Exemple Modèle MVC : Volumes

Résumé

Cet exemple introduit l'usage du modèle MVC, telle que proposé par Oracle (anciennement Sun) [1]. Dans cet exemple, le modèle représente la valeur d'un volume quelconque stocké, tandis que les vues proposent l'interface utilisateur pour manipuler ce modèle et le contrôle gère les interactions entre les vues et le modèle. Quatre types de vues sont proposés, dont deux dites « actives », qui permettent à l'utilisateur de modifier la valeur du volume, et deux « passives », qui ne le permettent pas. Les deux premières utilisent les composants Swing JSpinner et JSlider, tandis que les deux dernières utilisent les composants JList et JProgressBar.

Enfin, il convient de rappeler que cet exemple est basé sur celui proposé par Baptiste Wicht [2].

Table des matières

Structure de l'application

Le diagramme de classe ci-dessous résume l'application de gestion de volumes qu'on propose.

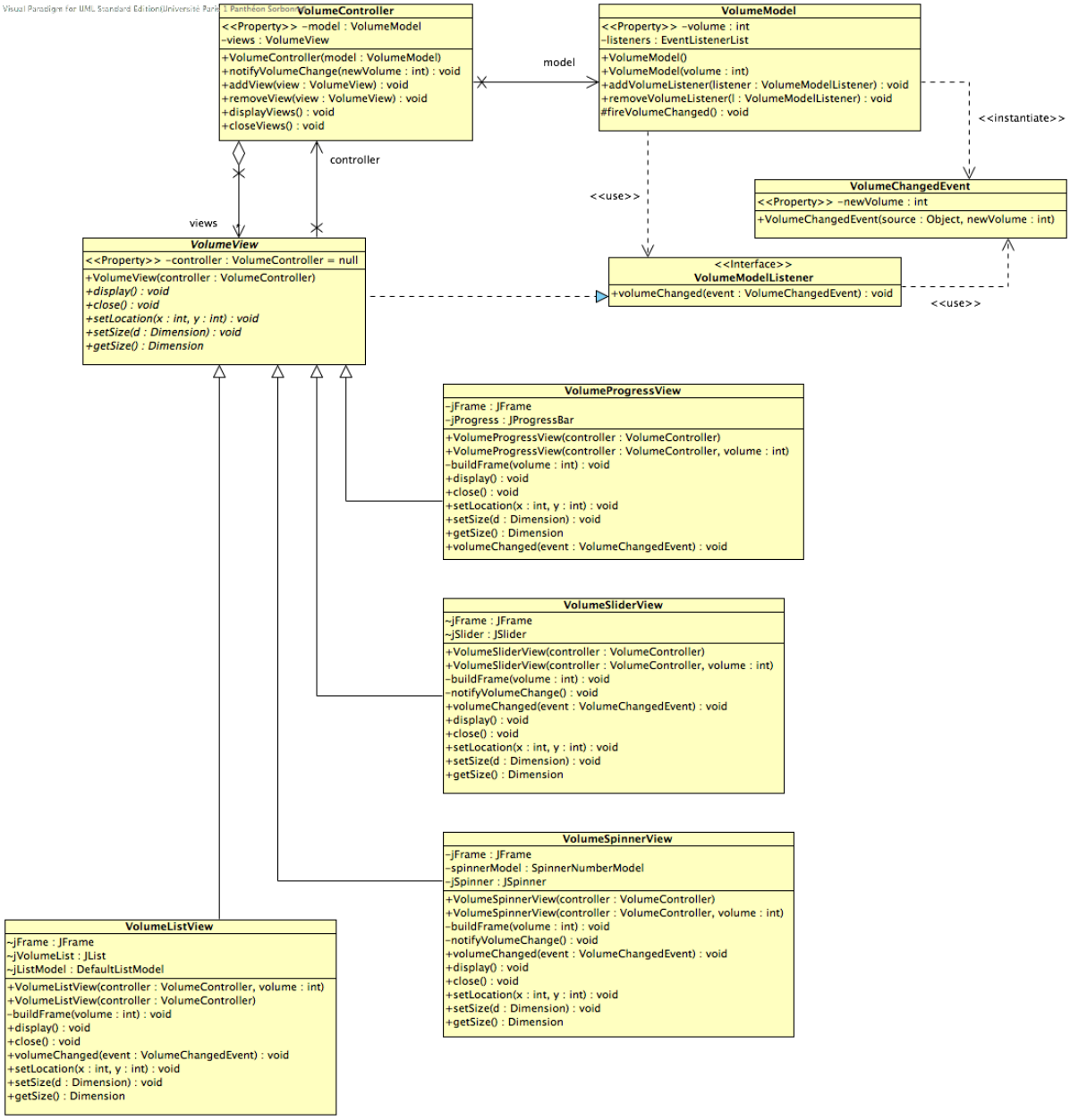


Figure 1. Diagramme de classes de l'application Volume

La première chose à observer dans ce diagramme est la position du modèle. La classe *VolumeModel* est isolée. La seule dépendance qu'elle reste les événements qu'elle même est capable de générer, en occurrence ici l'événement *VolumeChangedEvent*, et l'interface définissant leurs écouteurs (*VolumeModelListener*). La connaissance que le modèle a du monde s'arrête dans l'interface *VolumeModelListener*. Peu importe au modèle qui implémente réellement celle-ci, tout ce dont il a besoin se résume à ça. Cette application du *design pattern Observer* est la clé de voute du modèle

MVC. Le modèle reste indépendant des autres éléments (il ne connaît ni le contrôleur, ni les vues), tout en étant capable de notifier ses observateurs (*listeners*) de toute modification survenue. On observe alors l'importance des *événements générés par le modèle* (*on ne parle pas ici des événements Swing*) dans l'application du MVC : sans ces événements, l'application de MVC est incomplète car le modèle ne pourrait avertir personne lors d'un quelconque changement.

L'autre point clé du modèle MVC est la relation entre le (ou les) contrôleur(s) et les vues. Ces deux éléments fonctionnent bien souvent en tandem : un contrôleur est censé gérer un type de vue, et les vues sont censées être gérées par un contrôleur d'un type bien précis. Ici, afin d'augmenter la réutilisation des éléments, notre contrôleur est capable de manipuler non une seule vue, mais tout un ensemble de vues d'une famille bien précise, fait signalé par l'agrégation entre les classes *VolumeControler* et *VolumeView*. Cette famille de vues, symbolisée par la classe abstraite *VolumeView*, représente des vues qui ont un comportement commun, mis en œuvre par cette classe abstraite. Le contrôleur gère ainsi un ensemble de vues présentant ce comportement attendu. Chacune de ces vues connaît leur contrôleur : à chaque action de l'utilisateur, elles vont l'invoquer, lui transmettant les informations nécessaires (dans notre cas, la nouvelle valeur qui doit être attribuée au volume).

