Objet et développement d'applications
Cours 4 - Pattern Observer

Patrons de conception pour POO

Florian Richoux

2014-2015

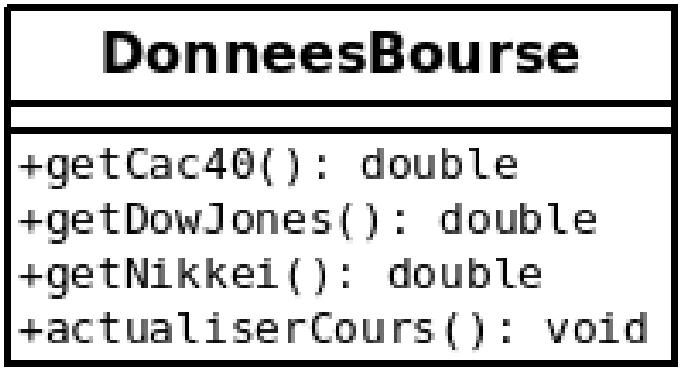
Suivre le cours de la bourse

Affichage de la bourse

On vous donne la description d'une classe DonneesBourse qui contient des données sur le cours de la bourse en temps réel et on vous demande d'écrire un programme avec trois modes d'affichages :

- Cours actuel.
- Statistiques.
- Prédiction de demain.

Description de DonneesBourse



Les méthodes de DonneesBourse

- Accesseurs sur les valeurs de DonneesBourse
- actualiserCours() appellée quand ces valeurs changent.

On commence à modifier DonneesBourse

Ajout de la méthode actualiserAffichages()

```
class DonneesBourse
{
  // Méthodes définies à l'origine
  // getCac40(), ...

  public void actualiserCours()
  {
    double cac40 = getCac40();
    double dowJones = getDowJones();
    double nikkei = getNikkei();

    cours.actualiser( cac40, dowJones, nikkei );
    stats.actualiser( cac40, dowJones, nikkei );
    prevision.actualiser( cac40, dowJones, nikkei );
  }
}
```

Stop !

On peut déjà s'arrêter là

Avec `cours`, `stats` et `prevision`, on implémente des classes concrètes.

Stop !

On peut déjà s'arrêter là

Avec `cours`, `stats` et `prevision`, on implémente des classes concrètes.

Et ?

Stop !

On peut déjà s'arrêter là

Avec `cours`, `stats` et `prevision`, on implémente des classes concrètes.

Et ?

Rappelez-vous des canards...

Et alors la suppression ou l'ajout d'éléments à afficher devra passer par la modification du code.

Stop !

On peut déjà s'arrêter là

Avec `cours`, `stats` et `prevision`, on implémente des classes concrètes.

Et ?

Rappelez-vous des canards...

Et alors la suppression ou l'ajout d'éléments à afficher devra passer par la modification du code.

Rappel : objectif "maintenance facile"

On souhaite faciliter la maintenance du code, donc devoir le modifier le moins possible.

Repérer ce qui ne change pas

Interface commune

```
cours.actualiser( cac40, dowJones, nikkei );
stats.actualiser( cac40, dowJones, nikkei );
prevision.actualiser( cac40, dowJones, nikkei );
```

Idée du pattern Observer (Observateur)

Fonctionne comme un flux RSS

1. Les news internationales du monde.fr vous intéresse, vous souhaitez rester informé.
2. Vous vous abonnez au flux RSS des new internationales.
3. Quand un nouvel article est publié, vous en êtes averti.
4. Si cela ne vous intéresse plus, vous vous désabonnez.

Pattern Observer

Le pattern Observer fonctionne sur le même principe qu'un flux RSS, à une différence de vocabulaire :

- Les news du monde.fr est ici le sujet.
- Vous, vous êtes les observateurs.

