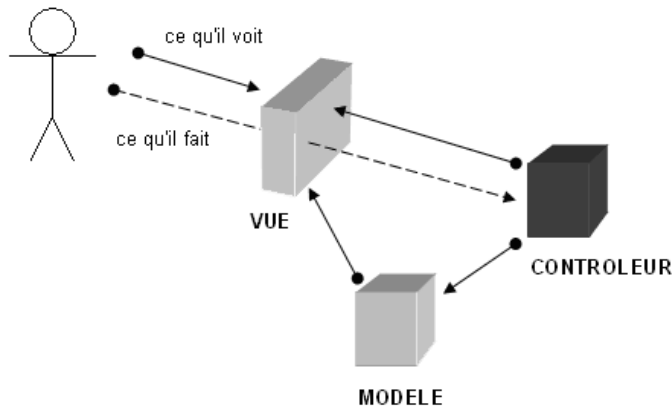


1 Présentation

Le projet SA_Cagou a été bâti en prenant soin dès la phase d'analyse de séparer les classes selon leur type : dialogue, contrôle et entité. Le **modèle MVC** ou design pattern **Model View Controller** formalise cette séparation et vise en outre à la synchronisation des vues (classes dialogue) et des modèles (classes entité) en s'appuyant sur les contrôleurs (classes contrôle).

Dans un contexte MVC, les données constituent les modèles, les classes graphiques les vues. Modèles et vues sont totalement indépendants les uns des autres. Nous dirons qu'ils ne se connaissent pas. Comment alors les actions de l'utilisateur via les vues peuvent-elles modifier les modèles et inversement, comment les données attendues peuvent-elles être affichées ? C'est justement les contrôleurs qui vont établir le lien.

Autre particularité importante. Le modèle, s'il ne connaît pas les vues qui affichent ses données, a néanmoins la possibilité de notifier à celles-ci tout changement le concernant.



Le schéma nous montre que l'utilisateur n'a aucune connaissance du contrôleur. C'est pourtant lui qui transmet de manière transparente pour l'utilisateur les requêtes au modèle pour traitement. Les vues sont mises à jour soit par le contrôleur soit par le modèle par un système de notification basé sur la notion d'événements et d'écouteurs d'événements.

Jusqu'à présent dans notre projet, pour effectuer une mise à jour de la fenêtre présentant les données en mode table, il faut la refermer, ouvrir une autre fenêtre pour effectuer les traitements CRUD puis rouvrir la première fenêtre. Une nouvelle ouverture correspond en fait à une nouvelle instance de la fenêtre en mode table avec interrogation de la base et mise à jour du composant JTable (revoir au besoin les sections Gestion de l'affichage des données et Gestion des traitements du chapitre Développement).

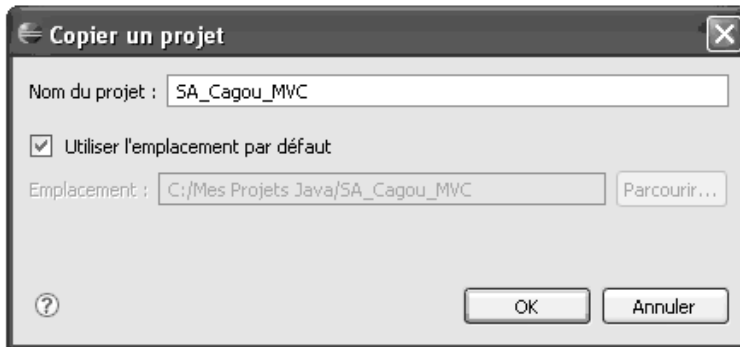
Ce procédé fonctionne mais ne permet pas de réaliser une mise à jour immédiate de la fenêtre en mode table. Il devient alors impossible de mettre à jour de manière simultanée différentes vues des mêmes données suite à des modifications les concernant.

Dans ce chapitre, nous voyons donc comment mettre en œuvre le modèle MVC avec le composant swing JTable pour nous affranchir de cette limite.

2 Création du modèle

Nous allons auparavant créer une copie du projet.

- Effectuez un clic droit sur le projet SA_Cagou dans l'explorateur de packages puis choisissez **Copier**.
- Puis toujours dans l'explorateur de packages effectuez un autre clic droit et renommez le projet "SA_Cagou_MVC".



Pour rendre les vues totalement indépendantes des données, il faut commencer par créer un modèle. La classe **Client** va être modifiée pour permettre la création du modèle.

- Ouvrez cette classe dans le paquetage **entite** et procédez à l'importation suivante :

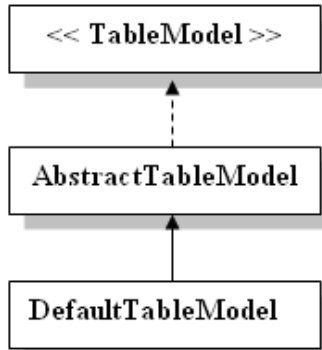
```
.....  
import javax.swing.table.DefaultTableModel;
```



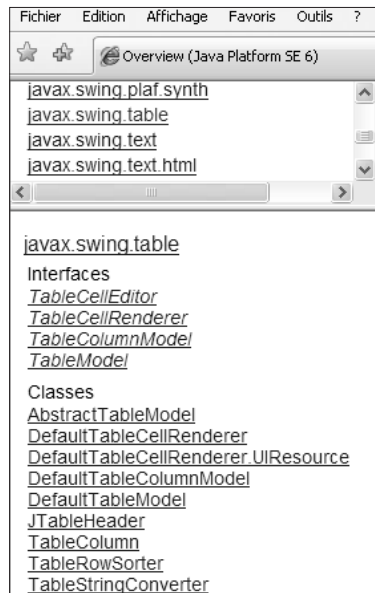
La classe **DefaultTableModel** étend la classe **AbstractTableModel** qui elle-même implémente l'interface **TableModel**.

