Génie logiciel : architecture logicielle (2/2) : interface graphique

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

10 octobre 2025

amU Faculté des sciences Aix Marseille Université

Section 1

Patrons (MVC, MVP et MVVM) pour interface graphique

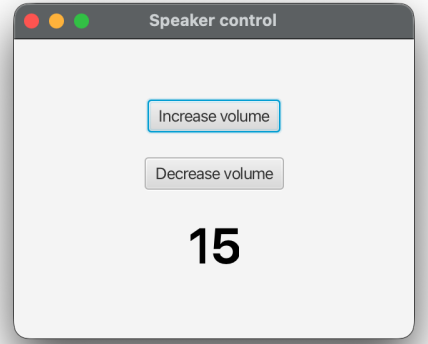
Introduction

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.

Exemple d'application



Gestion de volume

- un bouton pour augmenter le volume
- un bouton pour diminuer
- un champ texte affichant le volume courant

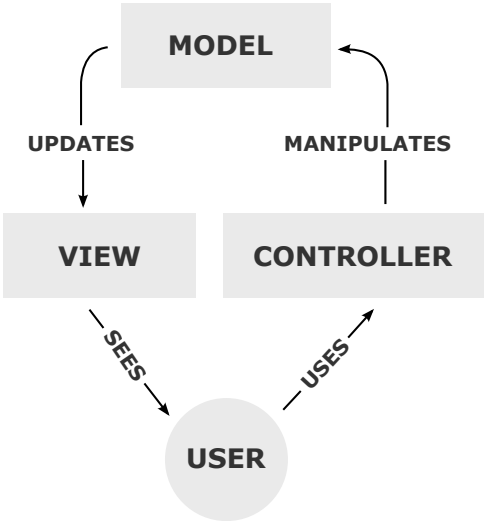
Questions

- Comment découper les responsabilités entre classes ?
- Comment rendre le maximum de code testable ?

Section 2

Première solution : Modèle Vue Contrôleur (MVC)

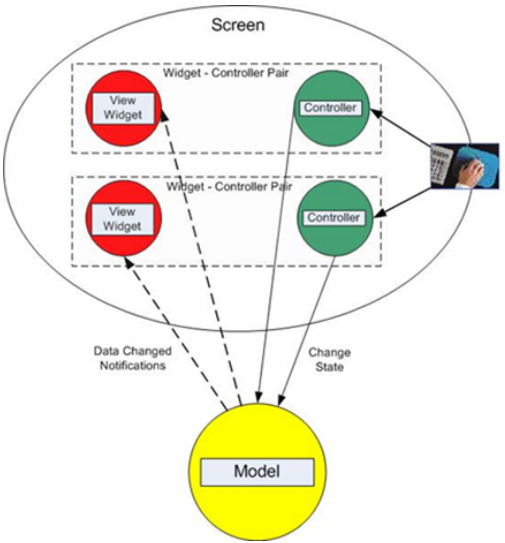
Diagramme MVC : première approche



Découpage en 3 parties

- **Modèle** : rassemble des données du domaine/système. Contient les classes dont les instances doivent être vues et manipulées
- **Vue** : utilisé pour présenter/afficher les données du modèle dans l'interface utilisateur
- **Contrôleur** : contient les fonctionnalités nécessaires pour gérer et contrôler les interactions de l'utilisateur avec la vue et le modèle

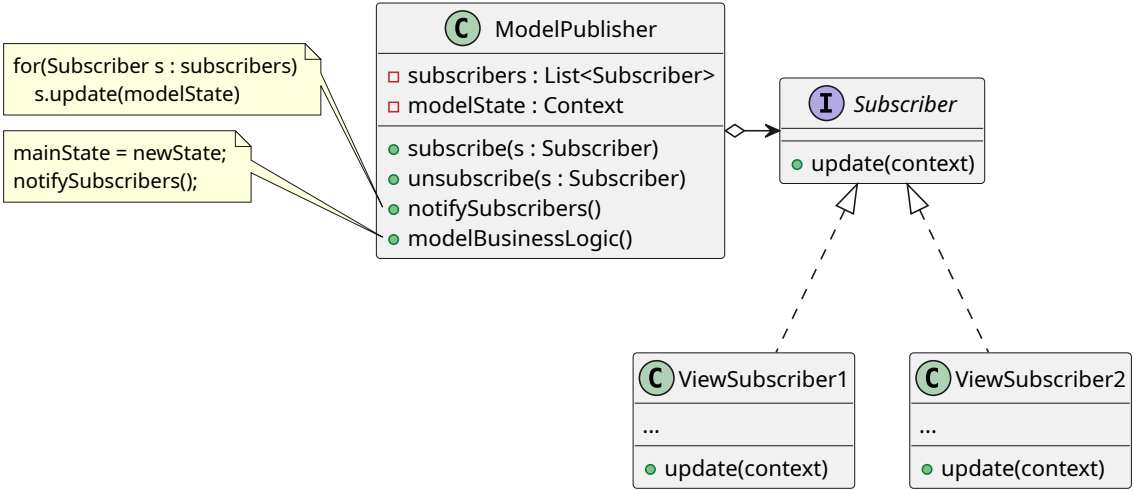
MVC classique: Trygve Reenskaug (1979)



