Génie Logiciel Avancé : architecture logicielle

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

12 février 2025

amU Faculté des sciences Aix Marseille Université

Section 1

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 ;
- Micro-services ;
- Modèle-Vue-Contrôleur et variantes ;
- ...

Section 2

Architecture en couches

Architectures en couches

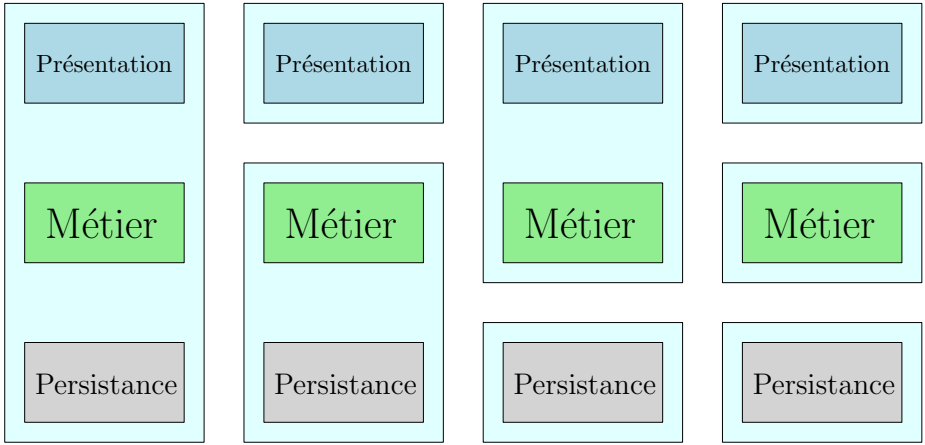
Découper en couches est une des techniques les plus utilisées en développement logiciel

- 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

Trois couches les plus communes

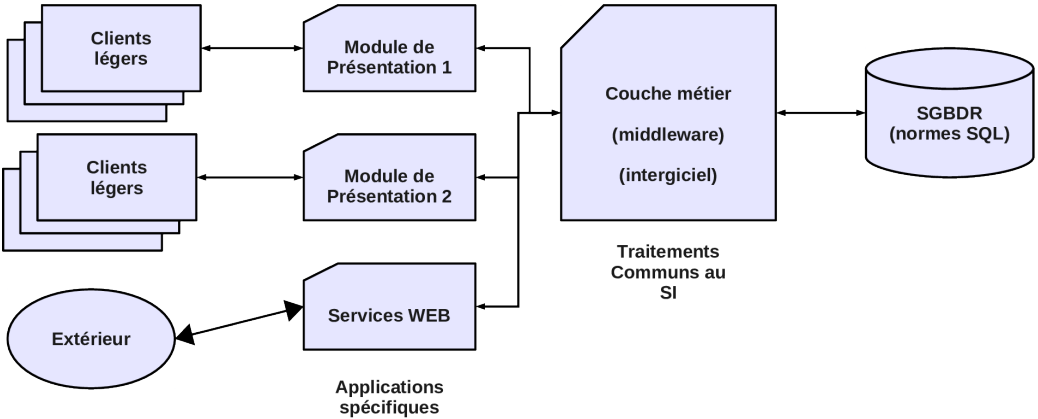
- **Présentation/IHM+affichage** :
  - traite les interactions entre utilisateur et logiciel,
  - affiche les données et transforme les commandes de l'utilisateur en actions sur les autres couches.
- **Métier/Domaine** :
  - cœur de l'application,
  - valide les données issues de la couche présentation,
  - effectue les traitements à partir des entrées venant de la couche présentation et des données issues de la couche persistance.
- **Persistance/Données** :
  - gère les communications avec les bases de données et/ou les autres services,
  - responsable du stockage persistant des données,
  - permet de découpler le code de l'application de la solution de stockage choisie.

