Présentation patrons de conception - MVC et ses amis

Charlie LEGER
Nathan LIBOUTET
Léo VIGUIER
Vincent VISSER

Plan

I – Qu’est ce qu’un pattern ?

II – Présentation du pattern MVC

III – Les patterns dérivés de MVC

IV – Conclusion

V - Questionnaire

I – Qu’est ce qu’un design pattern ?

Qu'est-ce qu'un pattern ?

Défini par : un nom, un problème, une solution et des conséquences

3 types de patrons : patron de conception, idiotisme, **patron d'architecture**

A Pattern Language, Christophe Alexander

Qu'est-ce qu'un *architectural pattern* ?

Patron qui apporte des solutions sur la manière de concevoir la structure générale.

Ce n'est pas le "quoi faire" mais le "comment faire"

Comment séparer les données de l'interface ?

II – Présentation du design pattern MVC

Ses origines ?

Le motif MVC a été créé par **Trygve Reenskaug** en **1978.**

La première implémentation de MVC a été réalisé dans un langage appelé *smalltalk*.

Mais qu'est ce que le pattern MVC ?

Modèle-Vue-Contrôleur : pattern architectural qui sépare les données (le modèle), l'interface homme-machine (la vue) et la logique de contrôle (le contrôleur).

Une séparation en 3 couches :

Le modèle : Il représente les données de l'application
La vue : Elle représente l'interface utilisateur, ce avec quoi il interagit
Le contrôleur : Il gère l'interface entre le modèle et le client

La synchronisation entre la vue et le modèle se passe avec le pattern Observer.

II – Présentation du design pattern MVC

Pourquoi a-t-il été créé ?

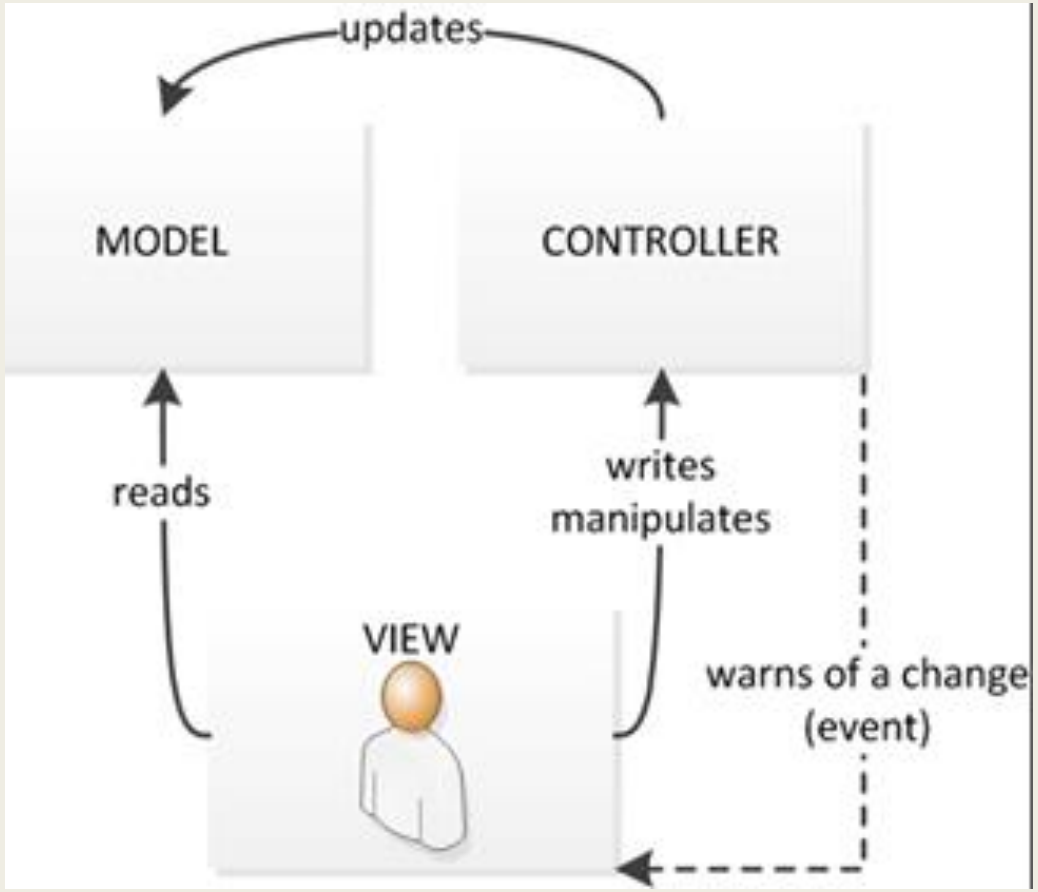
Selon son architecte Trygve Reenskaug :

«MVC a été créé pour résoudre le problème général qui consiste à donner aux utilisateurs le contrôle de leurs informations sous de multiples angles».

