POO / IHM
Architectures pour systèmes interactifs

Architecture Logicielle : Pourquoi ?

Organiser le code (rangement)
Organiser le travail
Ré-utiliser

Simplifier (diviser régner)
Modifier (une partie)

Notions : modularité, évolutivité, flexibilité

Séparation possible
– Code pour IHM
– Code «Métier»

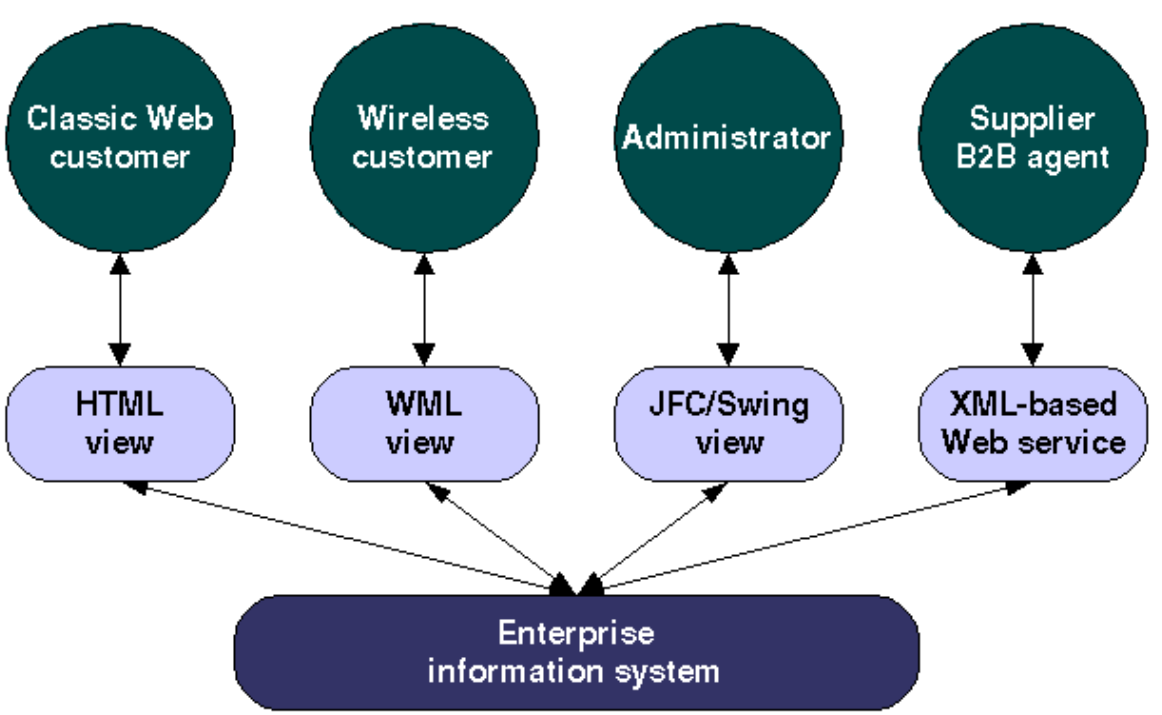
• *Exemple*
– IHM différente pour une Gestion d'un stock de chaussures ou de bibelots ?
– Linux sous gnome ou kde, le système change-t-il ?

• Objectif : éviter de tout modifier si on change la partie fonctionnelle ou la partie IHM

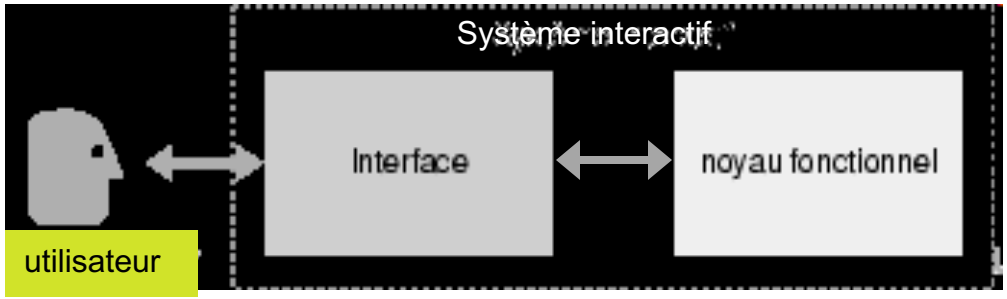
Un problème classique

INTERACTION

FONCTIONNALITES



Architectures et Systèmes Interactifs



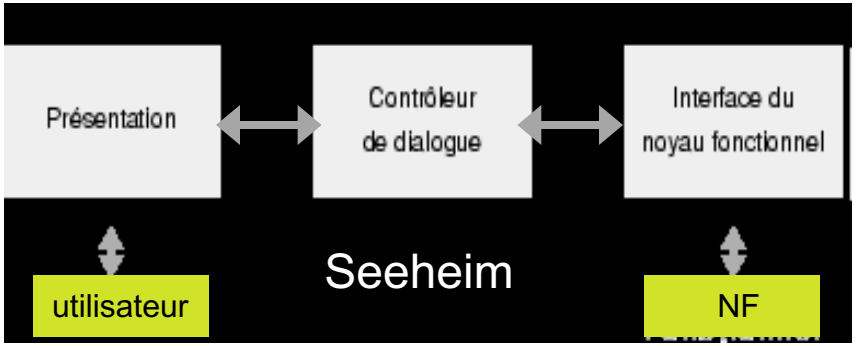
Tous les modèles partent du **principe :**
un **système interactif** comporte une partie **interface** et une partie **application pure**
Cette dernière est souvent appelée *noyau fonctionnel*.

La plupart des modèles identifient au moins trois types d'éléments :

un ``coté utilisateur'' (*présentations, vues*),
un ``côté noyau fonctionnel'' (*interfaces du noyau fonctionnel, abstractions, modèles*),
et des éléments articulatoires (*contrôleurs, adaptateurs*).

http://tel.archives-ouvertes.fr/docs/00/04/82/79/HTML/2_2modelesinterface_referen.html

