Patrons : *Adapter, Decorator, Composite*

Pratique de la programmation orientée-objet

Michel Schinz – 2015–05–04

Patron n° 6 :

Adapter

Illustration du problème

La classe `Collections` de la bibliothèque Java possède une méthode statique permettant de mélanger les éléments d'une liste :

```
public static void shuffle(List<?> list)
```

Est-il possible d'utiliser cette méthode pour mélanger un tableau Java ?

Directement, cela n'est clairement pas possible, les tableaux n'étant pas des listes. Mais existe-t-il un moyen indirect d'y parvenir ?

Solution

Pour permettre l'utilisation d'un tableau Java là où une liste est attendue, il suffit d'écrire une classe qui adapte le tableau en le présentant comme une liste.
Cette classe doit implémenter l'interface `List` du paquetage `java.util` et effectuer les opérations de cette interface directement sur le tableau qu'elle adapte.

Adaptateur de tableaux

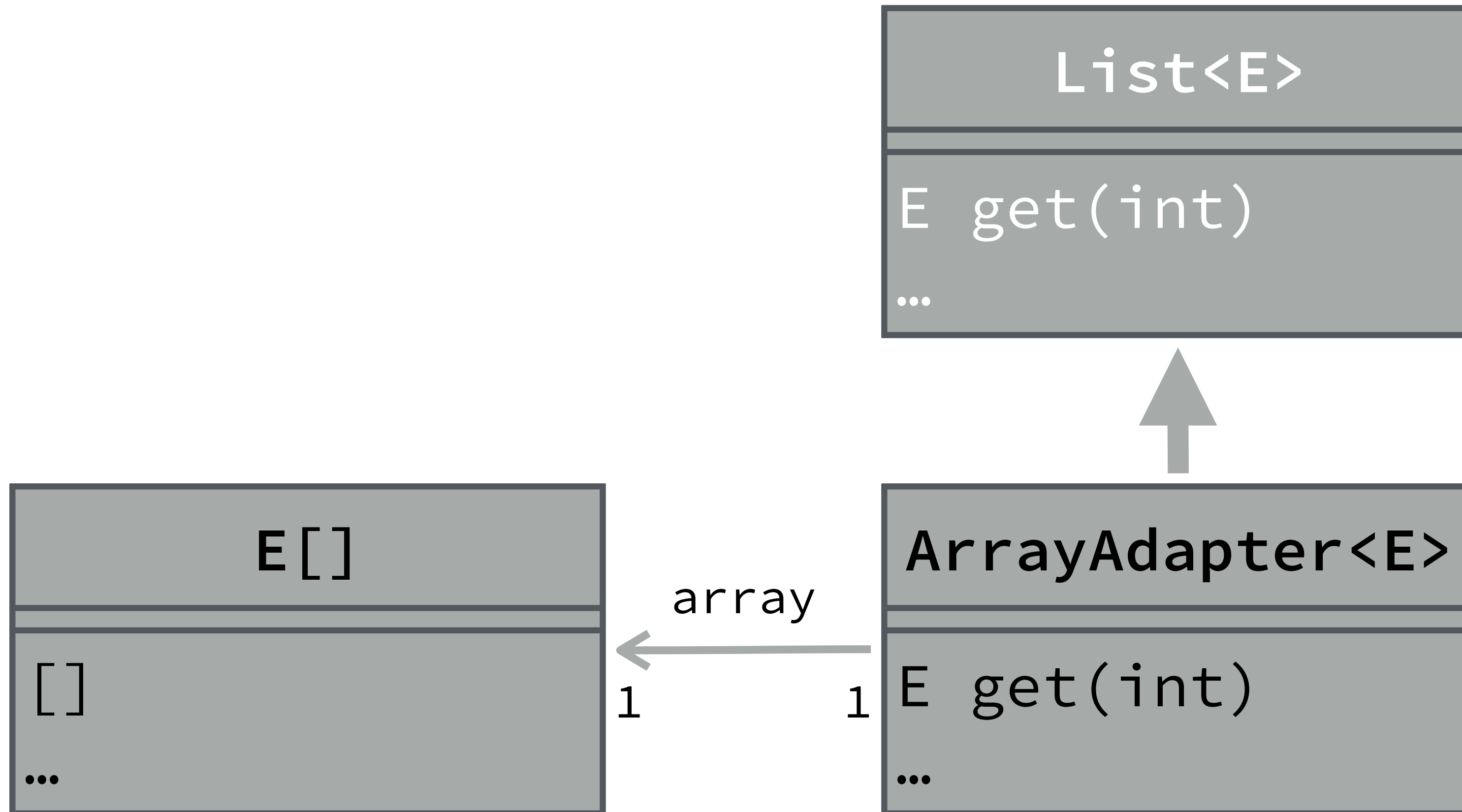
```
public final class ArrayAdapter<E>
    implements List<E> {
  private final E[] array;
  public ArrayAdapter(E[] array) {
    this.array = array;
  }
  public E get(int i) { return array[i]; }
  public E set(int i, E e) {
    E curr = array[i];
    array[i] = e;
    return curr;
  }
  // … les 21 autres méthodes de List
}
```

Utilisation de l'adaptateur

Une fois l'adaptateur défini, il est possible de l'utiliser pour mélanger un tableau au moyen de la méthode `shuffle` :

```
String[] array = …;
List<String> adaptedArray =
  new ArrayAdapter<String>(array);
// mélange les éléments de array
Collections.shuffle(adaptedArray);
```

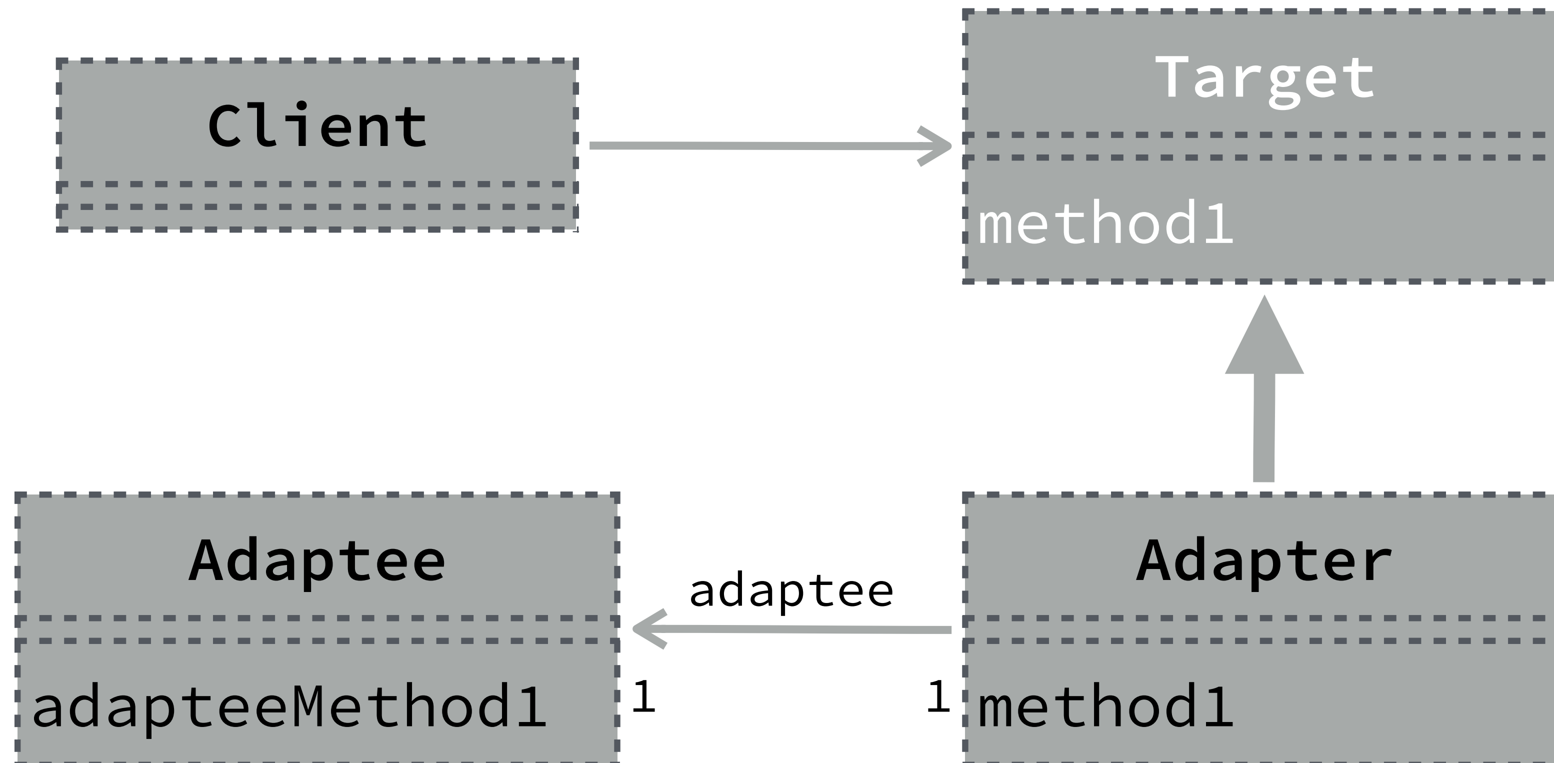
Diagramme de classes



Généralisation

De manière générale, lorsqu'on désire utiliser une instance d'une classe *A* là où une instance d'une classe *B* est attendue, il est possible d'écrire une classe servant d'adapteur.
Bien entendu, il faut que le comportement des classes *A* et *B* soit relativement similaire, sans quoi l'adaptation n'a pas de sens.
Le patron de conception *Adapter* décrit cette solution.