Pattern Observer

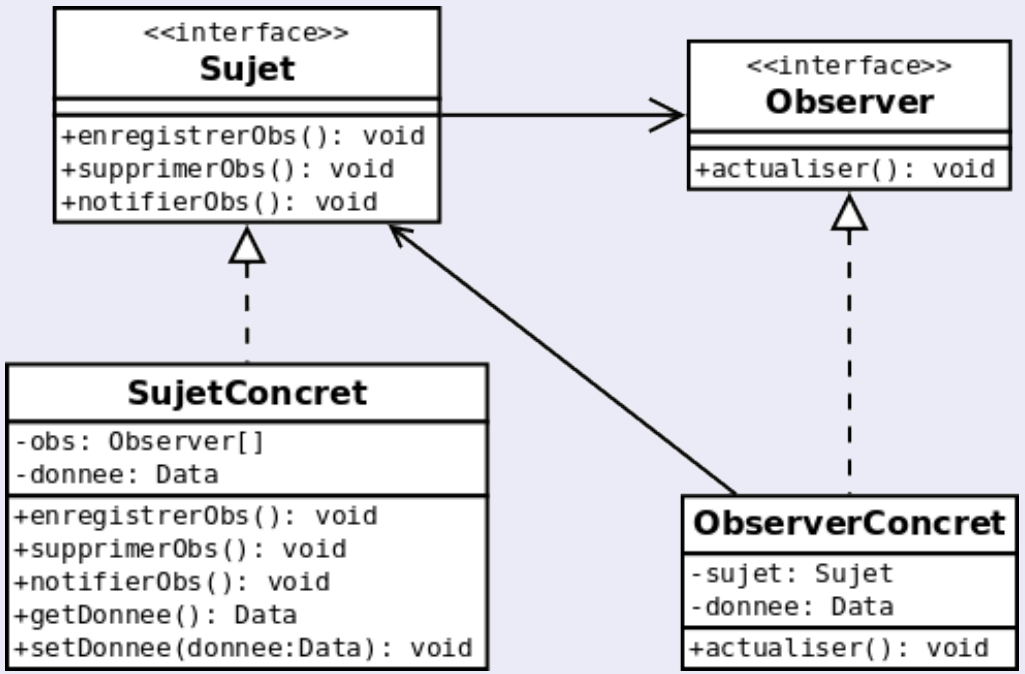
Définition

Le pattern Observer (Observateur) définit une relation entre objets de type un-à-plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

Relation sujet-observateurs

Les observateurs sont donc "dépendants" du sujet : quand l'état du sujet change, les observateurs en sont informés.

Diagramme UML



Faible couplage

Communiquer sans se connaître

Les observateurs et les sujets ne savent pratiquement rien des uns les autres.

Principe de découplage

On vérifie le principe de découplage : des classes faiblement liées seront plus faciles à modifier sans mauvaise surprise.

On pourra aussi réutiliser ailleurs des sujets ou observateurs sans problème.

Faible couplage

En détails

- Un sujet sait seulement qu'un observateur implémente une certaine interface. Il ne connait pas les observateurs concrets.
- L'ajout et la suppression d'observateurs à n'importe quel moment ne change rien pour un sujet : il continuera à mettre à jour les informations comme d'habitude.
- Pas besoin de modifier le sujet pour ajouter un nouveau type d'observateurs.

Pattern Observer pour les cours de la bourse

Schéma

En java, les interfaces

Interface Sujet

```
public interface Sujet
{
  public void enregistrerObs(Observateur o);
  public void supprimerObs(Observateur o);
  public void notifierObs();
}
```

Interface Observateur et Affichage

```
public interface Observateur
{
  public void actualiser(double cac40, double dowJones,
                         double nikkei);
}

public interface Affichage
{
  public void afficher();
}
```

Class DonneesBourse 1/4

```
public class DonneesBourse implements Sujet
{
  private ArrayList<Observateur> observateurs_;
  private double cac40_;
  private double dowJones_;
  private double nikkei_;

  public DonneesBourse()
  {
    observateurs_ = new ArrayList();
  }

  ...
}
```

Implémenter Sujet

Class DonneesBourse 2/4

```
public class DonneesBourse implements Sujet
{
  ...

  public void enregistrerObs(Observateur o)
  {
    observateurs_.add(o);
  }

  public void supprimerObs(Observateur o)
  {
    observateurs_.remove(o);
  }

  ...
}
```

Class DonneesBourse 3/4

```
public class DonneesBourse implements Sujet
{
   ...

   public void notifierObs()
   {
      for( int i = 0; i < observateurs_.size(); ++i)
      {
         observateurs_.get(i).actualiser(cac40_,
                                         dowJones_,
                                         nikkei_);
      }
   }

   ...
}
```

Class DonneesBourse 4/4

```
public class DonneesBourse implements Sujet
{
   ...

   public void setCours(double cac40, double dowJones,
                        double nikkei)
   {
      cac40_     = cac40;
      dowJones_  = dowJones;
      nikkei_    = nikkei;

      notifierObs();
   }

   // Accesseurs et autres methodes
}
```

Classe AffichageCours 1/2

```
public class AffichageCours
                implements Observateur, Affichage
{
  private double cac40_;
  private double dowJones_;
  private double nikkei_;
  private Sujet  donneesBourse_;

  public AffichageCours(Sujet donneesBourse)
  {
    donneesBourse_ = donneesBourse;
    donneesBourse_.enregistrerObs(this);
  }

  ...
}
```

Classe AffichageCours 2/2

```
public void actualiser(double cac40, double dowJones,
                       double nikkei)
{
  cac40_    = cac40;
  dowJones_ = dowJones;
  nikkei_   = nikkei;
  afficher();
}

public void afficher()
{
  System.out.println(...);
}
```

