LI314 - Programmation Orientée Objet en Java : Examen Juin 2014

Université Pierre et Marie Curie [1] **.- 2 heures**

ASCII art : des images de caractère !

Les images ascii sont une forme d'art apparu avec les premiers ordinateurs, dont l'écran n'était capable d'afficher que des caractères. Comme le montre l'exemple en annexe, ces "images" sont cependant parfois assez convaincantes. Nous allons développer dans ce sujet un package **pobj.asciiart** permettant leur manipulation.

1. Image est une abstraction !

Soit l'interface `Image`, décrite sur le diagramme UML suivant. La méthode `affiche` se contente d'afficher l'image sur la sortie standard `System.out`, `nbLignes` et `nbColonnes` donnent respectivement le nombre de lignes et de colonnes de l'image, supposée rectangulaire, la `taille` de l'image est définie comme le nombre de caractères qui la compose.

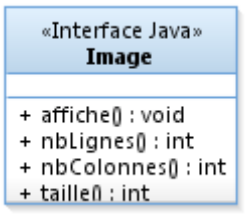

Figure 1: Diagramme UML de l'interface Image.

⇒ a) Donnez le code Java correspondant à ce diagramme.

Image.java

```java
package pobj.asciiart;


public interface Image extends Comparable<Image> {
      void affiche(); // affiche l'image
      int nbLignes(); // retourne le nombre de lignes composant l'image
      int nbColonnes(); // retourne le nombre de colonnes composant l'image
      int taille();
}
```

Barème :

20% public interface, -10% si faute majuscule sur Interface

20%*4 par méthode correcte (nommage, signature). Ne donner que 10% si la méthode a un corps, même vide.

[1] Cet énoncé est librement inspiré d'un sujet de Denis Poitrenaud Univ. Paris 5 Descartes

On souhaite de plus pouvoir comparer et trier des images par taille croissante, comme dans l'extrait de code suivant.

```
List<Image> images = new ArrayList<>();
// ...
Collections.sort(images);
```

⇒ b) Quelle modification faut-il apporter à Image pour rendre ce comportement possible ? Ajoutez les déclarations utiles au code précédent.

Il faut ajouter à l'interface précédente la déclaration "extends Comparable<Image>".
Barème :
0 ou 100%, les autres solutions ne fonctionnant pas complétement.

⇒ c) Quel sera l'impact des déclarations de 1.b) sur une classe qui implémente `Image` ? Donnez plus précisément le code qu'il faut ajouter dans une de ces classes.

Il faut ajouter à toute classe concrète une opération

```
public int compareTo (Image other) {
   return Integer.compare(taille(),other.taille());
}
```

Barème :
50% la signature,
50% un code correct qui rend -1,0 ou 1 selon les cas.

NB : Pour éviter de trop lier les questions, dans la suite de l'énoncé on ignorera ces problèmes de comparaison entre images, on reste donc sur l'interface telle que décrite par le diagramme ci-dessus.

2. Image Brutale

⇒ Écrivez une classe `ImageBrute` pour laquelle les caractères composant l'image sont stockés dans un tableau à 2 dimensions nommé `pixels[][]`. La classe devra comporter un constructeur produisant une image vide (i.e. remplie d'espaces) dont les dimensions (lignes, colonnes) sont spécifiées par les paramètres. On donnera aussi un constructeur prenant directement un `char[][] pixels` en argument et qui utilise les données fournies telles quelles (utilisé en question 3, ligne 20). On implémentera toutes les opérations identifiées en question 1 (sauf la comparaison).

ImageBrute.java

```
package pobj.asciiart;

import java.util.Arrays;

public class ImageBrute implements Image {
      private char[][] pixels;

      public ImageBrute(int nbLignes, int nbColonnes) {
```

```
            pixels = new char[nbLignes][nbColonnes];
            for (char[] ligne : pixels)
                  Arrays.fill(ligne, ' ');
      }

      public ImageBrute(char[][] pixels) {
            this.pixels = pixels;
      }

      @Override
      public void affiche() {
            for (char[] ligne : pixels) {
                  for (char c : ligne)
                        System.out.print(c);
                  System.out.println();
            }
      }

      @Override
      public int nbLignes() {
            return pixels.length;
      }

      @Override
      public int nbColonnes() {
            return pixels[0].length;
      }

      @Override
      public int taille() {
            return nbLignes() * nbColonnes();
      }

      @Override
      public int compareTo(Image o) {
            return Integer.compare(taille(), o.taille());
      }
}
```

Barème :