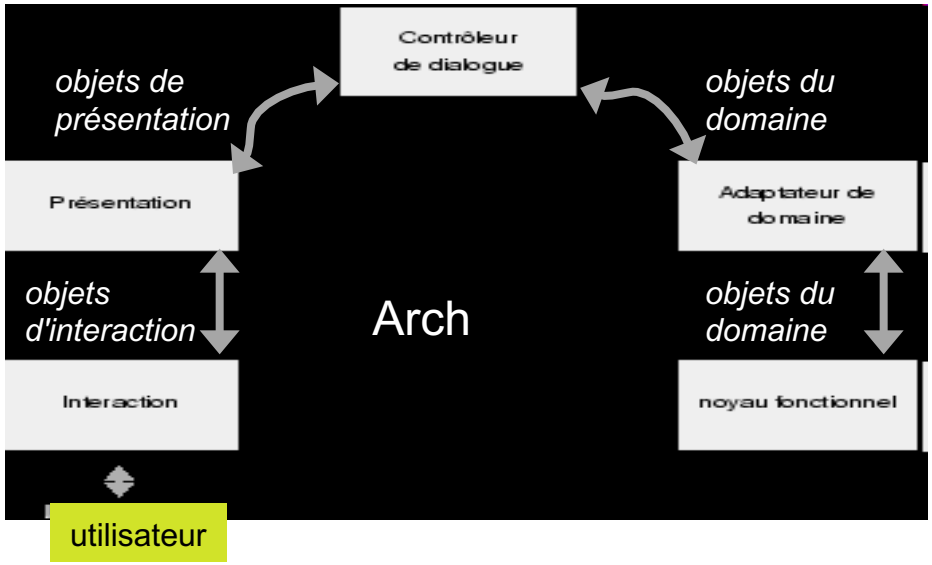
Modèles à couches : Seeheim



Premier modèle (groupe de travail à Seeheim en 1985) - destiné au traitement lexical des entrées et sorties dans les interfaces textuelles – a servi de base à beaucoup d'autres modèles.

La présentation est la couche en contact direct avec les entrées et sorties (interprétation des actions utilisateur + génération des sorties au niveau lexical)
Le contrôleur de dialogue gère le séquencement de l'interaction
L'interface du noyau fonctionnel convertit les entrées en appels du noyau fonctionnel et les données abstraites de l'application en des éléments présentables à l'utilisateur

Modèles à couches : Arch



Le modèle **Arch** [1992]
5 composants et 3 types de données

Composant d'interaction - ensemble des widgets + communication avec les périphériques
Composant de présentation - représentation logique des widgets indépendante de la boite à outils
Contrôleur de dialogue - responsable du séquencement des tâches et du maintien de la consistance entre les vues multiples.
Adaptateur du domaine - responsable des tâches dépendantes du domaine qui ne font pas partie du noyau fonctionnel mais qui sont nécessaires à sa manipulation par l'utilisateur.
Noyau fonctionnel *représente la partie non interactive de l'application.*

Le patron MVC

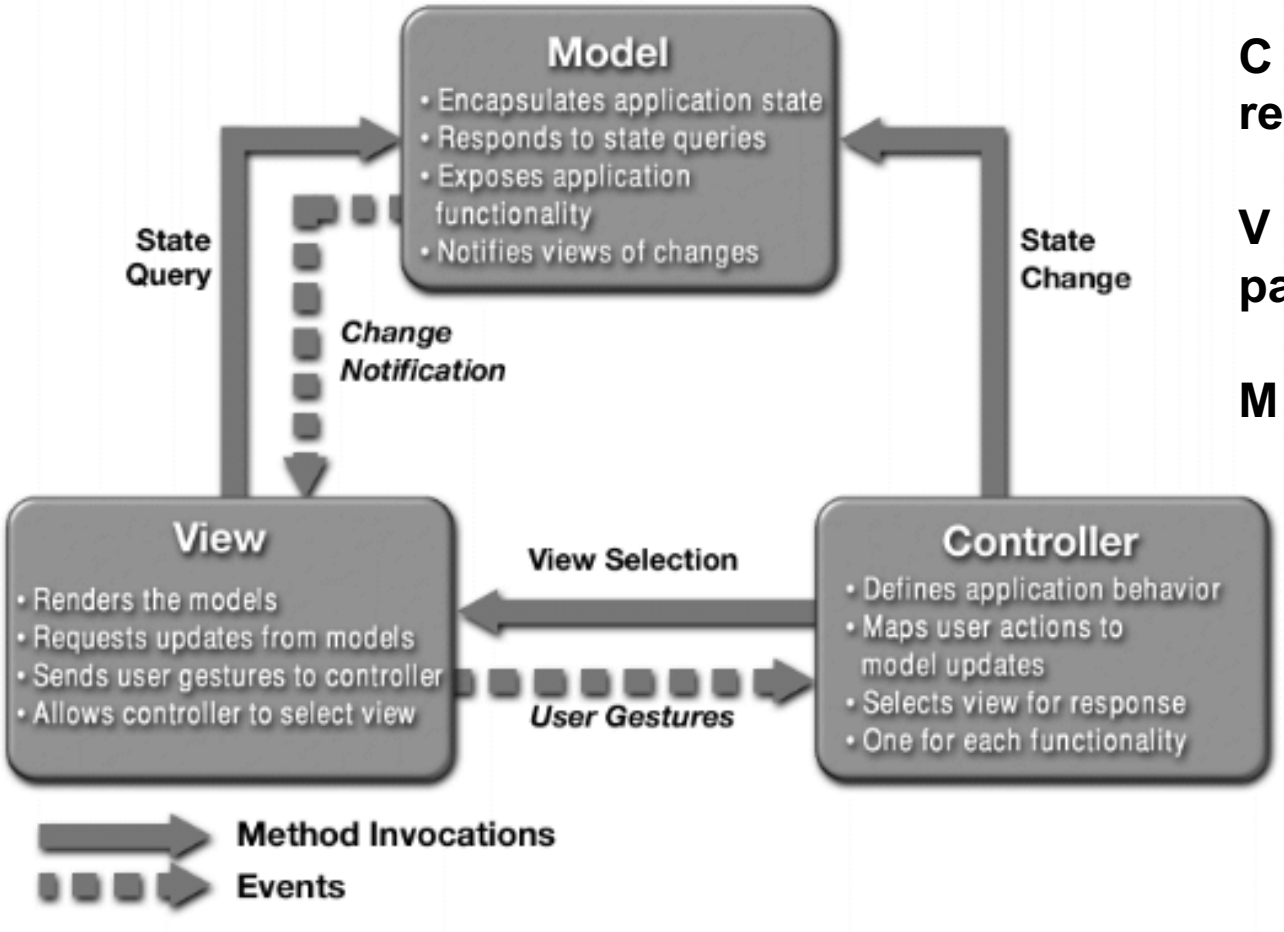
- Impose une séparation en 3 couches :
  - MODELE / NF /OBJET METIER : représente les données de l'application. Définit interaction avec la base de données et le traitement des données
  - VUE : représente l'interface utilisateur. Effectue aucun traitement, ne fait que l'affichage des données (fournit par M). Possibilité d'avoir plusieurs vues pour un même M
  - CONTROLEUR : gère l'interface entre le modèle et le client. Interprète la requête de ce dernier pour lui envoyer la vue correspondante. Effectue la synchronisation entre le modèle et les vues

Entités et fonctionnalités : ATTENTION AU MODELE

- Pour certains projets, il pourrait s'agir d'un système de base de données / fichiers, un ensemble d'entités, et un certain nombre de classes / bibliothèques qui fournissent une logique supplémentaire pour les entités (telles que l'exécution de calculs, gestion de l'état, etc)
- Implémentation : Créer des classes qui décrivent votre domaine et la fonctionnalité . Vous devriez vous retrouver avec un ensemble d'objets et un ensemble de classes qui manipulent ces objets.

