Université Paris 7 — Java
Licence Informatique — Année 2005-2006

TD n°3 - Correction

Interfaces, polymorphisme

Exercice 1
On veut construire un programme capable d'afficher et de déplacer des figures géométriques colorées dans un repère en deux dimensions.
Un graphique est un ensemble de figures affichables et déplaçables par translation. On veut pouvoir afficher des segments, des triangles, et des rectangles. Ces trois éléments graphiques sont définis à l'aide de points.
Après une première analyse, on décide de modéliser l'application de la manière suivante :
- une classe `Point` représentera les coordonnées qui serviront à créer des figures
- une classe `Segment` sera définie par deux points et une couleur, codée par un entier positif.
- une classe `Triangle` sera définie par trois points et une couleur, codée par un entier positif.
- une classe `Rectangle` sera définie par quatre points et une couleur, codée par un entier positif.
- la classe `Graphique` est la classe principale et permettra d'afficher un ensemble d'objets graphiques.
  Comme nous pensons devoir ajouter ultérieurement de nouveaux types de figures géométriques, nous décidons d'utiliser des interfaces pour décrire leurs comportements communs.

1. Après avoir étudié les différences et les points communs entre les quatre premières classes que nous avons distinguées, décrivez la ou les interfaces utiles et précisez les classes qui les implémenteront.
   Correction :

```
//fichier Drawable.java
public interface Drawable{
    // choisis une couleur
    void setColor( int color)  ;
    // retourner la couleur
    int getColor() ;
    // dessiner
    void draw() ;
}
//fichier Moveable.java
public interface Moveable{
    void translate( int dx , int dy ) ;
}
```

   La classe `Point` implementera l'interface `Moveable`. Les classes `Segment`, `Triangle` et `Rectangle` implémenteront les interfaces `Moveable` et `Drawable`.

2. Décrire la méthode `static` de la classe `Graphique` qui permet d'afficher un tableau de figures géométriques (ce qui permettra de bien nous convaincre que la réponse à la question précédente est un choix judicieux).
   Correction :

```
public class Graphique{
    public static void drawGraph( Drawable[] graph ){
        for( int i = 0 ; i < graph.length ; i++ ){
            ( graph[i] ).draw() ;
        }
    }
    public static void main( String[] argv ){
        Point p1 = new Point( 0 , 0 ) ;
        Point p2 = new Point( 0 , 2 ) ;
        Point p3 = new Point( 1 , 1 ) ;
        Point p4 = new Point( 3 , 3 ) ;
        Triangle t = new Triangle( p1 , p2 , p3 , 0 ) ;
        Segment s = new Segment( p3 , p4 , 1 ) ;
        Drawable [] graph = { t , s } ;

        drawGraph(graph) ;
    }
}
```

3. Vous avez déjà étudié une classe `Point`. Sans écrire toutes les méthodes que vous utiliserez pour celle-ci, donnez les constructeurs que vous utiliserez le plus vraisemblablement dans le reste du programme, la méthode `toString` et le squelette de cette classe.
   Correction :

```
public class Point implements Moveable{
    private int x ;
    private int y ;

    public Point( int x , int y ){
        this.x = x ;
        this.y = y ;
    }
    public Point( Point p ){
        this.x = p.x ;
        this.y = p.y ;
    }
    public int getX(){
        return x ;
    }
    public int getY(){
        return y ;
    }
    public void setX( int x ){
        this.x = x ;
    }
    public void setY( int y ){
        this.y = y ;
    }
    public String toString(){
        return( "(" + x + "," + y + ")" ) ;
    }
    public void translate( int dx , int dy ){
        x = x + dx ;
        y = y + dy ;
    }
```

```
}
```

