

Initiation génie logiciel : architecture logicielle (1/2)

Arnaud Labourel (arnaud.labourel@univ-amu.fr)

4 octobre 2024

amU Faculté
des sciences
Aix Marseille Université

Section 1

Introduction

Une **application**, c'est :

- des données (le **modèle**),
- des traitements (**couches logicielles** ou **services**),
- un point d'accès et une interface :
 - ▶ pour un utilisateur ou
 - ▶ pour une autre application

Objectifs d'une bonne architecture logicielle

Faciliter le développement, le déploiement, l'évolutivité et la maintenance d'un système logiciel.

- Laisser le plus longtemps possible les possibilités ouvertes (pas fixer des choix technologiques dès le début dans les choix d'architecture)
- Penser la problématique du déploiement
- Anticiper l'évolutivité du code, car la maintenance est généralement la partie la plus coûteuse d'un projet.

Contenu d'une description d'architecture logicielle

- Décrire l'organisation générale d'un système et sa décomposition en sous-systèmes ou composants
- Déterminer les interfaces entre les sous-systèmes :
 - ▶ Décrire les interactions et le flot de contrôle entre les sous-systèmes
 - ▶ Déterminer les dépendances entre sous-systèmes
- Décrire également les composants utilisés pour implanter les fonctionnalités des sous-systèmes :
 - ▶ Les propriétés de ces composants
 - ▶ Leur contenu (classes, package, autres composants, ...)
 - ▶ Les machines ou dispositifs matériels sur lesquels ces modules seront déployés

Utilité d'une architecture logicielle (Garlan 2000)

- **Compréhension** : facilite la compréhension des grands systèmes complexes en donnant une vue de haut-niveau de leur structure et de leurs contraintes. Les motivations des choix de conception sont ainsi mis en évidence
- **Réutilisation** : favorise l'identification des éléments réutilisables, parties de conception, composants, caractéristiques, fonctions ou données communes
- **Construction** : fournit un plan de haut-niveau du développement et de l'intégration des modules en mettant en évidence les composants, les interactions et les dépendances

Utilité d'une architecture logicielle (Garlan 2000)

- **Évolution** : met en évidence les points où un système peut être modifié et étendu. La séparation composant/connecteur facilite une implémentation du type « plug-and-play »
- **Analyse** : offre une base pour l'analyse plus approfondie de la conception du logiciel, analyse de la cohérence, test de conformité, analyse des dépendances
- **Gestion** : contribue à la gestion générale du projet en permettant aux différentes personnes impliquées de voir comment les différents morceaux du casse-tête seront agencés. L'identification des dépendances entre composants permet d'identifier où les délais peuvent survenir et leur impact sur la planification générale

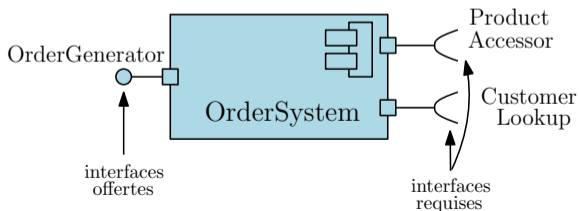
Permet de visualiser les composants dans l'environnement de développement :

- Offre une vue de haut niveau de l'architecture du système
- Utilisé pour décrire le système d'un point de vue implémentation
- Permet de décrire les composants d'un système et les interactions entre ceux-ci
- Illustre comment grouper concrètement et physiquement les éléments du système au sein des composants

Composant

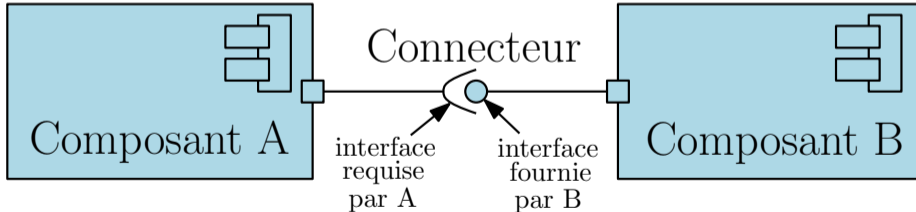
Unité modulaire avec des interfaces bien définies qui est remplaçable dans son environnement :

- Requier une ou plusieurs interfaces
- Fournit une ou plusieurs interfaces
- Sa partie interne n'est pas détaillée
- Ses dépendances sont conçues de telle sorte que le composant peut être traité de façon aussi autonome que possible



Connecteurs entre composants

- Assurent les interactions entre composants
- Peuvent être de complexité variable (appel de méthode, système de requêtes avec ordonnanceurs, API REST, ...)
- Permettent la flexibilité et l'évolution



Objectif

Construire un système logiciel permettant de gérer des commandes en ligne.

