

Guide du *designer* MVC¹

Cette présentation a pour but de vous faire découvrir les aspects importants et fortement utilisés du design pattern MVC.

1 INTRODUCTION

MVC est un *design pattern* (modèle de conception ou DP) de conception d'interface utilisateur permettant de découpler

- le modèle (logique métier et accès aux données)
- des vues (interfaces utilisateur [présentation des données et interface de saisie pour l'utilisateur]).

Des modifications de l'un n'auront ainsi, idéalement, aucune conséquence sur l'autre ce qui facilitera grandement la maintenance.

Ce *design pattern* a tendance à multiplier le nombre de classes à définir et semble alourdir la conception d'application mais le découplage ainsi obtenu assure une maintenance facilitée. Attention, ce DP peut être appliqué de manières fort différentes mais abordons-en le principe avant de voir des exemples simples d'application.

2 PRINCIPE

- **Modèle**: gère les données et reprend la logique métier (le modèle lui-même peut être décomposé en plusieurs couches mais cette décomposition n'intervient pas au niveau de MVC). **Le modèle ne prend en compte aucun élément de présentation!**
- **Vue**: elle affiche les données, provenant exclusivement du modèle, pour l'utilisateur et/ou reçoit ses actions. **Aucun traitement – autre que la gestion de présentation - n'y est réalisé.**
- **Contrôleur**: son rôle est de traiter les événements en provenance de l'interface utilisateur et les transmet au modèle pour le faire évoluer ou à la vue pour modifier son aspect visuel (pas de modification des données affichées mais des modifications de présentation (couleur de fond, affichage ou non de la légende d'un graphique, ...)).

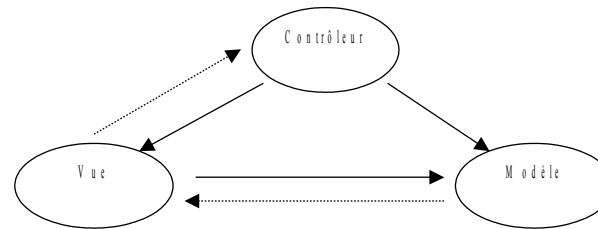
Le contrôleur 'connaît' la (les) vues qu'il contrôle ainsi que le modèle.

Le contrôleur pourra appeler des méthodes du modèle pour réagir à des événements (demande d'ajout d'un client par exemple), il pourra faire modifier à la vue son aspect visuel. Il pourra aussi instancier de nouvelles vues (demande d'affichage de telle ou telle info). Pour faire cela, le contrôleur sera à l'écoute d'événements survenant sur les vues.

La vue observera le modèle qui l'avertira du fait qu'une modification est survenue. Dans ce cas, la vue interrogera le modèle pour obtenir son nouvel 'état'.

¹ Ce document a été rédigé initialement par nos prédécesseurs –et principalement par *adt*-- qu'ils en soient ici remerciés.

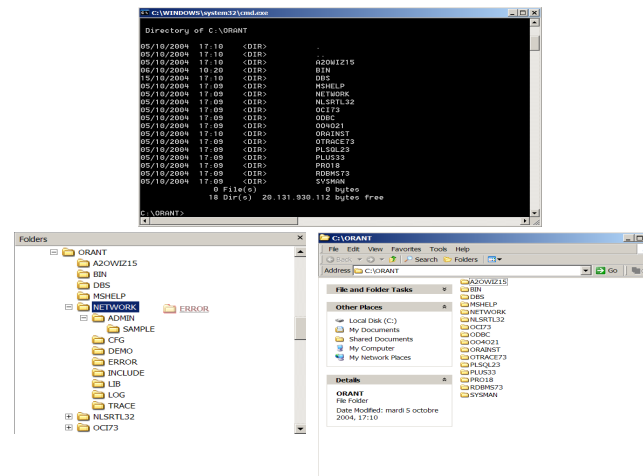
Nous pouvons schématiser cela comme suit en percevant bien que nous pouvons avoir plusieurs contrôleurs, plusieurs vues mais un seul modèle.



