Patrons de conception II

Michel Schinz

2018-04-30

1 Le patron *Adapter*

1.1 Illustration du problème

La classe `Collections` de la bibliothèque Java possède la méthode statique `shuffle` 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 ?

1.2 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. Pour simplifier sa définition, on peut en faire une sous-classe de `AbstractList`, une classe héritable dont le but est justement de faciliter la définition de nouveaux types de listes.

```
final class ArrayAdapter<E> extends AbstractList<E> {
  private final E[] array;

  public ArrayAdapter(E[] array) {
    this.array = array;
  }

  @Override
  public E get(int index) {
    return array[index];
  }
```

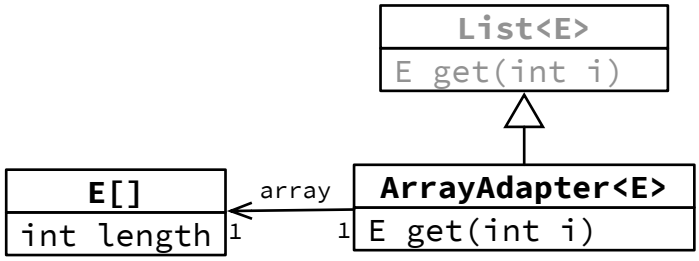
```
  @Override
  public E set(int index, E newValue) {
    E oldValue = array[index];
    array[index] = newValue;
    return oldValue;
  }

  @Override
  public int size() {
    return array.length;
  }
}
```

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<>(array);
// mélange les éléments de array
Collections.shuffle(adaptedArray);
```

Le diagramme de classes ci-dessous illustre cette solution :



Notez qu'un tel adaptateur est fourni dans la bibliothèque Java par la méthode `asList` de `Arrays`. Il est donc possible de mélanger un tableau en écrivant simplement :