Cette organisation n'est possible car les vues qu'on manipule dans cet exemple sont particulièrement simples. Elles ne manipulent qu'une seule valeur (le volume, en occurrence) et sont ainsi capable de gérer elles-mêmes leurs événements Swing. Toute la partie graphique et l'interaction avec l'utilisateur restent ainsi limitées aux vues, elles ne se propagent pas au contrôleur. Le contrôleur se limite à son rôle de communication entre l'interface graphique (les vues) et le modèle. Le fait de dégager du contrôleur la gestion des événements Swing nous permet de proposer un contrôleur plus générique, qui peut être plus facilement réutilisé, avec d'autres vues de la même famille (qui héritent de *VolumeView*). La dépendance entre le contrôleur et les vues, même si elle reste importante, est moins forte que si le contrôleur avait en plus la charge des événements Swing. Ceci a l'avantage de permettre une extension plus facile de l'application par l'ajout de nouvelles vues accouplées au même contrôleur : le contrôleur étant capable de gérer plusieurs vues, il suffit d'y ajouter un nouveau objet héritant de *VolumeView* pour qu'il puisse aussi le gérer. En somme : plus d'extensibilité et plus de réutilisation.

Ce qui doit intriguer plus d'un dans le diagramme ci-dessus est l'implémentation de l'interface *VolumeModelListener.* Celle-ci se trouve, comme l'indique la Figure 1, sur la classe abstraite *VolumeView* et non sur le contrôleur, comme on voit souvent. Il s'agit d'une application plus traditionnelle de MVC, telle que définit le Sun's BluePrint [1] ou la GoF[1] [3]. En effet, l'application étant particulièrement simple, l'événement *VolumeChangedEvent* peut porter, en lui-même, toute l'information nécessaire aux vues, qui peuvent alors se mettre à jour rapidement. Tels que les événements Swing (dont l'implémentation on a discuté ci-dessus), les vues héritant *VolumeView* vont être également capable d'écouter les événements relatifs aux changements effectués sur les données (*VolumeChangedEvent*). Cependant, les vues sont incapables de s'abonner elles-mêmes à ces événements (contrairement aux événements Swing), car elles ne connaissent pas leur source,

[1] GoF (« *Gang of Four* ») est le surnom donné au groupe d'auteurs du livre tenu comme précurseur de *design Patterns* [3].

qui est le modèle. Seul le contrôleur ou l'application (main) seront capables de souscrire les vues aux événements de type *VolumeChangedEvent,* car ils sont les seuls qui connaissent l'objet *VolumeModel*. C'est à eux donc de procéder à l'abonnement.

