Analyse de cas

Programmation objets
et Java

(B. Sonntag)

Emboîtement de classes

- Il arrive souvent qu' un objet autonome soit intimement lié à un autre objet.
- En fait un tel objet n'a aucun sens s'il n'est pas associé à un autre.
- Par exemple un itérateur de liste n'a aucun sens s'il n'est pas attaché à une liste particulière.

Emboîtement de classes

- En Java, on exprime assez précisément une telle décision de conception par un emboîtement de classes. Si la déclaration de la classe ***I*** est emboîtée dans la classe ***G***, alors une instance de ***I*** ne peut être créée que dans le contexte d'une instance de la classe ***G***. Autrement dit, un ***I*** ne peut être créé que par un ***G*** et le ***I*** aura accès à l'*intérieur* du ***G*** qui l'a créé, y compris les membres privés du ***G***.

Emboîtement de classes

- Les objets Enumeration fournis par les classes Java Vector et Hashtable sont en fait implantés par des classes emboîtées.

Emboîtement de classes

```
class Intervalle {
    private int bi, bs;
    public Intervalle (int bi, int bs)
    {
        this.bi = bi ;
        if (bs < bi) {
            this.bs = bi - 1 ; // intervalle vide
        } else {
            this.bs = bs ;
        }
    }
    public Iterateur donneIterateur ()
    { return new Iterateur () ; }
    /* … */
```

Emboîtement de classes

```
class Intervalle {
    /* … */
    public class Iterateur {
        private int cour = bi ;
        public int next ()
        { return cour++ ; }
        public boolean fini ()
        { return cour == bs+1 ; }
    }
}
```

Les instances de Iterateur peuvent accéder directement aux variables d'instance de l'instance d'Intervalle.

Emboîtement de classes

L'emboîtement de classes a plusieurs avantages :

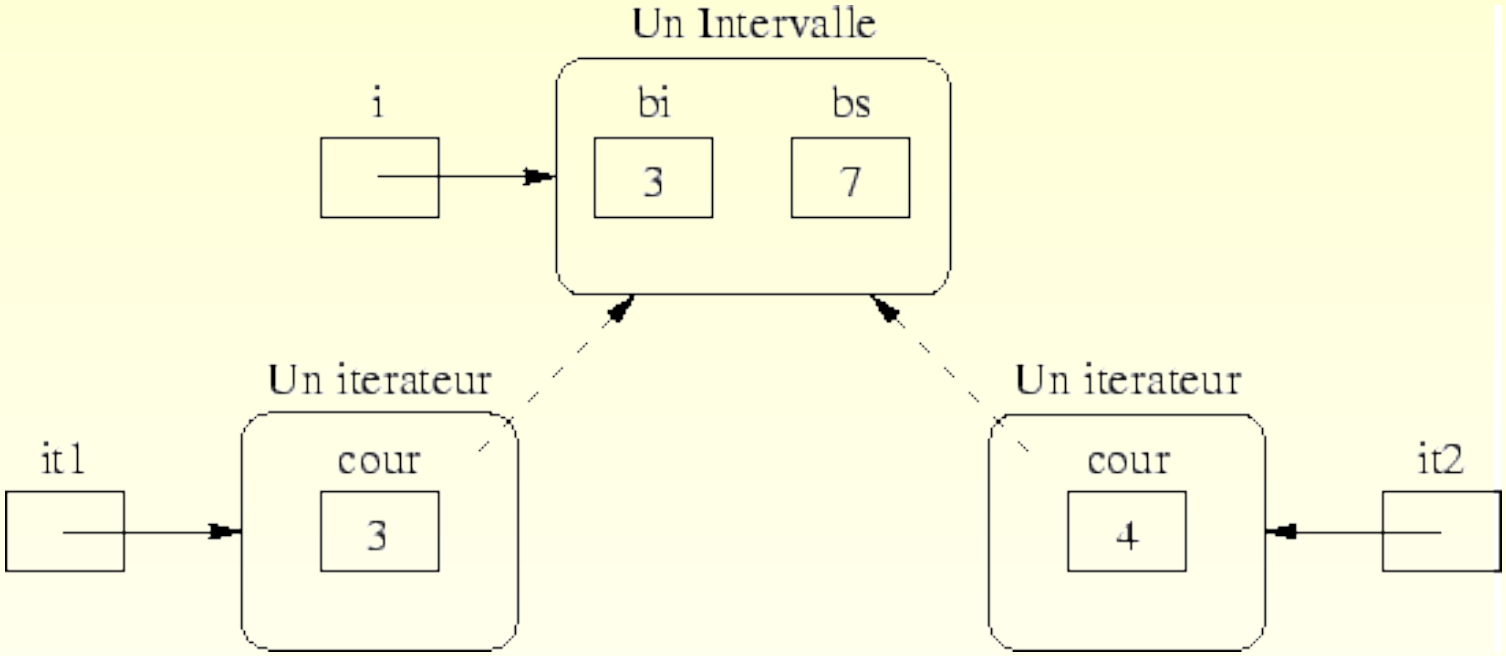
- il reflète bien une décision de conception ,
- les objets emboîtés sont dans un état cohérent dès leur création,
- il allège les écritures,
- il permet d'éviter de dévoiler le fonctionnement de l'objet englobant, ce qu'on serait obligé de faire si on concevait la classe Iterateur à l'extérieur de Intervalle,
- un même objet englobant, peut posséder un nombre quelconque d'objets emboîtés.

Emboîtement de classes

Ainsi après exécution du code :

```
Intervalle i = new Intervalle (3,7);
Intervalle.Iterateur
    it1 = i.donneIterateur(),
    it2 = i.donneIterateur() ;
    it2.next () ;
```

Emboîtement de classes



Un objet Intervalle et deux Itérateurs sur cet intervalle, les flèches en hachuré indiquent à quel Intervalle appartiennent ces deux Itérateurs, en Java on désigne ces flèches par la notation Intervalle.this.

Emboîtement de classes

Utilisons notre Iterateur pour faire la somme des premiers entiers :

```
class EssaiIntervalle {
    public static int somme (Intervalle inter)
    { int s = 0 ;
        for (Intervalle.Iterateur i = inter.donneIterateur() ; ! i.fini() ;) {
            s = s + i.next () ;
        }
        return s ;
    }
}
```

Emboîtement de classes

Dans une instance de classe emboîtée, this fait bien sûr référence à cette instance. Si on veut faire référence à l'instance de la classe englobante qui l'a créée on utilise la notation *ClasseEnglobante.this*.

```
class A {
    private int v = 1 ;
    class B {
        private int v = 2 ;
        public int xxx () {
            return this.v + A.this.v ;
            // le v de B le v de A
         }
    }
}
```

la méthode xxx() renvoie toujours la valeur 3.

Emboîtement de classes

Pour conclure :

- l'emboîtement de classes permet bien de faire de la composition.
- Mais la relation contenant/contenu est inversée :

en composition classique c'est le contenant qui connaît son contenu, en composition par emboîtement de classe, c'est le contenu qui connaît son contenant.

Héritage Alimentaire

- L'héritage alimentaire consiste à utiliser l'héritage uniquement pour réutiliser du code.

- Il faudrait donc cacher à l'utilisateur final un tel héritage et l'empêcher d'utiliser par erreur une méthode indûment héritée.

Héritage Alimentaire

La bibliothèque Java propose un bel exemple d'héritage alimentaire qui n'est pas caché.

Il s'agit de la classe **Stack** qui est implémentée par héritage de la classe **Vector** avec l'ajout de quelques méthodes spécifiques à la notion de *pile* comme push() et pop().

Cet héritage fait qu'il est tout à fait possible de considérer une pile comme un vecteur et, par exemple, d'insérer un élément en plein milieu de la pile !

Ceci ne correspond pas vraiment aux fonctionnalités *normale* d'une pile.

Héritage Alimentaire

Voici comment la composition permet de résoudre proprement le problème tout en réutilisant la classe Vector :

```
import java.util.Vector ;
public class Pile {
    private Vector v = new Vector () ;
    public Object sommet () {
        return v.lastElement () ;
    }
    public void empiler (Object o) {
        v.addElement(o) ;
    }
    public void depiler (Object o) {
        v.removeElementAt(v.size () - 1) ;
    }
}
```

Les 3 utilisations de final

final : pour déclarer une constante

Une variable qualifiée par final est en fait ... une constante ! et doit être initialisée.

```
final float TAUX = 0.206 ;
final int TAILLE = 29 ;
```

Une chose intéressante est qu'une variable d'instance finale peut être initialisée par le constructeur : la valeur de cette constante peut ainsi être fournie au dernier moment par l'utilisateur.

Les 3 utilisations de final

final : pour empêcher la redéfinition d'une méthode

Une méthode qualifiée par final ne peut pas être redéfinie (*overridden*) dans les sous-classes. Ceci est très utile pour interdire la redéfinition d'une méthode.

```
public final void raz () { v = 0 ; }
```

Au niveau conceptuel, une méthode finale correspond souvent à un algorithme général valable pour toutes les sous-classes de la classe qui définit cette méthode.
Bien entendu, une méthode finale ne peut pas être abstraite.

Les 3 utilisations de final

final : pour empêcher la spécialisation d'une classe

Une classe qualifiée par final ne peut pas avoir de sous-classes, elle ne peut pas non plus être abstraite.

public final class FEUILLE {...}

Ceci peut permettre d'interdire le sous classement de certaines classes critiques fournies par un éditeur de composants logiciels.

Les 3 utilisations de final

Une métode private est implicitement final

Une méthode private est de fait final car elle ne peut de toutes façons pas être redéfinie dans ses sous-classes puisqu'elle est inconnue de celles-ci.

Si, dans une sous classe on définit une méthode de même profil qu'une méthode privée de la super classe, il n'y a aucun lien entre ces deux méthodes.

Les 3 utilisations de final

```
class Mere {
    private void privee ()          { System.out.println ("Mere") ; }
    public void essai ()            { privee () ; }
}
class Fille extends Mere {
    public void privee ()           { System.out.println ("Fille") ; }
}
public class Privee {
    public static void main (String arg[]) {
        Fille f = new Fille () ;
        f.essai () ;
        f.privee () ;
    }
}
```

imprime :

```
Mere
Fille
```

Indication explicite du type d'un objet

```
class Pile {
    private int nb = 0 ;
    private Object v [] = new Object [20] ;
    void empiler (Object o)
    {
        v [nb++] = o ;
    }
    Object sommet ()
    {
        return v [nb-1] ;
    }
}
```

Indication explicite du type d'un objet

```
Pile p = new Pile () ;
Compteur c ;

p.empiler (new Compteur ()) ;
(p.sommet ()).incr () ;                    // erreur de compilation
c = p.sommet () ;                  // erreur de compilation
```

Indication explicite du type d'un objet

```
Pile p = new Pile () ;
int tab [] = new int [15] ;

p.empiler (new Compteur ()) ;
((Compteur) p.sommet ()).incr () ;
Compteur c = (Compteur) p.sommet () ;

p.empiler (tab) ;
int v [] = (int []) p.sommet () ;
```

Évidemment si à l'exécution il s'avère que le sommet de la pile n'est pas compatible avec Compteur (ou un peu plus loin avec le type int[]) alors l'exception ClassCastException sera déclenchée.

Égalité de référence ou égalité d'objet ?

Un problème classique est de savoir ce qu'on entend par égalité de deux objets :

• s'agit-il de savoir si on a deux fois le même objet, c'est une simple **égalité de référence** qu'on teste avec l'opérateur ==,

• ou bien s'agit-il de savoir si deux objets sont **jumeaux**, auquel cas il faut comparer les composants des deux objets.
Cette comparaison peut être récursive pour les composants qui sont eux-mêmes des références. Cette **égalité d'objet** doit être programmée en redéfinissant la méthode :
public boolean equals (Object obj)
définie dans la classe Object. Par défaut, cette méthode fait simplement une égalité de référence.

Égalité de référence ou égalité d'objet ?

```
class Compteur {
    private int v = 0 ;
    public boolean equals (Object obj)
    {
        if (obj instanceof Compteur) {
            return v = = ((Compteur) obj).v ;
        } else {
            return false ;
        }
    }
}
```

Égalité de référence ou égalité d'objet ?

• L'opérateur **instanceof** permet de savoir si son opérande gauche est compatible avec le type référence qui est à sa droite. En particulier l'expression **obj instanceof Compteur** ne vaut vrai que si **obj** est non null et que c'est une instance de **Compteur** ou d'une de ses sous-classes.

• Une méthode d'instance peut *voir* les membres privés d'un autre objet que this à condition qu'elle soit définie dans la classe qui définit ces membres. C'est ce qui se passe dans l'instruction

return v = = ((Compteur) obj).v ;

Égalité de référence ou égalité d'objet ?

```
class Compteur {
    private int v = 0 ;
    public boolean equals (Compteur c)
    {
        if (c != null) {
            return v = = c.v ;
        } else {
            return false ;
        }
    }
}
```

Surcharge !

Égalité de référence ou égalité d'objet ?

```
{ Compteur c1 = new Compteur (), c2 = new Compteur () ;

    if (c1.equals (c2)) {
        // appel de la methode equals (Compteur c)
        ...
    }
}
```

*Pas de problème, **mais** …*

Égalité de référence ou égalité d'objet ?

```
{
    Compteur c1 = new Compteur ();
    Object c2 = new Compteur () ;

    if (c1.equals (c2)) {
        // appel de la methode equals (Object obj) de Object
        ...
    }
}
```

Gros problème !

Classe abstraite ou interface ?

Points communs entre interface et classe abstraite

1. tous deux spécifient un comportement (éventuellement abstrait),
2. tous deux permettent d'utiliser le polymorphisme et la liaison dynamique grâce à la compatibilité de type,
3. tous deux peuvent donner lieu à implémentation multiple :
    - une classe abstraite peut avoir plusieurs sous-classes concrètes,
    - une interface peut être implémentée par plusieurs classes.

Classe abstraite ou interface ?

Différences entre interface et classe abstraite

1. le comportement d'une interface est purement abstrait, par contre une classe abstraite peut implémenter tout ou partie du comportement, dit autrement : une interface ne permet pas de factoriser l'implémentation alors qu'une classe abstraite peut le faire,

2. une classe peut implémenter plusieurs interfaces (chaque interface représente alors un vue différente sur un même objet), par contre une classe hérite exactement d'une super-classe (héritage simple),

3. les interfaces permettent de représenter des relations inter-classes indépendantes des relations d'héritage.

Classe abstraite ou interface ?

- les interfaces ne permettent de factoriser que la spécification d'un comportement, pas son implémentation,

- une classe abstraite permet de factoriser à la fois la spécification et au moins une partie de l'implémentation.

Ceci semble faire pencher la balance du côté des classes abstraites. Cependant les interfaces reprennent l'avantage dès qu'on veut organiser des comportements qui ne *collent* pas avec la hiérarchie de classes. Un exemple frappant est l'interface prédéfinie Runnable qui permet de donner un scénario à un objet qu'on veut rendre actif en lui attachant son propre flot d'exécution (Thread) : grâce à cette interface, toute classe peut proposer un scénario pour ces instances, quelle que soit sa situation dans la hiérarchie des classes.

Méthodes et variables de classe : membres **static**

Certains comportements liés à une classe sont parfois indépendant de toute instance de cette classe.

Il s'agit alors d'un comportement de la classe et non pas d'une instance en particulier.

Un tel comportement est implémenté en Java par **une méthode de classe** (par opposition à méthode d'instance qui est un comportement d'objet et non pas de classe). Une méthode qualifiée par **static** est une **méthode de classe** (c'est le cas de **main**()).

Méthodes et variables de classe : membres **static**

De la même manière, certaines variables ou constantes sont communes à toutes les instances, on les appelle des **variables de classe** et on les déclare **static**.

Elles existent alors en **un seul exemplaire**, quel que soit le nombre d'instances de la classe. Ces variables de classe existent pendant toute la durée de l'exécution du programme (on peut parler de **variables globales**).

Méthodes et variables de classe : membres **static**

```
class Bidon {
    public static final float TVA = 0.206 ;       // constante de classe
    private static int nbInstances = 0 ;          // variable de classe public

    static int combien ()
    { // methode de classe
        return nbInstances ;
    }
    public Bidon ()
    {
        nbInstances = nbInstances + 1 ;
    }
    public float prixTTC (float prixHT)
    { // methode d'instance utilisant une variable de classe
        return prixHT + prixHT*TVA;
    }
}
```

Constitution des paquetages

Le mot clef **package** suivi du nom du paquetage doit figurer en **premier** dans un fichier source Java.

Toutes les classes de ce fichier font alors partie de ce paquetage, **mais** une seule de ces classes peut être publique et doit alors donner son nom au fichier.

```
package desCompteurs ;
public class Compteur {
    ...
}
```

Constitution des paquetages

Plusieurs fichiers sources peuvent participer au contenu d'un même paquetage, ceci permet d'éviter des fichiers monstrueux et des compilations trop longues, et aussi de faire qu'un paquetage propose plusieurs classes publiques.
Si on veut que le paquetage desCompteurs propose aussi la classe CompteurMod on écrit un autre fichier :

```
package desCompteurs ;
public class CompteurMod extends Compteur {
    ...
}
```

Constitution des paquetages

La structuration en paquetages est hiérarchique et utilise une notation pointée, chaque point séparant deux niveaux consécutifs de la hiérarchie, par exemple les paquetages **java.lang** et **java.awt.event**.

Utiliser une classe ou interface définie dans un autre paquetage

Le nom complet d'une classe/interface est formé en préfixant le nom de la classe/interface par le nom du paquetage. Par exemple **desCompteurs.Compteur** est le nom complet de la classe **Compteur** du paquetage **desCompteurs**, et **java.util.Vector** est le nom complet de la classe **Vector** du paquetage **java.util**.

Utiliser une classe ou interface définie dans un autre paquetage

Il y a deux possibilités pour utiliser un paquetage :

1. soit on donne le *nom complet* de l'interface ou de la classe à chaque fois qu'on veut référencer une classe/interface publique du paquetage,

2. soit on insère une clause d'importation de la classe/interface en début de fichier, on pourra alors utiliser le nom simple de la classe/interface.

Utiliser une classe ou interface définie dans un autre paquetage

Sans clause d'importation

```
public class EssaiNomComplet {
    public static void main (String argv [])
    {
        desCompteurs.Compteur c = new desCompteurs.Compteur () ;
        c.incr () ;
        ...
    }
}
```

Utiliser une classe ou interface définie dans un autre paquetage

Avec clause d'importation préalable

Chaque clause d'importation est introduite par le mot clef **import** suivi du *nom complet* de la classe ou de l'interface.

```
import desCompteurs.Compteur ;
import java.util.Vector ;
public class EssaiImport {
   public static void main (String argv [])
   {
      Vector v = new Vector () ;
      Compteur c = new Compteur () ;
      c.incr () ;
      ...
   }
}
```

Utiliser une classe ou interface définie dans un autre paquetage

Si on veut importer brutalement toutes les classes d'un paquetage, on peut remplacer le nom de classe par le caractère * dans la clause d'importation.

```
import desCompteurs.* ;

public class EssaiCompteur {
   public static void main (String argv [])
   {
      Compteur c = new Compteur () ;
      c.incr () ;
      ...
   }
}
```