3 UN EXEMPLE PRATIQUÉ QUOTIDIENNEMENT

Plusieurs vues différentes nous sont offertes par un *operating system* (système d'exploitation, SE) sur son *file system* (système de fichier, FS). Un contrôleur pourra recevoir au travers de l'interface utilisateur

- les événements de mise à jour du modèle (par exemple: demande de l'utilisateur de la suppression d'un fichier, d'ajout d'un répertoire, ...) et le transmettra au modèle. Les vues observatrices du modèle seront prévenues de la modification du modèle et l'interrogeront pour mettre à jour leur affichage.
- les événements de modification de présentation des données (par exemple: demande par l'utilisateur de la présentation détaillée - plutôt que par icône - de la liste des fichiers et des répertoires) seront transmis à la vue qui modifiera sa présentation en tenant compte de la demande et sans nécessairement réinterroger le modèle.



Il est à remarquer que si différents *file system* (cf. NTFS et FAT) peuvent coexister, il suffit qu'ils présentent le même interface pour que les vues et les contrôleurs soient réutilisables.

Remarquons, comme dans les images ci-dessus, que nous pouvons avoir plusieurs vues simultanément sur le même modèle.

Si les vues doivent refléter en « temps réel » les changements d'état du modèle, elles devront être abonnées (par le contrôleur ou via le contrôleur) comme observateur (écouteur) du modèle.

4 UN EXEMPLE SIMPLE EN JAVA²

Nous allons maintenant présenter une structure simple de mise en œuvre du *design pattern* MVC³.

Dans ce cas, le modèle devra proposer une interface permettant aux vues de l'observer, ces vues devront pouvoir avoir la possibilité de s'inscrire ou de se désinscrire en tant qu'observateur du modèle. Cette interface devra également proposer une ou plusieurs méthodes permettant d'agir sur le modèle⁴. Ces méthodes dépendent du problème à résoudre.

```
/*
 * MonProblemeModele.java
 */

package modele;

/**
 * Contrainte propre à « MonProbleme »
 */
public interface MonProblemeModele {
    /**
     * Permet à la vue de s'abonner en tant que observateur du modele
     */
    public void addMonProblemeListener (MonProblemeVue add);

    /**
     * Permet à la vue de se desabonner en tant que observateur
     * du modele
     */
    public void removeMonProblemeListener (MonProblemeVue add);

    /**
     * Ajouter ici des signatures de methodes propres au probleme et
     * permettant de mettre l'etat du modele à la disposition des vues
     * (via le(les) contrôleurs
     */
}
```

Le modèle proposera également un contrat –via une interface– que devront respecter toutes les vues. Les vues devront pouvoir s'inscrire et se désinscrire auprès du modèle et présenter au modèle une interface permettant de leur notifier les évolutions du modèle.

```
/**
 * MonProblemeVue.java
 */
package modele;
```

² Il doit être clair que le *design pattern* MVC est proposé par l'orienté objet et non par Java. Il n'empêche que nous en donnons ici une illustration en langage Java.

³ Cette « solution » est largement issue du travail de *mcd* et *adt*. Qu'ils en soient ici remerciés.

⁴ Ceci devrait être le rôle du contrôleur dont nous parlons peu dans cet exemple.

```

/**
 * Contrat à respecter par les vues
 */
public interface MonProblemeVue {
    /**
     * L'etat du modele a change, il notifie la (les) vue(s)
     * du changement
     */
    public void notifieChangement();
}

```

Pour le modèle, il reste à écrire une classe implémentant les contraintes du modèle. Il est clair que cette classe peut faire appel à d'autres classes « métiers » c'est le cœur de l'application côté modèle.

```

/*
 * MonProbleme.java
 */

package modele;

import java.util.Vector;

/**
 * Modele pour resoudre mon probleme
 */
public class MonProbleme implements MonProblemeModele {

    private Vector<MonProblemeVue> listeners;
    // ... autres attributs propres au probleme à resoudre
    /** Creates a new instance */
    public MonProbleme() {
        listeners=new Vector<MonProblemeVue>();
        // ...
    }

    /**
     * abonne une vue comme observateur du modele
     * @param vue à abonner
     */
    public void addMonProblemeListener(MonProblemeVue vue) {
        listeners.add(vue);
        fireChange();
    }

    /**
     * desabonne une vue comme observateur du modèle
     * @param vue à desabonner
     */
    public void removeMonProblemeListener(MonProblemeVue vue) {
        listeners.remove(vue);
        fireChange();
    }

    private void fireChange(){
        for (MonProblemeVue vue : listeners){
            vue.notifieChangement();
        }
    }
}

