ISI5
Développement d'interfaces Homme-Machine

Manuele Kirsch Pinheiro
Maître de conférences en Informatique
Centre de Recherche en Informatique
Université Paris 1 – Panthéon Sorbonne
Manuele.Kirsch-Pinheiro@univ-paris1.fr
http://mkirschp.free.fr

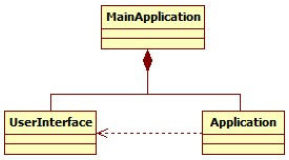
Présentation

- **Contenu prévisionnel**
  - Modèle MVC
  - MVC & Swing

Modèle MVC

Limites des 2 couches

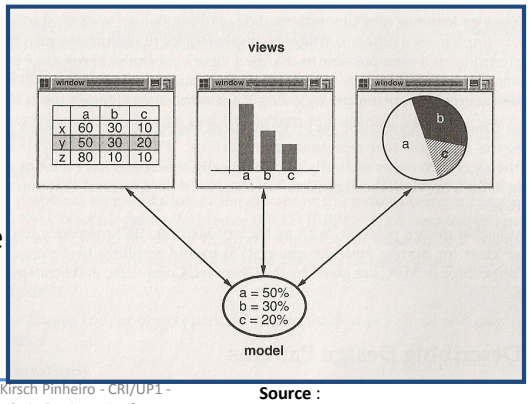
- Dépendances entre les couches
- Évolution des éléments difficile
  - Évolution de l'interface
    - *Ex. : migration AWT - Swing*
  - Évolution du comportement de l'interface
    - *Ex. : vérification des données après une action*
  - Évolution de la couche application
    - *Ex. : nouvelles fonctionnalité dans la couche métier*
  - Évolution du modèle de données
    - *Ex. : changement de base des données*



Réécriture

Modèle MVC

- Model – View – Control (MVC)
- Origines aux années 80
  - Développement d'interfaces en Smalltalk
- Objectif :
  - Séparation interface et application (métier)
  - Modifications dans un élément n'entraine pas des modifications sur les autres



Source :
Gamma *et al.* *Design patterns*, 94

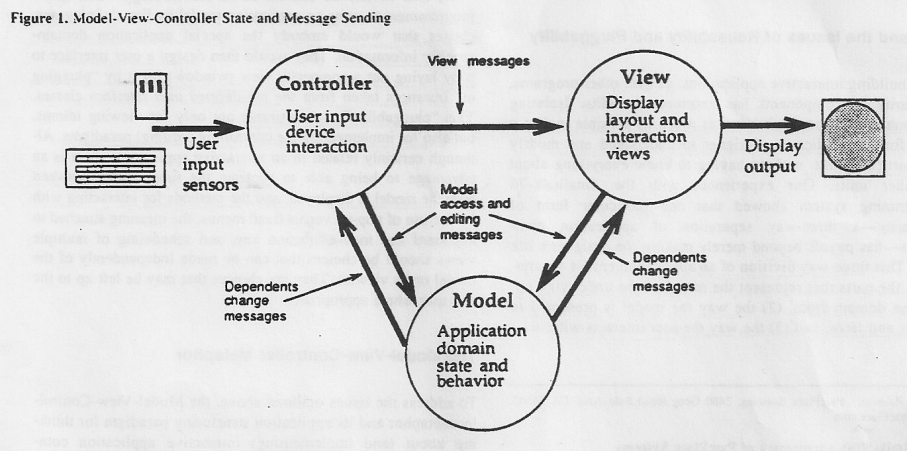
Modèle MVC

- Éléments (participants) :
  - **Model** :
    - « *application object* » (Gamma et al., 94)
    - « *the domain-specific software simulation or implementation of the application's central structure* » (Krasner & Pope, 88)
    - **Les composants qui font réellement le travail (métier)**
  - **View** :
    - « *screen presentation* » (Gamma et al., 94)
    - « *views deal with everything graphical*: *they request data from their model and display the data* » (Krasner & Pope, 88)
    - **La représentation graphique**
  - **Controller** :
    - « *the way user interface reacts to user input* » (Gamma et al., 94)
    - « *controllers […] provide the interface between the model with its associated views* » (Krasner & Pope, 88)
    - **Le lien entre la vue et le modèle**

Modèle MVC

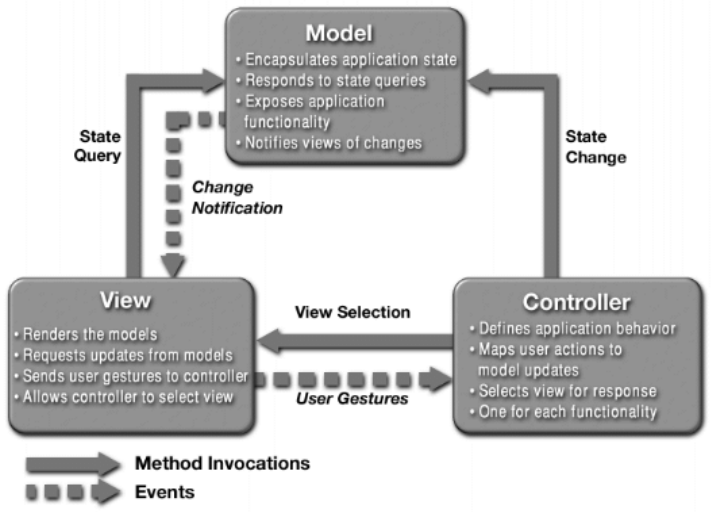
- **Division du travail entre le modèle, la vue et le contrôle**
  - Le modèle représente les données et la logique métier
  - Le modèle peut être représenté visuellement de différentes manières, par différentes vues
  - Les vues doivent être synchronisées
  - Les modes d'interaction peuvent changer en fonction de la vue
- **Le modèle est totalement indépendant des pairs vue/contrôle**
- **À chaque vue correspond un contrôle**
  - Un modèle peut avoir plusieurs pairs vue / contrôle
- **La vue doit assurer que la présentation suit l'état du modèle**
  - Si l'état du modèle change, celui-ci doit notifier les pairs vue / contrôle
  - Les modifications sur l'état du modèle impliquent la mise à jour de la vue
- **Mécanisme de *subscribe/notify* entre les éléments**

MVC à l'origine



Source : G.E. Krasner, S.T. Pope, *A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80,* Journal of Object-Oriented Programming (JOOP), vol. 1, n° 3 (Aug/Sep 1988), pp. 26 - 49.

