



Implémentation du pattern MVC


par Baptiste Wicht ([home](#))

Date de publication : Le 24 Avril 2007

 **MVC** est un  **design pattern** très puissant, mais néanmoins assez complexe, qui permet de bien structurer de grosses applications graphiques. Nous allons apprendre à le maîtriser avec ce tutoriel.

I - Le design pattern MVC.....	3
II - Implémentation.....	5
II-A - Le modèle.....	5
II-B - Le contrôleur.....	6
II-C - Les vues.....	8
II-D - La classe lanceur.....	11
II-E - Résultat.....	11
III - Changement de framework graphique.....	12
IV - Conclusion.....	14

I - Le design pattern MVC

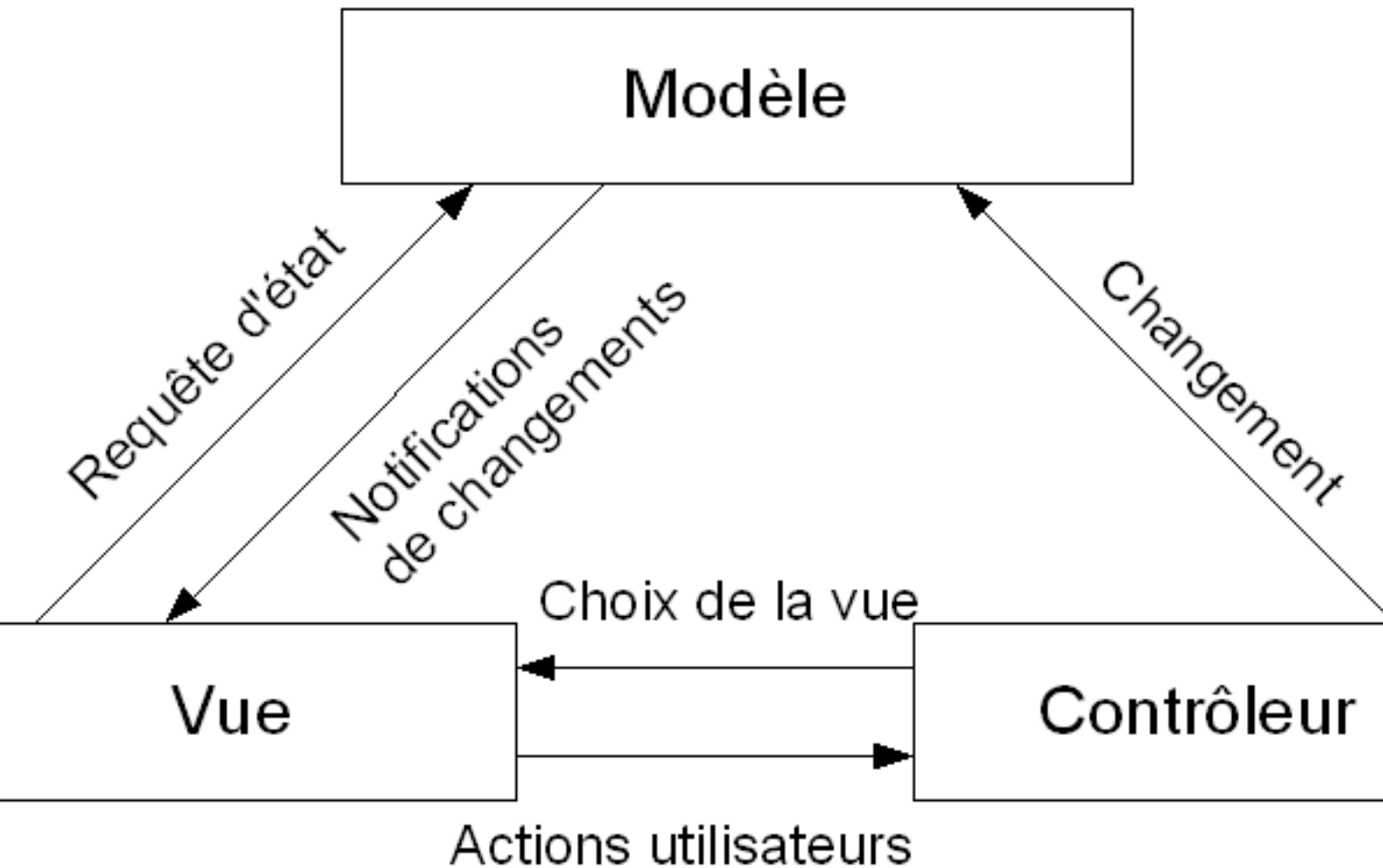
Le  **design pattern** Modèle-Vue-Contrôleur (MVC) est un pattern architectural qui sépare les données (le modèle), l'interface homme-machine (la vue) et la logique de contrôle (le contrôleur).