```

Les vues devront implémenter l'interface prévue par le modèle (dans notre exemple MonProblemeVue) . Pour faciliter le travail d'écriture d'une vue, nous pouvons passer par une classe abstraite réalisant déjà une partie du travail.

Dans cet exemple les vues seront des **composants**⁵ (JComponent). Ces composants pourront donc être ajoutés dans un *container* quelconque; JFrame ou JPanel.

```
/*
 * AbstractMonProblemeVue.java
 */

package vue;

import javax.swing.JComponent;
import javax.swing.JPanel;
import modele.MonProblemeModele;
import modele.MonProblemeVue;

/**
 *
 */
public abstract class AbstractMonProblemeVue extends JComponent
    implements MonProblemeVue {

    protected MonProblemeModele modele;

    /**
     * Donne le modele associe
     * @return le modele
     */
    public MonProblemeModele getMonProbleme() {
        return modele;
    }

    /**
     * Permet de preciser à la vue quel est son modele
     * @param modele le modele
     */
    public void setMonProbleme(MonProblemeModele modele){
        removeMonProbleme();
        this.modele=modele;
        addMonProbleme();
        notifieChangement();
    }

    /**
     * Desinscrit la vue en la retirant de la liste des ecouteurs
     * du modele
     */
    public void removeMonProbleme() {
        if (modele != null)
            modele.removeMonProblemeListener(this);
    }
}
```

⁵ Ceci n'est en rien une obligation, c'est un choix pour cet exemple.

```

/**
 * Inscrit la vue en l'ajoutant à la liste des ecouteurs
 * du modele
 */
public void addMonProbleme() {
    this.modele.addMonProblemeListener(this);
}

/**
 * Permet de desinscrire la vue lorsque le composant
 * est retire de son Container (reecriture d'une methode de
 * Jcomposant
 */
public void removeNotify() {
    super.removeNotify();
    removeMonProbleme();
}
}

```

Il reste à écrire une classe représentant une vue proprement dite, cette classe héritera de la classe abstraite `AbstractMonProblemeVue`. À ce stade il est « facile » d'écrire plusieurs vues différentes du même modèle.

```

/*
 * MonProblemeVue1.java
 */

package vue;

import modele.*;
/**
 * Une vue du probleme ...
 */
public class MonProblemeVue1 extends AbstractMonProblemeVue {

    /** Creates new form */
    public MonProblemeVue1() {
        initComponents();
    }
    /** Creates new form */
    public MonProblemeVue1(MonProblemeModele modele) {
        initComponents();
        setMonProbleme(modele);
    }

    /**
     * Le modele me notifie son changement, je vais l'interroger
     * et me mettre à jour.
     */
    public void notifieChangement() {
        // Interrogation du modele afin d'obtenir sont etat
        // ... modele ...

        // Mise à jour de la vue sur base de l'etat du modele
    }
}

```

Le terme « contrôleur » est utilisé dans cet exemple pour parler d'une multitude d'objets (les contrôleurs) réagissant à des événements. En pratique, dans ce cas, les contrôleurs seront des classes anonymes à l'écoute de vues ou de partie de vues (boutons, `textFields`, `comboBox`, `items` de menu ...).

Pour illustrer l'intérêt d'une telle technique il faudra créer plusieurs vues différentes du problème et visualiser ces différentes vues d'un **même modèle**. Pour ce faire on peut envisager un `JFrame` (lançant le modèle) et permettant de lancer différentes vues du modèle simultanément. Ces vues se mettront ensemble à jour lors de la modification du modèle.