**Efrei++: gestionnaire d'école**

***Projet Programmation JAVA***

**DAVID Aurélien, RAMOS Adrien**



SOMMAIRE

[Efrei++ en quelques chiffres : 2](#_Toc354523158)

[Introduction 2](#_Toc354523159)

[Description de l'application. 3](#_Toc354523160)

[Liste des fonctions développées 4](#_Toc354523161)

[Réalisation de l’application 5](#_Toc354523162)

[Emploi de Design Pattern 5](#_Toc354523163)

[Diagramme UML de classes du model 8](#_Toc354523164)

[Liste des tests automatiques effectués 9](#_Toc354523165)

[Conclusion 10](#_Toc354523166)

[Annexe 10](#_Toc354523167)

# Efrei++ en quelques chiffres :

* 1 projet Open-source utilisant git.
* 6 différents design patterns utilisés parmi les plus connus.
* Environs 60 classes dont de nombreuses classes abstraites et interfaces.
* Environs 50 heures de code et de conception pour 2000 lignes de code rédigé intégralement par les 2 développeurs et concepteurs du projet.

# Introduction

En cette troisième année de l’EFREI, le cours de Programmation en Java est venu en remplacement de l’enseignement en programmation C++ dispensé l’année précédente. Ce changement a impliqué de nouvelles attentes: changer sa façon de concevoir un programme informatique, celui-ci passe d'abord par une phase d'analyse puis de conception.

Pour ce second projet, le sujet n'est plus libre: il s'agit de réaliser un gestionnaire d'école. Au-delà d’un projet scolaire de Java, nous souhaitons avant tout réalisé un projet bien conçus qui pourrait évoluer post-rendu de projet et être un bon plus sur un CV.

Il nous a donc fallu étudier longuement la façon dont nous allions aborder le gestionnaire, le concevoir et enfin l'optimiser. Cela a été rendu plus simple du fait que nous maitrisons bien le Java grâce à notre précédent projet: Drasus, un jeu de stratégie en réseau multi-joueurs.

Ce dernier projet a été pour nous l’occasion de travailler à une conception objet avancée, cette fois-ci nous choisi d’y consacrer encore plus de temps en utilisant notamment plusieurs design pattern.

# Description de l'application.

Nous avons décidé de faire de Efrei++ un gestionnaire de type clients/serveur, c'est à dire que plusieurs clients se connectent au même serveur. Le fonctionnement se déroule de la façon suivante: le client se connecte puis le serveur détermine ces droits d'utilisateurs, ce qui permet de limiter les actions de chacun. Un élève peut consulter ses notes tandis qu'un administrateur peut ajouter des élèves.

Ce projet a été réalisé en Java en utilisant la librairie graphique **Swing** ainsi que **RMI** pour la gestion de la partie réseau et **Junit** pour les tests unitaires. Ces choix nous ont permis de réaliser une interface utilisateur complète et agréable tout en utilisant le modèle **MVC** qui facilite grandement l'organisation du code.

La sécurité a été mise en avant puisque toutes les vérifications s'effectuent coté serveur notamment la gestion des droits d'accès. Les données sont également toutes stocké dans la base de données **SQLite** du serveur afin d'assurer la **sécurité** des celles-ci, elles transitent ensuite vers les différents clients.

Ainsi nos sources se composent de quatre **packages**, eux même divisés en plusieurs subpackages :

* **model,** qui contient la partie données du projet, c'est à dire les étudiants, professeurs.... mais également la récupération depuis la base de donnés (model.dao).
* **view**, qui comprend notre interface graphique les différentes actions.
* Le package **controller**, qui gère et vérifie les actions permettant de relier des packages sus-jacents à l'aide de plusieurs classes de contrôle.
* Et enfin notre package **test** qui contient les tests unitaires de notre programme, partie serveur.

Afin de concevoir ce projet facilement nous avons utilisé comme à notre habitude le gestionnaire de version **Git**. Le système de commit nous permettant de savoir sur quelle partie du code chacun des membres du binôme avait travaillé et de comprendre les modifications. De plus l'utilisation de différentes branches nous a permis de travailler chacun de notre côté sans gêner l'autre ni attendre la fin des modifications...

Nous veillons à rendre notre projet Open-Source une priorité, c'est pourquoi nous attachons une grande importance au code source qui a été rédigé intégralement en anglais. Les commentaires respectent la convention **Javadoc** afin de générer facilement une documentation complète.

