Cours de base d'Ingéniérie des applications objet. Polymorphisme, Sous-typage, Héritage, Réutilisation, Hiérarchies

Support de Cours
Christophe Dony
Université Montpellier-II - ISIM - IG2
dernière mise à jour : 2001

1 Polymorphisme

morphisme : du grec "morphe" signifiant forme.

morphologie: etude de la structure externe (la forme) des êtres vivants **polymorphe (vs. monomorphe)** : qui peut se presenter sous des formes différentes (vs. forme unique).

monomorphisme en programmation : les valeurs ont un type unique (ex: en Pascal, 1 est un entier).

polymorphisme en programmation : les valeurs peuvent appartenir a plusieurs types.

ex: en Smalltalk, 1 est un entier et un nombre, en lisp, nil appartient à la fois au type "liste" et au type "booléen".

Fonction polymorphique : fonction dont les operandes sont polymorphes.

Type polymorphique: type dont les opérations sont des fonctions polymorphiques.

2 Polymorphisme d'inclusion

Modele autorisant un type T' a etre un sous-type d'un autre type T.

- **interprétation extensionnelle** : inclusion ensembliste :

  tous les éléments de T' sont aussi des éléments de T.

  Ex: Tous les entiers sont des nombres .

- **interprétation intensionnelle** :

  l'ensemble des propriétés définissant en intension les éléments de T' est un sur-ensemble de l'ensemble de celles définissant les éléments de T.

  Ex: Etudiant est un sous-type de Personne.

  Cette interprétation est celle qui est utilisée en programmation par objets, elle est parfois cohérente avec l'interprétation ensembliste (exemple etudiant-personne) mais pas toujours. Ainsi dans l'exemple suivant, `ParallépipedeRectangle` sera defini en PPO comme un sous-type de `Rectangle` alors que l'interprétation ensembliste est visiblement non satisfaisante dans ce cas, ce sont plutôt les rectangles qui sont des sortes de PR.

```
Rectangle (IV: longueur, largeur) (Meth: aire)

ParallélépipèdeRectangle (IV: longueur, largeur, hauteur)
(Meth: aireBase, volume).
```

3 polymorphisme d'inclusion, description différentielle et héritage

L'interprétation intensionnelle implique que :

Tout élément d'un sous-type ST possède l'ensemble des proprietes, variables d'instances et méthodes, definies par un super-type T de ST.

Toute opération definie par T (utilisant ces variables) est applicable aux instances de ST. Il n'est pas nécessaire de la définir lorsqu'on definit ST.

Les méthodes définies par T sont applicables aux instances de T et de ST. Elles sont polymorphes. T est un type polymorphique.

Description différentielle : La definition de ST peut donnée en intension en se limitant à l'expression des différences avec la définition de T.

Toute les propriétés de ST définies au niveau de T, sont dites héritées par ST.

4 Sous-classe

4.1 Définition

Sous-classe : Classe étendant une classe existante C, c'est à dire décrivant des objets dont l'ensemble de membres est un sur-ensemble de celui des instances de C.

On déclare et définit dans une sous-classe SC, tout ce qui, dans la description statique et fonctionnelle des éléments de SC, n'est pas déjà déclaré et définit dans les **sur-classes** (ou **super-classes**)de SC.

Définition en JAVA (mot-clé *extends*).

Exemple : `Point3D` sous-classe de `Point`.

```
class Point3D extends Point {
    private double z;
    ...
    public double getz() {return z;}
    ...
}

{\bf Sous-type} : une sous-classe SC définit un type ST, sous-type du
type T défini par la surclasse C de SC.
```

4.2 Nouveau modificateurs et mots clés

- *protected* : modificateur de membre d'une classe, le *membre* concerné est visible dans la classe et dans ses sous-classes.
- *final* : modificateur de classe, la classe concernée ne peut avoir de sous-classes.

5 Héritage

La relation "est-sous-classe-de" induit un héritage. Une sous-classe hérite de tous les membres de sa super-classe.

Attention, ceci ne signifie pas que tous les membres héritées soient nécessairement accessibles dans les méthodes de la nouvelle classe (voir les définitions des mots-clés *private, package, protected.*

5.1 Héritage des définitions structurelles

Une instance d'une classe possède autant de champs qu'il y a de variables d'instance dans sa classe et ses super-classes (héritage, dit statique car calculable à la compilation, des attributs).

Exemple : une instance de Point3D possède 2+1 soit 3 champs.

5.2 Héritage des méthodes

Toute méthode définie sur une sur-classe C est applicable aux instances des sous-classes de C.