Fonctionnement

- Une paire *widget-controller* par élément.
- Couplage fort entre les deux éléments de la paire.
- Les contrôleurs ont une référence vers une instance correspondant au modèle.
- Les vues s'abonnent aux modifications du modèle via 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.

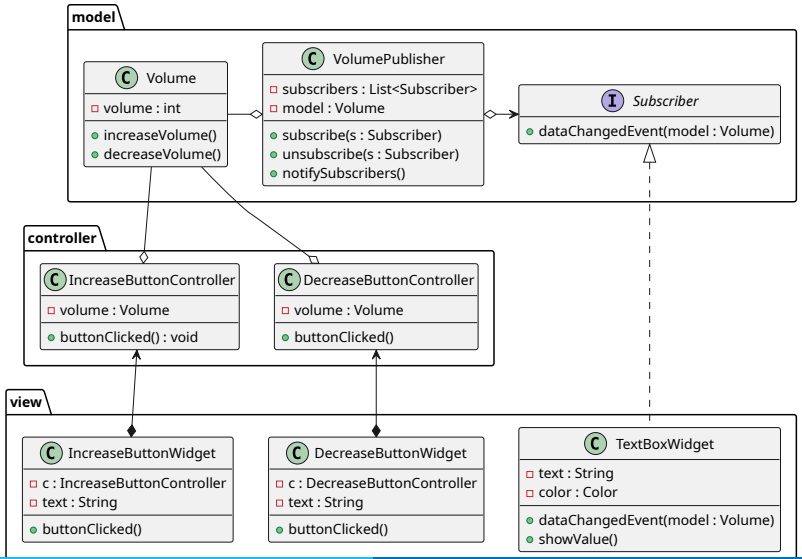
Avantages :

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

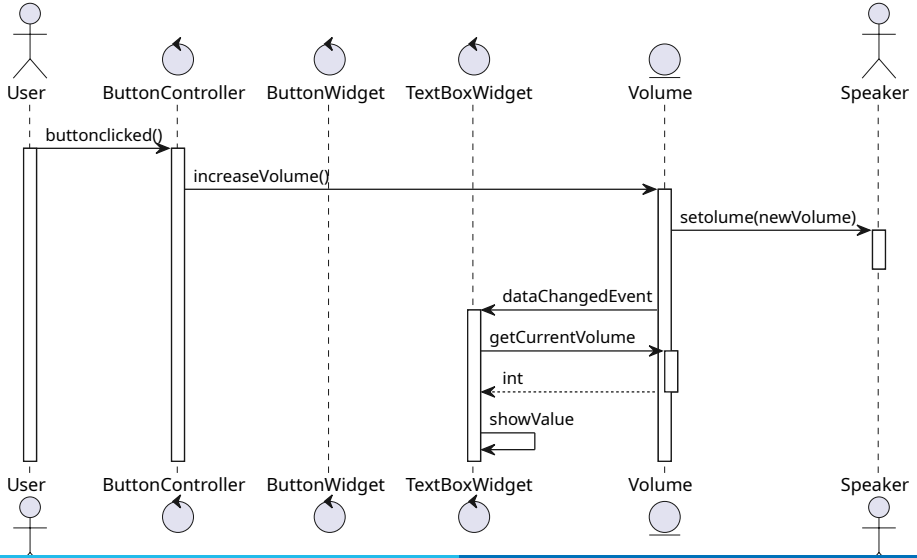
Désavantage :

