

CPOA - Support TD 6



Version corrigée

Cette version comporte des indications pour les réponses aux exercices.

PreRe q	1. Je sais programmer en Java. 2. J'ai compris le modèle MVC.
ObjTD	Le but de ce TD est de comprendre un composant Swing complexe : le JTable.
Durée	1 TD de 1,5h.

1. Motivation

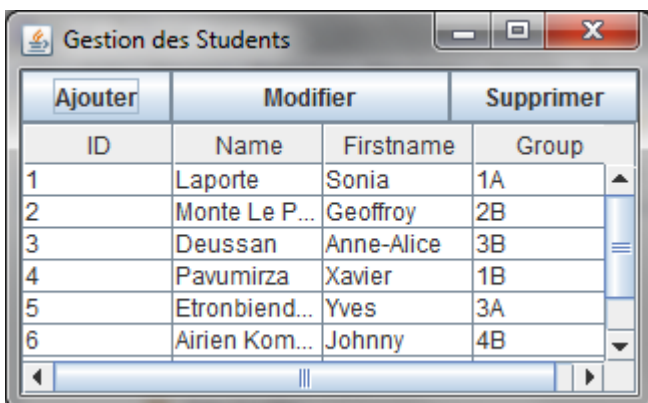
La bibliothèque java Swing (disponible depuis la version 2 de Java (java 1.5)) définit des composants graphiques pour construire des interfaces utilisateurs indépendantes des plateformes d'exécution (cf. <http://docs.oracle.com/javase/tutorial/uiswing/>).

Le but de ce TD et du prochain TP est de manipuler un composant Swing complexe : **JTable**. Nous verrons :

1. En TD : les principes du composant et la séparation MV/C.
2. En TP : la mise en œuvre de ce composant.

2. Présentation JTable : ce qu'il n'est pas

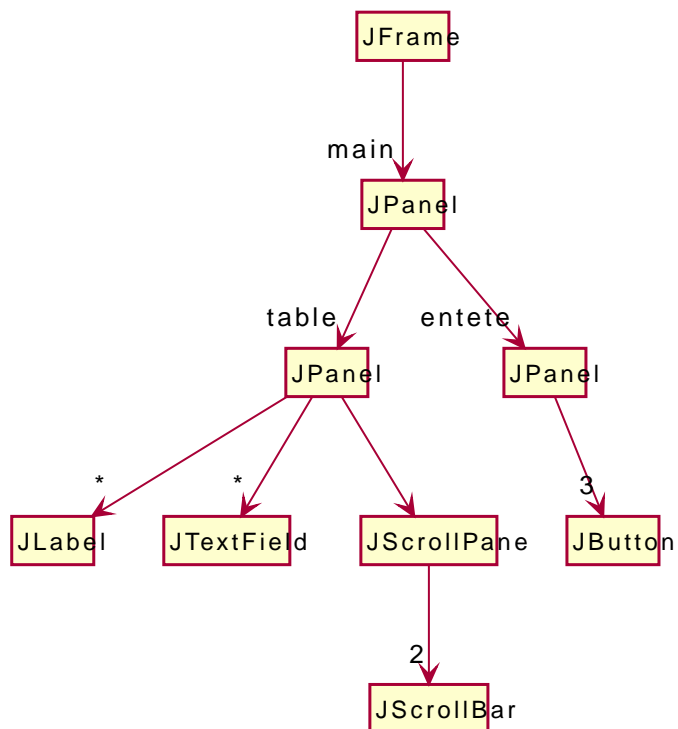
Imaginons que nous voulions programmer cette interface en java :



Avec ce que nous savons de Swing, quels composants graphiques utiliserions nous pour programmer le tableau ci-dessus ?

- ### Eléments de solution

- 



This model has been done by Groupe 1 on December 10th

Quelles sont les limites de cette approche ? Imaginez l'UT2J \Rightarrow 30.000 étudiants! ...

-
-
-
-
-
-
-
-
-
-
-
-

Éléments de solution



1. Non, mauvaise solution :
 - a. Trop de JTextField (30 000 x 4 == 120 000 pour l'université)
 - b. Or tout le tableau n'est pas visible, maximum 50 lignes ⇒ maximum 400 JTF utiles
 - c. Les données sont dans les composants graphiques
 - d. Mises à jour ?
 - e. Ajout d'un étudiant ?
 - f. ...