4. Donnez le constructeur de la classe `Segment`, et la méthode qui effectuera une translation.
   Correction :

```
public class Segment implements Drawable , Moveable{
    private Point beginning ;
    private Point ending ;
    private int aColor ;

    Segment( Point beginning , Point ending , int aColor){
        this.beginning = new Point( beginning ) ;
        this.ending = new Point( ending ) ;
        this.aColor = aColor ;
    }
    public int getColor(){
        return aColor ;
    }
    public void setColor(int aColor){
        this.aColor = aColor ;
    }
    public void translate ( int dx , int dy ){
        beginning.translate ( dx , dy ) ;
        ending.translate ( dx , dy ) ;
    }
    public void draw(){
        System.out.println( aColor + " debut = " + beginning + " fin = " + ending ) ;
    }
}
```

   Attention : le constructeur doit recopier le point en utilisant le constructeur par copie de la classe `Point` ! En effet, un point est modifiable par translation et peut servir à la création d'autres figures elles mêmes translatables.

5. Décrire entièrement la classe `Triangle`.
   Correction :

```
public class Triangle implements Drawable , Moveable{
    private Segment segment1 ;
    private Segment segment2 ;
    private Segment segment3 ;
    private int aColor ;

    Triangle( Point p1 , Point p2 , Point p3 , int aColor ){
        this.segment1 = new Segment( p1 , p2 , aColor ) ;
        this.segment2 = new Segment( p2 , p3 , aColor ) ;
        this.segment3 = new Segment( p3 , p1 , aColor ) ;
        this.aColor = aColor ;
    }
    public void setColor( int color){
        this.aColor = aColor ;
    }
    public int getColor(){
        return aColor ;
    }
    public void draw(){
```

```
        segment1.draw() ;
        segment2.draw() ;
        segment3.draw() ;
    }
    public void translate( int dx , int dy ){
        segment1.translate( dx , dy ) ;
        segment2.translate( dx , dy ) ;
        segment3.translate( dx , dy ) ;
    }
}
```

Exercice 2

Le but du TD sera de réaliser un formateur de texte fonctionnant sur le même principe que le formateur de texte `fmt` sous Unix.

Un objet, le parseur, prend un texte, dont il lit les mots un à un. Un deuxième objet, le formateur, accumule ces mots en listes de lignes dont chacune contient des mots séparés par des espaces. Le formateur imprime ensuite la liste de ces lignes. Ainsi, le programme supprime les espaces redondants entre deux mots, les lignes vides redondantes entre deux paragraphes, ou les passages à la ligne non nécessaires. En option, le formateur est capable de justifier du texte, mais nous n'étudierons pas cette fonctionalité.

On modélise le problème de la façon suivante : on introduit le concept de Boîte qui représente les objets composant le texte formaté.

- une *Boîte* est un élément du texte formaté qui a une taille et peut être affiché.
- une *Boîte espace* est un élément du texte formaté qui séparera les mots. Ce sera un seul espace dans le cas ou le texte n'est pas justifié, et potentiellement plusieurs espaces sinon.
- une *Boîte mot* est un élément du texte formaté qui représentera un mot.
- une *Boîte composite* représentera une ligne de texte.
- le *Formateur* utilise le *Parseur* et les objets précédemment décrits pour construire le texte formaté.

La classe `Parseur` vous est donnée. Cette classe a un constructeur public `Parseur(String filename)`, qui construit un parseur qui lit à partir du fichier nommé, et une méthode publique `suivant()` de type `String`, qui retourne le mot suivant, la chaîne vide à la fin d'un paragraphe, et l'objet `null` à la fin du fichier.

Les objets implémentant l'interface `Boite` représenteront une unité de texte d'au plus une ligne, soit un mot, un espace, ou une ligne. Trois classes implémenteront cette interface :

- `BoiteEspace` : un objet de cette classe représente un espace ;
- `BoiteMot` : un objet de cette classe représente un mot ;
- `BoiteComposite` : un objet de cette classe représente une suite horizontale de boîtes ; nous nous en servirons pour représenter les lignes.

