
	}
	public void enregistreDans(DataOutputStream s) throws IOException {
		s.writeUTF(nom);
	}
}
```

Annexe 2 : Classe ExceptionVersionInconnue

```
public class ExceptionVersionInconnue extends Exception {
	public ExceptionVersionInconnue(byte version) {
		super("La version " + version + " des fichiers n'est pas reconnue !");
	}
}
```

Annexe 3 : Enumération OuverturePenseBete

```
public enum OuverturePenseBete {
	REUSSIE,
	VERSION_INCONNUE,
	PROBLEME_PENDANT_LECTURE
}
```

Annexe 4 : Version 1 de la classe PenseBete

```
public class PenseBete implements Iterable<ChoseAFaire> {
	private List<ChoseAFaire> contenu;
	public PenseBete() {
		contenu = new LinkedList<ChoseAFaire>();
	}
	public boolean estVide() {
		return contenu.isEmpty();
	}
	public void vide() {
		contenu.clear();
	}
	public void ajoute(ChoseAFaire c) {
		contenu.add(c);
	}
	public void retire(int i) {
		contenu.remove(i);
	}
	public void retire(String nom) {
		Iterator<ChoseAFaire> it = contenu.iterator();
		while(it.hasNext()) {
			ChoseAFaire c = it.next();
			if (nom.equals(c.designation())) {
				it.remove(); break;
			}
		}
	}
	public Iterator<ChoseAFaire> iterator() {
		return contenu.iterator();
	}
```

```
	public boolean enregistre(String nomFichier) {
		try (DataOutputStream s = new DataOutputStream(
						new FileOutputStream(
							new File(nomFichier)))) {
			s.writeByte(1);
			enregistreDans(s);
			return true;
		} catch(IOException eAutre) {
			return false;
		}
	}
	private void enregistreDans(DataOutputStream s) throws IOException {
		s.writeInt(contenu.size());
		for(ChoseAFaire c:contenu) {
			c.enregistreDans(s);
		}
	}
	public OuverturePenseBete charge(String nomFichier) {
		try (DataInputStream s = new DataInputStream(
						new FileInputStream(
							new File(nomFichier)))) {
			vide();
			byte version = s.readByte();
			if (version == 1) {
				chargeDepuis(s);
			} else {
				throw new ExceptionVersionInconnue(version);
			}
		} catch(ExceptionVersionInconnue eVersion) {
			return OuverturePenseBete.VERSION_INCONNUE;
		} catch(IOException eAutre) {
			return OuverturePenseBete.PROBLEME_PENDANT_LECTURE;
		}
		return OuverturePenseBete.REUSSIE;
	}
	private void chargeDepuis(DataInputStream s) throws IOException  {
		int n = s.readInt();
		for(int i = 0; i < n; i++) {
			ChoseAFaire c = new ChoseAFaire();
			c.chargeDepuis(s);
			contenu.add(c);
		}
	}
}
```

Annexe 5 : Version 2 de la classe ChoseAFaire

```
public class ChoseAFaire {
	protected String nom;
	protected boolean fini;
	public ChoseAFaire() { nom = null; fini = false; }
	public ChoseAFaire(String nom) { this.nom = nom; fini = false; }
	public String designation() { return nom; }
	public void designePar(String nom) { this.nom = nom; }
	public boolean fini() { return fini; }
	public void termine() { fini = true; }
	public String toString() {
		String texte = (fini) ? "X " : "  ";
		texte += nom;
		return texte;
	}
```

```
	public void chargeDepuis(DataInputStream s) throws IOException {
		nom = s.readUTF();
		fini = s.readBoolean();
	}
	public void enregistreDans(DataOutputStream s) throws IOException {
		s.writeUTF(nom);
		s.writeBoolean(fini);
	}
}
```

Annexe 6 : Version 3 de la classe ChoseAFaire

```
public abstract class ChoseAFaire implements Operable {
	protected String nom;
	public ChoseAFaire() { nom = null; }
	public ChoseAFaire(String nom) { this.nom = nom; }
	public String designation() { return nom; }
	public void designePar(String nom) { this.nom = nom; }
	public void realise(Operation op) { op.opereSur(this); }
	public abstract boolean fini();
	public abstract void termine();
	public abstract String toString();
}
```

Annexe 7 : Classe ChoseAFaireSimple

```
public class ChoseAFaireSimple extends ChoseAFaire {
	protected boolean fini;
	public ChoseAFaireSimple() {
		super();
		fini = false;
	}
	public ChoseAFaireSimple(String nom) {
		super(nom);
		fini = false;
	}
	public void realise(Operation op) { op.opereSur(this); }
	public boolean fini() {
		return fini;
	}
	public void termine() {
		fini = true;
	}
	public String toString() {
		String texte = (fini) ? "X " : "  ";
		texte += nom;
		return texte;
	}
}
```

Annexe 8 : Classe ChoseAFaireComplexe

```
public class ChoseAFaireComplexe extends ChoseAFaire {
	protected List<ChoseAFaire> contenu;
	public ChoseAFaireComplexe() {
		super(); contenu = new LinkedList<ChoseAFaire>();
	}
	public ChoseAFaireComplexe(String nom) {
		super(nom); contenu = new LinkedList<ChoseAFaire>();
	}
	public boolean estVide() { return contenu.isEmpty(); }
	public void vide() { contenu.clear(); }
	public void ajoute(ChoseAFaire c) { contenu.add(c); }
	public void retire(int i) { contenu.remove(i); }
```

```
	public void retire(String nom) {
		Iterator<ChoseAFaire> it = contenu.iterator();
		while(it.hasNext()) {
			ChoseAFaire c = it.next();
			if (nom.equals(c.designation())) { it.remove(); }
		}
	}
	public void realise(Operation op) { op.opereSur(this); }
	public boolean fini() { ... }
	public void termine() { ... }
	public String toString() { ... }
}
```

Annexe 9 : Quelques API de java

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 Iterator<E>

public interface **Iterator<E>**

An iterator over a collection.
This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

Collection, ListIterator, Enumeration

Method Summary	
boolean	**hasNext**() Returns true if the iteration has more elements.
E	**next**() Returns the next element in the iteration.
void	**remove**() Removes from the underlying collection the last element returned by the iterator (optional operation).

java.util
Interface Iterable<T>

All Known Subinterfaces:

… Collection<E>, … List<E>, … Set<E>, SortedSet<E>

All Known Implementing Classes:

AbstractCollection, AbstractList, … AbstractSet, … ArrayList, … LinkedList, … TreeSet, Vector

public interface **Iterable<T>**

Implementing this interface allows an object to be the target of the "foreach" statement.

Since:

1.5

Method Summary	
Iterator<T>	**iterator**() Returns an iterator over a set of elements of type T.

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.