3. Présentation JTable : mise en œuvre

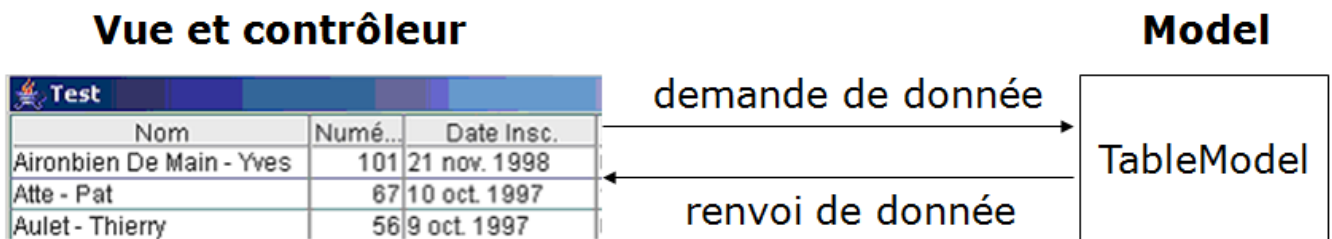
La bibliothèque java Swing propose une gestion graphique de tableaux dans une architecture MVC, ou plutôt M-VC. Il s'appuie sur 2 composants :

- **JTable** qui affiche des données dans un tableau
 - Il est à la fois Vue et Contrôleur.
 - Rôle : afficher les quelques lignes visibles et gérer les mises à jour à l'écran de ces lignes
 - Il est composé lui-même d'un UIDelegate qui est la Vue mais rarement utilisé
 - Un tableau est généralement entouré d'ascenseurs ⇒ le JTable ne demande au modèle QUE les données nécessaires à l'affichage, pas toutes
 - Programmé pour utiliser le minimum de composants graphiques possibles (habituellement 1 seul et même composant par type de données affichée)
- **TableModel** qui régit la gestion des données (les stocke)
 - Il est le Modèle
 - Rôle : contient et stocke toutes les données potentiellement affichables

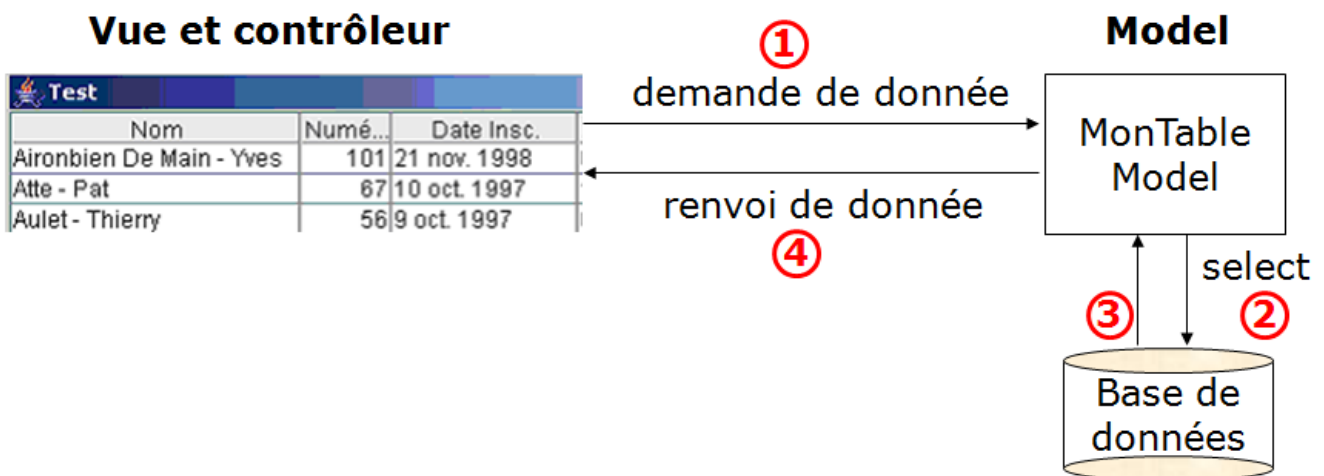
À l'usage :