```
String[] array = …;
List<String> adaptedArray = Arrays.asList(array);
// mélange les éléments de array, via adaptedArray
Collections.shuffle(adaptedArray);
```

1.3 Généralisation

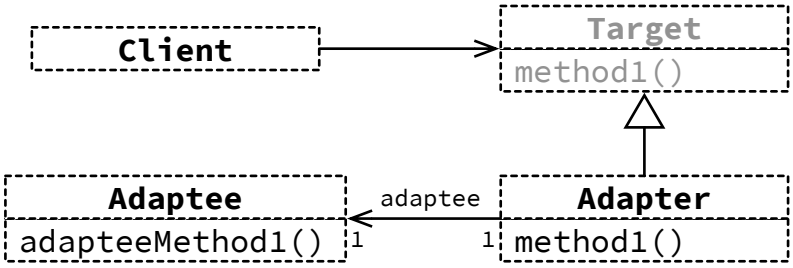
De manière générale, lorsqu'on désire utiliser une instance d'une classe donnée là où une instance d'une autre classe est attendue, il est possible d'écrire une classe servant

d'adaptateur. Bien entendu, il faut que le comportement des deux classes soit relativement similaire, sans quoi l'adaptation n'a pas de sens.

Le patron de conception *Adapter* décrit cette solution.

1.4 Diagramme de classes

Le diagramme de classes ci-dessous illustre l'adaptation au moyen d'un adaptateur `Adapter` d'une classe `Adaptee` en `Target`, pour le compte d'une classe cliente `Client`.



1.5 Exemples réels

Le patron *Adapter* se rencontre très souvent en pratique, comme les quelques exemples ci-dessous l'illustrent.

1.5.1 Collections Java

On l'a vu, la méthode `asList` de la classe `Arrays` utilise le patron *Adapter* pour adapter un tableau en liste.

D'autre part, la méthode `newSetFromMap` de la classe `Collections` prend en argument une table associative et l'adapte pour en faire un ensemble. Il est ainsi possible, par exemple, de l'utiliser pour adapter une instance de `HashMap` en un ensemble qui se comporte comme une instance de `HashSet` :

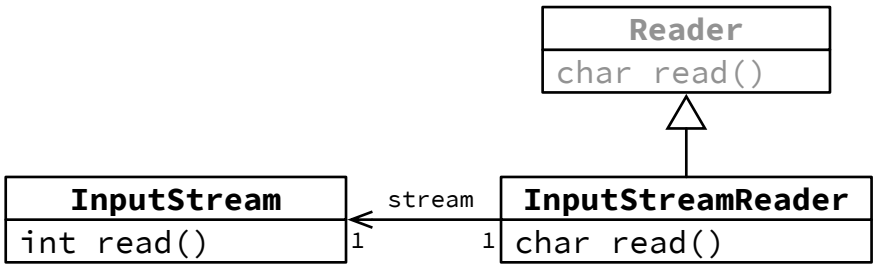
```
Set<String> s =
  Collections.newSetFromMap(new HashMap<>());
```

Bien entendu, étant donné l'existence de `HashSet`, il n'est pas nécessaire d'utiliser cette méthode ici, mais la possibilité d'adapter n'importe quel type de table associative pour en faire un ensemble est utile dans d'autres situations.

1.5.2 Entrées/sorties Java

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

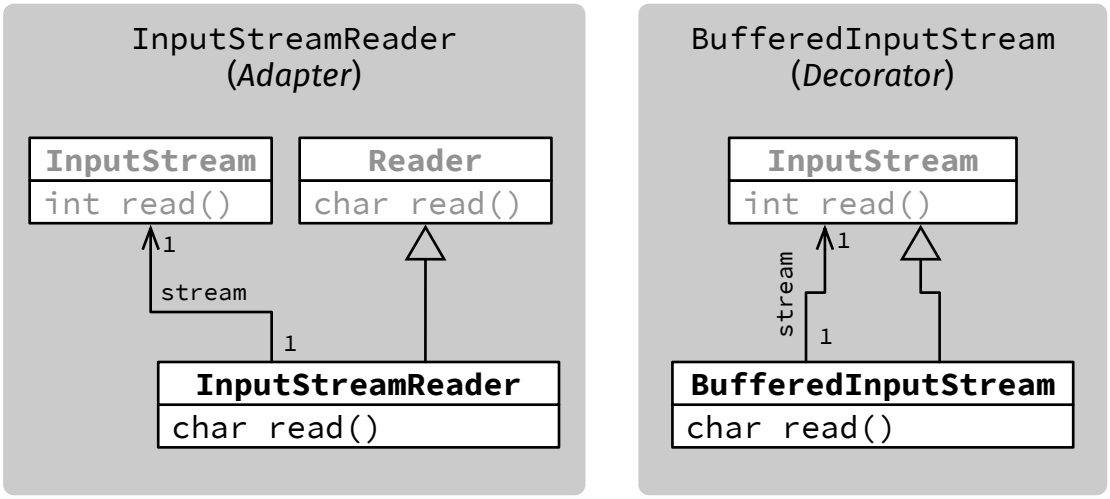
La classe `OutputStreamWriter` adapte un flot de sortie d'octets (`OutputStream`) pour en faire un écrivain (`Writer`), étant donné un encodage de caractères.



1.6 *Adapter* et *Decorator*

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 `InputStreamReader` (un adaptateur) et de `BufferedInputStream` (un décorateur).



Une conséquence de cette différence est qu'il est possible d'empiler des décorateurs, mais pas des adaptateurs. Par exemple, l'extrait de code ci-dessous empile deux décorateurs (de type `BufferedInputStream` et `GZIPInputStream`) sur un flot de base de type `FileInputStream`.