Ce modèle de conception impose donc une séparation en 3 couches :

- Le modèle : Il représente les données de l'application. Il définit aussi l'interaction avec la base de données et le traitement de ces données.
- La vue : Elle représente l'interface utilisateur, ce avec quoi il interagit. Elle n'effectue aucun traitement, elle se contente simplement d'afficher les données que lui fournit le modèle. Il peut tout à fait y avoir plusieurs vues qui présentent les données d'un même modèle.
- Le contrôleur : Il gère l'interface entre le modèle et le client. Il va interpréter la requête de ce dernier pour lui envoyer la vue correspondante. Il effectue la synchronisation entre le modèle et les vues.

La synchronisation entre la vue et le modèle se passe avec le pattern Observer. Il permet de générer des événements lors d'une modification du modèle et d'indiquer à la vue qu'il faut se mettre à jour.

Voici un schéma des interactions entre les différentes couches :



Interactions entre les couches

Ce modèle de conception permet principalement 2 choses :

- Le changement d'une couche sans altérer les autres. C'est-à-dire que comme toutes les couches sont clairement séparées, on doit pouvoir en changer une pour, par exemple, remplacer Swing par SWT sans porter atteinte aux autres couches. On pourrait aussi donc changer le modèle sans toucher à la vue et au contrôleur. Cela rend les modifications plus simples.
- La synchronisation des vues. Avec ce design pattern, toutes les vues qui montrent la même chose sont synchronisées.

Il faut tout de même garder en mémoire, que la mise en oeuvre de MVC dans une application n'est pas des plus simples. En effet, ce modèle de conception introduit tout de même un niveau de complexité assez élevé. De plus, implémenter MVC dans votre application nécessite une bonne conception dès le départ. Ce qui peut prendre du temps. Ce pattern n'est donc à conseiller que pour les moyennes et grandes applications.

II - Implémentation

Nous allons choisir un exemple très simple (voire même simpliste). Ce sera tout simplement une application permettant de modifier un volume. Il y aura plusieurs vues pour représenter ce volume et après toute modification, les vues devront être synchronisées. L'interface sera développée avec Swing.

C'est certes bidon et très petit comme exemple, mais cela permettra de voir simplement la séparation des couches et l'utilisation du design pattern Observer.

Par souci de simplicité, nous n'utiliserons ici qu'un seul package appelé volume. Dans une grosse application MVC, je vous conseille plutôt de clairement séparer le modèle, les vues et l'interface dans des packages différents.

II-A - Le modèle

Le modèle est assez simple à développer, vu qu'il ne gère qu'un volume. Pour commencer, on va déjà développer la base de notre modèle :

VolumeModel.java

```
public class VolumeModel {
    private int volume;

    public VolumeModel() {
        super();

        volume = 0;
    }

    public int getVolume() {
        return volume;
    }

    public void setVolume(int volume) {
        this.volume = volume;
    }
}
```

Voilà, la première fonction de notre modèle est remplie, il fournit maintenant un volume qui peut être modifié.

Mais maintenant, il faut que notre modèle puisse notifier un changement de volume. Pour cela, on va employer les listeners. On va donc créer un nouveau listener (VolumeListener) et un nouvel événement (VolumeChangedEvent) :

VolumeListener.java

```
import java.util.EventListener;

public interface VolumeListener extends EventListener {
    public void volumeChanged(VolumeChangedEvent event);
}
```

VolumeChangedEvent.java

```
import java.util.EventObject;

public class VolumeChangedEvent extends EventObject {
    private int newVolume;

    public VolumeChangedEvent(Object source, int newVolume) {
        super(source);

        this.newVolume = newVolume;
    }

    public int getNewVolume() {
```

VolumeChangedEvent.java

```
return newVolume;
}
}
```

Maintenant, nous allons implémenter ce système d'écouteurs dans le modèle pour que d'autres entités puissent "écouter" les changements du modèle.

