heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Ecole d'Ingénieurs de l'état de Vaud
ERIC LEFRANÇOIS
10 août09

Génie Logiciel- Le modèleMVC

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Sommaire

1 *En préliminaire: les modèles de conception*

- Ne pas réinventer la roue à chaque expérience
- Mieux réutiliser des solutions qui ont fait leurs preuves

☞Les **modèles de conception**sont des « bonnes solutions » que l'on réutilise systématiquement.

☺ Plus connus sous le vocable anglais «**design patterns**».

Documentation

- « Patterns in Java » de [Mark Grand] chez Wiley
- «UML & design patterns» de [Craig Larman] chez Wiley

DESCRIPTION D'UN MODÈLE DE CONCEPTION

S'articule autour de 3 particularités essentielles:

- **Le nom du modèle**
  Modèle « **MVC** »
  Modèle du « **Producteur-consommateur** »
  Etc..
  ☺ Quasi standardisé dans le monde informatique !

- **L'énoncé du problème**
  ☞ Situations dans lesquelles le modèle peut s'appliquer

- **La solution**
  - Décrit les éléments à mettre en jeux (classes, d'objets) et leurs associations
  - Solution toujours générique, à adapter à un contexte particulier

2 *Le modèle MVC*

Origine

Résoudre les problèmes d'interfaçage avec l'utilisateur avec le langage Smalltalk (Xérox, Palo Alto en Californie)

Aujourd'hui

☞Un design patternpour **concevoir l'architecture générale**d'une petite application comportant une interface graphique (p.e. : un labo en POO)

☺Non limité toutefois aux applications de petite taille !

⇨ Intégré depuis dans le **pattern architectural**«**Couches**» («**Layers**»)

PRINCIPE DU MODÈLE MVC

Séparer et **découpler** les 3 composantes d'un *module* :

- La composante **Modèle:** une ou plusieurs classes !
  ⇨**Structures d'informations** manipulées par le module

- La composante **Vue :** une ou plusieurs classes !
  ⇨**Interface avec l'utilisateur (Interaction)**

- La composante **Contrôleur :** une seule classe en général !
  ⇨**Contrôle général** : ordonnancement des opérations (« **workflow** »)

PRINCIPE DU MODÈLE MVC

Séparer et **découpler** les 3 composantes d'un module :

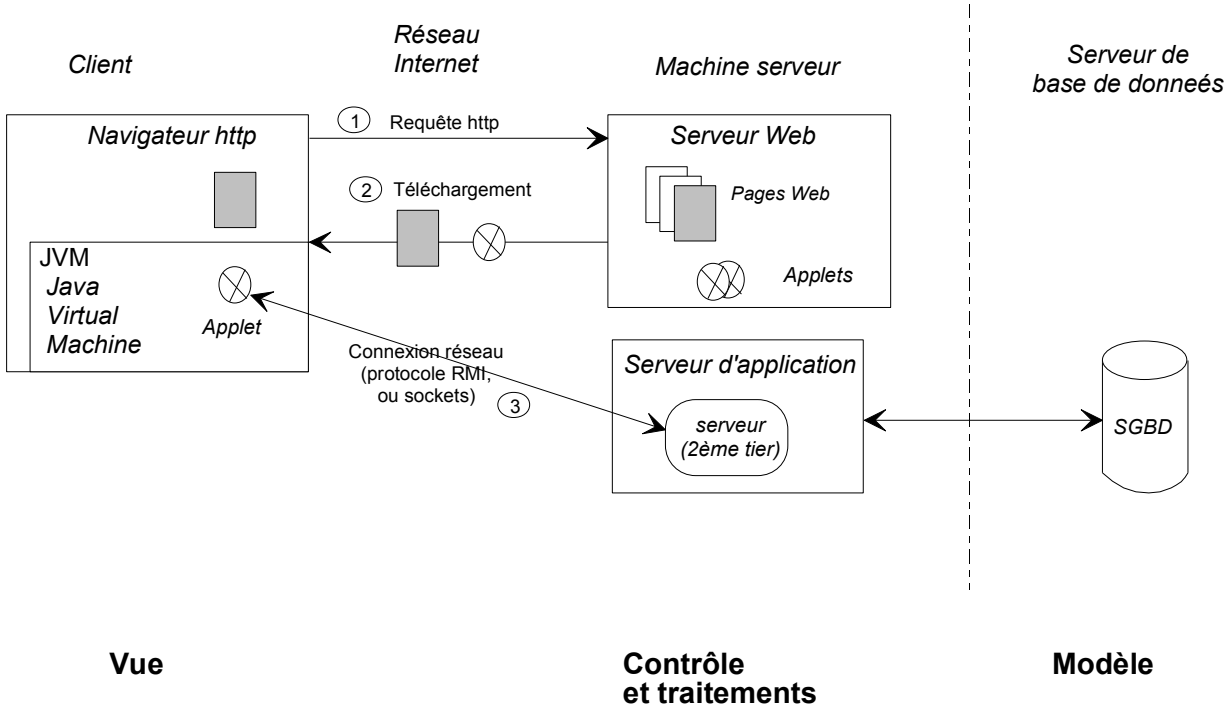
- **Modèle** une ou plusieurs classes !
- **Vue** une ou plusieurs classes !
- **Contrôleur** une seule classe en général !