```
InputStream s = new GZIPInputStream(
                  new BufferedInputStream(
                    new FileInputStream…()));
```

Un tel empilement ne serait pas possible avec des adaptateurs.

2 Le patron *Observer*

2.1 Illustration du problème

Admettons que l'on désire écrire un tableur simple. Dans un tableur, une feuille de calcul est composée de cellules qui contiennent soit des nombres, soit des formules. Une formule décrit comment calculer la valeur de la cellule en fonction de la valeur d'autres cellules. L'image ci-dessous présente un tableur simple, les formules commençant par le caractère `=` :

	A	B	C
1	10	6	
2	12	12	
3	=A1+A2	=B1+B2	=A3+B3

Le contenu d'une cellule contenant une formule doit être mis à jour dès que le contenu d'une cellule dont elle dépend change. Dans l'exemple ci-dessus, la cellule `A3` dépend des cellules `A1` et `A2`, `B3` dépend de `B1` et `B2`, et `C3` dépend de `A3` et `B3` — donc indirectement de `A1`, `A2`, `B1` et `B2`.

Comment organiser le programme pour garantir que les mises à jour nécessaires soient faites ?

2.2 Solution

Une solution consiste à permettre aux cellules d'en *observer* d'autres. Lorsque la valeur d'une cellule change, toutes les cellules qui l'observent sont informées du changement.

Une fois cette possibilité offerte, toute cellule contenant une formule observe la ou les cellules dont elle dépend — c-à-d celles qui apparaissent dans sa formule. Lorsqu'elle est informée du changement de l'une d'entre elles, elle met à jour sa propre valeur.

Afin d'illustrer cette solution, modélisons un tableur très simple dont les cellules contiennent soit un nombre, soit une formule. Pour simplifier les choses, nos formules sont toutes des sommes de la valeur de deux autres cellules — c-à-d qu'elles ont la forme `=C1+C2` où `C1` et `C2` sont des cellules.

2.2.1 Sujets et observateurs

Lorsqu'un objet observe un autre objet, on appelle le premier **l'observateur** (*observer*) et le second le **sujet** (*subject*). Notez qu'un objet donné peut être à la fois sujet et observateur. Ainsi, dans notre exemple, la cellule `A3` est à la fois observatrice — des cellules `A1` et `A2` — et sujet d'une observation — par la cellule `C3`.

Un observateur doit pouvoir être informé des changements du (ou des) sujet(s) qu'il observe. A cet effet, il possède une méthode, nommée `update` ci-dessous, qui est appelée lorsqu'un sujet qu'il observe a été modifié. Le sujet en question est passé en argument à cette méthode, pour permettre son identification. L'interface `Observer` ci-dessous représente un tel observateur :

```
public interface Observer {
  void update(Subject s);
}
```

Un sujet doit mémoriser l'ensemble de ses observateurs, et les avertir lors d'un changement de son état. Pour ce faire, le sujet offre des méthodes permettant l'ajout et la suppression d'un observateur à son ensemble. De plus, un sujet doit prendre garde à bien avertir ses observateurs lorsque son état change. L'interface `Subject` ci-dessous représente un tel sujet :

```
public interface Subject {
  void addObserver(Observer o);
  void removeObserver(Observer o);
}
```

2.2.2 Cellule

Le code commun à toutes les cellules est placé dans une classe abstraite de cellule nommée `Cell`. Toute cellule doit pouvoir être sujet d'une observation, donc `Cell` implémente l'interface `Subject` et met en œuvre les méthodes `addObserver` et `removeObserver`. Elle offre également une méthode protégée (`notifyObservers`) pour avertir les observateurs d'un changement.

```
public abstract class Cell implements Subject {
  private Set<Observer> observers = new HashSet<>();
  abstract public int value();

  public void addObserver(Observer o) {
    observers.add(o);
  }

  public void removeObserver(Observer o) {
    observers.remove(o);
  }

  protected void notifyObservers() {
    for (Observer o: observers)
      o.update(this);
```

```
  }
}
```

Une cellule contenant une valeur ne fait rien d'autre que mémoriser celle-ci et notifier ses observateurs lorsqu'elle change.

```
public final class ValueCell extends Cell {
  private int value = 0;
  public int value() {
    return value;
  }
  public void setValue(int newValue) {
    if (newValue != value) {
      value = newValue;
      notifyObservers();
    }
  }
}
```

Une cellule somme diffère d'une cellule valeur en ce qu'elle doit observer les deux cellules à sommer. Lorsque l'une d'elles change, la valeur de la cellule somme est mise à jour en fonction.

Pour pouvoir observer les cellules à sommer, la classe des cellules somme doit implémenter l'interface `Observer` et s'enregistrer comme observateur de ces cellules.

```
public final class SumCell
    extends Cell implements Observer {
  private final Cell c1, c2;
  private int sum = 0;

  public SumCell(Cell c1, Cell c2) {
    this.c1 = c1;
    this.c2 = c2;
    c1.addObserver(this);
    c2.addObserver(this);
  }

  public void update(Subject s) {
    int newSum = c1.value() + c2.value();
    if (newSum != sum) {
      sum = newSum;
      notifyObservers();
    }
  }
```