Diagramme de classes



Exemples réels : collections

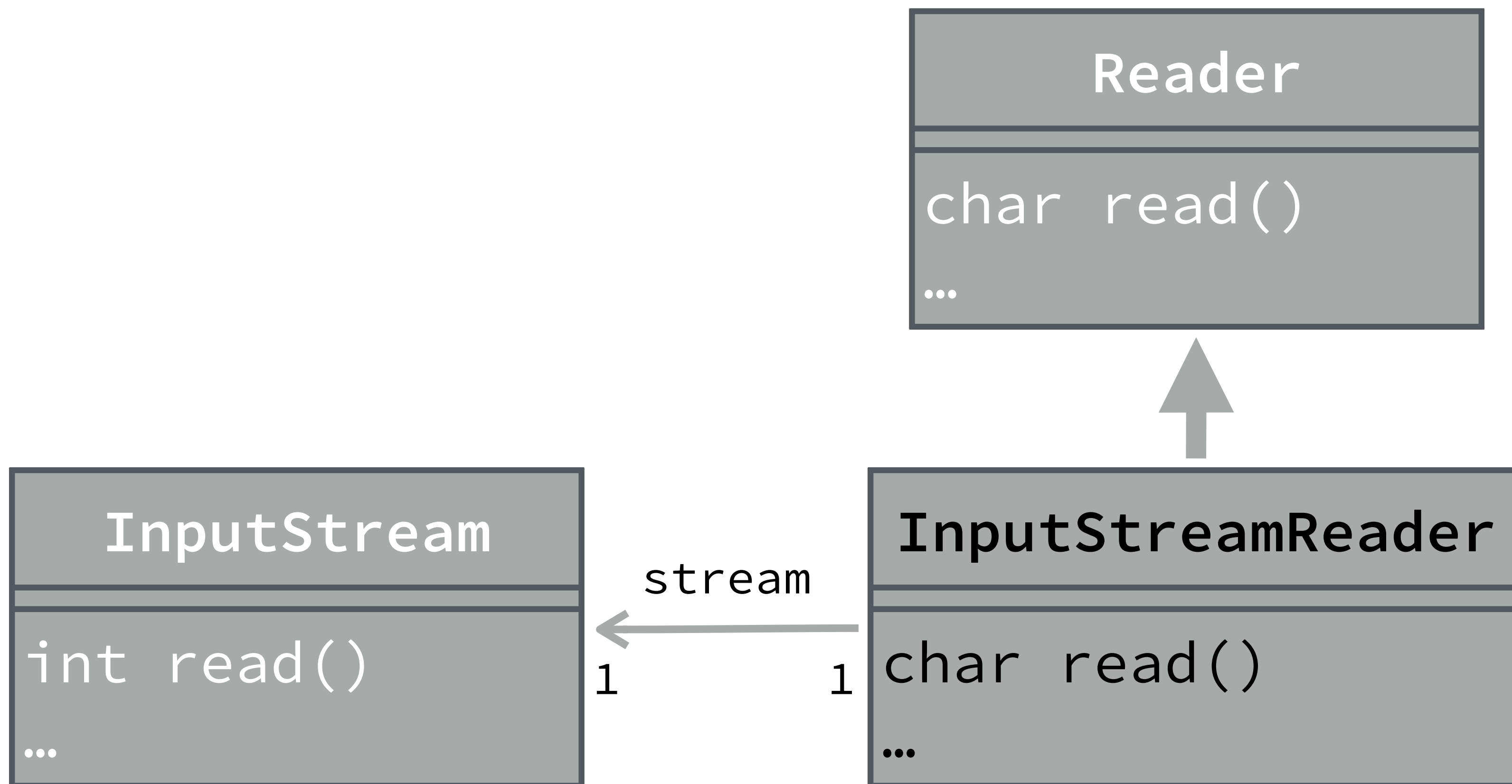
La méthode `asList` de la classe `Arrays` utilise le patron *Adapter* pour adapter un tableau en liste, exactement comme nous l'avons fait.

La classe `HashSet` peut être vue comme un adaptateur qui adapte une table associative de type `HashMap` pour en faire un ensemble. Il en va de même avec `TreeSet`, qui adapte `TreeMap`.

La méthode `asLifoQueue` de la classe `Collections` adapte un deque (de type `Deque<T>`) pour en faire une queue (de type `Queue<T>`).

Exemple réel : E/S

La classe `InputStreamReader` adapte un flot d'entrée d'octets (`InputStream`) pour en faire un lecteur (`Reader`), étant donné un encodage de caractères.



Patron n° 7 :

Decorator

Illustration du problème

On désire écrire un programme de dessin vectoriel 2D, permettant de dessiner et manipuler différentes formes géométriques. Celles-ci sont représentées par une interface Shape et plusieurs mises en œuvre, une pour chaque forme de base.

```
public interface Shape {
  public boolean contains(Point p);
  // … autres méthodes
}
public final class Circle implements Shape {
  …
}
// … Polygon, etc.
```

Illustration du problème

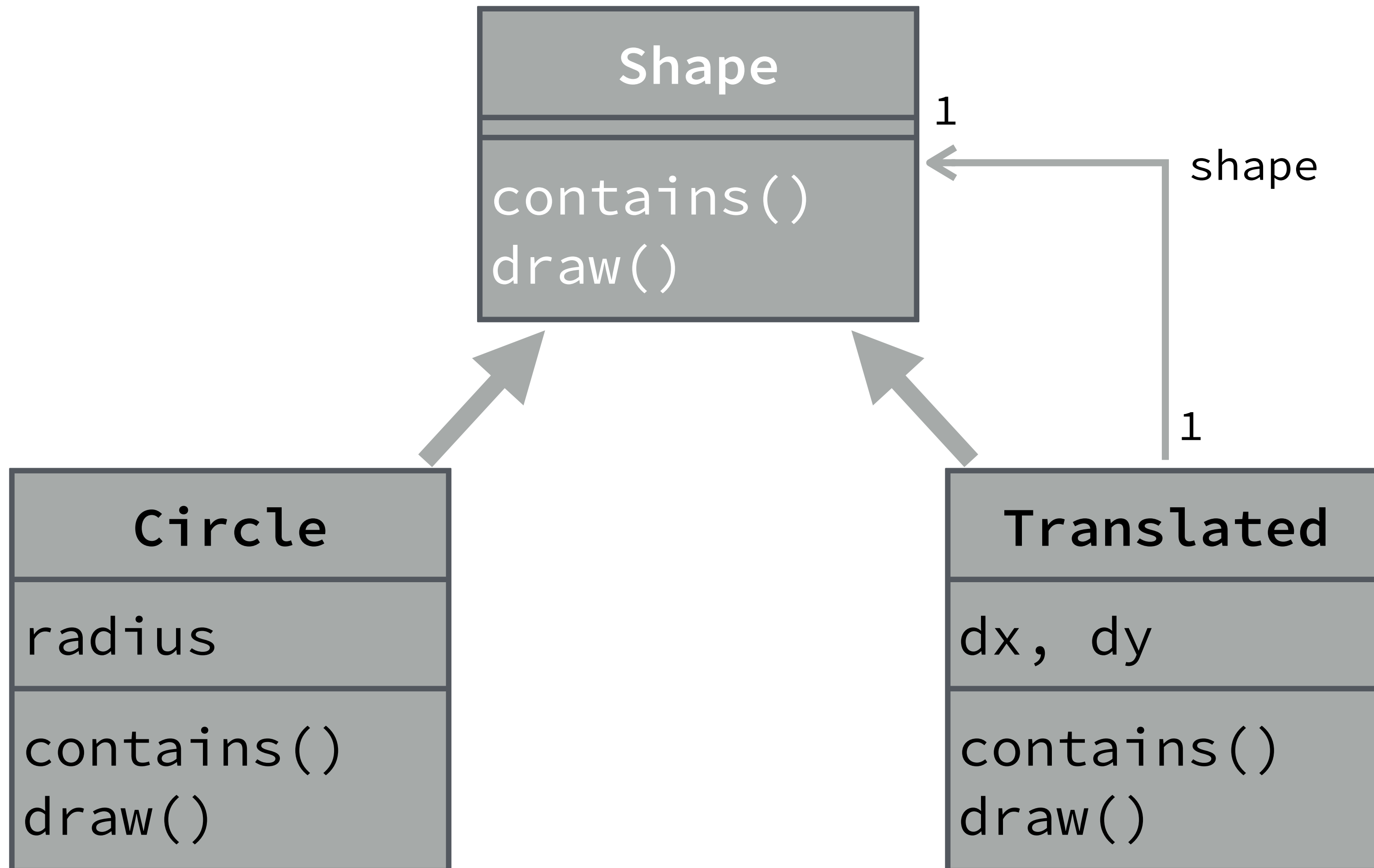
On désire offrir la possibilité d'appliquer différentes transformations géométriques aux formes de base : translation, rotation, symétrie, etc.
Comment faire ?