Codes sources commentés

Le modèle : la class VolumeModel

Dans la classe *VolumeModel*, il est important d'observer le déclanchement des événements de type *VolumeChangedEvent* et la gestion des écouteurs *VolumeModelListener*. Lors qu'une modification survient dans le modèle (c.a.d. que la valeur attribuée au volume est modifiée), un événement de ce type est créé et envoyé à chaque écouteur (*listeners)* enregistré, en invoquant sur celui-ci la méthode *volumeChanged*, définie dans l'interface *VolumeModelListener*. L'ensemble d'écouteurs est gardé dans une liste prévue pour cet effet (*EventListenerList*), laquelle permet de gérer des écouteurs de différents types. La méthode *addVolumeListener* permet l'abonnement de nouveaux écouteurs aux événements de type *VolumeChangedEvent*.

```
/*
 * UNIVERSITE PARIS 1 (PANTHEON SORBONNE)
 * MIAGE - UFR 27
 * Cours ISI5 / INF2 - Developpement d'Interface
 */
package review.ihm.volumeapplication;

import javax.swing.event.EventListenerList;

/**
 * Example d'usage du modele MVC
 * Cette classe definit le modele de donnees indicant un volume.
 *
 * Example base sur Baptiste Wicht, <b>"Implémentation du pattern MVC"</b>,
 * 24 Avril 2007 (derniere visite le 29 Janvier 2009).
 * @see http://baptiste-wicht.developpez.com/tutoriel/conception/mvc/
 * @author kirsch
 */
public class VolumeModel {

   public VolumeModel() {
      this(0);
   }

   public VolumeModel(int volume) {
      this.volume = volume;
      this.listeners = new EventListenerList();
   }

   /**
    * Cette methode permet de recuperer l'etat du modele, a savoir la valeur
    * du volume stocke.
    * @return int volume
    */
   public int getVolume() {
      return volume;
   }
```

```
    /**
     * Cette methode permet de modifier l'etat du modele, a savoir la valeur
     * du volume stocke.
     * @param volume nouvelle valeur de volume stocke
     */
    public void setVolume(int volume) {
        this.volume = volume;
        //on modifie l'etat du modele, on averti les listeners
        fireVolumeChanged();
    }

/**
     * Methode utilisee pour abonner un nouveau listener a ce modele
     * @param listener nouveau listener
     */
    public void addVolumeListener(VolumeModelListener listener) {
        listeners.add(VolumeModelListener.class, listener);
    }

    /**
     * Methode utilisee pour desabonner un listener de ce modele
     * @param l listener qui veut se desabonner
     */
    public void removeVolumeListener(VolumeModelListener l) {
        listeners.remove(VolumeModelListener.class, l);
    }

    /**
     * methode declanchant la notification des tous les listeners suite a une
     * modification dans l'etat du modele.
     */
    protected void fireVolumeChanged() {
        VolumeModelListener[] listenerList =
                (VolumeModelListener[]) listeners.getListeners(VolumeModelListener.class);
        //foreach listener, on appelle la methode volumeChanged
        for (VolumeModelListener listener : listenerList) {
            listener.volumeChanged(new VolumeChangedEvent(this, this.getVolume()));
        }
    }

    //volume
    private int volume;
    //listeners
    private EventListenerList listeners;
}
```

