Patrons de conception I

Pratique de la programmation orientée-objet

Michel Schinz

Design Patterns

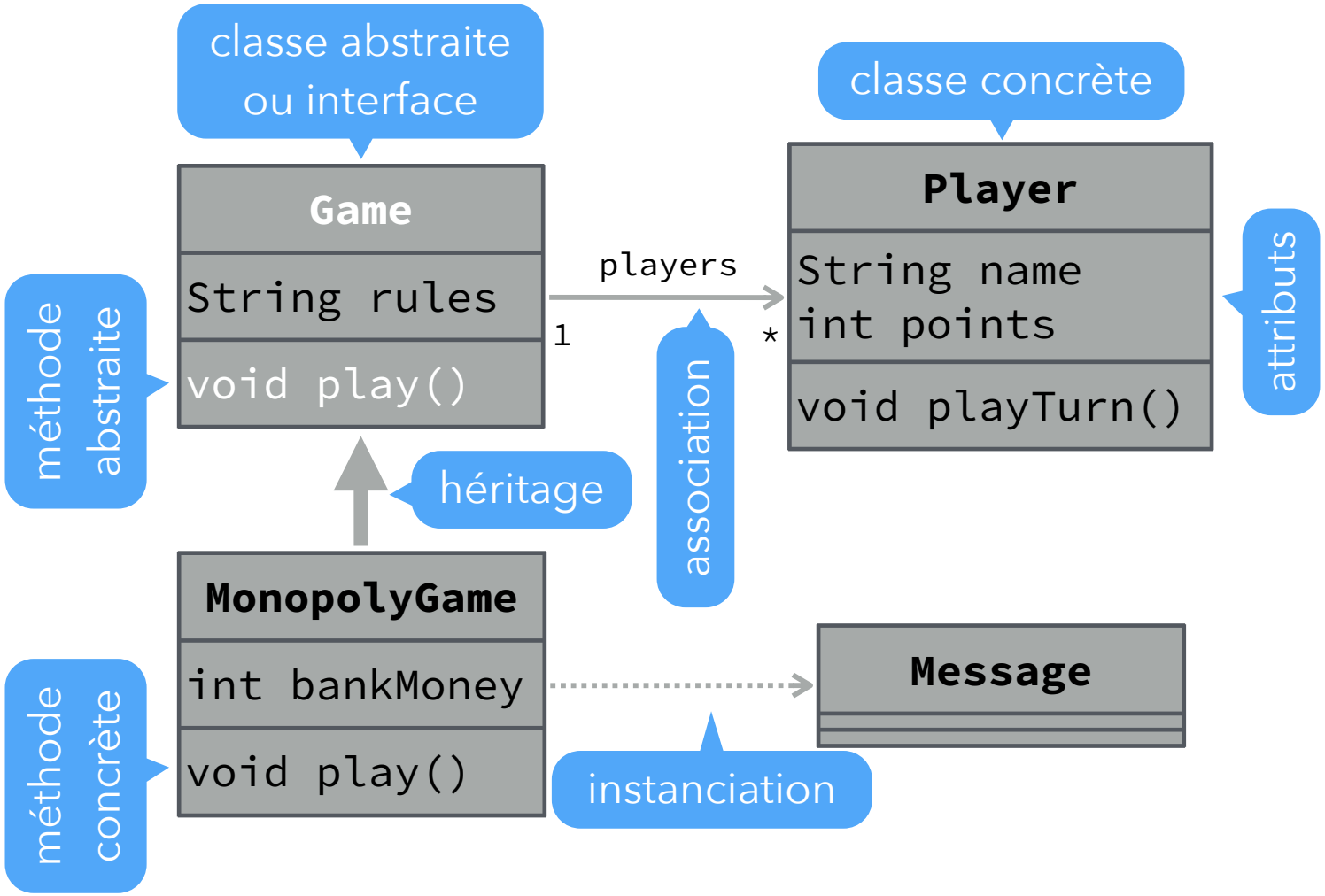
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

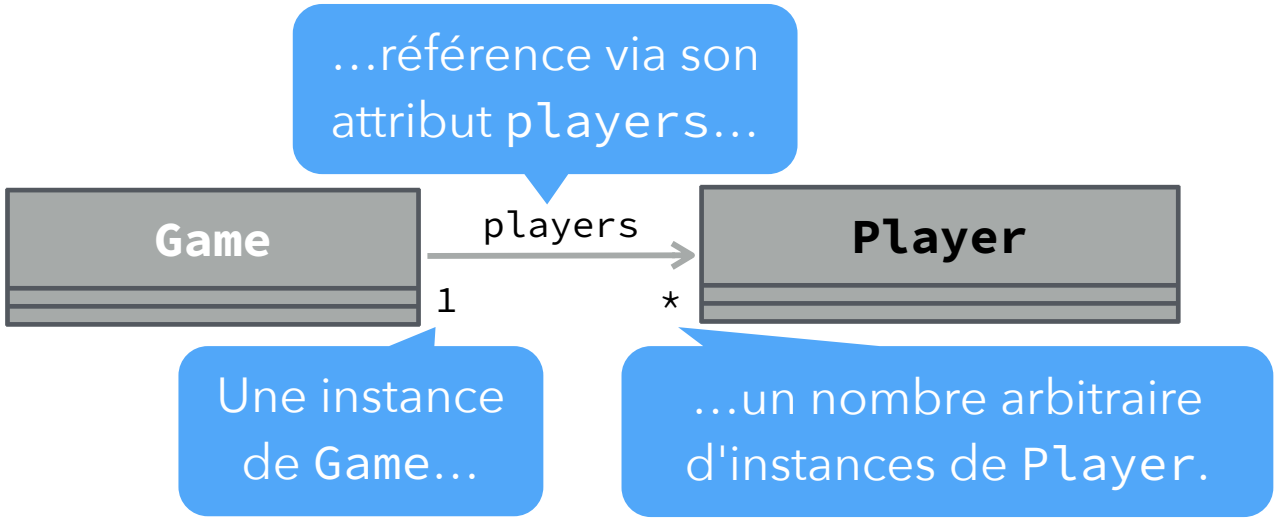
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Diagramme de classe



Relations d'association

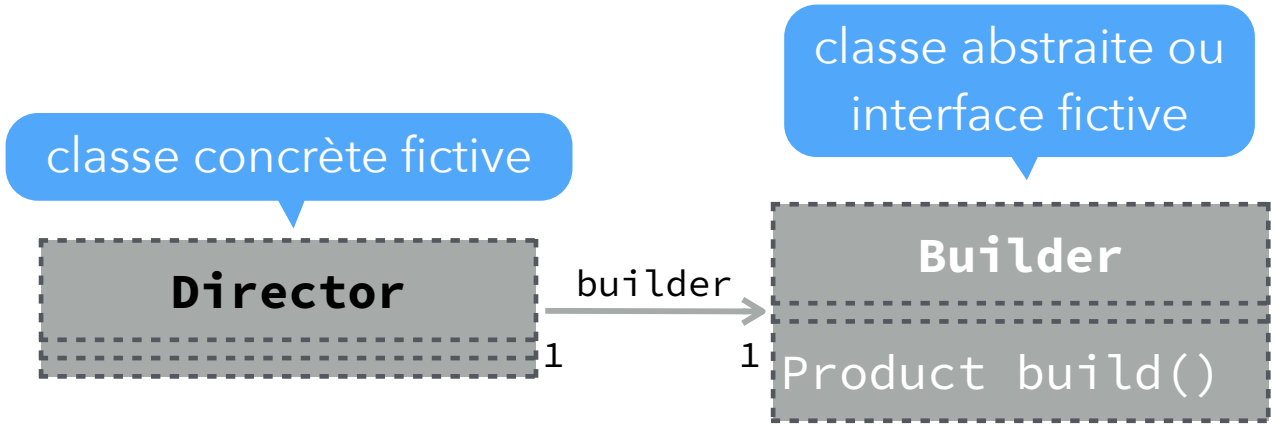
Les relations d'association sont annotées avec leur nom et leur arité. Cette dernière – un entier strictement positif ou `*` pour indiquer une valeur arbitraire – donne le nombre d'objets liés par l'association.