Solution

Une solution consiste à définir des pseudo-formes qui en transforment d'autres. Par exemple pour la translation :

```
public final class Translated
    implements Shape {
  private final Shape shape;
  private final double dx, dy;
  public Translated(…) { … }
  public boolean contains(Point p) {
    return s.contains(
      new Point(p.x() - dx, p.y() - dy));
  }
  // … autres méthodes
}
```

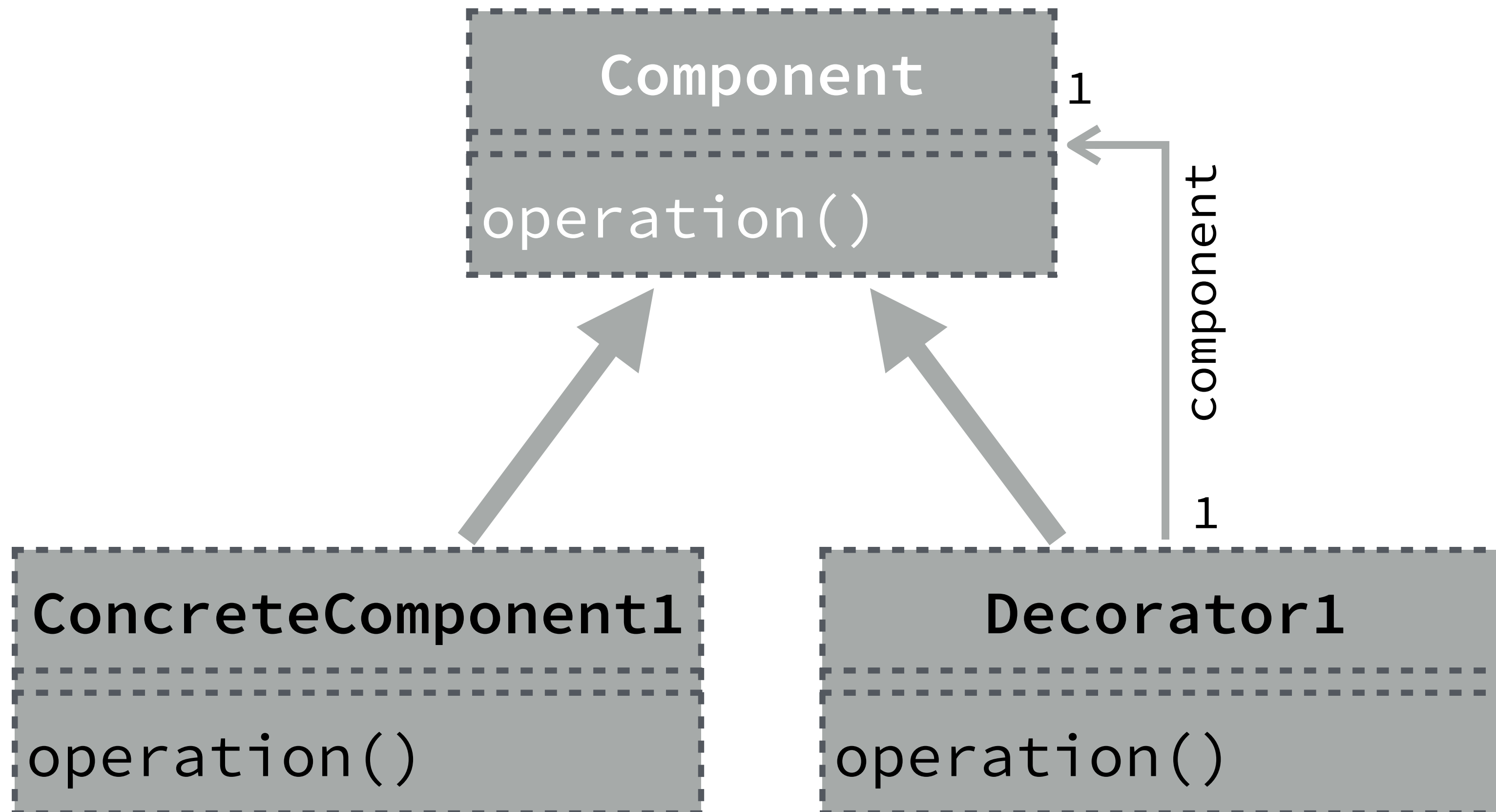
Diagramme de classes



Généralisation

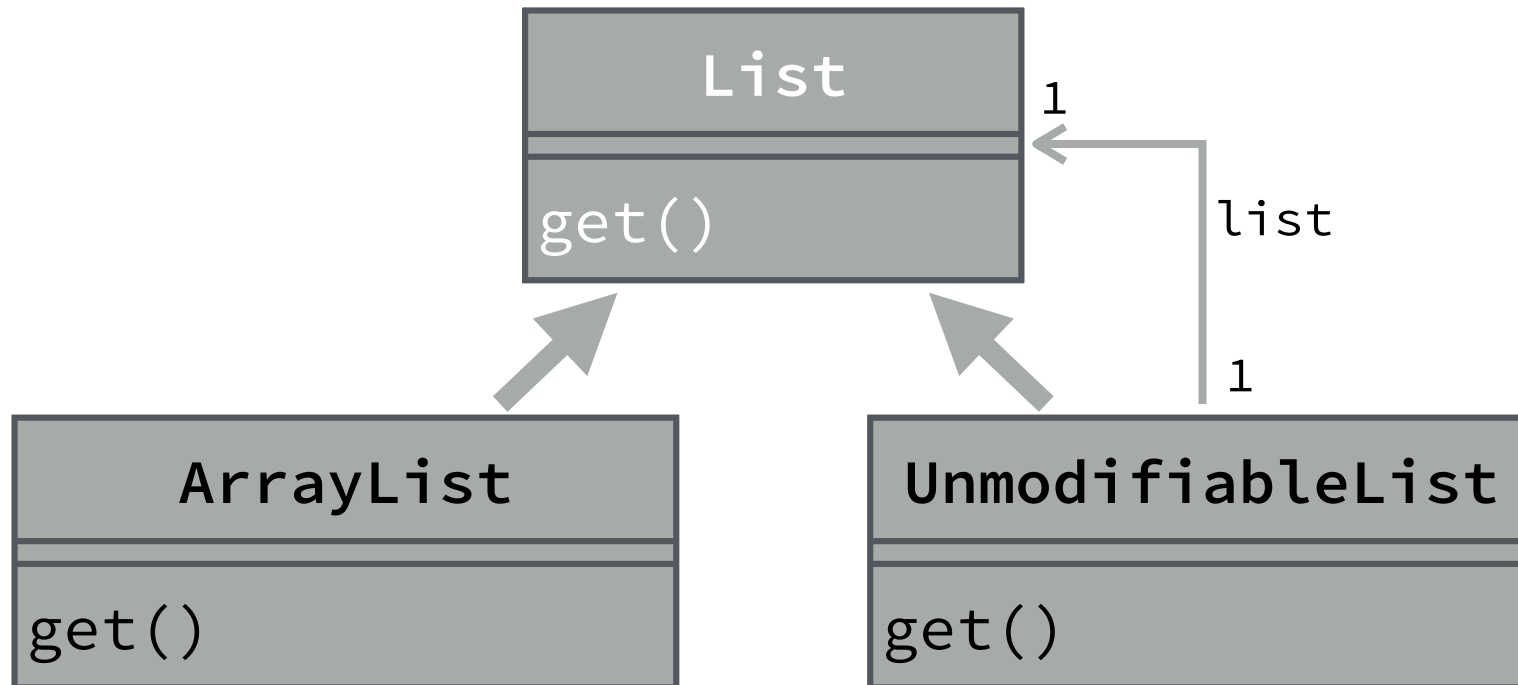
Chaque fois que l'on désire changer le comportement d'un objet sans changer son interface, on peut « l'emballer » dans un objet ayant la même interface mais un comportement différent. L'objet « emballant » laisse l'objet emballé faire le gros du travail, mais modifie son comportement lorsque cela est nécessaire.
Cette solution est décrite par le patron *Decorator*.

