Programmation Orientée Objet – Java

Bertrand Estellon

Département Informatique et Interactions
Aix-Marseille Université

12 novembre 2014

Problématique

Supposons que nous ayons la classe suivante :

```
public class ListSum {
  private int[] list = new int[10];
  private int size = 0;

  public void add(int value) {list[size] = value; size++;}

  public int eval() {
    int result = 0;
    for (int i = 0; i < size; i++)
      result += list[i];
    return result;
  }
}
```

Problématique

Supposons que nous ayons la classe suivante :

```
public class ListProduct {
  private int[] list = new int[10];
  private int size = 0;

  public void add(int value) {list[size] = value; size++;}

  public int eval() {
    int result = 1;
    for (int i = 0; i < size; i++)
      result *= list[i];
    return result;
  }
}
```

Problématique

Il est possible de “refactorer” les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListSum {
  /* Propriétés, constructeur et méthode add. */

  public int eval() {
    int result = neutral();
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]);
    return result;
  }

  private int neutral() { return 0; }
  private int compute(int a, int b) { return a+b; }
}
```

Problématique

Il est possible de “refactorer” les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListProduct {
  /* Propriétés, constructeur et méthode add. */

  public int eval() {
    int result = neutral();
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]);
    return result;
  }

  private int neutral() { return 1; }
  private int compute(int a, int b) { return a*b; }
}
```

Problématique

Après la refactorisation du code :

- seules les méthodes `neutral` et `compute` diffèrent ;
- il serait intéressant de pouvoir supprimer les duplications de code ;

Deux solutions :

- La délégation en utilisant une interface ;
- L'extension et les classes abstraites.

Délégation

Nous allons externaliser les méthodes `neutral` et `compute` dans deux nouvelles classes. Elles vont implémenter l'interface `Operator` :

```
public interface Operator {
  public int neutral();
  public int compute(int a, int b);
}
```

```
public class Sum implements Operator {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}
```

```
public class Product implements Operator {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

Délégation

Les classes `ListSum` et `ListProduct` sont fusionnées dans une unique classe `List` qui délègue les calculs à un objet qui implémente `Operator` :

```
public class List {
 /* Propriétés et méthode add */
 private Operator operator;

 public List(Operator operator){this.operator = operator;}

 public int eval() {
  int result = operator.neutral();
  for (int i = 0; i < size; i++)
   result = operator.compute(result, list[i]);
  return result;
 }
}
```

Délégation

Utilisation des classes ListSum et ListProduct :

```
ListSum listSum = new ListSum();
listSum.add(2); listSum.add(3);
System.out.println(listSum.eval());
ListProduct listProduct = new ListProduct();
listProduct.add(2); listProduct.add(3);
System.out.println(listProduct.eval());
```

Utilisation après la refactorisation du code :

```
List listSum = new List(new Sum());
listSum.add(2); listSum.add(3);
System.out.println(listSum.eval());
List listProduct = new List(new Product());
listProduct.add(2); listProduct.add(3);
System.out.println(listProduct.eval());
```

Classes abstraites

Une classe est abstraite si des méthodes ne sont pas implémentées :

```
public abstract class List {
  private int[] list = new int[10];
  private int size = 0;

  public void add(int value) { list[size] = value; size++; }

  public int eval() {
    int result = neutral(); // util. d'une méthode abstraite
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]); // idem
    return result;
  }

  public abstract int neutral(); // méthode abstraite
  public abstract int compute(int a, int b); // idem
}
```

Classes abstraites et extension

Évidemment, il n'est pas possible d'instancier une classe abstraite :

```
List list = new List(); // erreur
System.out.println(list.eval()) // car que faire ici ?
```

Nous allons étendre la classe `List` afin de récupérer les propriétés et les méthodes de `List` et définir le code des méthodes abstraites :

```
public class ListSum extends List {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}
```

La classe `ListSum` n'est plus abstraite, toutes ses méthodes sont définies :

```
ListSum listSum = new ListSum();
listSum.add(3); listSum.add(7);
System.out.println(listSum.eval());
```

Classes abstraites et extension

Évidemment, il n'est pas possible d'instancier une classe abstraite :