Différentes façons de répartir les couches



Architecture 3-tiers/3-couches

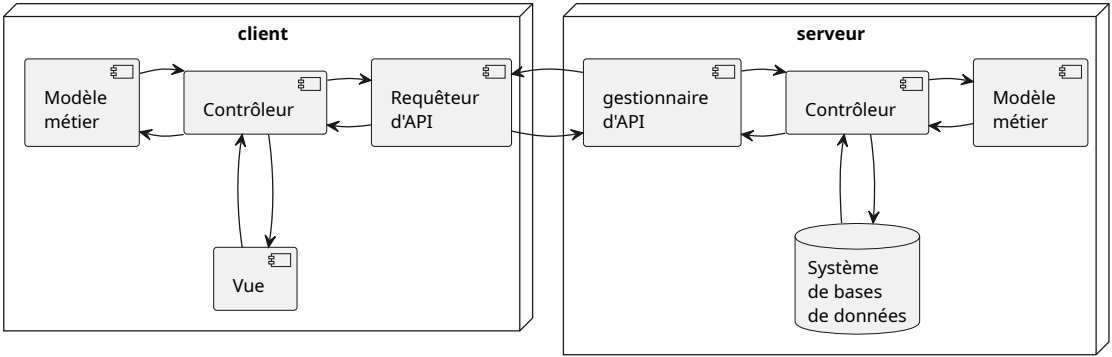
Principe : Séparation claire entre les trois couches



Les bonnes pratiques pour une architecture en couches

- Limiter les dépendances (choisir un sens unique avec l'inversion de dépendance) ;
- Limiter les couplages en utilisant des DTO pour communiquer des données entre couches ;
- Pour vérifier si un traitement appartient à la couche métier, vérifier qu'un changement important dans une des autres couches n'entraine pas de duplication de code :
  - Passage d'une interface web à une en ligne de commande ;
  - Passage d'un stockage en SQL à un utilisant XML.

Architecture du projet



- La partie métier est présente dans le client et le serveur ;
- Le contrôleur permet de court-circuiter le modèle pour des accès direct aux données.

Exemple plus complexe : analyse financière

On cherche à développer une application permettant :

- de traiter des données financières pour générer des données d'analyse ;
- d'utiliser ces données pour générer des rapports :
  - sous forme de pdf
  - sous forme de page web