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

Organisation en classe dans l'exemple



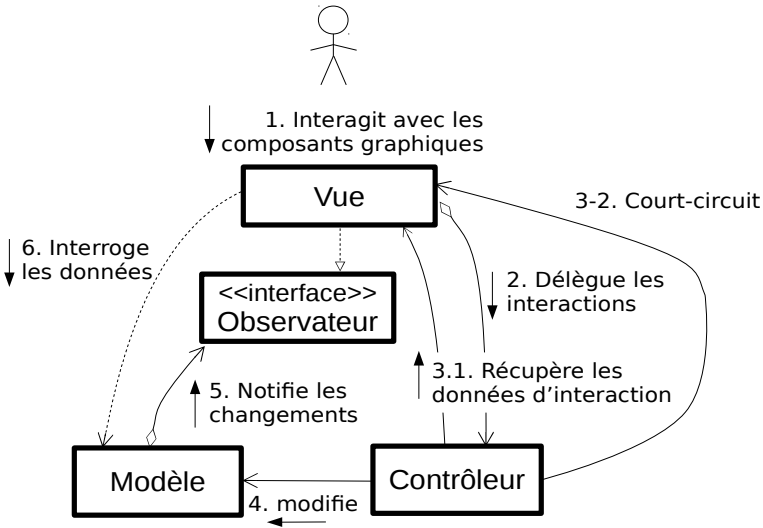
Interactions en MVC classique



Découpage MVC

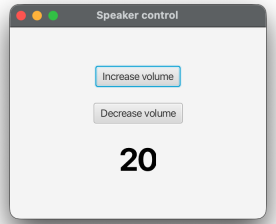
- le **modèle** contient la logique métier et l'état courant de l'interface durant le cycle dialogue avec l'utilisateur (de la simple valeur à un service avec persistance des données).
  - Enregistre les vues et les contrôleurs qui en dépendent
  - Notifie les composants dépendants des modifications aux données
- La **vue** (fenêtre, bouton, champ texte, ...) :
  - porte toute la logique de présentation ;
  - délègue la gestion des interactions avec les utilisateurs au contrôleur ;
  - Consulte les données du modèle
- Le **contrôleur** : partie de l'application qui prend les décisions
  - reçoit les interactions de l'utilisateur de la vue et les traite en modifiant le modèle ;
  - modifie directement la vue lorsque l'interaction ne change que la vue et pas le modèle (court-circuit)

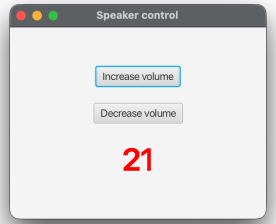
Diagramme MVC “classique”



Gestion de la logique de présentation

On souhaite que le texte du volume change de couleur s'il est > 20.





Solutions

- laisser la vue gérer $\Rightarrow$ difficile de tester le comportement
- mettre dans le modèle $\Rightarrow$ dépendance du modèle à une logique de présentation

Analyse MVC

Avantage :

- facile d'ajouter des vues (extension facile) car le patron de conception observateur isole les vues du modèle

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)
- lourdeur de la boucle d'action, car on passe par le modèle pour les changements

Solution

Casser le cycle en passant au MVP et simplifier le rôle de la vue.

Section 3

Deuxième solution : Modèle Vue Présentateur (MVP)

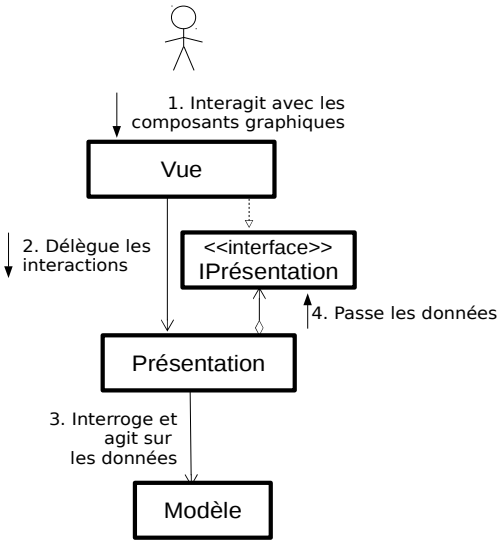
Patron d'architecture MVP

Objectif Modèle-Vue-Présentation

Réduire le couplage entre la vue et le modèle du MVC en imposant un composant central de présentation
$\Rightarrow$ mise en place d'une architecture 3-tiers

- Le contrôleur devient un Présentateur, car il devient le l'endroit de définition de la logique de présentation
- Toutes les interactions entre la Vue et le 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



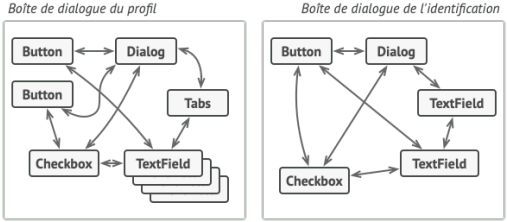
- Le **Modèle** s'occupe uniquement de la logique métier, ne dépend pas des autres composants
- La **vue** ne s'occupe que de la partie graphique
- La **présentation** contient :
  - la logique de présentation
  - l'état courant du dialogue avec l'utilisateur
  - est indépendante de la bibliothèque graphique
  - implémente le patron de conception *mediator*

