Hayar Malak
De Wancker Maddy
Margely Gwendal

BUT Informatique S2
IUT A de Lille

Rapport SAE 2.01/2.02

Notre projet a été de concevoir une application de gestion de tutorat. Les responsables du tutorat sont les *enseignants* de la *ressource* concernée puis il y a les *Étudiants* ciblés qui sont les *Tuteurs* et les *Tutorés.* Un enseignant d'une ressource ne pourra faire l'affectation que des tuteurs proposant un tutorat dans la ressource qu'ils enseignent.

A. Lancement de l'application

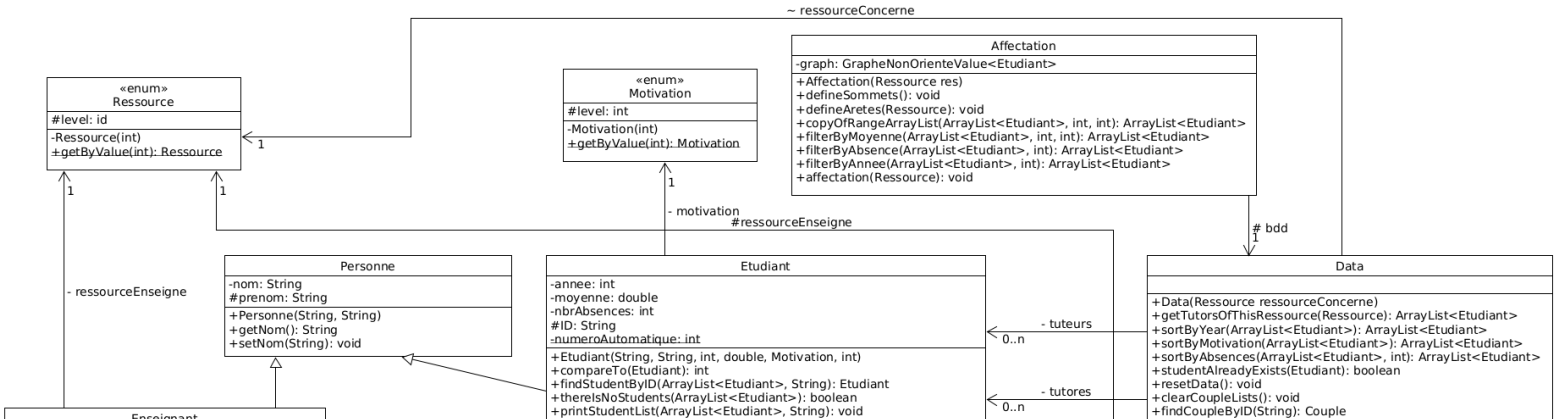
L'application contient déjà une base de données d'Enseignant, de tuteurs et de tutorés disponible pour commencer. Il est possible dans l'application de la modifier. Il est recommandé de la consulter dans le sous-menu *"2 : modifier la base de données"* avant de commencer toute affectation. Toutes les exportations sont consultables dans le répertoire *"res".*

Pour lancer l'application il suffit de se positionner dans le bon répertoire B-G3, de lancer un terminal pour exécuter un runnable jar mis à disposition avec la commande suivante : *java -jar app_tutorat.jar*

Si jamais vous rencontrez un problème lors de son exécution, il vaut mieux lancer l'application à partir *d'une console eclipse*, là c'est sûr que ça marchera.

PS : il se peut que certains caractères spéciaux ou des lettres avec accents apparaissent différemment sur votre terminal/console, car on ne peut pas contrôler comment ils sont encodés.

B. Diagramme UML



C. Mécanisme objet utilisés

Le différentes classes du projet ont été divisées en plusieurs packages :

- **1. app:** contient la classe de lancement de l'application
- **2. database :** contient toutes les classes construisant les objets pour la base de données *statique* de l'application.
- **3. affectation :** contient toutes les classes nécessaires aux affectations.
- **4. utils :** contient toutes les classes définissant divers outils nécessaires au fonctionnement de l'application (méthodes pour la dé.sérialisation, input sécurisé..).

La structure de données principale que nous avons choisis d'utiliser pour représenter les différentes listes de tuteurs, tutorés et les couples affecté est **l'arrayList**, car on avait besoin de structure ***de type liste*** pour représenter les deux parties du ***graphe d'affectation***, le choix l'arrayList nous a paru plus judicieux que par exemple une *linkedList* car on avait souvent besoin de trier nos différentes listes ou de récupérer un élément en particulier ou à un index en particulier.

- **1. app :**

App : Classe contenant le ***main*** qui lance l'application. En la lançant, on arrive sur une page de connexion où un professeur d'une ressource particulière devra entrer son ID pour se connecter. Une fois cela fait, on accède à un menu :

Que voulez-vous faire ?

1 : Affecter des couples tuteurs/tutorés

-> affectation manuelle/algorithmique des couples

2 : modifier la base de données

-> Supprimer/ajouter un étudiant, ou consulter leurs différentes listes.