- On va rarement modifier le JTable.
- On va très souvent créer son propre TableModel

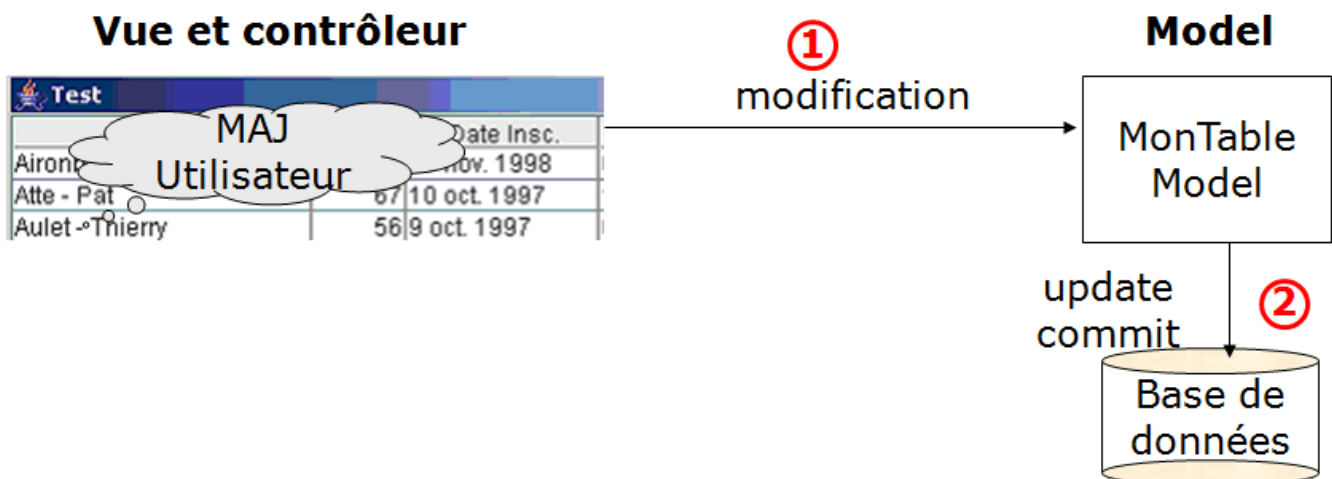
Illustration :

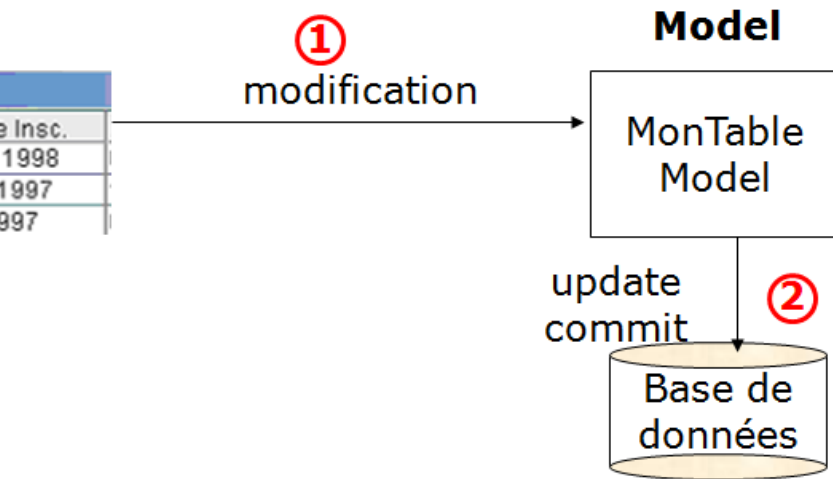


Avec accès à une base de données :



Avec mise à jour d'une base de données :





4. Utilisation d'un TableModel

4.1. Composant graphique

Côté graphique : un `JScrollPane` qui contient un `JTable` et une méthode `setModel()` pour associer un modèle.

```
// Partie composants : un JTable et son TableModel
JTable jta = new JTable ();
MonTableModel mm = new MonTableModel();
jta.setModel (mm);

// Partie graphique : le JTable est mis dans un JScrollPane
JScrollPane jsp = new JScrollPane();
jsp.setViewportViewView (jta);
```

4.2. TableModel : accès aux données

Les données sont accessibles par un modèle. Elles peuvent être stockées ou calculées, de façon transparente.