1. Décrivez la déclaration de l'interface `Boite`, qui contient les déclarations de deux méthodes publiques : `largeur()`, de type entier, `toString()`, de type String.
   Correction :

```
public interface Boite
{
    public int largeur();
    public String toString();
}
```

2. Décrivez ensuite les définitions de deux classes qui implémentent cette interface : `BoiteEspace` et `BoiteMot`. Une `BoiteEspace` a une longueur de 1, et se convertit en la chaîne réduite à un espace `" "` (à ne pas confondre avec la chaîne vide !). Une `BoiteMot` représente une chaîne arbitraire, sa méthode `toString` retourne cette chaîne, et la méthode `largeur()` retourne sa longueur.
Correction :

```
public class BoiteEspace implements Boite{
    public BoiteEspace(){
        ;
    }
    public int largeur(){
        return 1;
    }

    public String toString(){
        return " ";
    }
}
public class BoiteMot implements Boite{
    String mot;

    public BoiteMot(String contenu){
        mot = contenu;
    }
    public int largeur(){
        return mot.length();
    }
    public String toString(){
        return mot;
    }
}
```

3. Ecrivez maintenant une nouvelle classe `BoiteComposite` qui implémente l'interface `Boite`. Une boîte composite contient une suite de boîtes ; sa largeur est la somme des largeurs des boîtes qu'elle contient, et sa représentation sous forme de chaîne est la concaténation des représentations des boîtes qu'elle contient.

   Remarque : Les objets de la classe `BoiteComposite` *stockent des objets qui sont soit des mots, soit des espaces, soit eux-mêmes des boîtes composites. On réfléchira soigneusement au type de structure qu'il faudra utiliser.*

   En plus des méthodes de l'interface `Boite`, la classe `BoiteComposite` implémente une méthode publique `vide` qui détermine si une boîte composite est vide, et une méthode publique `accumule` qui ajoute une boîte à la fin d'une boîte composite. Il sera peut-être judicieux d'ajouter des méthodes auxiliaires privées.
   Correction :

```
public class BoiteComposite implements Boite{
    Boite boites[];
    int num;

    public BoiteComposite(){
        boites = new Boite[10];
```

```
        num = 0;
    }
    public int largeur(){
        int i;
        int l;
        l = 0;
        for(i = 0; i < num; i++)
            l = l + boites[i].largeur();
        return l;
    }
    public boolean vide(){
        return num == 0;
    }
    private void redimensionne(int n){
        Boite[] nboites;
        int i;
        nboites = new Boite[n];
        for(i = 0; i < num; i++) {
            nboites[i] = boites[i];
        }

        boites = nboites;
    }
    public void accumule(Boite b){
        if(num >= boites.length)
            redimensionne(2 * num);

        boites[num] = b;
        num = num + 1;
    }
    public String toString(){
        int i;
        StringBuffer s;
        s = new StringBuffer();

        for(i = 0; i < num; i++)
            s.append(boites[i].toString());

        return s.toString();
    }
}
```

4. Décrivez maintenant une classe `Formateur` qui accumule la suite des mots contenus dans un fichier texte. Le constructeur de cette classe prendra un nom de fichier sur lequel il construira un parseur.
   Décrivez les deux méthodes publiques : `lit`, qui lit la suite des mots retournés par le parseur et la range dans un tableau dont les éléments sont du type de la classe `BoiteMot`, et `imprime`, qui imprime cette suite de mots. Tous les mots d'un paragraphe sont accumulés dans une `BoiteComposite`. Le formateur commence avec une boîte composite vide. A chaque fois qu'il lit un nouveau mot, il y ajoute un espace et la boîte représentant ce mot. Enfin, lors d'une fin de paragraphe, il garde la boîte courante, et recommence avec une boîte composite vide. Il sera peut-être judicieux d'utiliser une méthode privée pour la fin d'un paragraphe. On fera attention à n'ajouter d'espaces ni au début ni à la fin d'un

paragraphe.

