Patrons : *Adapter, Observer, MVC*

Pratique de la programmation orientée-objet

Michel Schinz – 2014-04-28

Patron n°5 :

Adapter

Illustration du problème

Admettons l'existence d'une méthode permettant de mélanger une liste d'éléments :

```
<E> void shuffleList(List<E> 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 pour tableaux

```
import java.util.List;
public final class ArrayAdapter<E>
    implements List<E> {
  private E[] array;

  public ArrayAdapter(E[] array) {
    this.array = array;
  }
  public E get(int index) {
    return array[index];
  }
  // … les 22 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 de mélange de liste :

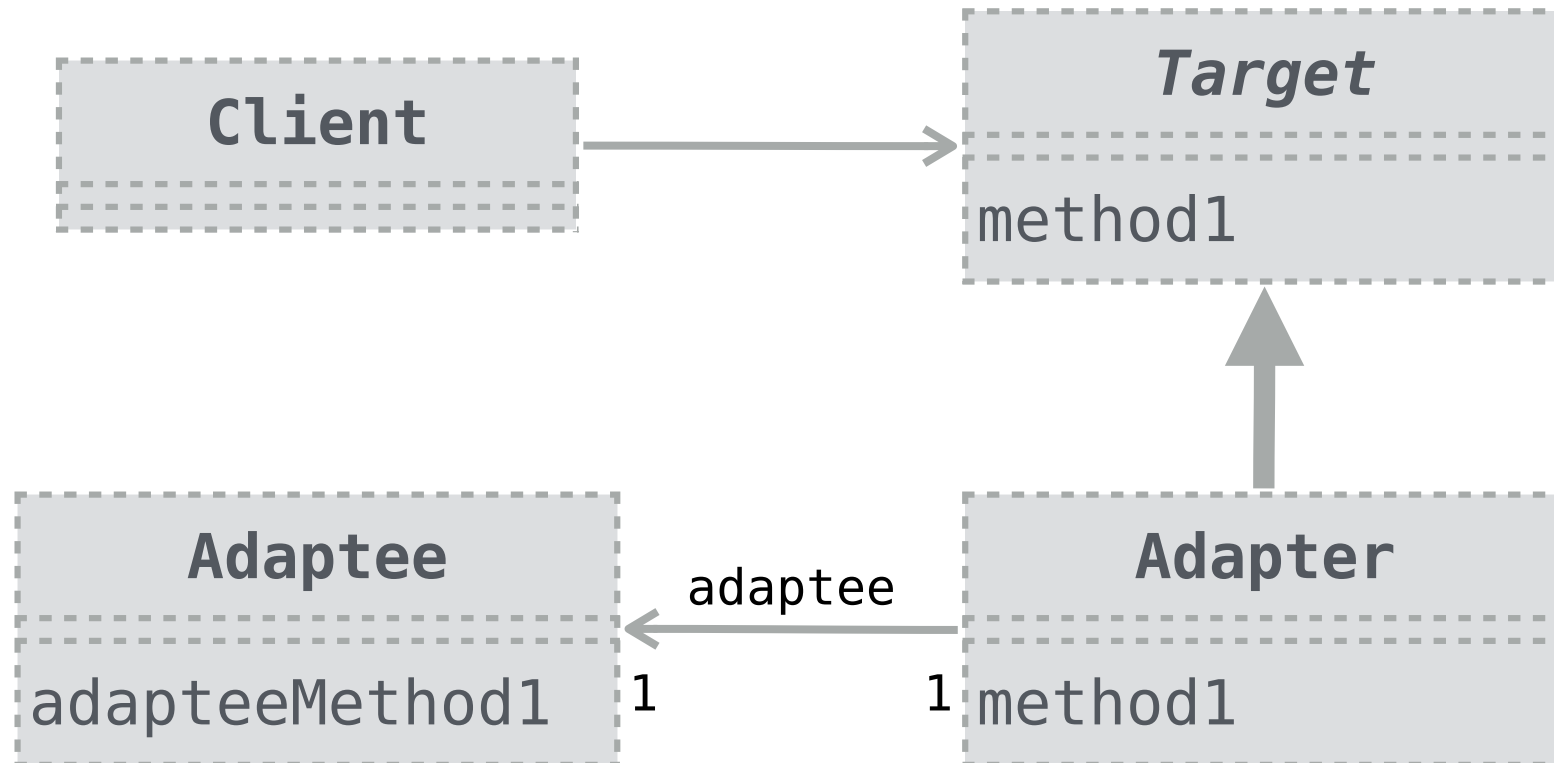
```
<E> void shuffleList(List<E> list) { … }

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

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



Exemple réel

La classe `java.util.Arrays` offre, sous la forme de méthodes statiques, un grand nombre d'opérations sur les tableaux.
Une de ces méthodes, nommée `asList`, permet justement d'adapter un tableau en liste.
Cette méthode utilise le patron *Adapter* pour faire cette adaptation, exactement comme nous l'avons fait.

`AbstractList`

(digression)

`java.util.List`

L'interface `java.util.List` contient 23 méthodes. Ecrire une classe implémentant cette interface représente donc un travail important.
Toutefois, un examen de cette interface montre d'une part que 9 méthodes sont optionnelles, et d'autre part que plusieurs méthodes peuvent facilement s'exprimer en termes d'un petit nombre de méthodes de base.

Opérations optionnelles

Dans la bibliothèque Java, certaines interfaces possèdent des méthodes correspondant à des opérations optionnelles.

Une classe implémentant une telle interface peut ne pas fournir ces opérations, c-à-d faire en sorte que les méthodes correspondantes lèvent l'exception `UnsupportedOperationException`.

Exemple que vous connaissez : `remove` de l'interface `Iterator`.

Opérations optionnelles

Dans la bibliothèque Java, les méthodes qui modifient les collections sont en général optionnelles. Cela permet la définition de collections non modifiables.
Ainsi, les méthodes `add`, `addAll`, `clear`, `remove`, `removeAll`, `retainAll` et `set` de l'interface `List` sont optionnelles.

Méthodes de base de `List`

L'interface `List` possède plusieurs méthodes qui sont facilement exprimables en fonction de méthodes de base.

Par exemple, au moyen de la méthode `size` il est trivial de fournir la méthode `isEmpty` :