```
List list = new List(); // erreur
System.out.println(list.eval()) // car que faire ici ?
```

Nous allons étendre la classe `List` afin de récupérer les propriétés et les méthodes de `List` et définir le code des méthodes abstraites :

```
public class ListProduct extends List {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

La classe `ListSum` n'est plus abstraite, toutes ses méthodes sont définies :

```
ListProduct listProduct = new ListProduct();
listProduct.add(3); listProduct.add(7);
System.out.println(listProduct.eval());
```

Généralisation aux classes non-abstraites

Plus généralement, l'extension permet de créer une classe en :

- conservant les services (propriétés et méthodes) d'une autre classe ;
- ajoutant de nouveaux services (propriétés et méthodes) ;
- redéfinissant certains services (méthodes).

En Java :

- On utilise le mot-clé `extends` pour étendre une classe ;
- Une classe ne peut étendre qu'une seule classe.

Il est toujours préférable de privilégier l'implémentation à l'extension.

Ajout de nouveaux services

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  public void setPosition(int x, int y) { this.x = x; this.y = y; }
}
```

Il est possible d'ajouter de nouveaux services en utilisant l'extension :

```
public class Pixel extends Point {
  public int r, g, b;

  public void setColor(int r, int g, int b)
    { this.r = r; this.g = g; this.b = b; }
}
```

Ajout de nouveaux services

Les services de `Point` sont disponibles dans `Pixel` :

```
Pixel pixel = new Pixel();
pixel.setPosition(4,8);
System.out.println(pixel.x); // → 4
System.out.println(pixel.y); // → 8
pixel.setColor(200,200,120);
```

Évidemment, les services de `Pixel` ne sont disponibles dans `Point` :

```
Point point = new Point();
point.setPosition(4,8);
System.out.println(point.x); // → 4
System.out.println(point.y); // → 8
point.setColor(200,200,120); // impossible !
```

Redéfinition de méthode

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  public void setPosition(int x, int y) { this.x = x; this.y = y; }
  public void clear() { x = 0; y = 0; }
}
```

Il est possible de redéfinir la méthode `clear` dans `Point` :

```
public class Pixel extends Point {
  public int r, g, b;

  public void setColor(int r, int g, int b)
    { this.r = r; this.g = g; this.b = b; }

  public void clear() { x = 0; y = 0; r = 0; g = 0; b = 0; }
}
```

Le mot-clé super

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  public void setPosition(int x, int y) { this.x = x; this.y = y; }
  public void clear() { setPosition(0, 0); }
}
```

Le mot-clé super permet d'utiliser la méthode `clear` de `Point` :

```
public class Pixel extends Point {
  public int r, g, b;

  public void setColor(int r, int g, int b)
    { this.r = r; this.g = g; this.b = b; }

  public void clear() { super.clear(); setColor(0, 0, 0); }
}
```

Le mot-clé super

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  public void setPosition(int x, int y) { this.x = x; this.y = y; }
}
```

Si la méthode n'a pas été redéfinie, le mot clé `super` est inutile :

```
public class Pixel extends Point {
  public int r, g, b;

  public void setColor(int r, int g, int b)
    { this.r = r; this.g = g; this.b = b; }

  public void clear() { /*super.*/setPosition(0, 0); setColor(0, 0, 0); }
}
```

Les constructeurs et le mot-clé super

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

La classe `Point` n'a pas de constructeur sans paramètre, il faut donc indiquer comment initialiser la partie de `Pixel` issue de la classe `Point` :

```
public class Pixel extends Point {
  public int r, g, b;

  public Pixel(int x, int y, int r, int g, int b) {
    super(x, y); // appel du constructeur de Point
    this.r = r; this.g = g; this.b = b;
  }
}
```

Les constructeurs

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  public Point() { x = 0; y = 0; }
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Par défaut, le constructeur sans paramètre est appelé :

```
public class Pixel extends Point {
  public int r, g, b;

  public Pixel(int r, int g, int b) {
    // appel du constructeur sans paramètre de Point
    this.r = r; this.g = g; this.b = b;
  }
}
```

Les constructeurs

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  //public Point() { x = 0; y = 0; }
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Ici, vous devez préciser les paramètres du constructeur avec `super` :

```
public class Pixel extends Point {
  public int r, g, b;

  public Pixel(int r, int g, int b) {
    // erreur de compilation (aucun constructeur sans paramètre)
    this.r = r; this.g = g; this.b = b;
  }
}
```

Transtypages et polymorphisme

Aucune méthode ou propriété ne peut être supprimée lors d'une extension. Par exemple, `Pixel` possède toutes les propriétés et méthodes de `Point` (même si certaines méthodes ont pu être redéfinies).