En pratique, `players` sera un attribut de `Game`, d'un type collection quelconque, p.ex. une liste.

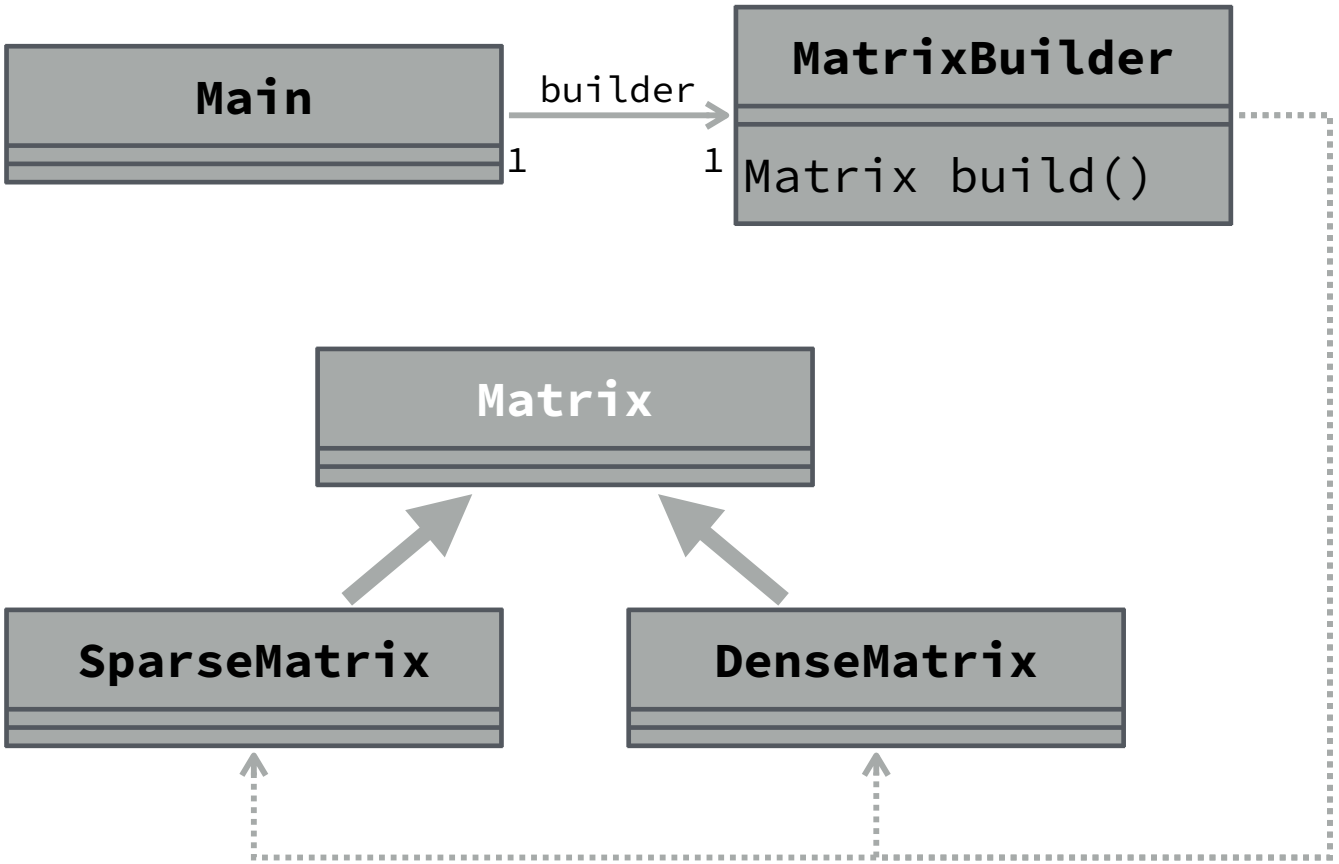
Classes fictives

Etant donné que les patrons décrivent une manière d'organiser un ensemble de classes – et pas un ensemble de classes réel – les diagrammes les illustrant utilisent des classes fictives signalées par des bords discontinus, dont le nom évoque généralement le rôle.

