Conseils de programmation : héritage, mutabilité

Théorie et pratique de la programmation
Michel Schinz – 2013-05-06

Erreurs du débutant

Les programmeurs débutants commettent fréquemment deux erreurs lorsqu'ils programment dans un langage orienté-objets comme Java :

1. ils utilisent l'héritage de manière excessive,
2. ils définissent ou utilisent trop souvent des données modifiables.

Le but de cette leçon est d'examiner les problèmes liés à l'héritage et aux données modifiables et de définir des règles simples liées à leur utilisation.

Héritage

Ensemble comptant

On désire ajouter à un ensemble `HashSet` la possibilité de compter le nombre total d'éléments ajoutés.
Comment faire ?
Une première idée serait de définir une sous-classe de `HashSet` redéfinissant les méthodes `add` et `addAll` pour compter les éléments ajoutés.
Examinons cette solution.

Ensemble comptant v1

```
public class CountingHashSet<E>
    extends HashSet<E> {
  private int addCount = 0;

  @Override
  public boolean add(E e) {
    /* ??? */
  }
  @Override
  public boolean addAll(Collection<?
                        extends E> c) {
    /* ??? */
  }
}
```


Ensemble comptant v1

Une fois la classe terminée, on écrit un test unitaire :

```
@Test
public void testCount() {
  CountingHashSet<Integer> s =
    new CountingHashSet<>();
  s.addAll(Arrays.asList(1, 2, 3, 4, 5));
  for (int i = 6; i <= 10; ++i)
    s.add(i);
  assertEquals(10, s.addCount());
}
```

Question : pensez-vous que ce test réussit ou échoue ?

Ensemble comptant v1

Avec la version actuelle de la classe `HashSet` de Oracle, ce test échoue car `addAll` utilise `add` en interne...

Mais attention :

- rien ne garantit que ce soit le cas avec d'autres mises en œuvre de `HashSet`, et
- rien ne garantit que ce sera toujours le cas à l'avenir.

Le fait que `addAll` utilise `add` est un détail de mise en œuvre interne à la classe qui ne devrait pas être visible de l'extérieur ! Ceci illustre le problème principal de l'héritage, à savoir qu'il casse l'encapsulation.

Question : voyez-vous une meilleure solution ici ?

Ensemble comptant v2

Une meilleure solution consiste à appliquer le patron *Decorator* en définissant un décorateur d'ensemble qui compte les ajouts.

Ensemble comptant v2

```
public class CountingSetDecorator<E>
    implements Set<E> {
  private final Set<E> s;
  private int addCount = 0;
  public CountingSetDecorator(Set<E> s) {
    this.s = s; }
  @Override
  public int size() { return s.size(); }
  @Override
  public boolean add(E e) {
    addCount += 1;
    return s.add(e);
  }
  …
}
```


Ensemble comptant v2

La nouvelle mise en œuvre des ensembles comptants passe le test avec succès :

```
@Test
public void testCount() {
  CountingSetDecorator<Integer> s =
    new CountingSetDecorator<>(
      new HashSet<>());
  s.addAll(Arrays.asList(1, 2, 3, 4, 5));
  for (int i = 6; i <= 10; ++i)
    s.add(i);
  assertEquals(10, s.addCount());
}
```

Autre avantage : ce décorateur s'applique à n'importe quel type d'ensemble, pas seulement `HashSet` !

Séparation des soucis

Le décorateur de l'ensemble comptant fait deux choses à la fois :

1. il compte le nombre d'éléments ajoutés à l'ensemble,
2. il transmet la plupart des messages à l'ensemble décoré.

Il est toujours bien de séparer autant que possible les « soucis » (*separation of concerns* en anglais).

Question : comment peut-on séparer ces deux aspects ici ?

Ensemble comptant v3

Solution : en séparant la mise en œuvre en deux classes :

1. `ForwardingSet`, un décorateur d'ensemble par défaut, qui ne fait rien d'autre que transmettre les messages à l'ensemble décoré.
2. `CountingSetDecorator`, qui hérite de `ForwardingSet` et redéfinit uniquement les méthodes `add` et `addAll` afin de compter les éléments ajoutés.