- 10% implemente Image
- 10% attribut tableau de char
- 20% constructeur image vide (dont 10 pour le new correct avec une syntaxe ok)
- 10% constructeur par aliasing (on accepte aussi s'il est fait par copie)
- 20% affiche avec les newline aux bons endroits
- 10% *3 par autre opération de Image (nblig, nbcol, taille)

3. Chargement Exceptionnel

On se propose d'introduire une classe utilitaire `ChargeurImage` permettant d'initialiser une image à partir d'un fichier texte. Une première version de cette classe est fournie ci-dessous.

```
public class ChargeurImage {
      public static ImageBrute charge(String nom) {
            Scanner sc;
            // erreur si le fichier n'existe pas
```

```
            sc = new Scanner(new File(nom));

            LinkedList<char[]> lignes = new LinkedList<char[]>();
            while (sc.hasNextLine()) {
                  lignes.add(sc.nextLine().toCharArray());
                  if (lignes.getFirst().length != lignes.getLast().length) {
                        sc.close();
                        // erreur : lignes de taille différente
                  }
            }
            sc.close();
            if (lignes.isEmpty()) {
                  // erreur : fichier vide
            }
            char [][] pixels = lignes.toArray(new char[0][0]);
            return new ImageBrute(pixels);
      }
}
```

⇒ a) Définissez une classe d'exception `ImageException`, et pour chaque erreur possible (fichier introuvable `BadImageFileName`, lignes de tailles différentes `BadImageLineSize` ou fichier vide `EmptyImageFile`) une `ImageException` particulière.

ImageException.java

```
package pobj.asciiart;

public class ImageException extends Exception {
}

class BadImageFileName extends ImageException {
}

class BadImageLineSize extends ImageException {
}

class EmptyImageFile extends ImageException {
}
```

Barème : 40% définir une classe mère dérivant de Exception, 3*20% classes dérivées.

⇒ b) Modifiez le chargeur proposé pour que les différentes erreurs identifiées conduisent à la levée de l'exception appropriée. L'exception levée sera différente pour chacun de ces cas.
Nous rappelons que l'invocation de

```
new Scanner(new File(nom))
```

lève une exception de type `FileNotFoundException` si le fichier `nom` n'existe pas.

ImageBruteEx.java

```
package pobj.asciiart;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.LinkedList;
```

```java
import java.util.Scanner;

public class ChargeurImage {

	public static ImageBrute charge(String nom) throws ImageException {
		Scanner sc;
		// erreur si le fichier n'existe pas
		try {
			sc = new Scanner(new File(nom));
		} catch (FileNotFoundException e) {
			throw new BadImageFileName();
		}

		LinkedList<char[]> lignes = new LinkedList<char[]>();
		while (sc.hasNextLine()) {
			lignes.add(sc.nextLine().toCharArray());
			if (lignes.getFirst().length != lignes.getLast().length) {
				sc.close();
				// erreur : lignes de taille differentes
				throw new BadImageLineSize();
			}
		}
		sc.close();
		if (lignes.isEmpty()) {
			// erreur : fichier vide
			throw new EmptyImageFile();
		}
		char[][] pixels = lignes.toArray(new char[0][0]);
		return new ImageBrute(pixels);
	}

	public static void main(String[] args) {
		try {
			Image img = ChargeurImage.charge("joconde.txt");
			img.affiche();
		} catch (ImageException e) {
			e.printStackTrace();
		}
	}

}
```

Barème :

- 20% déclaration throws
- 20% try catch sur Scanner
- 20% *3 par levée d'exception

⇒ c) Ecrivez un `main` qui instancie une image en chargeant le fichier "joconde.txt" et l'affiche.

Pour tester leur maitrise du try/catch.
Le main du corrigé est dans ChargeurBrut ci-dessus.
Barème :

- 20% déclaration main correcte (public static void et arguments String[])
- 30% invocation au chargeur dans try
- 30% catch avec un corps non vide, printStackTrace ou syserr OK.

- 20% affiche

NB : Pour éviter de trop lier les questions, dans la suite de cet énoncé on ne traitera pas la levée potentielle d'exceptions par la classe `ChargeurImage`.

4. Image voit Double

On souhaite construire des `ImageDouble` à partir de deux `Image` existantes de *même largeur*, à l'affichage la première étant située au dessus de la seconde.

⇒ a) Quel *design pattern* utiliser pour traiter ce problème ?

DP Composite.

Barème : 0 ou 100.

⇒ b) Programmez une classe `ImageDouble`. Bien entendu, faites en sorte que tout objet de cette classe soit une image. On contrôlera à l'aide d'une assertion que les deux images passées an argument au constructeur sont bien de même largeur.