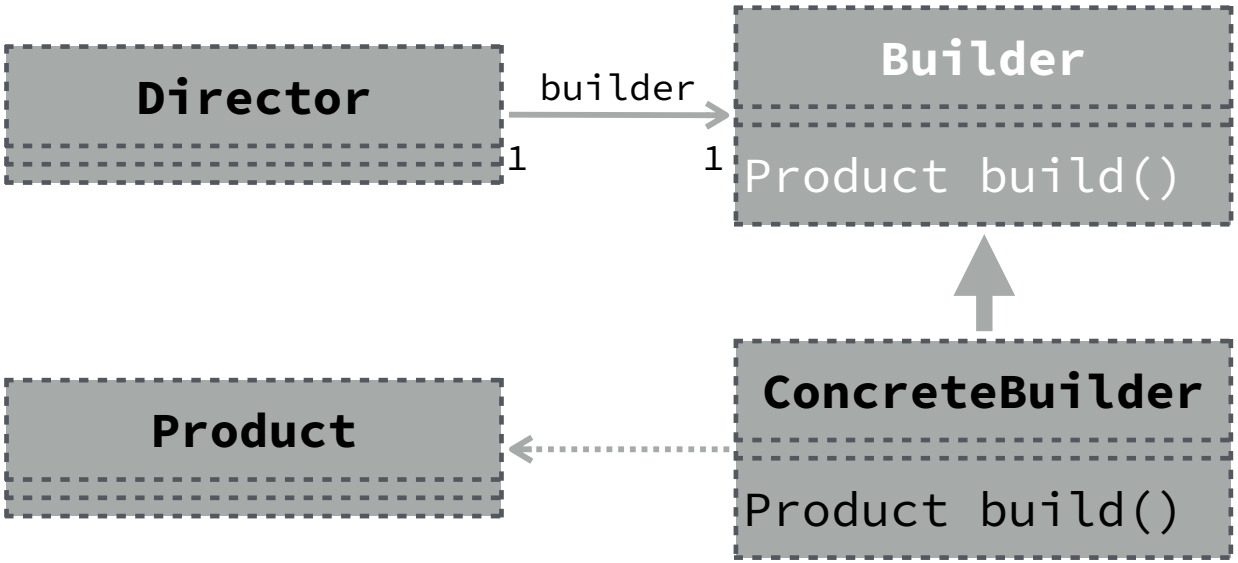


Patron *Builder*

Calcul matriciel

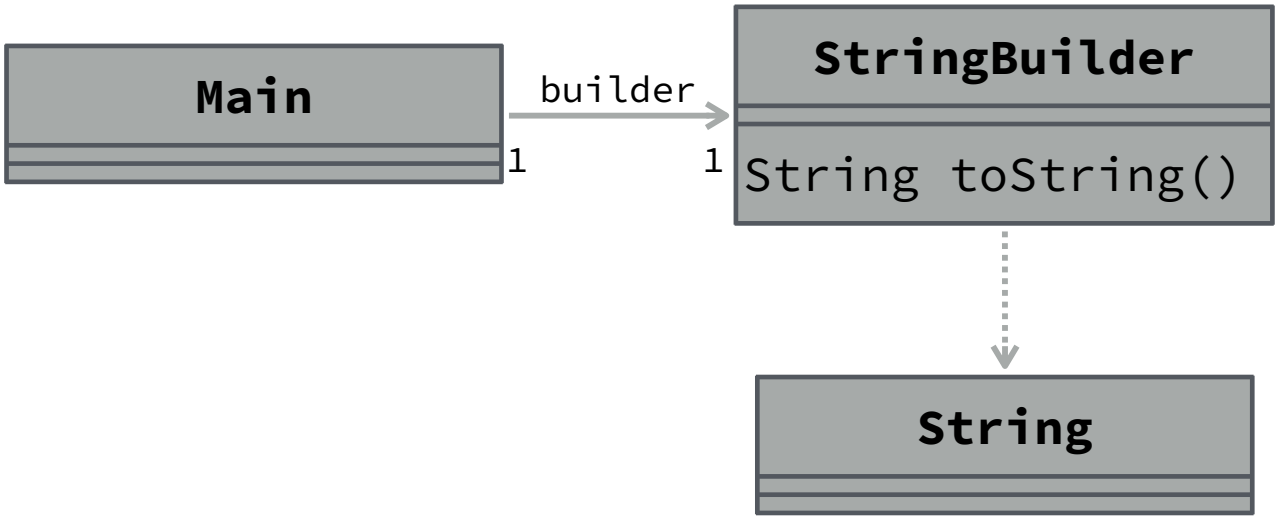


Patron *Builder*



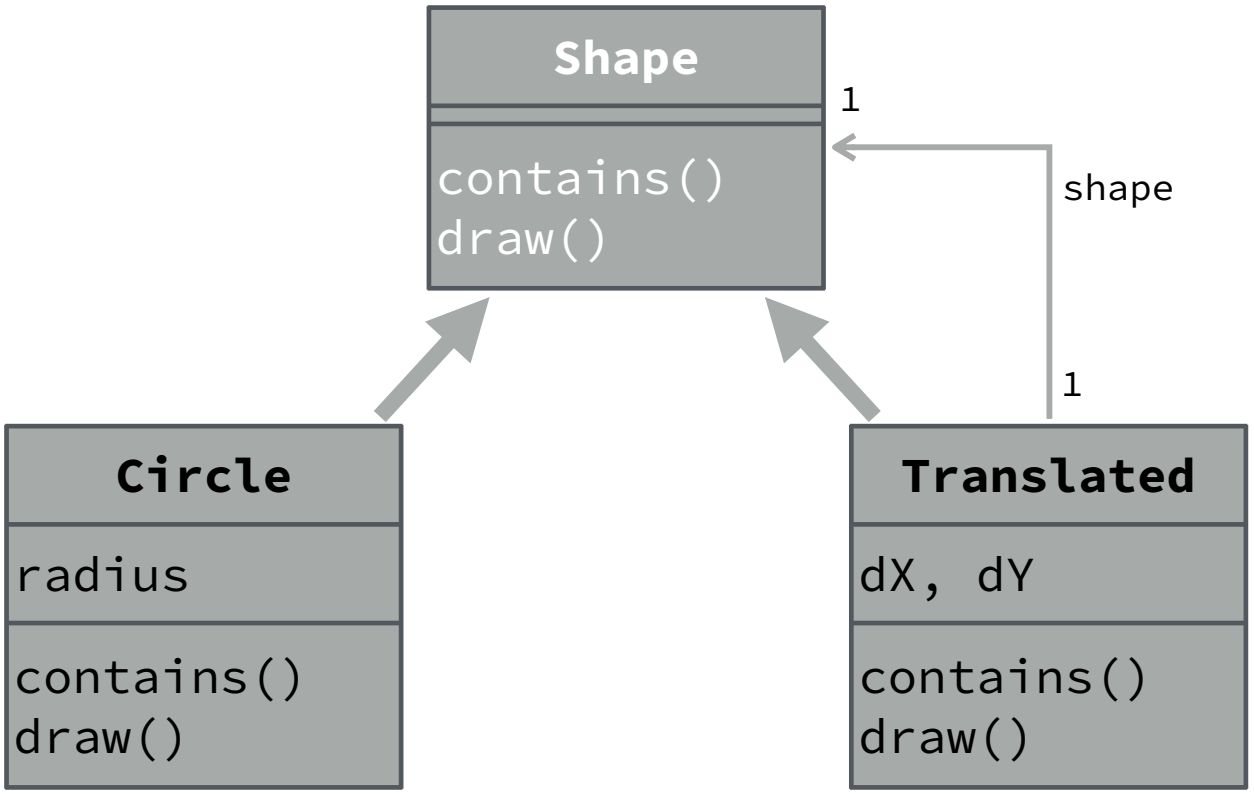
Exemple réel : `String`

La bibliothèque Java offre un bâtisseur nommé `StringBuilder` pour construire des instances de la classe immuable `String`.