ForwardingSet

```
public abstract class ForwardingSet<E>
    implements Set<E> {
  private Set<E> s;
  public ForwardingSet(Set<E> s) {
    this.s = s;
  }

  @Override
  public int size() { return s.size(); }
  @Override
  public boolean isEmpty() {
    return s.isEmpty();
  }
  …
}
```


Ensemble comptant v3

```
public final class CountingSetDecorator<E>
    extends ForwardingSet<E> {
  private int addCount = 0;
  public CountingSetDecorator(Set<E> s) {
    super(s);
  }
  @Override
  public boolean add(E e) {
    addCount += 1;
    return super.add(e);
  }
  // … addAll similaire
  public int addCount() { return addCount; }
}
```


Comparaison des solutions

Comment se fait-il qu'hériter de `ForwardingSet` soit une bonne idée, alors qu'hériter de `HashSet` en est une mauvaise ?

La différence cruciale est que `ForwardingSet` a été conçue pour servir de classe mère, ce qui n'est pas le cas de `HashSet`, conçue pour être utilisée comme « productrice d'objets » !

Cela se voit entre autres dans la documentation de `HashSet`, qui ne mentionne p.ex. pas le fait que `addAll` utilise `add` en interne.

Rôle des classes

De manière générale, une classe ne peut jouer correctement qu'un seul des deux rôles suivants :

- celui de créateur d'objets (classe « instantiable »),
- celui de super-classe (classe « héritable »).

Malheureusement, les langages actuels ne permettent pas de faire clairement la différence entre ces deux rôles.

De plus, par défaut toute classe peut jouer les deux rôles et c'est au programmeur d'interdire explicitement l'un ou l'autre usage, ce qu'il ne fait généralement pas.

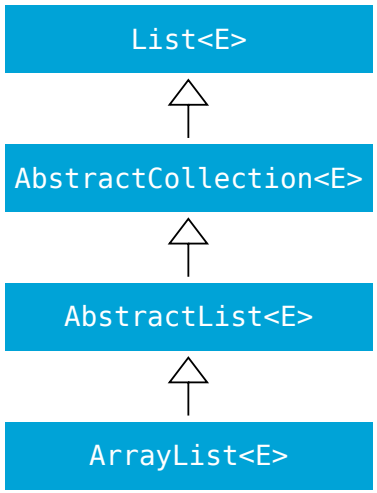
Classe instantiable

Une classe instantiable est une classe dont le (seul) rôle est de permettre la création d'objets.
Pour empêcher son utilisation dans le rôle de super-classe, il est conseillé d'empêcher la définition de sous-classes en la déclarant finale.

Classe héritable

Une classe héritable est une classe dont le (seul) rôle est de servir de super-classe.
Pour empêcher son utilisation en tant que productrice d'objets, il est conseillé de la déclarer abstraite, même si elle ne l'est pas réellement, c-à-d même si elle ne possède pas de méthode abstraite. C'est p.ex. ce qui a été fait avec `ForwardingSet`.

Exemple : collections Java



Interface : décrit le *concept* de liste.

Classe héritable : fournit des mises en œuvre par défaut de plusieurs méthodes (p.ex. `addAll` en termes de `add`).

Classe héritable (similaire à `AbstractCollection`).

Classe instantiable (et actuellement aussi héritable, malheureusement)

Conseil : héritage

Conseil : lorsque vous écrivez une classe, décidez s'il s'agit d'une classe héritable ou instantiable. Rendez-la abstraite dans le premier cas, finale dans le second.

Mutabilité

Classe des dates

On désire concevoir une classe représentant les dates dans le calendrier grégorien. On la commence ainsi :

```
public final class GregorianDate {
  private int year, month, day;

  public GregorianDate(int year,
                       int month,
                       int day) {
    this.year = year;
    this.month = month;
    this.day = day;
  }
  …
}
```


Egalité des dates

Pour définir l'égalité entre deux dates, deux possibilités s'offrent à nous :

1. ne pas redéfinir `equals`, auquel cas l'identité des objets est utilisée, ou
2. redéfinir `equals` pour faire une comparaison **structurelle**, c-à-d utilisant le contenu des champs jour, mois et année et ignorant l'identité des objets.