Zoom sur l'architecture MVC

- **Smalltalk[Goldberg et Robson1979-1983]** : séparation des classes en trois paquets
  - **Model** : données et comportement applicatif
  - **View:** représentation manipulable des données (structure IHM)
  - **Control** : interprétation des entrées utilisateur (comportement IHM)



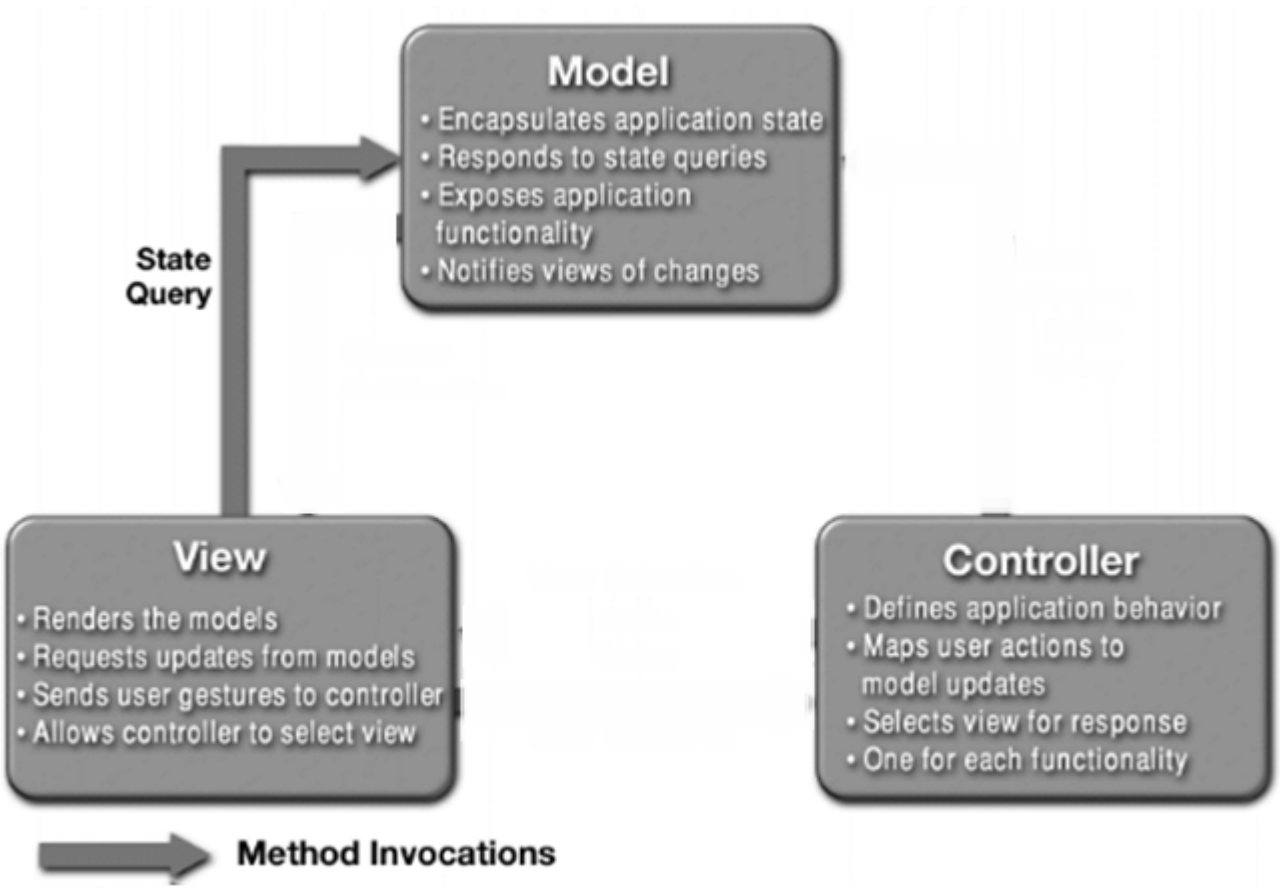
C connait M et V sans restriction

V peut connaitre M partiellement (abstraction)