Diagramme de classes



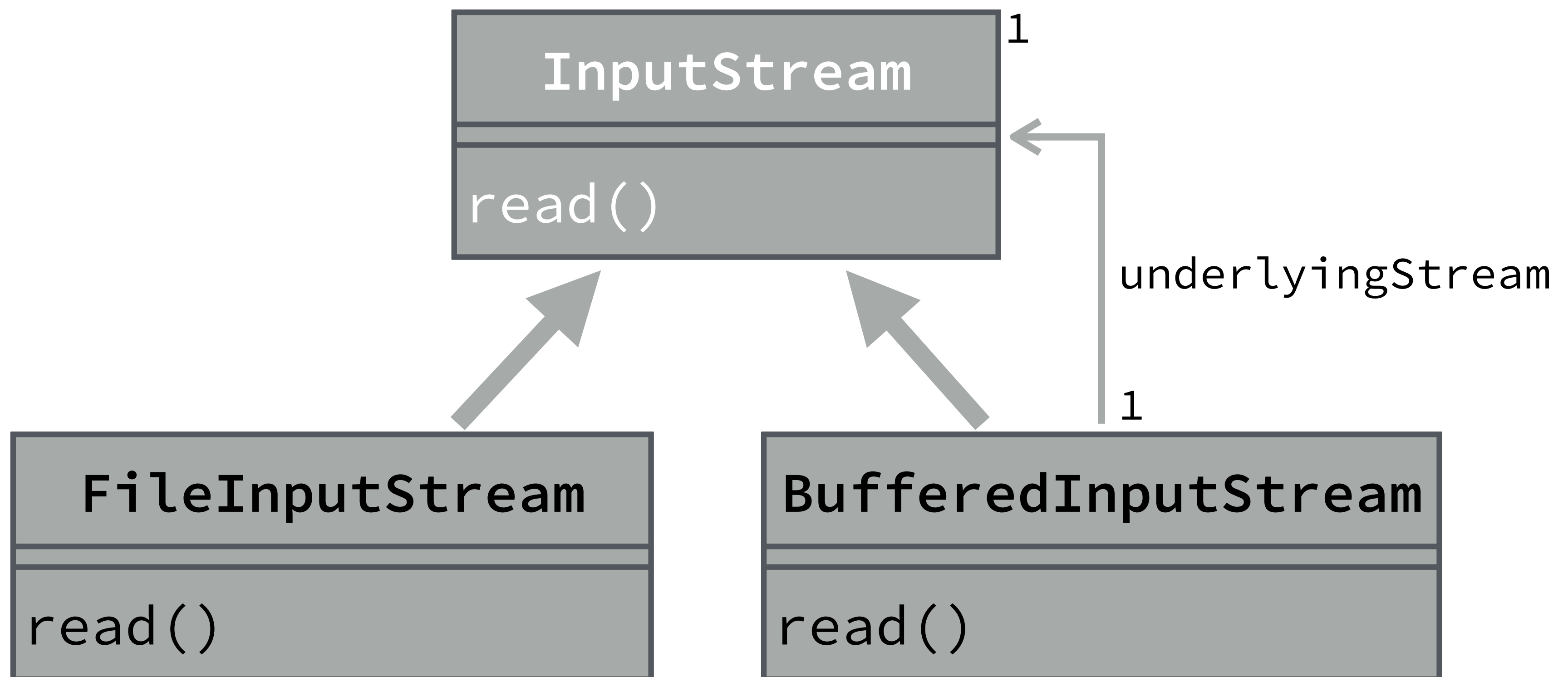
Exemple réel : collections

La bibliothèque Java offre plusieurs méthodes permettant d'obtenir des vues sur des (parties de) collections, p.ex. `subList` dans `List`, `unmodifiableList` dans `Collections`, etc. Les classes mettant en œuvre ces vues sont des décorateurs.



Exemple réel : E/S

Les flots filtrants Java ne sont rien d'autre que des décorateurs de flots. Par exemple, `BufferedInputStream` est un décorateur ajoutant une mémoire tampon au flot sous-jacent.



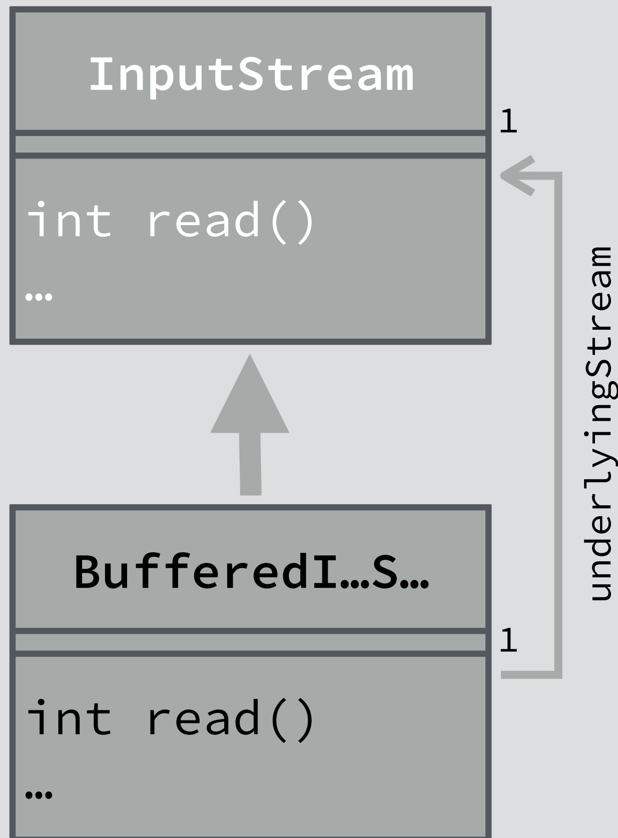
Exemple réel : GUI

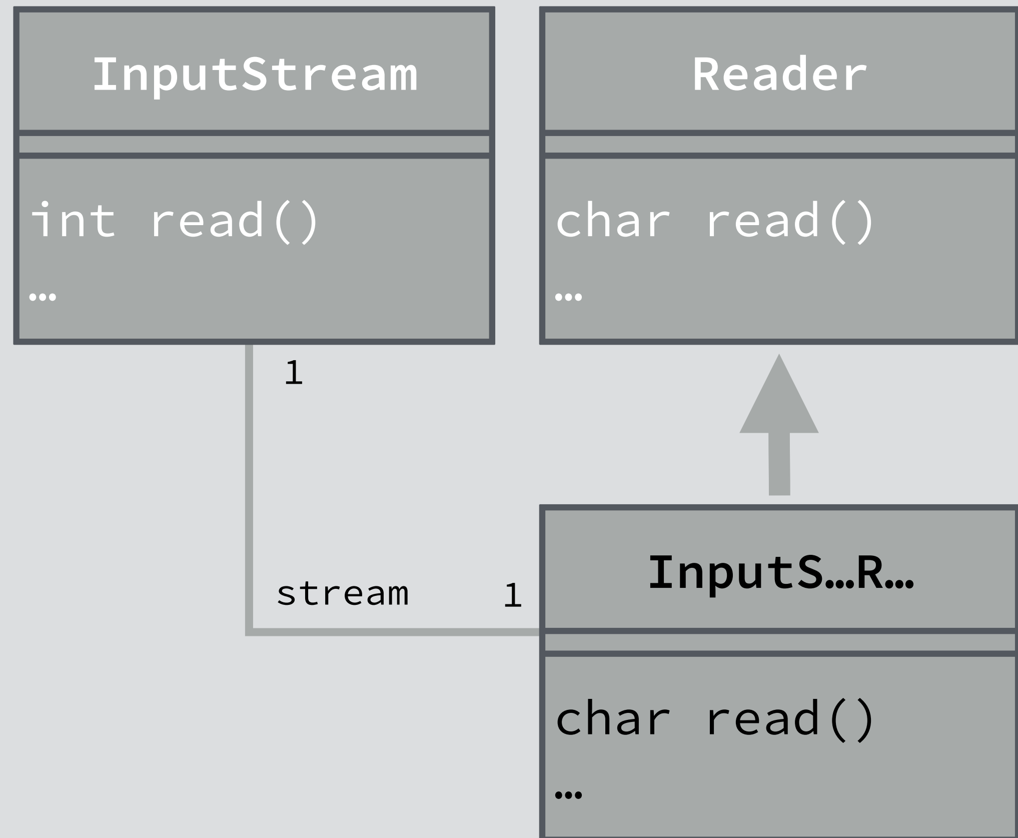
Les décorateurs sont souvent utilisés dans les bibliothèques de gestion d’interfaces graphiques pour ajouter des ornements aux éléments graphiques : bordures, barres de défilement, etc.
Le nom *Decorator* vient de cette utilisation.