Les événements : la classe VolumeChangedEvent et l'interface VolumeModelListener

La classe *VolumeChangedEvent* est une classe simple, représentant un changement survenu sur le volume. En tant qu'événement, elle s'inscrit dans l'hiérarchie Java en héritant la classe *EventObject*, classe définissant la notion d'événement en Java. Ainsi comme d'autres événements en Java, celui-ci contient une charge utile, composée de sa source (objet sur lequel l'événement a eu lieu) et d'une information, dans notre cas, le nouveau volume. La méthode *getNewVolume* permet ainsi aux écouteurs de récupérer cette information.

```
/*
UNIVERSITE PARIS 1 (PANTHEON SORBONNE)
MIAGE - UFR 27
Cours ISI5 / INF2 - Developpement d'Interface

 */
package review.ihm.volumeapplication;

/**
 * Example d'usage du modele MVC
 * Cette classe definit un evennement de modification de l'etat du modele.
 * Un objet VolumeChangedEvent indique que le modele (indicant un volume)
 * a ete modifie.
 *
 * Example base sur Baptiste Wicht, <b>"Implémentation du pattern MVC"</b>,
 * 24 Avril 2007 (derniere visite le 29 Janvier 2009).
 * @see http://baptiste-wicht.developpez.com/tutoriel/conception/mvc/
 * @author kirsch
 */
class VolumeChangedEvent extends java.util.EventObject {

    public VolumeChangedEvent(Object source, int newVolume) {
        super(source);
        this.newVolume = newVolume;
    }

    public int getNewVolume() {
        return newVolume;
    }

    private int newVolume;
}
```

L'interface *VolumeModelListener* est encore plus simple. Elle ne définit qu'une seule méthode, *volumeChanged*, qui est celle invoqué par la source de l'événement afin de notifier les écouteurs implémentant cette interface qu'un changement a bien eu lieu.

```
/*
 UNIVERSITE PARIS 1 (PANTHEON SORBONNE)
 MIAGE - UFR 27
 Cours ISI5 / INF2 - Developpement d'Interface

 */

package review.ihm.volumeapplication;
```

```
/**
 * Example d'usage du modele MVC
 * Cette classe definit un listenener pour le modele de donnees.
 * A chaque modification sur le modele, un evennement du type VolumeChangedEvent
 * sera envoye a tous les listeners du modele.
 *
 * Example base sur Baptiste Wicht, <b>"Implémentation du pattern MVC"</b>,
 * 24 Avril 2007 (derniere visite le 29 Janvier 2009).
 * @see http://baptiste-wicht.developpez.com/tutoriel/conception/mvc/
 * @author kirsch
 */
public interface VolumeModelListener extends java.util.EventListener {
   public void volumeChanged(VolumeChangedEvent event);
}
```

Le contrôleur : la classe VolumeController

La classe *VolumeController* contient un ensemble de vues de type *VolumeView*, qu'elle doit gérer. Elle gère également l'abonnement de ces vues aux événements *VolumeChangedEvent* : à chaque nouvelle vue ajoutée, dans la méthode *addView*, le contrôleur va l'abonné au modèle (*this.model.addVolumeListener(view)*). Inversement, chaque vue peut communiquer à son contrôleur une nouvelle valeur pour le volume, à travers la méthode *notifyVolumeChange*. Il ne faut pas non plus oublier que le contrôleur reçoit une référence au modèle par paramètre. En aucun cas, il n'instancie pas le modèle. Celui-ci est toujours injecté sur le contrôleur.

```
/*
 UNIVERSITE PARIS 1 (PANTHEON SORBONNE)
 MIAGE - UFR 27
 Cours ISI5 / INF2 - Developpement d'Interface

 */

package review.ihm.volumeapplication;

import java.util.ArrayList;
import java.util.List;

/**
 * Example d'usage du modele MVC
 * Cette classe definit le controler pour modele de volume.
 *
 * Example base sur Baptiste Wicht, <b>"Implémentation du pattern MVC"</b>,
 * 24 Avril 2007 (derniere visite le 29 Janvier 2009).
 * @see http://baptiste-wicht.developpez.com/tutoriel/conception/mvc/
 * @author kirsch
 */
public class VolumeController {

   public VolumeController(VolumeModel model) {
      this.model = model;
      this.views = new ArrayList<VolumeView>();
   }
```

```
  public VolumeModel getModel() {
     return model;
  }

  public void setModel(VolumeModel model) {
     this.model = model;
  }

  /**
   * ce methode est utilise par les vue pour informer lors des
   * demandes (de l'utilisateur) de modification du volume.
   * @param newVolume nouveau valeur de volume
   */
  public void notifyVolumeChange(int newVolume) {
     this.model.setVolume(newVolume);
  }

  public void addView (VolumeView view) {
     this.views.add(view);
     this.model.addVolumeListener(view);
  }

  public void removeView (VolumeView view) {
     this.views.remove(view);
     this.model.removeVolumeListener(view);
   }

  public void displayViews() {
     for (VolumeView aView: this.views) {
        aView.display();
     }
   }

  public void closeViews() {
     for (VolumeView aView: this.views) {
        aView.close();
     }
  }

  private VolumeModel model;
  private List<VolumeView> views;
}
```