Un modèle doit implémenter l'interface `TableModel` définissant 9 méthodes utilisées pour beaucoup par le composant `JTable`.

La classe `AbstractTableModel` propose des services complémentaires pour les modèles de table et implémente par défaut les méthodes du modèle de table `TableModel`, sauf :

- `public int getRowCount()` : nombre de lignes affichables
- `public int getColumnCount()` : nombre de colonnes affichables
- `public Object getValueAt(int ligne, int colonne)` : l'objet à afficher dans la ligne et la colonne indiquées (sa méthode `toString` est utilisée).

Les numérotations de lignes et colonnes commencent à 0.

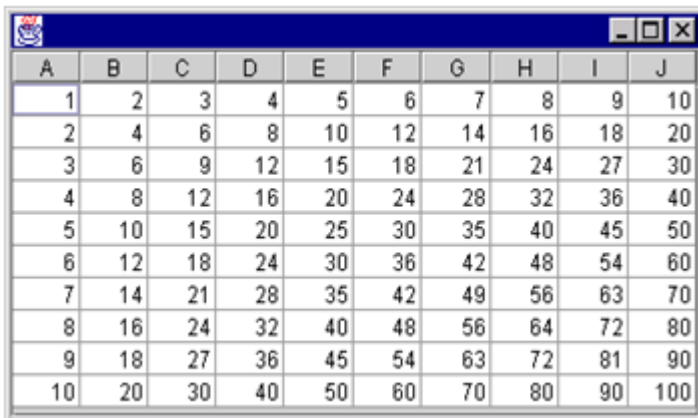
L'exemple suivant crée un modèle de 2000 lignes x 30 colonnes, chaque cellule vaut ligne*colonne.

```
class SimpleTableModel extends AbstractTableModel {
    public int getColumnCount() { return 30; }
    public int getRowCount() { return 2000;}
    public Object getValueAt(int row, int col) {
        return new Integer((1+row)*(1+col));
    }
}

class SimpleJPanelTable extends JPanel {
    public SimpleJPanelTable() {
        this.setLayout(new BorderLayout());

        TableModel dataModel = new SimpleTableModel() ;
        JTable table = new JTable();
        table.setModel (dataModel);
        this.add(new JScrollPane(table));
    }
}
```

Notez que le JTable n'affiche pas les 2000 lignes ...