MVC par Sun



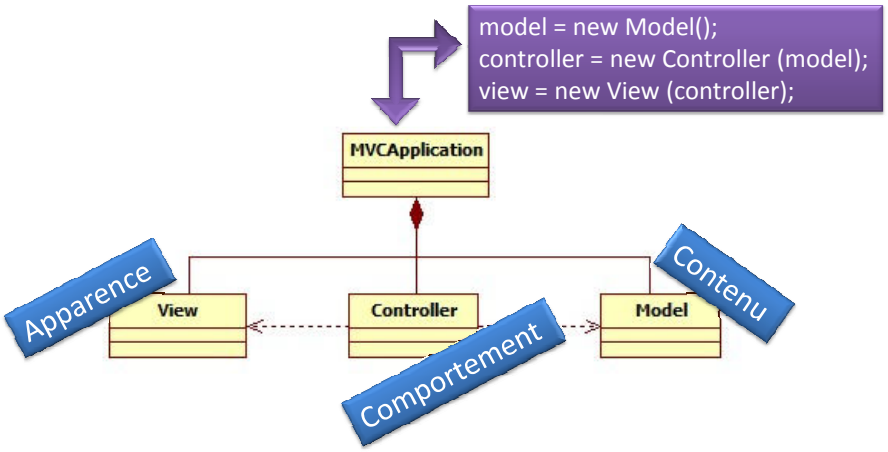
Source : Java BluePrints, *Model-View-Controller*
http://java.sun.com/blueprints/patterns/MVC-detailed.html

MVC par Sun

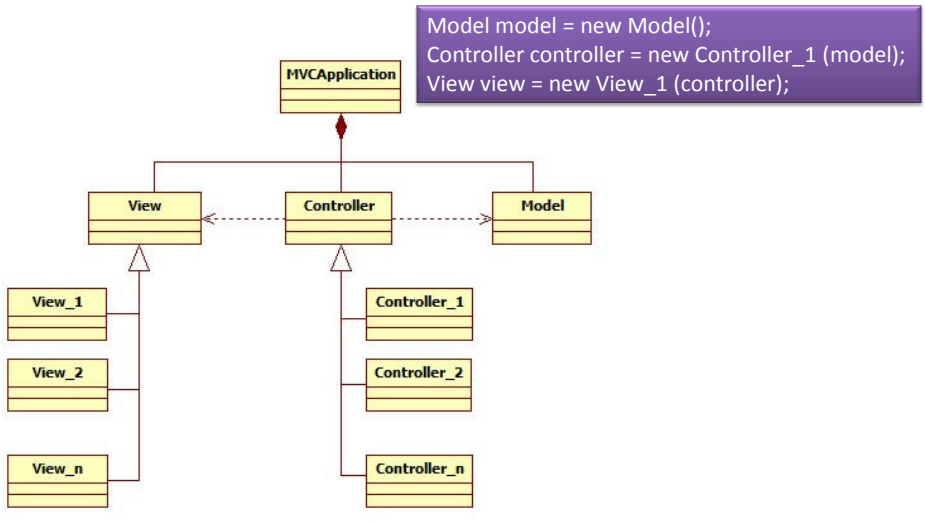
- *By applying the Model-View-Controller (MVC) architecture […], you separate core **business model** functionality from the **presentation** and **control logic** that uses this functionality*
- ***Model** - The model represents enterprise data and the business rules that govern access to and updates of this data.*
- ***View** -The view renders the contents of a model.*
- ***Controller** - The controller translates interactions with the view into actions to be performed by the model.*

Source : Java BluePrints, *Model-View-Controller*
http://java.sun.com/blueprints/patterns/MVC-detailed.html