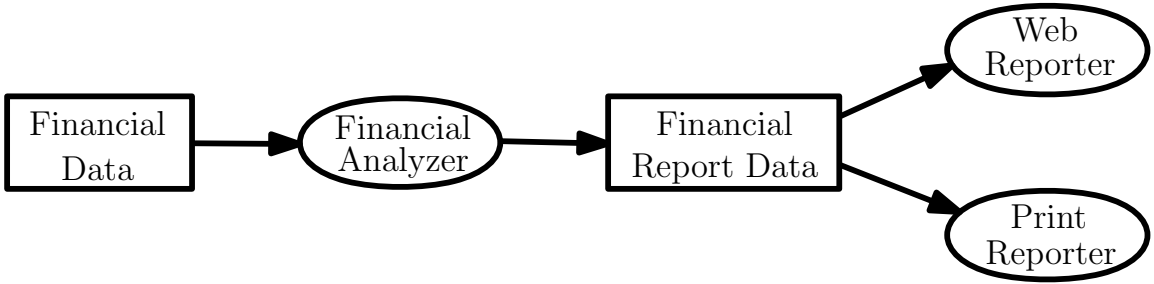


Diagramme de composants

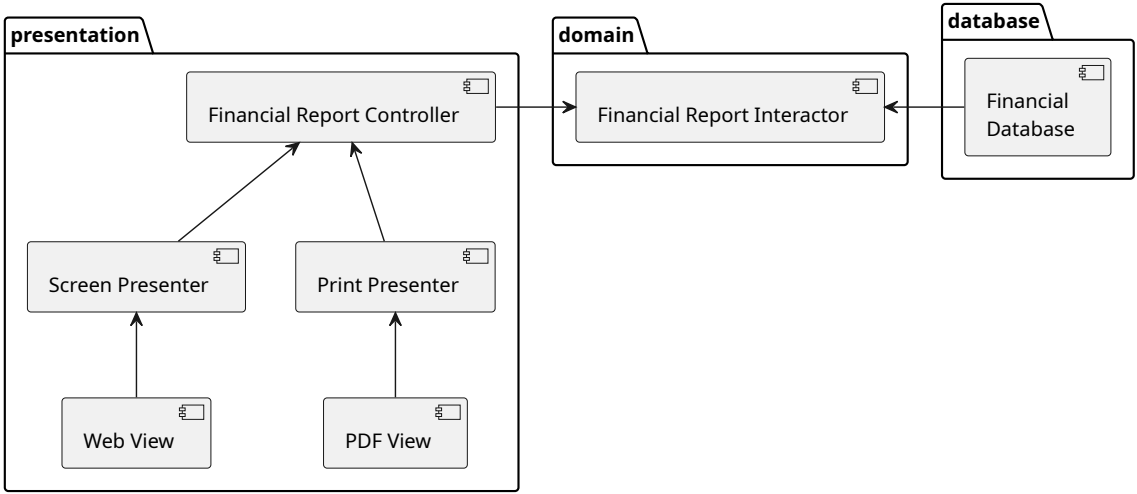
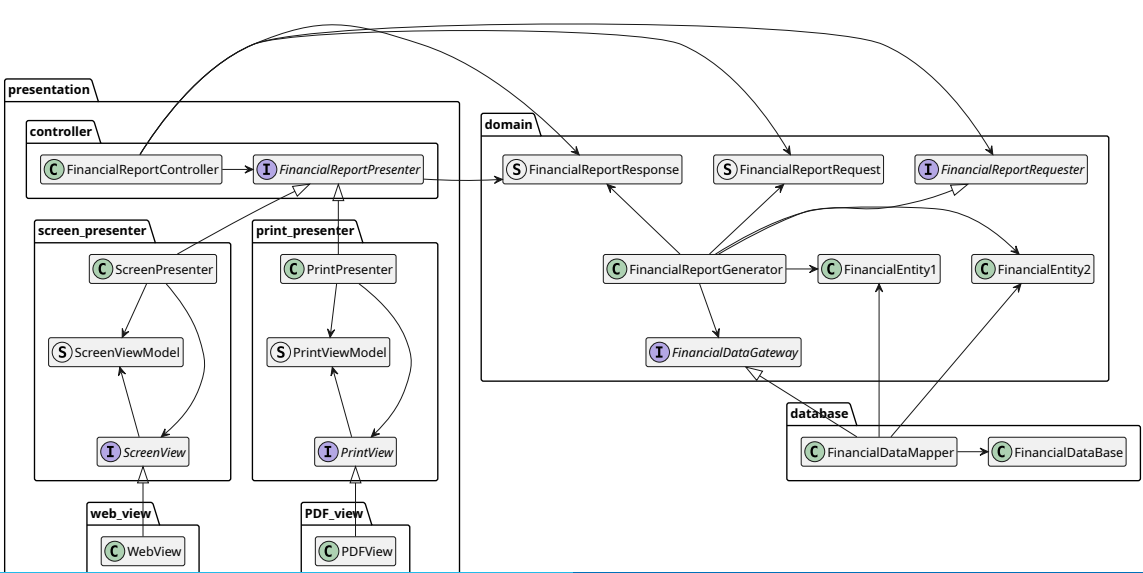
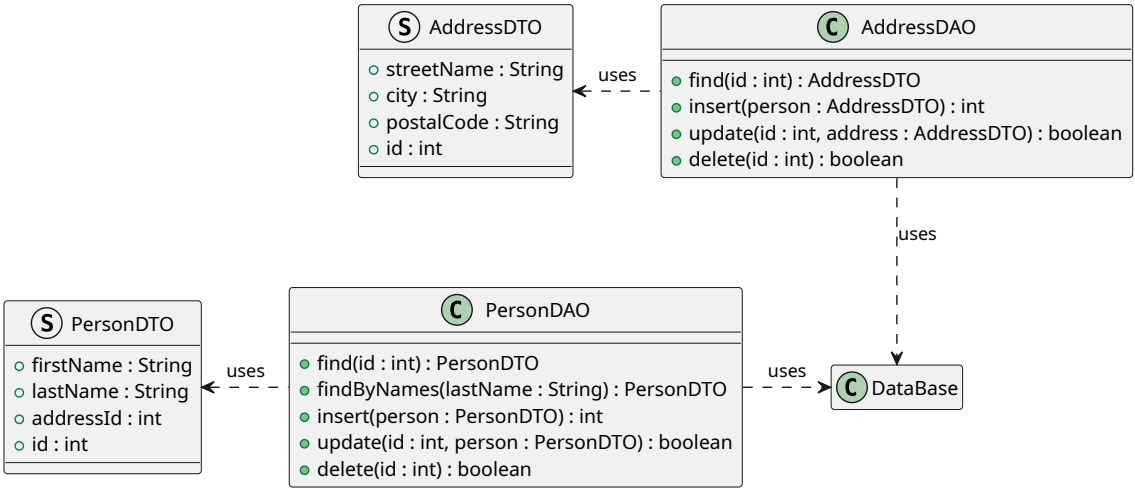