M connait V partiellement (

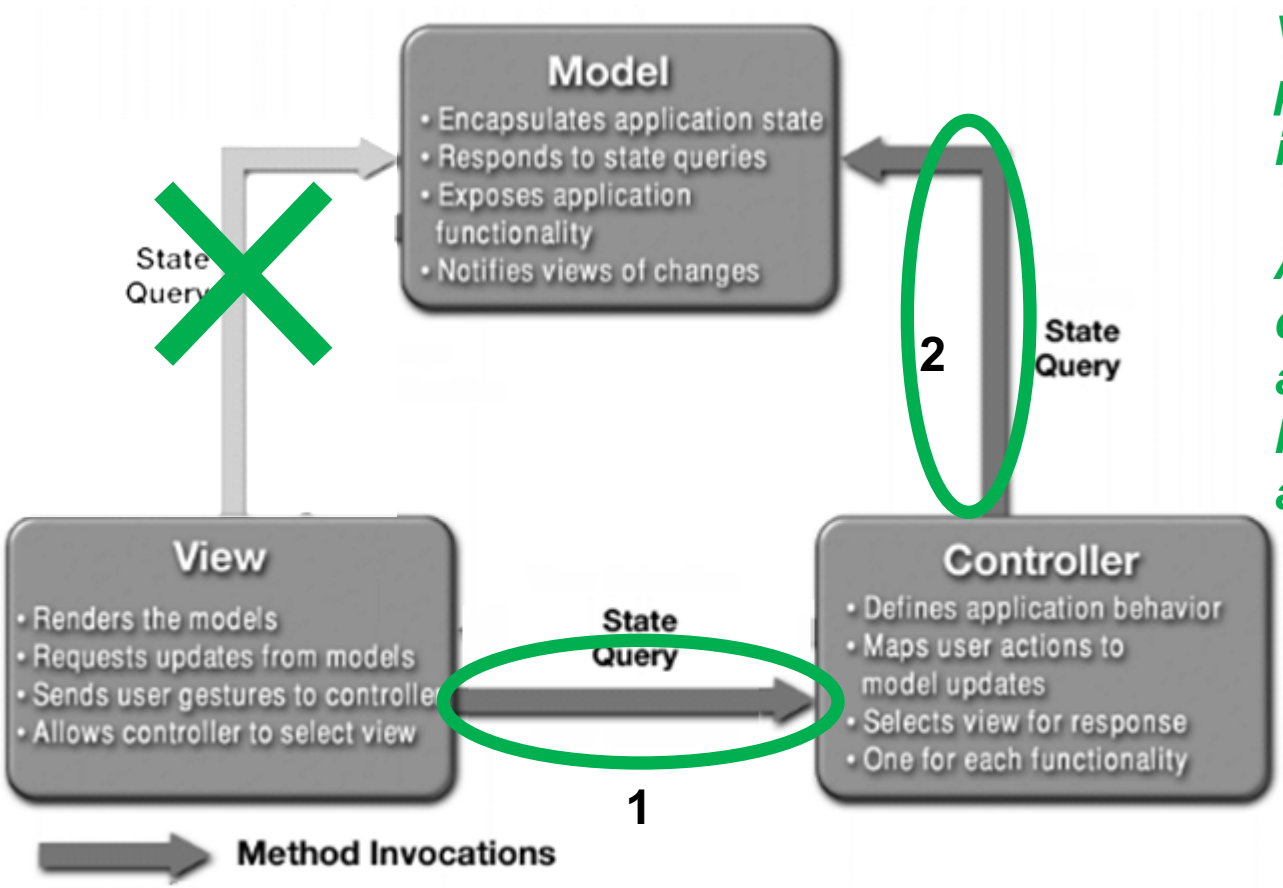
MVC – accès en lecture

Accès à des fonctionnalités qui ne modifient pas l'état du modèle
La vue interroge le modèle pour se mettre à jour (typiquement à son initialisation)
Obligation de passer par une interface d'abstraction



MVC – accès en lecture

Accès à des fonctionnalités qui ne modifient pas l'état du modèle
La vue interroge le modèle pour se mettre à jour (typiquement à son initialisation)
Obligation de passer par une interface d'abstraction

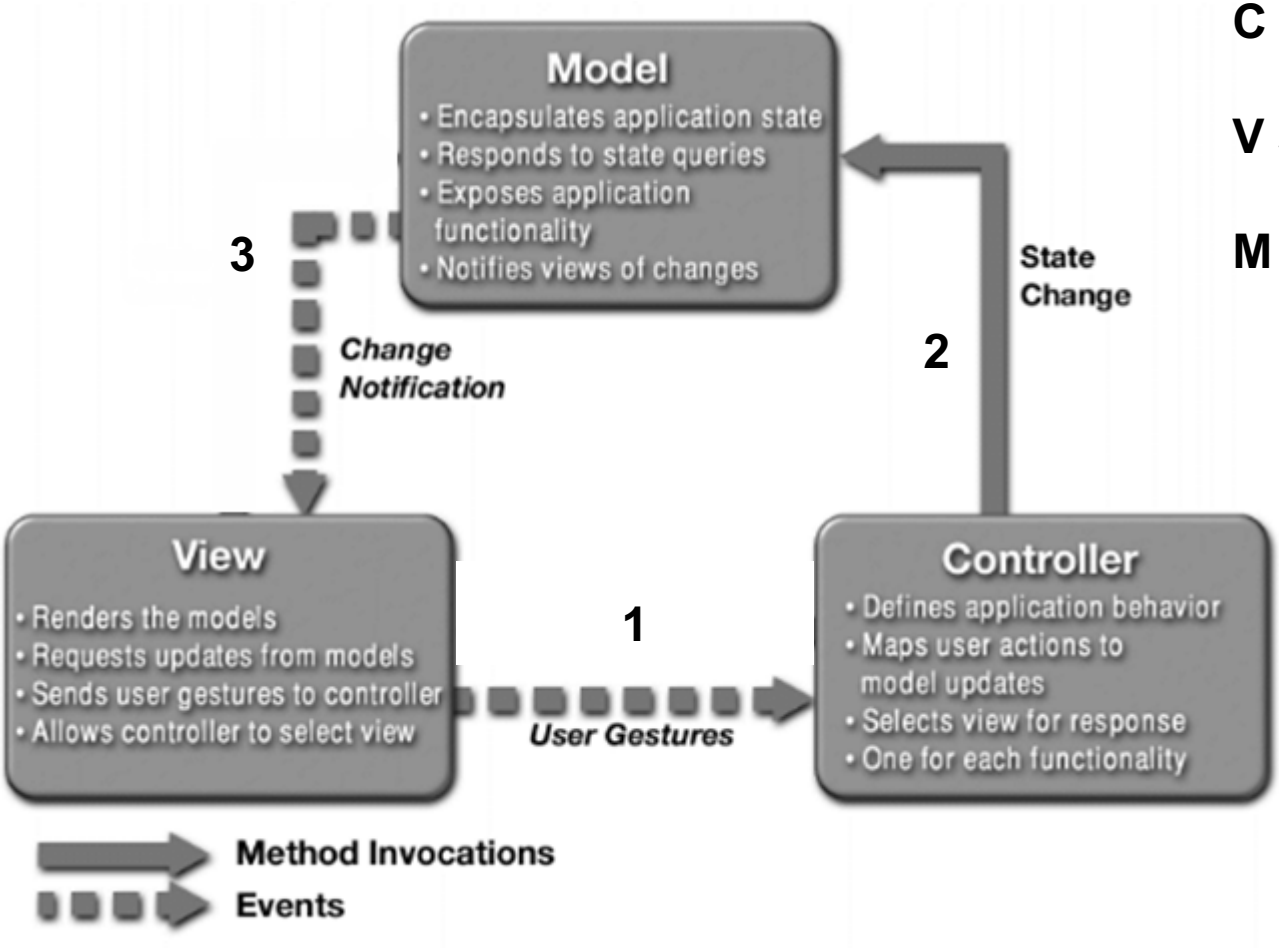


Variante : la vue passe par le contrôleur pour interroger le modèle

Avantage : pas besoin de gérer l'interfaçage abstrait avec le modèle, le contrôleur lui peut y accéder + simplement

MVC – accès en écriture

Accès à des fonctionnalités qui modifient l'état du modèle
La vue déclenche des évènements système écoutés par le contrôleur
Obligation de laisser la main au modèle – MAJ de la vue dépendante de l'état du modèle