ImageDouble.java

```java
package pobj.asciiart;


public class ImageDouble implements Image {
	private Image im1, im2;

	public ImageDouble(Image im1, Image im2) {
		assert(im1.nbColonnes() == im2.nbColonnes());
		this.im1 = im1;
		this.im2 = im2;
	}

	@Override
	public void affiche() {
		im1.affiche();
		im2.affiche();
	}

	@Override
	public int nbLignes() {
		return im1.nbLignes() + im2.nbLignes();
	}

	@Override
	public int nbColonnes() {
		return im1.nbColonnes();
	}

	@Override
	public int taille() {
		return im1.taille() + im2.taille();
	}

	@Override
```

```
	public int compareTo(Image o) {
		return Integer.compare(taille(), o.taille());
	}
}
```

Barème :

- 10% implemente Image
- 10% attributs typés Image
- 20% constructeur (dont 10 pour l'assertion)
- 15% * 4 par méthode de Image (nblig, nbcol, taille, affiche)

5. Image Titrée

⇒ Dans le cadre du DP Decorator, programmez une classe `ImageTitre` représentant une image munie d'un titre, i.e. une chaîne de caractères de longueur inférieure ou égale à la largeur de l'image. On testera cette propriété dans le constructeur à l'aide d'une assertion. Cette ligne de titre sera affichée au dessus de l'image à décorer. On demande que la réalisation se conforme strictement au pattern, i.e. avec une classe de décoration abstraite munie d'un attribut Image déclaré **private**. La ligne de titre contribue au nombre de lignes et à la taille de l'image.

ImageDecorator.java

```
package pobj.asciiart;

public abstract class ImageDecorator implements Image {

	private Image aDecorer;

	public ImageDecorator (Image aDecorer) {
		this.aDecorer = aDecorer;
	}

	@Override
	public void affiche() {
		aDecorer.affiche();
	}

	@Override
	public int compareTo(Image o) {
		return aDecorer.compareTo(o);
	}

	@Override
	public int nbLignes() {
		return aDecorer.nbLignes();
	}

	@Override
	public int nbColonnes() {
		return aDecorer.nbColonnes();
	}
```

```
	@Override
	public int taille() {
		return aDecorer.taille();
	}

}
```

ImageTitre.java

```
package pobj.asciiart;

public class ImageTitre extends ImageDecorator {

	private String titre;

	public ImageTitre(Image adeco, String titre) {
		super(adeco);
		assert(titre.length() <= adeco.nbColonnes());
		this.titre = titre;
	}

	@Override
	public int nbLignes() {
		return super.nbLignes() + 1;
	}

	@Override
	public int taille() {
		// On peut aussi accepter : super.taille() + titre.length()
		return super.taille() + super.nbColonnes();
	}

	@Override
	public void affiche() {
		System.out.println(titre);
		super.affiche();
	}
}
```

Barème :

- Sur 50% le décorateur abstrait, 10% implémente, 10% attribut, 10% constructeur, 20% les 4 délégations
- Sur 50% le décorateur concret, 10% extends Deco, 10% constructeur, 10% par autre opération (nbLig, taille, affiche).

6. Cercle en Ascii

La classe AsciiCercle est une classe sachant dessiner un cercle (ou plus précisément un disque) en ascii. On donne une partie de la classe ici :

```
public final class AsciiCercle {

	private double ox,oy;
	private double rayon;
```

```
      public AsciiCercle(double ox, double oy, double rayon) {
            ox = ox;
            oy = oy;
            rayon = rayon;
      }

      public void dessine (int largeur) {
            for (int line=0; line <= largeur ; line ++) {
                  for (int car=0; car <= largeur; car++) {
                        if (estDansCercle(line,car)) {
                              System.out.print("o");
                        } else {
                              System.out.print(" ");
                        }
                  }
                  System.out.println();
            }
      }
}
```

⇒ a) Le code du constructeur comporte des erreurs, indiquez lesquelles.

⇒ b) Par déduction quelle est la signature de `estDansCercle` ?

⇒ c) Quelle est sa visibilité et dans quelle classe placer cette méthode ?

⇒ d) Implémentez cette opération qui indique si la distance qui sépare la position donnée en argument du centre du cercle (ox,oy) est inférieure ou égale au rayon. On utilisera Pythagore ($\sqrt{(x-x_o)^2+(y-y_o)^2}$) pour mesurer les distances.

Barème : une seule note sur 100 à saisir:

- 20% a) les this manquent à gauche
- 20% b) on acceptera int ou double pour les arguments, retour boolean.
- 20% c) private a priori, dans la classe elle meme.
- 40% d) Math.sqrt(...Math.pow(x-ox,2)....