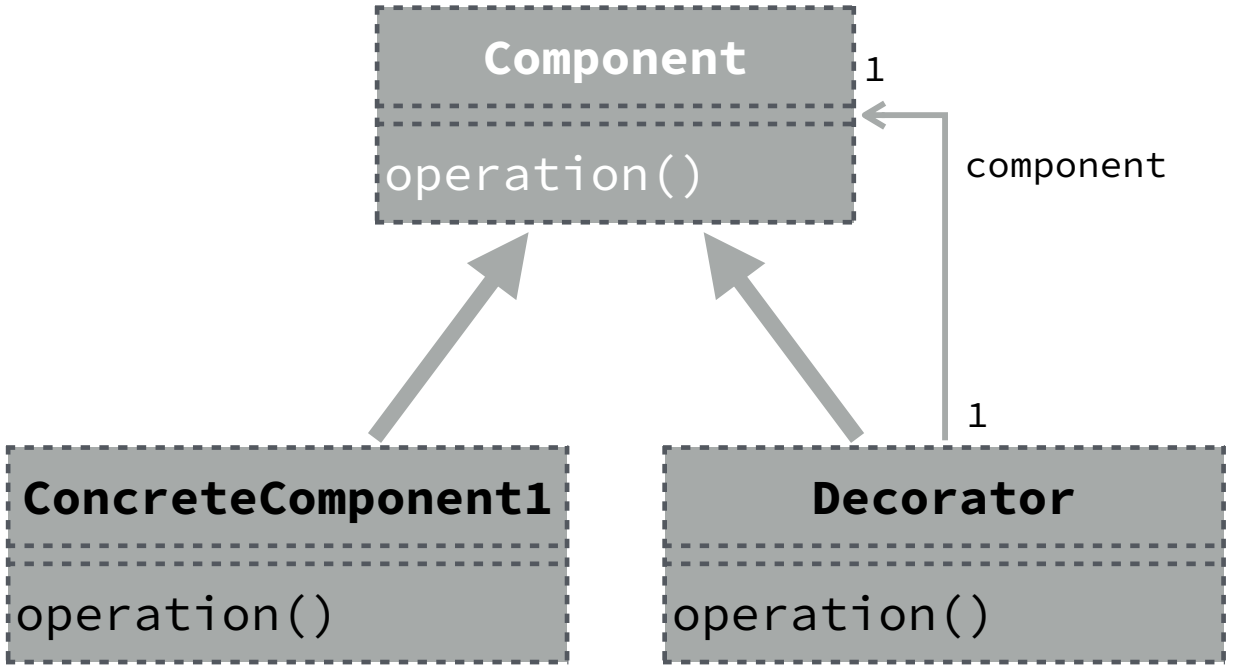


Patron *Decorator*

Figures géométriques

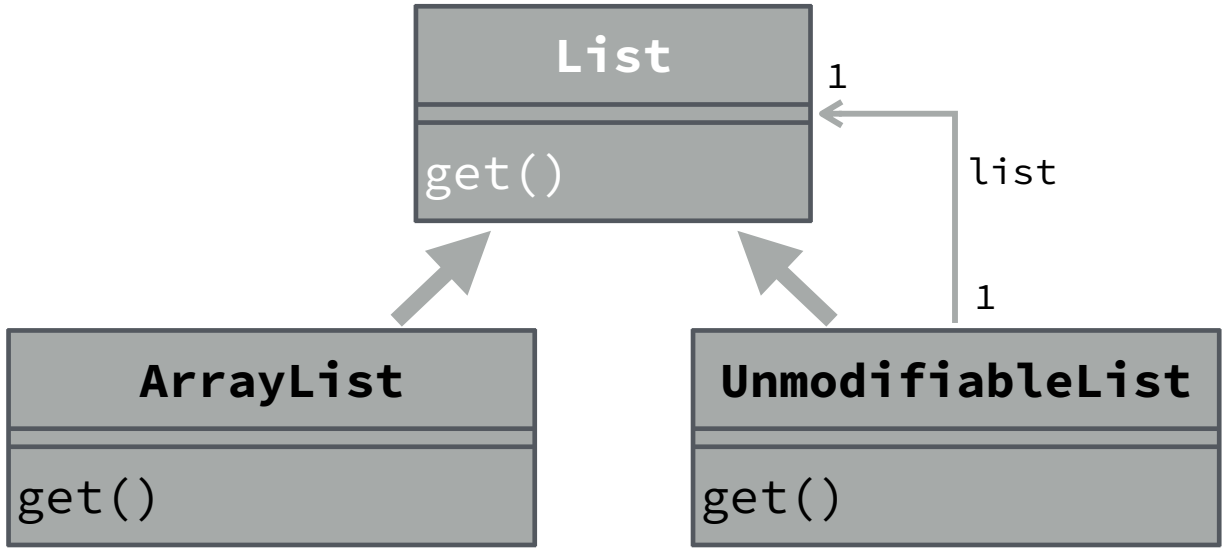


Patron *Decorator*



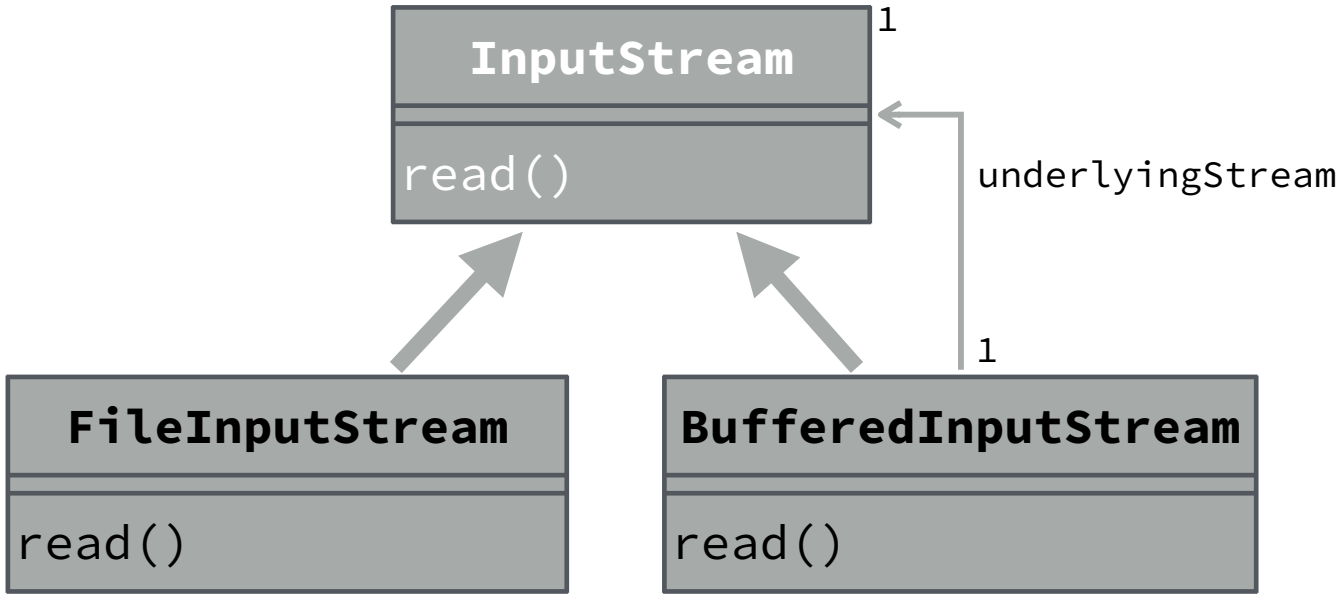
Exemple réel : vues

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.