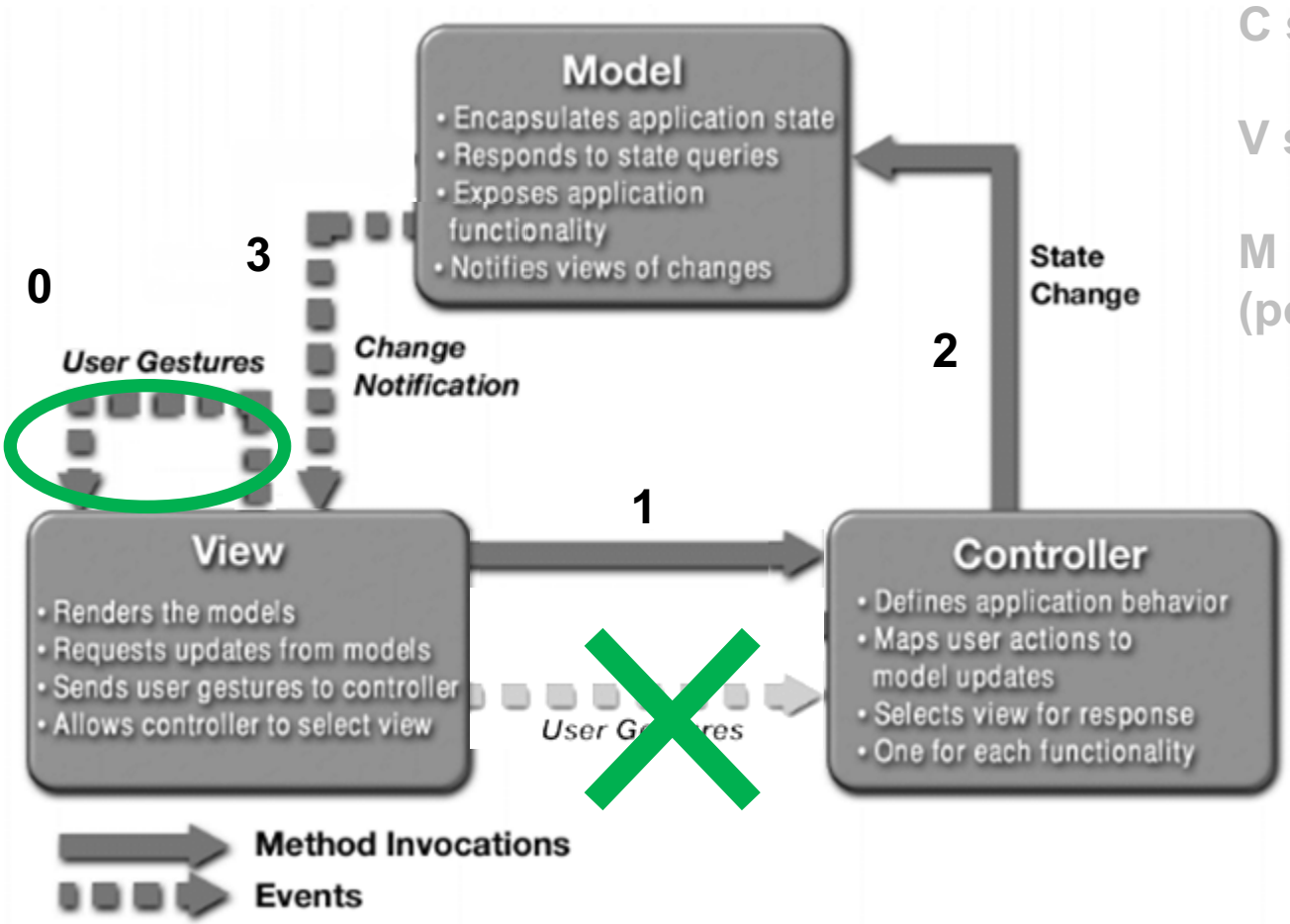
C s'abonne à V

V s'abonne à M

M modifie V implicitement

MVC – accès en écriture

Accès à des fonctionnalités qui modifient l'état du modèle
La vue déclenche des évènements système écoutés par le contrôleur
Obligation de laisser la main au modèle – MAJ de la vue dépendante de l'état du modèle



C s'abonne à V

V s'abonne à M

M modifie V implicitement (polymorphisme)

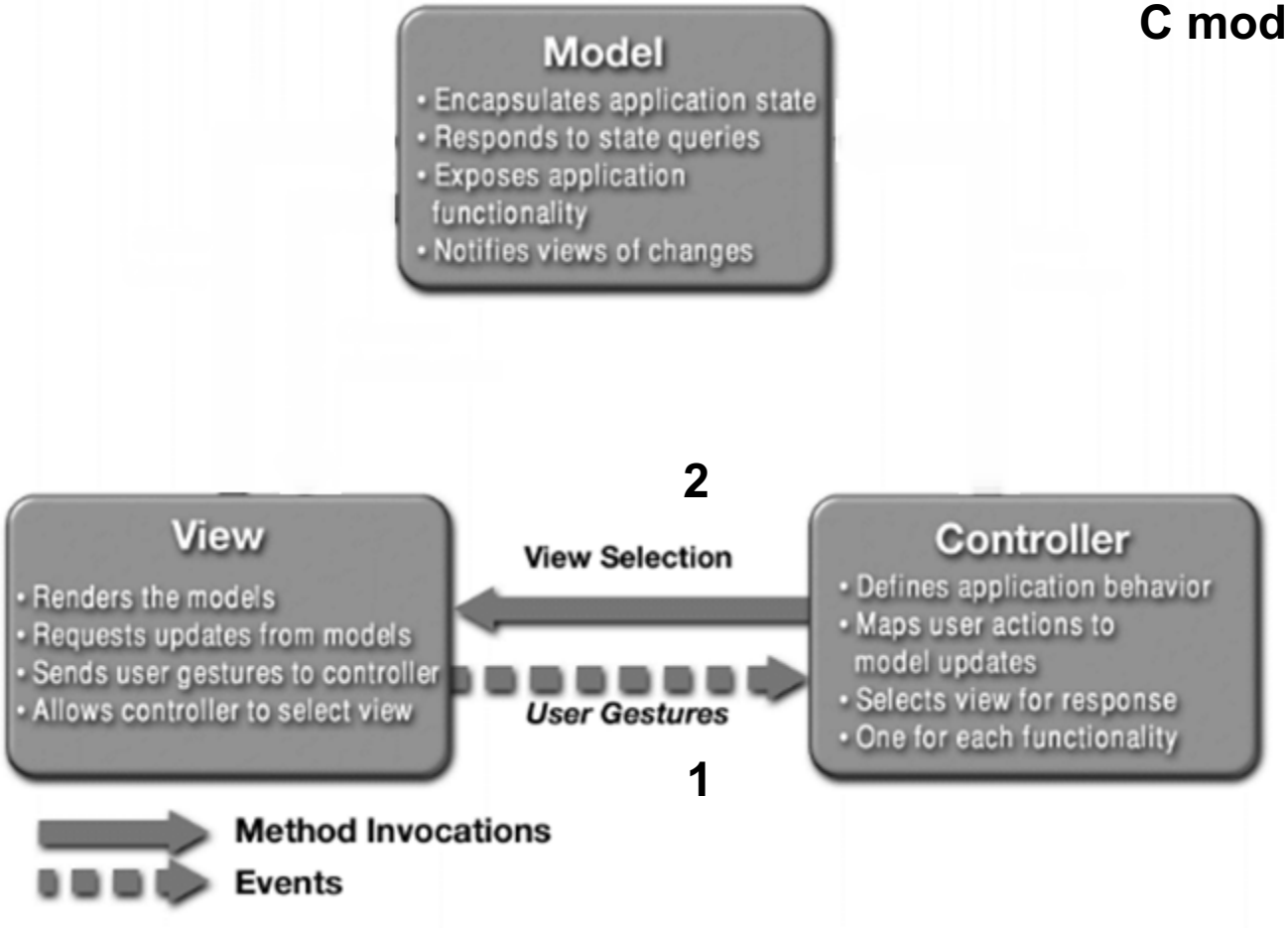
Variante : V écoute les évènements système