Objectif ?

Fort découplage de M, V et C ⇨ pour améliorer

- **Maintenance et mise au point** - Modification indépendante
- **Réutilisation** - Nouvelles applications

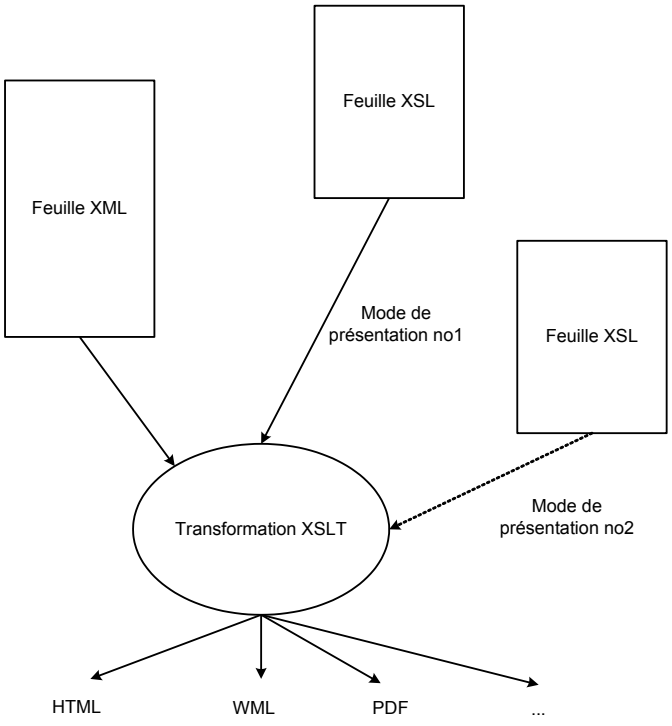
ANALOGIES MVC AVEC MODÈLE CLIENT-SERVEUR « 3-TIERS »



ANALOGIES MVC AVEC CONCEPT XML-XSL

XML(modèle) + XSL (vue)

> HTML



2.1 Le «M» comme «Modèle»

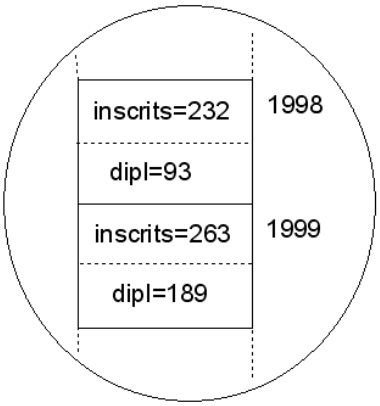
Définition 1: La couche Modèle

Paquetage qui comprend l'ensemble des objets modèles de l'application.

Un exemple⇨ Application destinée à établir et visualiser le bilan d'une école

Un modèle pour deux types d'informations

- le nombre d'inscriptions (`nbInscrits`)
- le nombre de diplômés (`nbDiplômés`)



2.2 Le «V» comme «Vue»

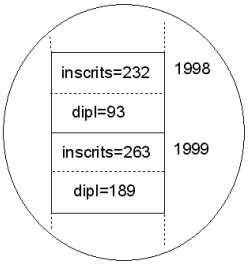
Définition 2: La couche Vue

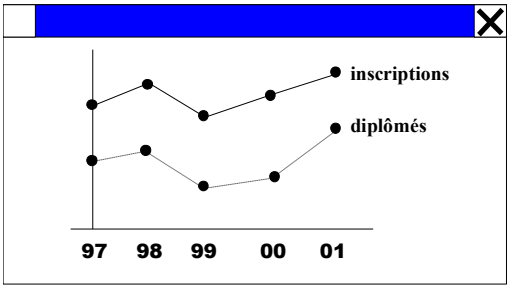
Paquetage qui comprend l'ensemble des vues de l'application.

