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

TD n°4 - Correction

Interfaces, polymorphisme (suite)

Exercice 1 Nous poursuivons cette semaine l'écriture d'un outil formateur de texte. Le texte produit dans le td précédente n'était pas justifié : la marge droite n'était pas alignée. Pour justifier le texte, nous allons introduire une nouvelle interface `BoiteEtirable` qui étend l'interface `Boite`. Les objets de l'interface `BoiteEtirable` pourront être convertis en chaînes de longueur arbitraire, ce qui nous permettra de justifier les lignes.

1. Justification

Afin de produire du texte justifié, on modifiera la méthode d'impression de la classe `Formateur` pour qu'elle imprime des espaces de largeur variable.

La méthode `etirable` et l'interface `BoiteEtirable` Commencez par ajouter à l'interface `Boite` une nouvelle méthode booléenne `etirable`, et ajoutez cette méthode à toutes les classes qui implémentent `Boite`. Pour le moment, cette méthode retourne `false` pour tous les objets.
Correction :

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

Il faut donc ajouter aux classes `BoiteEspace`, `BoiteMot` et `BoiteComposite` la méthode `etirable`.

```
public boolean etirable() { return false; }
```

Définissez maintenant une nouvelle interface `BoiteEtirable` qui étend `Boite` en lui ajoutant une méthode `toString(int n)` de type `String`. Dans le reste de cette partie, nous implémenterons cette nouvelle méthode qui doit convertir une boîte en une chaîne, mais en ajoutant `n` espaces supplémentaires aux endroits où cela peut se faire.
Correction :

```
public interface BoiteEtirable extends Boite
{
    public String toString(int n);
}
```

Les espaces étirables Modifiez maintenant la définition de la classe `BoiteEspace` pour qu'elle implémente l'interface `BoiteEtirable`. Toutes les `BoiteEspace`s sont étirables (la méthode `etirable` retourne toujours `true`), et `toString(n)` retourne simplement une chaîne de `n+1` espaces (l'espace d'origine, et `n` espaces ajoutés).
Correction :

```
public boolean etirable() { return true; }

public String toString(int n)
{
    StringBuffer s;
    int i;

    s = new StringBuffer(" ");
    for(i = 0; i < n; i++) s.append(" ");
    return s.toString();
}
```

Les boîtes composites étirables Le cas d'une boîte composite est un peu plus compliqué. Une boîte composite peut-être étirée dès qu'une des boîtes qu'elle contient peut l'être : la méthode `etirable` devra donc vérifier si c'est le cas.

La méthode `toString(n)` devra ajouter un certain nombre d'espaces à chaque boîte étirable contenue. Malheureusement, ce nombre n'est pas toujours constant : si une boîte composite contient deux boîtes étirables, et il faut rajouter trois espaces, il faudra ajouter deux espaces à la première, mais un seul à la seconde.

Supposons qu'une boîte composite contienne e boîtes étirables et qu'on veuille l'étirer de n espaces ; le nombre exact d'espaces à rajouter à chaque boîte étirable contenue est alors

$$n_{\text{esp}} = \frac{n}{e}.$$

Cependant, n peut très bien ne pas être divisible par e ; on calcule donc

$$n_{\min} = [n_{\text{esp}}],$$

la partie entière de n_{esp}, qui est le nombre minimal d'espaces à rajouter à une boîte étirable. Le nombre d'espaces qui nous restent est alors

$$n_{\text{supl}} = n - e \times n_{\min}.$$

La méthode `toString(n)` de la classe `BoiteComposite` devra donc calculer les entiers $n_{\min}$ et n_{supl} comme ci-dessus, et ensuite retourner la concaténation de ses éléments ; les n_{supl} premiers éléments étirables devront être étirés de $n_{\min} + 1$ espaces, tandis que les autres devront l'être de $n_{\min}$ espaces seulement.
Correction :

```
public boolean etirable()
{
    int i;

    for(i = 0; i < num; i++)
        if(boites[i].etirable())
```

```
            return true;

    return false;
}

public String toString(int n)
{
    int netirable = 0;      // nombre de boites etirables
    int nmin;               // nombres d'unites a ajouter par boite
    int nsupl;              // nombre d'unites qui restent
    int i;

    StringBuffer s = new StringBuffer();

    if(!etirable())
        return toString();

    for(i = 0; i < num; i++)
        if(boites[i].etirable())
            netirable = netirable + 1;

    nmin = n / netirable;
    nsupl = n - nmin * netirable;

    for(i = 0; i < num; i++) {
        if(boites[i].etirable()) {
            BoiteEtirable b = (BoiteEtirable)boites[i];
            if(nsupl > 0) {
                s.append(b.toString(nmin + 1));
                nsupl = nsupl - 1;
            } else {
                s.append(b.toString(nmin));
            }
        } else {
            s.append(boites[i].toString());
        }
    }
    return s.toString();
}
```