```
  public int value() {
    return sum;
  }
}
```

2.2.3 Tableur

La classe représentant le tableur (très) simplifié se contente de créer les cellules et de leur attribuer une valeur ou une formule. Afin de pouvoir observer les changements de valeur des cellules, le tableur est lui-même un observateur — c-à-d qu'il implémente l'interface `Observer`. Sa méthode d'observation affiche le nouvel état de la cellule modifiée à l'écran.

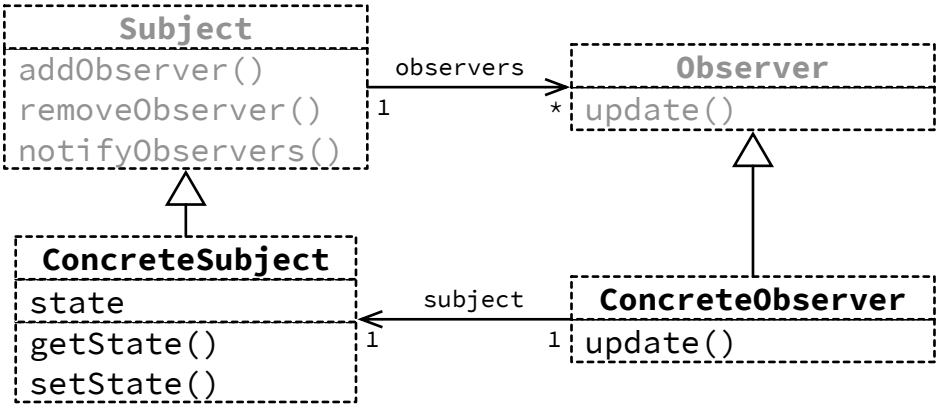
```
public final class SpreadSheet implements Observer {
  public SpreadSheet() {
    ValueCell A1 = new ValueCell(), A2 = …;
    SumCell A3 = new SumCell(A1, A2);
    // … C3
    A1.addObserver(this); // A2 … C3 idem
    A1.setValue(10);      // A2 … C3 idem
  }
  public void update(Subject s) {
    Cell c = (Cell)s;
    System.out.println("nouvelle␣valeur␣de␣"
                       + c + ":␣" + c.value());
  }
}
```

2.3 Généralisation

Chaque fois que l'état d'un premier objet — appelé observateur — dépend de l'état d'un second objet — appelé sujet — l'observateur peut observer l'état de son sujet et se mettre à jour dès que celui-ci change. Pour que l'observation soit possible, il faut que le sujet gère l'ensemble de ses observateurs et les avertisse lors d'un changement.

C'est l'idée du patron de conception *Observer*.

2.4 Diagramme de classes



2.5 Avantages et inconvénients du patron

Le gros intérêt du patron *Observer* est qu'il découple les sujets et les observateurs, en ce que le sujet ne sait généralement rien de ses observateurs. Pour cette raison, le patron *Observer* est très souvent utilisé par les bibliothèques d'interface graphique, par exemple.

Cela dit, le patron *Observer* possède également plusieurs inconvénients :

- il impose un style de programmation impératif (non immuable), dans lequel les objets ont un état qui change au cours du temps,
- il peut rendre les dépendances cycliques entre états difficiles à voir,
- il peut rendre observables des états qui ne devraient pas l'être (*glitches*).

Examinons plus en détail ces trois problèmes.

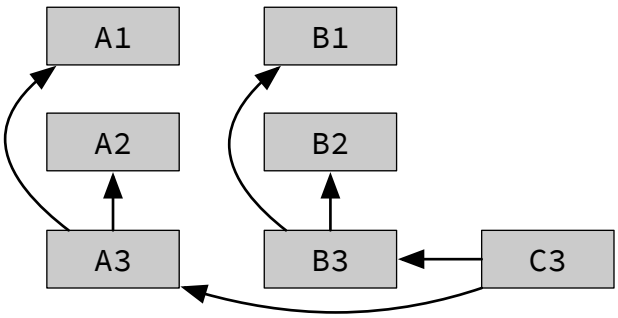
2.5.1 *Observer* et immuabilité

Le patron *Observer* est fondamentalement incompatible avec l'immuabilité, dans le sens où tout sujet doit avoir un état modifiable. Observer un objet immuable n'est d'aucune utilité ! L'utilisation du patron *Observer* force donc un style de programmation basé sur des objets non immuables ce qui, nous l'avons vu, pose plusieurs problèmes.

2.5.2 Dépendances

Lorsqu'on utilise le patron *Observer*, il peut être utile de dessiner le **graphe d'observation** du programme afin de détecter d'éventuels problèmes. Ce graphe a les sujets et observateurs du programme comme nœuds et un arc va de tout observateur à son (ou ses) sujet(s).

Ainsi, le graphe d'observation de notre tableur simplifié est :



Pour mémoire, le tableur lui-même se présente ainsi :

	A	B	C
1	10	6	
2	12	12	
3	=A1+A2	=B1+B2	=A3+B3

Normalement, le graphe d'observation devrait être acyclique — dénué de cycles — car ceux-ci peuvent provoquer des boucles infinies d'envoi de notifications. En pratique, ces cycles sont parfois difficiles à éviter et on utilise donc différents moyens — plus ou moins propres — pour éviter de boucler à l'infini lors de l'envoi de notifications.