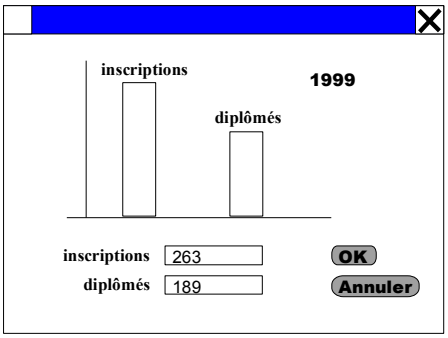
La vue permet

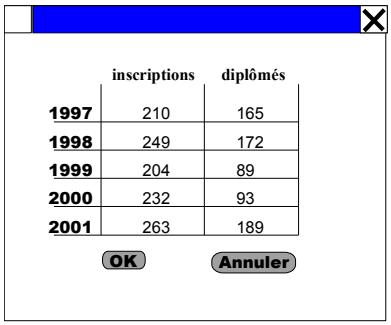
- d'une part de visualiser la valeur d'un objet **modèle**
- d'autre part (le cas échéant), de modifier la valeur d'un tel objet

Exemple avec 3 vues présentées **simultanément** :









	inscriptions	diplômés
1997	210	165
1998	249	172
1999	204	89
2000	232	93
2001	263	189

OK Annuler

2.3 Couplage Vue-Modèle: Le modèle «Observable-Observé»

☹ **Un constat : La non-réutilisation des couches Vue & Contrôleur !**

☺ La couche Modèle (p.e. une « Pile d'informations ») a plus de chance de pouvoir être réutilisée..

⇨ Bien séparer le Modèle de la Vue

Séparer le Modèle de la Vue ⇨ Objectifs

- Permettre de développer séparément les couches Modèle et Vue
- Réduire l'impact de l'évolution des besoins de l'interface sur la couche Modèle
- Permettre d'ajouter facilement de nouvelles vues sans avoir à modifier la couche Modèle
- Permettre d'avoir plusieurs vues simultanées sur le même objet modèle
- Permettre de déployer la couche Modèle et la couche Vue de manière distribuée (couches communiquant à distance sur le réseau, avec le modèle du coté Serveur et la Vue du côté Client).
- Améliorer la portabilité, la réutilisation de la couche Modèle

Pour appliquer ce principe:

- Les objets Modèle ne doivent pas «connaître» les objets Vue
  ~~Leur demander par exemple d'afficher quelque chose, de changer de couleur, etc.~~

  ☺ Mais les objets de la couche Vue peuvent connaître directement les objets de la couche Modèle !
  La couche Vue n'a pas la prétention d'être une couche réutilisable.

- Les objets de la Vue sont dotés de peu d'intelligence et doivent se contenter :
  - de gérer les entrées/sorties,
  - de capturer les événements de l'interface,

  ~~Ne doivent surtout pas offrir de fonctionnalités propres à l'application (le propre des composants Modèle et/ou Contrôleur)~~