«Le but essentiel de MVC est de combler le fossé entre le modèle mental de l'utilisateur humain et le modèle numérique existant dans l'ordinateur. La solution MVC idéale soutient l'illusion de l'utilisateur de voir et de manipuler directement les informations du domaine. La structure est utile si l'utilisateur doit voir le même élément de modèle simultanément dans différents contextes et / ou sous différents points de vue. »

II – Présentation du design pattern MVC

Comment fonctionne-t-il ?



Ce patron fonctionne donc en cycle.

L'utilisateur interagit avec les composants graphiques des vues à sa disposition, ce qui se déclenche la création d'événements qui sont envoyés au contrôleur qui leur est associé.

Le contrôleur vérifie la conformité des interactions et déduit les modifications à apporter au modèle qui les intègre selon ses règles métier.

Les modifications du modèle sont ensuite signalées à toutes les vues qui se mettent à jour en conséquence.

II – Présentation du design pattern MVC

Avantages

Architecture claire et normalisée

Séparation des tâches

Grande capacité d'extension

Adapté aux applications graphiques

Désavantages

Double responsabilité de la vue
Viol de certains principes SOLID

Séparation des tâches

Lourdeur de la boucle d'action

Et les principes SOLID dans tout ça ?

Single Responsability Principle (SRP) : Non respecté

Open-Closed Principle (OCP) : Respecté

Liskov Substitution Principle (LSP) : N'intervient pas directement dans ce pattern

Interface Segregation Principle (ISP) : N'intervient pas directement dans ce pattern

Dependency Inversion Principle (DIP) : N'intervient pas directement dans ce pattern

```java
public class TelevisionModel extends Observable {

    private int volume;

    public TelevisionModel() {
        this.volume = 10;
    }

    public void monterLeVolume() {
        this.volume++;

        setChanged();
        notifyObservers();
    }

    public void baisserLeVolume() {
        this.volume--;

        setChanged();
        notifyObservers();
    }

    public int getVolume() {
        return this.volume;
    }
}
```

Une télévision
Le modèle

Elle peut :

- Monter le volume
- Baisser le volume

- **setChanged()** permet de préciser que le modèle a été changé
- **notifyObservers()** permet de notifier ses observateurs qui appelleront leur méthode **update()**

```java
public class TelevisionController {

    private TelevisionModel model;

    public TelevisionController(TelevisionModel model) {
        this.model = model;
    }

    public void monterVolume() {
        model.monterLeVolume();
    }

    public void baisserVolume() {
        model.baisserLeVolume();
    }

}
```

Le contrôleur

```java
public class TelevisionView implements Observer {

    private TelevisionModel model;
    private TelevisionController controller;

    private JFrame jf_television;

    private JLabel jl_messageVolumeTelevision;
    private JLabel jl_volumeTelevision;
    private JButton jb_monterVolume;
    private JButton jb_baisserVolume;

    public TelevisionView(TelevisionModel model, TelevisionController controller) {
        this.model = model;
        this.controller = controller;

        /*
         * On définit nos éléments graphiques...
         */

        //On ajoute nos écouteurs sur les boutons
        this.addUpListener();
        this.addDownListener();

        // Connexion entre le modèle et la vue
        this.model.addObserver(this);

        this.jf_television.setVisible(true);
```

La vue

```java
public void modifierLabelVolume(int volume) {
    this.jl_volumeTelevision.setText(Integer.toString(volume));
}

public void addUpListener() {
    jb_monterVolume.addActionListener(new ActionListener() {

        @Override
        public void actionPerformed(ActionEvent e) {
            controller.monterVolume();

        }
    });
}

public void addDownListener() {
    jb_baisserVolume.addActionListener(new ActionListener() {

        @Override
        public void actionPerformed(ActionEvent e) {
            controller.baisserVolume();

        }
    });
}

@Override
public void update(Observable o, Object arg) {
    this.modifierLabelVolume(this.model.getVolume());

}
```