Patron de conception *mediator*

Exemple d'application

Formulaire de dialogue

Selon les boutons cochés, le formulaire contient plus ou moins de champs texte



Problème à régler

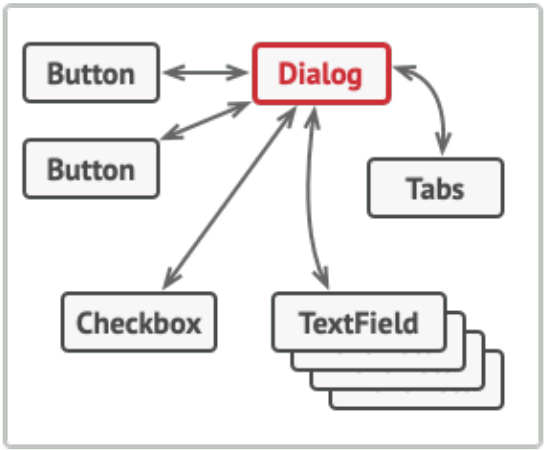
Dépendances entre les éléments de l'interface utilisateur pouvant devenir chaotiques avec l'évolution de l'application.

Solution

Introduire une classe `Mediator`

Solution : créer une classe servant de médiateur

Boîte de dialogue du profil



Boîte de dialogue de l'identification

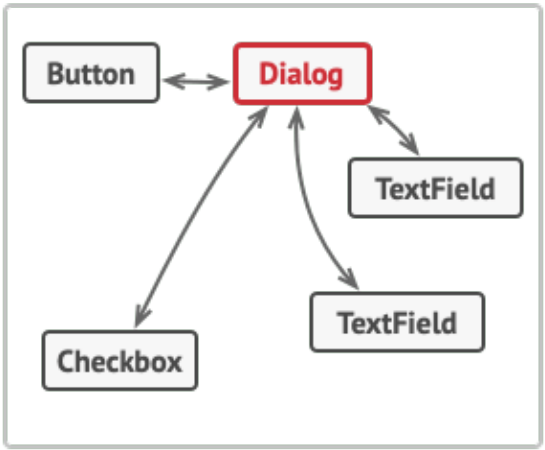
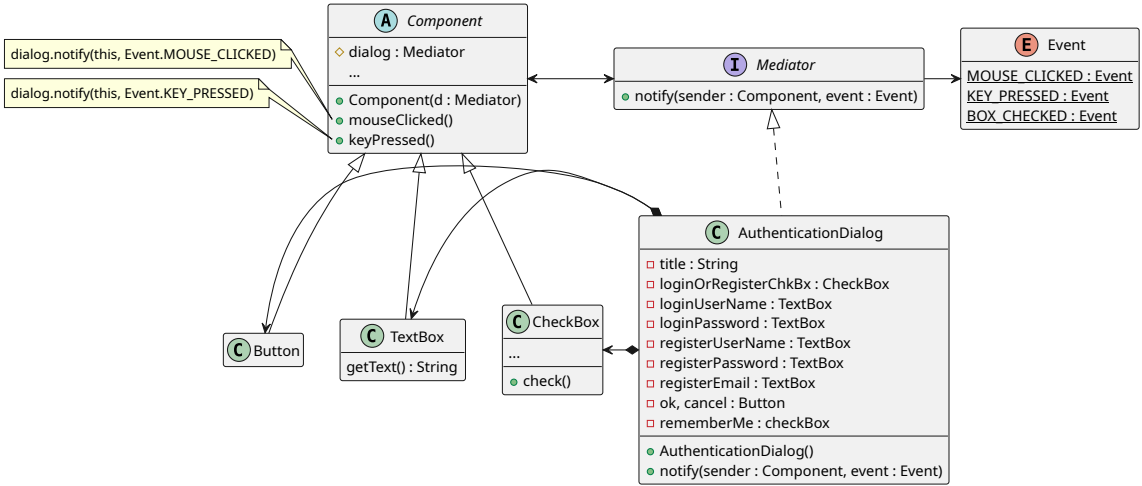


Diagramme de classe `Mediator`



Patron de conception *mediator*

Intention

Restreindre les communications directes entre les objets et les force à collaborer uniquement via un objet médiateur.

Avantages :

- Isoler la communication entre composant dans une classe à part (SRP).
- Permet de mettre en place de nouvelles relations entre composants en créant d'autres classes médiateurs (OCP).
- Diminution du couplage entre composants.

Désavantages :

- Avec l'évolution du code, un médiateur peut évoluer en objet omniscient (*God object*).