Decorator / Adapter

Les patrons *Decorator* et *Adapter* sont très proches et l'un comme l'autre sont parfois désignés par le nom *Wrapper*.
La différence entre les deux est qu'un décorateur a le même type que l'objet qu'il décore, alors qu'un adaptateur a un type différent – c'est son but !
La différence peut se voir p.ex. en comparant les diagrammes de classes de `BufferedInputStream` (un décorateur) et de `InputStreamReader` (un adaptateur).

Decorator / Adapter





Patron n° 8 : Composite

Illustration du problème

On désire améliorer le programme de dessin vectoriel 2D pour offrir à l'utilisateur la possibilité de grouper plusieurs formes. Bien entendu, un groupe doit se comporter comme une forme normale, à laquelle il est possible d'appliquer des transformation, ou qu'il est possible de placer dans un groupe plus grand.

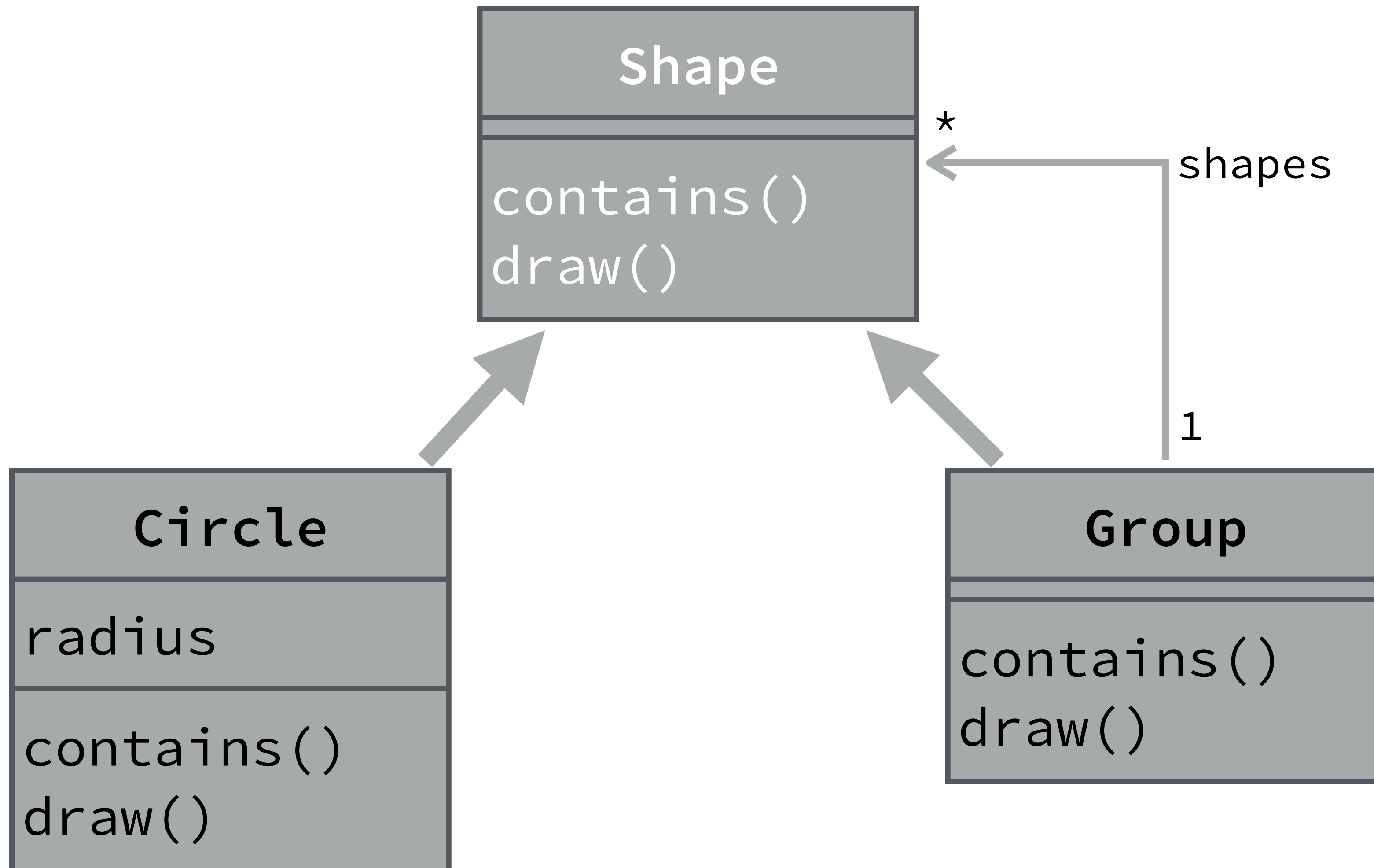
Comment faire ?

Solution

Une solution consiste à définir une pseudo-forme qui représente un groupe :

```
public final class Group implements Shape {
  private final List<Shape> shapes;
  public Group(List<Shape> shapes) { … }
  public boolean contains(Point p) {
    for (Shape s: shapes)
      if (s.contains(p))
        return true;
    return false;
  }
  // … autres méthodes
}
```