- L'interface **TableModel** spécifie des méthodes utiles pour gérer les données tabulaires contenues par un modèle (`getColumnName(int columnIndex)`, `getRowCount()`, ...)

- La classe `AbstractTableModel` redéfinit la plupart des méthodes de l'interface `TableModel` en proposant un code par défaut. Elle fournit en outre les méthodes pour l'enregistrement des vues auprès du modèle et celles pour la notification des événements survenus.
- La classe `DefaultTableModel` est une sous-classe de la classe `AbstractTableModel`. Elle peut donc utiliser et redéfinir si nécessaire les méthodes de cette dernière.



La consultation de l'aide de SUN s'avère indispensable. Lancez celle-ci puis choisissez dans le volet **Packages**, le paquetage **javax.swing.table**.



➤ Ajoutez la propriété suivante :

```
.....  
private DefaultTableModel leModele = new DefaultTableModel();
```

➤ Ajoutez un accesseur pour cette propriété.

```
.....  
public DefaultTableModel getLeModeleClients(){  
    return leModele;  
}
```



C'est ce modèle qui sera attaché au composant JTable.

Actuellement dans le projet, lors de l'instanciation de la classe **FenTableClient**, une sélection de tous les enregistrements de la table **Client** est effectuée. Il faut donc intervenir au niveau du code effectuant cette opération dans la classe **Client**.

➤ Modifiez la méthode **chercherCRUD_Clients(Client lesClients)**.

```
.....  
public int chercherCRUD_Clients(Client_MVC lesClients){  
    ...  
    for( i=1 ; i <= infojeuEnregistrements.getColumnCount(); i++){  
        // les titres des colonnes sont récupérés avec la méthode getColumnLabel  
        String nomColonneModel = infojeuEnregistrements.getColumnLabel(i);  
        // puis ajoutés au modèle  
        // -----  
        leModele.addColumn(nomColonneModel);  
    }  
    while(jeuEnregistrements.next()) {  
        Vector<String> ligne = new Vector<String>();  
        for(i=1; i <= infojeuEnregistrements.getColumnCount(); i++) {  
            String _champ = jeuEnregistrements.getString(i);  
            ligne.add(chaine_champ);  
        }  
        nbClients = nbClients + 1;  
        // le vecteur ligne est ajouté au modèle  
        // -----  
        leModele.addRow(ligne);  
    }  
    ...  
}
```



Après l'exécution de la méthode, le modèle contient les données et les titres des colonnes.

☑ Lancez l'application.

Plus aucune donnée n'est visible mais ceci est tout à fait normal puisque l'affichage est assuré par le composant JTable et non par le modèle.

3 Création de la vue

Pour que les données soient visibles, il suffit de préciser quel modèle doit être affiché par le composant JTable.

☑ Ouvrez la classe **FenTableClient** dans le paquetage **dialogue** et procédez aux importations suivantes.

```
.....
import javax.swing.event.TableModelListener;
import javax.swing.event.TableModelEvent;
```



Toute vue sur un modèle doit implanter l'interface **TableModelListener**. Cette interface permet à la vue d'être à l'écoute des événements modifiant l'état du modèle. Elle ne dispose que d'une seule méthode **tableChanged(TableModelEvent e)** qui est redéfinie en fonction des événements survenus.

☑ Implémentez l'interface **TableModelListener**.

```
.....
public class FenTableClient extends JFrame implements TableModelListener {
```

☑ Eclipse vous signale que la méthode **tableChanged(TableModelEvent e)** doit être implémentée. Double cliquez sur **Ajouter des méthodes non implémentées**.

<pre>public class FenTableClient extends JFrame implements TableModelListener {</pre>	<p>1 méthode(s) à implémenter : - javax.swing.event.TableModelListener.tableChanged()</p>
---	--

```
.....
public void tableChanged(TableModelEvent arg0) {
    // TODO Raccord de méthode auto-généré
}
```



Pour chaque type d'événement, il est possible de programmer une action particulière. La mise à jour de la vue avec le composant `JTable` est par contre automatique. Notez le type du paramètre. Il s'agit de la classe **TableModelEvent** qui étend la classe **EventObject**.

```
java.lang.Object
├── java.util.EventObject
│   └── javax.swing.event.TableModelEvent
```

➤ Ajoutez la propriété suivante :

```
.....
private DefaultTableModel leModele;
```



La propriété **leModele** est initialisée lors de l'exécution de la méthode **remplirTable()**.

Il faut maintenant intervenir sur le code de cette méthode à l'origine de l'appel des méthodes permettant de sélectionner les enregistrements de la table **Client**.

➤ Modifiez la méthode **remplirTable()**.

```
.....
private JTable remplirTable() {
    String vCode = "";
    entite.Client lesClients = new entite.Client (vCode);
    lesClients.chercherCRUD_Clients (lesClients);

    leModele = lesClients.getLeModeleClients ();

    laTable_Table = new JTable (leModele);

    return laTable_Table;
}
```



La table est retournée avec le modèle.