7. Cercle en Image

⇒ a) Quel DP utiliser pour permettre de considérer une instance de `AsciiCercle` comme une `Image` ? On suppose qu'on ne peut pas modifier la classe AsciiCercle.

DP Adapter

Barème : 0 ou 100

⇒ b) Implémentez le DP choisi, dans une nouvelle classe `CercleImage`. On écrira un constructeur à quatre arguments définissant la position du centre (ox,oy), le rayon, et la largeur (qui est aussi la hauteur) de l'image.

ImageTitre.java

```
package pobj.asciiart;

public class CercleImage implements Image {

```

```
	private AsciiCercle cercle;
	private int largeur;

	public CercleImage(double ox, double oy, double rayon, int largeur) {
		this.cercle = new AsciiCercle(ox, oy, rayon);
		this.largeur = largeur;
	}

	@Override
	public int compareTo(Image o) {
		return Integer.compare(taille(), o.taille());
	}

	@Override
	public void affiche() {
		cercle.dessine(largeur);
	}

	@Override
	public int nbLignes() {
		return largeur;
	}

	@Override
	public int nbColonnes() {
		return largeur;
	}

	@Override
	public int taille() {
		return largeur*largeur;
	}
}
```

Barème :

- 10% implemente Image
- 20% attributs Cercle et largeur
- 20% constructeur avec le new Cercle
- 20% affiche
- 10%*3 par méthode de Image (nblig, nbcol, taille)

8. Chargement paresseux

Dans le cadre d'une application manipulant de nombreuses images de grande taille, il a été observé que les images de type ImageBrute sont régulièrement chargées (via le constructeur de la question 3) bien avant d'être employées. On souhaite à la place ne charger réellement l'image que *lors de la première utilisation effective* (i.e. lors du premier appel à une méthode). Cette optimisation doit être transparente, le client ne sait pas s'il manipule une Image à chargement retardé ou non.

⇒ a) A quel design pattern correspond ce comportement ?

DP Proxy, plus précisément Virtual Proxy.

Barème : 0 ou 100 dès qu'on cite Proxy.

⇒ b) Créez une classe d'image nommée `ImageRetardee` réalisant le DP choisi. Pour ce faire, vous vous reposerez sur le mécanisme de délégation. La taille d'une image retardée est 0 tant que l'image n'a pas réellement été chargée. On pourra au besoin créer un accesseur "intelligent" `getImageBrute()` qui teste si l'image est créée ou non.

ImageRetardee.java

```java
package pobj.asciiart;

public class ImageRetardee implements Image {

	private Image image = null;
	private String filename;

	public ImageRetardee (String filename) {
		this.filename = filename;
	}

	private Image getImage() {
		if (image == null) {
			try {
				image = ChargeurImage.charge(filename);
			} catch (ImageException e) {
				image = new ImageBrute(10,20);
				e.printStackTrace();
			}
		}
		return image;
	}

	public void affiche() {
		getImage().affiche();
	}


	public int compareTo(Image arg0) {
		return getImage().compareTo(arg0);
	}

	public int nbLignes() {
		return getImage().nbLignes();
	}

	public int nbColonnes() {
		return getImage().nbColonnes();
	}

	@Override
	public int taille() {
		if (image == null)
			return 0;
		return getImage().taille();
	}


}
```

NB: il s'agit là dune réalisation très standard du DP "Virtual Proxy" vu en cours mais ni en TD, ni en TME.
Barème :

- 10% implemente Image
- 20% attributs Image et filename
- 20% getImage qui fait le test, (sans les exceptions, pas demandé)
- 20% taille qui rend 0 si besoin
- 10%*3 par méthode de Image correctement protégés par getImage() (nblig, nbcol, taille)

9. Programme Principal

⇒ Ecrivez un programme principal qui crée une image double constituée d'une occurrence à chargement retardé de la "joconde.txt" donnée en annexe, et d'une occurrence d'un `CercleImage` (la largeur est 64 pour être cohérent avec l'image Joconde). L'image double elle même portera le titre "Joconde Ronde". Affichez l'image résultante.

ImageMain.java

```
package pobj.asciiart;

public class ImageMain {

	public static void main(String[] args) {
		Image j1 = new ImageRetardee("joconde.txt");
		Image j2 = new CercleImage(12,10,3,64);
		Image doublej = new ImageDouble(j1, j2);
		Image res = new ImageTitre(doublej, "Joconde ronde");

		res.affiche();
	}

}
```

Barème : 20% par instruction