## Liste des fonctions développées

Nous avons réalisé l'intégralité des fonctions demandées par le sujet:

1. Inscrire un étudiant à l’école, une majeure, un cours

2. Créer de nouveaux cours

3. Associer un professeur à un cours

4. Associer une note à un étudiant pour un cours donné

5. Alerter un professeur périodiquement pour les étudiants en difficulté jusqu’à ce que, le tuteur réagisse.

6. Rechercher un cours par titre

7. Sauvegarder l’ensemble des entités de l’application afin de pouvoir en cas de redémarrage retrouver les informations sur l’école.

Nous avons ajouté un système d'authentification séparant les droits entre l'administrateur, les étudiants et professeurs.

# Réalisation de l’application

## Emploi de Design Pattern

En codant ce projet nous sommes partis du principe que celui-ci allait être amené à évoluer et prendre de l'ampleur au fil de la conception via de nouveaux ajouts mais également au delà du rendu de projet !

La sécurité de l'appel des méthodes utilisant RMI se fait à l'aide du design pattern proxy de protection:

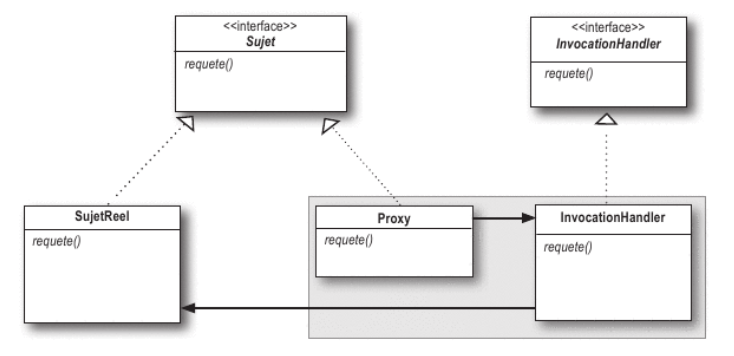


Figure 2 - Proxy de protection

Nous avons alors réalisé différents **InvocationHandler**:

* InvocationHandler**Anonymous**: lorsque le client est inconnu celui-ci peut seulement appeler la méthode login.
* InvocationHandler**Student**: le client a été identifié comme étudiant, celui-ci peut appeler les méthodes permettant de choisir ses cours et afficher ses notes.
* InvocationHandler**Teacher**: le client a été identifié comme professeur, celui-ci peut appeler les méthodes pour ajouter des notes, pour ces cours.
* InvocationHandler**Admin**: celui-ci peut appeler toutes les méthodes.

Rajouter un nouveau droit d'accès pour un chef de département par exemple nous est donc très simple !

Si une méthode est appelé alors que le client n'a pas les droits (un étudiant qui modifierais les notes d'un autre étudiant ou tentative de piratage), alors le serveur renvoi une **IllegalAccessException**.

Par défaut le client se connecte via RMI via l'interface RemoteController avec le proxy InvocationHandlerAnonymous, ensuite la méthode de login retourne au client un RemoteController avec le proxy associé à ses droits d'utilisateur.

Ce pattern peu connu permet d'utiliser RMI avec plusieurs clients simultanément est appelé Remote Session.

MainController est notre classe principale, elle instancie les différents Controller (DatabaseController,StudentController,TeacherController,CourseController...) toutes ces classes qui instancie des données du model sont des singleton pour éviter les duplicatas de données.

Un professeur est alerté lorsqu'un étudiant a reçu 3 notes inférieures à la moyenne, c'est typiquement le cas d'un pattern Observeur, nous avons implémenté directement celui proposé par Java dans java.util.Observer et java.util.Observable.