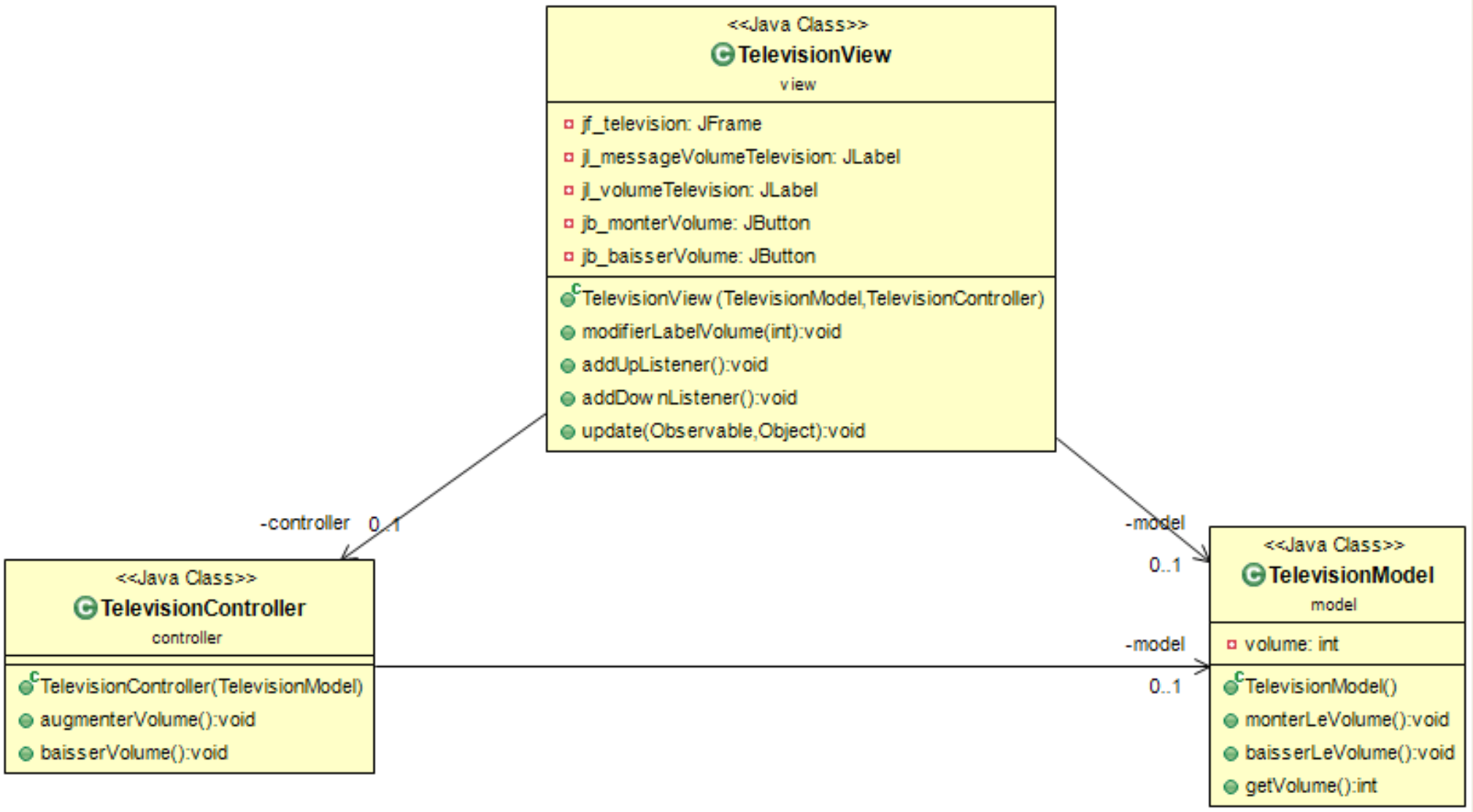
On ajoute nos **écouteurs** sur les boutons qui appelleront la méthode du contrôleur adaptée

On redéfinit la méthode **update()** qui va modifier le label en fonction des données du **modèle**

On créé ensuite notre lanceur :