	A	B	C	D	E	F	G	H	I	J
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

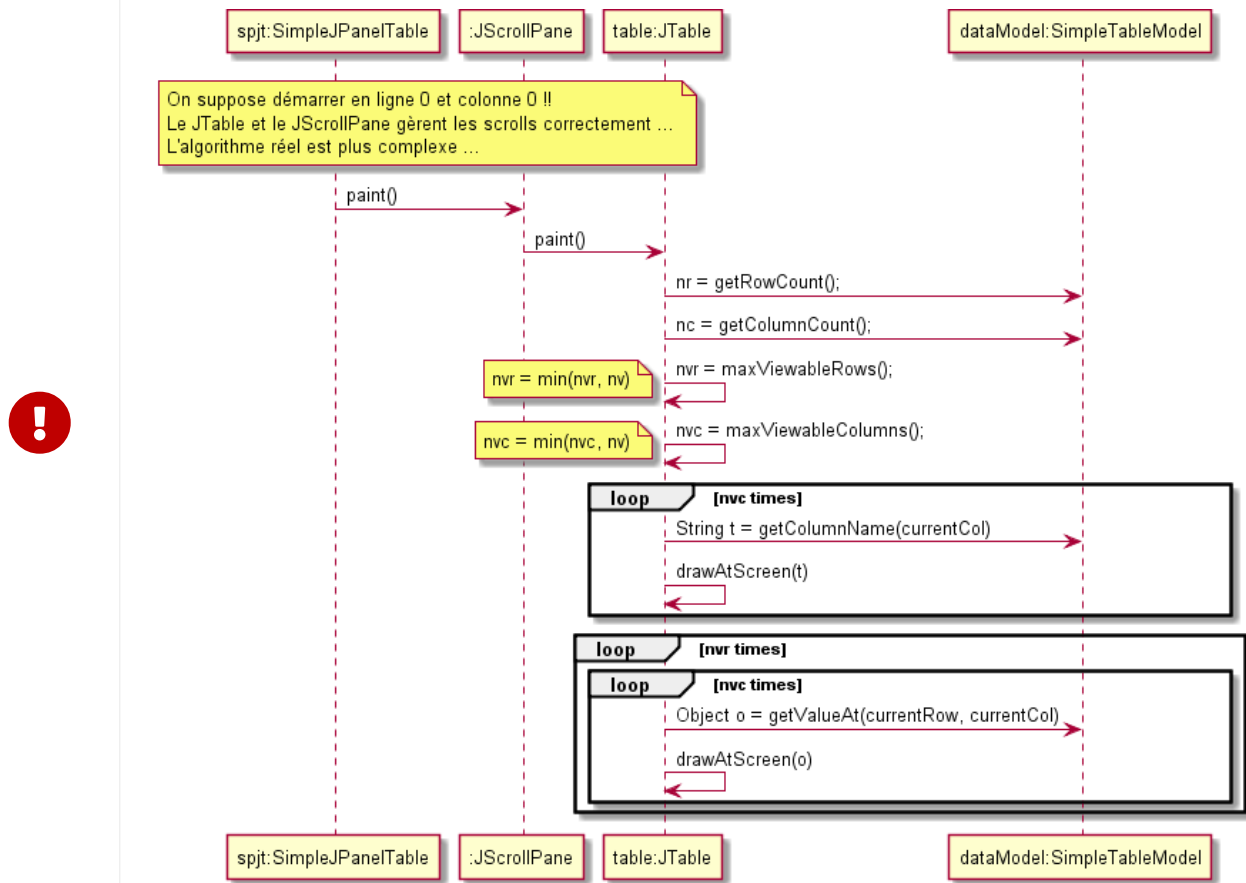
Au plus simple, créer un TableModel revient donc à créer une classe :

- qui hérite de AbstractTableModel
- qui, a minima, redéfinit `getRowCount()`, `getColumnCount()` et `getValueAt()`
- redéfinit aussi `public String getColumnName(int colonne` qui donne les en-têtes de colonne (sinon "A", "B", ... par défaut).

Question : En supposant que le table ci-dessus reçoive le message `paint()` pour s'afficher, donner le pseudo algorithme java exécuté par table pour s'afficher.



Eléments de solution



4.3. Table Model : Mise à jour des données

Pour la mise à jour (update) :

- Elle est faite dans la vue (JTable)
- La mise à jour doit être reportée dans le modèle (TableModel)