PRINCIPE DU MODÈLE OBSERVABLE-OBSERVE

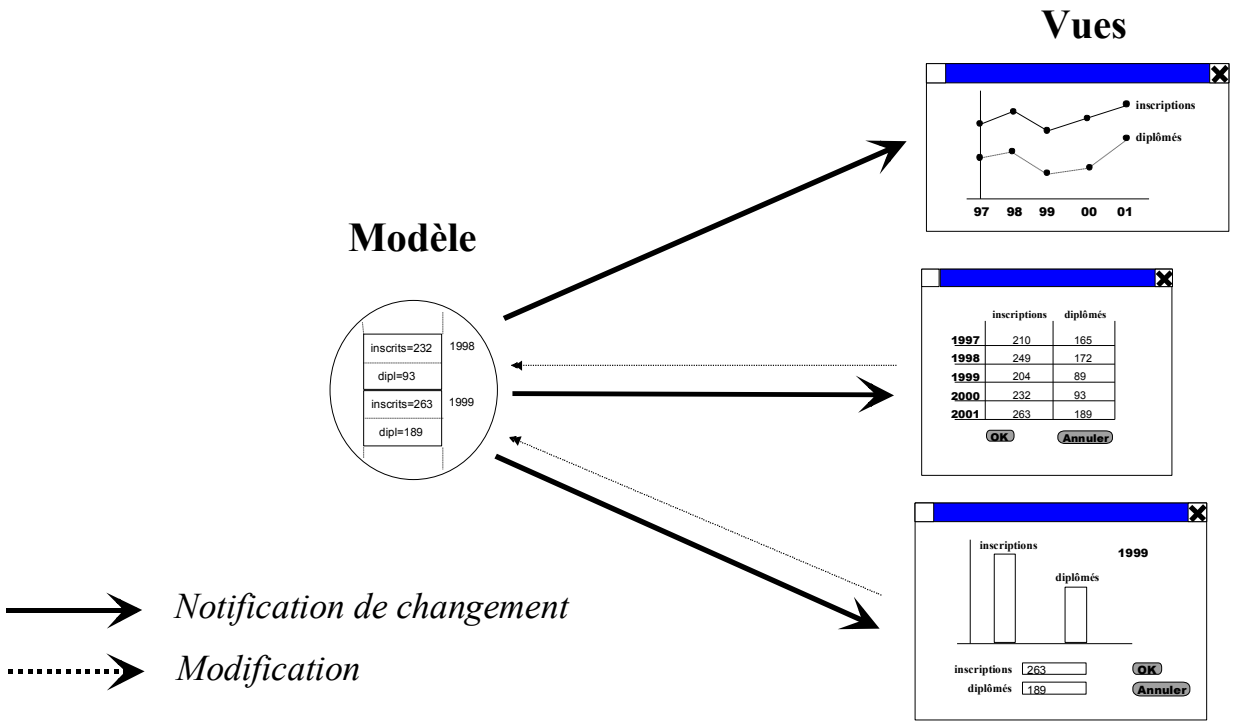
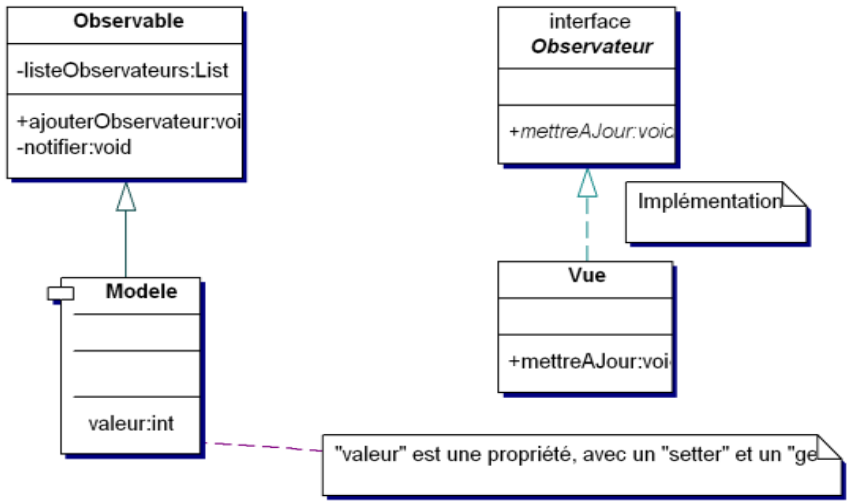
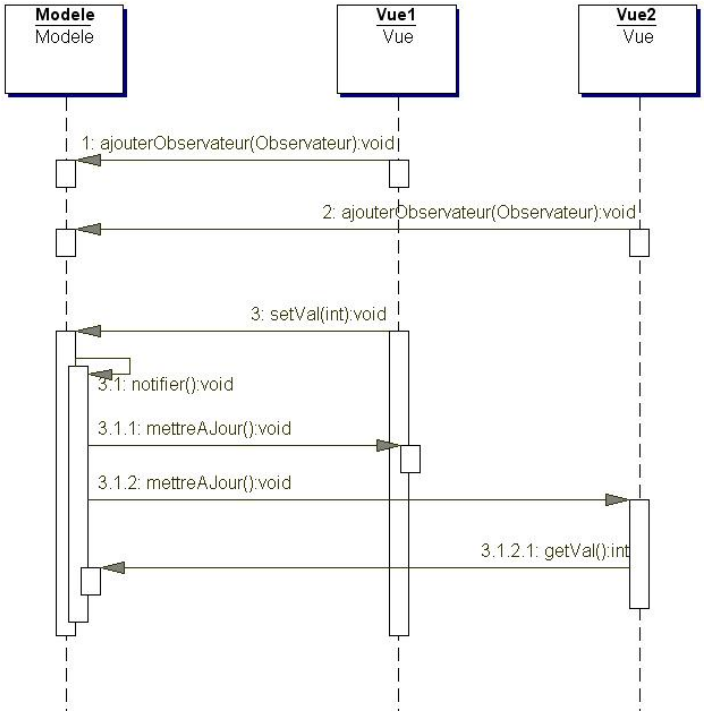