Le Client peut :

- Rechercher et parcourir des articles dans l'inventaire
- Créer des commandes et y ajouter des articles
- Créer un compte

On veut aussi gérer le stock.

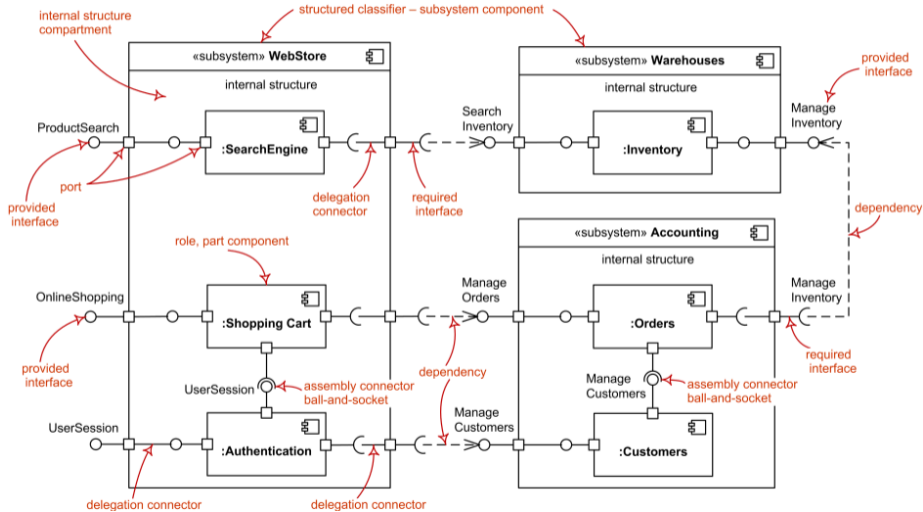
Interfaces des sous-systèmes

- WebStore (magasin en ligne)
 - ▶ fournissant les interfaces `ProductSearch`, `OnlineShopping` et `UserSession`
 - ▶ ayant besoin des interfaces `SearchInventory`, `ManageOrder` et `ManageCustomers`
- WareHouses (entrepôts)
 - ▶ fournissant les interfaces `SearchInventory` et `ManageInventory`
- Accounting (comptabilité)
 - ▶ fournissant les interfaces `ManageOrder` et `ManageCustomers`
 - ▶ ayant besoin de l'interface `ManageInventory`

Décomposition des sous-systèmes en composants :

- WebStore (magasin en ligne) qui contient 3 composants :
 - ▶ SearchEngine : moteur de recherche de produit
 - ▶ ShoppingCart : gestion de panier
 - ▶ Authentication : gestion de l'authentification des utilisateurs
- WareHouses (entrepôts) avec un composant :
 - ▶ Inventory : gestion de l'inventaire
- Accounting (comptabilité) avec 2 composants :
 - ▶ Customers : gestion des comptes clients
 - ▶ Orders : gestion des commandes

Diagramme de composants pour magasin en ligne



Section 2

Styles d'architecture logicielle

Styles d'architecture logicielle

- L'architecture logicielle, tout comme l'architecture traditionnelle, peut se catégoriser en styles.
- Un système informatique pourra utiliser plusieurs styles selon le niveau de granularité ou l'aspect du système souhaité.

Quelques exemples de styles/patrons d'architecture logicielle

- Multi-couches ;
- Orientée service ;
- Modèle-Vue-Contrôleur et variantes ;
- ...

Évolution des architectures

Architecture monolithique :


IHM + App. + Métier + Données



Serveurs de données :

IHM + App. + Métier


Données



Architecture client-serveur :

IHM + App.

Métier + Données




Architecture 3-tiers :

IHM + App.