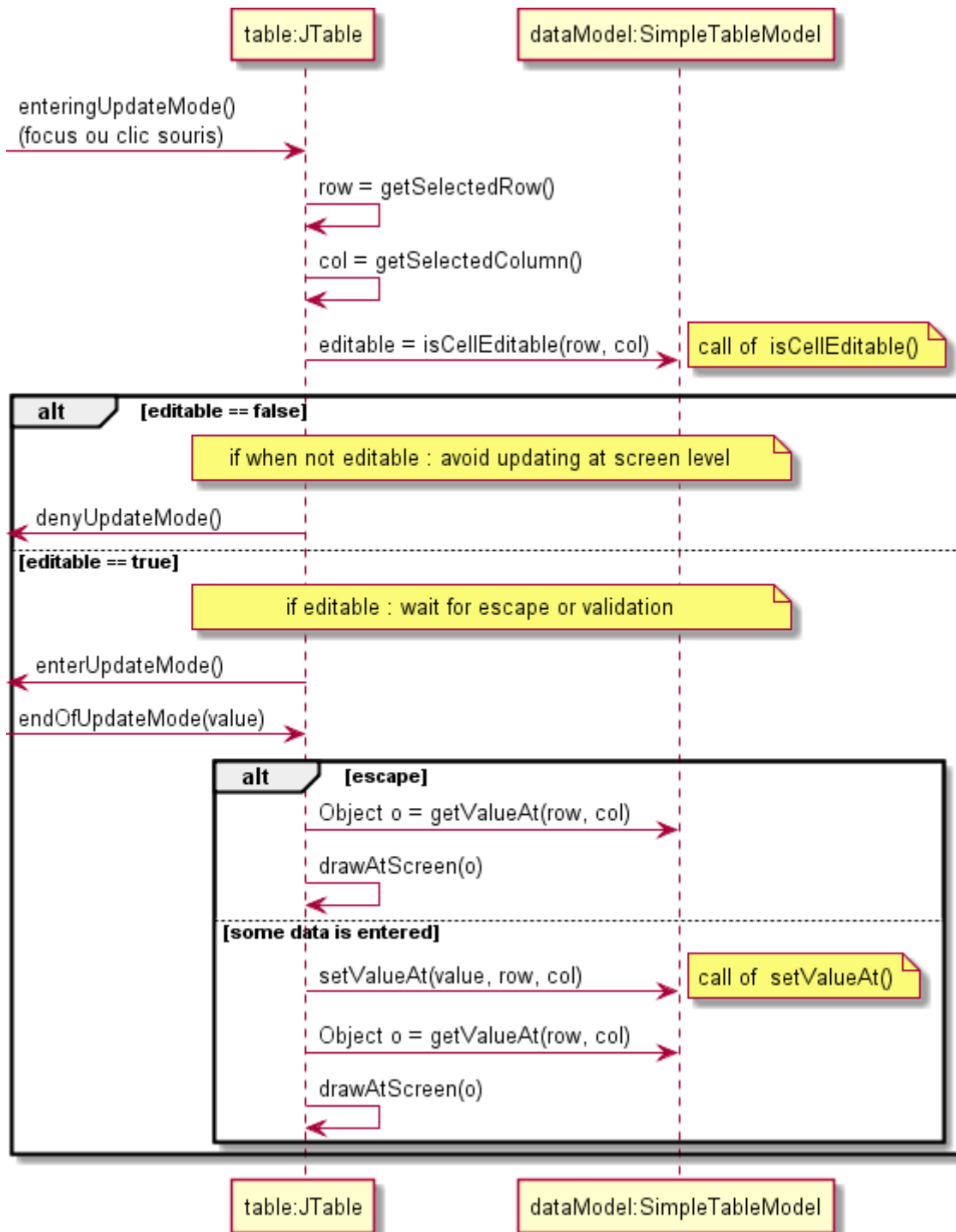
Méthodes de TableModel utilisées par JTable :

- **boolean isCellEditable(int l, int c)** : renvoie true si la cellule peut être modifiée (par défaut false)
- **void setValueAt(Object val, int l, int c)** : modifie une donnée du modèle (par défaut aucune modification)

Elles sont donc à redéfinir selon les besoins :

- Colonnes non modifiables (ids, ...)
- Contrôle des données saisies (chaînes vides, nombre valides, ...)
- ...

Diagramme de séquence illustrant l'utilisation des méthodes par le JTable (pseudo algorithme, l'algorithme réel est plus complexe) :



5. Compléments

5.1. Contrôle du JTable

Beaucoup de méthodes existent (cf. javadoc) dont :

- `int getSelectedRow()` : ligne actuellement sélectionnée dans le JTable
- `int getSelectedColumn()` : colonne actuellement sélectionnée dans le JTable
- Relier la souris avec une donnée
 - `int columnAtPoint(Point p)` : indice de la colonne relative à un point donnée (souris)

- `int rowAtPoint(Point p)` : indice de la ligne relative à un point donnée (souris)
- ...

5.2. Contrôle du TableModel

Des méthodes peuvent être ajoutées au TableModel pour ajouter des lignes, supprimer des lignes, ... cf. TPs.

Cependant, lorsqu'il est mis à jour en dehors du JTable, le modèle doit prévenir tous les JTable associés. Ca vous rappelle un design pattern ?



