Génie Logiciel, FISE2-INFO4

TD Modèle-Vue-Contrôleur

Frédérique LAFOREST *& Guillaume* MULLER

Exercices d'entraînement à faire durant le TD... et après.

1 Introduction : MVC et `java.util`

Cette section est reprise du TP de Lilian Casiez, Université Lille 1. Il a été modifié et simplifié.
Ce TP est également disponible sur MooTSE, dans la section FISE2 Génie Logiciel. Télécharger le code associé sur MooTSE.

1.1 Objectif

L'objectif est de créer une interface permettant le contrôle de la température en degrés Celsius ou Fahrenheit. L'interface se compose de trois vues représentant la même température sous des formes différentes. La modification d'une des vues doit mettre automatiquement à jour les autres vues. Nous allons voir comment appliquer le modèle MVC pour réaliser cette interface.

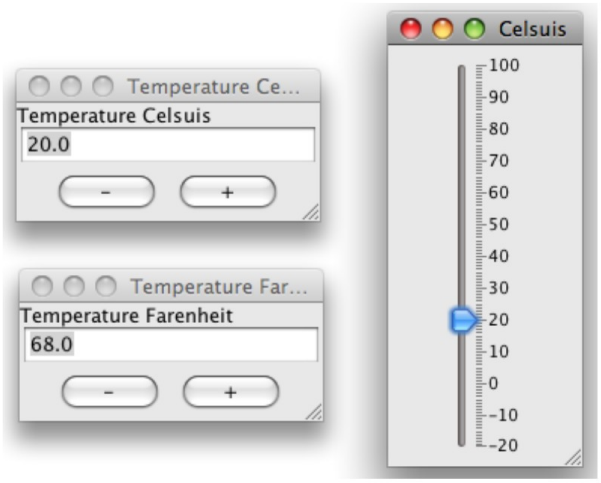

FIG. 1: Trois vues pour un même modèle

1.2 Préambule

Le modèle MVC comprend :

1. **Un modèle**. Celui-ci comprend les données et un certain nombre de méthodes pour les lire et les modifier. Le modèle n'a aucune connaissance de la façon dont les données sont présentées à l'utilisateur (la Vue). Le modèle peut toutefois enregistrer une ou plusieurs vues (une liste de dépendants) qui sont notifiées quand les données du modèle ont subi une modification (patron de conception **Observateur**). En Java, le modèle comprend une ou plusieurs classes de type `java.beans.PropertyChangeSupport` (ou d'une interface ou classe équivalente).
2. **Une ou plusieurs vues**. Une vue fournit une représentation graphique de tout ou partie des données du modèle. Chaque vue obtient les données à afficher en en faisant la demande au modèle. Quand un utilisateur manipule une vue d'un modèle, la vue informe

le contrôleur du changement désiré. En Java, les vues doivent implémenter l'interface `java.beans.PropertyChangeListener` (ou interface équivalente).

3. **Plusieurs contrôleurs**. Les vues sont associées à des contrôleurs qui mettent à jour le modèle. Chaque vue est associée à un unique contrôleur. Le contrôleur interprète les actions de l'utilisateur (par exemple augmenter la température) et appelle les méthodes requises du modèle pour le mettre à jour. Le modèle informe alors toutes les vues qu'un changement est survenu. Ces dernières se mettent à jour. Il peut aussi vérifier les actions de l'utilisateur et modifier la vue en conséquence si besoin.