Les vues

La classe abstraite VolumeView

La classe *VolumeView* détermine un comportement commun à toutes ses sous-classes. Celui-ci consiste à avoir un contrôleur de type *VolumeController*. La manipulation de ce contrôleur est donc définie dans cette classe.

```
/*
 UNIVERSITE PARIS 1 (PANTHEON SORBONNE)
 MIAGE - UFR 27
 Cours ISI5 / INF2 - Developpement d'Interface
```

```
*/

package review.ihm.volumeapplication;

import java.awt.Dimension;

/**
 * Example d'usage du modele MVC
 * Cette classe definit le view pour modele de volume.
 *
 * Example base sur Baptiste Wicht, <b>"Implémentation du pattern MVC"</b>,
 * 24 Avril 2007 (derniere visite le 29 Janvier 2009).
 * @see http://baptiste-wicht.developpez.com/tutoriel/conception/mvc/
 * @author kirsch
 */
public abstract class VolumeView implements VolumeModelListener {

   public VolumeView(VolumeController controller) {
      this.controller = controller;
      this.controller.addView(this);
   }

   public final VolumeController getController(){
                  return controller;
            }

   public abstract void display();
   public abstract void close();
   public abstract void setLocation(int x, int y);
   public abstract void setSize(Dimension d);
   public abstract Dimension getSize();

   private VolumeController controller = null;
}
```

La classe VolumeListView

La classe *VolumeListView* est une implémentation de *VolumeView* proposant une vue « passive », qui ne permet pas la modification du volume. La méthode *buildFrame* se charge de construire la partie visible de l'interface avec l'utilisateur. La méthode *volumeChange* se charge de mettre à jour la liste, en ajoutant la nouvelle valeur correspondant au volume actuel.

```
/*
 UNIVERSITE PARIS 1 (PANTHEON SORBONNE)
 MIAGE - UFR 27
 Cours ISI5 / INF2 - Developpement d'Interface
*/

package review.ihm.volumeapplication;


import java.awt.BorderLayout;
import java.awt.Dimension;
import javax.swing.DefaultListModel;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.ListModel;
```

```
/**
 * Example d'usage du modele MVC
 * Cette classe definit une vue basé sur une liste de valeurs
 * pour un modele de volume
 *
 * Example base sur Baptiste Wicht, <b>"Implémentation du pattern MVC"</b>,
 * 24 Avril 2007 (derniere visite le 29 Janvier 2009).
 * @see http://baptiste-wicht.developpez.com/tutoriel/conception/mvc/
 * @author kirsch
 */
public class VolumeListView extends VolumeView {

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

      this.buildFrame(volume);
   }

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

   private void buildFrame(int volume) {
      this.jFrame = new JFrame();
      jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       /* on definit le layoutmanager avec 2x plus d'espace vert. et horiz. */
      BorderLayout border = new BorderLayout();
      border.setHgap(border.getHgap() * 2);
      border.setVgap(border.getVgap() * 2);
      jFrame.setLayout(border);

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

      jVolumeList = new JList (jListModel);

      JScrollPane scrollPane = new JScrollPane (jVolumeList);
      jFrame.add(scrollPane, BorderLayout.CENTER);

      jFrame.pack();
   }

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

   @Override
   public void close() {
      this.jFrame.setVisible(false);
   }

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

   @Override
   public void setLocation(int x, int y) {
```

```
        this.jFrame.setLocation(x, y);
    }

    @Override
    public void setSize(Dimension d) {
        this.jFrame.setSize(d);
    }

    @Override
    public Dimension getSize() {
        return this.jFrame.getSize();
    }

    JFrame jFrame;
    JList jVolumeList;
    DefaultListModel jListModel;
}
```

La classe VolumeProgressView

La classe *VolumeProgressView* est également une vue « passive ». Elle montre uniquement une barre de progression à la verticale représentant le volume actuel. La méthode *volumeChanged* va ainsi mettre à jour la barre à chaque fois qu'une nouvelle valeur de volume est reçue.

```
/*
 UNIVERSITE PARIS 1 (PANTHEON SORBONNE)
 MIAGE - UFR 27
 Cours ISI5 / INF2 - Developpement d'Interface
*/

package review.ihm.volumeapplication;

import java.awt.Dimension;
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JProgressBar;
import javax.swing.SwingConstants;

/**
 * Example d'usage du modele MVC
 * Cette classe definit une vue basé sur une barre de progression qui
 * permet de visualiser la valeur du modele de volume
 *
 * @author kirsch
 */
public class VolumeProgressView extends VolumeView {
    private JFrame jFrame;
    private JProgressBar jProgress;


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

    public VolumeProgressView(VolumeController controller, int volume) {
        super(controller);
        this.buildFrame(volume);
    }
```

```
    private void buildFrame(int volume) {
        this.jFrame = new JFrame();
        jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        jFrame.setLayout(new FlowLayout());

        jProgress = new JProgressBar (0, 100);
        jProgress.setOrientation(SwingConstants.VERTICAL);
        jProgress.setValue(volume);

        jFrame.add(jProgress);

        jFrame.pack();
    }

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

    @Override
    public void close() {
        this.jFrame.setVisible(false);
    }

    @Override
    public void setLocation(int x, int y) {
        this.jFrame.setLocation(x, y);
    }

    @Override
    public void setSize(Dimension d) {
        this.jFrame.setSize(d);
    }

    @Override
    public Dimension getSize() {
        return this.jFrame.getSize();
    }


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

}
```