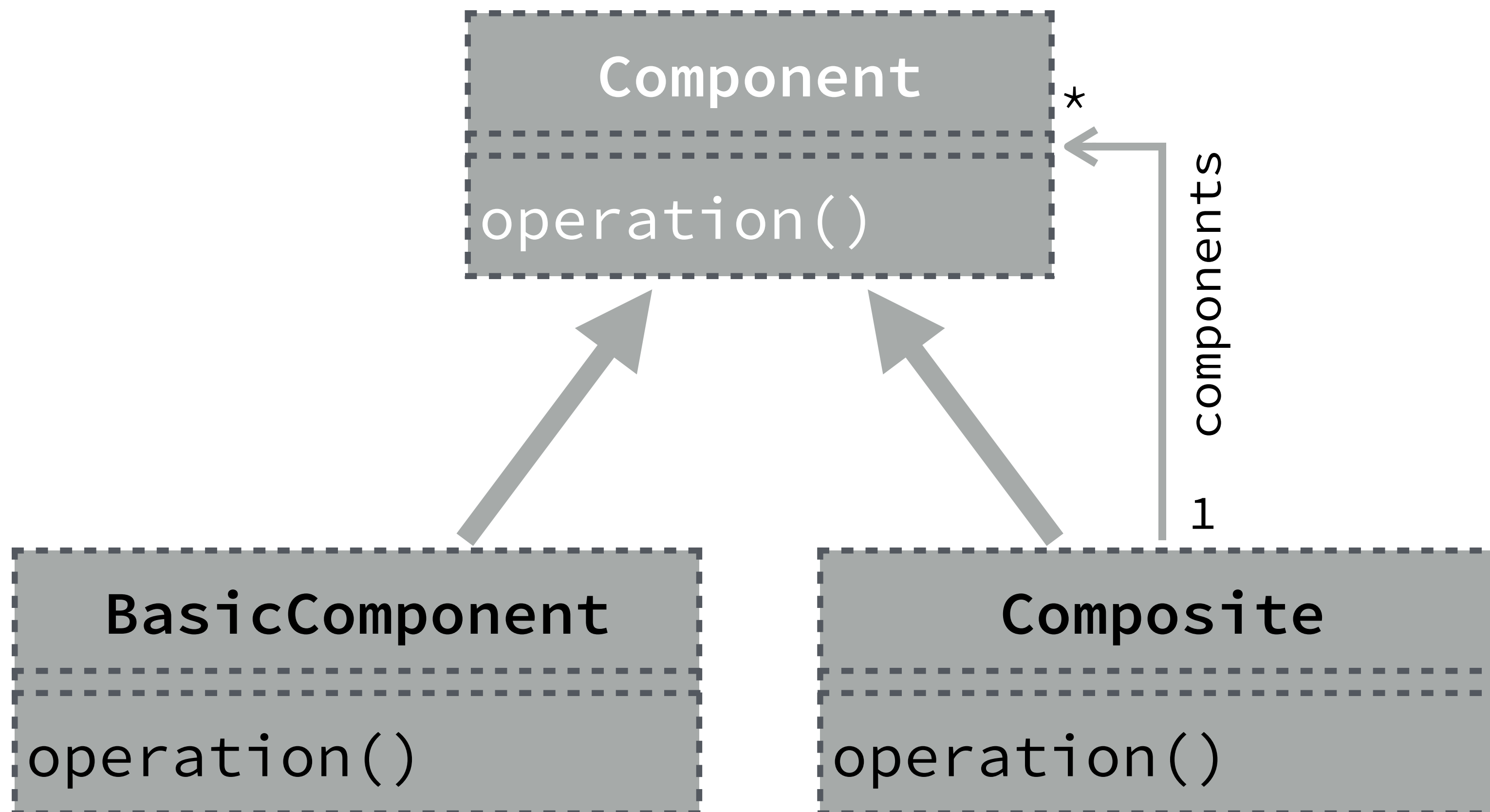
Diagramme de classes



Généralisation

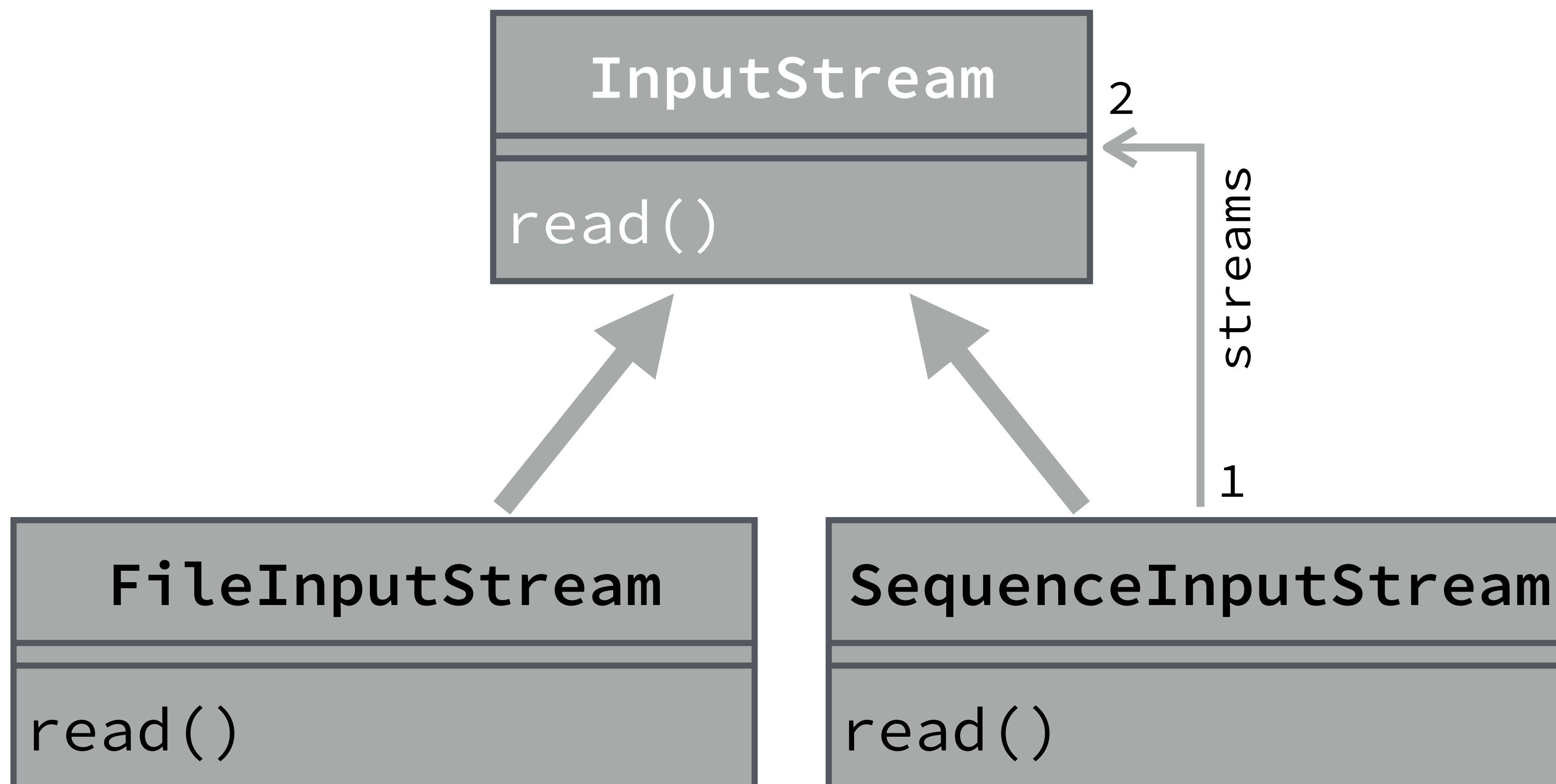
Lorsque les objets d'un programme peuvent être composés entre eux pour former des macro-objets, il est judicieux de faire en sorte que ceux-ci aient le même type que les objets qu'ils composent. De la sorte, il est possible de traiter les objets et les macro-objets de manière uniforme.
Cela implique entre autres que les macro-objets peuvent être composés d'autre macro-objets, de manière récursive.
Cette idée est décrite par le patron *Composite*.

Diagramme de classes



Exemple réel : E/S Java

La classe `SequenceInputStream` prend deux flots d'entrée et produit un flot composite qui fournit d'abord les valeurs du premier puis celles du second.



Exemple réel : GUI

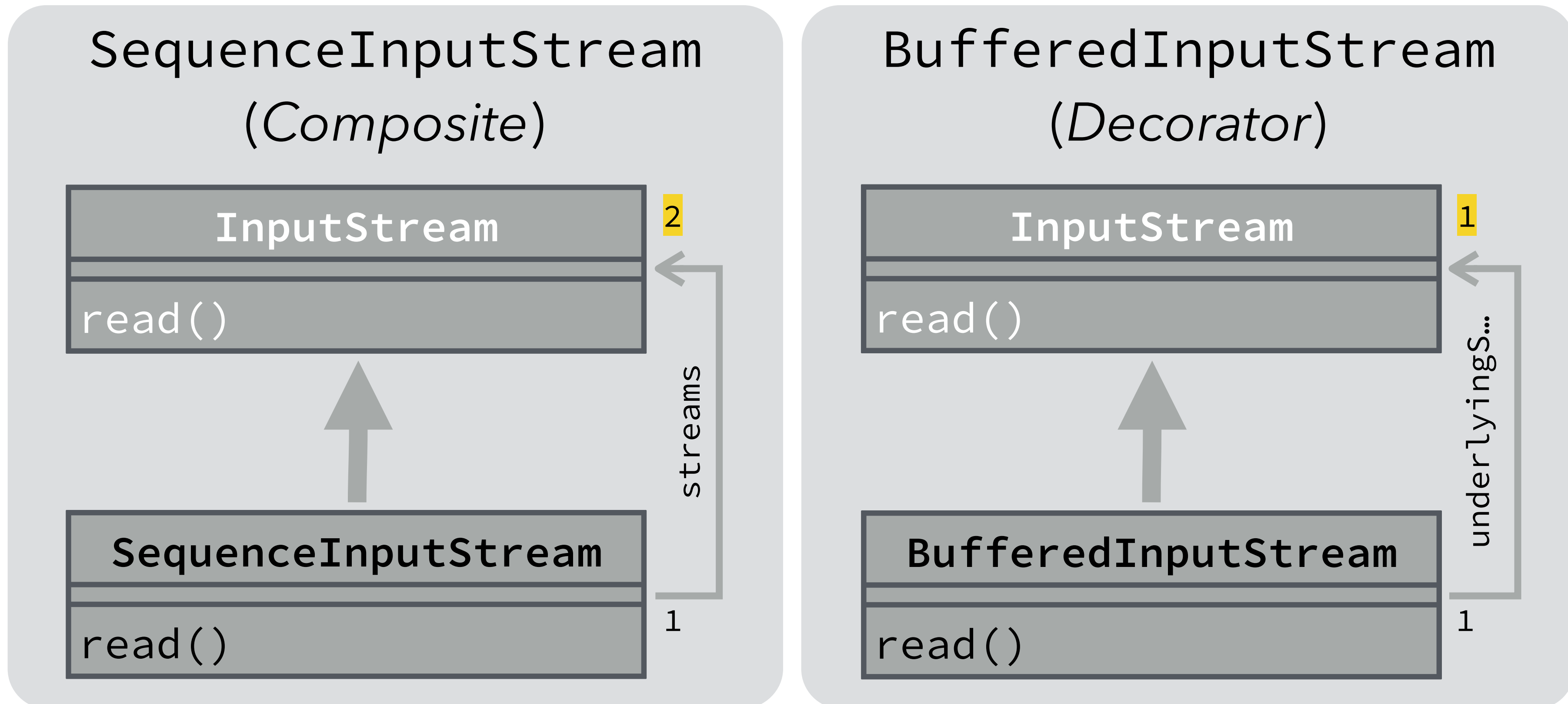
Les interfaces graphiques sont construites en combinant des composants de base – boutons, champs textuels, menus, etc.

Certains composants sont des conteneurs, c-à-d que leur rôle principal est de contenir d'autres composants. Par exemple, un panneau à onglets permet d'organiser une interface complexe en plusieurs onglets séparés.

Les composants conteneurs sont eux aussi des composants, et il est dès lors possible de placer des conteneurs dans d'autres conteneurs.

Composite / *Decorator*

La différence entre *Composite* et *Decorator* est minime et se résume au fait que le premier référence plusieurs objets de son propre type, le second un seul.