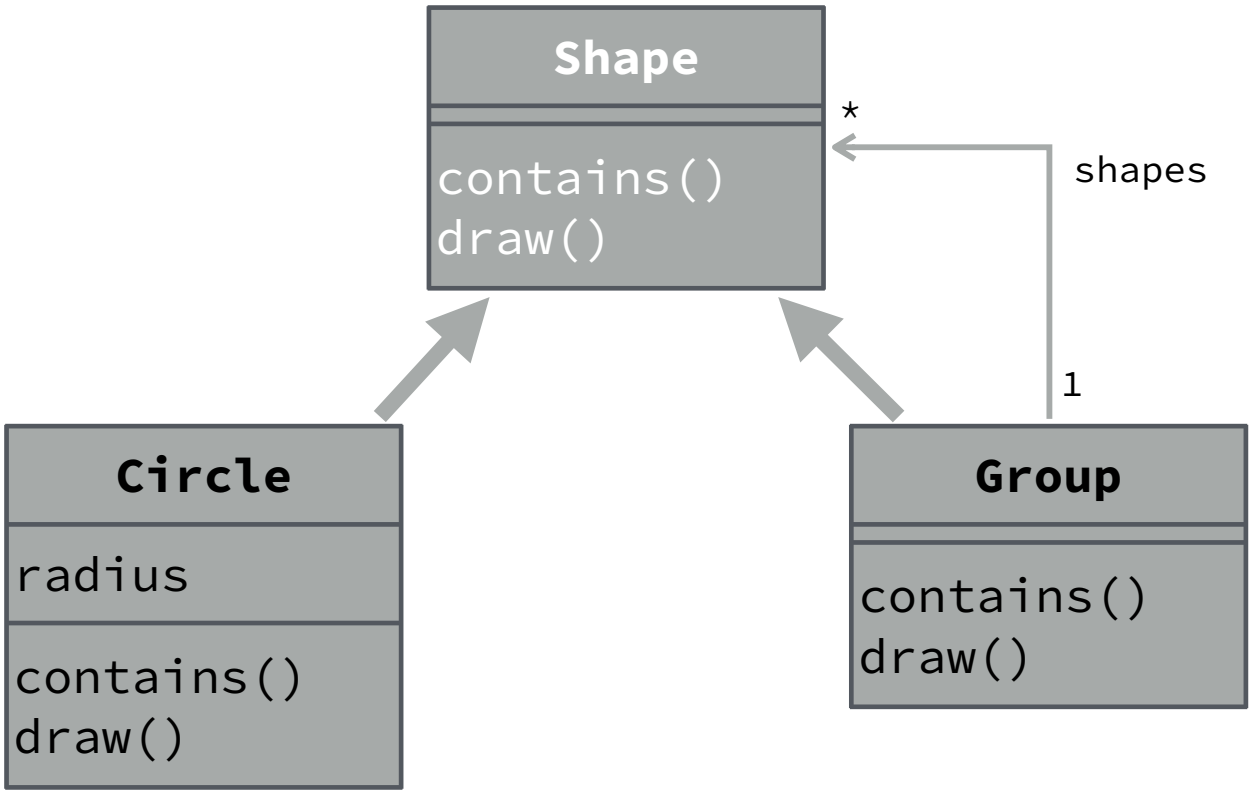
Exemple réel : flots filtrants

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.