Notre programme

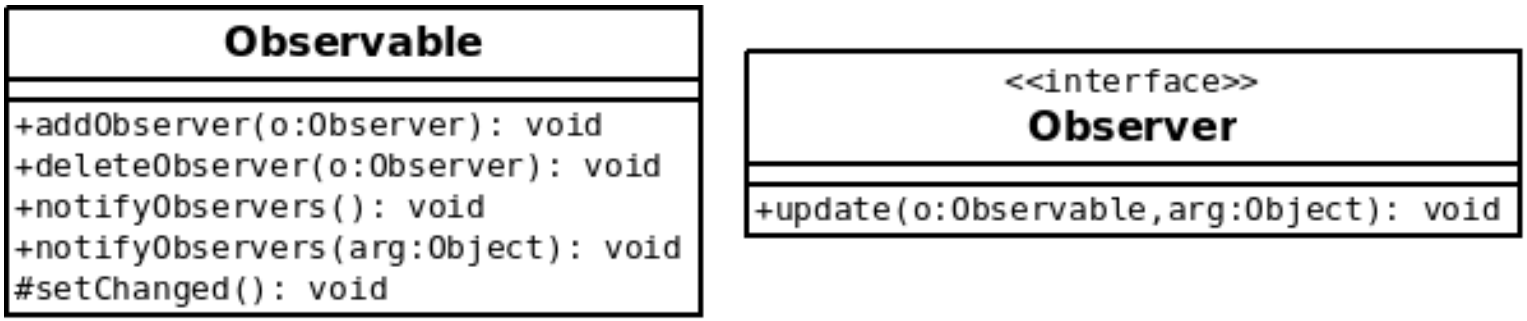
Programme principal

```
public class Bourse
{
  public static void main(String [] args)
  {
    DonneesBourse data = new DonneesBourse();

    AffichageCours affCours = new AffichageCours(data);
    AffichageStats affStats = new AffichageStats(data);
    AffichagePrev affPrev = new AffichagePrev(data);

    data.setCours(3376, 13443, 8596);
    data.setCours(3245, 13224, 8612);
    data.setCours(3189, 13378, 8703);
  }
}
```

Le pattern Observer est dans l'API Java !

`java.util.Observer` et `java.util.Observable`

- Un sujet concret héritera de la classe `Observable` de Java.
- Un observateur concret devra lui implémenter l'interface `Observer`.



Comment ça marche

Pour devenir Observateur

Comme avant : implémenter `Observer` et appeler la méthode `addObserver` du sujet (et `deleteObserver` pour se désinscrire).

Pour devenir Sujet

- Devenir `Observable` par héritage,
- **Nouveau :** d'abord appeler la méthode `setChanged` pour signaler que des valeurs ont changé.
- Appeler soit `notifyObservers()` soit `notifyObservers(Object arg)`.

Comment ça marche

Pour recevoir les changements

Un observateur appellera `update(Observable o, Object arg)`.

- `arg` sera l'objet transmit par `notifyObservers(Object arg)`.
- Si un tel objet n'existe pas, `arg` sera null.

Intuitivement

- Avec `notifyObservers(Object arg)`, seules les données encapsulées dans arg sont transmises.
- Avec `notifyObservers()`, c'est l'objet sujet entier qui est transmis.

Comment est faite la classe Observable

Dans l'API

```
setChanged()
{
  changed = true;
}

notifyObservers(Object arg)
{
  if( changed )
  {
    // notifier tout le monde
  }

  changed = false;
}

notifyObservers()
{
  notifyObservers(null);
}
```

Class DonneesBourse 1/2

```
import java.util.*;
class DonneesBourse extends Observable {

  // plus d'ArrayList pour les observateurs

  private double cac40_;
  private double dowJones_;
  private double nikkei_;

  public DonneesBourse() { } // RAS

  public void actualiserCours() {
    setChanged();
    notifyObservers();
  }

  ...
}
```

Class DonneesBourse 2/2

```
class DonneesBourse extends Observable {
  ...

  public void setCours(double cac40, double dowJones,
                       double nikkei)
  {
    cac40_    = cac40;
    dowJones_ = dowJones;
    nikkei_   = nikkei;

    actualiserCours(); // 1) setChanged, 2) notify
  }
}
```

Class AffichageCours 1/2

```
import java.util.*;
class AffichageCours implements Observer, Affichage
{
  private Observable donneesBourse_;
  private double cac40_;
  private double dowJones_;
  private double nikkei_;

  public AffichageCours(Observable data)
  {
    donneesBourse_ = data;
    data.addObservers(this;)
  }

  ...
}
```

Class AffichageCours 2/2

```
class AffichageCours implements Observer, Affichage
{
   ...

   public void update(Observable data, Object arg)
   {
      if( data instanceof DonneesBourse )
      {
         DonneesBourse bourse = (DonneesBourse)data;
         cac40 = bourse.getCac40();
         dowJones = bourse.getDowJones();
         nikkei = bourse.getNikkei();

         afficher();
      }
   }
}
```

Deux problèmes avec ce pattern dans l'API

1) Ordre de notification

Dans la description officielle de `Observable` : *The order in which notifications will be delivered is unspecified.*

Ainsi, ne **jamais** utiliser le pattern Observer de l'API Java si vous avez un programme qui dépend d'un ordre de notification spécifique.