La classe VolumeSliderView

La classe *VolumeSliderView* représente une vue « active », qui permet le changement de la valeur du volume à travers une barre glissante. Elle va donc devoir gérer non seulement la mise à jour de la barre, à travers la méthode *volumeChanged*, mais également les actions de l'utilisateur. Celles-ci sont gérées par des événements de type *ActionEvent*, avec un *ActionListener* anonyme. A chaque occurrence d'un événement de ce type, l'écouteur anonyme se charge d'invoquer la méthode *notifyVolumeChange*, laquelle invoque, à son tour, le contrôleur.

```
/*
UNIVERSITE PARIS 1 (PANTHEON SORBONNE)
MIAGE - UFR 27
Cours ISI5 / INF2 - Developpement d'Interface
*/
package review.ihm.volumeapplication;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JSlider;
import javax.swing.SwingConstants;

/**
 * Example d'usage du modele MVC
 * Cette classe definit une vue basé sur un "Slider", une "barre" deroulante qui
 * pour visualiser et modifier la valeur du modele de volume
 *
 * @author kirsch
 */
public class VolumeSliderView extends VolumeView {

    JFrame jFrame;
    JSlider jSlider;

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

    public VolumeSliderView(VolumeController controller, int volume) {
        super(controller);
        this.buildFrame(volume);
    }

    private void buildFrame(int volume) {
        this.jFrame = new JFrame();
        jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        /* on definit le layoutmanager avec 2x plus d'espace vert. et horiz. */
        BorderLayout border = new BorderLayout();
        border.setHgap(border.getHgap() * 2);
        border.setVgap(border.getVgap() * 2);
        jFrame.setLayout(border);

        //JSlider(int min, int max, int value)
        this.jSlider = new JSlider(0, 100, volume);
        this.jSlider.setOrientation(SwingConstants.HORIZONTAL);

        JButton jButton = new JButton("Change");
        jButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                notifyVolumeChange();
            }
        });

        jFrame.add(jSlider, BorderLayout.CENTER);
        jFrame.add(jButton, BorderLayout.SOUTH);
```

```
        jFrame.pack();
    }

    private void notifyVolumeChange() {
        this.getController().notifyVolumeChange(jSlider.getValue());
    }

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

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

    @Override
    public void close() {
        this.jFrame.setVisible(false);
    }

    @Override
    public void setLocation(int x, int y) {
        this.jFrame.setLocation(x, y);
    }

    @Override
    public void setSize(Dimension d) {
        this.jFrame.setSize(d);
    }

    @Override
    public Dimension getSize() {
        return this.jFrame.getSize();
    }
}
```