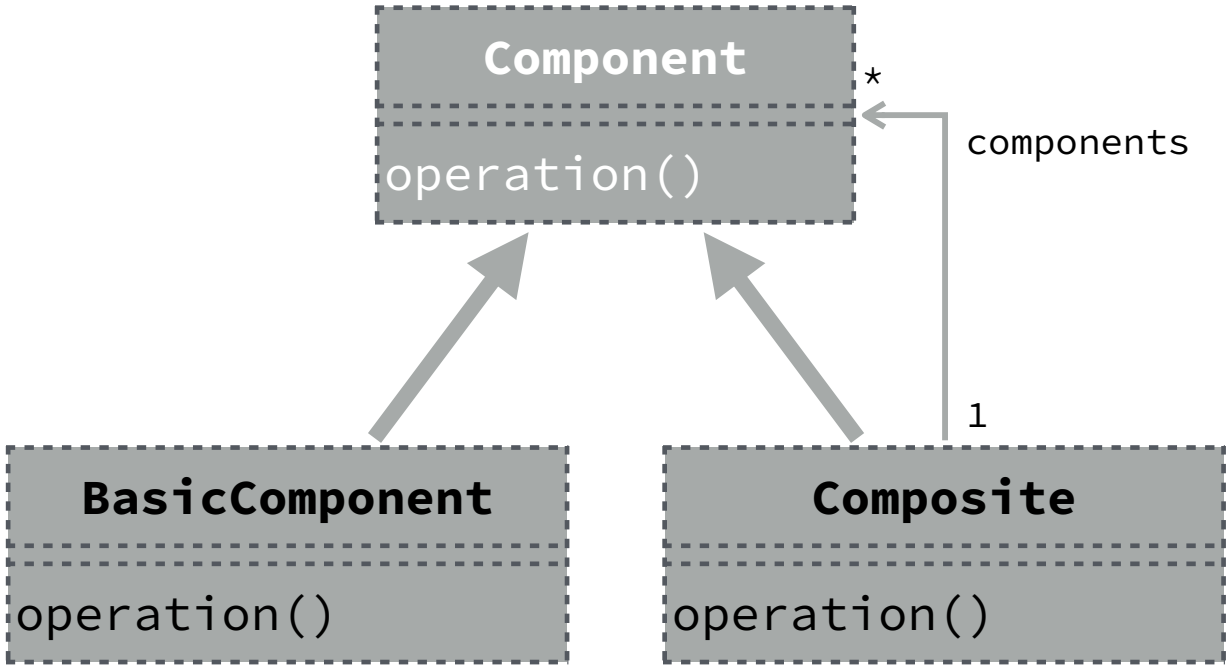


Patron *Composite*

Figures géométriques

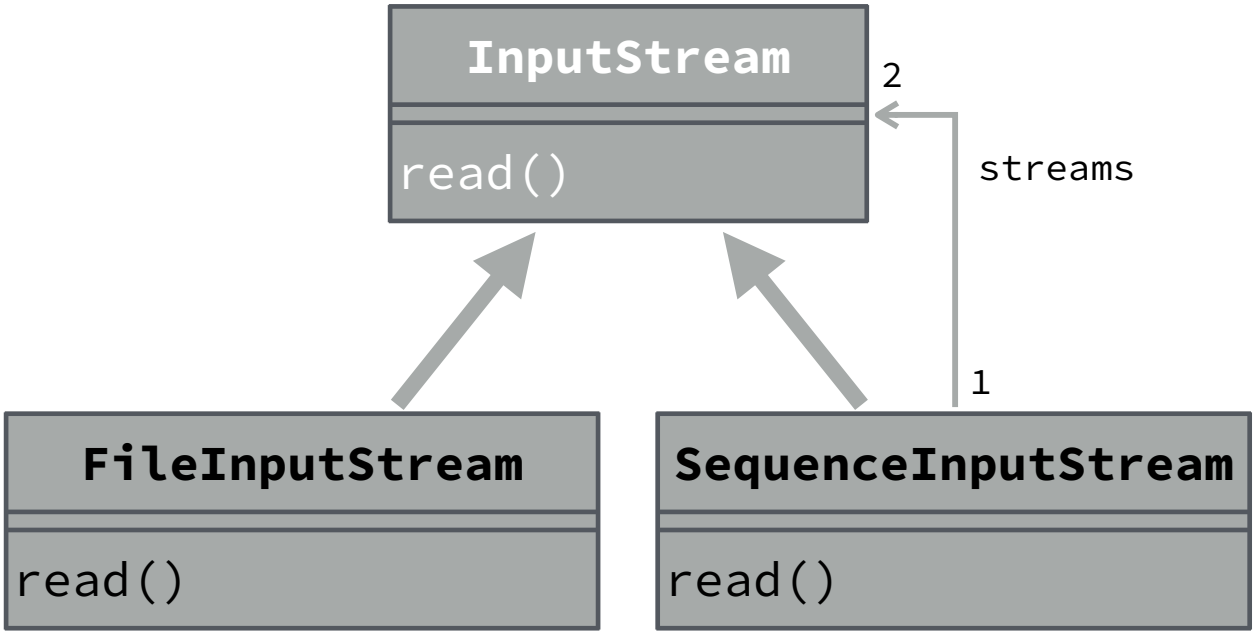


Patron *Composite*



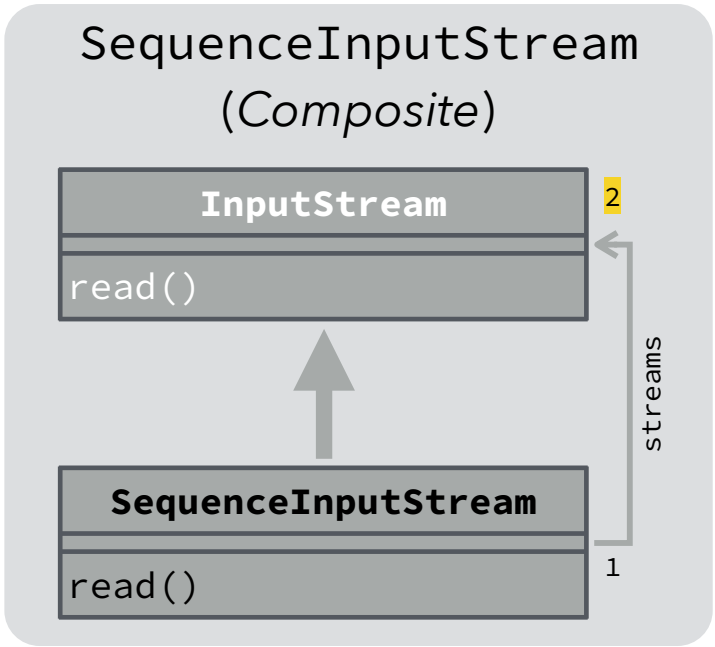
Exemple réel : E/S Java

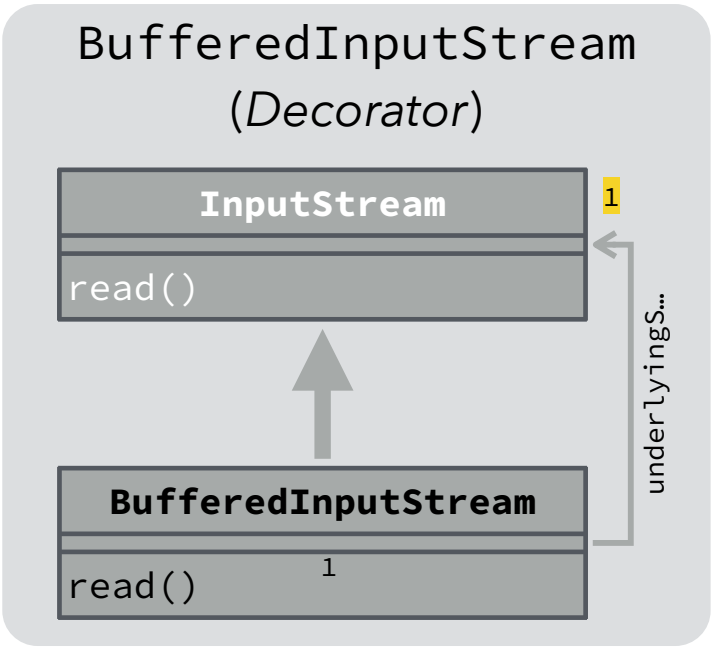
La classe `SequenceInputStream` prend deux flots d'entrée et produit un flot composite qui fournit d'abord les valeurs du premier puis celles du second.



Composite / Decorator

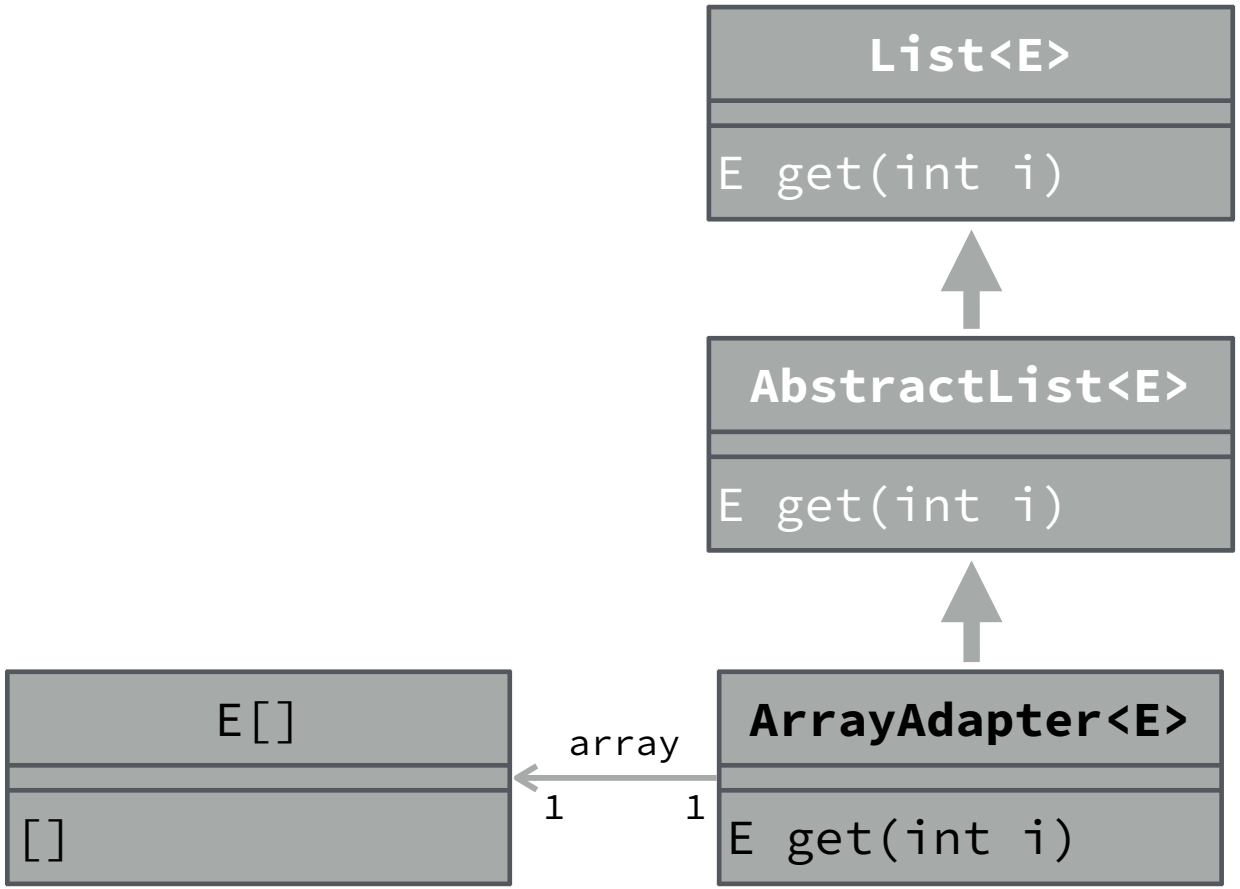
La différence entre *Composite* et *Decorator* est minime et se résume au fait que le premier référence plusieurs objets de son propre type, le second un seul.