Avantage : V transfère les données impliquées à C, peut encapsuler les évènements en évènements de plus haut niveau

MVC – interaction pure

Interaction utilisateur nécessitant seulement un traitement du contrôleur
La vue déclenche des évènements système écoutés par le contrôleur
Pas d'accès au modèle – MAJ de la vue indépendante des données du modèle

C s'abonne à V
C modifie V directement



Exemple MVC pour un éditeur simple

0. initialisation de la vue avec les données du modèle

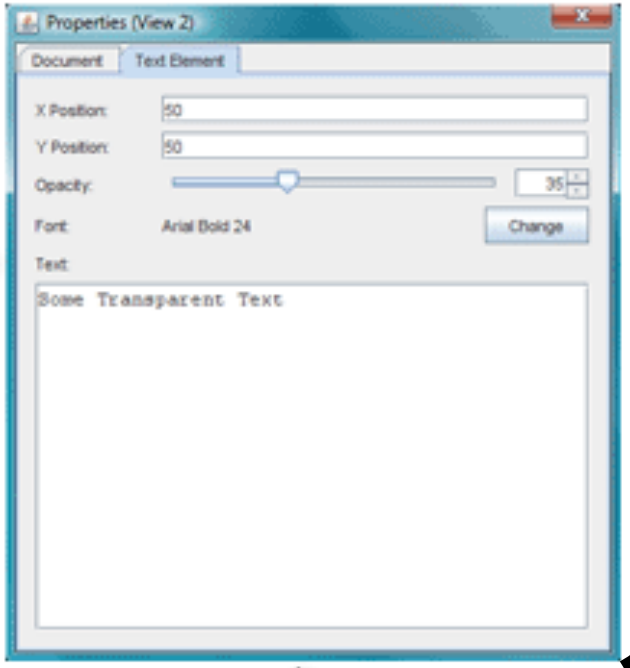
4. écoute des changements d'états du modèle

Model

```
int xPosition = 50;
int yPosition = 50;
double opacity = 0.35;
Font font = new Font("Arial",
   Font.BOLD, 24)
String text = "Some Transparent
   Text";
```

3. délégation au modèle

View



2. appel d'une des 5 méthodes du contrôleur avec la donnée à modifier

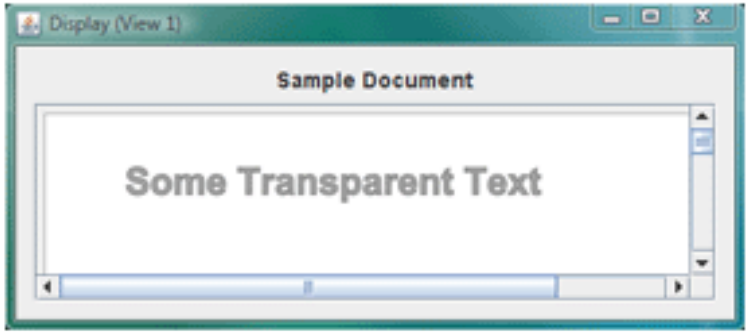
Controller

```
public void changeXPosition(int newXPosition);
public void changeYPosition(int newYPosition);
public void changeOpacity(double newOpacity);
public void changeFont(Font newFont);
public void changeText(String newText);
```

1. écoute de l'événement système lié à l'édition d'un des 5 paramètres

1 modèle – Plusieurs vues

View

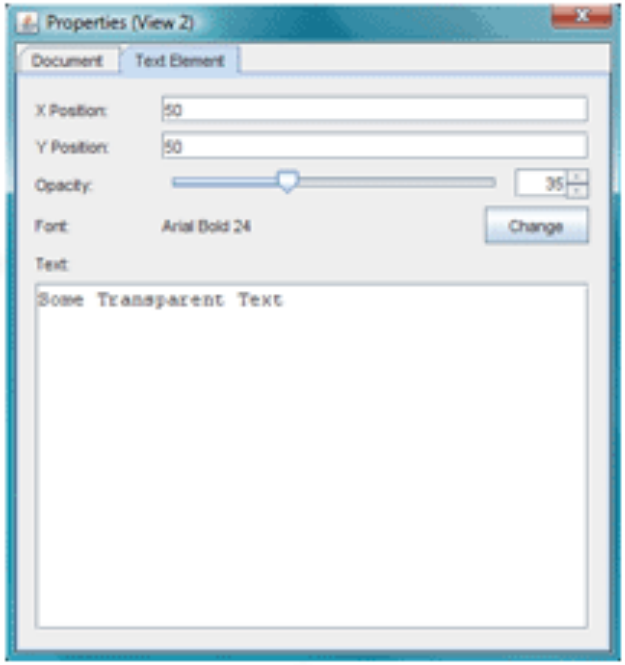


Un ou plusieurs contrôleurs ?