Métier

Données



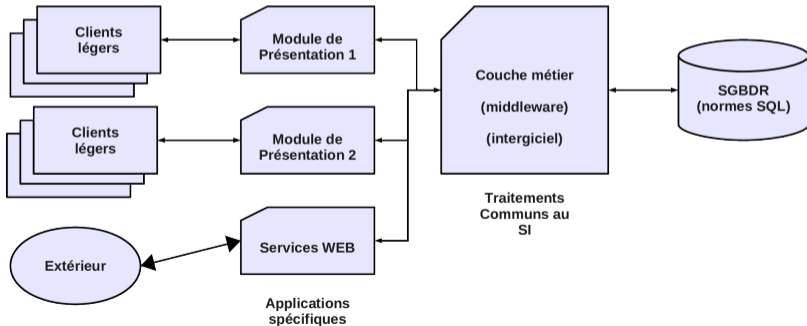
Architecture multi-couches (1990)

- Organisation hiérarchique du système en un ensemble de couches
- Des interfaces bien définies entre les couches
- Chaque couche agit à la fois comme un
 - ▶ Serveur : Fournisseur de services de couches supérieures
 - ▶ Client : consommateur de services de couches inférieures
- Les connecteurs sont des protocoles d'interaction entre couches
- Objectifs :
 - ▶ Réduire la complexité,
 - ▶ Améliorer la modularité, réutilisabilité, maintenabilité
- Différents critères de stratification selon les besoins

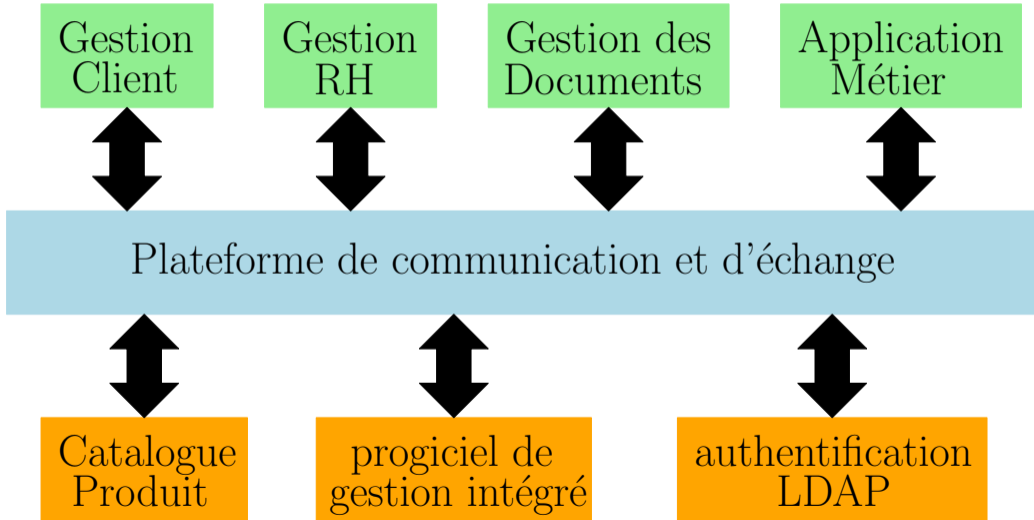
Architecture 3-tiers

Principe : Séparation entre

- la couche de gestion des données (données métier),
- la couche de présentation (logique applicative) et
- la couche métier (actions métier de traitement des données métier).



Architecture orienté service SOA (1995)



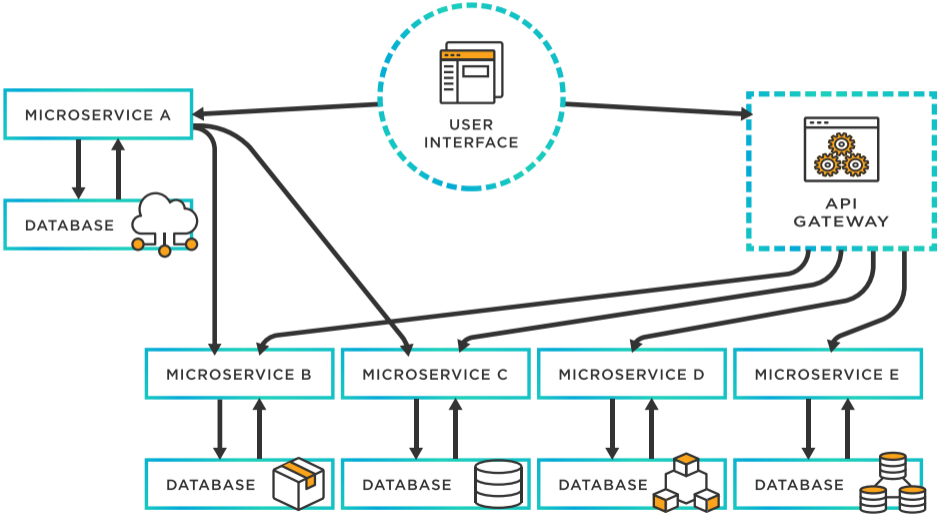
Objectif d'une architecture orientée service