DIAGRAMME DE CLASSES DU MODÈLE OBSERVABLE-OBSERVÉ



- Une vue s'abonne au modèle : `leModèle.ajouterObservateur(this) ;`
- A chaque changement d'état le modèle invoque sa méthode privée `notifier()` ⇨diffuse le message `mettreAJour()` à tous les abonnés.

DIAGRAMME DE SEQUENCE

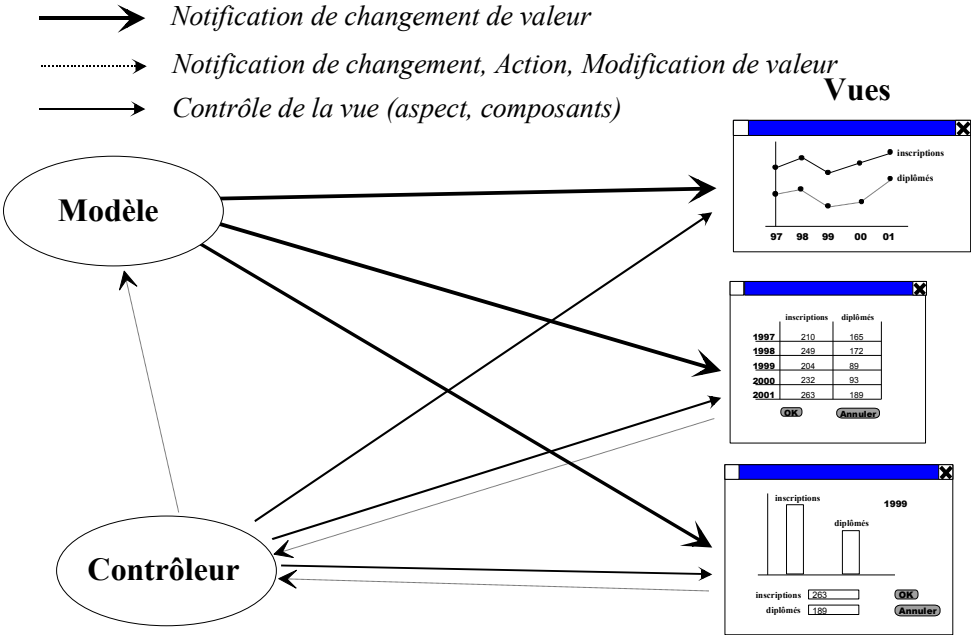


2.4 «C» COMME CONTROLEUR

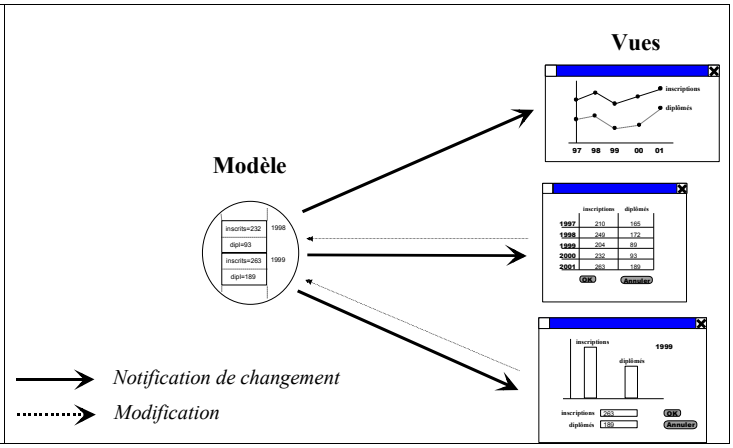
Contrôleur

⇨ **Big boss**

- Définit l'ordre des opérations à effectuer («**workflow**»)
- Autorise ou non les actions commandées par l'utilisateur
- Agit sur l'aspect des différentes vues (en activant tel ou tel widget)



Elimination du contrôleur ?