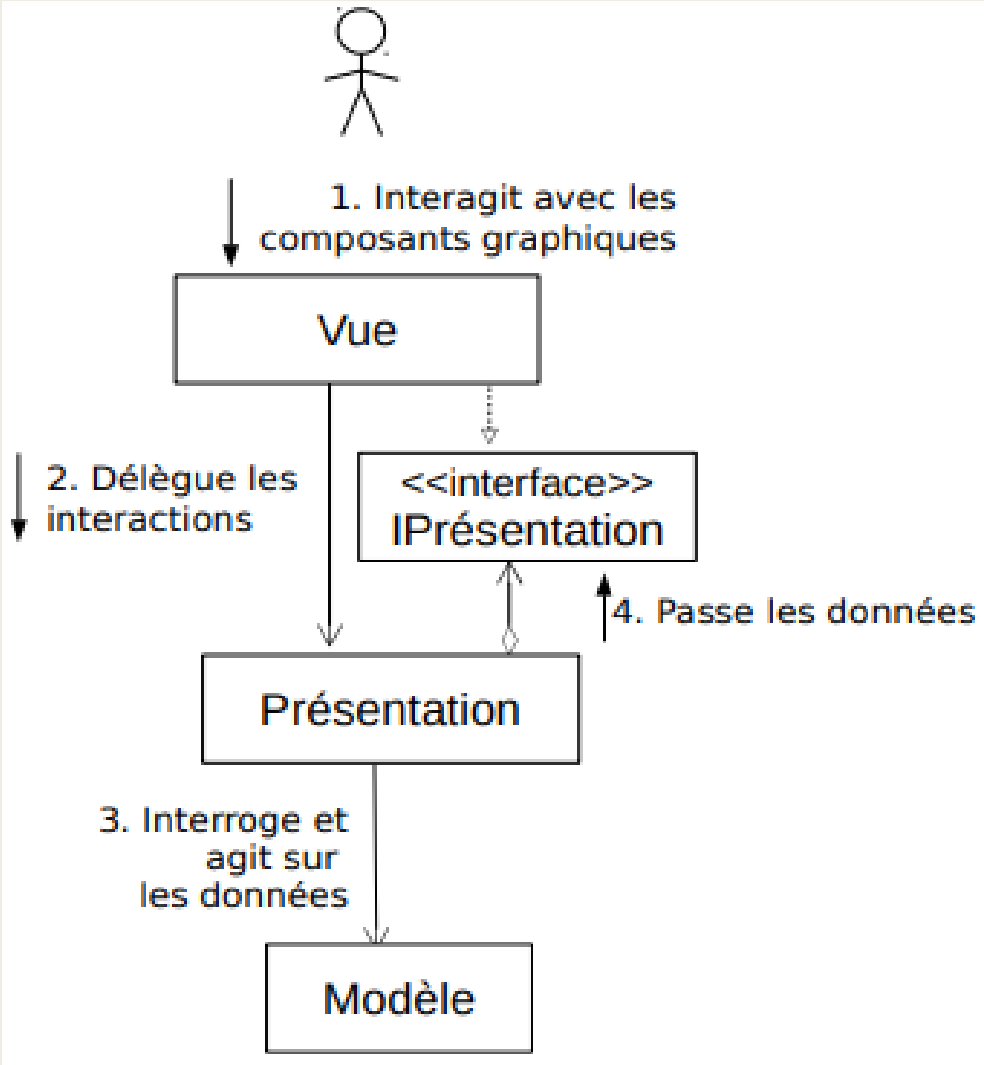
```java
public class TelevisionMVC {

    @SuppressWarnings("unused")
    public static void main(String[] args) {

        TelevisionModel model = new TelevisionModel();
        TelevisionController controller = new TelevisionController(model);
        TelevisionView view = new TelevisionView(model, controller);
    }
}
```

Diagramme de classe avec le Pattern MVC



III – Les patterns dérivés de MVC

A) Design pattern MVP (Model – View – Presenter)