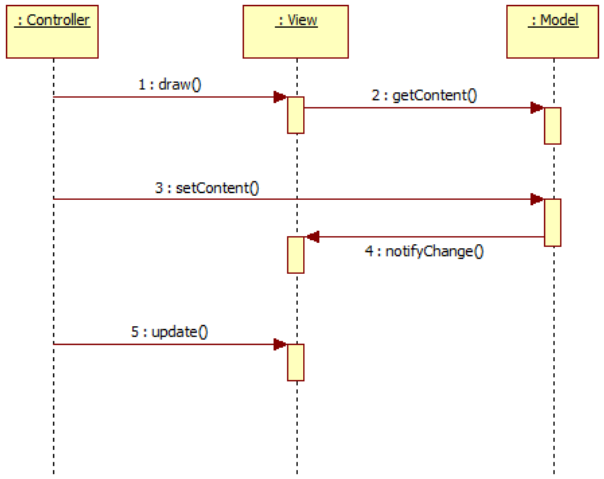
Une application selon MVC



Une application selon MVC

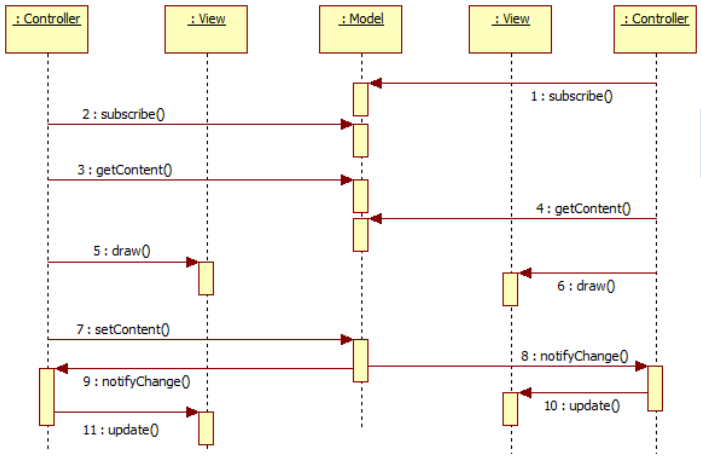


Interactions



Toute modification sur l'état du modèle déclenche une notification

Interactions



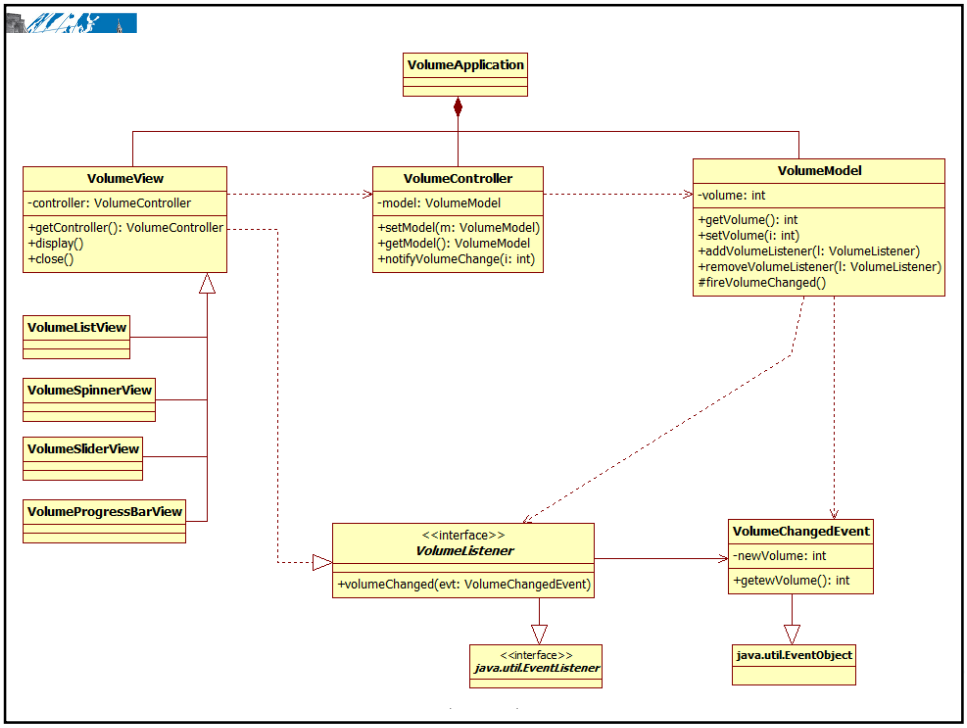
Avec une vue ⊕ indépendante

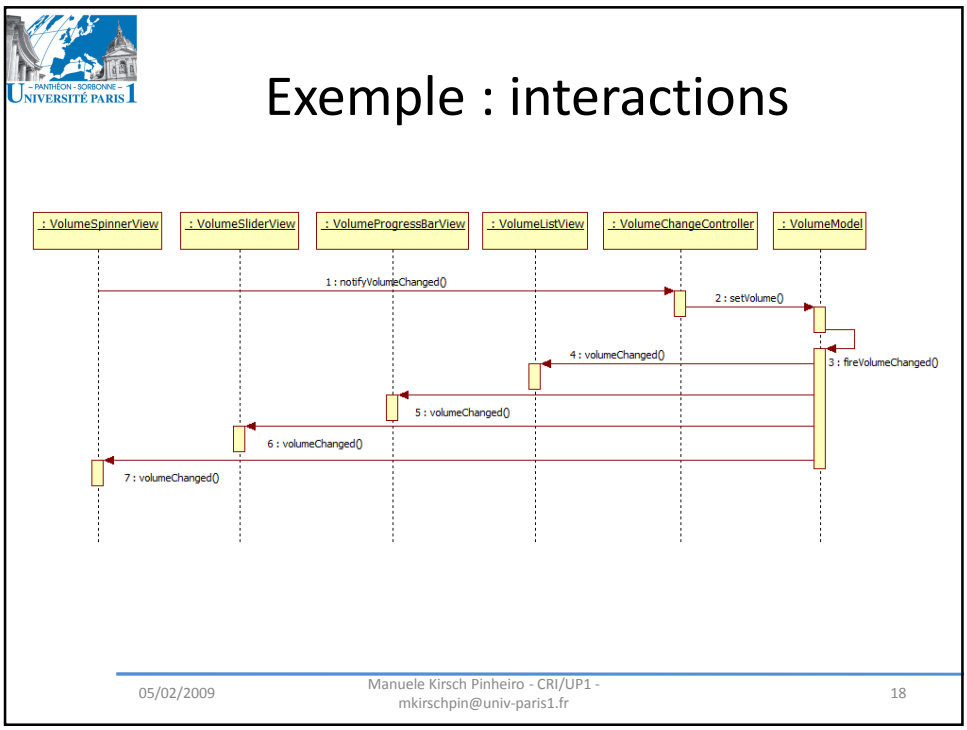
Avantages MVC

- Évolution / réutilisation
- Possibilité d'avoir différentes représentations
  - Une même application sur multiples plateformes
- Changement dynamique de la représentation et du comportement devient possible
  - Multi-modalité
- Adaptation / Personalisation