2) `Observable` est une classe

`Observable` n'étant pas une interface, votre sujet doit dériver de `Observable` et ne peut donc dériver de rien d'autre !

Vous ne pouvez même pas créer une instance de `Observable` et la composer avec vos classes car `setChanged` est **protected**, donc hors d'accès à une classe qui n'hérite pas de `Observable` !

Le pattern Observer dans la JDK

Un monde observable

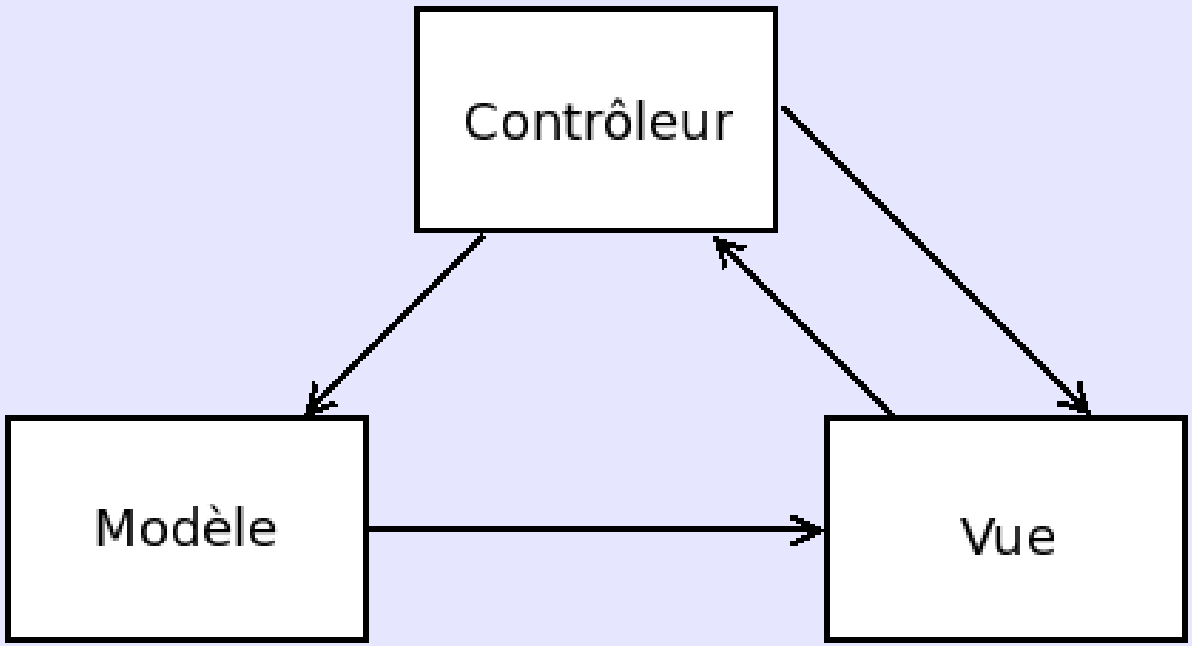
On retrouve le pattern Observer partout dans l'API de Java. Par exemple, il est omniprésent en Swing.

Exemple : JButton

Lors que l'on appelle la méthode `addActionListener` dans un JButton, on ajoute en réalité un observateur de notre bouton.

Pattern d'architecture *Modèle-Vue-Contrôleur*

- **Le modèle** : contient les données du programme.
- **La (les) vue(s)** : est l'IHM, affichant les données qu'on lui fournit.
- **Le contrôleur** : fait la synchronisation entre le modèle et les vues. C'est lui qui reçoit les requêtes de l'utilisateur.



Avantages du MVC

- Les trois modules sont faiblement couplés : par exemple on peut en réécrire le modèle sans modifier le reste.
- La synchronisation des vues grâce au pattern Observer.

Inconvéniants du MVC

- Parfois complexe à mettre en place, surtout si le code n'est pas bien structuré dès le départ.

La prochaine fois

Le pattern Observer est le dernier pattern de comportement que l'on étudiera.

La prochaine fois, on verra notre premier pattern de structure.