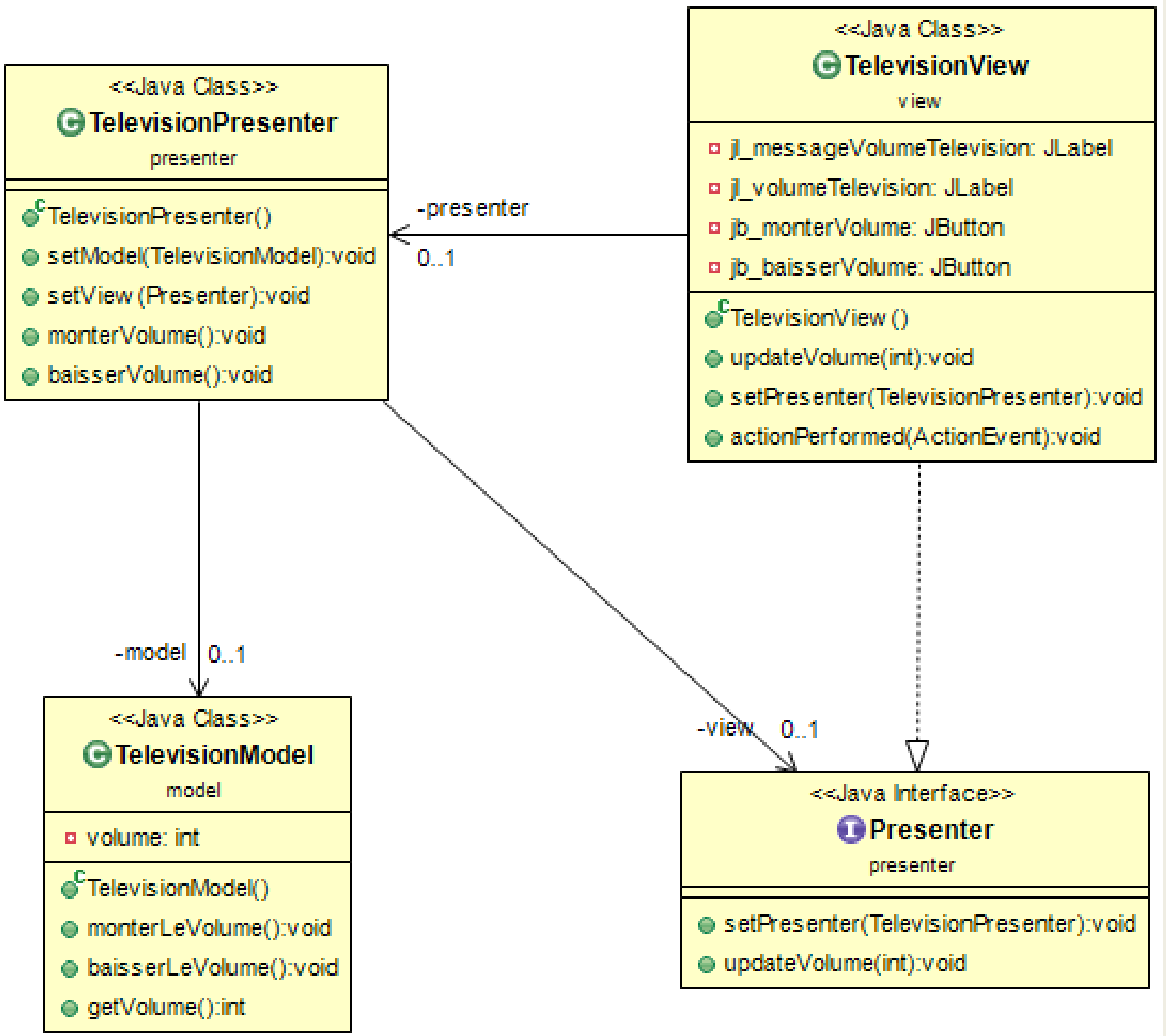


Le modèle est complètement déconnecté du dialogue avec l'utilisateur. Il est totalement indépendant des autres parties.

La vue ne concerne plus que la gestion graphique. Elle est totalement indépendante du reste de l'application, elle ne connait pas le modèle.

La présentation sait réagir aux événements de l'utilisateur en modifiant les données si nécessaire puis répercute les changements vers la vue. Elle sert de lien entre le modèle et la vue.

Exemple : le site de covoiturage développé en PHP.