Diagramme de classes

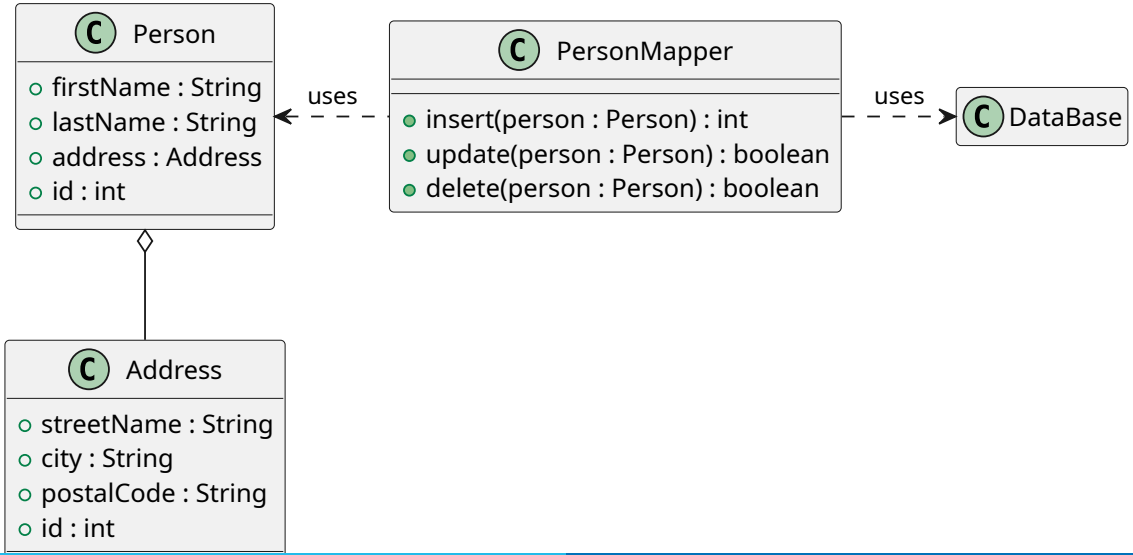


Patron DAO pour gérer le lien entre objets et stockage



Patron simple et facile à utiliser : adapté pour les domaines simples

Patron Data Mapper pour gérer le lien entre objets et stockage



Automatisation via la Jakarta Persistence API (JPA)

Entité

Classes dont les instances peuvent être stockées de manière persistante dans une base de données.

Version Java des entités :

- Interface définie via JPA ;
- Implémentation via un ORM comme *hibernate* ;
- Entité : objet java simple avec getters/setters et constructeurs ;
- Utilisation d'annotations.

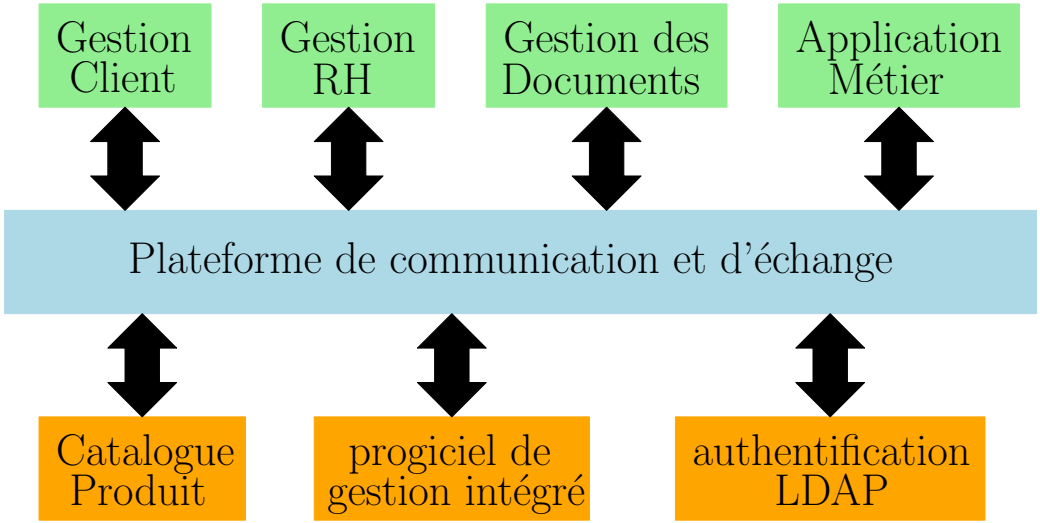
Exemple entité de personne

```
public class Person {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private long id;
  private String lastName;
  private String firstName;
  @Embedded
  private Address address;
  @Version
  private long version = 0;
  // constructeur, getters, setters, ...
}
```

Section 3

Architectures orientées services et micro-services

Architecture orienté service SOA (1995)



Architecture orienté service SOA (1995)

Objectif d'une architecture orientée service (SOA)

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

Avantages :

- couplages faibles entre services : les services sont connectés aux clients et aux autres services via des standards ;
- interopérabilité augmentée, car les échanges se font via des standards très répandus (API REST, XML, JSON, ...) ;
- mécanisme de publication des services disponibles afin de les incorporer facilement (annuaire de service).

Point clés à considérer pour une SOA

- Gestion d'un annuaire de services : quels sont les services mis à disposition et par qui ?
- Quels sont les dépendances entre services ?
- Description des contrats des services :
  - quelles sont les données nécessaires à l'appel du service, que fournit-il en retour ?
  - Quel est le format des données échangées : XML, Json ?
- Gestion de la sécurité des communication avec : SSL, Jeton d'authentification, ...
- Orchestration : outils pour automatiser la configuration, le déploiement, la gestion des services.

Avantages et inconvénient d'une SOA

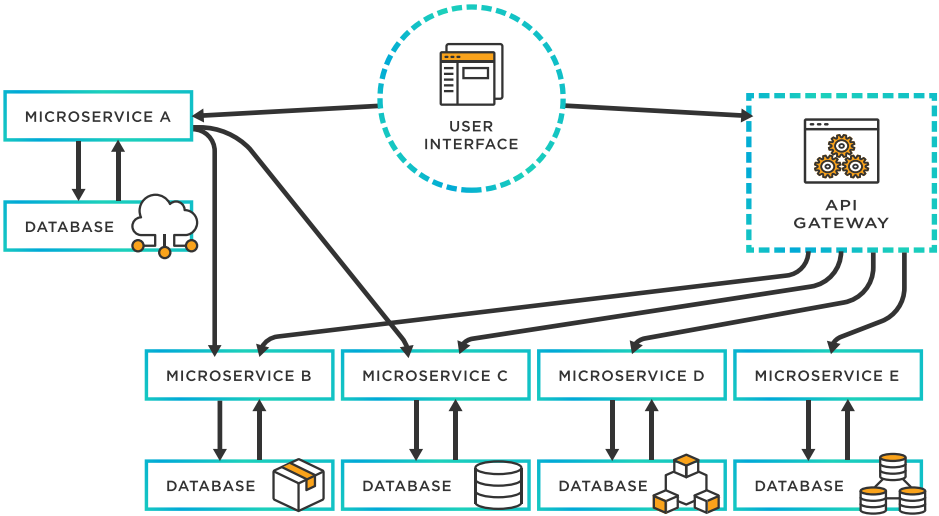
Avantages:

- Plus flexible et plus facile de passer à l'échelle qu'une approche par couche ;
- Plus facile d'intégrer des systèmes existants comme des services.

Inconvénients

- Complexe à mettre en place et à configurer ;
- Gouvernance et gestion des services difficile ;
- Problèmes de latences quand de nombreux services sont sollicités pour une requête.

Architecture en microservices (2005)



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

Propriétés souhaitables d'un micro-service

- Elastique : doit être capable de s'adapter facilement à la charge (réplication facile)
- Tolérant aux pannes : une panne sur un micro-service ne doit impacter que le service
- Composable : son interface doit permettre de facilement mettre en place des proxies, caches, ...
- Minimal : difficile à découper en deux services
- Complet : doit fournir un service complet

Différences entre Microservices et SOA

Caractéristique	SOA	Microservices
Taille des services	Services de grande taille	Services plus petits
Communication entre services	Via le bus entreprise	Via API REST
Couplage entre services	Plus fort	Plus faible
Déploiement des services	Centralisée	Indépendant

Avantages et inconvénients des micro-services

Avantages :

- Évolutivité et flexibilité
- Développement et déploiement rapide
- Indépendance des équipes
- Optimisation des performances

Inconvénients :

- Complexité accrue
- Gestion des données distribuées
- Monitoring et debugging difficiles
- Problèmes de latence réseau
- Sécurité et gestion des accès complexe

Section 4

Patrons pour interface graphiques GUI : MVC, MVP, MVVM

Introduction

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)

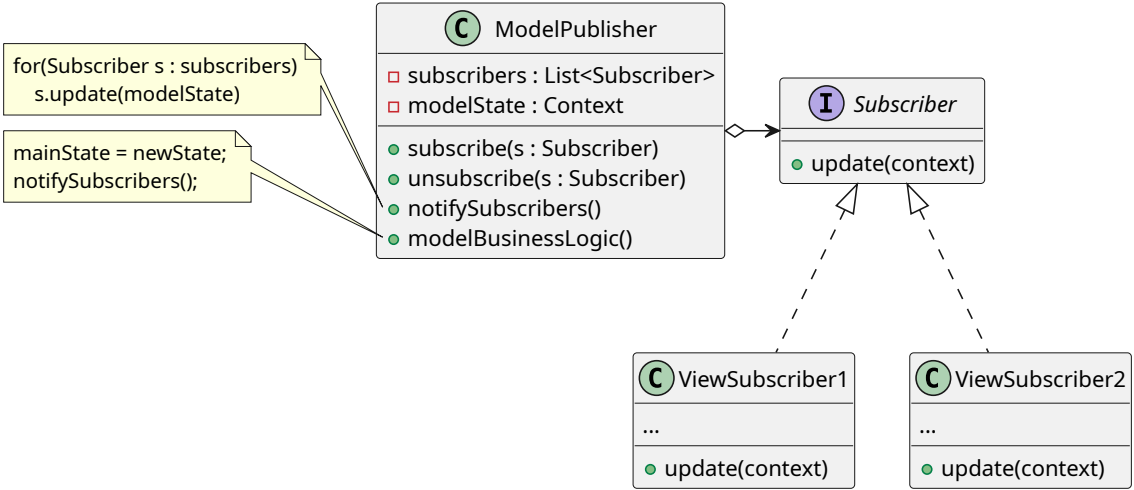
Objectif

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 traitent, 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*



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.