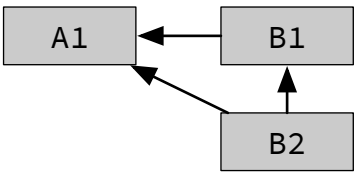
Notez que les tableurs détectent et interdisent les cycles, mais rien n'empêche un graphe d'observation d'être cyclique dans un programme réel utilisant le patron *Observer*.

2.5.3 Glitches

Même en l'absence de cycles d'observation, le patron *Observer* peut poser problème. On peut l'illustrer au moyen de la feuille de calcul suivante :

	A	B
1	10	=A1-1
2		=1/(A1-B1)

dont le graphe d'observation est :



Que peut-il se passer si on introduit la valeur 9 dans la cellule `A1` ?

Etonnamment, il est possible qu'une division par zéro se produise, même si cela n'est mathématiquement pas possible étant donnée la formule utilisée ! En effet, si la nouvelle valeur de `A1` est d'abord communiquée à `B2` et que celle-ci met sa valeur à jour en utilisant les valeurs actuelles des cellules, elle le fait en ayant déjà connaissance de la nouvelle valeur de `A1` (9) mais pas encore de la nouvelle valeur de `B1`, qui vaut toujours 9.

Le patron *Observer* ne donne aucune garantie concernant l'ordre dans lequel les notifications sont propagées dans le graphe d'observation. En conséquence, il est possible d'observer des états théoriquement impossibles, que l'on nomme *glitches* en anglais, comme ci-dessus.

Notez que les tableurs s'assurent que les *glitches* ne se produisent pas en propageant les modifications en ordre topologique, mais là encore rien de similaire n'existe dans un programme utilisant le patron *Observer*.

2.6 Exemples réels

Le patron de conception *Observer* est très fréquemment utilisé dans les bibliothèques de gestion d'interfaces graphiques.

Les composants graphiques d'une telle application doivent souvent refléter l'état interne de l'application. Pour rester à jour, ils observent simplement la partie de l'état interne de l'application dont ils dépendent.

Par exemple, un composant affichant un graphique dans un tableur peut observer les données qu'il représente pour savoir quand se mettre à jour.

3 Le patron *MVC*

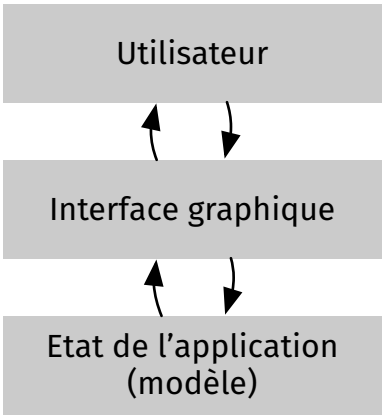
3.1 Illustration du problème

Une application graphique typique est composée d'un certain nombre de composants standards (boutons, menus, listes, etc.) qui permettent à l'utilisateur de visualiser et de manipuler l'état de l'application.

Comment faire en sorte que les composants n'aient aucune connaissance précise de l'application et soient ainsi réutilisables ?

3.2 Solution

La solution (évidente) à ce problème consiste à découpler la gestion de l'interface utilisateur de la gestion de l'état propre à l'application.

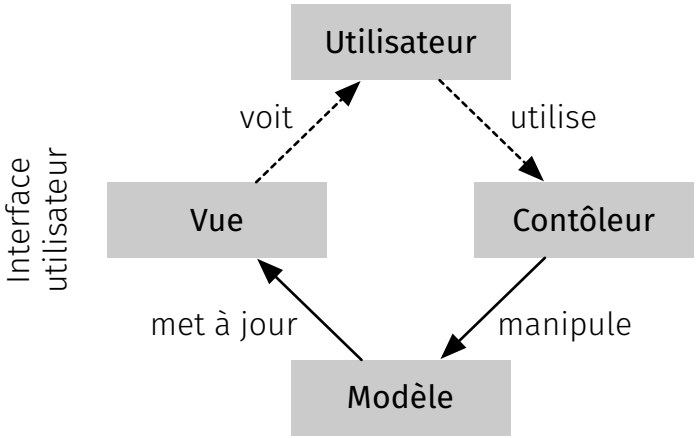


3.3 Généralisation

Le découplage entre la gestion de l'interface utilisateur et la logique propre à l'application peut se faire de différentes manières. Le patron *Model-View-Controller* (MVC) propose d'organiser le code du programme en trois catégories :

1. le modèle, qui contient la totalité du code propre à l'application et qui n'a aucune notion d'interface graphique,
2. la vue, qui contient la totalité du code permettant de représenter le modèle à l'écran,