```
public boolean isEmpty() {
  return size() == 0;
}
```

La classe `AbstractList`

L'idée de la classe `java.util.AbstractList` est de définir autant de méthodes de `List` que possible en termes d'un petit nombre de méthodes de base, qui sont laissées abstraites. Les méthodes correspondant à des opérations optionnelles lèvent quant à elle l'exception `UnsupportedOperationException`
En héritant de cette classe et en définissant les méthodes de base, on peut ainsi facilement définir un nouveau type de listes.

Adaptateur pour tableaux

La classe `AbstractList` nous permet d'écrire beaucoup plus simplement notre adaptateur pour tableaux.
Au lieu de définir les 23 méthodes de l'interface `List`, il nous suffit d'hériter d'`AbstractList` et de définir les trois méthodes de base requises : `get`, `set` et `size`. C'est tout !

Adaptateur pour tableaux

```
import java.util.AbstractList;
public final class ArrayAdapter<E>
    extends AbstractList<E> {
  private E[] array;
  public ArrayAdapter(E[] array) {
    this.array = array;
  }
  @Override
  public E get(int index) { ??? }
  @Override
  public E set(int index, E element) { ??? }
  @Override
  public int size() { ??? }
}
```

Patron n°6 :

Observer

Illustration du problème : tableur

Cellules d'un tableur

Dans un tableur, une feuille de calcul est composée de cellules qui contiennent soit des données, soit des formules. Une formule décrit comment calculer la valeur de la cellule en fonction de la valeur d'autres cellules.

Exemple (les formules commencent par le caractère '='):

	A	B	C
1	10	6	
2	12	12	
3	=A1+A2	=B1+B2	=A3+B3
	22	18	40

Illustration du problème

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 notre exemple, 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 ?

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 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.

Tableur simplifié

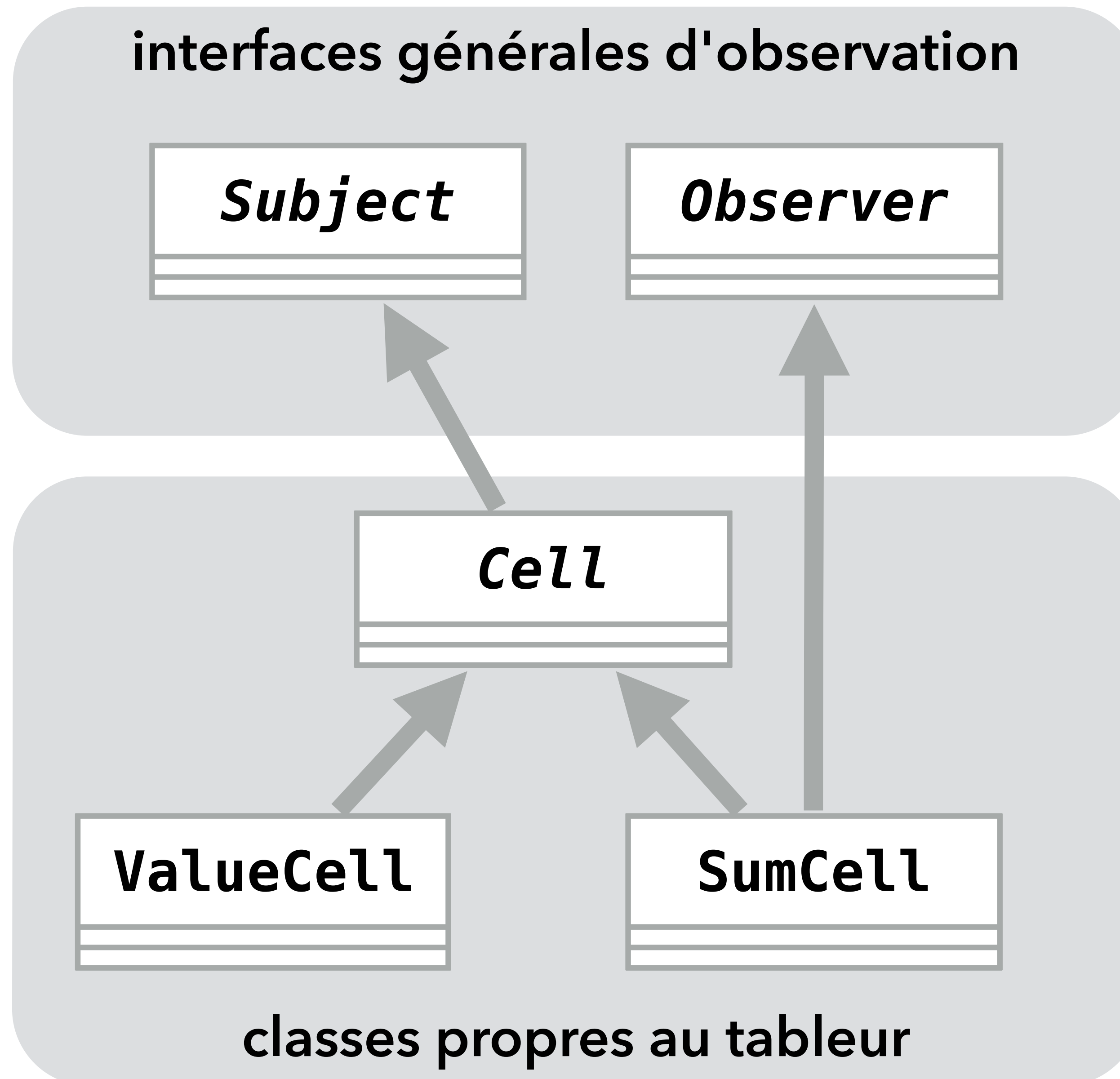
Afin d'illustrer la solution, modélisons un tableur très simple dont les cellules contiennent soit un nombre, soit une formule.
Pour simplifier les choses, nos formules sont toutes des sommes de la valeur de deux cellules – c-à-d qu'elles ont la forme `=C1+C2` où `C1` et `C2` sont des cellules.

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`.

Diagramme de classes



Observateur

Un observateur doit pouvoir être informé des changements du (ou des) sujet(s) qu'il observe.

A cet effet, il possède une méthode 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.

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

Sujet

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.

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

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.

Cellule