Model

```
int xPosition = 50;
int yPosition = 50;
double opacity = 0.35;
Font font = new Font("Arial",
    Font.BOLD, 24)
String text = "Some Transparent
    Text";
```

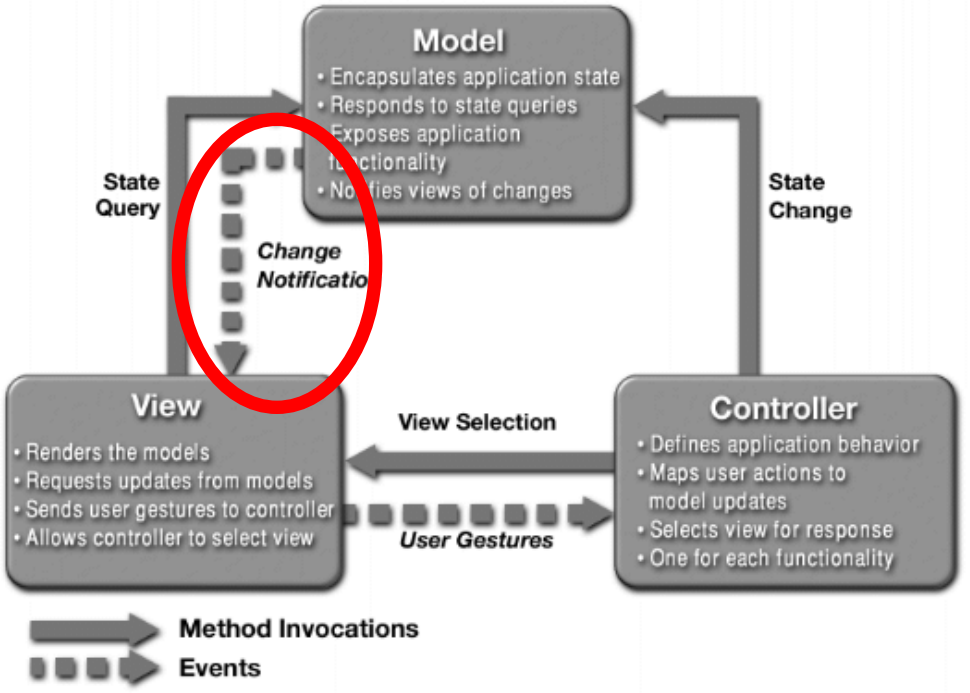
View



Synchronisation entre la vue et le modèle

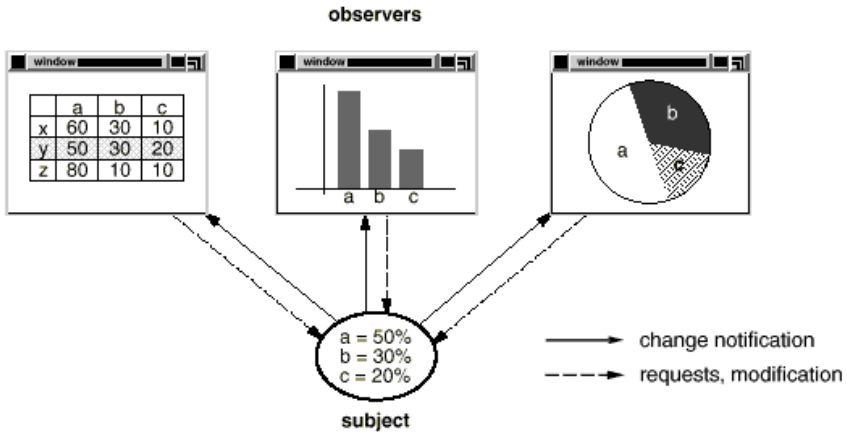
- Se fait par l'utilisation du pattern Observer.
- Permet de générer des événements lors d'une modification du modèle et d'indiquer à la vue qu'il faut se mettre à jour

- Observable = le modèle
- Observers = les vues



Le pattern Observer en bref

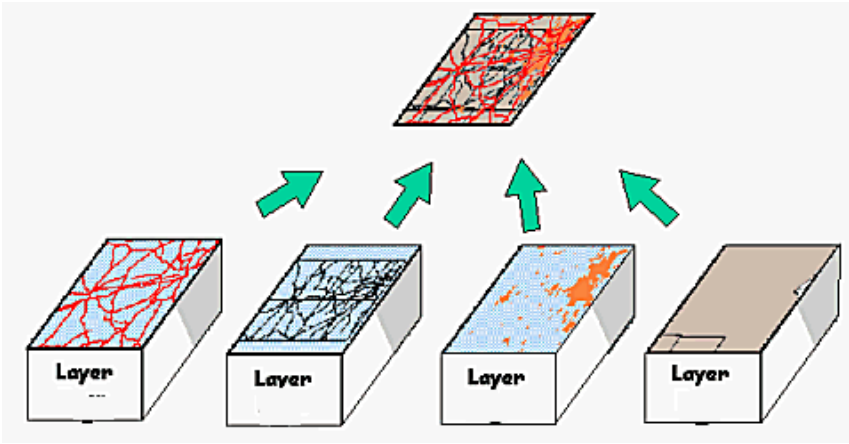
Motivation : Maintenir la consistance entre des objets dépendants les uns des autres sans lier étroitement les classes, parce que cela aurait comme effet de réduire leur réutilisabilité



Moyen : Définir une dépendance de “1” à “n” entre des objets telle que :

lorsque l’état d’un objet change, tous ses dépendants sont informés et mis à jour automatiquement

Utilisation du polymorphisme



Exemples applicatifs : MVC, communication “broadcast”

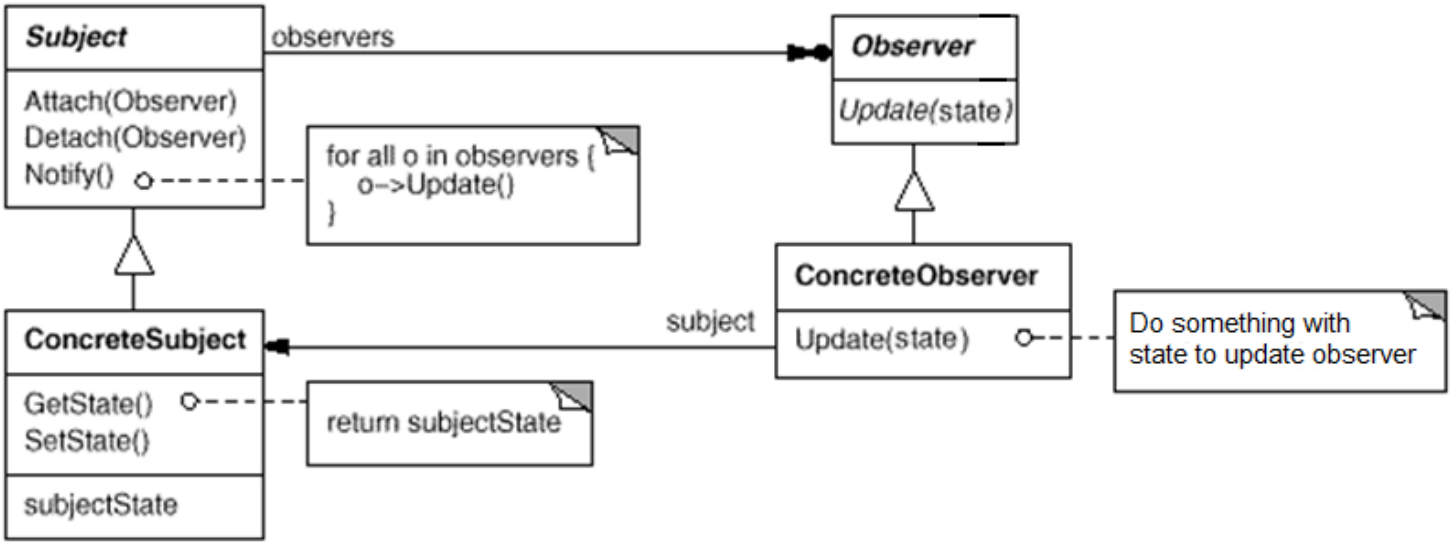
Les listeners java sont une implémentation de ce design pattern