Structurer une application comme un ensemble de services communiquant entre eux via des protocoles de communication standardisés.

Avantages :

- couplages faibles entre services
- interopérabilité augmentée, car les échanges se font via des standards très répandus (API REST, XML, ...)
- mécanisme de publication des services disponibles afin de les incorporer facilement (annuaire de service)

Architecture en microservices (2005)



Définition

Extension de l'approche SOA mais avec des services plus petits.

- services petits (pas de réel consensus sur la taille mais potentiellement gérable par une équipe de 3 à 10 personnes)
- dépendant d'outils de déploiement continu

Avantages de la granularité plus petite :

- permet une meilleure gestion des équipes
- permet de mieux gérer les mises à l'échelle potentielles

Section 3

Patrons pour interface graphiques GUI : MVC, MVP, MVVM

La motivation les patrons d'architecture de GUI est la séparation des responsabilités afin d'augmenter la cohésion et réduire le couplage.

Les responsabilités dans les interfaces graphiques font référence à deux logiques :

- **Logique du domaine ou logique métier** : partie de l'application qui crée, stocke et modifie les données et qui leur donne du sens. Elle est spécifique du domaine d'application et indépendante des problèmes d'interaction avec l'utilisateur.
- **Logique de présentation** : désigne tout type de logique qui spécifie comment l'interface graphique réagit aux interactions avec l'utilisateur ou aux changements induits dans le modèle de domaine.

Modèle-Vue-Contrôleur (Tygve Reenskaug 1979)

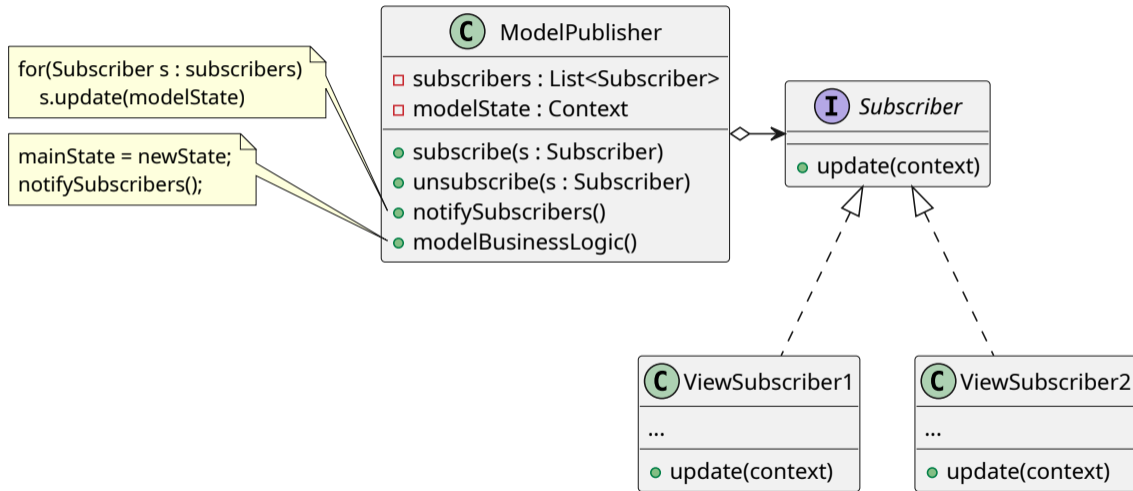
Objectif

Séparer la couche interface utilisateur des autres parties du système (car les interfaces utilisateurs sont beaucoup plus susceptibles de changer que la base de connaissances du système)

- **Modèle** : logique métier et état courant de l'interface durant le dialogue avec l'utilisateur
- **Vue** : logique de présentation (exemple une fenêtre), ne gère pas les interactions de l'utilisateur
- **Contrôleur** : reçoit les interactions de l'utilisateur et les traite, fait la liaison entre la vue et le modèle

La mise à jour de la vue lors des changements du modèle se fait par le patron de conception *observer*.