<<Java Class>>
TelevisionPresenter
presenter
TelevisionPresenter()
setModel(TelevisionModel):void
setView (Presenter):void
monterVolume():void
baisserVolume():void
<<Java Class>>
TelevisionView
view
jl_messageVolumeTelevision: JLabel
jl_volumeTelevision: JLabel
jb_monterVolume: JButton
jb_baisserVolume: JButton
TelevisionView ()
updateVolume(int):void
setPresenter(TelevisionPresenter):void
actionPerformed(ActionEvent):void
-presenter
0..1
-model
0..1
-view
0..1
<<Java Class>>
TelevisionModel
model
volume: int
TelevisionModel()
monterLeVolume():void
baisserLeVolume():void
getVolume():int
<<Java Interface>>
Presenter
presenter
setPresenter(TelevisionPresenter):void
updateVolume(int):void

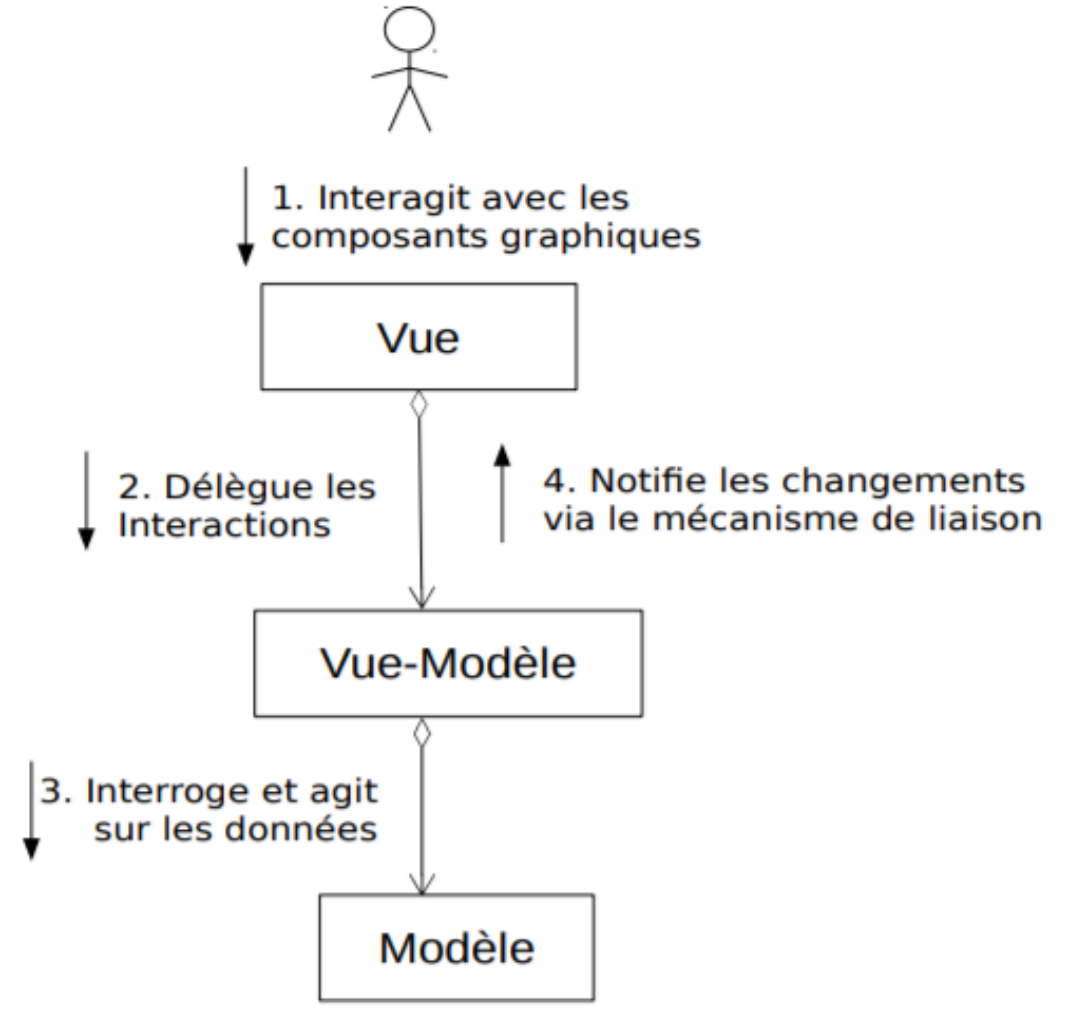
III – Les patterns dérivés de MVC
B) Design pattern MVVM

Le pattern MVVM est pratiquement le même que le MVP, à l'exception d'une différence majeure

Ce qu'il y a de nouveaux :

- Mécanisme de liaison de données
- La présentation devient la vue-modèle
- Le pattern Observer est sur chaque composant graphique de la vue

III – Les patterns dérivés de MVC
B) Design pattern MVVM



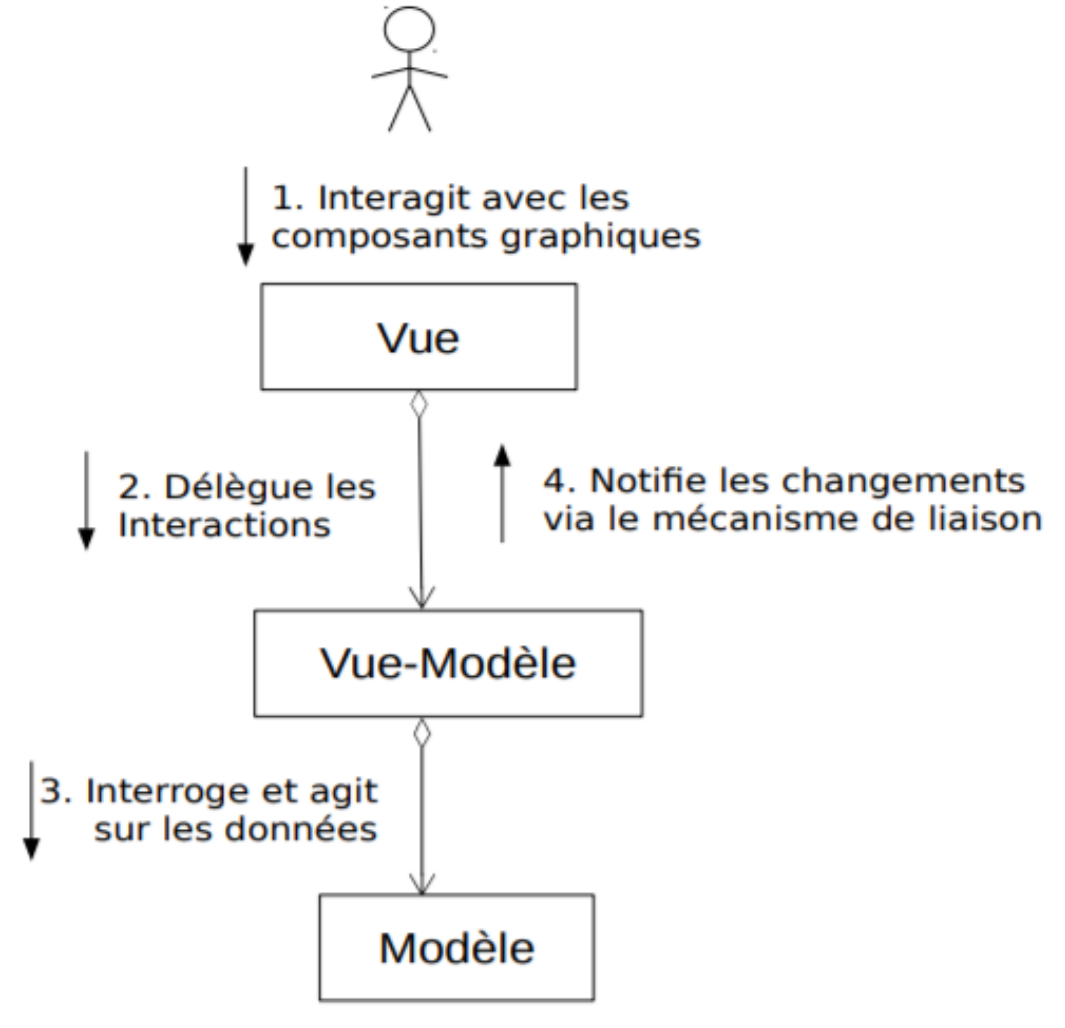
Caractéristiques détaillées

La vue ne connait aucune logique

La vue-modèle représente l'état de la vue, mais ne sait pas de quoi elle est composée. La vue-modèle contient toute la logique de l'interface utilisateur nécessaire pour définir l'état de la vue.
Assure la médiation entre la vue et le modèle

Le modèle contient la logique métier

III – Les patterns dérivés de MVC
B) Design pattern MVVM



Ce qu'il y a de nouveaux :

- La vue connait la vue-modèle mais ne connait pas le modèle
- La vue-modèle connait le modèle mais pas la vue
- Le modèle ne connait ni l'un ni l'autre

Des avantages ?

Questionnaire en ligne

https://qcm-mvc.netlify.com/