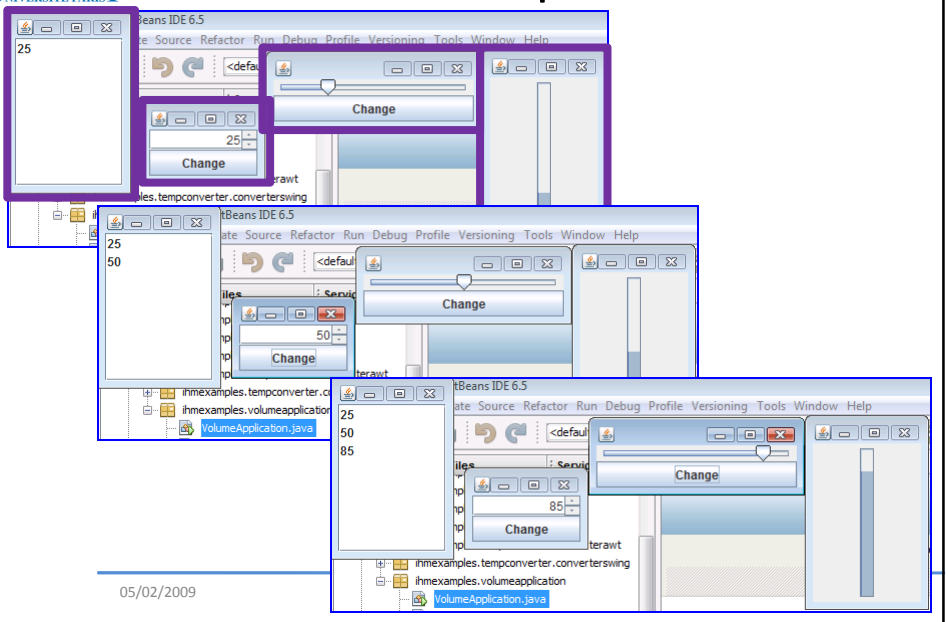
Exemple

- Gestionnaire de volume
  - Modèle : un volume stocké
  - Vue : différents indicateurs de ce volume
  - Contrôle : différents contrôle selon le type de vue
- Inspiré de l'exemple de Baptiste Wicht, « Implémentation du pattern MVC » sur Developpez.com
  - http://baptiste-wicht.developpez.com/tutoriel/conception/mvc/


VolumeApplication
VolumeView
-controller: VolumeController
+getController(): VolumeController
+display()
+close()
VolumeController
-model: VolumeModel
+setModel(m: VolumeModel)
+getModel(): VolumeModel
+notifyVolumeChange(i: int)
VolumeModel
-volume: int
+getVolume(): int
+setVolume(i: int)
+addVolumeListener(l: VolumeListener)
+removeVolumeListener(l: VolumeListener)
#fireVolumeChanged()
VolumeListView
VolumeSpinnerView
VolumeSliderView
VolumeProgressBarView
<<interface>>
VolumeListener
+volumeChanged(evt: VolumeChangedEvent)
VolumeChangedEvent
-newVolume: int
+getewVolume(): int
<<interface>>
java.util.EventListener
java.util.EventObject


Exemple : interactions
: VolumeSpinnerView
: VolumeSliderView
: VolumeProgressBarView
: VolumeListView
: VolumeChangeController
: VolumeModel
1 : notifyVolumeChanged()
2 : setVolume()
3 : fireVolumeChanged()
4 : volumeChanged()
5 : volumeChanged()
6 : volumeChanged()
7 : volumeChanged()
05/02/2009
Manuele Kirsch Pinheiro - CRI/UP1 - mkirschpin@univ-paris1.fr
18

Exemple



Exemple : le modèle

```
public void setVolume(int volume) {
    this.volume = volume;
    //on modifie l'etat du modele, on averti les listeners
    fireVolumeChanged();
}
```

- **Classe VolumeModel**
- Attributs :
  - int volume : le volume stocké
  - EventListenerList listeners : les *listeners* abonnés
- Méthodes :
  - getVolume : récupère la valeur du volume
  - **setVolume** : modifie la valeur du volume et déclenche *fireVolumeChanged*
  - **fireVolumeChanged** : notification des *listeners*
  - addVolumeListener / removeVolumeListener : abonnement / désabonnement d'un *listener*

Exemple : le modèle

- **Classe Volu**
- Attributs :
  - int volume
  - EventListenerList listeners : les *listeners* abonnés
- Méthodes :
  - getVolume : récupère la valeur du volume
  - **setVolume** : modifie la valeur du volume et déclenche *fireVolumeChanged*
  - **fireVolumeChanged** : notification des *listeners*
  - addVolumeListener / removeVolumeListener : abonnement / désabonnement d'un *listener*

```
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()));
    }
}
```


Exemple : le modèle

- **Classe VolumeModel**
- Attributs :
  - int volume : le volume s
  - EventListenerList listeners : les *listeners* abonnés
- Méthodes :
  - getVolume : récupère la valeur du volume
  - **setVolume** : modifie la valeur du volume et déclenche *fireVolumeChanged*
  - **fireVolumeChanged** : notification des *listeners*
  - addVolumeListener / removeVolumeListener : abonnement / désabonnement d'un *listener*

```
/**...*/
public void addVolumeListener(VolumeModelListener listener) {
    listeners.add(VolumeModelListener.class, listener);
}

/**...*/
public void removeVolumeListener(VolumeModelListener l) {
    listeners.remove(VolumeModelListener.class, l);
}
```

