Patrons Observateur/MVC
programmation évènementielle

jean-michel Douin, douin au cnam point fr
version : 14 Septembre 2010

Notes de cours

Sommaire

- **Patron Observateur**
- **Programmation évènementielle**
- **Patron MVC Modèle Vue Contrôleur**
- **Patron Publish/subscribe**

Principale bibliographie utilisée

- [Grand00]
  - Patterns in Java le volume 1
    http://www.mindspring.com/~mgrand/
- [head First]
  - Head first : http://www.oreilly.com/catalog/hfdesignpat/#top
- [DP05]
  - L'extension « Design Pattern » de BlueJ : http://hamilton.bell.ac.uk/designpatterns/
- [divers]
  - Certains diagrammes UML : http://www.dofactory.com/Patterns/PatternProxy.aspx
  - informations générales http://www.edlin.org/cs/patterns.html

Patrons/Patterns pour le logiciel

- **Origine C. Alexander un architecte**
- **Abstraction dans la conception du logiciel**
  - **[GoF95] la bande des 4 : Gamma, Helm, Johnson et Vlissides**
    - **23 patrons/patterns**
- **Architectures logicielles**

Introduction : rappel

- **Classification habituelle**
  - Créateurs
    - Abstract Factory, Builder, Factory Method Prototype Singleton
  - Structurels
    - Adapter Bridge Composite Decorator Facade Flyweight Proxy
  - Comportementaux

    Chain of Responsability. Command Interpreter Iterator

    Mediator Memento **Observer** State

    Strategy Template Method Visitor

Les patrons déjà vus …

- **Adapter**
  - Adapte l'interface d'une classe conforme aux souhaits du client
- **Proxy**
  - Fournit un mandataire au client afin de contrôler/vérifier ses accès

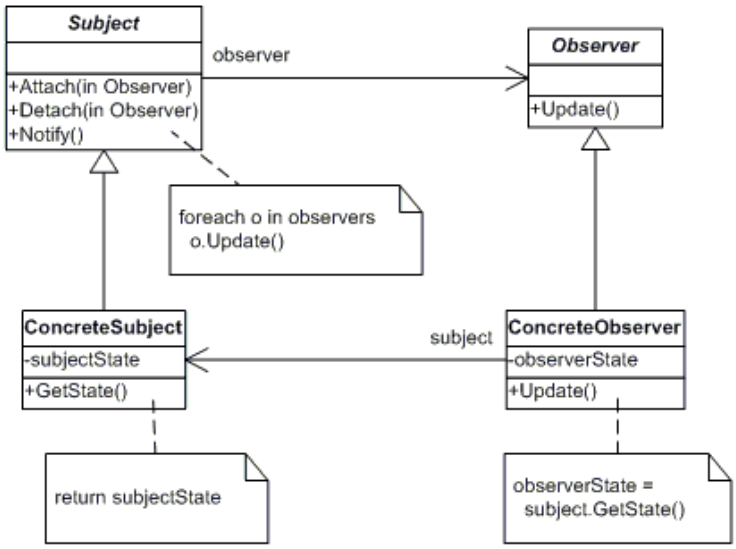
Patron Observer/observateur

– *Notification d'un changement d'état d'une instance aux observateurs inscrits*

- **Un Observé**
  - **N'importe quelle instance qui est modifiée**
    - **i.e. un changement d'état comme par exemple la modification d'une donnée d'instance**
- **Les observateurs seront notifiés**
  - **A la modification de l'observé,**
  - **Synchrone, (et sur la même machine virtuelle …)**
- **Un ou plusieurs Observés / un ou plusieurs observateurs**
- **Ajout et retrait dynamiques d'observateurs**

ESIEE 7

UML & le patron Observateur, l'original



- **http://www.codeproject.com/gen/design/applyingpatterns/observer.gif**

ESIEE 8

Le patron observateur en Java

– Lors d'un changement d'état : notification aux observateurs inscrits

```
// les observés
public interface Subject{
  public void attach(Observer o);
  public void detach(Observer o);

  public void notify();
}

// les observateurs
public interface Observer{
  public void update();
}
```


ConcreteObservable

```
public class ConcreteSubject implements Subject{

  private Collection<Observer> observers = new ……


  public void attach(Observer observer){
    observers.add(observer);
  }

  public void detach(Observer observer){
    observers.remove(observer);
  }


public void notify(){
    for(Observer obs : observers)
      obs.update();
    }
}
```

ConcreteObserver

```
public class ConcreteObserver implements Observer{

  public void update(){

   // une notification a eu lieu ….
  }

}
```

Quelle est l'observé initiateur ?

Où sont les paramètres ?

Observer : mise en oeuvre

```
Observable o = new ConcreteObservable();
Observer obs1= new ConcreteObserver();

o.addObserver(obs1);

o.notify();  // obs1 est réveillé


Observer obs2= new ConcreteObserver();


o.addObserver(obs2);


o.notify();  // obs1 et obs2 sont réveillés…
```

Démonstration / discussion

Observer et ses paramètres …

– **A chaque notification l'observé est transmis**

```
public interface Observer{
    public void update(Observable obs);
}
^
```

– **A chaque notification l'observé et un paramètre sont transmis**

```
public interface Observer{
    public void update(Observable obs, Object param);
}
```

Existe déjà, voir java.util

- **java.util.Observable** ***Subject***
- **java.util.Observer** ***Observer***
- ***xxxxxxListener Quel rapport ?***
  - ***Interface AcionListener …. Et les autres***

java.util, java.awt.event et plus

- **java.util.Observer** **&** **java.util.Observable**
  - **update**
  - **addObserver**
  - **removeObserver**
  - **notifyObservers**
  - **….**

EventListener

- **java.awt.event.EventListener**
  - Les écouteurs/observateurs
- **Convention syntaxique de Sun pour ses API**
  - XXXXXListener extends EventListener
    « update » addXXXXXListener

    exemple l'interface ActionListener → addActionListener
- **EventObject comme ActionEvent**

java.util.Observer

```
public interface Observer{
   void update(Observable o, Object arg);
}
```

L'Observé est transmis en paramètre `Observable o`

accompagné éventuellement de paramètres `Object arg`

« update » est appelée à chaque notification

java.util.Observable

```
public class Observable{
  public void addObserver(Observer o) ;
  public void deleteObserver(Observer o) ;
  public void deleteObservers() ;
  public int countObservers() ;

  public void notifyObservers() ;
  public void notifyObservers(Object arg) ;

  public boolean hasChanged() ;
  protected  void setChanged() ;
  protected  void clearChanged() ;
}
```


Un Exemple : une liste et ses observateurs

- **Une liste est observée, à chaque modification de celle-ci, ajout, retrait, … les observateurs inscrits sont notifiés**

```
public class Liste<E> extends java.util.Observable{
  …
  public void ajouter(E e){
    … // modification effective de la liste
    setChanged();       // l'état de cette liste a changé
    notifyObservers(e); // les observateurs sont prévenus
}
```

Une liste ou n'importe quelle instance …

Un Exemple : un observateur de la liste

```
Liste<Integer> l = new Liste<Integer>();

l.addObserver( new Observer(){
      public void update(Observable o,Object arg){
        System.out.print( o + " a changé, " );
        System.out.println( arg + " vient d'être ajouté !");
      }
      });
```

C'est tout ! démonstration

Démonstration/discussion

Observateur comme XXXXListener

java.util

Interface EventListener

All Known Subinterfaces:

Action, ActionListener, AdjustmentListener, AncestorListener, AWTEventListener, BeanContextMembershipListener, BeanContextServiceRevokedListener, BeanContextServices, BeanContextServicesListener, CaretListener, CellEditorListener, ChangeListener, ComponentListener, ConnectionEventListener, ContainerListener, ControllerEventListener, DocumentListener, DragGestureListener, DragSourceListener, DragSourceMotionListener, DropTargetListener, FlavorListener, FocusListener, HandshakeCompletedListener, HierarchyBoundsListener, HierarchyListener, HyperlinkListener, IIOReadProgressListener, IIOReadUpdateListener, IIOReadWarningListener, IIOWriteProgressListener, IIOWriteWarningListener, InputMethodListener, InternalFrameListener, ItemListener, KeyListener, LineListener, ListDataListener, ListSelectionListener, MenuDragMouseListener, MenuKeyListener, MenuListener, MetaEventListener, MouseInputListener, MouseListener, MouseMotionListener, MouseWheelListener, NamespaceChangeListener, NamingListener, NodeChangeListener, NotificationListener, ObjectChangeListener, PopupMenuListener, PreferenceChangeListener, PropertyChangeListener, RowSetListener, RowSorterListener, SSLSessionBindingListener, StatementEventListener, TableColumnModelListener, TableModelListener, TextListener, TreeExpansionListener, TreeModelListener, TreeSelectionListener, TreeWillExpandListener, UndoableEditListener, UnsolicitedNotificationListener, VetoableChangeListener, WindowFocusListener, WindowListener, WindowStateListener

- **Une grande famille !**

Une IHM et ses écouteurs

- **Chaque item est un sujet observable … avec ses écouteurs…**
  - **Pour un « Bouton », à chaque clic les écouteurs/observateurs sont prévenus**

```
public class Button extends Component{
  …
  public void addActionListener(ActionListener al){

  }
```

Un bouton prévient ses écouteurs …

Une instance de la classe java.awt.*Button*
prévient
ses instances inscrites java.awt.event.Action*Listener* …

```
Button b = new Button("empiler");
b.addActionListener(unEcouteur);      // 1
b.addActionListener(unAutreEcouteur); // 2
b.addActionListener(
      new ActionListener(){           // 3 écouteurs
        public void actionPerformed(ActionEvent ae){
          System.out.println("clic !!! ");
        }
      });
```


Un écouteur comme Action Listener

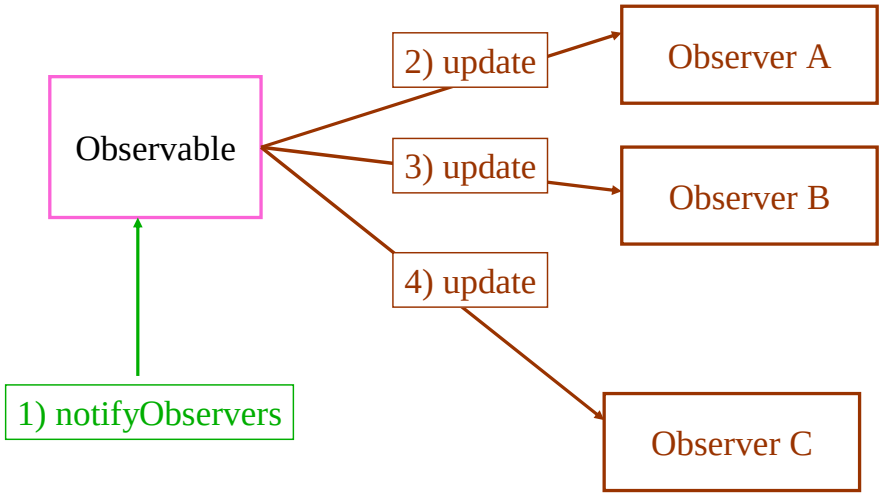
```
import java.util.event.ActionListener;
import java.util.event.ActionEvent;

public class EcouteurDeBouton
  implements ActionListener{

  public void actionPerformed(ActionEvent e){
   // traitement  à chaque action sur le bouton
  }

}

//c.f. page précédente
ActionListener unEcouteur = new EcouteurDeBouton();
b.addActionListener(unEcouteur);      // 1
```

Démonstration / Discussion

API Java, patron Observateur, un résumé

- **Ajout/retrait dynamiques des observateurs ou écouteurs**
- **L'observable se contente de notifier**
  - **Notification synchrone à tous les observateurs inscrits**
- **API prédéfinies java.util.Observer et java.util.Observable**

Observer distribué ? Un petit pas à franchir



- **Synchrones/asynchrones ?**
  - **Technologies java comme rmi, JMS (Java Messaging Service), JINI**
  - **cf. le Patron publish-subscribe**

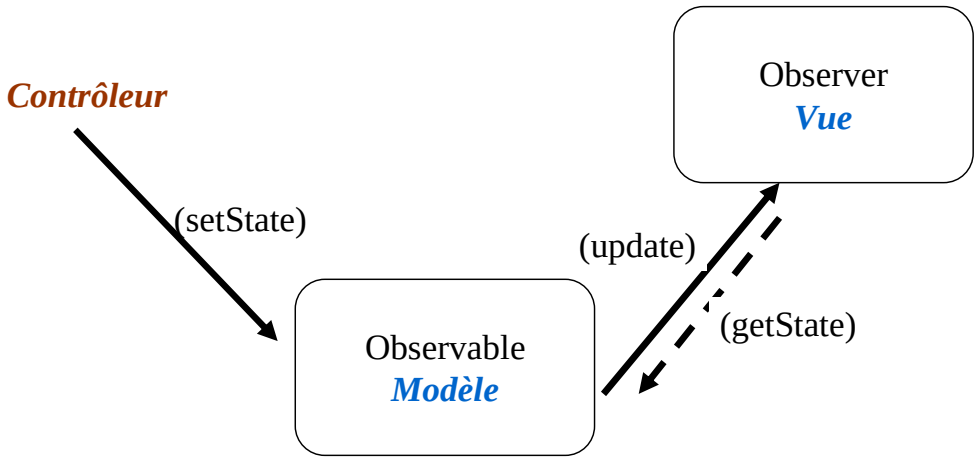
patrons Observer / MVC

Observés / Observateurs

- **Modèle** **Vue** **Contrôleur**

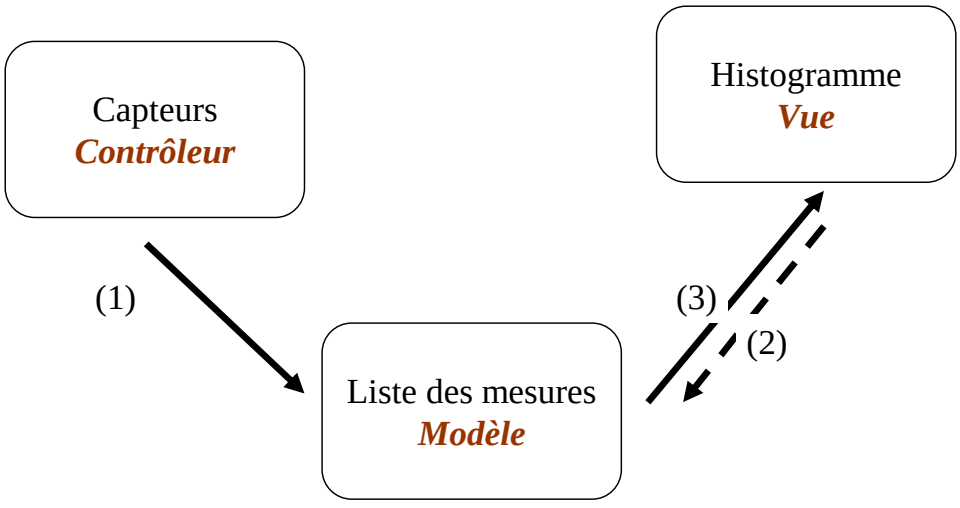
agit, modifie le Modèle

MVC : Observer est inclus



ESIEE 31

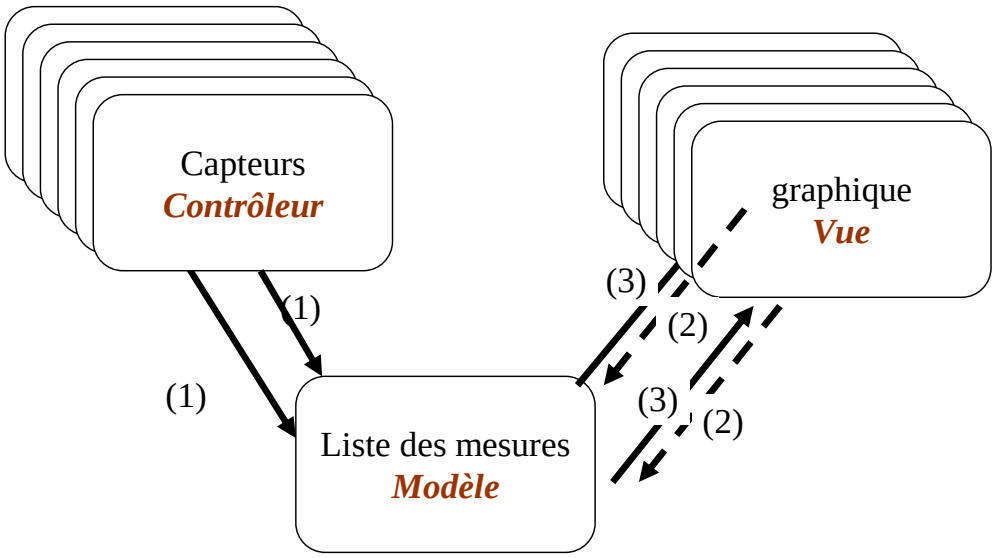
MVC : exemple de capteurs …



- **Une architecture simple et souple …**

ESIEE 32

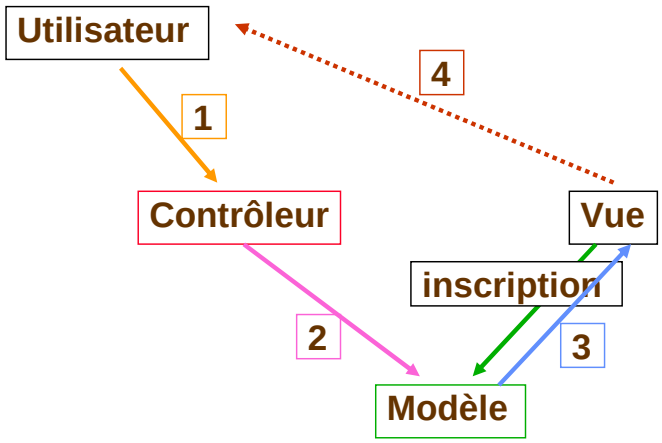
MVC Avantages



- **Souple ?**

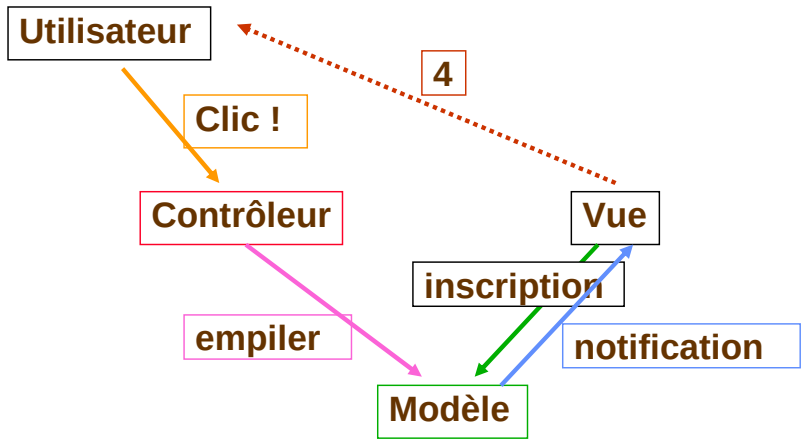
ESIEE 33

Un cycle MVC

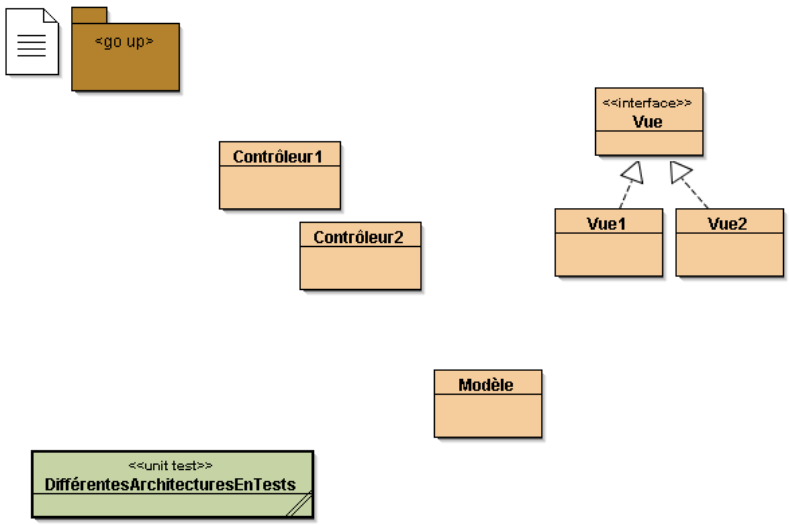


ESIEE 34

Un cycle MVC, le bouton empiler



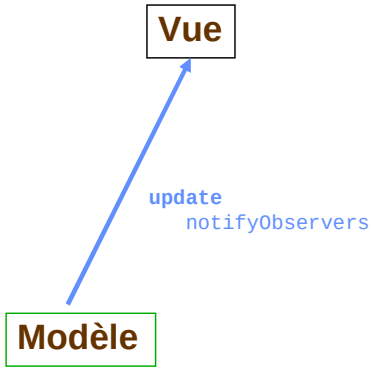
Démonstration / MVC en pratique



- **Un Modèle**
  - **Plusieurs Contrôleurs**
  - **Plusieurs Vues**

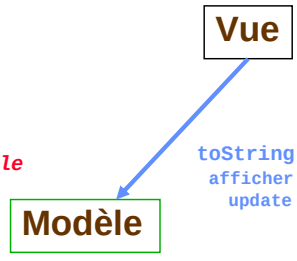
Démonstration : le Modèle i.e. un Entier

```
import java.util.Observable;

public class Modèle extends Observable{

  private int entier;

  public int getEntier(){
    return entier;
  }

  public String toString(){
    return "entier : " + entier;
  }

  public void setEntier(int entier){
    this.entier = entier;
    setChanged();
    notifyObservers(entier);
  }
}
```



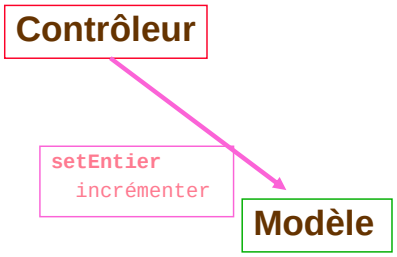
Démonstration : une Vue

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


import java.util.Observable;
import java.util.Observer;

public class Vue1 implements Vue, Observer{
  private Modèle modèle;

  public Vue1( Modèle modèle){   // inscription auprès du modèle
    this.modèle = modèle;
    modèle.addObserver(this);
  }

  public void afficher(){
    System.out.println(" Vue1 : le modèle a changé : " + modèle.toString());
  }

  public void update(Observable o, Object arg){    // notification
    if(o==modèle) afficher();
  }

}
```



Démonstration : un contrôleur

```
public class Contrôleur1{
  private Modèle modèle;

  public Contrôleur1(Modèle modèle){
    this.modèle = modèle;
  }

  public void incrémenter(){
    modèle.setEntier(modèle.getEntier() +1);
  }

}
```



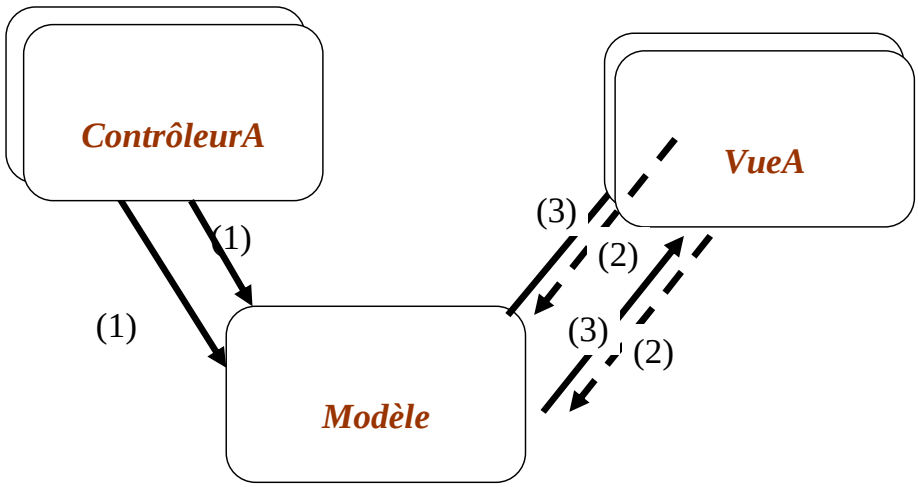
Un modèle, une vue, un contrôleur

```
    // Un Modèle
    Modèle modèle = new Modèle();

    // Ce modèle possède une vue
    Vue vue = new Vue1(modèle);

    // un Contrôleur ( déclenche certaines méthodes du modèle)
    Contrôleur1 contrôleur = new Contrôleur1(modèle);

    contrôleur.incrémenter();
    contrôleur.incrémenter();
  }
```

Un modèle, deux vues, deux contrôleurs

```
// Un Modèle
Modèle modèle = new Modèle();

// deux vues
Vue vueA = new Vue1(modèle);
Vue vueB = new Vue1(modèle);

// 2 Contrôleurs
Contrôleur1 contrôleurA = new Contrôleur1(modèle);
Contrôleur1 contrôleurB = new Contrôleur1(modèle);

contrôleurA.incrémenter();
contrôleurB.incrémenter();
```


Discussion



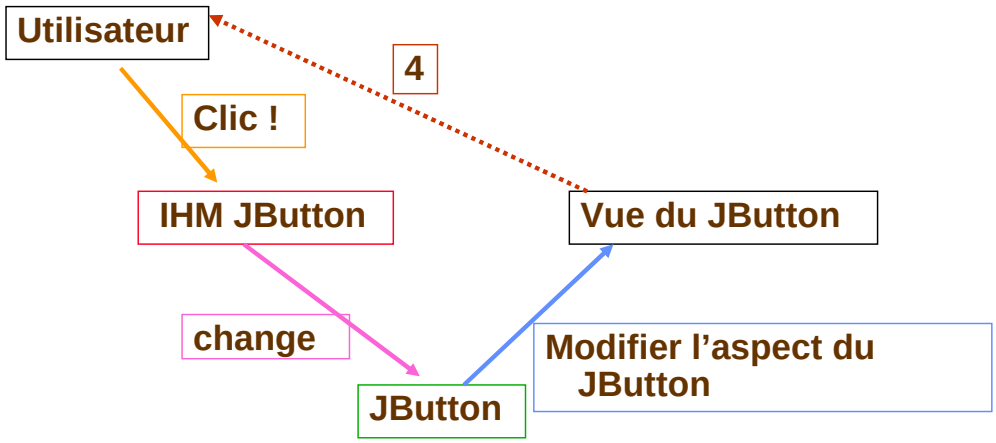
AWT / Button, discussion

- Un « Button » (le contrôleur) contient un MVC
  - À part entière

- Text, TextField, Label, … « sont » des Vues
- Button, Liste, … « sont » des contrôleurs

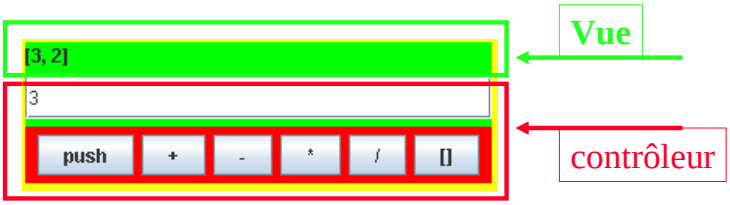
- Une IHM (JApplet,…) contient la Vue et le Contrôle
  - Alors le compromis architecture/lisibilité est à rechercher

Un JButton comme MVC



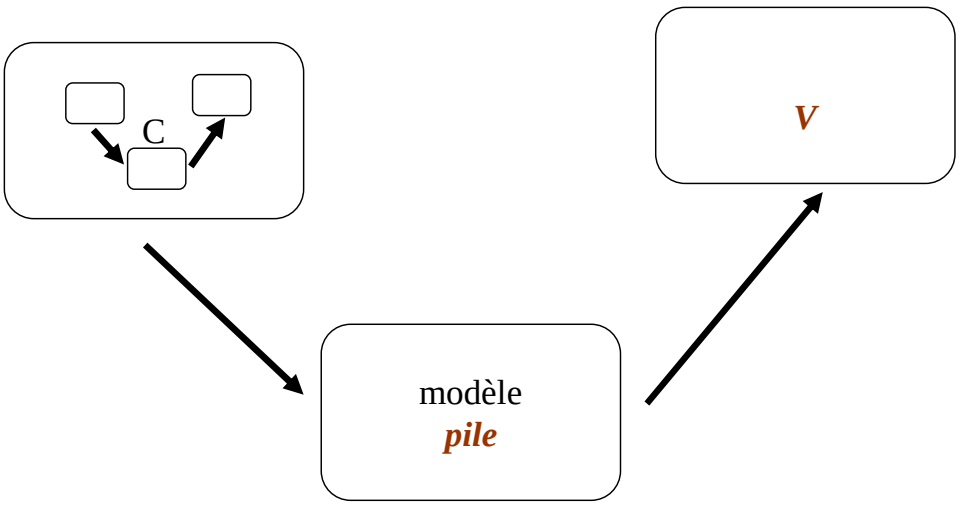
- Au niveau applicatif appel de tous les observateurs inscrits
  - actionPerformed(ActionEvent ae), interface ActionListener

Proposition



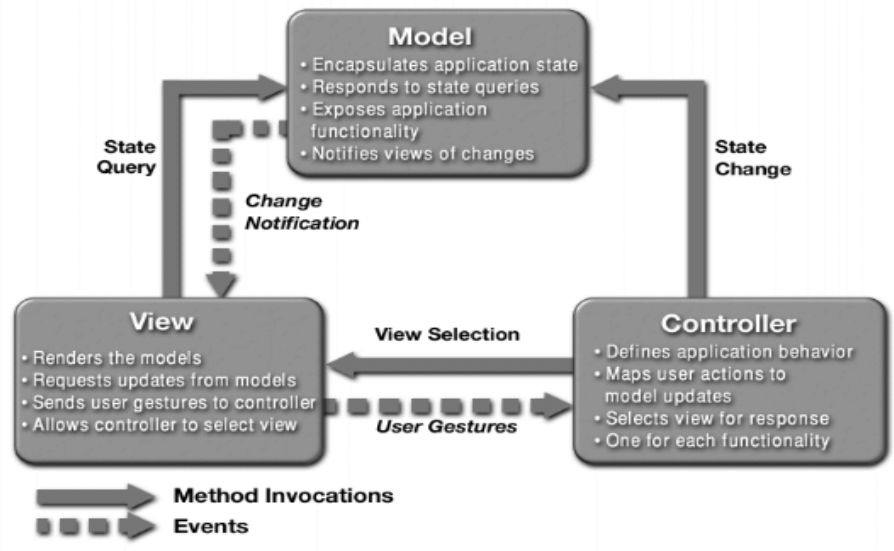
- **MVC proposé :**
  - **Le** Contrôleur est un JPanel,
    - Transforme les actions sur les boutons ou l'entrée d'une opérande en opérations sur le Modèle
  - ou bien Le Modèle est une calculette qui utilise une pile
    - Est un « Observable »
  - **La** Vue est un JPanel,
    - Observateur du Modèle, la vue affiche l'état du Modèle à chaque notification

Proposition : MVC Imbriqués



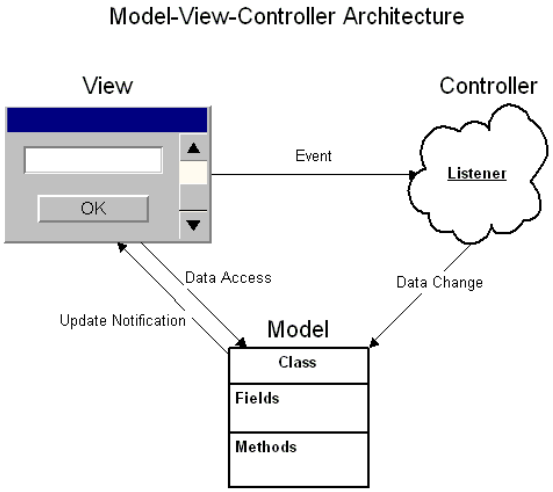
- **Architecture possible**
  - **Le contrôleur inclut la gestion des actions de l'utilisateur**
  - **Niveau 1 : Gestion des « Listeners »**
  - **Niveau 2 : Observable et Observer**

MVC doc de Sun



- **http://java.sun.com/blueprints/patterns/MVC-detailed.html**

IHM et MVC assez répandu …

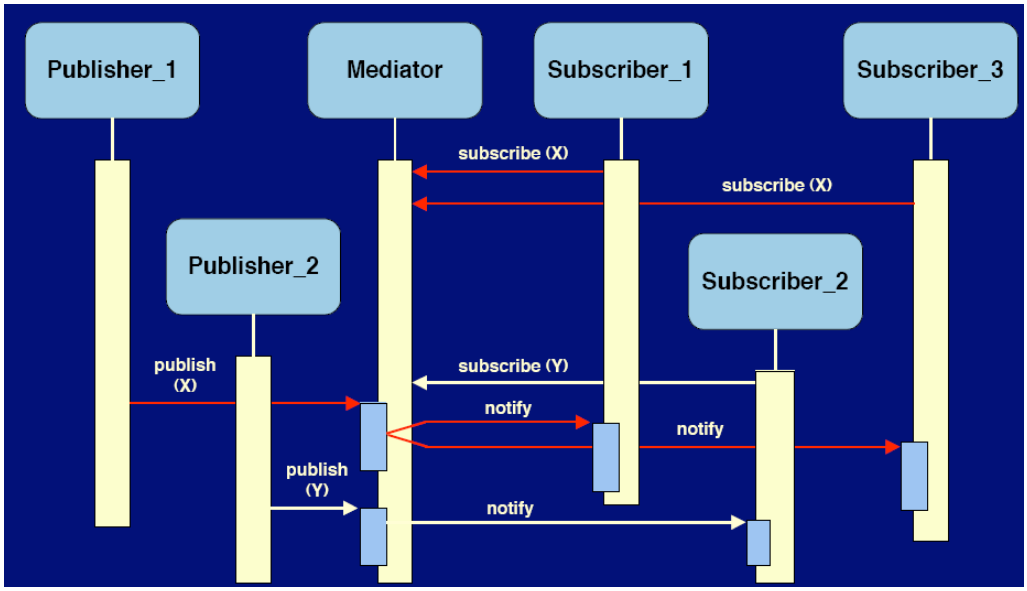


- **Discussion**
  - **Evolution, maintenance, à la recherche du couplage faible**
  - **Exemple**
    peut-on changer d'IHM ?, peut-elle être supprimée ?
    peut-on placer le modèle sur une autre machine ? …

Publish-subscribe

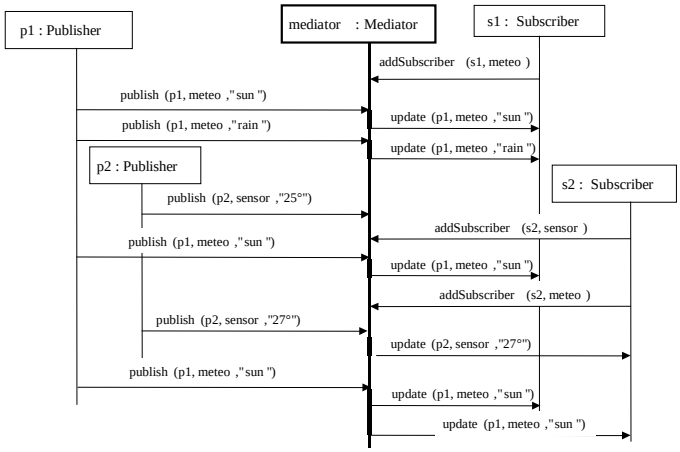
- **Patron Observateur/Observé**
  - Les observateurs sont « réveillés » à chaque notification
  - Filtrage des sources d'évènements ?
- **Patron Publish/Subscribe**
  - Observateurs inscrits auprès d'un médiateur pour un sujet (topic)
  - Les « Observés » notifient au médiateur sur ce sujet

Publish/subscribe



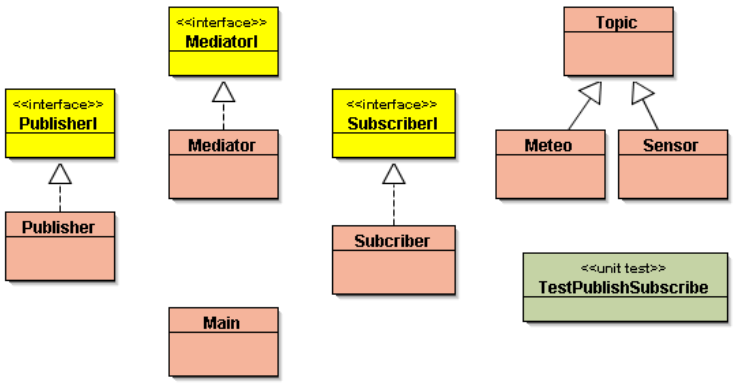
- Extrait de http://www2.lifl.fr/icar/Chapters/Intro/intro.html

Un exemple, le diagramme de séquences



- Une notification d'évènements « météo » et de valeurs d'un « sensor » (*p1,p2 : Publisher*),
- Un médiateur de gestion des abonnements et de notifications (*mediator*),
- Deux souscripteurs aux thèmes « météo » et « sensor » (*s1, s2 : Subscriber*).

Diagramme de classes UML/BlueJ



MediatorI

```
public interface MediatorI{

/** Ajout d'un souscripteur pour ce thème de publication.
 *  Une seule occurrence d'un souscripteur par thème.
 *  @param subscriber le souscripteur.
 *  @param topic le thème de publication.
 */
public void addSubscriber(SubscriberI subscriber, Topic topic);


/** Publication sur ce thème.
 *  Appel de la méthode update de chaque souscripteur c.f. interface
 SubscriberI.
 *  @param source l'émetteur de la publication.
 *  @param topic le thème de publication.
 *  @param arg les arguments liés au thème.
 *  @throws Exception si il n'y a aucun souscripteur pour ce thème.
 */
public void publish(Object source, Topic topic, Object arg) throws Exception;


}
```


SubscriberI, PublisherI

```
public interface SubscriberI{
   /** Notification sur ce thème.
   *  @param source l'émetteur.
   *  @param topic le thème de publication.
   *  @param arg les arguments liés au thème.
   *  @throws Exception si le souscripteur a levé une exception ou est injoignable.
   */

  public void update(Object source, Topic topic, Object arg) throws Exception;
}



public interface PublisherI{
   /** Notification sur ce thème.
   *  @param mediator le mediator concerné.
   *  @param topic le thème de publication.
   *  @param arg les arguments liés au thème.
   *  @throws Exception si il n'y a aucun abonné pour ce thème.
   */
 public void publish(MediatorI mediator, Topic topic, Object arg) throws Exception;
}
```

Topic, les thèmes possibles

```
public class Topic implements Serializable{
  public final static Topic ALL = new Topic(){
                                    public String toString(){
                                      return "TOPIC_ALL";
                                    }
                                  };


    // méthodes toString, equals et hashCode redéfinies
    //……
}
```

les classes **Meteo** et **Sensor**, héritent de **Topic**.
Si un souscripteur souhaite être notifié quelque soit le thème,
il choisira **le thème** la constante **Topic.ALL**.

Main, associé au diagramme de séquences

```
public static void main(String[] args) throws Exception{
    Topic meteo = new Meteo();
    MediatorI   mediator  = new Mediator();
    SubscriberI s1 =  new Subcriber("s1");
    PublisherI  p1 = new Publisher("p1");

    mediator.addSubscriber(s1, meteo);
    mediator.publish(p1,meteo,"sun");
    mediator.publish(p1,meteo,"rain");

    Topic sensor = new Sensor();
    PublisherI p2 = new Publisher("p2");
    SubscriberI s2 =  new Subcriber("s2");
    mediator.addSubscriber(s2, sensor);
    mediator.publish(p1,meteo,"sun");
    mediator.addSubscriber(s2, meteo);
    mediator.publish(p2,sensor,"27°");
    mediator.publish(p1,meteo,"sun");
}
```

Conclusion

- **MVC**
  - **Très utilisé**
  - **Couplage faible obtenu**
  - **Intégration claire du patron Observateur**
  - **Publish/subscribe**
    - **Cf. architectures réparties**

Modèle Vue Contrôleur (MVC) est une méthode de conception pour le développement d'applications logicielles qui sépare le modèle de données, l'interface utilisateur et la logique de contrôle. Cette méthode a été mise au point en 1979 par Trygve Reenskaug, qui travaillait alors sur Smalltalk dans les laboratoires de recherche Xerox PARC[1].

–Extrait de http://fr.wikipedia.org/wiki/MVC