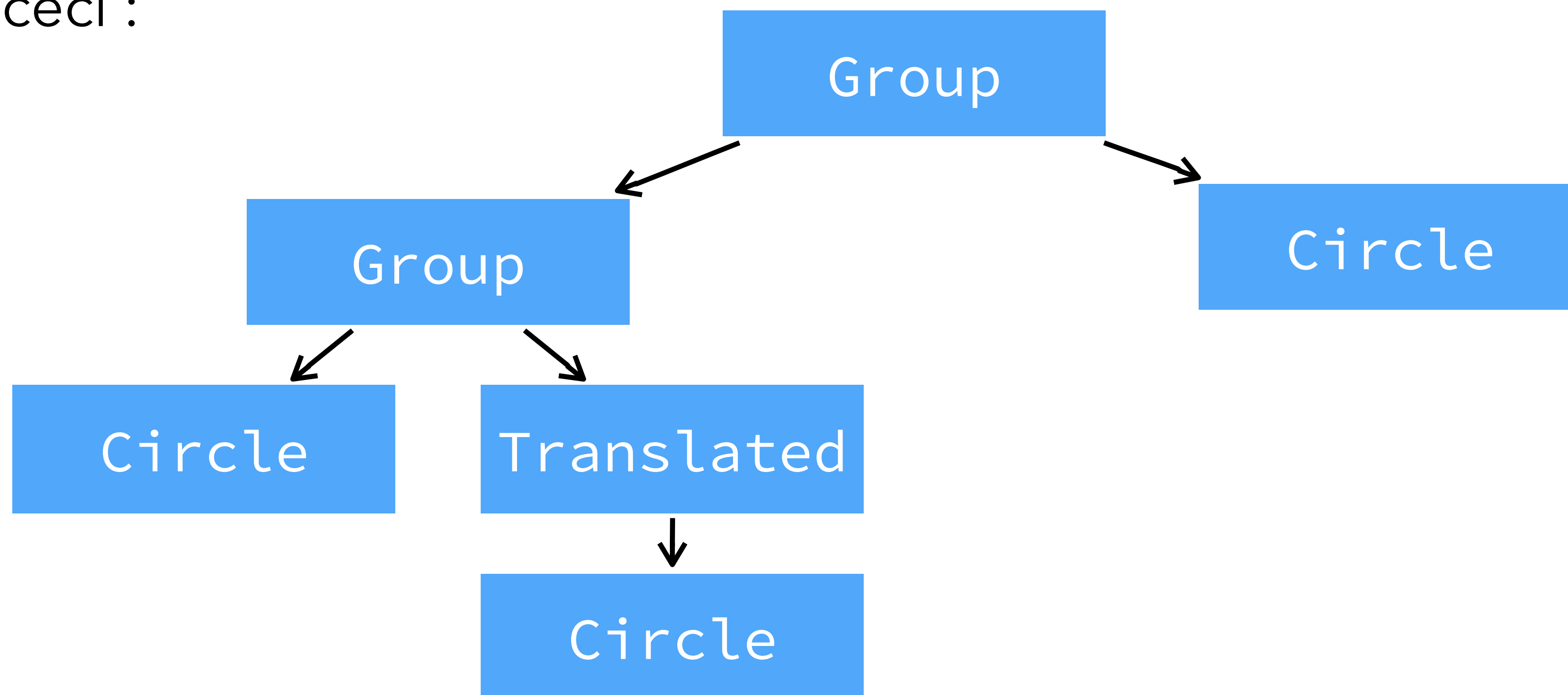


Decorator + Composite

Les patrons *Decorator* et *Composite* sont très souvent utilisés ensemble pour permettre la création d'objets complexes par composition – au sens large, incluant la décoration – d'objets plus simples.
Cette approche, très puissante, est parfois qualifiée de **compositionnelle** ou d'**algébrique**. En effet, la manière dont les décorateurs et composites permettent d'obtenir de nouveaux objets à partir d'objets existants rappelle la manière dont les opérateurs algébriques – la négation, l'addition, etc. – permettent d'obtenir de nouveaux objets mathématiques à partir d'objets existants.

Hiérarchie en arbre

En mémoire, les objets résultant d'une telle organisation forment une structure en arbre dont les feuilles sont les objets de base et les nœuds les décorateurs – s'ils ont un seul fils – ou les composites – s'ils en ont plusieurs.
Par exemple, un dessin vectoriel pourrait ressembler à ceci :



Lambdas, *Decorator* et *Composite*

Lambdas et *Decorator*

Lorsque le type à décorer est une interface fonctionnelle, les lambda expressions couplées au méthodes par défaut permettent de définir très facilement des décorateurs, sans vraiment s'en apercevoir.
Pour s'en convaincre, transformons petit à petit une classe nommée `SLC` qui compare des chaînes par longueur. Elle implémente une version simplifiée de l'interface `Comparator`, dont elle hérite une méthode par défaut – `reversed` – qui inverse le comparateur. Cette dernière définit un décorateur, même si cela n'est pas évident au premier abord.

Comparateur de chaînes

```
interface Comparator<T> {
  public int compare(T o1, T o2);
  public default Comparator<T> reversed() {
    return (o1, o2) -> this.compare(o2, o1);
  }
}
class SLC implements Comparator<String> {
  public int compare(String o1, String o2) {
    return Integer.compare(o1.length(),
                           o2.length());
  }
}
```

1 : héritage

Dans un premier temps, la méthode par défaut `reversed` de `Comparator`, héritée par la classe `SLC`, peut y être copiée :

```
class SLC implements Comparator<String> {
  public int compare(String o1, String o2) {
    return Integer.compare(…);
  }
  public Comparator<String> reversed() {
    return (o1, o2) -> this.compare(o2, o1);
  }
}
```

(L'interface `Comparator` ne change pas et n'est plus présentée dans ce qui suit).

2 : « délambdaisation »

La lambda expression étant équivalente à une création d'instance de classe anonyme, on peut la traduire ainsi en prenant soin d'adapter la référence à `this` :

```
class SLC implements Comparator<String> {
   public int compare(String o1, String o2)
     { return Integer.compare(…); }
   public Comparator<String> reversed() {
     return new Comparator<String>() {
       public int compare(String o1, String o2)
         { return SLC.this.compare(o2, o1); }
     };
   }
}
```

3 : désanonymisation

La classe anonyme peut être nommée RC et devenir ainsi une classe intérieure (c-à-d imbriquée mais non statique) :

```
class SLC implements Comparator<String> {
  public int compare(String o1, String o2)
    { return Integer.compare(…); }
  public Comparator<String> reversed()
    { return new RC(); }
  private class RC
      implements Comparator<String> {
    public int compare(String o1, String o2)
      { return SLC.this.compare(o2, o1); }
  }
}
```

4 : imbrication statique

Une classe intérieure (imbriquée non statique) peut être transformée en une classe imbriquée statique, pour peu qu'on lui fournisse une référence à l'instance de la classe englobante à laquelle elle est attachée.

4 : imbrication statique

```
class SLC implements Comparator<String> {
  public int compare(String o1, String o2)
    { return Integer.compare(…); }
  public Comparator<String> reversed()
    { return new RC(this); }
  private static class RC
      implements Comparator<String> {
    private final SLC outerThis;
    public RC(SLC outerThis)
      { this.outerThis = outerThis; }
    public int compare(String o1, String o2)
      { return outerThis.compare(o2, o1); }
  }
}
```