Éléments de solution

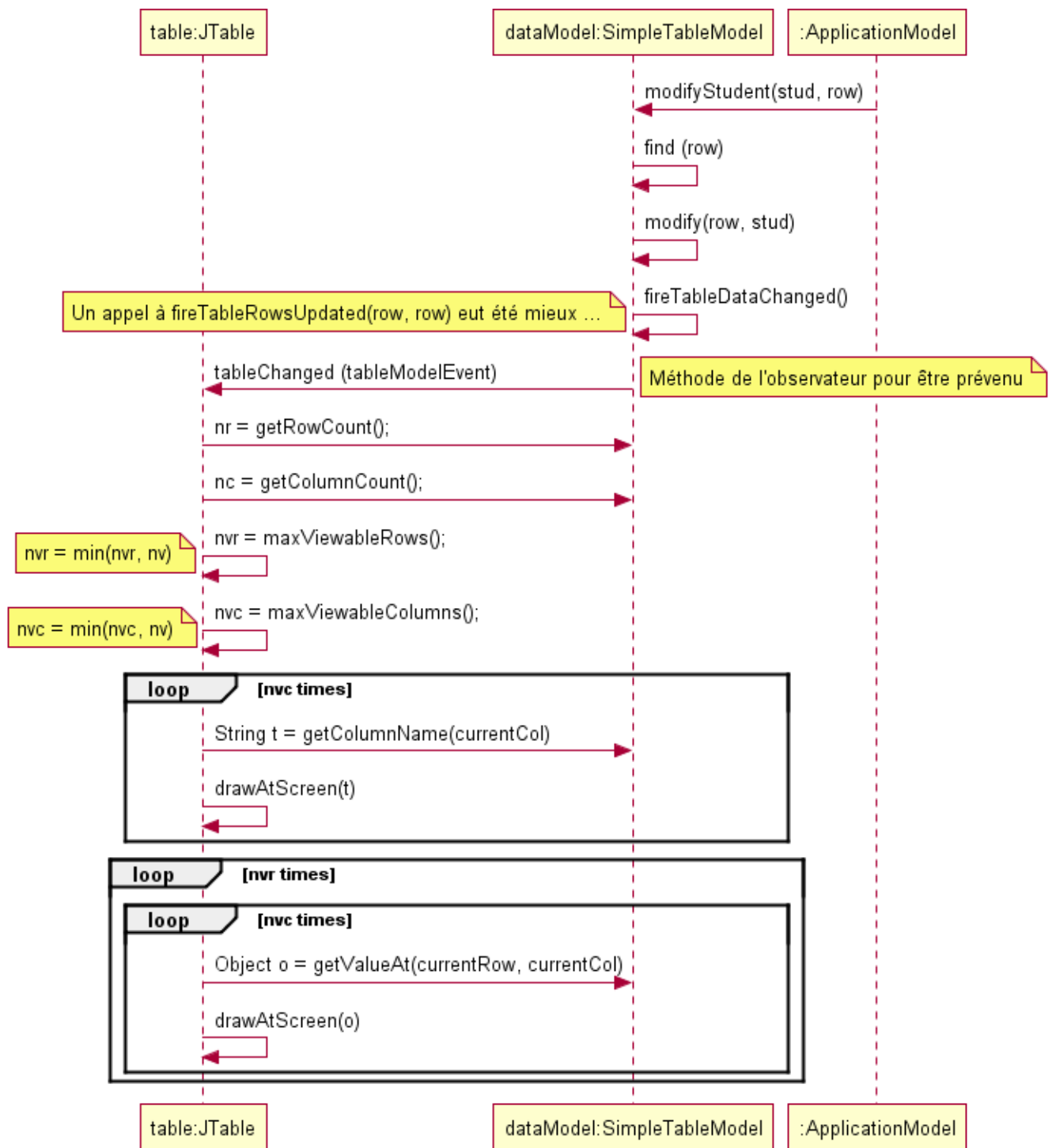
Observateur :

- Observateur == JTable
- Observé == TableModel

Pour prévenir les JTable ⇒ il faut une méthode pour les notifier ⇒ pré-programmé dans AbstractTableModel :

- `void fireTableDataChanged()`
- `void fireTableStructureChanged()`
- `void fireTableRowsInserted(int first, int last)`
- `void fireTableRowsUpdated(int first, int last)`
- `void fireTableRowsDeleted(int first, int last)`
- `void fireTableCellUpdated(int row, int col)`
- `void fireTableChangedEvent(TableModelEvent e)`

Exemple d'appel :



6. Conclusion

Encore une fois on retrouve ici plusieurs éléments des design patterns : "un problème de fond est la gestion des changements". Pour cela, il faut isoler les changements. En objet, cela est délégué à un objet particulier.