Retour sur MVP

Différences entre MVC et MVP :

- la vue n'écoute pas les modifications du modèle via le patron de conception *observer* : les mises à jour passent par `IPresentation` (application de DIP)
- le cycle de dialogue est centré sur la présentation : elle met à jour le modèle et répercute les effets sur les vues
- la logique de présentation est sortie de la vue et peut donc être plus facilement réutilisable

Analyse MVP

Avantages :

- la vue ne connait pas le modèle
- la vue ne contient pas la logique de présentation (testabilité améliorée, car on peut tester 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 `displayText(text)` ou `setButton1Enabled(true)`

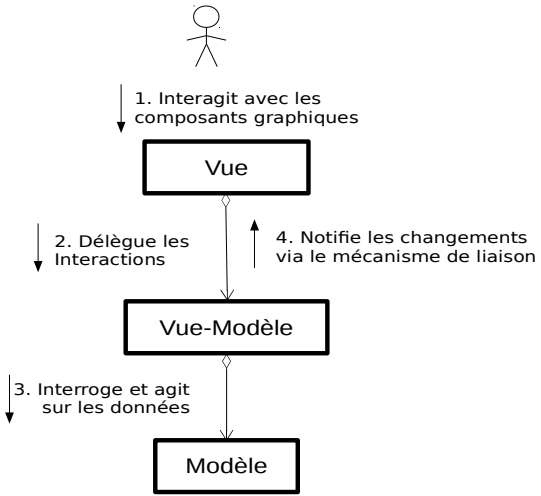
Solution

Passer au MVVM

Section 4

Troisième solution : Modèle Vue Vue-Modèle (MVVM)

Diagramme MVVM



Comme MVP sauf que le retour d'information passe par un mécanisme de liaison de données (*data binding*).
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

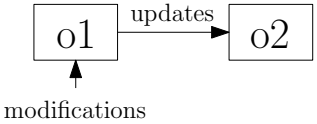
Data binding

Définition : *data binding*

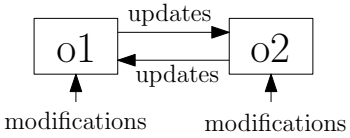
Permet de lier l'état de deux objets
⇒ l'état de l'objet lié est changé à chaque changement de l'autre objet.

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

o2.bind(o1)



o2.bindBidirectional(o1)



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 : *biderectional bind*

```
IntegerProperty num1 = new SimpleIntegerProperty(0);
IntegerProperty num2 = new SimpleIntegerProperty(0);

// lie num2 à l'état de num1 et inversement
num2.bindBidirectional(num1);
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
```

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 à jour qui empêche l'utilisation de *bind*
- Le coût des *bind* pour une grosse application peut être trop important

Section 5

MV* et architecture client-serveur

Évolution des architectures

Architecture monolithique :

Serveurs de données :

Architecture client-serveur :

Architecture 3-tiers :

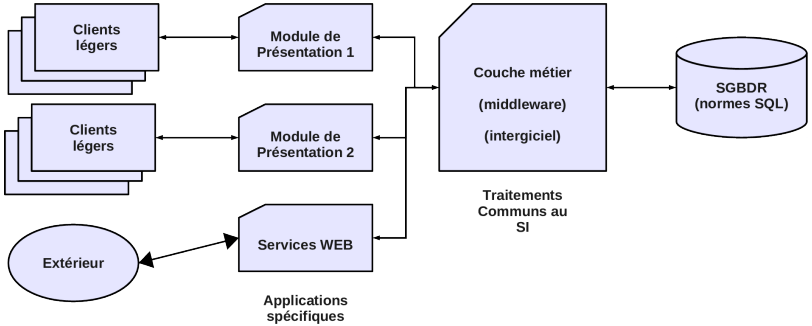
Architecture multi-couches (1990)

- 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).



Modèle actif

Pour le moment, on a considéré que des modèles passifs.

Modèle passif

Le modèle ne peut être changé que par l'intermédiaire de l'interface graphique.

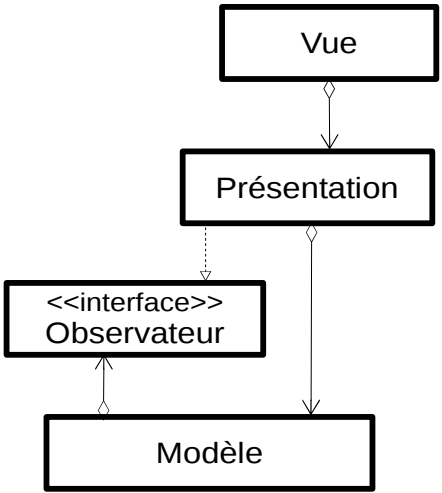
Dans certains cas le modèle peut être *actif*, par exemple s'il dépend de données présentes sur un serveur qui serait modifié par une autre instance de l'application.

