Conception et programmation objet avancées

TP : Mise en oeuvre de MVC

Objectifs

- se familiariser avec les patrons de conception observateur et stratégie,
- se familiariser avec la conception d'une application suivant le modèle MVC

1 Illustration de la mise en oeuvre de MVC

1.1 Objectif

L'objectif est de créer une interface permettant le contrôle de la température en degrés Celsius ou Farenheit. 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.

Le code de l'application est disponible ici [1].

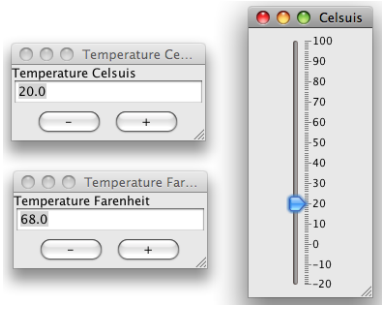

FIGURE 1 – L'interface de contrôle de température. Les trois vues représentent un même modèle. La modification d'une vue met automatiquement à jour les autres vues.

1.2 Préambule

1. `TPtemperature.zip`

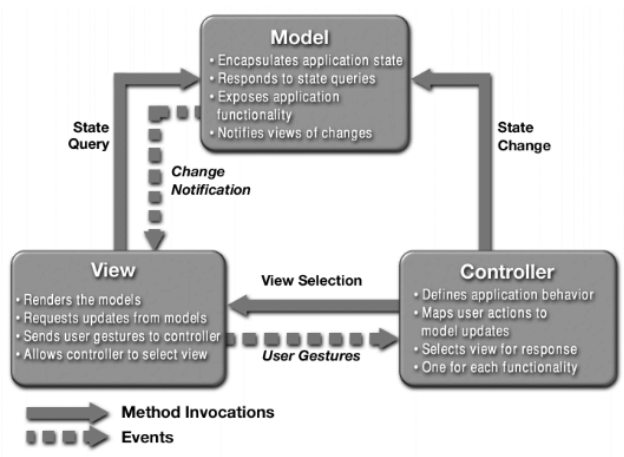

FIGURE 2 – Le modèle MVC.

Le MVC (Model View Controller), Modèle, Vue, Contrôleur 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 qui dérivent de la classe java.util.Observable [2] (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 sont construites à base de composants Swing. Les vues doivent cependant implémenter l'interface java.util.Observer [3] (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 intervenu. Ces dernières se mettent à jour. Il peut aussi vérifier les actions de l'utilisateur et modifier la vue en conséquence si besoin.

1.3 Le modèle

Le modèle dérive de la classe java.util.Observable. La classe doit définir les accesseurs et les mutateurs pour obtenir et modifier l'état courant du modèle. Chaque mutateur doit appeler la méthode setChanged() et notifyObservers() après avoir modifié les données du modèle. notifyObservers va notifier chaque vue qu'un changement d'état du modèle a eu lieu. notifyObservers permet également de passer des informations supplémentaires concernant le changement d'état (variante push du patron de conception Observateur). Voici la classe du modèle correspondant à notre interface :

```
public class TemperatureModel extends Observable {
        private double temperatureC = 20;

        public double getC(){
                return temperatureC;
        }

        public void setC(double tempC){
```

2. http://java.sun.com/j2se/1.4.2/docs/api/java/util/Observable.html
3. http://java.sun.com/j2se/1.4.2/docs/api/java/util/Observer.html

```
                temperatureC = tempC;
                setChanged();
                notifyObservers();
        }

        public double getF(){
                return temperatureC*9.0/5.0 + 32.0;
        }

        public void setF(double tempF){
                temperatureC = (tempF - 32) * 5.0 / 9.0;
                setChanged();
                notifyObservers();
        }
}
```

1.4 Les vues

Il est maintenant possible de créer une ou plusieurs vues. Chaque vue doit implémenter l'interface java.util.Observer et par conséquent implémenter la méthode update. 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. Une vue peut éventuellement avoir plusieurs modèles. 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 la classe abstraite suivante :

```
public abstract class TemperatureVue implements Observer {
        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("-");

        TemperatureVue(String label, TemperatureModel model,
                        TemperatureController controller, int posX, int posY) {
                this.label = label;
                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);
                model.addObserver(this); // Connexion entre la vue et
le modele
                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){}
                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);}

        protected TemperatureModel model(){
                return model;
        }
}
```

La classe TemperatureVueCelsuis dérive de TemperatureVue et implémente la méthode update :

```
public class TemperatureVueCelsuis extends TemperatureVue {
        public TemperatureVueCelsuis(TemperatureModel modele,
                        TemperatureController controleur, int posX, int posY) {
                super("Temperature Celsuis",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 update(Observable s, Object o) {
                setDisplay(""+model().getC());
        }
}
```

Les autres vues sont définies de manière similaire.