La classe VolumeSpinnerView

La classe *VolumeSpinnerView* est similaire à la précédente, une vue active, qui utilise un composant de type *Spinner* pour l'affichage et le changement du volume.

```
/*
UNIVERSITE PARIS 1 (PANTHEON SORBONNE)
MIAGE - UFR 27
Cours ISI5 / INF2 - Developpement d'Interface
*/
package review.ihm.volumeapplication;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
```

```
import javax.swing.JFrame;
import javax.swing.JSpinner;
import javax.swing.SpinnerNumberModel;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

/**
 * Example d'usage du modele MVC
 * Cette classe definit une vue basé sur un "Spinner" (textfield contenant
 * des boutons pour augmenter/reduire la valeur) pour visualiser et
 * modifier la valeur du modele de volume
 *
 * Example base sur Baptiste Wicht, <b>"Implémentation du pattern MVC"</b>,
 * 24 Avril 2007 (derniere visite le 29 Janvier 2009).
 * @see http://baptiste-wicht.developpez.com/tutoriel/conception/mvc/
 * @author kirsch
 */
public class VolumeSpinnerView extends VolumeView {

   private JFrame jFrame;
   private SpinnerNumberModel spinnerModel;
   private JSpinner jSpinner;

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

   public VolumeSpinnerView(VolumeController controller, int volume) {
      super(controller);
      this.buildFrame(volume);
   }

   private void buildFrame(int volume) {
      this.jFrame = new JFrame();
      jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      /* on definit le layoutmanager avec 2x plus d'espace vert. et horiz. */
      BorderLayout border = new BorderLayout();
      border.setHgap(border.getHgap() * 2);
      border.setVgap(border.getVgap() * 2);
      jFrame.setLayout(border);

      //SpinnerNumberModel (val, min, max, step);
      spinnerModel = new SpinnerNumberModel(volume, 0, 100, 1);
      jSpinner = new JSpinner(spinnerModel);

      JButton jButton = new JButton("Change");
      jButton.addActionListener(new ActionListener() {

         public void actionPerformed(ActionEvent e) {
            // notifyVolumeChange(e); //solution alternative
            notifyVolumeChange();
         }
      });

      jFrame.add(jSpinner, BorderLayout.CENTER);
      jFrame.add(jButton, BorderLayout.SOUTH);

      jFrame.pack();
   }
```

```
// solution alternative
/* private void notifyVolumeChange(ChangeEvent e) {
JSpinner spinner = (JSpinner)e.getSource();
int nv = spinner.getModel().getNumber().intValue();
this.getController().notifyVolumeChange(nv);
}
 */

private void notifyVolumeChange() {
    int newVolume = this.spinnerModel.getNumber().intValue();
    this.getController().notifyVolumeChange(newVolume);
}

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

@Override
public void close() {
    this.jFrame.setVisible(false);
}

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

@Override
public void setLocation(int x, int y) {
    this.jFrame.setLocation(x, y);
}

@Override
public void setSize(Dimension d) {
    this.jFrame.setSize(d);
}

@Override
public Dimension getSize() {
    return this.jFrame.getSize();
}
}
```

L'application : la classe VolumeApplication

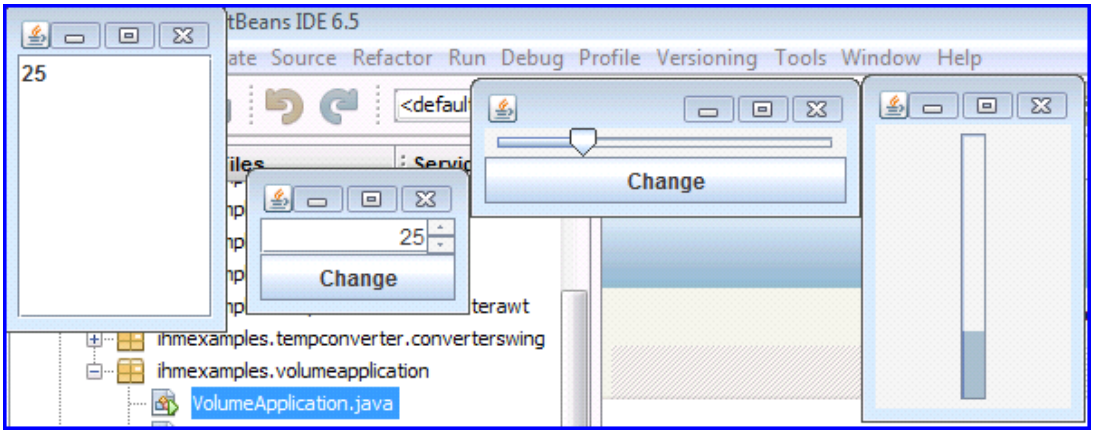
L'application *VolumeApplication* est la responsable par l'instanciation de tous les objets, du modèle aux vues. C'est lui qui se chargera de transmettre au contrôleur son objet modèle, et aux vues, leur contrôleur.

