***Projet de Gestion d’École***

Arnaud Fouillard

Nam Nguyen

Introduction

Ce projet est un développement d’une interface de gestion pour une école de type secondaire (lycée ou collège). Il doit donc être simple d’utilisation et fonctionnel.

L’utilisateur principal de cette application est un directeur d’école ou un gestionnaire qui renseigne les informations sur les étudiants et/ou modifie les notes de l’étudiant.

I] Modélisation et Conception

**1 - Diagramme de classe**

**Figure 1 Diagramme de classe (simplifié)**

**2- Base de donnée**

Le SGBD utilisée dans ce programme est MySQL.

L’interface utilisée pour la création, modification et gestion des tables est phpMyAdmin.

Cette base de donnée à été lié au projet Netbeans.

**4 - Choix des interfaces**

II] Architecture et choix technique

**1 - Description de notre modèle MVC**

- Le Modèle : Partie représentant les données de l’application. Il définit également

l’interaction avec la base de données (MySQL dans notre cas), et le traitement de ces

données.

\_ **ConnexionBD**

Cette classe regroupe tous les paramètres nécessaires à la connexion avec la

base de données.

On y a développé des méthodes d’insertion, de récupération de données qui

permettent de faire automatiquement le lien avec la base.

De plus, on y trouve diverses méthodes en lien direct avec notre application,

permettant de mettre à jour en base de données, les champs dont l’accès est

fréquent, ou lié à l’application dans sa globalité.

\_ **EleveIdentite**

Cette classe permet d’instancié un objet représentant un élève le minimum

d’information permettant l’affichage de son nom et prénom dans la liste, et la

récupération, ainsi que son id qui sera ensuite utilisée pour récupérer et

afficher toutes ses autres informations par la suite.

\_ **Liste**

Contient des éléments d’une liste. Actuellement de type EleveIdentite, Classe,

ou Matiere.

Classe et matière sont des classes internes de Liste. Ce choix a été fait de

manière à respecter un maximum le principe d’encapsulation. Les classes

Matiere et Classe étant directement et uniquement lié à la classe Liste.

Toutes les autres classes du modèle sont en lien direct avec la base de données. En

effet, ces classes sont la représentation JAVA de chaque table de la base de données.

- La Vue : L’interface principale permet de permettre à l’utilisateur d’avoir une vue

d’ensemble sur l’école. C’est-à-dire de pouvoir visualiser la liste des classes, des

matières, et des élèves de l’établissement.

Un lien est fait entre les classes et les élèves, et le requêtes SQL (implémentées dans

le modèle) permettent de récupérer les élèves en fonction de chaque classe.

\_ **InterfaceAccueil**

Cette classe permet de modéliser l’ensemble de la l’IHM d’accueil. Elle intègre

tous les composants swing qui lui sont liés.

\_ **InterfaceAjoutChamp**

Cette classe permet l’instanciation d’un formulaire à un champ. Il permet, par

le passage en paramètre d’un libelle, de s’adapter à l’ajout de n’importe quel

type de valeur.

Elle est actuellement utilisée pour l’ajout de matières et l’ajout de classes.

Cependant, elle faciliterait également l’évolution de l’application, il l’on

imagine que le client souhaite ajouter, par exemple, une liste de groupes de

TD.

\_ **InterfaceAjoutEleve**

Cette classe correspond à l’interface du formulaire d’ajout d’un élève.

\_ **InterfaceListe**

Cette classe correspond à l’interface du squelette de toute nouvelle liste.

- Le Contrôleur : Il gère l’interface entre le modèle et l’utilisateur. Il interprète les

requêtes et détermine, et/ou actualise la vue correspondante.

Dans notre cas, les différentes classes vont jouer le rôle d’écouteur. Et, de cette

manière, selon « action » envoyé, (chaque type d’action étant un objet de type String

définit en static) va interpréter les commandes à exécuter.

\_ **Controleur**

Cette classe représente le contrôleur principal. Il écoute toutes les frames,

ainsi que tous les composants interne à chaque container.

\_ **CliqueListener**

Cette classe hérite d’une classe qui lui permet d’écouter les commandes

souris.