3 : Exporter la base de données

-> Exporter les résultats de l'affectation et les éventuelles modifications de la base de données dans des des fichiers JSON.

4 : Se deconnecter

-> Sortir de l'application

- ## 2. database :

Personne : classe contenant des informations de base (nom, prénom), classe parente des classes **Enseignant** et **Étudiant**.

Ressource : Enum contenant les différentes ressources concernées par le tutorat : **DEV, WEB, ARCHI, MATHS**.

Motivation : Enum contenant les différents niveaux de motivation des étudiants : **FAIBLE, MOYEN et ÉLEVÉ**, avec des valeurs "poids" différentes. Une méthode permettant de récupérer l'enum à partir de son poids y est aussi définie.

Enseignant : ***classe héritant de Personne***, qui représente les enseignants utilisateurs de la plateforme. Chaque **Enseignant** possède un **ID** unique, une **ressource** enseignée. La classe contient également un **HashSet** commun (static) d'enseignants représentant leur liste, l'utilité d'avoir choisi un HashSet étant que cette structure de donnée *n'accepte pas les doublons* et nous permettra de ne pas avoir plusieurs fois le même objet Enseignant en son sein, une **DuplicateTeacherException** (Exception que nous avons définit et qui hérite de **Throwable**) est soulevée. Pour pouvoir itérer sur ce HashSet, nous avons ***implémenté l'interface iterable sur l'objet enseignant.***

Étudiant : Classe héritant de la classe **Personne**, elle contient les information communes à tous les étudiants, qui sont leurs moyennes, leur année d'étude, leur motivation et leur nombre d'absence, chacun de ces attribut pourrait être un critère d'affectation. ***Cette classe implémente l'interface comparable***, qui nous servira à classer une liste donnée d'étudiants selon un attribut choisi, ici la moyenne.

Tuteur : Classe héritant de la classe **Etudiant** (et donc aussi de **Personne**) ajoutant des spécificités propres aux tuteurs. Chaque tuteur a une seule **ressource** pour laquelle il propose un tutorat (ce qui permettra plus tard à l'enseignant de filtrer à l'affectation). *Sa moyenne D'Étudiant sera la base sur laquelle il sera classé en priorité pour l'affectation*, et son année d'étude peut être égale qu'à 2 ou 3, sa La classe contient une *Arraylist* commune (static) contenant chaque instance de Tuteur, qui est à chaque fois classée du Tuteur ayant la meilleure moyenne à celui qui a la pire.

Tutoré : Classe héritant de la classe ***Etudiant*** (et donc aussi de ***Personne***) ajoutant des spécificités propres au tutoré en plus de celle qu'il hérite : un relevé de note modélisé sous forme de *HashMap*, avec comme *clé* la **Ressource** et comme *valeur* la note (*Integer*) relative à cette ressource. Dans le cadre d'une affectation, on récupérera dans chaque relevé de note celle qui nous intéresse. La classe contient une *Arraylist* commune (static) contenant chaque instance de *Tutoré*.

Couple : Classe créant des objets représentant chaque couple de tuteurs/tutorés affecté par l'algorithme, chaque **Couple** possède un *ID* unique qui l'identifie créé à partir d'un numéro incrémenté instanciation.

- **3. Affectation :**

Data : Classe encapsulant les données utilisées pendant l'affectation, afin d'y pouvoir y opérer des changements sans pour autant altérer les listes de tuteurs/tutorés de base. Elle contient trois arrayLists (static) reprenant la même liste de tutorés, mais uniquement les tuteurs dans une ressource en particulier, cette ressource étant celle de l'Enseignant connecté (un enseignant de DEV ne pourra ainsi pas faire d'affectation avec tuteurs de MATHS). Et c'est dans cette classe que sont définies les méthodes qui permettent d'abord de filtrer les étudiants en fonctions de leurs moyennes, année d'études, et/ou de leurs nombre d'absence (Exemple : n'avoir que les tuteur qui ont plus de 12, les tutorés qui ont moins de 10, uniquement les tuteurs de 3ème année, les étudiants avec moins de 5 absences…). Il y'a également des méthodes pour la gestion de la base de données, la méthode finStudentByID soulève une **ObjectNotFoundException** (Exception que nous avons définit et qui hérite de **Throwable**) si aucun étudiant avec l'ID passé en paramètre n'existe. Il y a également une troisième ArrayList contenant les couples affectés par l'affectation algorithmique ou manuelle (méthode **ManuallyAssign).**

Affectation : Classe définissant toutes les méthodes nécessaires à l'affectation algorithmique des couples tuteur/tutoré, elle utilise une librairie permettant d'implémenter l'algorithme hongrois de calcul d'affectation optimal sur un graphe bi-parties, ces deux parties représenté d'un côté par une ArrayList de tuteur, et de l'autre par une ArrayList, toutes deux déjà filtré (ou pas) par les méthodes de la classe ***Data.*** On y retrouve des méthodes permettant la mise en place des critères à prendre en compte (ou pas) dans l'affectation : l'année d'étude, la motivation, et/ou le nombre d'absence, l'ordre d'importance (pondération) se fait en fonction de celle que l'on choisit au début. A partir de là on définit les sommets (**DefineSommets**) et les arêtes du graphes (**DefineAretes**). La méthode **Affectation** réalise le calcul et remplit la liste des couples assignés (de la classe *Data*).