Quelle solution est la meilleure ?

Egalité des dates

Evidemment, l'égalité sur les dates doit être structurelle ! Il faut donc redéfinir `equals` :

```
public final class GregorianDate {
  …
  @Override
  public boolean equals(Object that) {
    return that instanceof GregorianDate
      && year == ((GregorianDate)that).year
      && month == ((GregorianDate)that).month
      && day == ((GregorianDate)that).day;
  }
}
```

Hachage des dates

Rappel : `equals` et `hashCode` doivent *toujours* être compatibles, c-à-d que si deux objets sont égaux au sens de `equals`, leur code de hachage doit être le même (mais l'inverse n'est pas forcément vrai !).
Cela nous force à redéfinir également `hashCode`, p.ex. de la manière suivante :

```
public final class GregorianDate {
  …
  @Override
  public int hashCode() {
    return ((year << 4) | month) << 5 | day;
  }
}
```


Ordre des dates

Les dates sont naturellement ordonnées, il est donc logique que notre classe implémente l'interface `Comparable` :

```
public final class GregorianDate
    implements Comparable<GregorianDate> {
  …
  @Override
  public int compareTo(GregorianDate o) {
    // ???
  }
}
```


Accesseurs

Pour donner accès aux champs, on ajoute ensuite des accesseurs (*getters*) :

```
public final class GregorianDate {
  …
  public int year() { return year; }
  public int month() { return month; }
  public int day() { return day; }
}
```


« Mutateurs » ?

Question : faut-il maintenant également ajouter des « mutateurs » (*setters*) pour modifier les attributs ?

```
public final class GregorianDate {
  …
  public void setYear(int year) {
    this.year = year;
  }
  public void setMonth(int month) {
    this.month = month;
  }
  public void setDay(int day) {
    this.day = day;
  }
}
```

Dates mutables ?

Rendre les dates mutables implique que leur valeur de hachage (retournée par `hashCode`) dépend de l'état. Cela induit des comportements très contre-intuitifs – même s'ils sont logiques – des collections lorsqu'on modifie des dates après stockage :

```
Set<GregorianDate> set = new HashSet<>();
GregorianDate date =
  new GregorianDate(1912, 6, 23);
set.add(date);
date.setYear(2013);
set.contains(date); // false !!!
```


Dates mutables ?

Bien entendu, la situation n'est pas meilleure avec les collections utilisant l'ordre plutôt que le hachage :

```
Set<GregorianDate> set = new TreeSet<>();
GregorianDate date =
  new GregorianDate(1912, 6, 23);
set.add(date);
for (int y = 1800; y <= 2000; ++y)
    set.add(new GregorianDate(y, 6, 23));
date.setYear(2013);
set.contains(date); // false !!!
```

(Notez au passage que l'invariant de l'arbre binaire de recherche utilisé par `TreeSet` est violé par notre faute !)

Dates mutables ?

Finalement, rendre les dates mutables forcerait les clients de notre classe à faire des copies défensives des dates reçues. Exemple :

```
public final class Person {
  …
  private final GregorianDate birthdate;
  public Person(…, GregorianDate bdate) {
    …
    this.birthdate = new GregorianDate(
      bdate.year(),
      bdate.month(),
      bdate.day());
  }
  …
}
```


Copie défensive

Ces copies défensives forcées ont plusieurs inconvénients majeurs :

- Elles compliquent le travail de *tous* les utilisateurs de la classe, qui peuvent être très nombreux (pensez aux classes de l'API Java, utilisées par des milliers de programmeurs dans le monde).
- Il est facile de les oublier, ce qui produit des bugs dont l'origine est souvent très difficile à identifier.
- Elles impliquent généralement l'ajout de constructeurs de copie et/ou la mise en œuvre de la méthode `clone` dans la classe mutable, ce qui la complique.

Dates modifiables ?