Modèle actif

Modèle modifié par d'autres composant que la présentation/vue-modèle.

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

Modèle actif dans MVP



Nécessité de mettre un patron de conception *observer* entre la présentation et le modèle pour :

- MVP
- MVVM

Pas nécessaire pour MVC car la vue écoute déjà le modèle via *observer*.

Choix d'architecture MVP répartie

Question

Comment répartir les parties d'une application graphique entre client et serveur ?

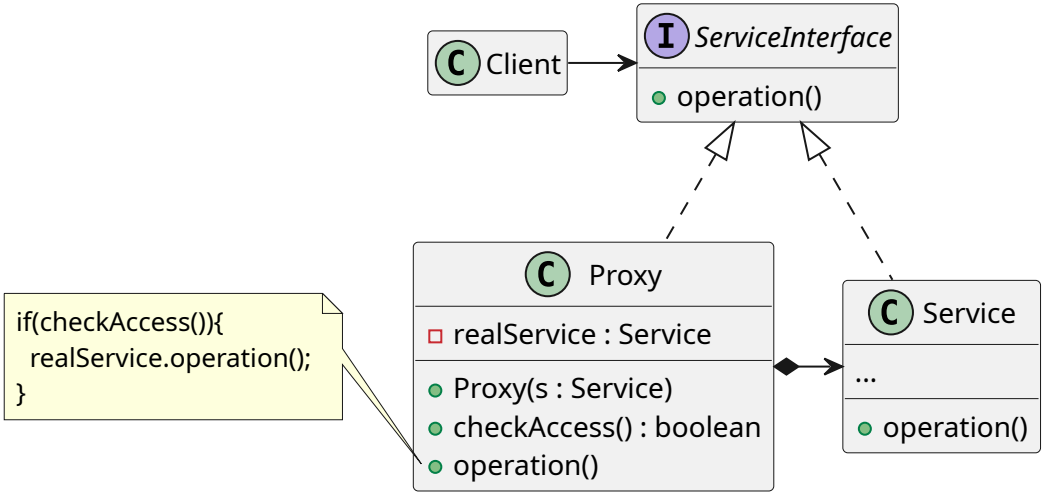
Dans tous les cas :

- le **modèle** est au niveau du serveur.
- la **vue** est au niveau du client.

Deux choix pour la partie présentation

- **client léger** : présentation sur le serveur (le client n'a qu'un *proxy* de la présentation)
- **client lourd** : présentation sur le client (le client a un *proxy* du modèle)

Patron de conception *proxy*



Patron de conception *proxy*

Intention

Permet d'utiliser un substitut pour un objet donnant le contrôle sur l'objet original.

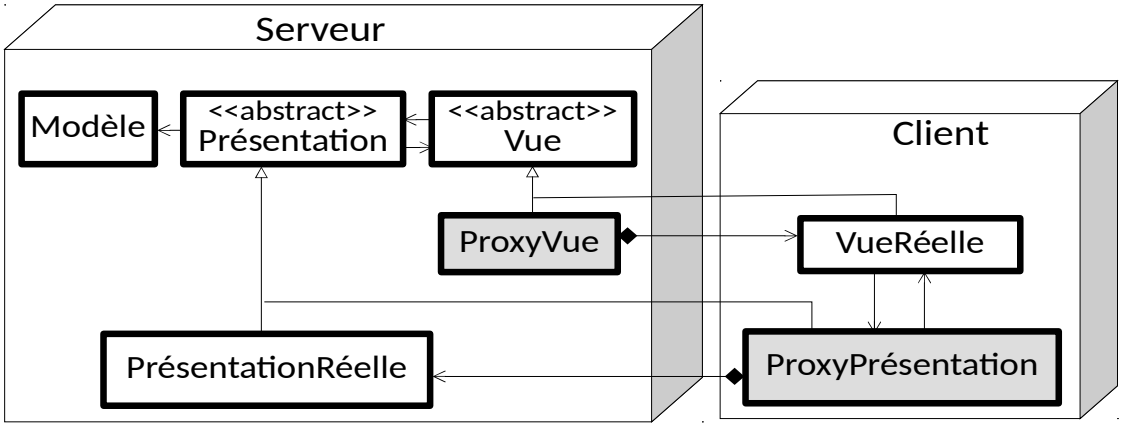
Avantages :

- Contrôler l'objet du service sans que le client ne s'en aperçoive.
- Ajout possible de procuration sans toucher au service (OCP)
- Gestion du cycle de vie de l'objet du service originel

Désavantages :