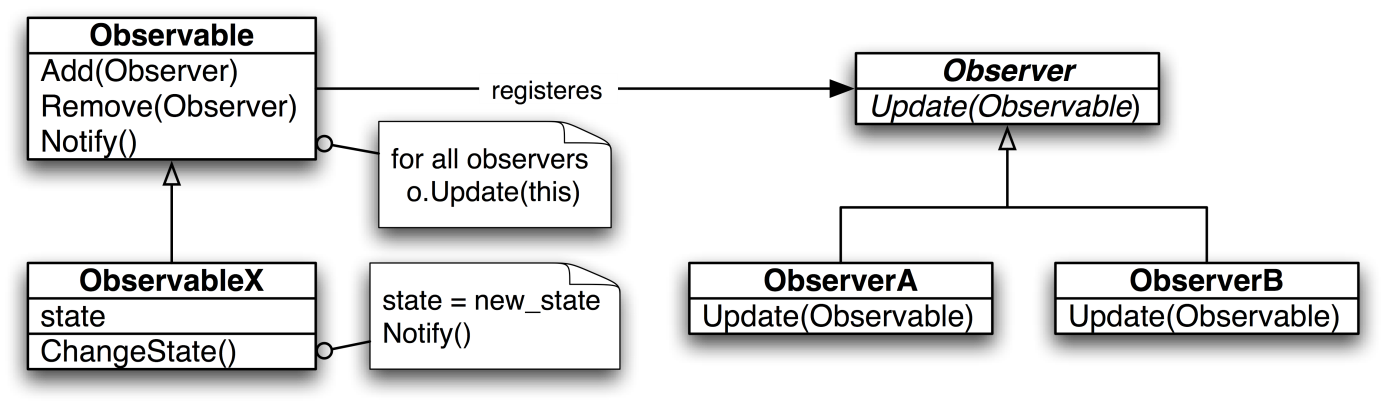


Figure 2 - Pattern Observer-Observable

La récupération des données depuis la base de données s'effectue à l'aide du pattern Dao et d'une factory, afin de pouvoir s'adapter, si l'on change par exemple la base de données pour du XML ou du JSON...

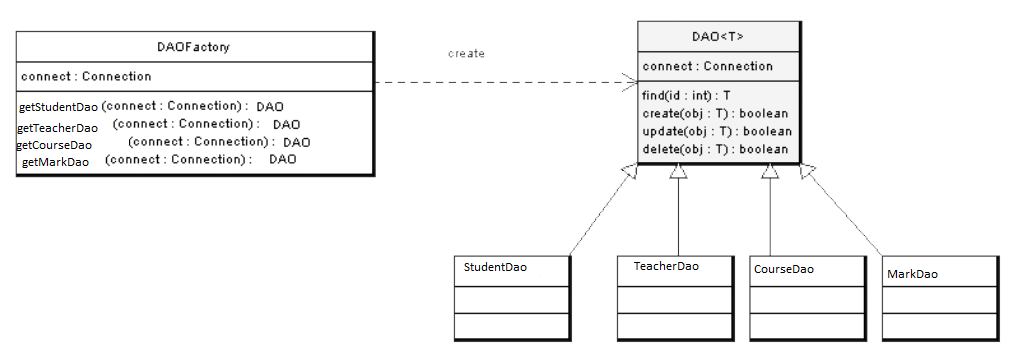


Figure 2 - Pattern Dao

## Diagramme UML de classes du model

## Liste des tests automatiques effectués

De part notre architecture nous contrôlons chacune des actions d'un utilisateur avant de modifier le model, ce qui implique de nombreux tests, de plus la base de données présente des contraintes d'intégrité afin d'assurer la pérennité des données.

En plus de ces tests nous avons accordé une importance aux tests unitaires via la bibliothèque dédiée aux tests Junit. Ces derniers sont destinés à vérifier que les méthodes du serveur qui sont appelées par le client fonctionnent correctement et ce malgré un ajout ou une suppression de code. Par exemple si l'on ajoute un champ "date de naissance" à un étudiant, les tests s'assureront du bon fonctionnement du code malgré la modification. Les tests s'assurent également que les contraintes d'intégrité sont respectées: supprimer un professeur entraine la suppression des liaisons entre les cours enseigné par le prof et ce dernier.

# Conclusion

Pour projet de Java nous sommes fiers du résultat obtenu surtout la facilité à laquelle nous avons réalisé la partie réseau en utilisant RMI.

Programmer ce projet nous a permis avant tout de consolider notre programmation en Java mais aussi d’avoir un certain recul sur la difficulté à produire un client/serveur sécurisé.

Pour terminer,

# Annexe

Github

L'intégralité du gestionnaire de version du projet est disponible de façon open source à l'adresse suivante:

<https://github.com/spyl94/efreiplusplus>

Sources

* Design Patterns - Tête la première, apprentissage des pattern et utilisation des diagrammes correspondant.
* Apprenez à programmer en Java - Site du Zéro, pour le pattern Dao et le diagramme associé.