3. le contrôleur, qui contient la totalité du code gérant les entrées de l'utilisateur et les modifications du modèle correspondantes.

Cette organisation et les interactions qu'elle implique avec l'utilisateur sont illustrées dans la figure ci-dessous :



Notez que le patron *MVC* n'est pas au même niveau que les autres patrons que nous avons vus jusqu'à présent. En effet, il propose une organisation de la totalité du programme, pas seulement d'une petite partie de celui-ci pour résoudre un problème local. Pour cette raison, *MVC* est parfois qualifié de **patron architectural** (*architectural pattern*).

3.3.1 Modèle

Le **modèle** (*model*) est l'ensemble du code qui gère les données propres à l'application. Le modèle ne contient aucun code lié à l'interface utilisateur.

Par exemple, dans un navigateur Web, le modèle contient le code responsable de gérer les accès au réseau, d'analyser les fichiers HTML reçus, d'interpréter les programmes JavaScript, de décompresser les images, etc.

3.3.2 Vue

La **vue** (*view*) est l'ensemble du code responsable de l'affichage des données à l'écran.

Par exemple, dans un navigateur Web, la vue contient le code responsable de transformer le contenu HTML reçu en quelque chose d'affichable, l'affichage de l'état d'avancement des téléchargements en cours, etc.

3.3.3 Contrôleur

Le **contrôleur** (*controler*) est l'ensemble du code responsable de la gestion des entrées de l'utilisateur.

Par exemple, dans un navigateur Web, le contrôleur contient le code responsable de gérer les clics sur des liens, l'entrée de texte dans des zones textuelles, la pression sur le bouton d'interruption de chargement, etc.

3.4 *MVC* et *Observer*

Le patron *Observer* joue un rôle central dans la mise en œuvre du patron *MVC*. Par exemple, pour que le modèle puisse être découplé de la vue, il faut que cette dernière ait la possibilité de réagir aux changements de ce premier. Cela se fait généralement au moyen du patron *Observer*.

3.5 Intérêt du patron

Le patron *MVC* a plusieurs avantages :

- le modèle peut être réutilisé avec différentes interfaces utilisateur (p.ex. application de bureau, application Web, application mobile),
- le modèle est relativement facile à tester puisqu'il n'est absolument pas lié à une interface graphique,
- le code gérant les composants graphiques standards (boutons, menus, etc.) est réutilisable pour de nombreuses applications.

Ces avantages proviennent principalement de la séparation entre le modèle et le couple (vue, contrôleur). En pratique, il est d'ailleurs souvent difficile de découpler la vue du contrôleur, et cela n'est donc que partiellement fait, voire pas du tout.

3.6 Exemples réels

De nombreuses bibliothèques d'interface utilisateur graphique utilisent le patron *MVC* ou l'un de ses dérivés, p.ex. :

- Swing utilise une variante de *MVC* dans laquelle la vue et le contrôleur ne font souvent qu'un,
- Cocoa — la bibliothèque graphique de Mac OS — utilise le patron MVC,
- beaucoup de bibliothèques de gestion d'interface graphique pour les applications Web sont basées sur *MVC*,
- etc.

4 Références

- *Design Patterns : Elements of Reusable Object-Oriented Software* de Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides.