
NFP121, Cnam/Paris
Cours 4
Patrons Observateur/MVC
programmation événementielle

jean-michel Douin, douin au cnam point fr
version : 7 Octobre 2013

Notes de cours

Sommaire

- **Patron Observateur**
- **Programmation évènementielle**
- **Patron MVC Modèle Vue Contrôleur**

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 Responsibility. 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**

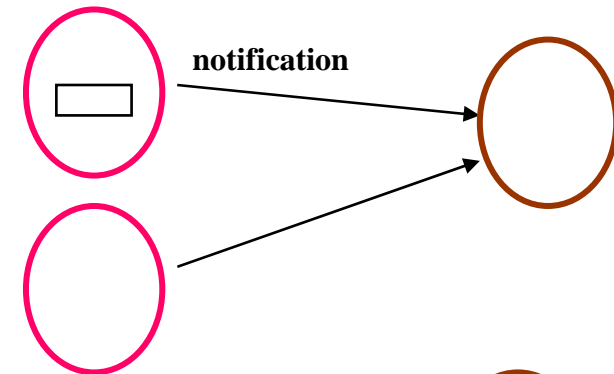
Patron Observé/observateur 1)

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

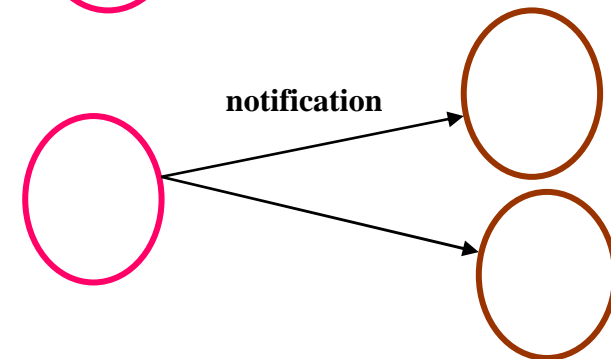
- **Un Observé**
 - N'importe quelle instance qui est modifiée
 - i.e. un changement d'état comme la modification d'une donnée d'instance
usage d'un mutateur, une opération...
- **Des Observateurs, seront notifiés**
 - A la modification de l'observé,
 - Synchrones, (sur la même machine virtuelle)

Patron Observé/observateur 2)

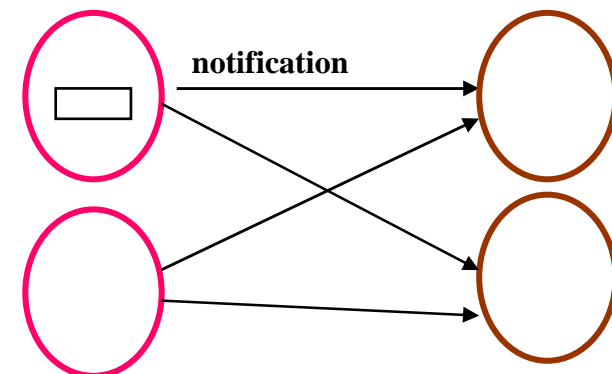
- **Plusieurs Observés / un observateur**



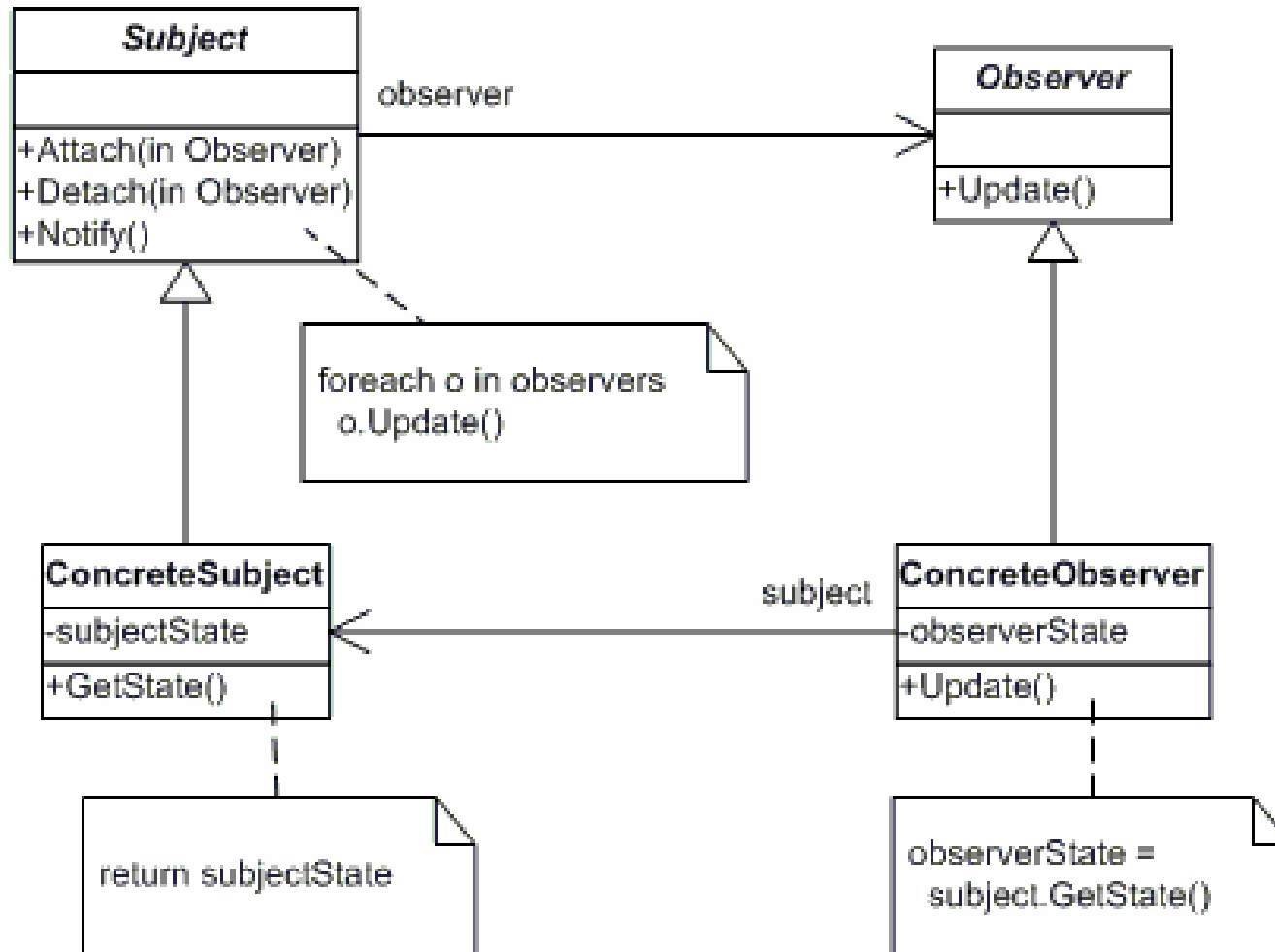
- **Un observé / plusieurs observateurs**



- **Plusieurs observés / plusieurs observateurs**

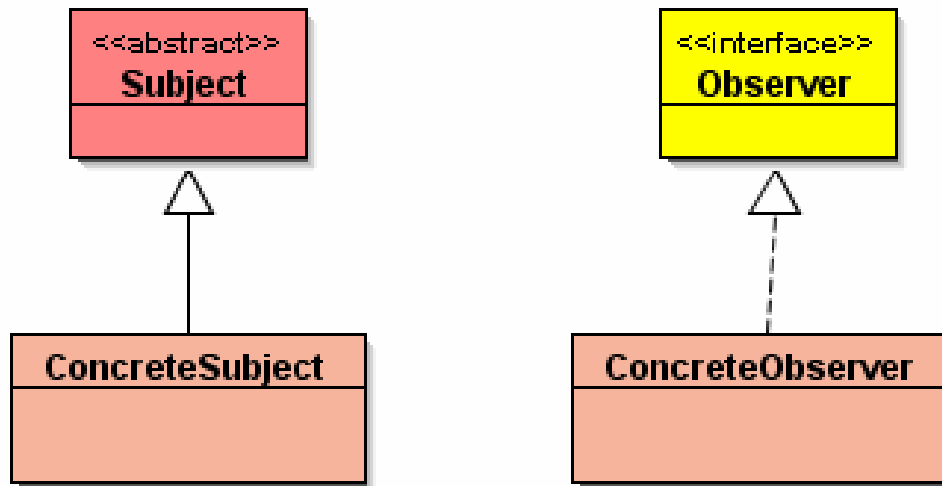


UML & le patron Observateur, l'original



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

Démonstration

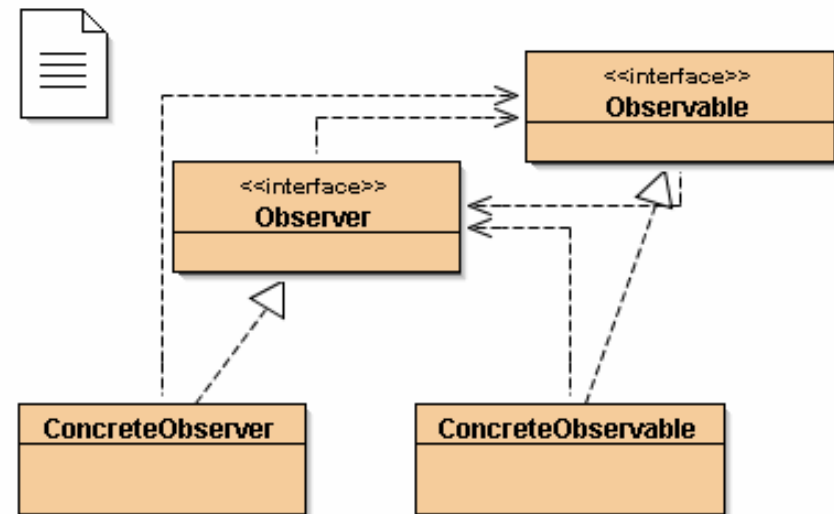


Une réalisation : Observateur en Java

- Lors d'un changement d'état, notification aux observateurs inscrits

```
public interface Observable{  
    public void addObserver(Observer o);  
    public void removeObserver(Observer o);  
    public void notifyObservers();  
  
    public int getState();  
    public void setState(int state);  
}
```

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



ConcreteObservable

```
public class ConcreteObservable implements Observable{
    private Collection<Observer> observers = new .....
    private int state = 0;
    public void addObserver(Observer observer){
        observers.add(observer);
    }
    public void removeObserver(Observer observer){
        observers.remove(observer);
    }

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

    public void setState(int state){
        this.state = state;
        notifyObservers();
    }

    public int getState(){return this.state;}
}
```

Observer : mise en oeuvre

```
Observable o = new ConcreteObservable();
```

```
Observer obs1= new ConcreteObserver();
```

```
o.addObserver(obs1);
```

```
o.setState(3); // obs1 est réveillé, notifié
```

```
Observer obs2= new ConcreteObserver();
```

```
o.addObserver(obs2);
```

```
o.setState(33); // obs1 et obs2 sont réveillés, notifiés ...
```

Démonstration / discussion

- **Variante syntaxique, cf. démonstration**
 - **Observé est une classe abstraite**
 - **Observateur est une interface**

 - **Une liste est un « Observé » concret**
 - **Un Observateur de Liste implémente l'interface Observateur**

Notification ... amélioration

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

Nous pourrions ajouter un paramètre *(la cause de la notification par exemple)*

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

Paquetage java.util !

Classes existantes : java.util.Observable, java.util.Observer

Observé/Observateurs

java.util

Class Observable

[java.lang.Object](#)

└─ `java.util.Observable`

```
public class Observable  
extends Object
```

This class represents an observable object, or "data" in the model-view paradigm.

java.util

Interface Observer

```
public interface Observer
```

A class can implement the `Observer` interface when it wants to be informed of changes in observable objects.

- **Prédéfinies *model-view paradigm* ...**

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

Observateur comme « Listener »

- **java.awt.event.EventListener**
 - Les écouteurs/observateurs sont tous des « EventListener »
- **Convention syntaxique de Sun pour ses API**
 - **XXXXX**Listener extends EventListener
« *Observer* »
 - **addXXXXX**Listener
« *addObserver* »

exemple l'interface **Action**Listener / **addAction**Listener
- **EventObject** Source de la notification,
 - **XXXXX**Event extends EventObject

exemple **Action**Event

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](#), [MouseListener](#), [MetaEventListener](#), [MouseInputListener](#), [MouseMotionListener](#), [MouseWheelListener](#), [NamespaceChangeListener](#), [NamingListener](#), [NodeChangeListener](#), [NotificationListener](#), [ObjectChangeListener](#), [PopupMenuListener](#), [PreferenceChangeListener](#), [PropertyChangeListener](#), [RowSetListener](#), [RowSorterListener](#), [SSLSessionBindingListener](#), [StatementEventListener](#), [TableColumnModelListener](#), [TableModelListener](#), [TextListener](#), [TreeExpansionListener](#), [TableModelListener](#), [TreeSelectionListener](#), [TreeWillExpandListener](#), [UndoableEditListener](#), [UnsolicitedNotificationListener](#), [VetoableChangeListener](#), [WindowFocusListener](#), [WindowListener](#), [WindowStateListener](#)

- **Une grande famille !**

update(Observable comme EventObject ...

java.util

Class EventObject

java.lang.Object

java.util.EventObject

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AWTEvent: BeanContextEvent, CaretEvent, ChangeEvent, ConnectionEvent, DragGestureEvent, DragSourceEvent, DropTargetEvent, FlavorEvent, HandshakeCompletedEvent, HyperlinkEvent, LineEvent, ListDataEvent, ListSelectionEvent, MenuEvent, NamingEvent, NamingExceptionEvent, NodeChangeEvent, Notification, PopupMenuEvent, PreferenceChangeEvent, PrintEvent, PropertyChangeEvent, RowSetEvent, RowSorterEvent, SSLSessionBindingEvent, StatementEvent, TableColumnModelEvent, TableModelEvent, TreeExpansionEvent, TreeModelEvent, TreeSelectionEvent, UndoableEditEvent, UnsolicitedNotificationEvent

java.awt

Class AWTEvent

java.lang.Object

java.util.EventObject

java.awt.AWTEvent

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

ActionEvent: AdjustmentEvent, AncestorEvent, ComponentEvent, HierarchyEvent, InputMethodEvent, InternalFrameEvent, InvocationEvent, ItemEvent, TextEvent

```
package java.util;
```

```
public class EventObject implements ...{
```

```
    public EventObject(Object source){ ...}
```

```
    public Object getSource(){ ...}
```

```
    public String toString(){ ...}
```

```
}
```

```
ActionEvent extends AWTEvent extends EventObject
```


Exemple: 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`
notifie
ses observateurs `java.awt.event.ActionListener` ...

```
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 ActionListener

```
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
```

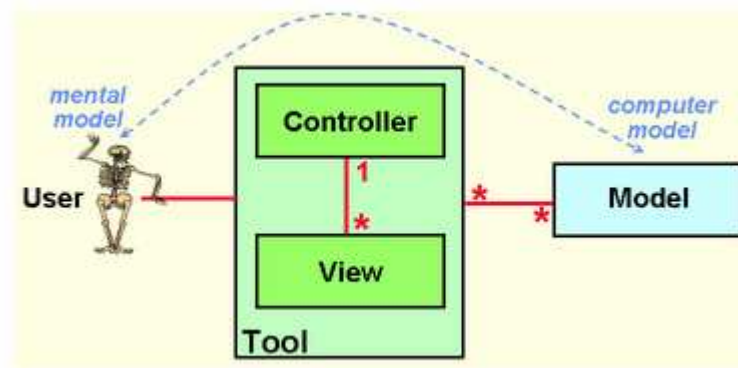
Démonstration / Discussion

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`**
- **Listener comme Observateur**
- **La grande famille des « `EventListener` » / « `EventObject` »**

Modèle Vue Contrôleur

MVC *XEROX PARC 1978-79*



- <http://heim.ifi.uio.no/trygver/themes/mvc/mvc-index.html>

Patrons Observer / MVC

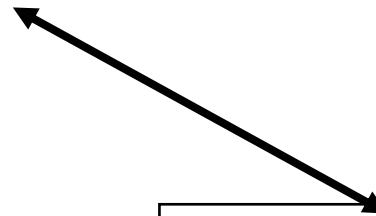
Observés / Observateurs



• **Modèle**

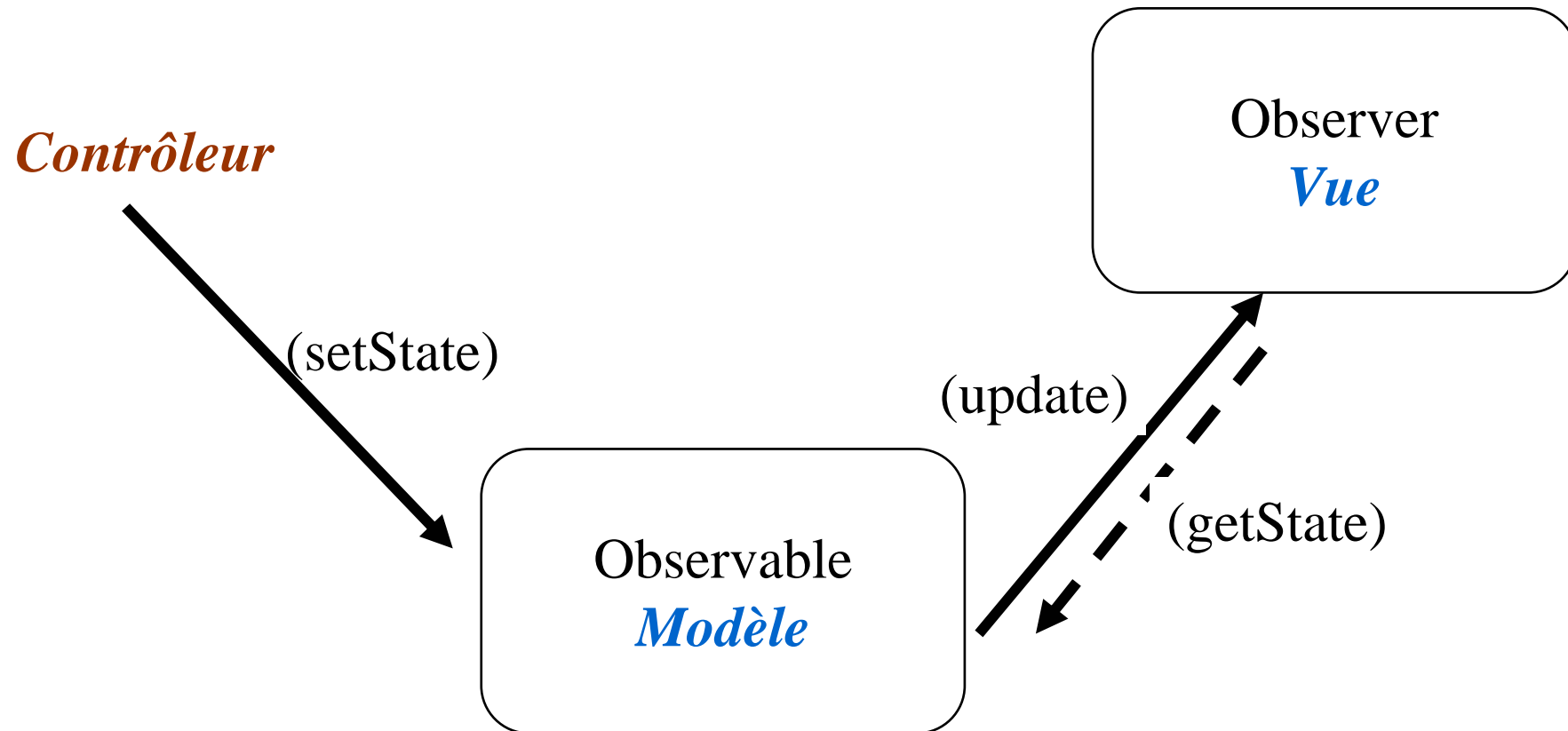
Vue

Contrôleur

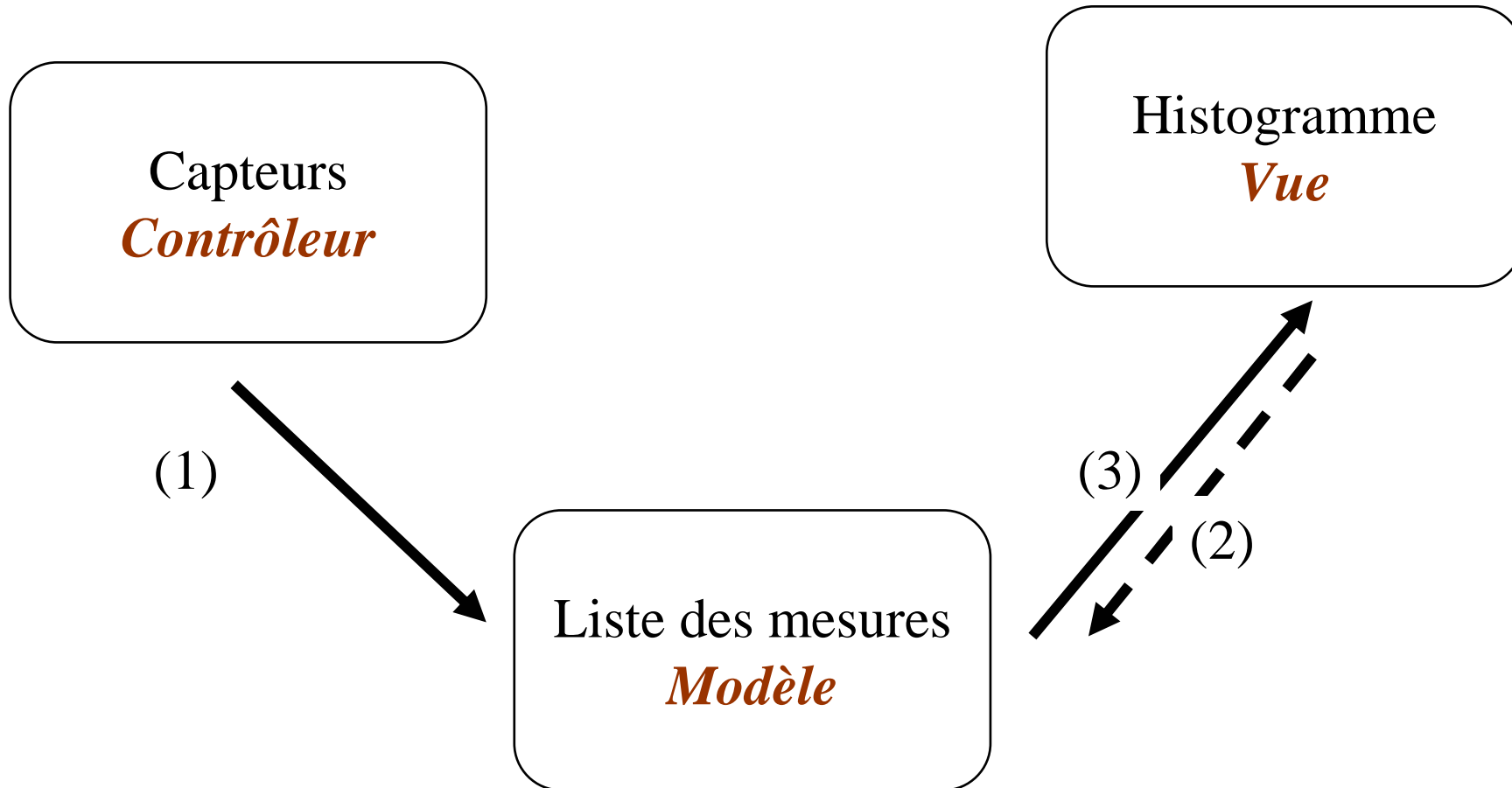


agit, modifie le **Modèle**

MVC : Observer est inclus

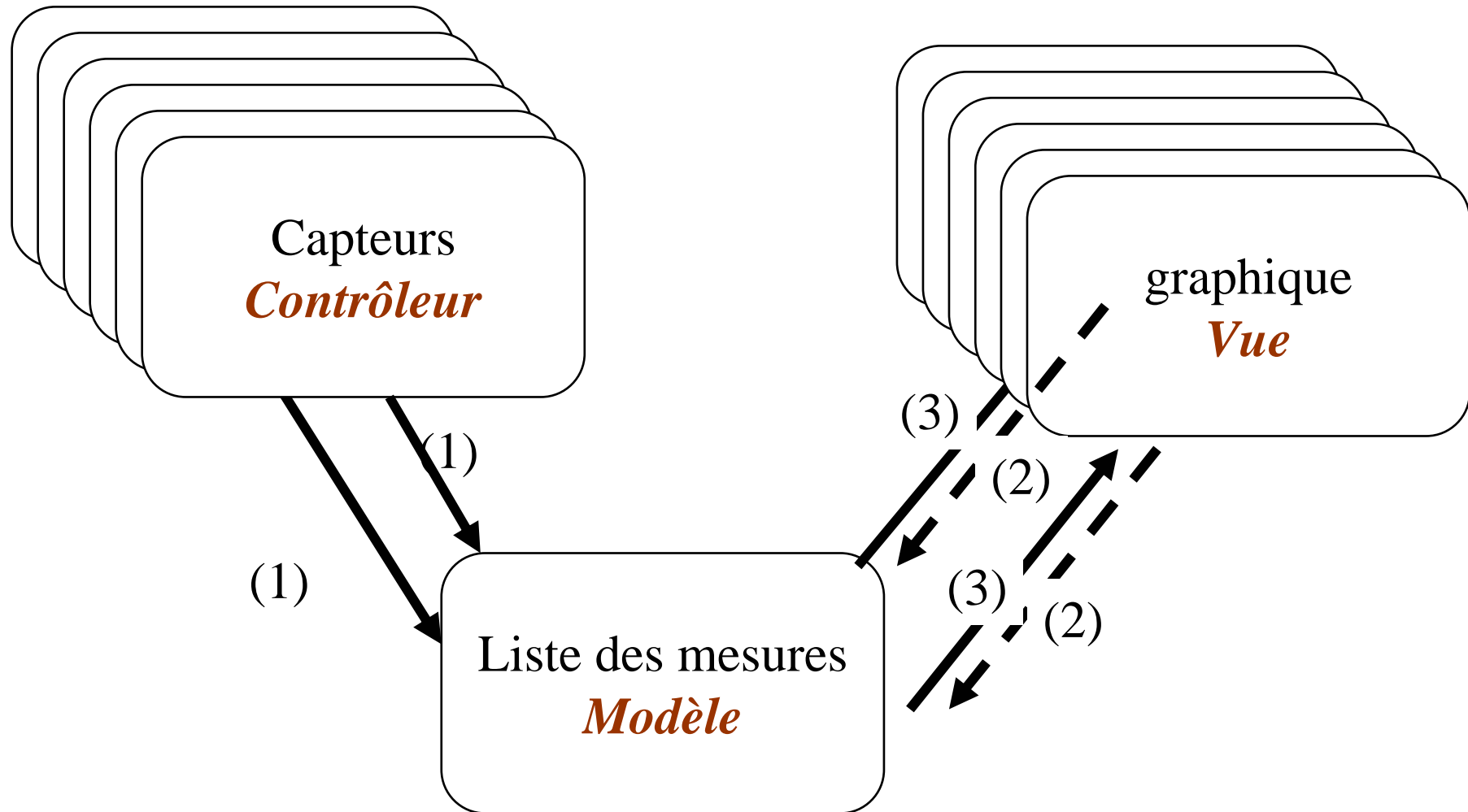


MVC : exemple de capteurs ...



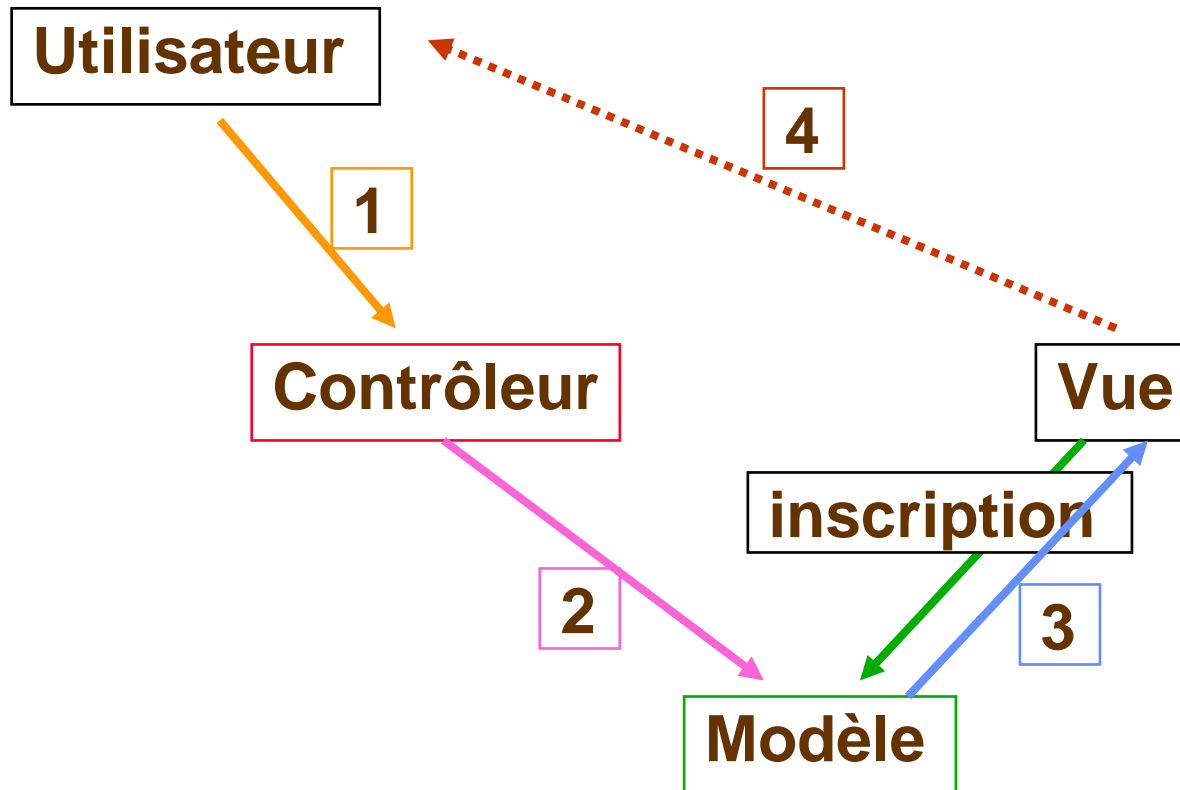
- Une architecture simple et souple ...

MVC Avantages

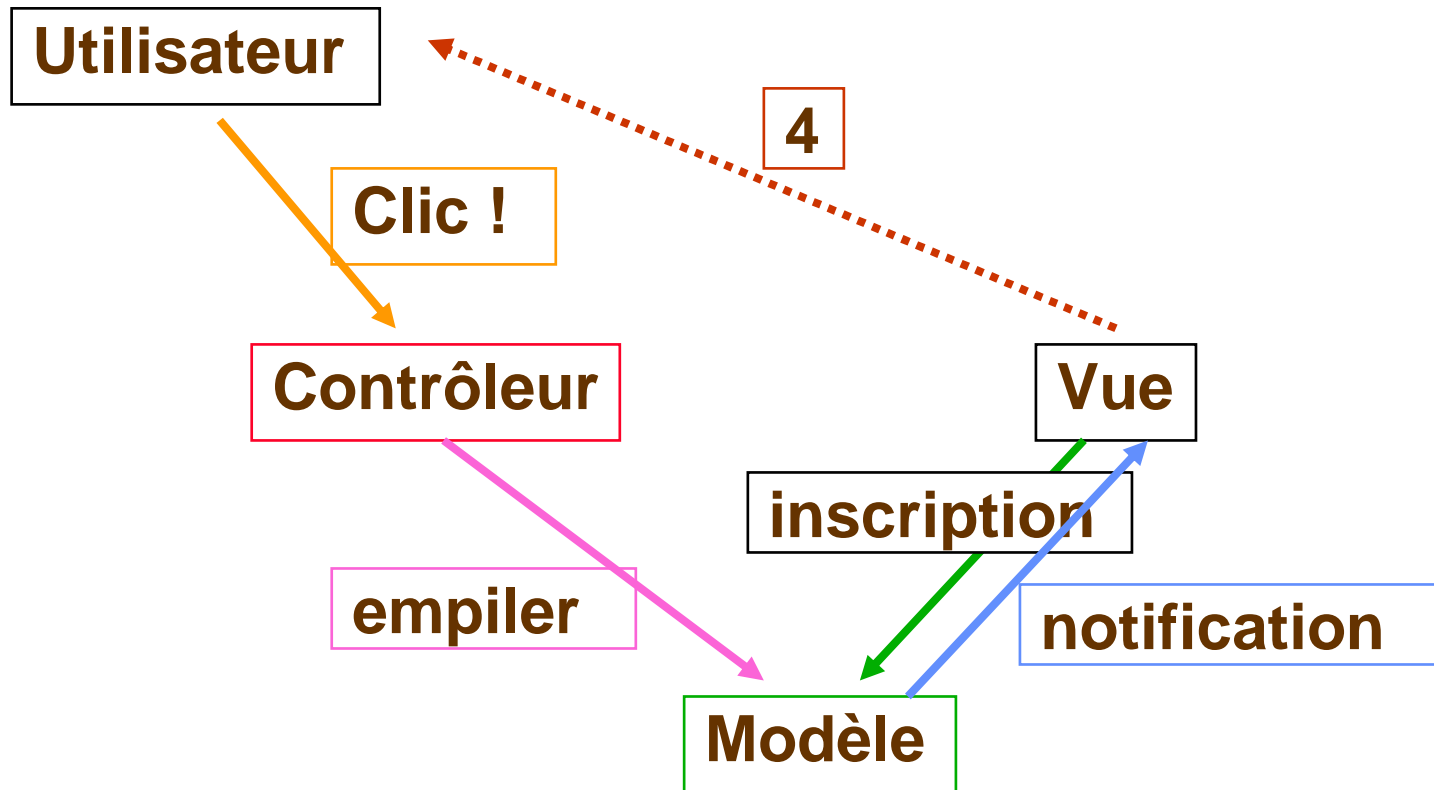


- **Souple ?**

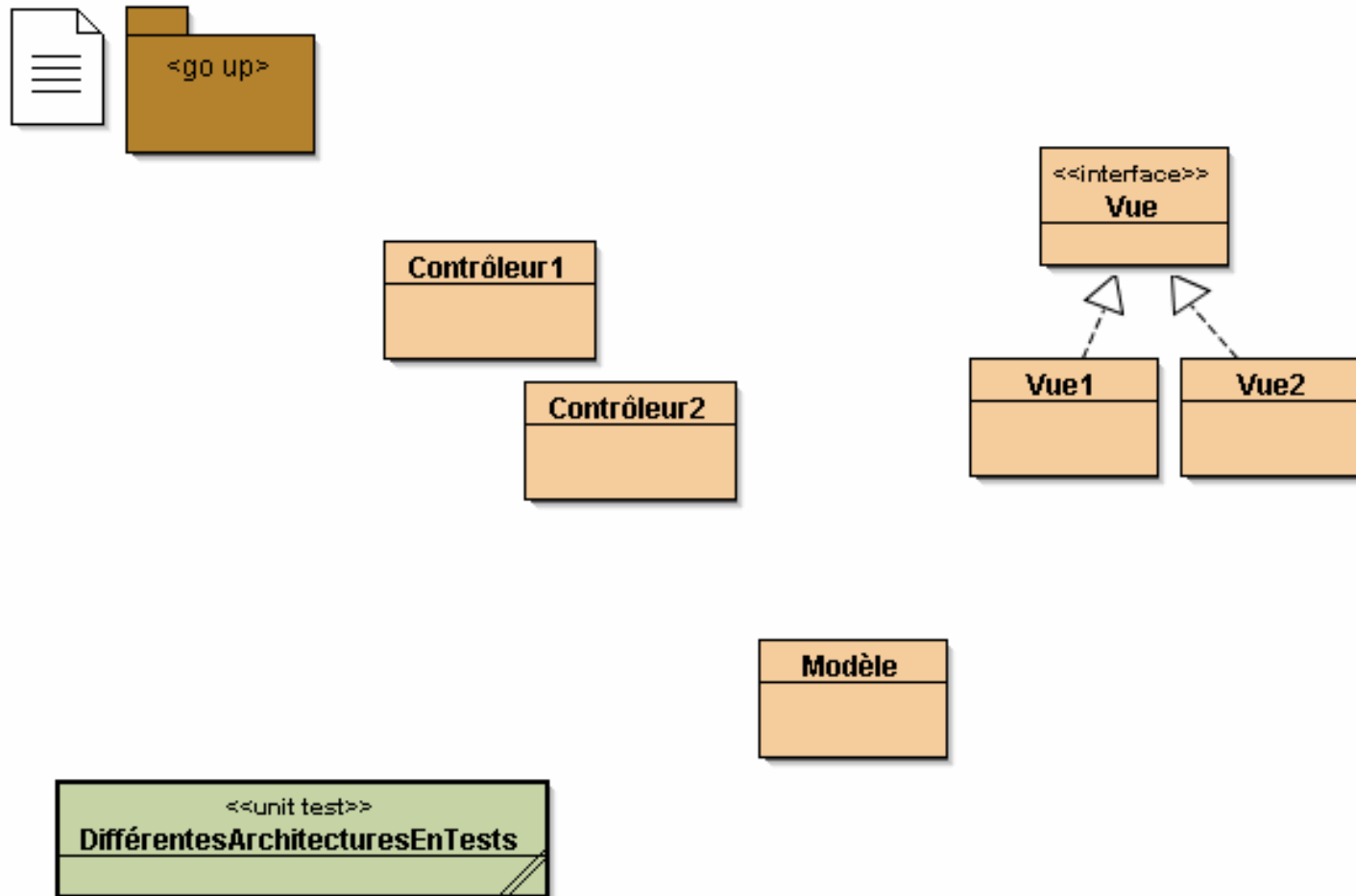
Un cycle MVC



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;

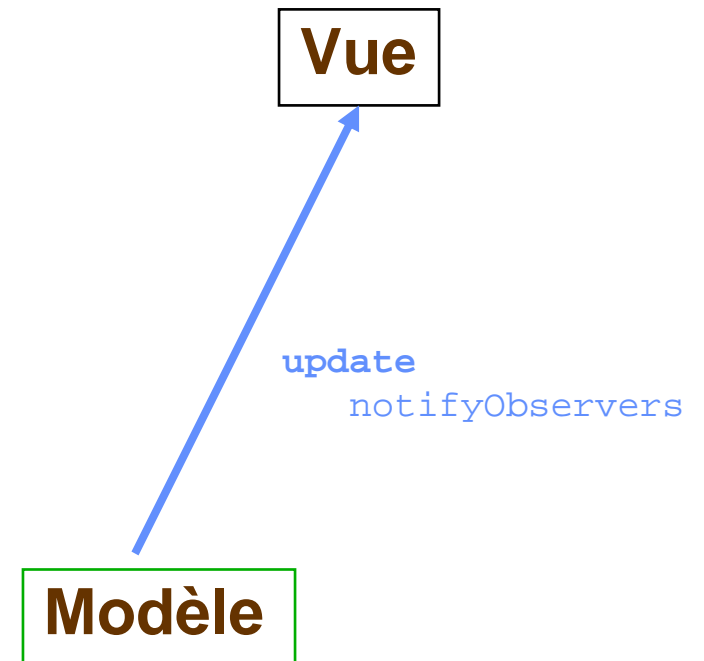
public class Modèle extends Observable{

    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();
}

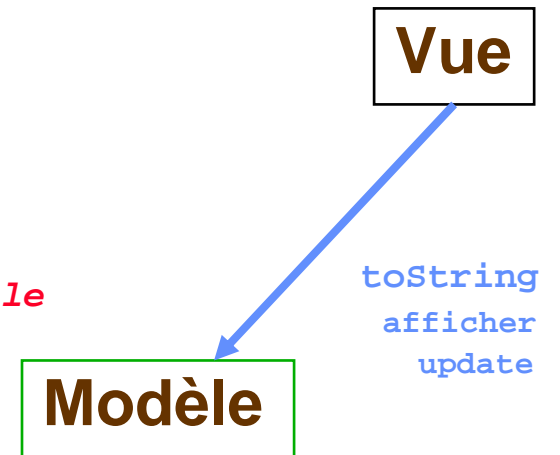
import java.util.Observable;
import java.util.Observer;

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);
    }
}
```

Contrôleur

setEntier
incrémenter

Modèle

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
```

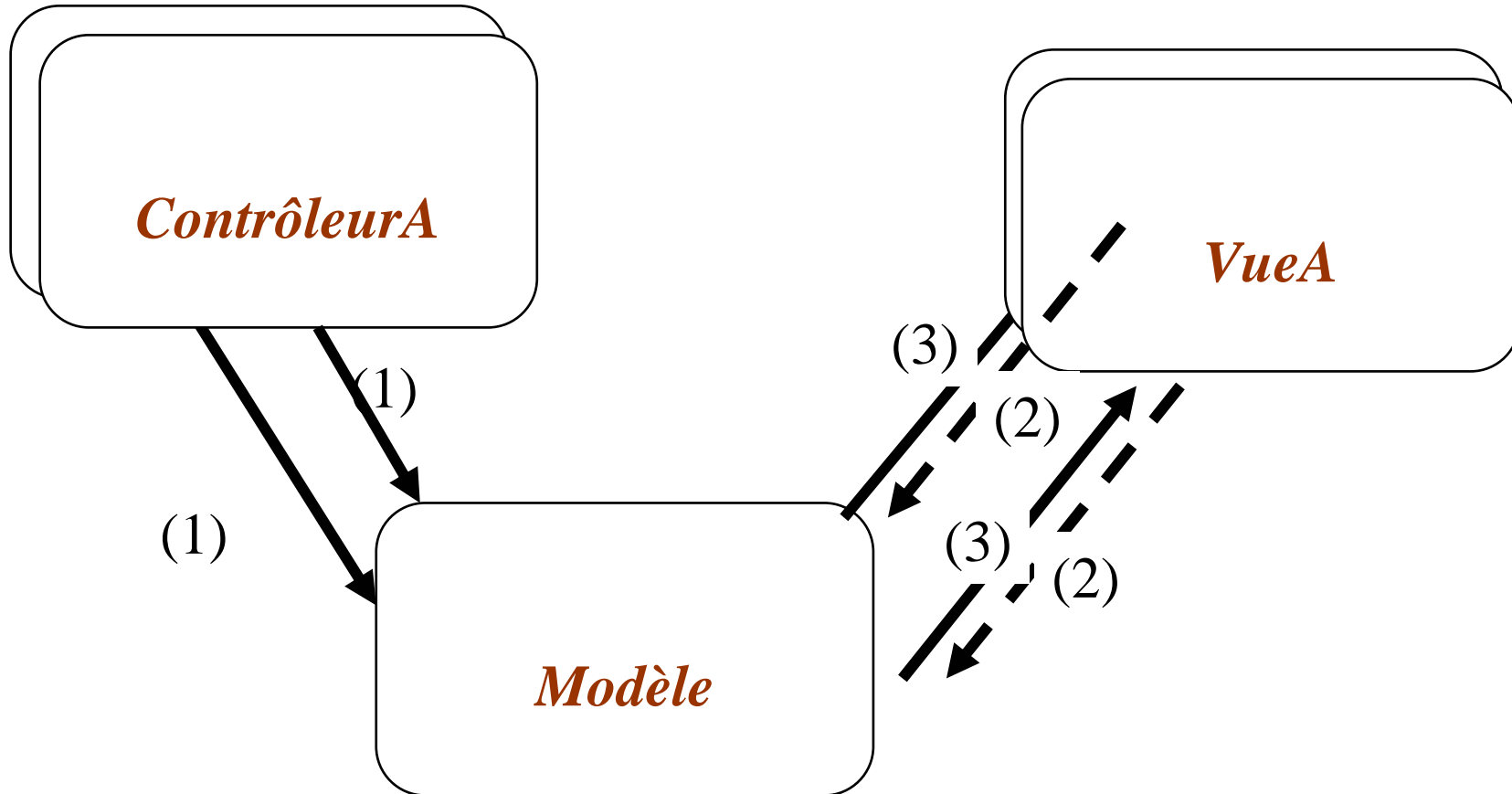
```
Contrôleur1 contrôleurA = new Contrôleur1(modèle);
```

```
Contrôleur1 contrôleurB = new Contrôleur1(modèle);
```

```
contrôleurA.incrémenter();
```

```
contrôleurB.incrémenter();
```

Discussion



AWT / Button, discussion

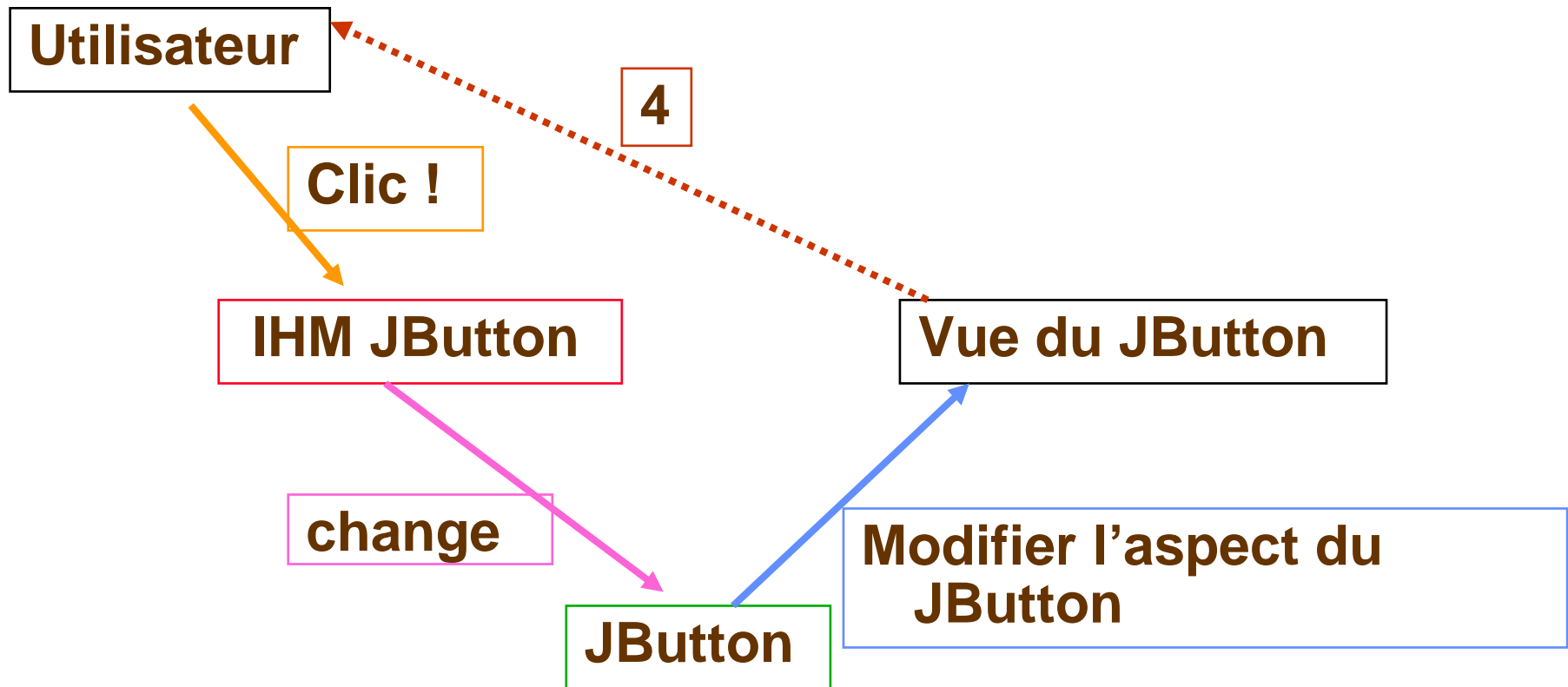
- **Un « Button » (le contrôleur) contient un MVC**
À part entière



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

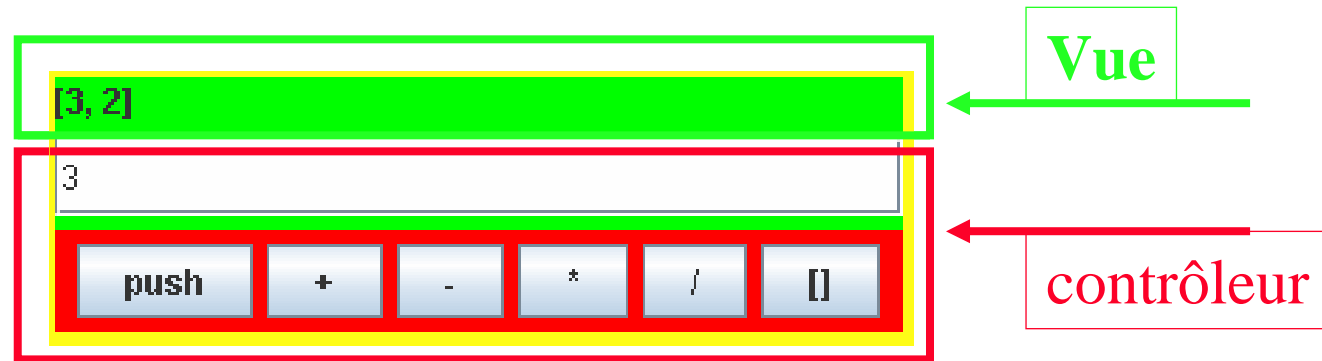
- **Une IHM (JApplet,...) contient la Vue et le Contrôle**
 - Alors le compromis architecture/lisibilité est à rechercher

Un JButton comme MVC



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

Proposition c.f. le TP



- **MVC proposé :**

- Le **Contrôleur** est un JPanel,

- Transforme les actions sur les boutons ou l'entrée d'une opérande en opérations sur le Modèle

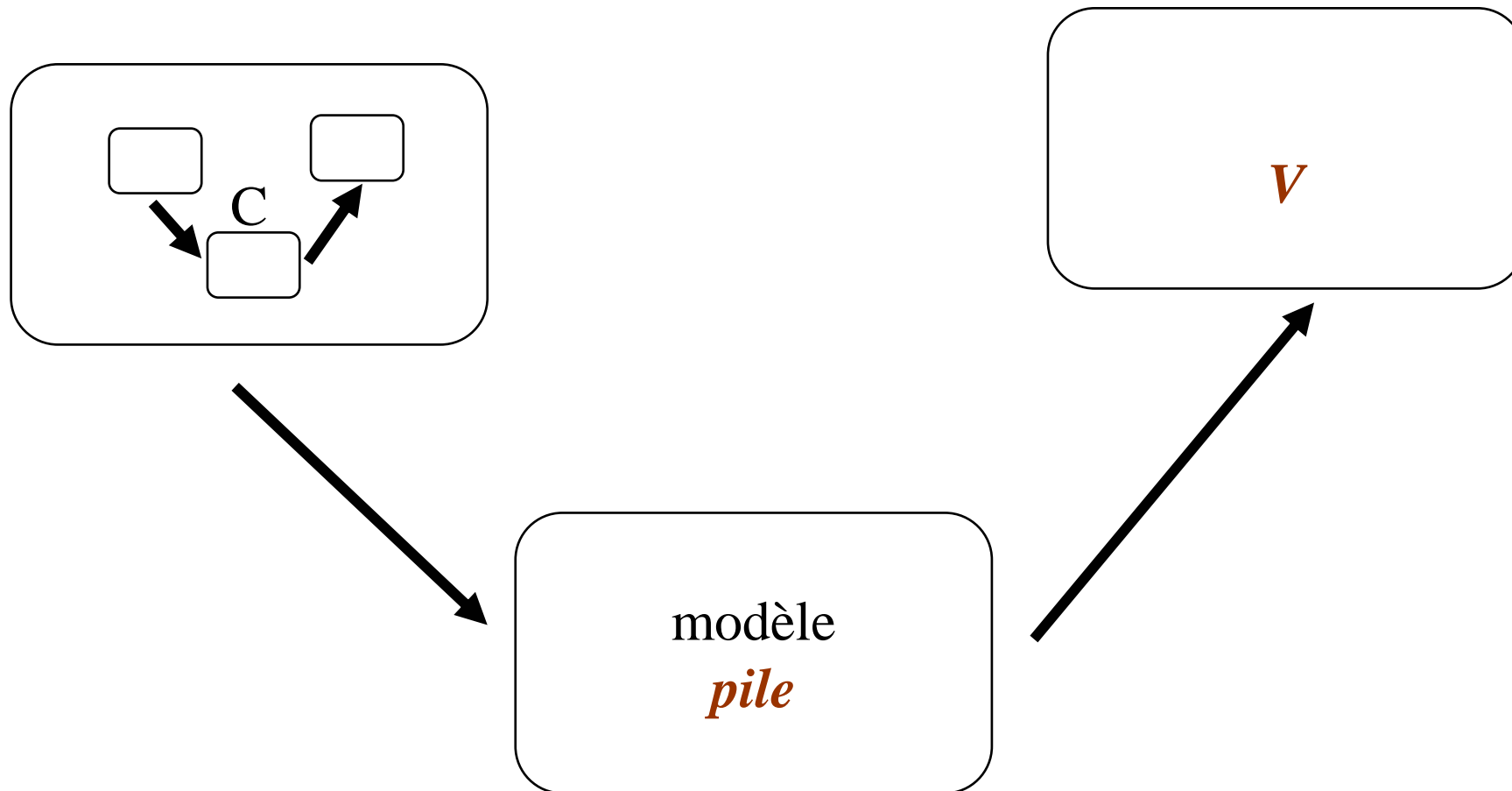
- ou bien Le **Modèle** est une calculette qui utilise une pile

- Est un « Observable »

- La **Vue** est un JPanel,

- Observateur du Modèle, la vue affiche l'état du Modèle à chaque notification

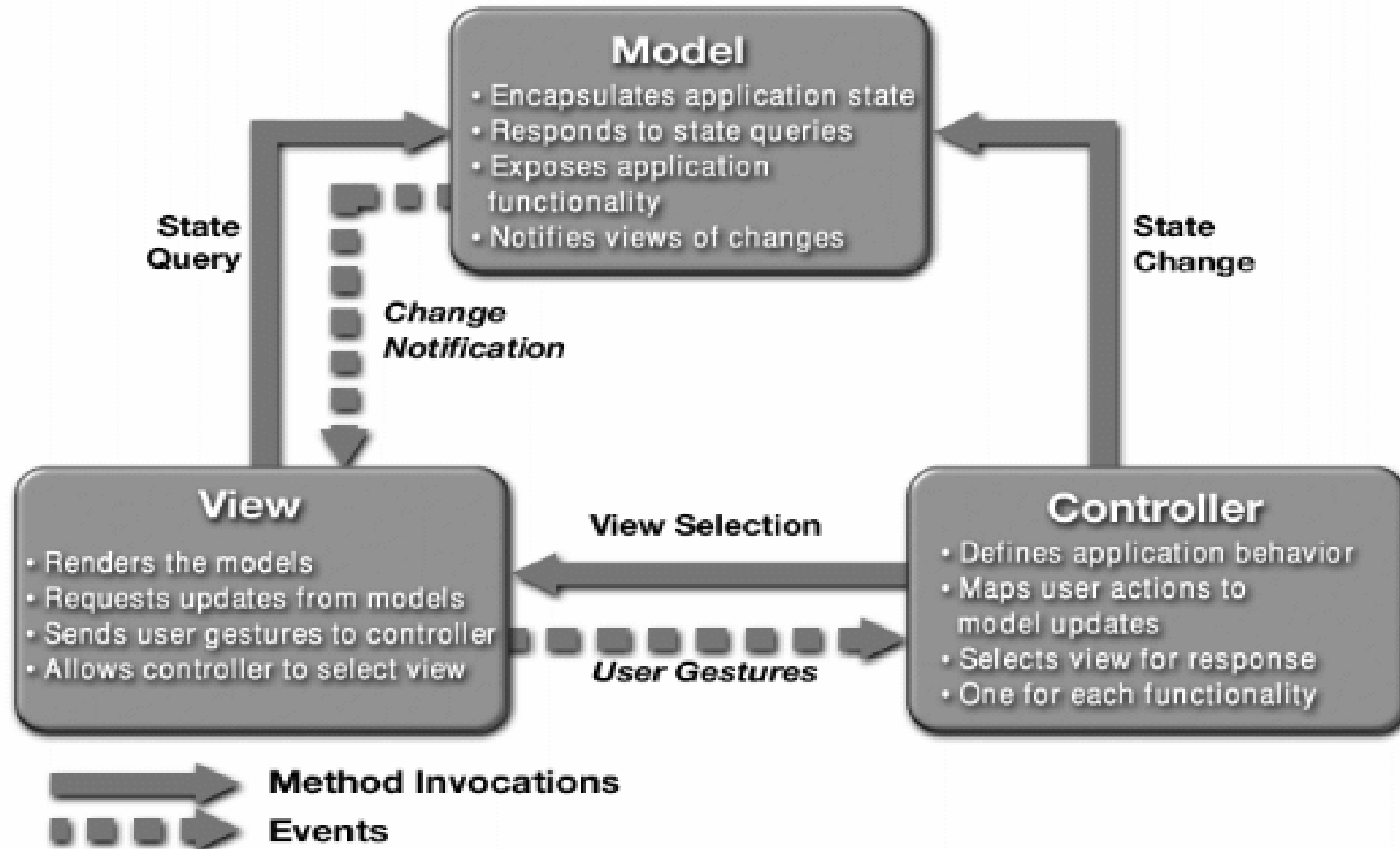
Proposition : MVC Imbriqués



- **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>

Nouvelle Proposition

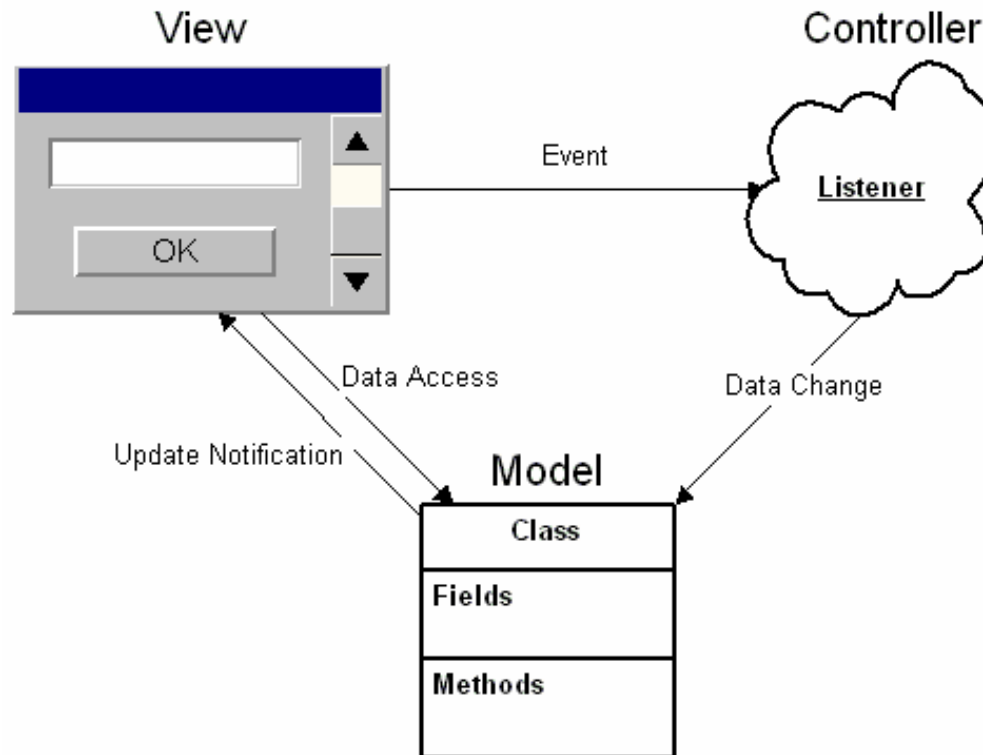


- **MVC proposé :**

- Le **Contrôleur** implémente tous les listeners
 - Transforme les actions sur les boutons ou l'entrée d'une opérande en opérations sur le Modèle
- Le **Modèle** est une calculette (*qui utilise une pile en interne*)
 - Est un « Observable »
- La **Vue** est un JPanel, une applette (*ce que l'on voit*)
 - Observateur du Modèle, la vue affiche l'état du Modèle à chaque notification

IHM et MVC assez répandu ...

Model-View-Controller Architecture



- **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 ? ...

Conclusion

- **MVC**
 - Très utilisé

 - Couplage faible obtenu

 - Intégration claire du patron Observateur

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>