Exemple : Les événements et les écouteurs

Événements déclenchés à chaque modification du modèle

- **Classe VolumeChangedEvent**
- Extends : **EventObject**
- Attributs :
  - int **newVolume** : nouvelle valeur de modèle
- Constructeur :
  - VolumeChangedEvent (Object source, int newVolume)
- Méthodes :
  - **getNewVolume** : récupère le nouveau valeur du volume

Écouteurs (listeners) abonnés aux événements

- **Interface VolumeModelListener**
- Extends : **EventListener**
- Méthodes :
  - volumeChanged (VolumeChangedEvent event) : *notification* occurrence de l'événement

Exemple : Les événements et les écouteurs

Événements déclenchés à chaque modification du modèle

- **Classe VolumeChangedEvent**
- Extends : **EventObject**
- Attributs :
  - int **newVolume** : nouvelle valeur de modèle
- Constructeur :
  - VolumeChangedEvent (Object source, int newVolume)
- Méthodes :
  - **getNewVolume** : récupère le nouveau valeur du volume

Écouteurs (listeners) abonnés aux événements

- **Interface VolumeModelListener**
- Extends : **EventListener**
- Méthodes :
  - volumeChanged (VolumeChangedEvent event) : *notification* occurrence de

```
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;
}
```

Exemple : Les événements et les écouteurs

Événements déclenchés à chaque modification du modèle

- **Classe VolumeChangedEvent**
- Extends : **EventObject**
- Attributs :
  - int **newVolume** : nouvelle valeur de modèle
- Constructeur :
  - VolumeChangedEvent (Object source, int newVolume)
- Méthode
  - **getNe**
    nouveau valeur du volume

Écouteurs (listeners) abonnés aux événements

- **Interface VolumeModelListener**
- Extends : **EventListener**
- Méthodes :
  - volumeChanged (VolumeChangedEvent event) : *notification* occurrence de l'événement

```
public interface VolumeModelListener extends java.util.EventListener {
    public void volumeChanged(VolumeChangedEvent event);
}
```


Exemple : le contrôleur

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

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

- **Classe VolumeController**
- Attributs :
  - VolumeModel **model** : le lien vers le *modèle*
  - List<VolumeView> views : ensemble de *vues* dont s'occupe le contrôler
- Méthodes :
  - getModel / setModel
  - displayViews / closeViews
  - **addView / removeView** : ajoute / enlève une vue de la liste internet et *adonne* / *désabonne* la vue au *modèle*
  - **notifyVolumeChange** : informe le modèle lorsque les données ont été modifiées par l'utilisateur

Exemple : l

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

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

- **Classe VolumeController**
- Attributs :
  - VolumeModel **model** : le lien vers l
  - List<VolumeView> views : ensemble de *vues* dont s'occupe le contrôler
- Méthodes :
  - getModel / setModel
  - displayViews / closeViews
  - **addView / removeView** : ajoute / enlève une vue de la liste internet et *adonne* / *désabonne* la vue au *modèle*
  - **notifyVolumeChange** : informe le modèle lorsque les données ont été modifiées par l'utilisateur

Exemple : le contrôleur

```
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);
 }
```

- **Classe VolumeController**
- Attributs :
  - VolumeModel **model** : le lien
  - List<VolumeView> views : en
- Méthodes :
  - getModel / setModel
  - displayViews / closeViews
  - **addView / removeView** : ajoute / enlève une vue de la liste internet et *adonne* / *désabonne* la vue au *modèle*
  - **notifyVolumeChange** : informe le modèle lorsque les données ont été modifiées par l'utilisateur

Exemple : le contrôleur

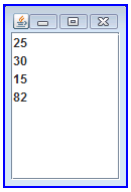
- **Classe VolumeCont**
- Attributs :
  - VolumeModel **model** : le lien vers le *modèle*
  - List<VolumeView> views : ensemble de *vues* dont s'occupe le contrôler
- Méthodes :
  - getModel / setModel
  - displayViews / closeViews
  - **addView / removeView** : ajoute / enlève une vue de la liste internet et *adonne* / *désabonne* la vue au *modèle*
  - **notifyVolumeChange** : informe le modèle lorsque les données ont été modifiées par l'utilisateur

```
public void notifyVolumeChange(int newVolume) {
    this.model.setVolume(newVolume);
}
```


Exemple : la vue abstraite

- **Classe Abstract VolumeView**
- Implements **VolumeModelListener**
- Attributs :
  - **VolumeController controller** : le contrôleur responsable par cette vue
- Constructeurs :
  - **VolumeView(VolumeController controller)**
- Méthodes :
  - **VolumeController getController**
  - abstract void display
  - abstract void close
  - abstract void setLocation
  - abstract void setSize
  - abstract Dimension getSize

Exemple : les vues concrètes



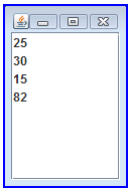
- Vue « passive » : uniquement visualisation
- **Classe VolumeListView**
- Extends **VolumeView**
- Attributs :
  - JFrame jFrame : fenêtre de visualisation
  - JList jVolumeList : liste (JList) contenant l'historique du volume
  - DefaultListModel jListModel
- Méthodes (liste non-exhaustive) :
  - **buildFrame** : construction de la fenêtre
  - **volumeChanged** : notification d'une modification dans le modèle (interface VolumeModelListener)