Correction :

```
public class Formateur
{
    private Parseur parseur;
    private Boite[] boites;
    private int num;
    private BoiteComposite boite;

    public Formateur(String filename){
        parseur = new Parseur(filename);
        num = 0;
        boites = new Boite[10];
        boite = new BoiteComposite();
    }

    public void lit(){
        String s;

        do {
            s = parseur.suivant();
            if(s == null || s.equals("")) {
                nouveauParagraphe();
                if(s == null)
                    break;
            } else {
                BoiteMot m;
                m = new BoiteMot(s);
                if(!boite.vide())
                    boite.accumule(new BoiteEspace());
                boite.accumule(m);
            }
        } while(s != null);
    }

    public void imprime(){
        for(int i = 0; i < num; i++) {
                    System.out.println(boites[i]) ;
       }
    }

    private void nouveauParagraphe(){
        if(!boite.vide()) {
            accumule(boite);
            boite = new BoiteComposite();
        }
    }

    private void redimensionne(int n){
        Boite[] nboites;
        int i;
        nboites = new BoiteComposite[n];
        for(i = 0; i < num; i++) {
            nboites[i] = boites[i];
```

```
        }

        boites = nboites;
    }

    private void accumule(Boite b){
        if(num >= boites.length)
            redimensionne(2 * num);
        boites[num] = b;
        num = num + 1;
    }
}
```

5. Modifiez le programme précédent pour qu'il coupe les lignes. Le formateur passe maintenant à une nouvelle boîte composite dès lors qu'ajouter le nouveau mot à la boîte courante lui ferait dépasser la largeur de la page (fixée arbitrairement, par exemple à 75). Cependant, on ne passe jamais à une nouvelle boîte si la boîte courante est vide (pourquoi ?).

   Pour des raisons esthétiques, le programme devra insérer une ligne blanche entre deux paragraphes.

   Correction :

```
public class Formateur
{
    static final int LARGEUR = 75;
    private Parseur parseur;
    private Boite[] boites;
    private int num;
    private BoiteComposite boite;

    public Formateur(String filename){
        parseur = new Parseur(filename);
        num = 0;
        boites = new Boite[10];
        boite = new BoiteComposite();
    }

    public void lit(){
        String s;

        do {
            s = parseur.suivant();
            if(s == null || s.equals("")) {
                nouveauParagraphe();
                if(s == null)
                    break;
            } else {
                BoiteMot m;
                m = new BoiteMot(s);
                if((!boite.vide()) &&
                   (boite.largeur() +
                    (new BoiteEspace().largeur() + m.largeur()) > LARGEUR)){
                    nouveauParagraphe();
                }
                if(!boite.vide())
```

```
                boite.accumule(new BoiteEspace());
            boite.accumule(m);
        }
    } while(s != null);
}

public void imprime(){

    for(int i = 0; i < num; i++) {
          System.out.println(boites[i].toString());
          System.out.println();
    }
}

private void nouveauParagraphe(){
   if(!boite.vide()) {
        accumule(boite);
        boite = new BoiteComposite();
    }
}

private void redimensionne(int n){
    Boite[] nboites;
    int i;
    nboites = new BoiteComposite[n];
    for(i = 0; i < num; i++) {
        nboites[i] = boites[i];
    }

    boites = nboites;
}

private void accumule(BoiteComposite b){
    if(num >= boites.length)
        redimensionne(2 * num);
    boites[num] = b;
    num = num + 1;
}
}
```

6. Ecrire une classe `Test` pour tester votre programme.
   Correction :

```
public class Test
{
    public static void main(String args[])
    {
        int i;
        for(i = 0; i < args.length; i++) {
            Formateur f;
            String s;
            f = new Formateur(args[i]);
            f.lit();
            f.imprime();
        }
```

```
    }
}
```