```
public abstract class Cell implements Subject{
  private Set<Observer> observers =
    new HashSet<Observer>();
  abstract public double value();
  public void addObserver(Observer o) {
    observers.add(o);
  }
  public void removeObserver(Observer o) {
    observers.remove(o);
  }
  protected void notifyObservers() {
    // ???
  }
}
```

Cellule de valeur

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

```
public final class ValueCell extends Cell {
  private double value = 0;
  public double value() {
    return value;
  }
  public void setValue(double newValue) {
    // ???
  }
}
```

Cellule somme

Une cellule somme diffère d'une cellule normale 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.

Cellule somme

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

  public SumCell(Cell c1, Cell c2) {
    this.c1 = c1;
    this.c2 = c2;
    c1.addObserver(this);
    c2.addObserver(this);
  }
  // … suite sur la prochaine page
```

Cellule somme

```
  // … suite de la classe SumCell
  public void update(Subject s) {
    // ???
  }

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

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.

Tableur

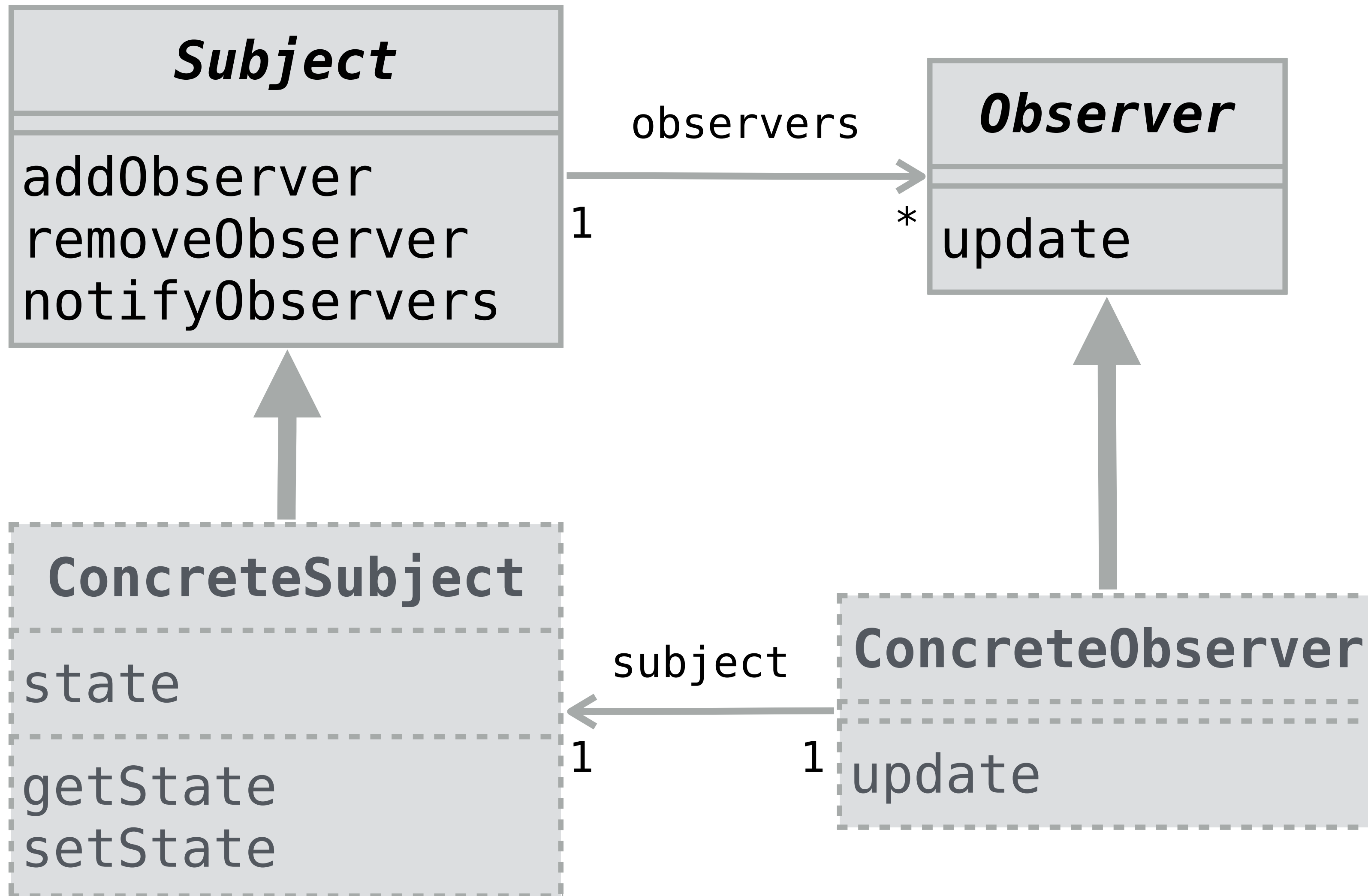
```
public final class SpreadSheet
    implements Observer {
  public SpreadSheet() {
    ValueCell A1 = new ValueCell(), A2 = …;
    SumCell A3 = new SumCell(A1, A2);
    // … C3
    A1.addObserver(this); // A2 … C3 idem
    A1.setValue(10);      // A2 … C3 idem
  }
  public void update(Subject s) {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

Généralisation : patron *Observer*

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*.

Diagramme de classes



Intérêt 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, p.ex.

Inconvénients du patron

Le patron *Observer* possède également plusieurs inconvénients :

- il impose un style de programmation impératif, dans lequel les objets ont un état modifiable,
- 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…

Observer et mutabilité

Le patron *Observer* est fondamentalement impératif, dans le sens où tout sujet doit avoir un état modifiable. Observer un objet non modifiable n'est d'aucune utilité !

L'utilisation du patron *Observer* force donc un style de programmation impératif, qui a lui-même plusieurs problèmes :

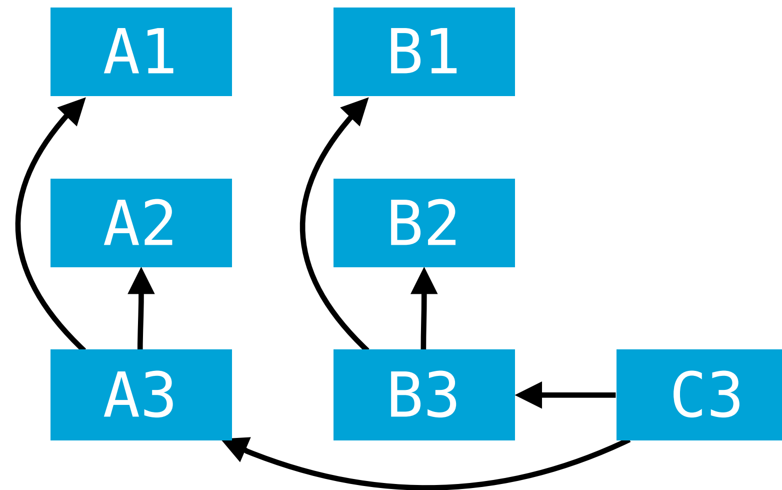
- il rend les programmes difficiles à comprendre,
- il interagit mal avec la concurrence.

Graphe d'observation

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).

Graphe d'observation

Le graphe d'observation de notre tableur simplifié est :



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

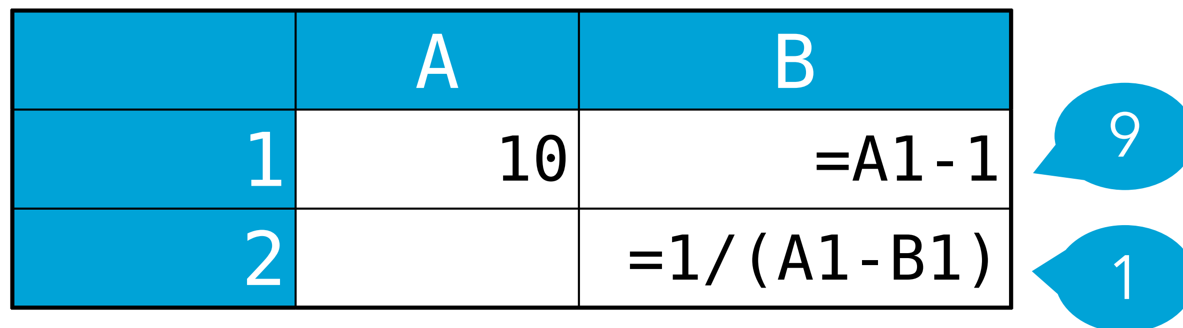
	A	B	C
1	10	6	
2	12	12	
3	=A1+A2	=B1+B2	=A3+B3