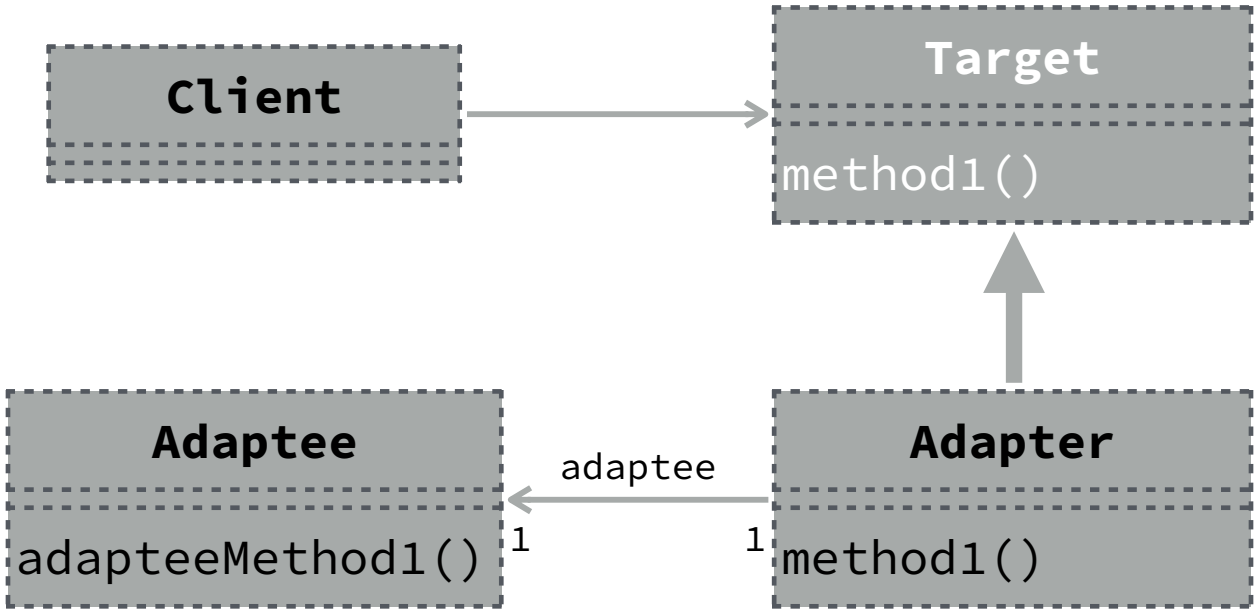


Patron *Adapter*

Adaptateur de tableau

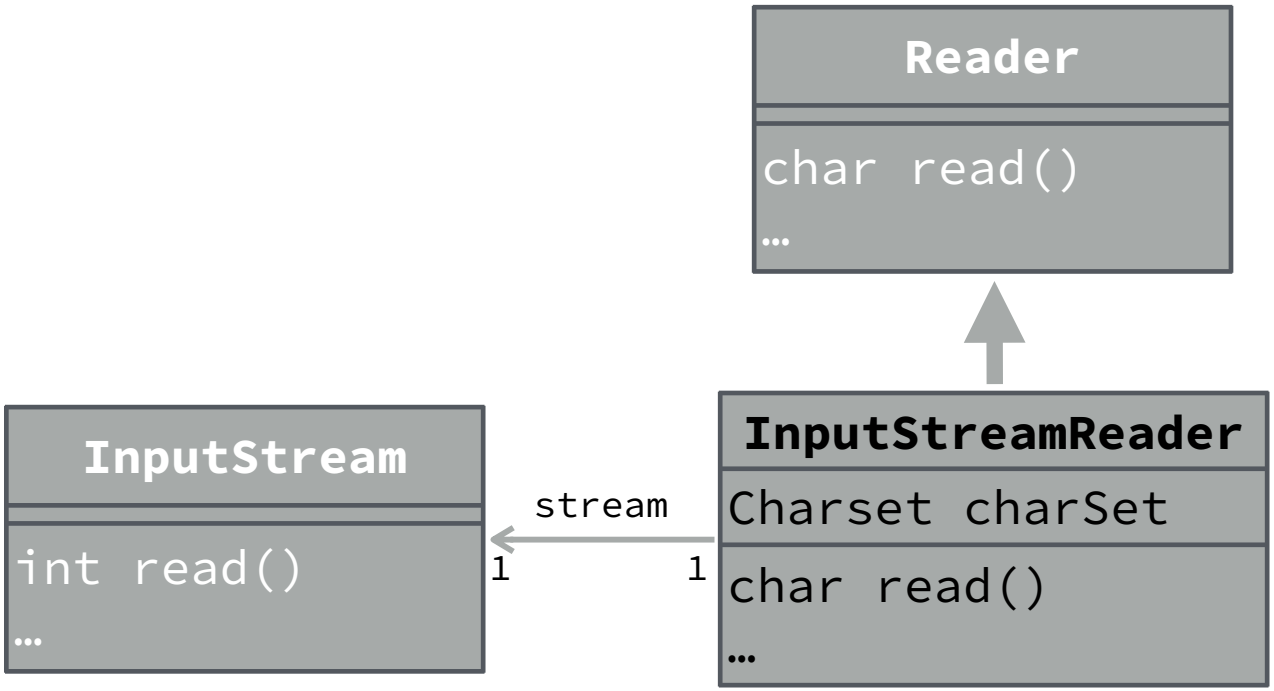


Patron *Adapter*

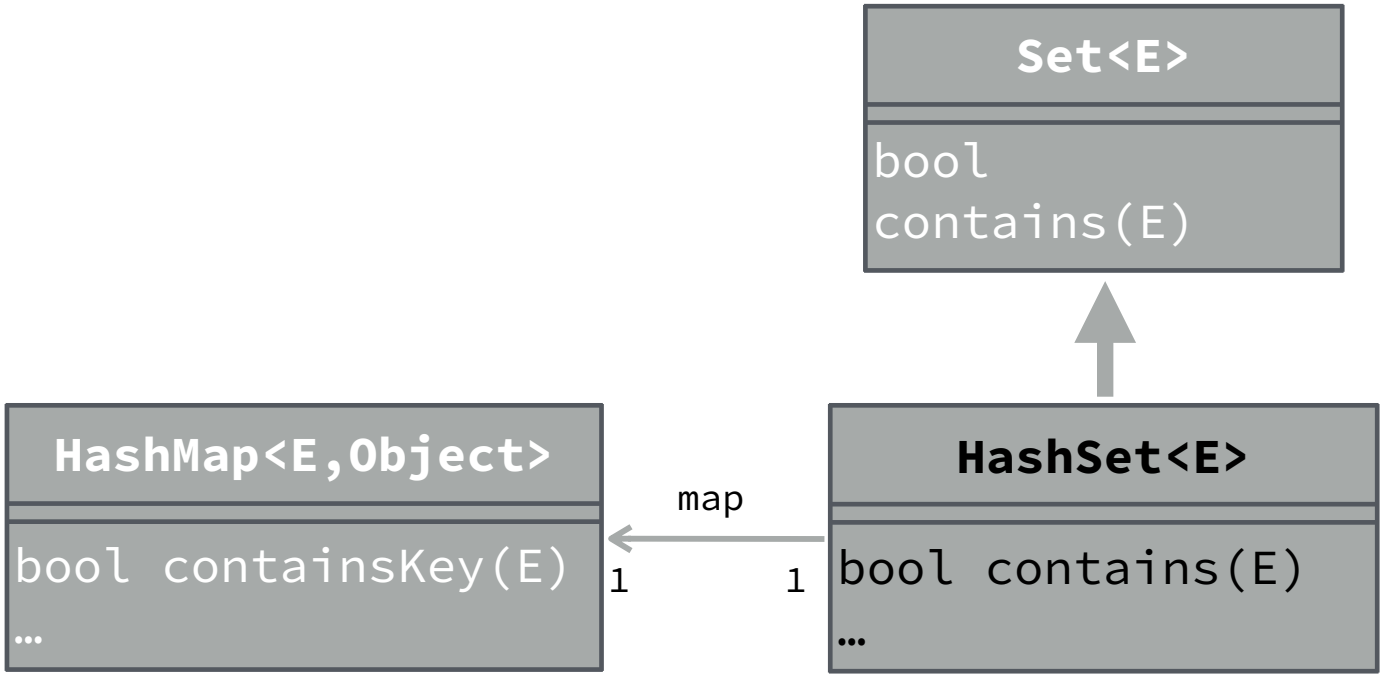


Exemple réel : E/S

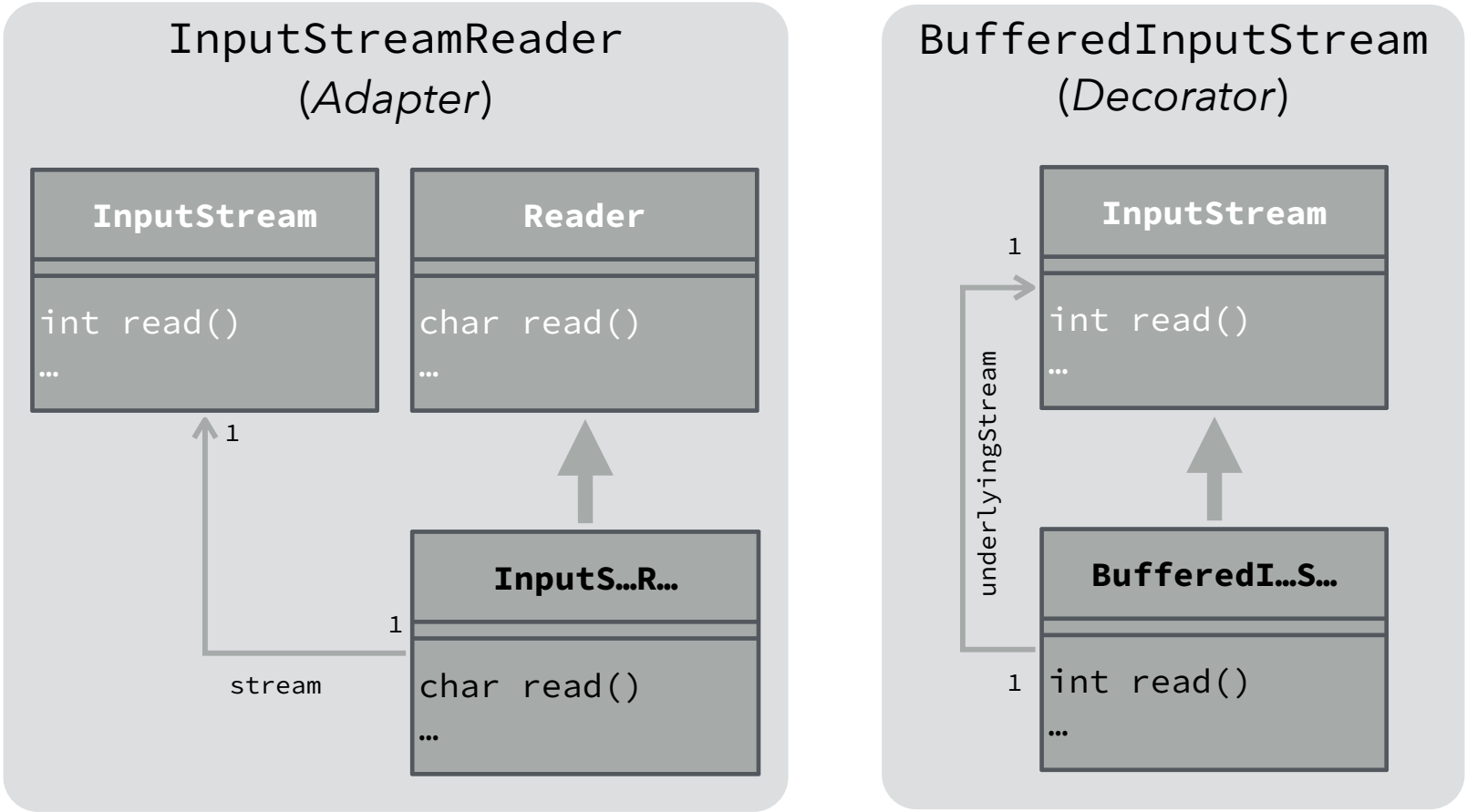
La classe `InputStreamReader` adapte un flot d'entrée d'octets (`InputStream`) pour en faire un lecteur (`Reader`), étant donné un encodage de caractères.



Exemple réel : collections



Adapter / Decorator