Casts contravariants Dans la classe `Formateur`, on a un tableau de `Boites` ; parmi celles-ci, certaines sont étirables, d'autres ne le sont pas. Afin de nous servir de la méthode `toString(n)` de ces dernières, il faudra informer le système que ce qui n'est apparemment qu'une `Boite` est en fait une `BoiteEtirable`. Cela se fait au moyen d'un changement de type (cast) *contravariant*, par exemple comme ceci :

```
Boite b;
BoiteEtirable be;

b = ...;
be = (BoiteEtirable)b;
```

Ecrire la méthode `imprime` de la classe `Formateur` pour qu'elle étire les lignes étirables[1] afin d'arriver à une largeur uniforme de 75 caractères.
Correction :

```
public void imprime()
{
    int i;
    int nadd;
    for(i = 0; i < num; i++) {
        nadd = LARGEUR - boites[i].largeur();
        if(boites[i].etirable()) {
            System.out.println(((BoiteEtirable)boites[i]).
                               toString(nadd));
        } else {
            System.out.println(boites[i].toString());
            System.out.println();
        }
    }
}
```

2. Gestion des fins de paragraphes

Le programme précédent a un défaut flagrant : il justifie toutes les lignes, même celles qui sont à la fin d'un paragraphe. Il va donc falloir le modifier pour inhiber la justification de ces dernières.

On pourrait penser à définir une nouvelle classe qui ressemble à `BoiteComposite` mais dont les objets ne sont jamais étirables. Cependant, Java n'offre pas de facilités pour changer la classe d'un objet après sa création[2], et donc nous n'aurions aucun moyen de changer la classe de la boîte courante au moment d'arriver à la fin du paragraphe.

La solution que nous avons retenue consiste à inclure dans la classe `BoiteComposite` un nouveau champ booléen qui sert à inhiber l'étirage. Une nouvelle méthode, `inhibeEtirage()` affecte `true` à ce champ, la méthode `setEtirable(boolean b)`, et la méthode `etirable` retourne toujours `false` lorsque ce champ est vrai.
Correction : Il faut ajouter le champs `boolean inhibe` à la classe `BoiteComposite` , ainsi que les trois méthodes suivantes :

```
public void inhibeEtirage()
{
    inhibe = true;
}


public void setEtirable(boolean b)
{
    inhibe = b;
}

public boolean etirable()
{
    int i;
```

[1] Pourquoi certaines lignes risquent-elles de ne pas être étirables ?
[2] Un exemple de langage qui offre de telles facilités est Common Lisp.

```
        if(inhibe)
            return false;

        for(i = 0; i < num; i++)
            if(boites[i].etirable())
                return true;

        return false;
    }
```

Implémentez la gestion des lignes en fin de paragraphe dans la classe `Formateur`.
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(false);
                if(s == null)
                    break;
            } else {
                BoiteMot m;
                m = new BoiteMot(s);
                if((!boite.vide()) &&
                   (boite.largeur() +
                    (new BoiteEspace().largeur() + m.largeur()) >
                    LARGEUR)) {
                    nouveauParagraphe(true);
                }
                if(!boite.vide())
                    boite.accumule(new BoiteEspace());
                boite.accumule(m);
            }
        } while(s != null);
```

```
    }

    public void imprime()
    {
        int i;
        int nadd;
        for(i = 0; i < num; i++) {
            nadd = LARGEUR - boites[i].largeur();
            if(boites[i].etirable()) {
                System.out.println(((BoiteEtirable)boites[i]).
                                    toString(nadd));
            } else {
                System.out.println(boites[i].toString());
                System.out.println();
            }
        }
    }

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

    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;
    }

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

        boites[num] = b;
        num = num + 1;
    }
}
```