VolumeModel.java

```
import javax.swing.event.EventListenerList;

public class VolumeModel {
    private int volume;

    private EventListenerList listeners;

    public VolumeModel() {
        this(0);
    }

    public VolumeModel(int volume) {
        super();

        this.volume = volume;

        listeners = new EventListenerList();
    }

    public int getVolume() {
        return volume;
    }

    public void setVolume(int volume) {
        this.volume = volume;

        fireVolumeChanged();
    }

    public void addVolumeListener(VolumeListener listener) {
        listeners.add(VolumeListener.class, listener);
    }

    public void removeVolumeListener(VolumeListener l) {
        listeners.remove(VolumeListener.class, l);
    }

    public void fireVolumeChanged() {
        VolumeListener[] listenerList = (VolumeListener[]) listeners.getListeners(VolumeListener.class);

        for (VolumeListener listener : listenerList) {
            listener.volumeChanged(new VolumeChangedEvent(this, getVolume()));
        }
    }
}
```

Voilà, maintenant notre modèle avertit tous ses écouteurs à chaque changement de volume. Ensuite, en fonction de l'application, on peut tout à fait imaginer plusieurs listeners par modèles et d'autres événements dans les listeners, par exemple quand le volume dépasse certains seuils... Vous voyez donc qu'un modèle peut très vite devenir conséquent.

II-B - Le contrôleur

Nous allons maintenant développer notre contrôleur. Comme le contrôleur doit le moins possible être dépendant de Swing, on va créer une classe abstraite représentant une vue du volume.

VolumeView.java

```
public abstract class VolumeView implements VolumeListener {
    private VolumeController controller = null;

    public VolumeView(VolumeController controller) {
        super();

        this.controller = controller;
    }

    public final VolumeController getController() {
        return controller;
    }

    public abstract void display();
    public abstract void close();
}
```

Maintenant notre contrôleur ne manipulera que des objets de type View et non plus de type Swing.

A nouveau dans un souci de simplicité, nous allons créer un seul contrôleur pour les 3 vues que nous aurons. Dans notre cas, c'est plus simple de faire ainsi, vu que nos 3 vues font toutes la même chose et font très peu de choses. Dans le cas de vue fondamentalement différentes, il est fortement conseillé d'utiliser plusieurs contrôleurs.

VolumeController.java

```
public class VolumeController {
    public VolumeView fieldView = null;
    public VolumeView spinnerView = null;
    public VolumeView listView = null;

    private VolumeModel model = null;

    public VolumeController (VolumeModel model) {
        this.model = model;

        fieldView = new JFrameFieldVolume(this, model.getVolume());
        spinnerView = new JFrameSpinnerVolume(this, model.getVolume());
        listView = new JFrameListVolume(this, model.getVolume());

        addListenersToModel();
    }

    private void addListenersToModel() {
        model.addVolumeListener(fieldView);
        model.addVolumeListener(spinnerView);
        model.addVolumeListener(listView);
    }

    public void displayViews() {
        fieldView.display();
        spinnerView.display();
        listView.display();
    }

    public void closeViews() {
        fieldView.close();
        spinnerView.close();
        listView.close();
    }

    public void notifyVolumeChanged(int volume) {
        model.setVolume(volume);
    }
}
```

II-C - Les vues

Comme nous l'avons vu avec le développement du contrôleur, nous allons avoir 3 vues :

- Une vue permettant de modifier le volume avec un champ texte avec un bouton permettant de valider le nouveau volume : JFrameFieldVolume
- Une vue permettant de modifier le volume à l'aide d'un spinner avec un bouton permettant de valider le nouveau volume : JFrameSpinnerVolume
- Une vue listant les différents volumes et qui ajoutera chaque nouveau volume dans une liste déroulante : JFrameListVolume

Toutes ces vues seront représentées par une JFrame.