Patron de conception *observer*



Intention

Mécanisme de souscription pour envoyer des notifications à plusieurs objets observateurs.

Avantages :

- Facile de rajouter des classes d'observateurs (OCP).
- Permet d'établir des relations à l'exécution.

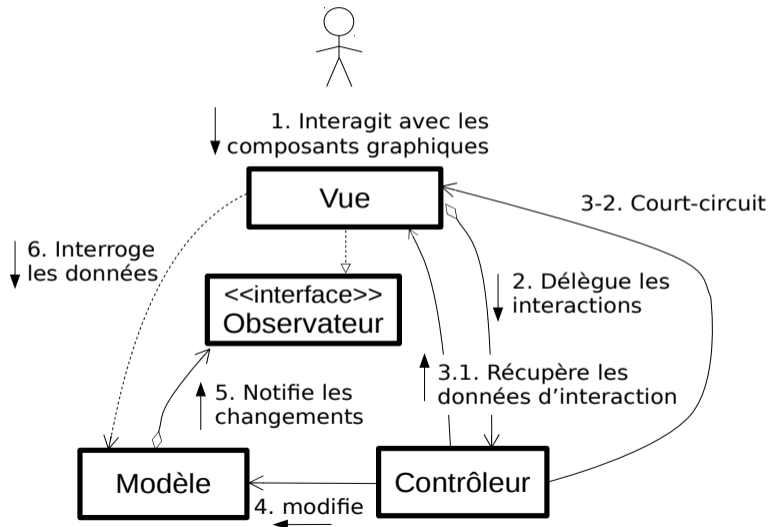
Désavantage :

- pas de réel contrôle sur l'ordre de notification des observateurs.

Fonctionnement de MVC

- **Modèle** : noyau de l'application
 - ▶ Enregistre les vues et les contrôleurs qui en dépendent
 - ▶ Notifie les composants dépendants des modifications aux données
- **Vue** : interface (graphique) de l'application
 - ▶ Crée et initialise ses contrôleurs
 - ▶ Affiche les informations destinées aux utilisateurs
 - ▶ Implante les procédures de mises à jour nécessaires pour demeurer cohérente
 - ▶ Consulte les données du modèle
- **Contrôleur** : partie de l'application qui prend les décisions
 - ▶ Accepte les événements correspondant aux entrées de l'utilisateur
 - ▶ Traduit un événement en demande de service adressée au modèle ou bien en demande d'affichage adressée à la vue
 - ▶ Implémente les procédures indirectes de mise à jour des vues si nécessaire

Diagramme MVC



Avantage :

- facile d'ajouter une vue (extension facile)

Désavantages :

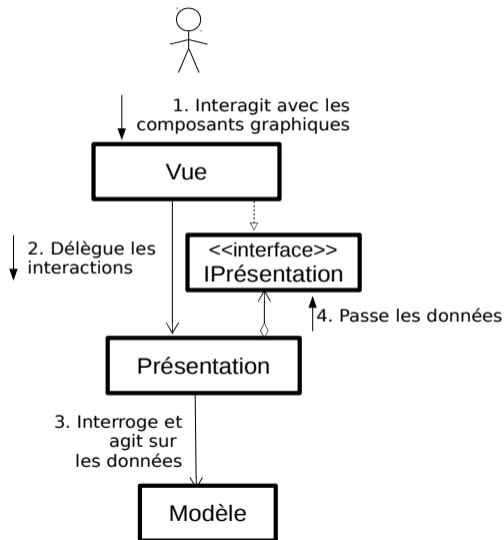
- état interface utilisateur présente dans le modèle
- la logique de présentation est difficile à tester, car fortement lié à des affichages
- la séparation des responsabilités n'est pas totalement claire : la vue est responsable de l'affichage et de la récupération des données (2 raisons de changer)