Conclusion : les instances de la classe `GregorianDate` ne doivent pas être mutables.

```
public final class GregorianDate {
  private final int year, month, day;

  public void setYear(int year) {
    this.year = year;
  }
  public void setMonth(int month) {
    this.month = month;
  }
  public void setDay(int day) {
    this.day = day;
  }
}
```


Classes mutables ?

Le même genre de raisonnement s'applique à toutes les classes pour lesquelles l'égalité est naturellement structurelle, p.ex. :

- toutes les classes modélisant des entités mathématiques : nombres complexes, vecteurs, matrices, points, lignes, polynômes, etc.
- une classe modélisant les couleurs,
- une classe modélisant les chaînes de caractères (`String`),
- etc.

Toutes ces classes doivent donc être immutables !

Collections mutables ?

La décision de rendre une classe mutable ou non est plus compliquée dans d'autres situations.

Par exemple, les classes des collections doivent-elles être mutables ou non ? La réponse est moins claire dans ce cas et dépend généralement de l'utilisation que l'on fait de la collection.

Les bibliothèques standard de différents langages ne s'accordent pas sur ce point, certaines offrant uniquement des collections mutables (p.ex. Java), d'autres uniquement des collections immutables (p.ex. Haskell) et d'autres les deux (p.ex. Scala).

Cohérence avec l'égalité

Quel que soit le choix effectué quant à la mutabilité des instances d'une classe, il importe d'être cohérent :

> Les instances d'une classe immutables doivent être comparées structurellement, celles d'une classe mutable doivent l'être par identité.

Malheureusement, beaucoup de classes ne respectent pas cette règle, p.ex. celles des collections de l'API Java, qui sont modifiables mais comparées de manière structurelle !

D'autres classes de cette API souffrent du même problème, p.ex. les classes géométriques du paquetage `java.awt.geom` : `AffineTransfom`, `Point`, etc.

Autres problèmes

En plus de ceux déjà mentionnés, les classes mutables souffrent d'autres problèmes :

- elles interagissent très mal avec la concurrence avec mémoire partagée (*shared memory concurrency*), qui est le modèle utilisé par Java,
- il est difficile de raisonner à leur sujet en raison du (souvent grand) nombre d'états dans lequel elles peuvent se trouver.

Conseil : mutabilité

Conseil : par défaut, définissez des classes immutables.
Pour ce faire, procédez ainsi :

- Attachez l'attribut `final` à tous les champs.
- Ne fournissez aucun « mutateur » (*setter*).
- Assurez-vous que les éventuelles valeurs mutables contenues dans les champs ne sont pas accessibles par d'autres.

Bien entendu, dans certains cas il peut être utile de définir des classes mutables. Dans ce cas, songez à utiliser le patron *Builder* si une version immutable de la classe a un sens.

Construction

Lors de la définition d'une classe immutable, prenez garde à ne jamais stocker dans un champ (immutable) une valeur mutable fournie en argument au constructeur sans la copier en profondeur.
En effet, rien ne garantit qu'une telle valeur n'est pas accessible (et donc modifiable ultérieurement) par d'autres.

Construction

Par exemple, un constructeur d'une classe de matrices (immutable, bien entendu) qui accepte un tableau en argument *doit* le copier en profondeur :

```
public final class Matrix {
  private final double[][] m;

  public Matrix(double[][] m) {
    double[][] m2 = new double[m.length][];
    for (int i = 0; i < m.length; ++i)
      m2[i] = m[i].clone();
    this.m = m2;
  }
}
```

Résumé

Conseils concernant l'héritage :

- lorsque vous écrivez une classe, décidez s'il s'agit d'une classe héritable ou instantiable. Rendez-la abstraite dans le premier cas, finale dans le second,
- n'héritez que de classes qui sont clairement conçues comme héritables, et appliquez le patron *Decorator* dans les autres cas.

Conseil concernant la mutabilité :

- définissez si possible des classes immutables,
- utilisez une notion d'égalité (et d'ordre) cohérente avec la mutabilité,
- si les versions mutables et immutables d'une classe sont utiles, appliquez le patron *Builder*.