Cycles d'observation

Normalement, le graphe d'observation devrait être acyclique (c-à-d 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.
(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*.)

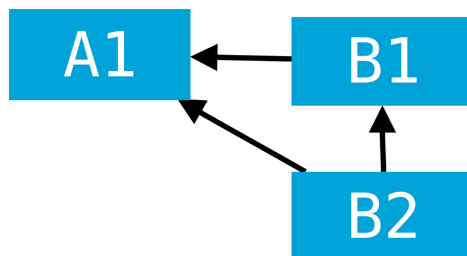
Glitches

Même en l'absence de cycles d'observation, le patron *Observer* peut poser problème. On peut les 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 **`A1`** ?

Glitches

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.
(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*).

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.

Patron n°7 :

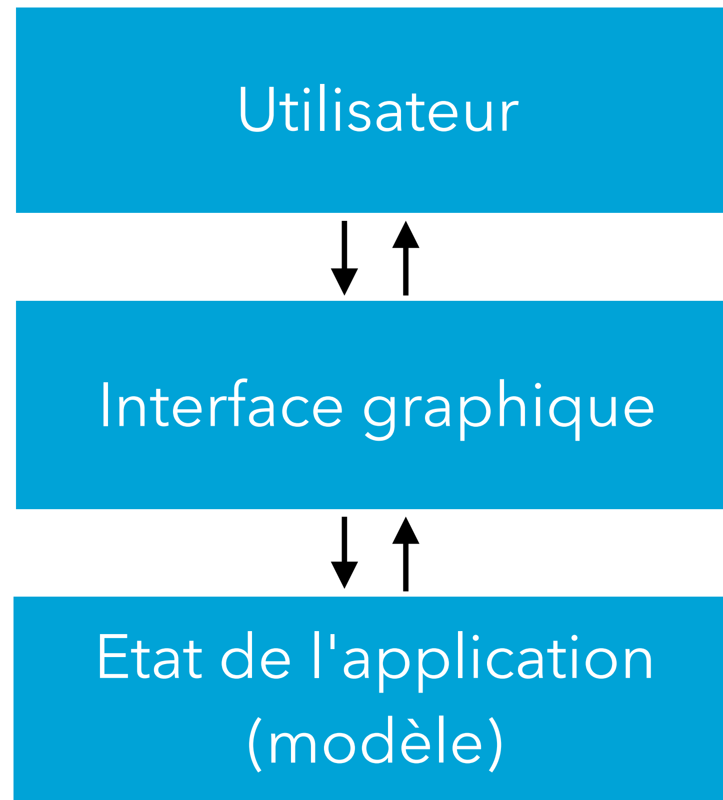
MVC

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 ?

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.



Généralisation

Le découplage entre la gestion de l'interface utilisateur et la logique propre à l'application peut se faire de différentes manières.