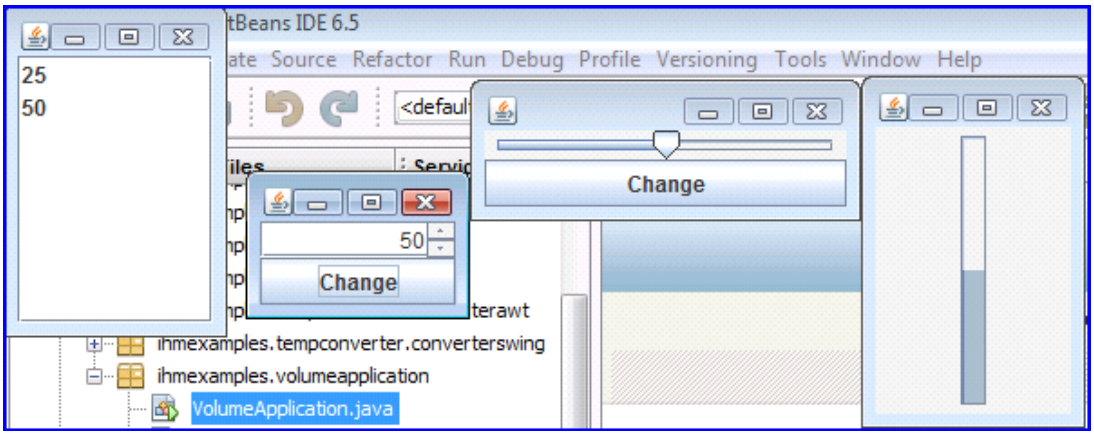
```
/*
UNIVERSITE PARIS 1 (PANTHEON SORBONNE)
MIAGE - UFR 27
Cours ISI5 / INF2 - Developpement d'Interface
*/
package review.ihm.volumeapplication;
```

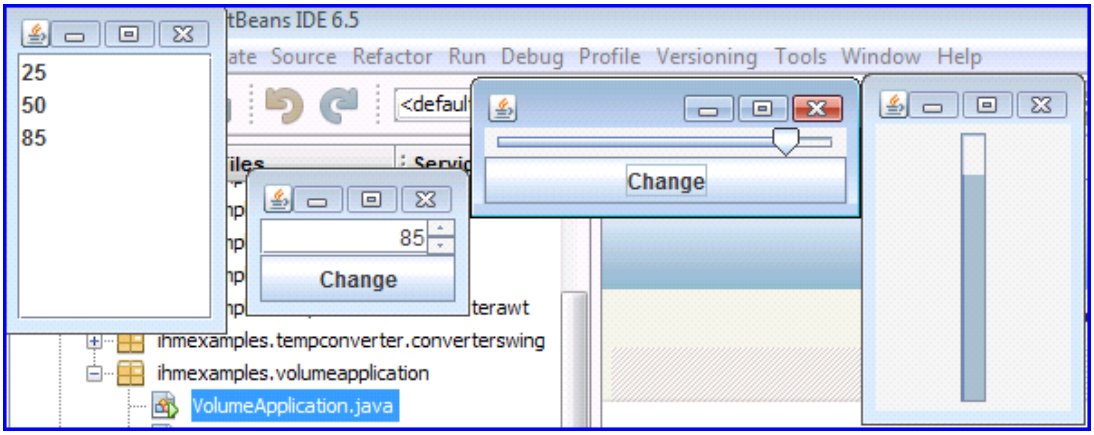
```
public class VolumeApplication {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        VolumeModel model = new VolumeModel(25);

        VolumeController controller1 = new VolumeController(model);

        VolumeListView listView = new VolumeListView(controller1, model.getVolume());
        VolumeSpinnerView spinnerView = new VolumeSpinnerView(controller1, model.getVolume());
        VolumeSliderView sliderView = new VolumeSliderView(controller1, model.getVolume());
        VolumeProgressView progressView = new VolumeProgressView(controller1, model.getVolume());

        spinnerView.setLocation((int) listView.getSize().getWidth() + 10,
                (int) (listView.getSize().getHeight() / 2));
        sliderView.setLocation((int) (listView.getSize().getWidth() +
                spinnerView.getSize().getWidth() + 10),
                (int) (spinnerView.getSize().getHeight() / 2));
        progressView.setLocation ((int) (listView.getSize().getWidth() +
                spinnerView.getSize().getWidth() +
                sliderView.getSize().getWidth() + 10),
                (int) (sliderView.getSize().getHeight() / 2));

        controller1.displayViews();
    }
}
```

Le résultat

Les écrans ci-dessous illustrent le résultat final de l'application, avec ces quatre vues et dans laquelle tout changement effectué sur une vue est répercuté sur les autres.







Références

1. Java BluePrints, « Model-View-Controller ». Disponible sur : http://java.sun.com/blueprints/patterns/MVC-detailed.html (dernière visite Janvier 2009)
2. Wicht, B., « Implémentation du pattern MVC ». Disponible sur : http://baptiste-wicht.developpez.com/tutoriel/conception/mvc/ (dernière visite 29 Janvier 2009).
3. Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J., « Design Patterns: Elements of reusable object-oriented software », Adison-Wesley, 1994