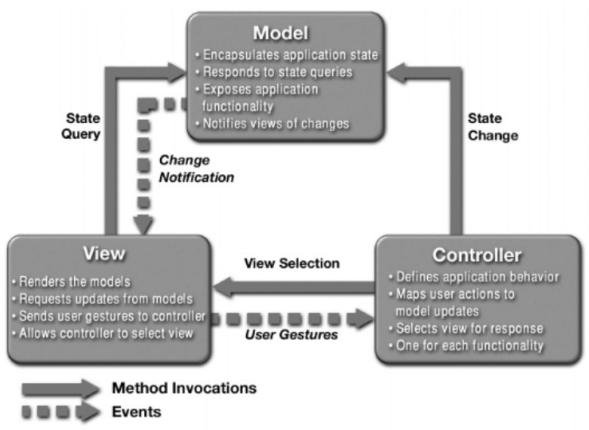

FIG. 2: Le modèle MVC

1.3 Explications de l'organisation MVC du code de l'exemple

1.3.1 Le modèle

Le modèle possède un élément de type `java.beans.PropertyChangeSupport`. Cette classe va gérer pour nous la liste des objets à notifier de changements, ainsi que de *tous* les notifier quand nous le lui demanderons. Nous appellerons `firePropertyChange()` chaque fois que nous modifierons les données du modèle. La méthode `firePropertyChange()` notifie chaque objet enregistré pour recevoir une notification (ici les vues) qu'un changement d'état du modèle a eu lieu. Elle permet également de passer des informations supplémentaires concernant le changement d'état (variante push du patron de conception **Observateur**). La méthode `addPropertyChangeListener()` permet d'enregistrer les objets à notifier dans la liste maintenue par `java.beans.PropertyChangeSupport`. Voici la classe du modèle correspondant à notre interface :

```
package model;

import java.beans.PropertyChangeSupport;
import java.beans.PropertyChangeListener;

public class TemperatureModel {
    private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);
    public void addPropertyChangeListener(PropertyChangeListener listener) {
         this.pcs.addPropertyChangeListener(listener);
     }


    private double temperatureC = 20;
    public double getC () {
        return temperatureC ;
    }
```

```
    public void setC (double tempC) {
        double oldVal = temperatureC;
        temperatureC = tempC ;
        this.pcs.firePropertyChange("temperatureC", oldVal, temperatureC);
    }
    public double getF () {
        return temperatureC * 9.0 / 5.0 + 32.0;
    }
    public void setF (double tempF) {
        double oldVal = temperatureC;
        temperatureC = (tempF - 32) * 5.0 / 9.0;
        this.pcs.firePropertyChange("temperatureC", oldVal, temperatureC);
    }
}
```

1.3.2 Les vues

Il est possible de créer une ou plusieurs vues. Chaque vue doit implémenter l'interface `java.beans.PropertyChangeListener` et par conséquent implémenter la méthode `propertyChange()`. Le premier paramètre permet d'identifier le sujet à l'origine de la mise à jour et le second permet de recevoir des informations supplémentaires concernant la mise à jour. Chaque vue a besoin de connaître le modèle et le contrôleur. La connaissance du modèle permet d'enregistrer la vue auprès de ce dernier et d'appeler les méthodes du modèle pour mettre à jour la vue. Les actions de l'utilisateur sur l'interface sont envoyées au contrôleur.

Comme nous avons deux vues très semblables, nous définissons une classe abstraite qui rassemble tous les éléments communs aux deux vues semblables. La troisième vue ne réutilisera pas cette classe abstraite car trop différente. Voici la classe abstraite qui concerne les deux premières vues :

```
package vue;

import control.TemperatureController;
import model.TemperatureModel;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionListener;
import java.beans.PropertyChangeListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public abstract class TemperatureVue implements PropertyChangeListener {
    //private String label;
    protected TemperatureModel model;
    protected TemperatureController controller;
    private JFrame temperatureJFrame;
    private JTextField display = new JTextField();
    private JButton upJButton = new JButton("+");
    private JButton downJButton = new JButton("-");
```

```
    public TemperatureVue(String label, TemperatureModel model,
        TemperatureController controller, int posX, int posY) {
        this.model = model;
        this.controller = controller;
        temperatureJFrame = new JFrame(label);
        temperatureJFrame.add(new JLabel(label), BorderLayout.NORTH);
        temperatureJFrame.add(display, BorderLayout.CENTER);
        JPanel panelbuttons = new JPanel();
        panelbuttons.add(downJButton);
        panelbuttons.add(upJButton);
        temperatureJFrame .add(panelbuttons, BorderLayout.SOUTH);
        temperatureJFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // We register the View as a listener of events from the Model
        model.addPropertyChangeListener(this);
        temperatureJFrame.setSize(200, 100);
        temperatureJFrame.setLocation (posX, posY);
        temperatureJFrame.setVisible(true);
    }
    public void setDisplay(String s) {
        display.setText (s);
    }
    public void enableWarningColor() {
        display.setBackground(Color.RED);
    }
    public void disableWarningColor() {
        display.setBackground(Color.WHITE);
    }
    public double getDisplay() {
        double result = 0.0;
        try {
            result = Double.valueOf(display.getText()).doubleValue();
        }
        catch(NumberFormatException e) {
            System.err.println("String to Double Conversion error:" +
                e.getMessage());
        }
        return result;
    }
    public void addDisplayListener(ActionListener a) {
        display.addActionListener(a);
    }
    public void addUpListener(ActionListener a) {
        upJButton.addActionListener(a);
    }
    public void addDownListener(ActionListener a) {
        downJButton.addActionListener(a);
    }
}
```

Les classes `TemperatureVueCelsius` et `TemperatureVueFahrenheit` dérivent de `TemperatureVue` et implémentent la méthode `propertyChange()`. Voici la classe `TemperatureVueCelsius`.

```
package vue;

import control.TemperatureController;
```

```
import model.TemperatureModel;

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

import java.beans.PropertyChangeEvent;

public class TemperatureVueCelsius extends TemperatureVue {
    public TemperatureVueCelsius(TemperatureModel modele, TemperatureController
        controleur, int posX, int posY) {
        super(" Temperature Celsius ", modele, controleur, posX, posY);
        setDisplay(""+model.getC());
        addUpListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                controller.augmenteDegresC();
            }});
        addDownListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                controller.diminueDegresC();
            }});
        addDisplayListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                double tempC = getDisplay();
                controller.fixeDegresC(tempC);
            }});
    }

    public void propertyChange(PropertyChangeEvent e) {
        // Here we can use the values in the Event (Java8+ facility)
        // Or query the model for the new value (Strict respect of the MVC
            DesignPattern)
        setDisplay(""+model.getC());
    }
}
```

1.3.3 Les contrôleurs

Chaque vue est associée à un contrôleur personnel (un contrôleur par vue). Chaque contrôleur interprète les actions de l'utilisateur et met à jour le modèle. Le contrôleur peut utiliser le patron de conception **Stratégie** pour mettre en œuvre différentes stratégies d'interprétation des actions de l'utilisateur. Dans l'exemple ci-dessous, nous pouvons augmenter la température d'un degré quand on clique sur le bouton + mais il doit être possible de modifier la façon dont la température augmente ou diminue.