Si le contrôleur se contente de jouer les **intermédiaires** entre le modèle et les différentes vues..

⇨ Possible de l'éliminer ⇨ On gagne en flexibilité et en efficacité

Conseil : Placer son intelligence dans le Modèle (la Vue, devrait rester le plus «bête» possible)

☺ **De manière générale, il est déconseillé d'éliminer le contrôleur !**

2.4.1 Java et le modèle Observable-Observé

« `java.util` »

☞

- Interface `Observer`
- Classe `Observable`

API OBSERVER

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

- L'observable doit invoquer sa méthode **notifyObservers**
- ⇨**update** est alors invoquée pour tous ses observateurs

Paramètres:

o - l'objet observé (utile si le même observateur observe plusieurs objets différents)

arg - un argument passé par la méthode `notifyObservers`

API : LA CLASSE OBSERVABLE

- `public void addObserver(Observer o)`
- `public void deleteObserver(Observer o)`
- `protected void setChanged()`

  Pour enregistrer le fait que l'état de cet objet a été modifié: passage du flag interne «`modified`» à l'état `true`.

- `public void notifyObservers()`
  - Pour notifier tous les observateurs que cet objet a été modifié. L'argument `arg` reçu par `update`vaudra`null`
  - Cette notification a lieu si et seulement si la méthode **setChanged** a été invoquée au préalable !
  - L'exécution de `notifyObservers` a pour effet de remettre le flag interne «`modified`» à `false`.

- `public void` **`notifyObservers`**`()`
  - Pour notifier tous les observateurs que cet objet a été modifié. L'argument `arg` reçu par `update`vaudra`null`
  - Cette notification a lieu si et seulement si la méthode **`setChanged`** a été invoquée au préalable !
  - L'exécution de `notifyObservers` a pour effet de remettre le flag interne «`modified`» à `false`

- `public void` **`notifyObservers`**`(Object` **`arg`**`)`
  Idem..
  - L'argument **`arg`** sera reçu par la méthode `update` de l'observateur (deuxième paramètre de la méthode `update`).

2.4.2 Le modèles des listeners (Swing)

Le modèle Observable-Observé est efficace dans la mesure où l'observable ne génère qu'un seul type d'événements.

Sinon..les observateurs risquent d'être notifiés pour des événements qui ne les intéressent pas forcément !
⇨ Perte d'efficacité de l'application

Exemple :Le composant Swing `JButton`génère plusieurs événements

- Des événements de type **`Action`** (bouton pressé);
- Des événements de type **`Focus`** (le bouton a obtenu le focus);
- Des événements de type **`Key`** (une touche a été pressée alors que le bouton a le focus);
- Des événements de type **`MouseMotion`**(la souris est déplacée sur le bouton);
- etc..

Exemple :Le composant Swing `JButton`génère plusieurs événements

- Des événements de type **`Action`** (bouton pressé);
- Des événements de type **`Focus`** (le bouton a obtenu le focus);
- Des événements de type **`Key`** (une touche a été pressée alors que le bouton a le focus);
- Des événements de type **`MouseMotion`**(la souris est déplacée sur le bouton);
- etc..

Solution : Le «**modèle des listeners**»..

Une variante de « Observable-Observé »

⇨ Un observateur potentielpeut s'abonner uniquement à un sous-ensemble particulier d'événements

⇨`addActionListener`, `addFocusListener`, `addKeyListener`, ..

2.4.3 MVC et le pattern «Couches»

MVC intégré dans le **pattern architecturalCouches** («**Layers**»).

⇨ Base à l'élaboration de la structure à grande échelle d'un système

Modèle MVC		**Modèle Layers**
Vue	⇨	**Présentation**
Modèle	⇨	**Domaine**
Contrôleur	⇨	**Application**

LE PATTERN COUCHES (LAYERS)