Par conséquent, l'upcasting est toujours autorisé :

```
Point point = new Pixel();
point.setPosition(2,4);
System.out.println(point.x + " " + point.y);
point.clear();
```

Remarques :

- Le code a exécuté lors d'un appel de méthode est déterminé à l'exécution en fonction de la référence présente dans la variable.
- Le typage des variables permet de vérifier à la compilation l'existence des propriétés et des méthodes.

La classe Object

Par défaut, les classes étendent la classe `Object` de Java. Par conséquent, l'upcasting vers la classe `Object` est toujours possible :

```
Pixel pixel = new Pixel();
Object object = pixel;
Object[] array = new Object[10];
for (int i = 0; i < t; i++) {
  if (i%2==0) array[i] = new Point();
  else array[i] = new Pixel();
}
```

Notez que `object.setPosition(2,3)` n'est pas autorisé dans le code précédent car la classe `Object` ne possède pas la méthode `setPosition` et seul le type de la variable compte pour déterminer si l'appel d'une méthode ou l'utilisation d'une propriété est autorisé.

La méthode toString() de la classe Object

Par transitivité de l'extension, toutes les méthodes et propriétés de la classe `Object` sont disponibles sur toutes les instances :

```
Object object = new Object(); Point point = new Point(2,3);
System.out.println(object.toString());
                              // ↪ java.lang.Object@19189e1
System.out.println(point.toString());
                              // ↪ test.Point@7c6768
```

La méthode `toString` est utilisée par Java pour convertir une référence en chaîne de caractères :

```
Object object = new Object(); Point point = new Point(2,3);
String string = object+" ;"+point ;
System.out.println(string);
// ↪ java.lang.Object@19189e1;test.Point@7c6768
```

La méthode toString() de la classe Object

Évidemment, il est possible de redéfinir la méthode `toString` :

```
public class Point {
  public int x, y;

  public Point(int x, int y) { this.x = x; this.y = y; }

  public String toString() { return "("+x+","+y+")"; }
}
```

Le polymorphisme fait que cette méthode est appelée si la variable contient une référence vers un `Point` :

```
Point point = new Point(2,3); Object object = point;
System.out.println(point);  // → (2,3)
System.out.println(object); // → (2,3)
```

Extension d'interface

Supposons que nous ayons l'interface suivante :

```
public interface List {
  public int size();
  public void get(int index);
}
```

En Java, il est également possible d'étendre une interface :

```
public interface ModifiableList extends List {
  public void add(int value);
  public void remove(int index);
}
```

Une classe qui implémente l'interface `ModifiableList` doit implémenter les méthodes `size`, `get`, `add` et `remove`.

Extension d'interface

Supposons que la classe `ArrayModifiableList` implémente l'interface `ModifiableList`. Dans ce cas, nous pouvons écrire :

```
ModifiableList modifiableList = new ArrayModifiableList();
modifiableList.add(2);
modifiableList.add(5);
modifiableList.remove(0);
List list = modifiableList;
System.out.println(list.size());
```

En revanche, il n'est pas possible d'écrire :

```
list.remove(0);
  /* ↪ Cette méthode n'existe pas */
  /*    dans l'interface List. */
```

Extension d'interface

Supposons que nous avons l'interface suivante :

```
public interface Printable {
  public void print();
}
```

En Java, une interface peut étendre plusieurs interfaces :

```
public interface ModifiablePrintableList
                        extends ModifiableList, Printable {
}
```

Notons que nous ne définissons pas de nouvelles méthodes dans l'interface `ModifiablePrintableList`. Cette interface ne représente que l'union des interfaces `ModifiableList` et `Printable`. Bien évidemment, de nouvelles méthodes auraient pu être définies dans `ModifiablePrintableList`.