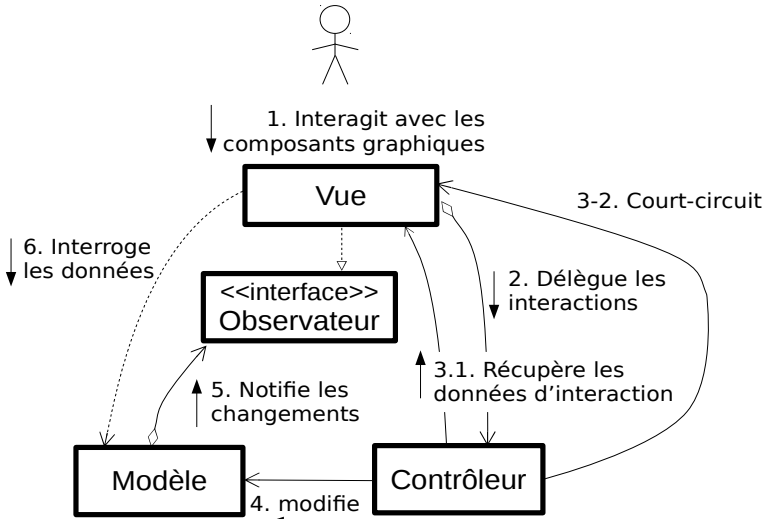
Désavantage :

- 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



Analyse 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)

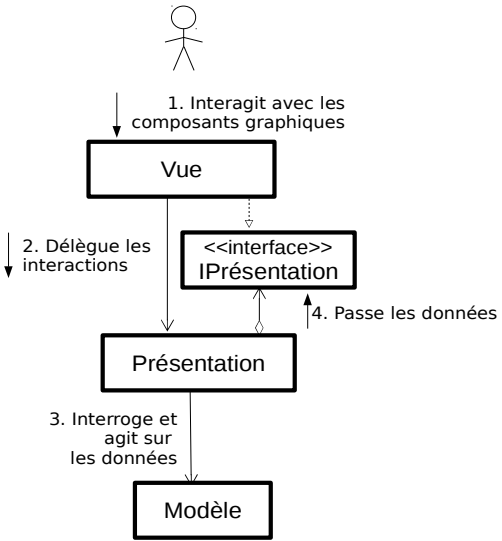
Patron d'architecture MVP

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 stoker 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

Analyse MVP

Avantages :

- la vue ne connait 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

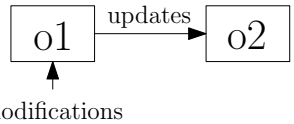
Similaire à MVP sauf que le retour d'information n'est plus pris en charge par la présentation, mais par un mécanisme de liaison de données (*data binding*).

Data binding

Permet de lier l'état de deux objets
$\Rightarrow$ 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)



o2.bindBidirectional(o1)

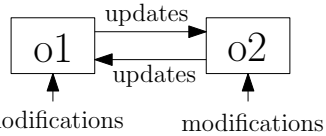
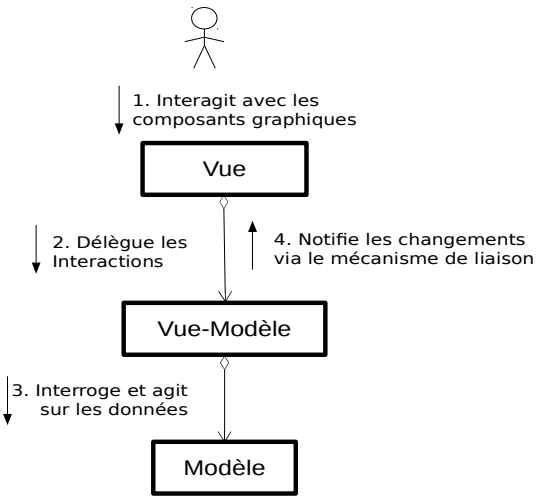