Dans le contexte de notre application, cela va nous permettre d’identifier les

doubles cliques sur nos objets de type Liste, et donc de provoquer le

l’événement adéquat.

\_ **JFrameListener**

Cette classe va devoir écouter toutes les fenêtres de notre application. En

effet, c’est celle-ci qui gèrera le cas des fenêtres en arrière-plan qui devront

être désactivé, pour mettre en avant la fenêtre de premier plan. Et donc de

cette façon, empêcher l’accès à l’utilisateur, à plusieurs fenêtres

simultanément. Mais également de détecter le cas où l’utilisateur souhaite

quitter l’application.

\_ **ControleurEtudiant :** Cette classe gère le lien entre le contrôleur et le modèle,

notamment connexionBD. Elle ne s'occupe que de la partie qui gère un élève.

\_ **ControleurFrontal** : Cette classe gère toute la partie qui s'occupe d'un élève.

Elle met à jour la vue a chaque changement effectué par l'utilisateur (clique

sur un bouton par exemple).

**2 - Difficultés rencontrées et choix effectuées**

*a - Disposition des éléments*

Concernant les vue, le layout principalement utilisé est GridBagLayout. En effet, il permet

une flexibilité sans pareil, concernant les différents éléments swing. L’interface n’en sort que

plus claire, organisée et professionnelle.

*b - Création de nouvelles fenêtres*

Il a fallu trouver une solution concernant l’ouverture de plusieurs fenetres au sein même de

l’application. En effet, le problème se posait lorsque plusieurs fenêtres s’ouvraient

simultanément. De cette façon, l'utilisateur pouvait sans trop de problème arriver à

saturation de la mémoire en ouvrant plein de fenêtres à la suite. (Par exemple il pouvait

faire en sorte que l’application instancie jusqu’à saturation de la mémoire des fenêtres

d’ajout d’élève en cliquant un bon nombre de fois sur le bouton « ajouter éleve », et de

cette manière, bloquer l’application

Pour pallier à ce problème, nous avons conçu la classe JFrameListener, qui prend en

paramètre deux Component (principalement des JFrame). Ces deux objets correspondent à

la fenêtre mère et la fenêtre fille. La fenêtre fille est la fenêtre qui va s’ouvrir après un clic.

(sur un bouton par exemple). La fenêtre mère est la fenêtre dont va « provenir » la fenêtre

fille. (Pour poursuivre avec le même exemple, la fenêtre qui contiendra le bouton cliqué).

De cette manière il sera très facile de rendre inaccessible l’accès aux fenêtres en arrière-plan

(méthode *setEnable()*) lorsque l’écouteur détecte l’ouverture d’une nouvelle fenêtre.

Précisons que si le paramètre « mere » est à null, alors on considère que l’application doit se

terminer, et donc une fenêtre de pop-up viendra demander la confirmation de bien vouloir

quitter l’application, comme spécifié dans le cahier des charges.

*c - Accès à la base de donnée*

L’accès à la base de données se fait grâce à la classe ConnexionDB.

Pour se faire, on déclare la variable en static, dans le contrôleur principal.

*private static ConnexionBD connectionBD;*

De cette manière il n’est plus nécessaire de ré instancier une nouvelle fois la classe à chaque

accès à la base comme nous le faisions au début du développement.

Cette initiative rentre donc dans le cadre de l’optimisation de la mémoire.

*d - Importations*

Pour permettre à l’application de fonctionner dans son intégralité il est nécessaire d’ajouter

manuellement plusieurs .jar

Mais également d’importer le driver JDBC qui n’ »est pas initialement présent dans le projet.

Les autres importations concernent notamment la conversion en PDF. Il s’agit des fichiers

Itextpdf-5.2.0 et PDFRender-0.9.1

**Figure 7 Importation des librairies**

Conclusion

La difficulté, lors du développement, a donc principalement résidé dans la mise en place de

l’infrastructure MVC.

Ce projet nous a néanmoins permis d’acquérir des réflexes de programmation bien propre

au langage java, mais également de renforcer nos connaissances dans le domaine du

développement.