Enfin, le contrôleur doit également avoir connaissance de la vue qui lui est associée pour éventuellement modifier cette dernière. En effet, des actions de l'utilisateur sur la vue peuvent avoir pour conséquence de modifier la vue sans pour autant modifier le modèle. Cela se fait par un appel de méthode de la vue. Dans l'exemple ci-dessous, nous faisons passer le champ de texte en rouge quand la température dépasse $40°C$ par un appel dans la méthode control de la méthode `enableWarningColor()` définie dans la vue. Autre exemple, on pourrait de la même façon ajouter un bouton qui, quand on clique dessus dans la vue, modifie la couleur de l'arrière plan de l'interface. Le contrôleur pourrait aussi vérifier que le texte saisi par l'utilisateur dans les champs de texte correspond bien à un nombre et, le cas échéant, informer la vue qu'une erreur de saisie a eu lieu.

```
package control;

import model.TemperatureModel;
import vue.TemperatureVue;

public class TemperatureController {
    private TemperatureModel model ;
    private TemperatureVue view = null ;
    public TemperatureController(TemperatureModel m) {
        model = m;
    }
    public void augmenteDegresC(){
        model.setC(model.getC()+1);
        control();
    }
    public void diminueDegresC(){
        model.setC(model.getC()-1);
        control();
    }
    public void fixeDegresC(double tempC){
        model.setC(tempC);
        control();
    }
    public void augmenteDegresF(){
        model.setF(model.getF()+1);
        control();
    }
    public void diminueDegresF(){
        model.setF(model.getF()-1);
        control();
    }
    public void fixeDegresF(double tempF){
        model.setF(tempF);
        control();
    }
    private void control() {
        if(view != null) {
            if(model.getC() > 40.0) {
                view.enableWarningColor();
            } else {
                view.disableWarningColor();
            }
        }
    }
    public void setView(TemperatureVue view) {
        this.view = view ;
    }
}
```

1.4 Intégration de l'ensemble

La classe `TemperatureMVC` permet de créer l'ensemble des éléments MVC, et de les associer.

```
import control.TemperatureController;
```

```
import model.TemperatureModel;
import vue.TemperatureVueCelsius;
import vue.TemperatureVueFarenheit;

public class TemperatureMVC {
    public TemperatureMVC() {
        TemperatureModel tempmod = new TemperatureModel();
        TemperatureController tempcontrolC = new TemperatureController(tempmod);
        TemperatureController tempcontrolF = new TemperatureController(tempmod);
        TemperatureVueCelsius pvc = new TemperatureVueCelsius(tempmod,
            tempcontrolC, 100, 200);
        TemperatureVueFarenheit tvf = new TemperatureVueFarenheit(tempmod,
            tempcontrolF, 100, 350);
        tempcontrolC.setView(pvc);
        tempcontrolF.setView(tvf);
    }
    public static void main(String args []) {
        new TemperatureMVC();
    }
}
```

1.5 À faire

— Étudier les explications dans le texte ci-dessus.
— Tester l'utilisation de l'exemple de code Java fourni.
— Faire un diagramme de classes UML à partir du code Java fourni
— Faire un diagramme de séquence retraçant les appels de méthodes au lancement de l'application
— Faire un diagramme de séquence depuis une saisie de valeur par l'utilisateur.

2 Exercice : Rectangle

L'objectif de cet exercice est de faire une application dessinant un rectangle dont la hauteur est modifiable par l'utilisateur (voir l'annexe pour un début de code). L'application comporte :

— un modèle pour stocker les informations sur le rectangle.
— deux vues : l'une permet la modification de la hauteur, l'autre trace le rectangle (cf. annexe en fin de sujet pour découvrir comment tracer un rectangle).
— un contrôleur, qui transmet les modifications de hauteur saisies par l'utilisateur au modèle dans le cas où ces modifications sont correctes (valeur de hauteur entre un min et un max définis dans le modèle), et change l'interface de saisie de hauteur dans le cas contraire.

Un exemple de fonctionnement est disponible sous forme d'un fichier `jar` sur MooTSE dans la section correspondant à ce TD. Dans cet exemple, les min et max sont respectivement fixés à 0 et 399. En vous inspirant du code de la partie 1 de ce sujet, développez votre propre version de cet exercice.

A Annexe : dessiner un rectangle de taille fixe – à adapter

```
public class RectanglePanel extends JPanel {
    public RectanglePanel() {
        super();
    }

    public Dimension getPreferredSize() {
```

```
        return new Dimension(200, 200);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawRect(10, 10, 200, 100);
    }

    public void refaire() {
        this.repaint(0);
    }
}
```