Structure et relations entre les classes du pattern Observer

- Le pattern “Observer” décrit comment établir les relations entre les objets dépendants

- La **source** (l’observé)
  - Peut avoir n’importe quel nombre d’observateurs
  - Tous les observateurs sont informés lorsque l’état de la source change

- L’**observateur**
  - ***peut*** demander à la source son état afin de se synchroniser



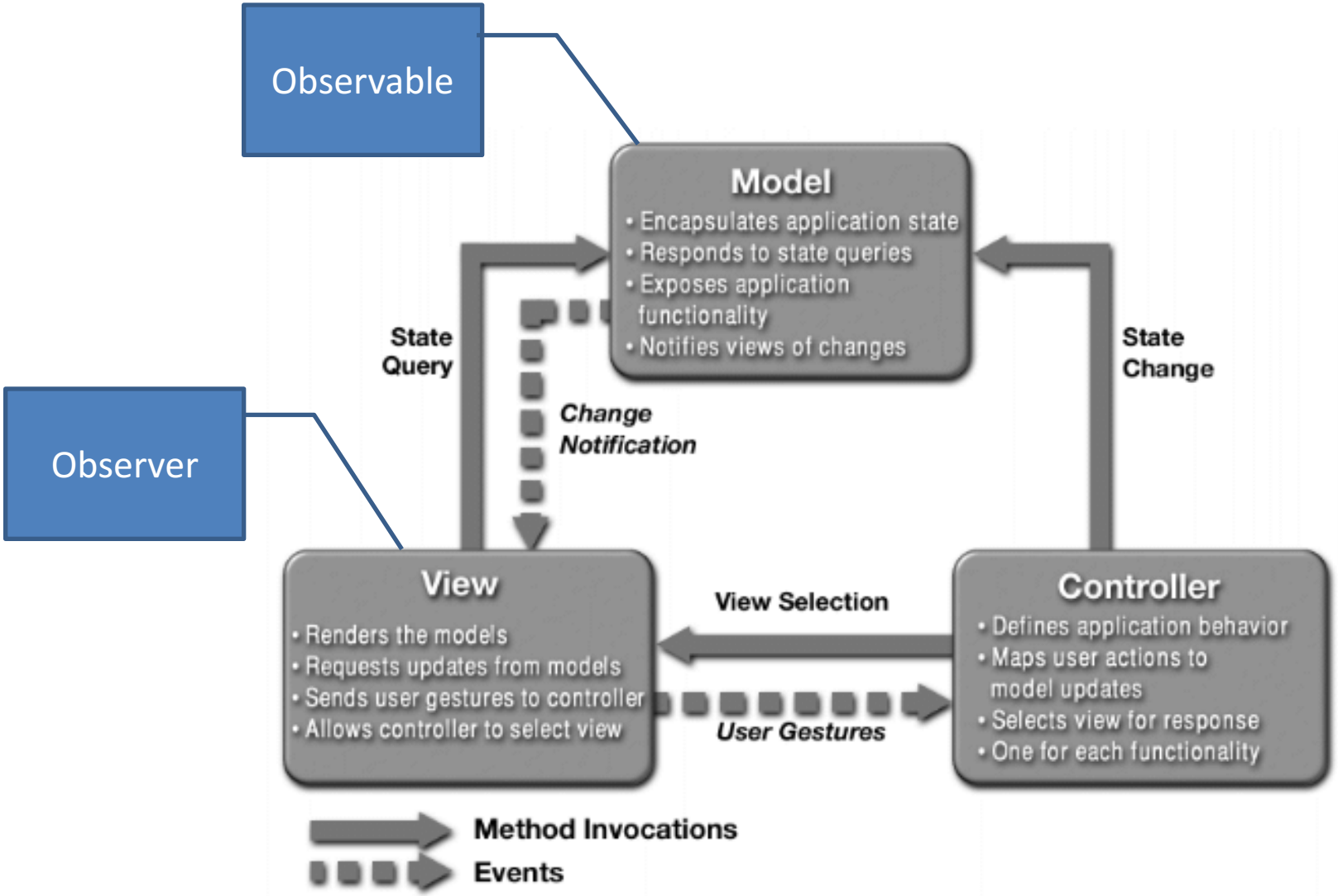
Quand l'appliquer

- Lorsqu'une abstraction possède deux aspects dont l'un dépend de l'autre.
  - L'encapsulation de ces aspects dans des objets séparés permet de les varier et de les réutiliser indépendamment.
  - Exemple : Modèle-Vue-Contrôleur

- Lorsqu'une modification à un objet exige la modification des autres, et que l'on ne sait pas a priori combien d'objets devront être modifiés.

- Lorsqu'un objet devrait être capable d'informer les autres objets sans faire d'hypothèses sur ce que sont ces objets,

Le pattern Observer et MVC



Implémentations Java du pattern Observer

METHODE 1 (solution clé en main)

Une classe et une interface : ***class Observable {... }*** et interface ***Observer*** dans le package java.util

Un objet Observable doit être une instance de la classe qui dérive de la classe Observable

void addObserver(Observer o)

void notifyObservers(Object arg)

Un objet observer doit être instance d'une classe qui implémente l'interface Observer

void update(Observable o, Object arg)

METHODE 2 (solution à mettre en place soi même)

Une interface ***EventListener*** et une classe ***EventObject*** dans le package java.util et une classe ***EventListenerList*** dans le package javax.swing.event

(cf exemple ci-après)

➡ Seconde méthode plus flexible et modulaire que la première mais moins rapide à mettre en oeuvre

Conclusions sur MVC

- MVC permet de :
  - Changer une couche sans altérer les autres
  - Synchroniser les vues. Toutes les vues qui montrent la même chose sont synchronisées
- Pour s'assurer une bonne séparation des couches, vérifier que :
  - Les vues ne connaissent pas le type concret des données des modèles
  - Les contrôleurs et les modèles ne connaissent pas le type concret des composants d'interaction des vues