Patron *Decorator* et héritage

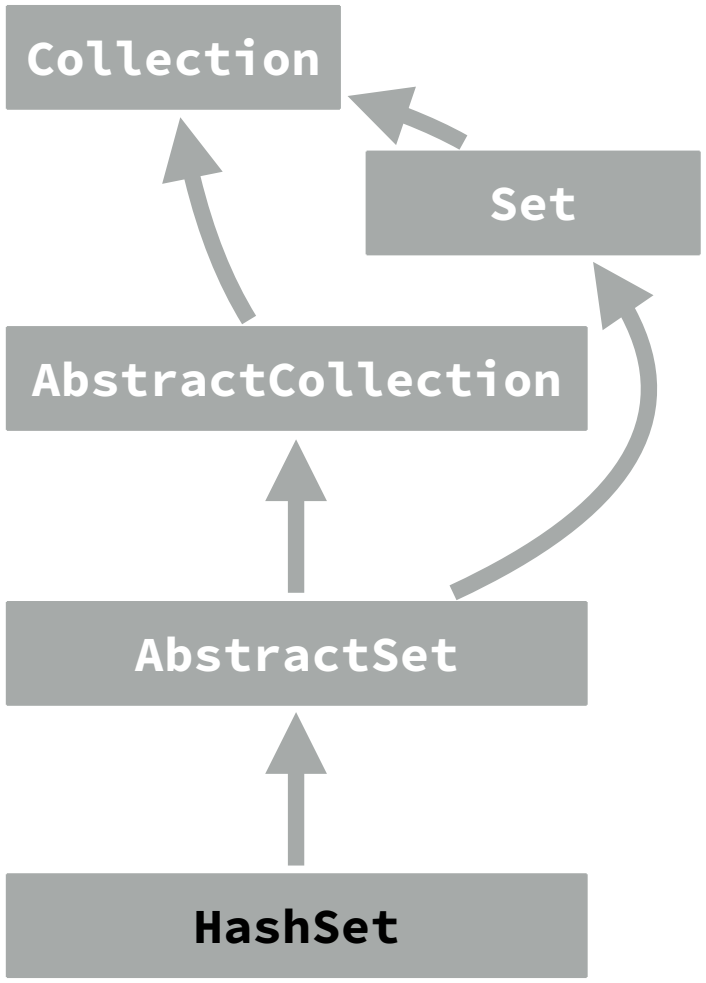
HashSet.addAll

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

Règle des classes

Lorsque vous écrivez une classe, décidez s'il s'agit d'une classe *héritable* ou *instanciable*.
Rendez-la abstraite dans le premier cas, finale dans le second.

Exemple : collections Java



Interfaces : décrivent le *concept* de collection/ensemble.

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)