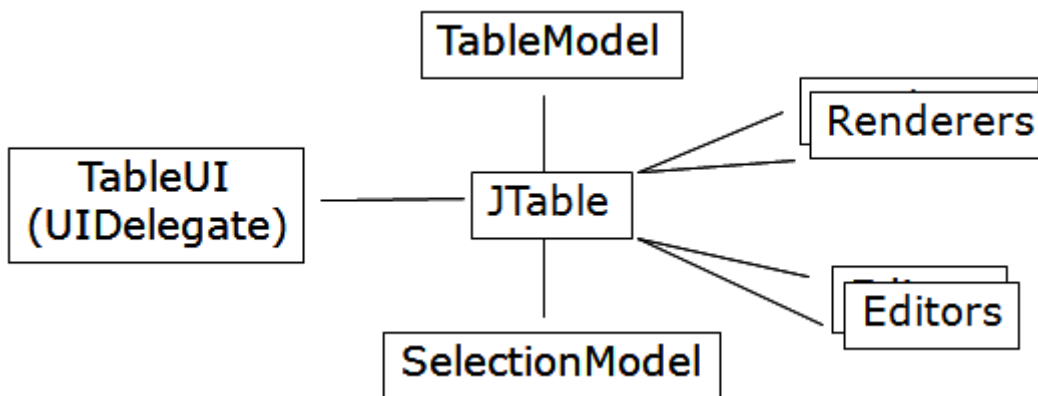
Déjà rencontré : séparation des objets graphiques (JPanel) de leur organisation spatiale (Layout Managers).

Le modèle MVC s'appuie sur ce constat. Il sépare :

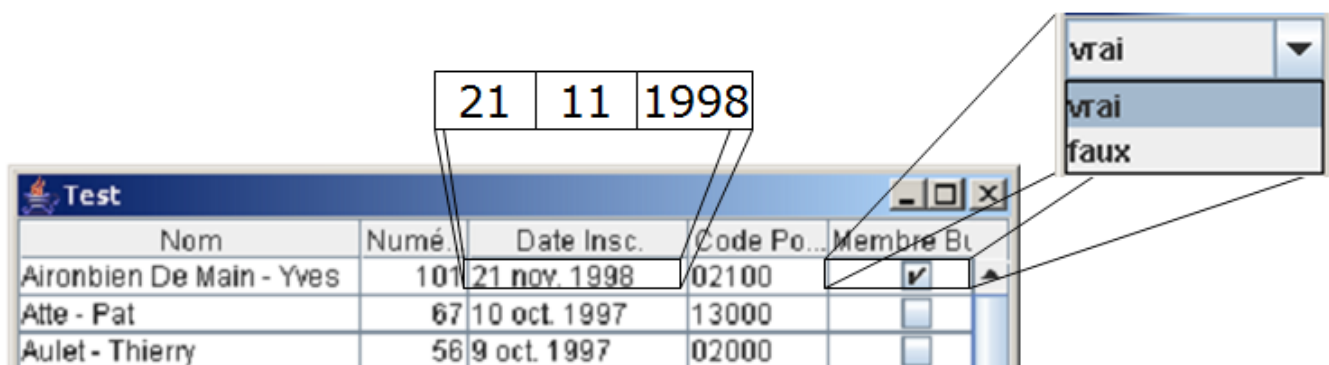
- Le Modèle qui est responsable des données → ici, le TableModel que l'on redéfinit à chaque fois

- La Vue qui est responsable de l'affichage écran de tout ou partie des données → ici le `UIDelegate` contenu dans le `JTable`
- Le Contrôleur : qui gère la cohérence Vue-Modèle → ici le `JTable` (et le pattern observateur entre Contrôleur et Modèle)

Et le reste du `JTable` (affichage d'images, de nombre, saisies avec listes déroulantes, ...) ? ⇒ Même combat : isoler les changements dans des objets particuliers. Voici une partie du modèle que cela donne :



On peut ainsi définir un composant **Render** pour redéfinir comment afficher une colonne, définir un composant **Editor** pour redéfinir comment saisir une valeur de colonne, ...



Derrière l'apparente complexité se cachent des possibilités importantes :

- Décomposition des problèmes et isolement des parties qui "varient"
- Tous les composants proposent des éléments par défaut
- Tout est redéfinissable : on peut créer des sous classes des classes par défaut existantes et les associer aux composants.