- **4. Utils :**

Keyboard : Classe contenant diverses méthodes statique permettant diverses input sécurisés. Elle contient un constructeur mis en private pour éviter son instanciation ailleurs vu que c'est une classe d'utilité

FileUtils : Classe contenant des outils pour la création des fichiers. Elle contient un constructeur mis en private pour éviter son instanciation ailleurs vu que c'est une classe d'utilité.

GsonUtils : Classe contenant les méthodes de sérialisation et de désérialisation des différents objets (données) de l'application. Cette classe utilise les méthodes de *la librairie Gson* pour les représenter en fichiers ***JSON***. Elle contient un constructeur mis en private pour éviter son instanciation ailleurs vu que c'est une classe d'utilité.

DuplicateTeacherException : Exception soulevée s'il y a plus ***d'un enseignant*** avec le même nom et prénom dans une ressource.

ObjectNotFoundException : Exception soulevée si dans une fonction de recherche l'objet recherché n'existe pas.

D. Tests :

1. Classes de test

Nous avons réalisé quatre classes de test qui recouvrent toutes les méthodes ayant un sens à être testées (les fonctions d'ajout manuelle d'un tuteur/tutoré, fonction d'input…. sont par exemple écartées).

a. La classe AffectationTest :

Classe testant toutes les méthodes concernant l'*Affectation* (méthodes de la classe Data/Affectation). Elles testent toutes les méthodes de filtrage des

étudiants par années/absences/motivation. Et aussi les tests sur l'affectation selon un critère particulier (pour tester les fonctions qui mettent en avant ce critère une par une).

b. La classe EnseignantTest :

Classe testant les méthodes concernant les *Enseignant* : ***findTeacherByID(String id)*** et ***findTeacherByRessource(String id).***

c. La classe MotivationTest :

Classe testant l'unique méthode de la classe *Motivation* : **getByValue(int value)**

d. La classe RessourceTest :

Classe testant l'unique méthode de la classe *Ressource* : **getByValue(int value)**

2. Utilisation de git :

Nous avons utilisé git afin de pouvoir travailler chacun dans notre machines et partager nos travaux, nous avons fait au total plus d'une quarantaine de commits. Quant à la répartition de travail, Malak a fait l'implémentation des graphes ainsi que la modélisation de l'application, Gwendal a fait la conception de l'ihm sur scène builder tandis que Maddy l'a codé en javafx et a réalisé les classes de tests. Nous nous assistions tout au long du projet dans les différentes tâches de chacun.

3. Qualité du code

Nous pensons avoir atteint une bonne qualité de code en sens de l'utilitaire PMD, que nous avons utilisé afin de purifier notre code. *La plupart des erreurs qui restent concernent la classe IHM* que nous avons à ce moment là pas encore terminé pour vérifier. *Le fichier de règle* que nous avons utilisé se trouve dans notre répertoire **B-G3**

4. Bilan et réflexion individuelle

Malak

Ce projet m'a donné l'occasion d'apprendre comment concevoir une vrai application. J'ai pu réfléchir sur les différents mécanismes objets les plus adaptés pour répondre à ses besoins, j'ai ainsi principalement travaillé sur sa conception, notamment dans l'implémentation de l'algorithme de calcul d'affectation qui a été un exercice assez difficile, mais aussi dans l'écriture des différentes classes de l'application.

Je pense que le projet s'est bien déroulé, je suis satisfaite de notre travail. On a eu comme tout projet des difficultés notamment dans la répartition des tâches mais nous avons réussi à les surmonter vers la fin du projet

Maddy

Ce projet de développement d'application m'a permis de travailler sur les différents aspects du développement d'une application de a à z. J'ai principalement travaillé sur l'élaboration de l'UML et les classes de test, et j'ai ainsi corrigé des parties du code en fonction des résultats attendus par ces tests. Je participe aussi beaucoup sur la partie interface pour le rendu d'IHM.

Selon moi, le projet s'est plutôt bien déroulé, malgré que nous ayons manqué de communication au sein de notre équipe, et que nous ayons eu du mal pour nous répartir les tâches la plupart du temps.

Gwendal

Lors de ce projet j'ai pu apprendre comment se déroulait vraiment la conception d'application réelle, ce qui a été nouveau et formateur par rapport à ce qu'on faisait en TP par exemple. Pour ma part j'ai aidé à la conception dans l'écriture du code mais je me suis surtout occupé de l'aspect utilisateur de l'application. J'ai réalisé son interface en très grande partie avec sceneBuilder, ainsi que Figma, un outil que je maîtrisais déjà.

Globalement ce projet s'est pour moi bien déroulé, on a eu quelques difficultés notamment en ce qui concernant la répartition des travaux mais on a pu communiquer pour remédier à ça.