5.3 Mise en oeuvre de l'héritage des méthodes

Interprétation de l'envoi de message - Seconde version - Liaison dynamique prenant en compte la relation "est-sous-classe-de".

```
send (rec sel listArgs){
        M := lookup (rec sel);
        if (M == null)
        then erreur : l'objet ne comprend pas le message;
        else apply (M, rec, listArgs));}

lookup (rec, sel){
   trouvé := false;
   m := null;
   c = class(rec);     // résolytion de la surcharche.
   while ((c != nil) and (not trouve)){
         // cherche la méthode sel dans c
         m := searchMethod (c, sel);
         if m != nil then trouvé := true;
         else c := superclass(c);}
   return (M);
```

5.4 La classe Object

`Object` est la racine du graphe d'héritage et est donc la sur-classe commune à toutes les classes.

Sur la classe `Object` sont définies les méthodes applicables à tous les objects (exemple la méthode `equals` de test d'égalité logique de deux objets ou la méthode `toString` fabriquant une chaîne de caractère représentative du receveur.

Exemple:

```
    new Point(3, 4).equals (new Point(3, 4));
    -> true
    new Point(1, 2).toString();
    -> "1@2"
```

Exercice : écrivez les méthodes `equals` et `toString` permettant d'obtenir les résultats précédents.

6 Sous-classes et initialisations des objets

Soit SC une sous-classe. L'exécution d'un constructeur de SC se déroule en trois phases :

1. exécution automatique ou spécifiée d'un constructeur de la sur-classe de SC,
2. initialisation des variables d'instance suivant les valeurs d'initialisation qui leur ont été donné au moment de leur déclaration,
3. exécution du corps du constructeur.

Ainsi, pour un constructeur `cons` d'une classe SC dérivée de C :

1. `cons` PEUT invoquer un constructeur de C grâce à une instruction de la forme : `super(...)`.
2. S'il ne le fait pas, le constructeur sans arguments de C, s'il existe, est automatiquement appelé avant toute autre chose.
3. Si C ne possède pas de constructeur sans arguments, `cons` DOIT appeler un constructeur de C (instruction `super(...)`) ou au autre constructeur `cons2` de SC, grâce à une instruction de la forme `this(...)`.
4. Retour à l' étape 1 pour `cons2`.

Exemples :

```
public class Point3D extends Point {
    double z;

    public Point3D(){
        super();    //optionnel, c'est fait par défaut.
        z = 0};}

    public Point3D(double i){
        this(0,0,i).

    public Point3D(double i, double j, double k){
        super(i, j);
        z = k;}
```

7 Redéfinitions

7.1 Redéfinition des variables d'instance

La sémantique de la redéfinition d'un attribut est variable voire floue selon les langages.

Dans un langage dynamiquement typée et sans modifieurs, elle ne présente aucun intérêt.

En Java elle pourrait être utile pour modifier le type ou la visibilité d'un attribut dans une sous-classe mais elle est ineffective. La redéfinition d'un attribut entraîne une surcharge i.e. la définition de deux attributs de même nom. Le débutant oubliera cette question.

7.2 Redéfinition de méthodes, masquage

Redéfinition de méthode : définition d'une méthode sur une sous-classe SC lorsque qu'une méthode de même nom est définie sur une des sur-classes de SC.

Toutes les méthodes non statiques (pourquoi cette restriction ?) peuvent être redéfinies.

Masquage : terme utilisé quand la nouvelle définition sur SC masque, pour les instances de SC et de ses futures éventuelles sous-classes, la définition précédemment héritée.

Contraintes :

– **Redéfinition invariante (en C++ et Java)** : Les signatures et le type de retour d'une redéfinition doivent être identiques à ceux de la méthode redéfinie.

– Chaque type d'exception signalée par la méthode masquante doit être un sous-type de celles de la méthode masquée (la covariance est acceptée pour ce qui concerne le type des exceptions signalées).

Dans l'exemple suivant, Ex2 doit être un sous-type de Ex1.

```
class A {
  void m() throws Ex1 { ... }

class B extends A{
  void m() throws Ex2 { ... }
```

– Si la méthode redéfinie ne signale pas d'exceptions, la redéfinition ne peut pas en signaler non plus.

8 Sous-classes et sous-types

Si les contraintes précédemment énoncées pour la redéfinition des méthodes sont respectées, toute sous-classe définit un sous-type du type défini par sa super-classe.

8.1 Affectation polymorphique

Affectation polymorphique : affectation d'un objet de type ST à une variable déclarée de type T lorsque ST est sous-type de T.

L'affectation suivante :

```
T x;
ST y = new ST();
x := y;
```

est possible lorsque ST est un sous-type de T.