Exemple : les vues concrètes

- Vue « passive » :
- **Classe VolumeLi**
- Extends **Volume**
- Attributs :
  - JFrame jFrame :
  - JList jVolumeList
  - DefaultListMode
- Méthodes (liste
  - **buildFrame** : co
  - **volumeChange**
    (interface Volum

```
public class VolumeListView extends VolumeView {

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

    public VolumeListView(VolumeController controller) {...}

    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();
    }
```

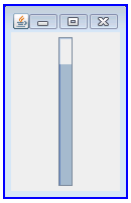
Exemple : les vues concrètes



- Vue « passive » : uniquement visualisation
- **Classe VolumeListView**
- Extends **VolumeView**
- Attributs :
  - JFrame jFrame : fenêtre de visualisation
  - JList jVolumeList : liste (JList) contenant l'historique du volume
  - DefaultListModel jListModel
- Méthodes (liste non-exhaustive) :
  - **buildFrame** : construction de la fenêtre
  - **volumeChanged** : notification d'une modification dans le modèle (interface VolumeModelListener)

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


Exemple : les vues concrètes



- Vue « passive » : uniquement visualisation
- **Classe VolumeProgressView**
- Extends **VolumeView**
- Attributs :
  - JFrame jFrame : fenêtre de visualisation
  - JProgressBar jProgress : barre de progression pour visualiser l'évolution du volume
- Méthodes (liste non-exhaustive) :
  - **buildFrame** : construction de la fenêtre
  - **volumeChanged** : notification d'une modification dans le modèle (interface VolumeModelListener)

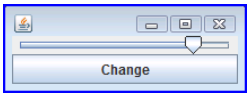
Exemple : les vues concrètes

- Vue « passive » : uniquement visualisation
- **Classe Volum**
- Extends **Volu**
- Attributs :
  - JFrame jFra
  - JProgressBa
    l'évolution d
- Méthodes (lis
  - **buildFrame**
  - **volumeChar**
    (interface Vo

```
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();
}

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


Exemple : les vues concrètes

- Vue « active » : permet visualisation et la modification du volume
- **Classe VolumeSliderView**
- Extends **VolumeView**
- Anonymous listener : ActionListener
- Attributs (liste non-exhaustive) :
  - JSlider jSlider : barre permettant de visualiser et modifier le volume
- Méthodes (liste non-exhaustive) :
  - buildFrame : construction de la fenêtre
  - **volumeChanged** : notification d'une modification dans le modèle (interface VolumeModelListener)
  - **notifyVolumeChange** : informe le contrôleur d'un changement de valeur effectué par le utilisateur



Exemple : les vues concrètes

- Vue « active » : perm
- **Classe VolumeSlider**
- Extends **VolumeVie**
- Anonymous listener
- Attributs (liste non-e
  - JSlider jSlider : bar
- Méthodes (liste non
  - buildFrame : const
  - **volumeChanged** : (interface Volume
  - **notifyVolumeChar** valeur effectué pa

```
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();
}
```

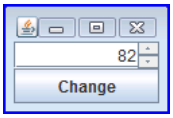

Exemple : les vues concrètes

- Vue « active » : permet visualisation et la modification du volume
- **Classe VolumeSliderView**
- Extends **VolumeView**
- Anonymous list
- Attributs (liste
  - JSlider jSlide
- Méthodes (liste
  - buildFrame : construction de la fenêtre
  - **volumeChanged** : notification d'une modification dans le modèle (interface VolumeModelListener)
  - **notifyVolumeChange** : informe le contrôleur d'un changement de valeur effectué par le utilisateur

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

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

Exemple : les vues concrètes

- Vue « active » : permet visualisation et la modification du volume
- **Classe VolumeSpinnerView**
- Extends **VolumeView**
- Anonymous listener : ActionListener
- Attributs (liste non-exhaustive) :
  - JSpinner jSpinner : zone de texte permettant la visualisation et la modification d'une valeur
  - SpinnerNumberModel spinnerModel
- Méthodes (liste non-exhaustive) :
  - buildFrame : construction de la fenêtre
  - **volumeChanged** : notification d'une modification dans le modèle (interface VolumeModelListener)
  - **notifyVolumeChange** : informe le contrôleur d'un changement de valeur effectué par le utilisateur



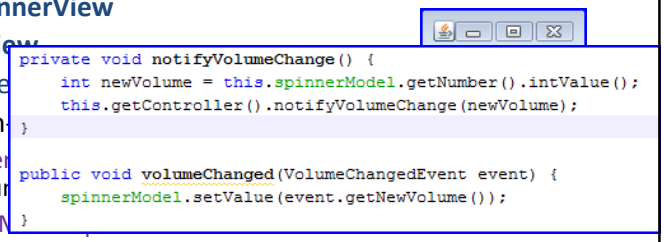
Exemple : les vues concrètes

- Vue « active » : permet visualisation et la modification du volume
- **Classe VolumeSpin**
- Extends **VolumeVie**
- Anonymous listene
- Attributs (liste non
  - JSpinner jSpinner
    modification d'u
  - SpinnerNumberM
- Méthodes (liste no
  - buildFrame : con
  - **volumeChanged**
    (interface Volum
  - **notifyVolumeCh**
    valeur effectué p

```
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();
}
```

Exemple : les vues concrètes

- Vue « active » : permet visualisation et la modification du volume
- **Classe VolumeSpinnerView**
- Extends **VolumeVie**
- Anonymous listene
- Attributs (liste non
  - JSpinner jSpinne
    modification d'u
  - SpinnerNumberM
- Méthodes (liste non-exhaustive) :
  - buildFrame : construction de la fenêtre
  - **volumeChanged** : notification d'une modification dans le modèle (interface VolumeModelListener)
  - **notifyVolumeChange** : informe le contrôleur d'un changement de valeur effectué par le utilisateur



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

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


Exemple : l'application

- Application VolumeApplication
  - Un modèle : VolumeModel model
  - Un contrôleur : VolumeController controller1
  - Plusieurs vues
    - VolumeListView
    - VolumeProgressView
    - VolumeSliderView
    - VolumeSpinnerView

Exemple : l'application

- Application VolumeApplication