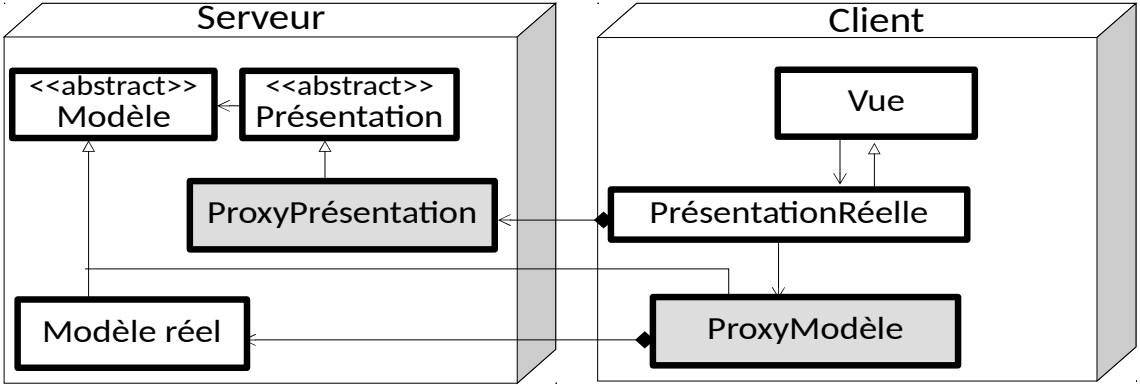
- Code plus complexe, car introduction de classes supplémentaires
- Intermédiaire pouvant ralentir les requêtes.

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)

Section 6

Implémentation des patrons MV* en Java avec JavaFX

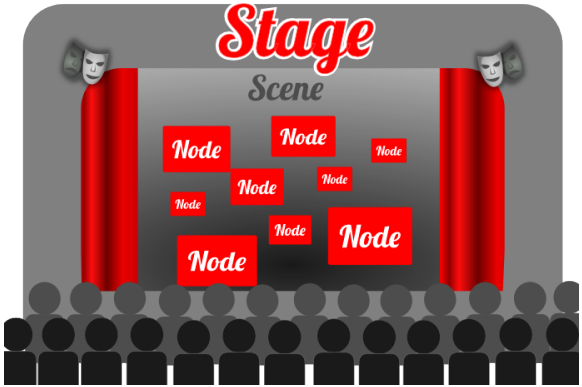
Qu'est-ce qu'est JavaFX ?

JavaFX

framework pour créer des interfaces graphique pour des applications de bureau ou smartphones.

- Première version en 2008
- successeur de `Swing`
- Créé et développé par Sun → Oracle → OpenJFX
- Inclus avec JDK 8 à 10, disponible séparément pour JDK 11+

Éléments de JavaFX : terminologie du théâtre



- `Stage` : estrade correspondant à la fenêtre (système de fenêtre des systèmes d'exploitation)
- `Scene` : décor où a lieu l'action (possibilité d'avoir plusieurs décors pour une même estrade)
- `Node` : nœuds placés dans le décor et donc correspondant aux acteurs

source image : https://mikarber.developpez.com/tutoriels/java/introduction-javafx/

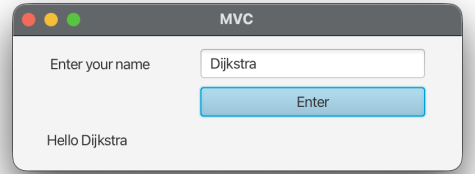
FXML langage pour décrire des `Node`

- FXML est un langage basé sur XML pour décrire des interfaces graphiques JavaFX.
- Possible de créer ces fichiers avec un éditeur de texte mais aussi via un outil graphique qui génère automatiquement le FXML
- Un fichier FXML est au format XML et décrit :
  - la structuration du graphe de scène,
  - les propriétés graphiques de chaque composant (taille, police, couleur, ...)
- Le code FXML est modifiable indépendamment du code métier de l'application
- Le fichier FXML est chargé par l'application (classe `FXMLLoader`) et un objet Java (racine) est créé avec les éléments que le fichier décrit

Avantage principal

Une séparation nette entre le code métier et l'interface graphique utilisateur.