Exemple :

```
Collection c;
Vector v = new Vector();
c = v:
```

L'affectation polymorphique est à la source de toutes les possibités d'extension et de réutilisation utilisant l'héritage.

8.2 Appel de méthode héritée et affectation polymorphique

Une affectation polymorphique est réalisée à chaque fois qu'un envoi de message entraine l'exécution d'une méthode héritée.

```
(new Point(2,3)).clone().
```

L'envoi de message précédent invoque la méthode `clone` définie sur la classe `Object`, *this* contient une référence sur le receveur courant qui est de type `Point` alors que le type statique de this dans cette méthode est `Object`. La liaison du paramètre formel *this* à l'argument est une affectation polymorphique

8.3 Coercion

Pour les types référence, les possibilités de coercion intègrent le sous-typage.

```
Point p = new Point3D(1,2,3); // Affectation polymorphique - ok

Point3D p3 = (Point3D) p;     // Coercion - ok
```

Pour la seconde ligne, le programmeur sait que p, déclaré de type Point contient néanmoins l'adresse d'un point3D, le compilateur ne le sait pas. La coercion renseigne le compilateur qui ne génère donc pas d'erreur mais néanmoins un test qui sera réalisé à l'exécution.

8.4 Redéfinitions Invariantes des méthodes

La théorie des types admet la notion de sous-type mais impose une redéfinition des méthodes covariante pour le type de retour et contravariante pour les types des paramètres.

Dans les langages à objets, le besoin dicterait plutôt une utilisation inverse i.e. des redéfinitions covariantes. Java les interdit et impose l'invariance.

Exemple : méthode `equals`

```
class Object {
  ...
  public Boolean equals (Object o) { ... }
  ...}

class Stack extends Object {
  int index = 0;
  Object[] buffer;

  public boolean equals (Object p) {
    Pile pp = (Pile) p;
    return(buffer.equals(pp.getBuffer()));}
```

8.5 Redéfinitions invariantes et coercion

Noter la coercion explicite dans la méthode equals de la classe `Stack`.

8.5.1 Comprendre la pré-compilation de l'envoi de message

Expliquer comment l'envoi de message est compilé en utilisant des tables. Construire la table des méthodes de `Object` et de `Pile`.

8.6 Le pourquoi du problème avec les redéfinitions covariantes

8.6.1 Le problème théorique

Un problème apparait avec l'utilisation conjuguée de redéfinitions non invariantes, de l'affectation polymorphique et de la liaison dynamique.

`Plus tard`

Une redéfinition covariance nécessiterait, pour ne pas poser de problèmes, un mécanisme d'envoi de message particulier dont la description sort du cadre de ce cours.

8.6.2 Comment Java traite les redéfinitions covariantes

```
class Object {
    ...
    public Boolean equals (Object o) { ... }
    ...}

class Stack extends Object {
    int index = 0;
    Object[] buffer;
    ...

    public Object[] getBuffer() {
        return((Object[])buffer.clone());
    }
...
```

```
    public boolean equals (Pile p) {
        return(buffer.equals(p.getBuffer()));}
```

Utilisation

```
class testPile {
   public static void main(String[] args) throws Exception {
      Pile p2 = new Pile(4);
      p2.push(new Integer(3));
      p2.equals(p2);
      Object o = p2; //affectation polymorphique
      o.equals(p2);
```

Le second envoi du message equals invoque la méthode equals de Object : le compilateur n'a pas considéré `equals` de `Pile` comme une redéfinition du `equals` de Object (signature différente).

9 Schémas de redéfinition et de réutilisation

Présentation des schémas classiques de réutilisation. La compréhension de ces schémas, du plus simple au plus complexe, est fondamentale pour la réalisation d'applications bien faites.

9.1 Definition de nouvelles méthodes sur une sous-classe

Exemple, la méthode `getz()` sur la classe Point3D.

9.2 Masquage

Exemple 1 : la méthode plus de la classe Point3D.

Exemple 2 : la méthode equals sur Point, sur Point3D, sur Pile.

9.3 Masquage partiel

Masquage partiel : Redéfinition faisant appel à la méthode redéfinie.

```
class Point {
...
void scale (float factor) {
x = x * factor;
y = y * factor; }

class Point3D {
...
void scale(float factor) {
super.scale(factor);
z = z * factor;}}
```

La forme syntaxique `super` est en fait un envoi de message au receveur courant avec une politique spécifique de recherche de méthode.

Envoyer un message au receveur *super*, revient à envoyer un message au receveur courant mais en commençant la recherche de méthode dans la surclasse de la classe dans laquelle a été trouvée la méthode en cours d'exécution.