```
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();
}
```

Design patterns & MVC

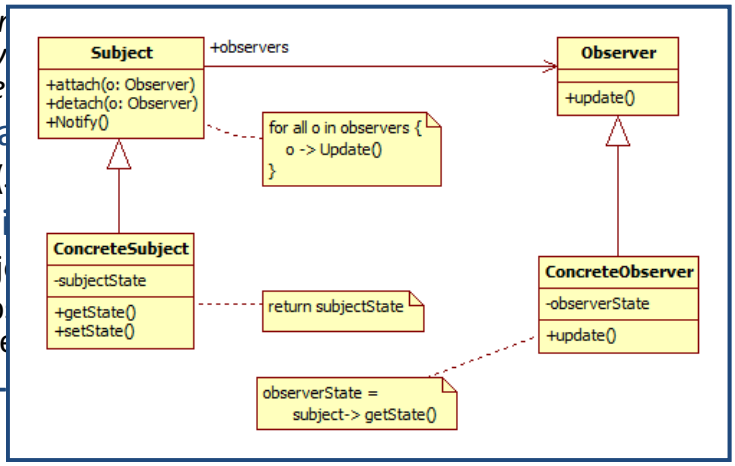
- Le modèle MVC s'inspire des *design patterns*
  - Observer
  - Strategie
  - Composite

Observer

- Intention
  - « *Define a one-to-many dependency between objects so that when one object changes state, all its depends are notified and updated automatically* » (Gamma et al, 94)
- Participants principaux
  - Sujet (*Subject*) et observateurs (Observer)
- Motivation
  - Un sujet possède plusieurs observateurs
  - Les observateurs sont notifié de tout changement d'état du sujet
- **Publish-subscribe model**

Observer

- Intention
  - « *Defi*
    that w
    notifie
- Participa
  - Sujet (
- Motivati
  - Un suj
  - Les ob
    du suj
- **Publish-**

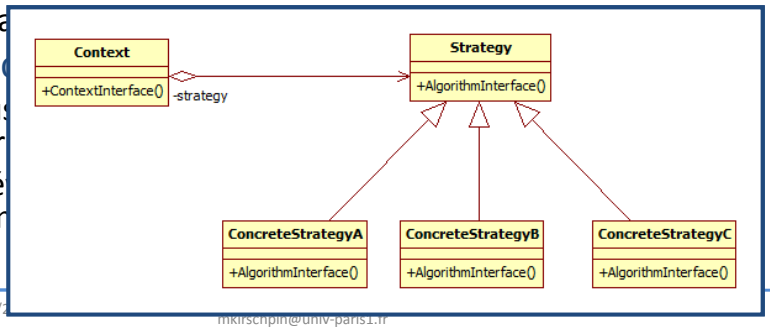


Strategie

- Intention
  - « *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it* » (Gamma et al., 94)
- Participants principaux
  - Stratégie (*Strategy*) et contexte (*Context*)
- Application
  - Plusieurs classes (stratégies) qui diffèrent seulement par leur comportement
  - Prévient l'exposition d'algorithmes et de structures de données complexes

Strategie

- Intention
  - « *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it* » (Gamma et al., 94)
- Participants principaux

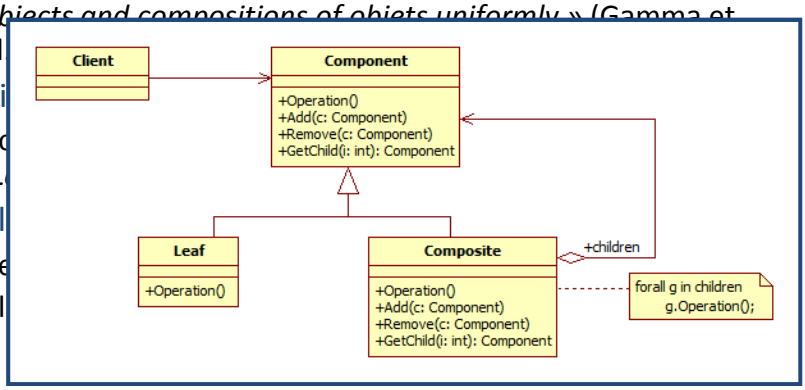


Composite

- Intention
  - « *Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objets uniformly* » (Gamma et al., 94)
- Participants principaux
  - Composition (*Component*), composite (*Composite*) et feuille (*Leaf*)
- Application
  - Représenter les feuilles et les compositions uniformément
  - Clients ne diffèrent pas les composites des objets individuels

Composite

- Intention
  - « *Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objets uniformly* » (Gamma et al
- Parti
  - Co (*L*
- Appl
  - Re
  - Cl



Swing & MVC

Swing & MVC

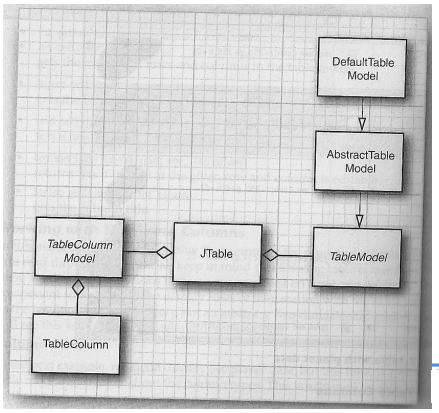
- Le modèle MVC peut s'appliquer à chaque élément du propre modèle
- Chaque composant Swing est conçu selon le modèle MVC
  - Chaque composant contient un modèle et un contrôleur associé
  - Le modèle garde la « valeur » du composant séparée de sa représentation graphique (vue)
    - Ex. bouton : état (pressé ou non), actif ou non, sélectionné ou non, son action (*action command* associé et *non son label*)…
  - La vue gère le *look & feel* du composant indépendamment de son modèle

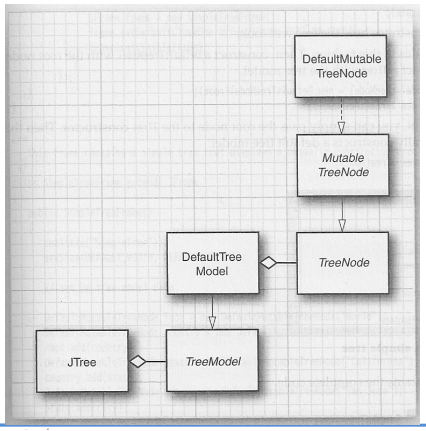
Swing & MVC

- Modèle
  - Interface xxxModel : *ButtonModel, ListModel, SpinnerModel, TableModel, TreeModel…*
  - Implémentation par défaut DefaultxxxModel :
    - DefaultButtonModel
    - DefaultListModel (AbstractListModel), DefaultListSelectionModel
    - SpinnerDateModel, SpinnerListModel, SpinnerNumberModel (AbstractSpinnerModel)
    - DefaultTableModel (AbstractTableModel)
    - DefaultTreeNode, DefaultMutableTreeNode

Swing & MVC

- Le modèle de chaque composant peut être étendu (spécialisation) en fonction des besoins de l'application





Source : Horstmann, Corte Java, vol. II

Exemple

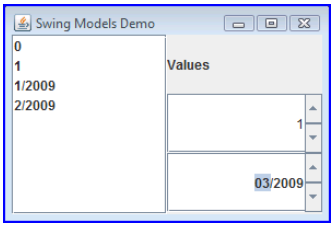
- JList & ListModel :
  - La classe JList n'offre pas de méthode pour l'insertion et la suppression d'une donnée
  - La manipulation des données se fait par le ListModel
  - La classe JList s'occupe uniquement de la présentation des données
- ListModel
  - getElement
  - addListDataListener
  - removeListDataListener

Exemple

- JSpinner & SpinnerModel
  - La manipulation des données se fait par le SpinnerModel
    - SpinnerNumberModel : la manipulation des nombres
    - SpinnerDateModel : manipulation des dates
  - La présentation est gérée par JSpinner

Exemple

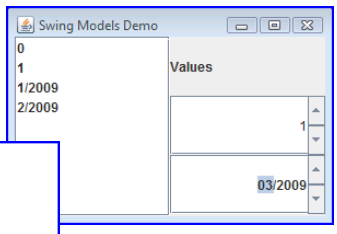
```
private void prepareList() {
    this.lmodel = new DefaultListModel();
    this.lmodel.addElement(new Integer(0));
    this.list = new JList(this.lmodel);
}
```