Objectif Modèle-Vue-Présentation

Réduire le couplage entre la vue et le modèle du MVC en imposant un composant central de présentation

- Toutes les interactions Vue-Modèle passent par la présentation
- La vue est débarrassé de la logique de présentation
- le modèle n'a pas à stocker l'état de l'interface

Diagramme MVP



- Le **Modèle** s'occupe uniquement de la logique métier, ne dépend pas des autres composants
- La **vue** ne s'occupe que de la partie graphique
- La **présentation** contient :
 - ▶ la logique de présentation
 - ▶ l'état courant du dialogue avec l'utilisateur

Avantages :

- la vue ne connaît pas le modèle
- la vue ne contient pas la logique de présentation
- la vue est la seule partie à dépendre du choix de la bibliothèque graphique

Désavantage :

- beaucoup de code redondant pour la présentation : les mises à jour passe par des appels de méthodes comme `displayText1(text)`

Modèle-Vue-Vue Modèle

Similaire à MVP sauf que le retour d'information n'est plus pris en charge par la présentation, mais par un mécanisme de liaison de données (*data binding*).

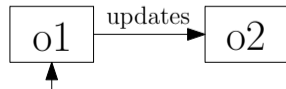
Data binding

Permet de lier l'état de deux objets

⇒ l'état de l'objet lié est changé à chaque changement de l'autre objet.

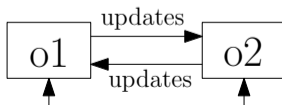
Le *binding* est réalisé par la bibliothèque graphique et n'a donc pas besoin d'être testé.

`o2.bind(o1)`



modifications

`o2.bindBidirectional(o1)`



modifications

modifications

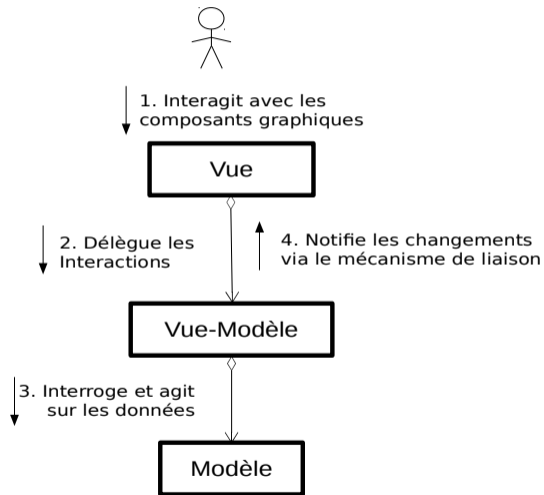
Property de JavaFX : *bind*

```
IntegerProperty num1 = new SimpleIntegerProperty(0);  
IntegerProperty num2 = new SimpleIntegerProperty(0);  
num2.bind(num1); // lie num2 à l'état de num1  
System.out.println(num2.getValue()); // affiche 0  
num1.setValue(2);  
System.out.println(num2.getValue()); // affiche 2
```

Property de JavaFX : *bidirectional bind*

```
IntegerProperty num1 = new SimpleIntegerProperty(0);
IntegerProperty num2 = new SimpleIntegerProperty(0);
num2.bindBidirectional(num1); // lie num2 à l'état de num1 et inversement
System.out.println(num2.getValue()); // affiche 0
num1.setValue(2);
System.out.println(num2.getValue()); // affiche 2
num2.setValue(4);
System.out.println(num1.getValue()); // affiche 4
```

Diagramme MVVM



Les composants ont les mêmes rôles que dans MVP

- Le **Modèle** s'occupe uniquement de la logique métier, ne dépend pas des autres composants
- La **vue** ne s'occupe que de la partie graphique
- La **vue-modèle** contient :
 - ▶ la logique de présentation
 - ▶ l'état courant du dialogue avec l'utilisateur

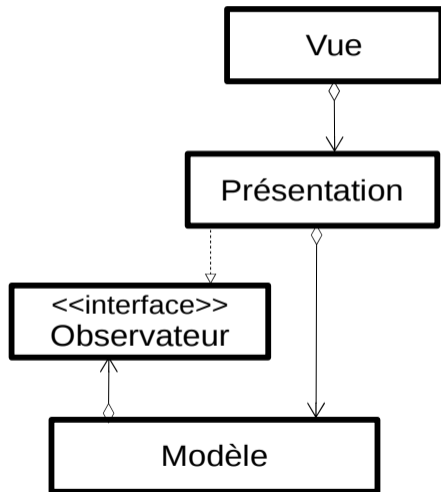
Avantages :