Un exemple simple d'application *login*



Fenêtre de login :

- champ texte pour entrer un nom
- bouton pour valider
- message d'accueil

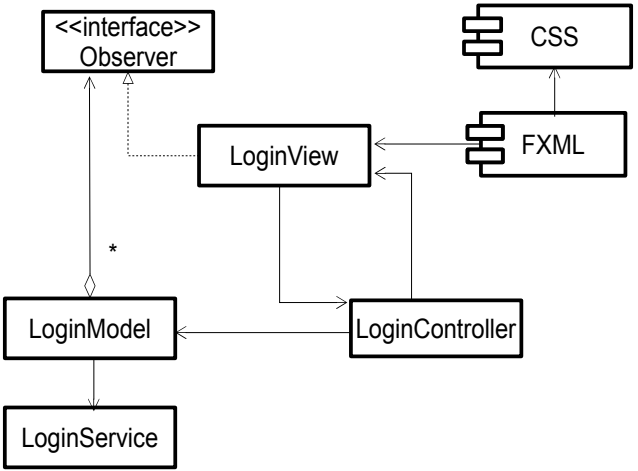
Cas simple :

- une seule vue
- une seule donnée (le nom)

Fichier FXML pour la vue

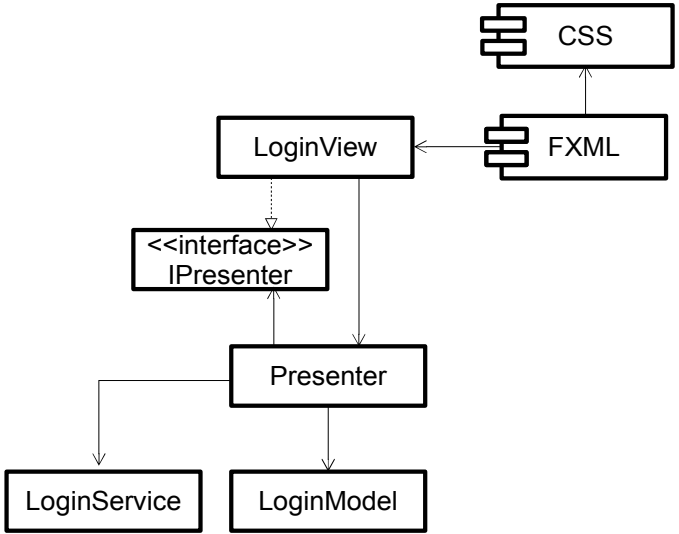
```
<AnchorPane prefHeight="120.0" prefWidth="600.0"
  stylesheets="@loginscreen.css"
  fx:controller="fr.univ_amu.m1info.login.view.LoginView"
  xmlns:fx="http://javafx.com/fxml" >
    <children>
      <Label layoutX="33.0" layoutY="17.0" text="Enter your name"/>
      <TextField fx:id="input" layoutX="167" layoutY="12" />
      <Button defaultButton="true" layoutX="167.0" layoutY="46.0"
              onAction="#handleSayHello" prefWidth="200.0"
              text="Enter"/>
      <Label id="console" fx:id="label" layoutX="30.0" layoutY="80"
        prefHeight="26.0" prefWidth="335.0"/>
    </children>
</AnchorPane>
```

Version MVC



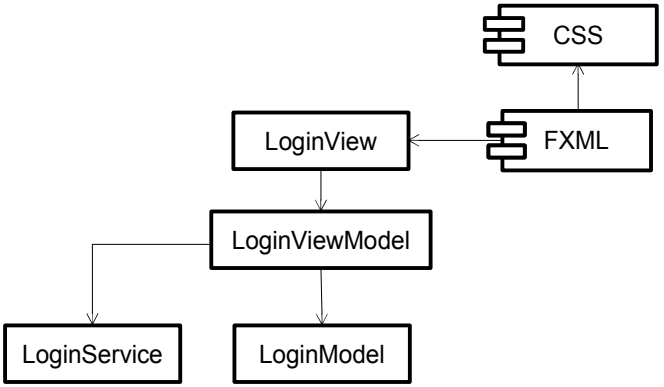
- Modèle :
  - `LoginService` construit le message (même classe pour toutes les versions)
  - `LoginModel` stocke le nom, notifie les *observers* et délègue la construction du message à `LoginService`
- Vue :
  - s'abonne aux modifications du modèle (*observer*)
  - Appelle `LoginModel` pour mettre à jour son message
  - transmet les événements utilisateur au contrôleur
- Contrôleur modifie le modèle

Version MVP



- Modèle :
  - `LoginService` construit le message (même classe pour toutes les versions)
  - `LoginModel` stocke le nom
- Vue :
  - implémente l'interface `IPresenter`
  - transmet les événements utilisateur au Présentateur
- Présentateur :
  - modifie le modèle
  - appelle le service pour construire le message
  - utilise le `IPresenter` pour mettre à jour la vue

Version MVVM



- Modèle :
  - `LoginService` construit le message (même classe pour toutes les versions)
  - `LoginModel` stocke le nom
- Vue :
  - lie les `Property` de ses composants graphiques au `Property` de la Vue-Modèle
  - transmet les événements utilisateur au Vue-Modèle
- Vue-Modèle :
  - construit des `Property`
  - modifie le modèle
  - appelle le service pour construire le message