Interfaces ou classes abstraites ?

Alexis Nasr (d'après les slides de Arnaud Labourel)

Classe ListSum

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;
  }
}
```

Classe ListProduct

Supposons que nous ayons aussi la classe suivante (très similaire) :

```
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;
  }
}
```

Refactoring pour isoler la répétition

Les deux classes sont très similaires et il y a de la répétition de code.

Il est possible de *refactorer* (réécrire le code) les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListSum {
  /* attributs, constructeur et méthode add. */
  private int neutral() { return 0; }
  private int compute(int a, int b) { return a+b; }
  public int eval() {
    int result = neutral();
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]);
    return result;
  }
}
```

Refactoring pour isoler la répétition

Il est possible de *refactorer* les classes précédentes de façon à isoler les différences dans les méthodes `neutral()` et `compute()`:

```
public class ListProduct {
  /* attributs, constructeur et méthode add. */
  private int neutral() { return 1; }
  private int compute(int a, int b) { return a*b; }
  public int eval() {
    int result = neutral();
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]);
    return result;
  }
}
```

Comment éviter la répétition ?

Après la *refactorisation* du code :

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

Deux solutions :

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

Section 1

Délégation + Interface

Interface Operator

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 en une unique classe `List` qui **délègue** les calculs à un objet qui implémente l'interface `Operator` :

```
public class List {
  /* attributs 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 de la classe `ListSum`:

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

Utilisation après la *refactorisation* du code :

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

Section 2

Extension + classe abstraite

Classe abstraite List

```
public abstract class List {
  private int[] list = new int[10];
  private int size = 0;
  public void add(int value) {
    list[size] = value; size++;
  }
  public abstract int neutral(); // méthode abstraite
  public abstract int compute(int a, int b); // idem
  public int eval() {
    int result = neutral();
    // utilisation d'une méthode abstraite
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]); // idem
    return result;
  }
}
```

Extension de la classe List

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

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

Classes abstraites et extension

Les classes `ListSum` et `ListProduct` ne sont plus abstraites, toutes leurs méthodes sont définies :

- les méthodes `add` et `eval` sont définies dans `List`. Les classes `ListSum` et `ListProduct` héritent du code de ces méthodes.
- les méthodes `neutral` et `compute` qui étaient abstraites dans `List` sont définies dans `ListSum` et `ListProduct`.

On peut donc les instancier :

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

Différences

Délégation

- Une seule classe : `List`
- Une interface : `Operator`
- Deux classes implémentant `Operator` : `Sum` et `Product`
- La classe `List` délègue le calcul à un `Operator`

Classes abstraites

- Deux classes : `ListSum` et `ListProduct`
- Une classe abstraite : `List`
- Les classes `ListSum` et `ListProduct` étendent la classe `List`