Voici donc nos 3 vues :

JFrameFieldVolume.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.text.NumberFormat;

import javax.swing.JButton;
import javax.swing.JFormattedTextField;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.text.DefaultFormatter;

public class JFrameFieldVolume extends VolumeView implements ActionListener{
    private JFrame frame = null;
    private JPanel contentPane = null;
    private JFormattedTextField field = null;
    private JButton button = null;
    private NumberFormat format = null;

    public JFrameFieldVolume(VolumeController controller) {
        this(controller, 0);
    }

    public JFrameFieldVolume(VolumeController controller, int volume){
        super(controller);

        buildFrame(volume);
    }

    private void buildFrame(int volume) {
        frame = new JFrame();

        contentPane = new JPanel();

        format = NumberFormat.getNumberInstance();
        format.setParseIntegerOnly(true);
        format.setGroupingUsed(false);
        format.setMaximumFractionDigits(0);
        format.setMaximumIntegerDigits(3);

        field = new JFormattedTextField(format);
        field.setValue(volume);
        ((DefaultFormatter) field.getFormatter()).setAllowsInvalid(false);
        contentPane.add(field);

        button = new JButton("Mettre à jour");
        button.addActionListener(this);
        contentPane.add(button);

        frame.setContentPane(contentPane);
        frame.setTitle("JFrameSpinnerVolume");
        frame.pack();
    }
}
```

JFrameFieldVolume.java

```
}

@Override
public void close() {
    frame.dispose();
}

@Override
public void display() {
    frame.setVisible(true);
}

public void volumeChanged(VolumeChangedEvent event) {
    field.setValue(event.getNewVolume());
}

public void actionPerformed(ActionEvent arg0) {
    getController().notifyVolumeChanged(Integer.parseInt(field.getValue().toString()));
}
}
```

JFrameListVolume.java

```
import javax.swing.DefaultListModel;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;

public class JFrameListVolume extends VolumeView {
    private JFrame frame = null;
    private JPanel contentPane = null;
    private JList listVolume = null;
    private JScrollPane scrollVolume = null;
    private DefaultListModel jListModel = null;

    public JFrameListVolume(VolumeController controller) {
        this(controller, 0);
    }

    public JFrameListVolume(VolumeController controller, int volume) {
        super(controller);

        buildFrame(volume);
    }

    private void buildFrame(int volume) {
        frame = new JFrame();

        contentPane = new JPanel();

        jListModel = new DefaultListModel();
        jListModel.addElement(volume);

        listVolume = new JList(jListModel);

        scrollVolume = new JScrollPane(listVolume);
        contentPane.add(scrollVolume);

        frame.setContentPane(contentPane);
        frame.setTitle("JFrameListVolume");
        frame.pack();
    }

    @Override
    public void close() {
        frame.dispose();
    }

    @Override
    public void display() {
```

JFrameListVolume.java

```
frame.setVisible(true);
}

public void volumeChanged(VolumeChangedEvent event) {
    jListModel.addElement(event.getNewVolume());
}
}
```

JFrameSpinnerVolume.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSpinner;
import javax.swing.SpinnerNumberModel;

public class JFrameSpinnerVolume extends VolumeView implements ActionListener{
    private JFrame frame = null;
    private JPanel contentPane = null;
    private JSpinner spinner = null;
    private SpinnerNumberModel spinnerModel = null;
    private JButton button = null;

    public JFrameSpinnerVolume(VolumeController controller) {
        this(controller, 0);
    }

    public JFrameSpinnerVolume(VolumeController controller, int volume){
        super(controller);

        buildFrame(volume);
    }

    private void buildFrame(int volume) {
        frame = new JFrame();

        contentPane = new JPanel();

        spinnerModel = new SpinnerNumberModel(volume, 0, 100, 1);

        spinner = new JSpinner(spinnerModel);
        contentPane.add(spinner);

        button = new JButton("Mettre à jour");
        button.addActionListener(this);
        contentPane.add(button);

        frame.setContentPane(contentPane);
        frame.setTitle("JFrameSpinnerVolume");
        frame.pack();
    }

    @Override
    public void close() {
        frame.dispose();
    }

    @Override
    public void display() {
        frame.setVisible(true);
    }

    public void volumeChanged(VolumeChangedEvent event) {
        spinnerModel.setValue(event.getNewVolume());
    }

    public void actionPerformed(ActionEvent arg0) {
        getController().notifyVolumeChanged(spinnerModel.getNumber().intValue());
    }
}
```