5 : désimbrication

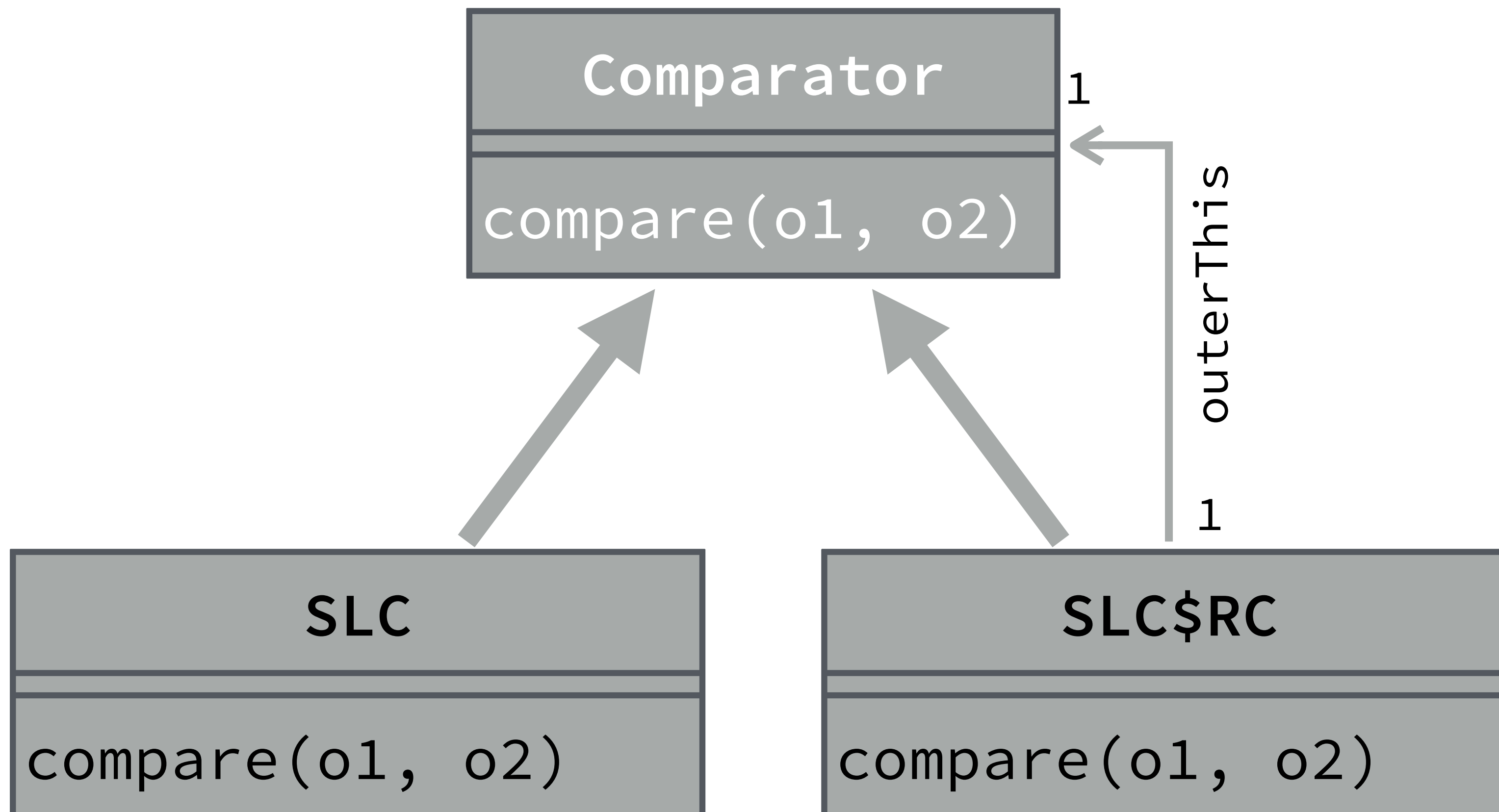
Une classe imbriquée statiquement étant – à quelques détails près – équivalente à une classe non imbriquée, on peut transformer la classe RC en une classe de niveau supérieur.
Pour éviter tout conflit de nom avec une autre classe imbriquée portant le même nom, on la renomme au passage en SLC$RC.

5 : désimbrication

```
class SLC implements Comparator<String> {
  public int compare(String o1, String o2)
    { return Integer.compare(…); }
  public Comparator<String> reversed()
    { return new SLC$RC(this); }
}
class SLC$RC implements Comparator<String> {
  private final SLC outerThis;
  public SLC$RC(SLC outerThis)
    { this.outerThis = outerThis; }
  public int compare(String o1, String o2)
    { return outerThis.compare(o2, o1); }
}
```

Diagramme de classes

En dessinant le diagramme de classes de cette dernière version, on s'aperçoit qu'on a bien affaire à un décorateur.



Lambdas et *Composite*

Les lambda expressions permettent aussi de très facilement mettre en œuvre le patron *Composite*, comme le fait p.ex. la méthode `thenComparing` :

```
interface Comparator<T> {
  public int compare(T o1, T o2);
  public default Comparator<T> thenComparing(
      Comparator<T> that) {
    return (o1, o2) -> {
      int c = this.compare(o1, o2);
      return c != 0 ? c : that.compare(o1, o2);
    };
  }
}
```

La « preuve » est laissée en exercice.

Decorator et héritage

Ensemble comptant

Admettons que l'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 final class CountingHashSet<E>
    extends HashSet<E> {
  private int addCount = 0;

  @Override
  public boolean add(E e) {
    // ???
  }
  @Override
  public boolean addAll(Collection<E> c) {
    // ???
  }
  public int addCount() { return addCount; }
}
```

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` d'Oracle, ce test échoue car `addAll` utilise `add` en interne :

```
boolean addAll(Collection<? extends E> c) {
  boolean modified = false;
  for (E e : c)
    if (add(e))
      modified = true;
  return modified;
}
```

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.

Ensemble comptant (v1)

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 : 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 final 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` !

Héritage / *Decorator*

Quelle est la différence cruciale entre la sous-classe et le décorateur ?

La sous-classe `CountingHashSet` a redéfini la méthode `add`, et donc la méthode appelée par `addAll` :

```
boolean addAll(Collection<? extends E> c) {
  … for (E e : c) … add(e); …
}
```

qui est maintenant celle de `CountingHashSet`.

Par contre, le décorateur n'a pas redéfini la méthode `add`, donc l'appel dans `addAll` fait toujours référence à la même méthode, celle de `HashSet`.

Séparation des soucis

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

- il compte le nombre d'éléments ajoutés à l'ensemble,
- 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 :

- `ForwardingSet`, un décorateur d'ensemble par défaut, qui ne fait rien d'autre que transmettre les messages à l'ensemble décoré.
- `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 :

1. celui de créateur d'objets (classe « instantiable »),
2. 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.

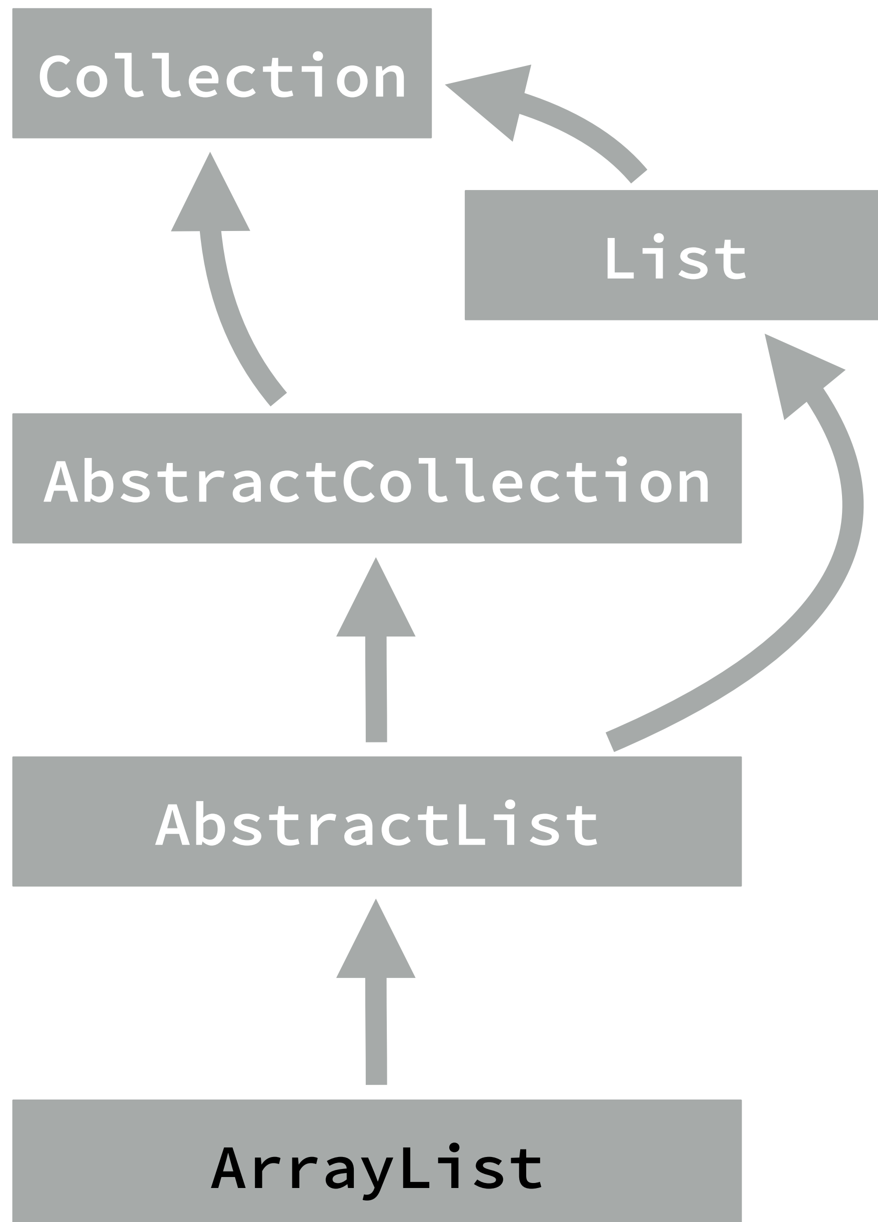
Classe instantiable

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



Interfaces : décrivent le *concept* de collection/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)

Règle des classes

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.

La documentation d'une classe héritable doit être bien plus détaillée que celle d'une classe instantiable, et décrire tous les aspects de sa mise en œuvre susceptibles d'influencer le comportement des sous-classes, p.ex. le fait qu'une méthode en utilise une autre.