1.5 Les contrôleurs

Chaque vue est associée à un unique contrôleur. Le 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 oeuvre 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 sans avoir à réécrire complètement la classe TemperatureController.

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 40C par un appel dans la méthode control de la méthode enableWarningColor() définie dans la vue. On pourrait de la même façon ajouter un bouton qui, quand on clique dessus sur la vue, modifie par exemple 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.

```
interface ModifieTemperature {
        public double augmenteDegres(double temp);
        public double diminueDegres(double temp);
}

class ModifieTemperaturePlus1 implements ModifieTemperature {
        public double augmenteDegres(double temp) {
                return temp + 1;
        }

        public double diminueDegres(double temp) {
                return temp - 1;
        }
}

public class TemperatureController {
        private TemperatureModel model;
        private TemperatureVue view = null;
        private ModifieTemperature modtemp = new ModifieTemperaturePlus1();

        public TemperatureController(TemperatureModel m) {
                model = m;
        }

        public void augmenteDegresC(){
                model.setC(modtemp.augmenteDegres(model.getC()));
                control();
        }

        public void diminueDegresC(){
                model.setC(modtemp.diminueDegres(model.getC()));
                control();
        }

        public void fixeDegresC(double tempC){
                model.setC(tempC);
                control();
        }

        public void augmenteDegresF(){
                model.setF(modtemp.augmenteDegres(model.getF()));
                control();
        }

        public void diminueDegresF(){
                model.setF(modtemp.diminueDegres(model.getF()));
                control();
        }

        public void fixeDegresF(double tempF){
                model.setF(tempF);
                control();
        }

        public void control() {
                if (view != null) {
                        if (model.getC() > 40.0) {
```

```
                                view.enableWarningColor();
                        } else {
                                view.disableWarningColor();
                        }
                }
        }

        public void addView(TemperatureVue view) {
                this.view = view;
        }
}
```

1.6 Intégration du tout

```
public class TemperatureMVC {
        public TemperatureMVC() {
                TemperatureModel tempmod = new TemperatureModel();
                TemperatureController tempcontrolC = new TemperatureController(tempmod);
                TemperatureController tempcontrolF = new TemperatureController(tempmod);
                TemperatureVueCelsuis pvc = new
                                TemperatureVueCelsuis(tempmod, tempcontrolC, 100, 200);
                TemperatureVueFarenheit tvf = new
                                TemperatureVueFarenheit(tempmod, tempcontrolF, 100, 350);
                tempcontrolC.addView(pvc);
                tempcontrolF.addView(tvf);
        }

        public static void main(String args[]) {
                //Schedule a job for the event-dispatching thread:
                //creating and showing this application's GUI.
                javax.swing.SwingUtilities.invokeLater(new Runnable() {
                        public void run() {
                        new TemperatureMVC();
                        }
                });
        }
}
```

Question 1. Ajoutez la vue sous forme de thermomètre (voir figure 1) en utilisant la classe JSlider. La modification de cette vue doit bien sûr modifier automatiquement les autres vues. Remarque : Il est fortement déconseillé de faire hériter cette vue de la classe abstraite `TemperatureVue`.

2 Mise en application

Question 2. Reprenez l'exercice sur le sélecteur de couleur et appliquez le modèle MVC en vous inspirant de l'exemple précédent.

Question 3. Modifiez les contrôleurs nécessaires pour afficher un retour visuel sur la vue correspondante quand une valeur erronée a été entrée par l'utilisateur (par exemple, une valeur en dehors de la plage 0-255 ou une valeur non numérique rentrée pour les représentations décimales de la couleur). Ce retour visuel peut, par exemple, prendre la forme d'un changement de la couleur d'arrière plan du champ de texte ou alors se présenter sous forme de popup.

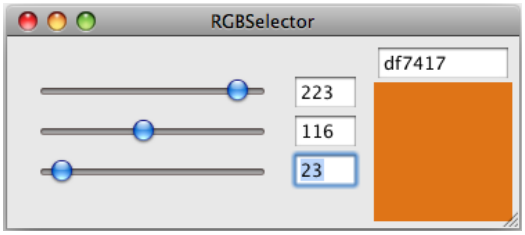

FIGURE 3 – Sélecteur de couleur.