```
public ListModelDemo() {
    prepareList();
    prepareSpinner();
    prepareFrame();
}
```

```
private void prepareSpinner() {
    this.nummodel = new SpinnerNumberModel();
    this.numSpinner = new JSpinner(this.nummodel);

    this.datemodel = new SpinnerDateModel(Calendar.getInstance().getTime(),
            null, null, Calendar.DAY_OF_MONTH);
    this.dateSpinner = new JSpinner(this.datemodel);
    this.dateSpinner.setEditor(new JSpinner.DateEditor(this.dateSpinner, "MM/yyyy"));

    this.numSpinner.addChangeListener(this);
    this.dateSpinner.addChangeListener(this);
}
```

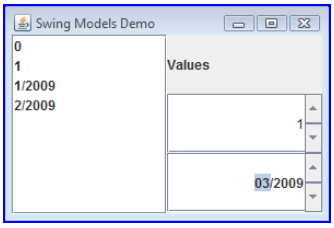

Exemple




```
public void stateChanged(ChangeEvent e) {
    Object obj = e.getSource();

    if (obj == this.numSpinner) {
        this.lmodel.addElement(this.nummodel.getNumber());

    } else if (obj == this.dateSpinner) {
        String nline = this.formatDate(this.datemodel.getDate());

        if (!last.equalsIgnoreCase(nline)) {
            this.lmodel.addElement(nline);
            this.last = nline;
        }
    }
}
```

```
public ListModelDemo() {
    prepareList();
    prepareSpinner();
    prepareFrame();
}
```

Exemple




```
private void prepareFrame() {
    this.frame = new JFrame("Swing Models Demo");
    this.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.frame.setLayout(new GridLayout(1, 2));

    JScrollPane scroll = new JScrollPane(this.list);
    this.frame.add(scroll);

    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(3, 1));
    panel.add(new JLabel("Values"));
    panel.add(this.numSpinner);
    panel.add(this.dateSpinner);
    this.frame.add(panel);

    this.frame.setSize(300, 200);
}
```

```
public ListModelDemo() {
    prepareList();
    prepareSpinner();
    prepareFrame();
}
```


Exercices

Exercices

1) Implémenter le convertisseur de température en MVC
   - Étendre le convertisseur à °C, °F et °Kelvin
   - Proposer au moins 2 vues distinctes

2) Implémenter une application qui affiche une arbre généalogique à l'aide du composant JTree

3) Jeu de scrabble : version MVC
   - Modèle : dictionnaire du données
   - Vue : interface en Swing
   - Contrôle : gestion des interactions