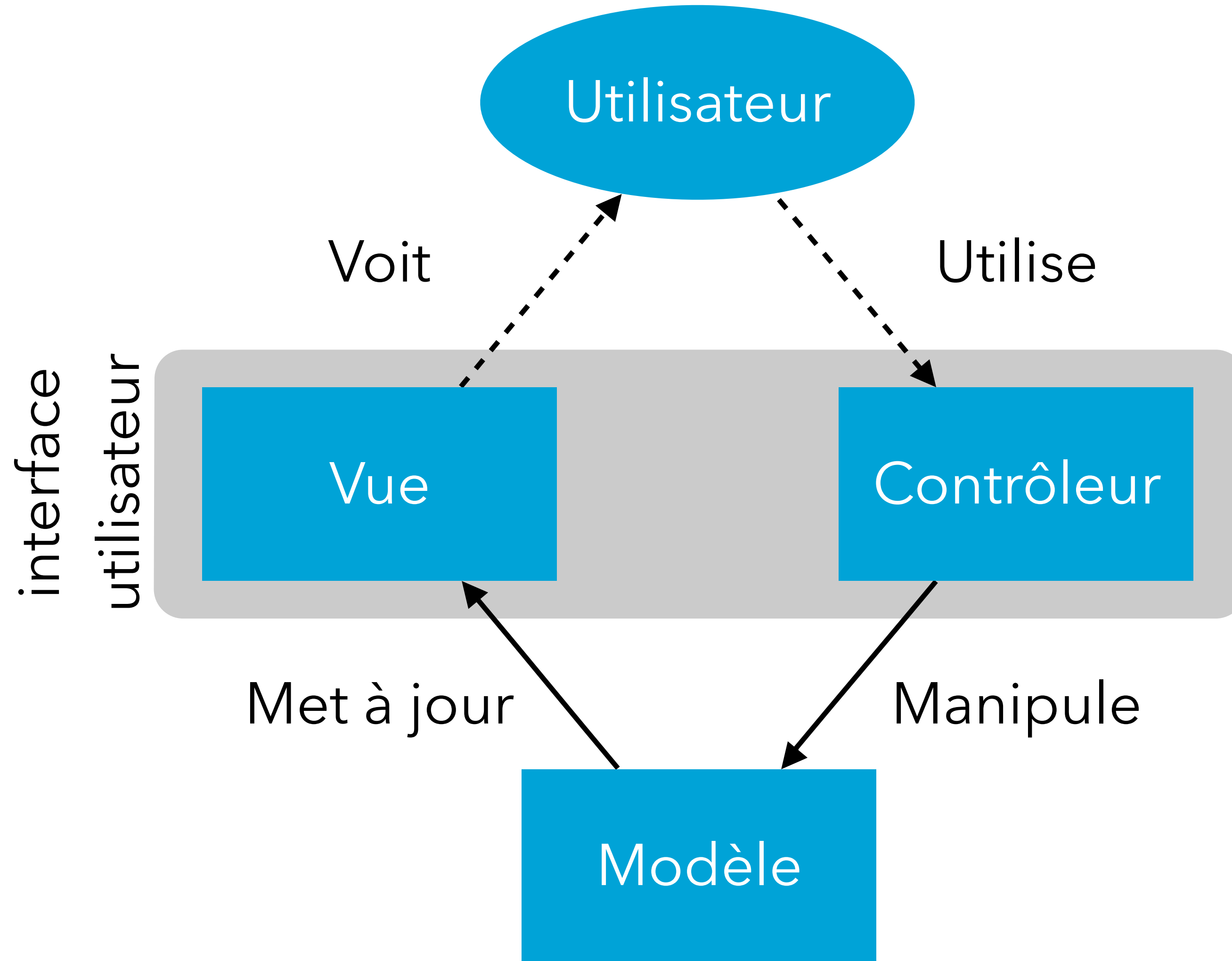
Le patron *Model-View-Controller* (*MVC*) propose d'organiser le code du programme en trois catégories :

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

Patron architectural

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

MVC



Modèle

Le **modèle** 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.

Vue

La **vue** 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.

Contrôleur

Le **contrôleur** 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.

MVC et *Observer*

Le patron *Observer* joue un rôle central dans la mise en œuvre du patron *MVC*.
P.ex. 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*.

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.

Exemples réels

La plupart des bibliothèques d'interface utilisateur graphique utilisent le patron *MVC* ou 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*,
- etc.

Résumé

Le patron *Observer* permet à un objet d'être informé des changements d'états d'un autre objet sans que les deux ne soient couplés. Souvent, l'objet observé n'a aucune connaissance détaillée de ses observateurs.

Le patron (architectural) *MVC* permet d'organiser un programme en découplant totalement le code gérant l'interface utilisateur de celui gérant la logique propre à l'application.