9.4 Paramétrage de méthodes existantes

Comment adapter une méthode dont on hérite a ses propres besoins sans la modifier et sans dupliquer de code ...

Comprendre avec la méthode `fillWith` de la classe `Stack`.

Voir un exemple plus complexe avec des méthodes paramétrables de la bibliothèque des collections.

La classe `AbstractList` représente les collections ordonnées. Vector en est une sous-classe. Méthodes de la classe `AbstractList`

```
int indexOf(Object o) signals NotFoundException {
    return recherche (o, 0, this.size())}

int recherche (Object o, int index, int size)
    throws NotFoundException {
    for (i = index, i < size, i++) {
       if (this.get(i) == o) return (index);}
    throw new NotFoundException(this, o);}
```

Expliquer comment paramétrer la méthode sur les sous-classes comme Vector par exemple.

9.5 Classes et méthodes abstraites

Méthode Abstraite : méthode déclarée sur une classe C mais non définie (corps vide).

Les méthodes de C utilisant une méthode abstaite MA ne seront utilisables que par des instances de sous-classes de C ou MA sera définie.

Classe Abstraite : classe déclarant des méthodes abstraites.

Une classe abstraite n'est pas instantiable.

10 Interfaces

10.1 Définitions

Interface : une collection de déclaration de méthodes abstraites et de variables de classes.

Exemple:

```
public interface Collection{
    boolean add(Object o);
    boolean isEmpty();
    boolean equals(Object o);
    boolean contains(Object o);
    boolean clear();
    Iterator iterator();
...  }
```

10.2 "Implantation" d'une interface

Implanter une interface : Définition d'une classe donnant une définition des méthodes déclarées dans une interface.

Réalisation en Java :

```
Class X extends Object implements Collection { ... }
```

10.3 Interface et types

Une interface définit un nouveau type.

Le type définit par une classe est un sous-type des interfaces qu'elle implémente.

Par exemple, *X* est un sous-type de *Object* et de *Collection*.

10.4 Contraintes pour la définition d'interfaces

Une interface peut étendre une autre interface.

Les méthodes déclarées dans une interfaces sont *abstraites* (inutile de le spécifier), *publiques* et ne peuvent pas être *statiques*.

Les variables déclarées dans une interfaces sont toutes *statiques* et *finales*.

Les classes implémentant une interface doivent définir toutes les méthodes de l'interface ou doivent être déclarées abstraites.

10.5 Exemple : les types *cloneable* et *serializable*

L'interface *cloneable* définit le type de même nom qui permet de de faire référence à des objets pouvant être clonés.

Toute classe implémentant cette interface doit définir ou hériter de la méthode `clone` (ou être déclarée abstraite).

10.6 Un exemple plus complexe - Voir Tutorial

Problème : Faire une applet, définie par la classe `GUIClock`, dans laquelle une horloge se réaffiche toutes les minutes.

Utilisation d'une classe `AlarmClock` qui propose la méthode `letMeSleepFor(unObjet, uneDurée)`.

En envoyant ce message à une instance de cette classe, `unObject` a la garantie qu'au bout de `uneDurée` il recevra le message `wakeUp`.

Question : `UnObject` doit donc posséder une méthode `wakeUp`. Comment assurer ceci au système de contrôle de types?

Réponse : Grâce à un type `Sleeper`, qui déclare la méthode `wakeUp`. Si `unObjet` est de type `Sleeper`, le compilateur sera satisfait.

```
public interface Sleeper {
    public void wakeUp();
    public long ONE_SECOND = 1000;      // in milliseconds
    public long ONE_MINUTE = 60000;     // in milliseconds}
```

La méthode `letMeSleepFor` de la classe `AlarmClock` a la signature suivante :

```
public synchronized boolean
       letMeSleepFor(Sleeper theSleeper, long time) {...}
```

L'applet peut alors être définie ainsi :

```
class GUIClock extends Applet implements Sleeper {
  private AlarmClock clock;
  . . .
  public void wakeUp() {
    repaint();
    clock.letMeSleepFor(this, ONE_MINUTE); } }
```

10.7 Interfaces et héritage multiple

Une interface peut heriter de plusieurs interface.

La relation d'implémentation peut aussi être multiple : une classe peut implémenter plusieurs interfaces.

10.8 Implémentation prédéfinies d'interfaces

Un problème récurent en Java : hériter d'une implantation prédéfinie d'une interface.

Exemple :

Soient l'interface `Volant` déclarant les méthodes `décoller` et `attérir`. Soit la classe `ObjetVolant` implantant l'interface et donnant une définition par défaut de ces méthodes.

Soit à réaliser la classe `Avion`.

- Si Avion n'est sous-classe d'aucune autre classe, en faire une sous-classe de `ObjetVolant`.
- Si Avion est par ailleurs une sous-classe de Véhicule.
  - solution 1 : écrire les méthode `décoller` et `attérir` en recopiant éventuellement le code de celles de `ObjectVolant`
  - solution 2 : (solution suggérée par le créateur de Java) utiliser la composition.
    un avion possède une variable d'instance *aspectAvion* contenant une instance de `ObjetVolant` et la méthode `décoller` implémentée comme suit:

    ```
    décoller() { ... aspectAvion décoller ... }
    ```

Figure 1: Les Interfaces pour les Collections - JDK1.2

Cette solution est fondamentalement non satisfaisante (perte du receveur courant et en conséquense de divers schémas de réutilisation).

11 Bibliothèques réutilisables

11.1 Exemples des Magnitude Smalltalk

Magnitude, Number.

La classe abstraite Magnitude fournit un ensemble de méthodes pour comparer deux objets comparables, donc instance d'une classe, ou éléments d'un ensemble muni d'une relation d'ordre, comme *Date*, *Time* ou *Number*

Exemple de fonctionnalités offertes par `Magnitude`: `<=`, `>=`, `max`, `min`.

Méthode abstraites déclarées sur Magnitude: `<`, `==`, `hash`.

Code de la méthode `max`.

```
public Magnitude max(Magnitude m){
    if (this > m) return (this) else return(m);}
```

La classe Number pourrait conceptuellement être définie comme sous-classe de `Magnitude`.

Fonctionnalités offertes par Number : abs, arcCos, even, odd, modulo, ...

Méthode abstraites déclarées sur Number: *, +, - , /.

11.2 Exemple des Collections Java

La biblithèque des collections de Java intègre un ensemble d'interfaces, un ensemble de classe abstraites qui fournissent des implantations par défaut des interfaces précédentes et des classes concrètes.

11.2.1 Interfaces

Les interfaces définissent les types abstraits fondamentaux de collections.

```
Collection : les collections dans toute leur généralité
    List : les collections ordonnées
    Set : les collections sans doubles
        SortedSet : les collections sans doucles et triées
Map : les objets qui lie des clés à des valeurs (font un mapping)
```

11.2.2 Classes

Les classes abstraites et concrètes issues de ces interfaces :

```
AbstractCollection implements Collection
```

Figure 2: Les Interfaces pour les Collections - JDK1.2

```
    AbstractList implements List
        AbstractSequentialList
            LinkedList
        ArrayList
        Vector
            Stack
    AbstractSet implements Set
        HashSet
        TreeSet
AbstractMap implements Map
    HashMap : une implantation de Map utilisant le hachage
    TreeMap implements SortedMap
            : une implantation utilisant les
              arbres ‘‘rouge - noir’’.
Dictionary
    HashTable implements Map
```

11.2.3 Méthodes

Voici quelques méthodes de ces classes et interfaces mettant en évidence les différents principes : description différentielle, héritage, méthodes paramétrables...

- Interface **Collection**

  ```
  boolean add(Object o),
  boolean remove(Object o),
  int size().
  ```

- Interface **List**

  ```
  boolean add(int index, Object o)
  int indexOf(Object o)
  Object get(int index)
  ```

- Classe **AbstractCollection**

  ```
  ...
  boolean add(Object o)
  abstract int size()
  ```

- Classe **AbstractList**

```
...
boolean add(Object o) : masquage de celle de AbstractCollection
boolean add(int index, Object o)
abstract Object get(int index)
```

- Classe **Vector**

```
...
int size()
Object get(int index)
```

11.3 A propos des Collections Java : les itérateurs

Les itérateurs sont des objets permettant de parcourir une collection ou d'appliquer une opération à tous les éléments d'une collection. L'interface `Iterator` déclare 3 méthodes à cet effet: `hasNext, next et remove`.

Chaque classe de collection définit une méthode `iterator` qui rend une instance d'une classe implémentant l'interface `Iterator`.

Un itérateur s'utilise de la façon suivante, avec un éventuel cast en plus selon l'endroit où est définie la méthode `operation`.

```
Vector V = ...
Iterator I = V.iterator();
while(I.hasNext()){
    I.next().operation();
```

11.4 Frameworks

Framework : Application pré-cablée dans un domaine spécialisé à partir de laquelle il est possible de générer de véritables applications en y ajoutant de nouvelles classes décrivant le contexte particulier de la nouvelle application.

Exemple : un framework pour la réalisation de jeux de balles.