JFrameSpinnerVolume.java

```
}  
}
```

Voilà, nos 3 vues sont maintenant développées. Elles sont tout ce qu'il y a de plus basiques et on pourrait bien sûr les optimiser, en les plaçant correctement sur l'écran, en donnant une bonne taille aux composants, ... Mais là n'est pas le but de ce tutoriel.

II-D - La classe lanceur

On va maintenant créer la classe "main" de l'application. Sa fonction est plus que simple, elle crée un nouveau modèle, crée un nouveau contrôleur en lui passant le modèle et demande au contrôleur d'afficher les vues.

Voici donc à quoi elle va ressembler :

JVolume.java

```
public class JVolume {  
    public static void main(String[] args) {  
        VolumeModel model = new VolumeModel(50);  
        VolumeController controller = new VolumeController(model);  
        controller.displayViews();  
    }  
}
```

Rien de bien compliqué donc.

II-E - Résultat

Voilà, notre exemple est maintenant terminé. Vous voyez que nous avons pu rapidement mettre en oeuvre une architecture MVC, mais que ça augmente tout de même le temps de développement et la taille du code et que nous sommes dans une toute petite application. Mais le résultat est tout de même là, toutes nos vues sont synchronisées et le moindre changement sur une des vues est tout de suite actifs sur les autres vues.

Avec cet architecture, si nous voulons ajouter une nouvelle vue, il suffit tout simplement d'ajouter 4 lignes dans le contrôleur (une pour la déclaration, une pour l'initialisation et une autre pour les méthodes close et display).

III - Changement de framework graphique

Nous allons changer de framework graphique pour une des vues. Nous allons donc modifier notre JFrameListVolume pour la faire passer en AWT. C'est donc une nouvelle classe FrameListVolume qui va venir la remplacer.

Le premier changement à faire est de remplacer dans le constructeur du contrôleur JFrameListVolume par FrameListVolume :

Constructeur du contrôleur

```
public VolumeController (VolumeModel model) {
    this.model = model;

    fieldView = new JFrameFieldVolume(this, model.getVolume());
    spinnerView = new JFrameSpinnerVolume(this, model.getVolume());
    listView = new FrameListVolume(this, model.getVolume());

    addListenersToModel();
}
```

Ensuite nous allons développer notre interface avec AWT :

FrameListVolume.java

```
import java.awt.Frame;
import java.awt.List;

public class FrameListVolume extends VolumeView {
    private Frame frame = null;
    private List listVolume = null;

    public FrameListVolume(VolumeController controller) {
        this(controller, 0);
    }

    public FrameListVolume(VolumeController controller, int volume) {
        super(controller);

        buildFrame(volume);
    }

    private void buildFrame(int volume) {
        frame = new Frame();

        listVolume = new List();
        listVolume.add(Integer.toString(volume));
        frame.add(listVolume);

        frame.setTitle("JFrameListVolume");
        frame.pack();
    }

    @Override
    public void close() {
        frame.dispose();
    }

    @Override
    public void display() {
        frame.setVisible(true);
    }

    public void volumeChanged(VolumeChangedEvent event) {
        listVolume.add(Integer.toString(event.getNewVolume()));
    }
}
```


IV - Conclusion

Voilà, vous venez de développer votre première (ou une nouvelle pour certains) application respectant l'architecture MVC. Comme vous avez pu le voir, cette architecture permet une très bonne séparation des couches et facilite l'ajout ou la modification de vues. Par contre, elle nécessite un travail supplémentaire et augmente la quantité de code à écrire. Elle n'est donc destinée qu'à de grosses applications ou à de moyennes applications en évolution constante. Mais rien ne vous empêche de l'utiliser dans n'importe quel développement.

Télécharger **les sources**.

Vous trouverez d'autres ressources sur MVC sur le **site de Serge Tahé**.

Merci à **trinityDev** pour ses corrections.