Diagramme MVVM



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

Analyse MVVM

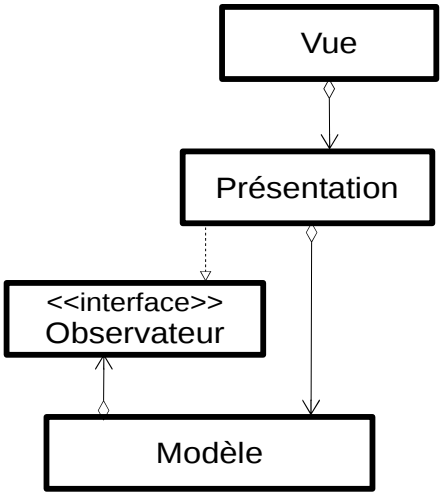
Avantages :

- 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



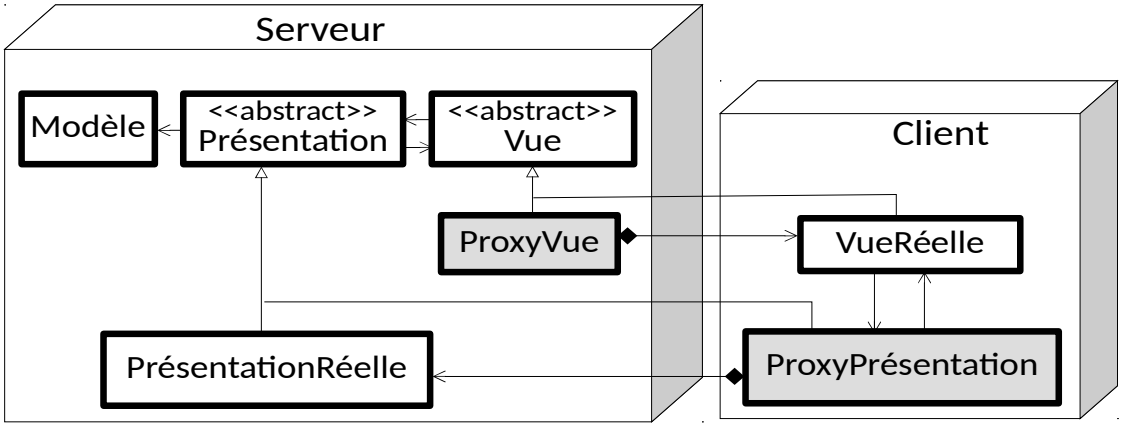
Dans certains cas le modèle peut être *actif*

Modèle actif

Modèle modifié par d'autres composant que la présentation.

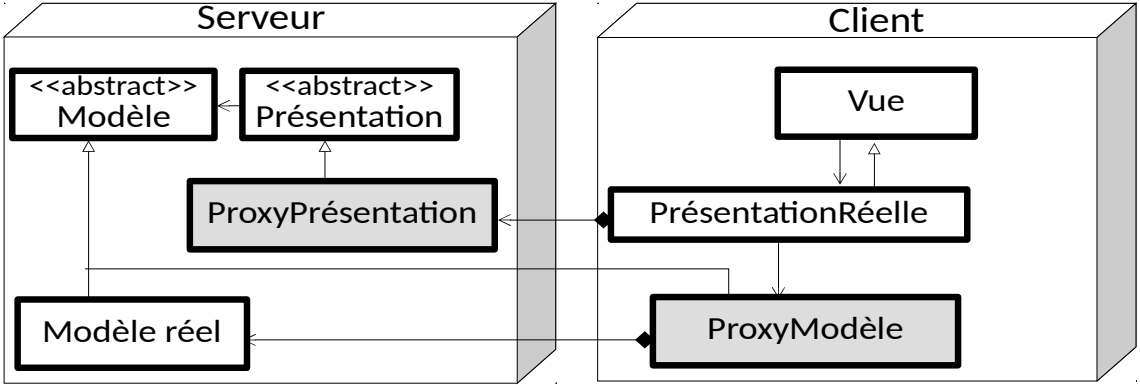
$\Rightarrow$ on doit mettre un mécanisme d'écoute (patron de conception *observer*) entre la présentation et le modèle.

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)