- globalement les mêmes que pour MVP
- moins de code de mise à jour de la vue, car les *bind* simplifient le code

Désavantages :

- Peu avantageux si la bibliothèque graphique ne prend pas en charge les mécanismes de *bind*
- Parfois on a besoin d'un contrôle très précis sur l'ordre des mises qui empêche l'utilisation de *bind*
- Le coût des *bind* pour une grosse application peut être trop important

Modèle actif dans MVP



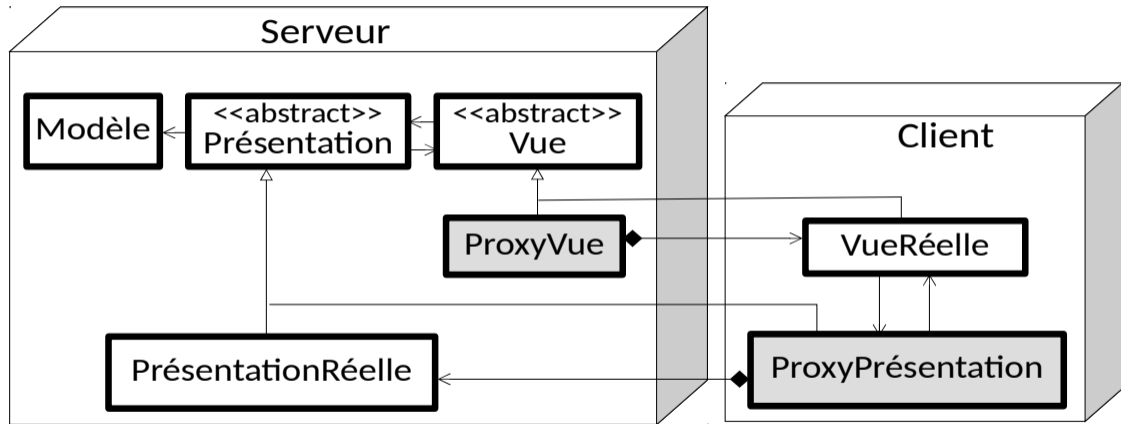
Dans certains cas le modèle peut être *actif*

Modèle actif

Modèle modifié par d'autres composants que la présentation.

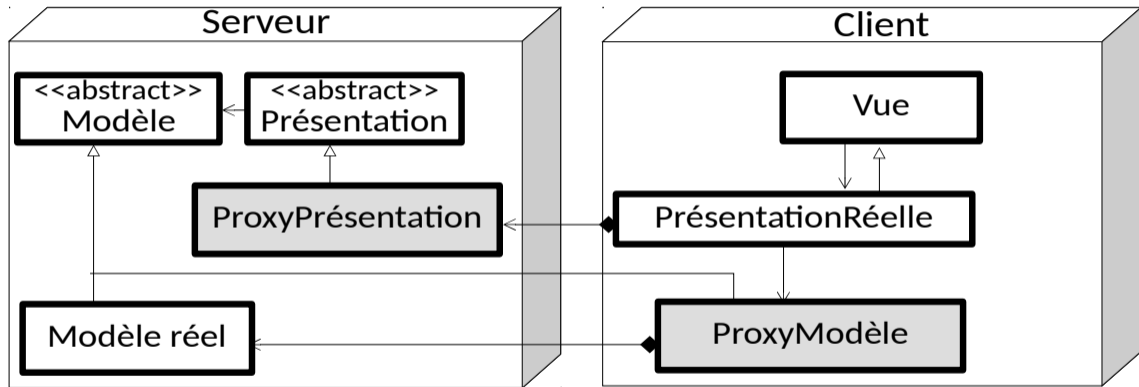
⇒ on doit mettre un mécanisme d'écoute (patron de conception *observer*) entre la présentation et le modèle.

Architecture MVP répartie en client-serveur



Le client ne contient que la vue (client léger)

Architecture MVP répartie en client-serveur



Présentation locale au client (client lourd)