Développement d'une application

S2.01-S2.02,
Attribution de tutorat

Git : https://gitlab.univ-lille.fr/sae2.01-2.02/2022/A-G5
Commit : 851226b1

8 JUIN

IUT de Lille, BUT Informatique S2-A
Alexandre Herssens
Léopold Varlamoff
Théo Franos

Université de Lille

Sommaire

I. Description

A. Exécution

L'exécution des scénarios que nous avons développés pour représenter notre implémentation des contraintes (voir point ci-dessous) est faite grâce au fichier .jar fourni dans l'archive du rendu. Il suffit d'entrer la commande « *java -jar <nom-fichier.jar>* » et vous serez redirigés vers la classe principale qui vous guidera.

B. Scénario

En exécutant notre application, vous aurez le choix entre deux scénarios.
Le premier est l'exemple du tutorat de web du début du S1, et met en avant les fonctionnalités implémentées : utilisation de nos classes, import d'étudiants à partir d'un fichier CSV, affectations avec différentes contraintes (deux étudiants ensemble, se focaliser sur les absences, etc…), ajout et suppression manuelle d'étudiants.
Le second est un exemple fictif de tutorat de système et d'architecture mis en place à la fin du S1. Il met en avant d'autres fonctionnalités, comme la gestion de plusieurs tutorats dans différentes matières et l'utilisation de filtres à partir d'un fichier JSON. Il met également en avant les fonctionnalités utilisées lors du premier scénario, mais d'une manière différente.

C. Fonctionnalités implémentées

1. Département

Notre programme permet de créer un département d'une université en lui donnant un nom, par exemple informatique.
Nous pouvons lors de sa création lui passer deux collections de *Student* et *Teacher*.
Ensuite, il est possible de charger une liste d'étudiants enrichie en notes, absences et motivation depuis un fichier CSV. Il est également possible d'ajouter des personnes, étudiants ou professeurs uniques ou sous forme de collection.
Concernant les tutorats, il est possible d'en créer pour une matière donnée avec un enseignant spécifié ou non puis d'y inscrire des étudiants uniques ou sous forme de collection ou bien à partir de la liste des étudiants du département, dans ce cas, seuls ceux disposant d'une note dans la matière du tutorat pourront y être inscrits.
Il est possible de connaitre le nombre d'étudiants, de professeurs ou encore de tutorats.

2. Enseignant

Un enseignant peut être créé à partir d'un nom, facultativement d'une matière ou liste de matières ou bien du nom de la matière (*ex : R101*).
Des matières peuvent lui être ajoutées.
Par défaut le poids des différents critères (enum *Coefficient*) pour le tutorat est fixé à 1 mais il est possible de les changer pour une valeur entre 0 et 5 (vérifiées et passées par défaut en cas de souci).
Les valeurs par défaut sont modifiables.

3. Étudiant

Soit tutorés soit tuteur, le type passé lors de l'instanciation définira les valeurs par défaut. Il est possible de lui ajouter une moyenne pour une matière, de modifier ses absences ou sa motivation le reste des traitements est effectué à l'instanciation.
Tutorés et tuteurs ont un calcul de poids spécifique à chacun.
Les tuteurs ont un nombre max de tutorés (1 ou 2) utilisé si le poly-tutorat est actif pour ce tutorat qui peut être modifié. Si le tuteur autorise 2 tutorés et que le poly-tutorat est actif, le programme peut le dupliquer en cas de manque de tuteurs. Le duplicata de tuteur possède le poids du tuteur modifier par un attribut éditable.

4. Tutorat

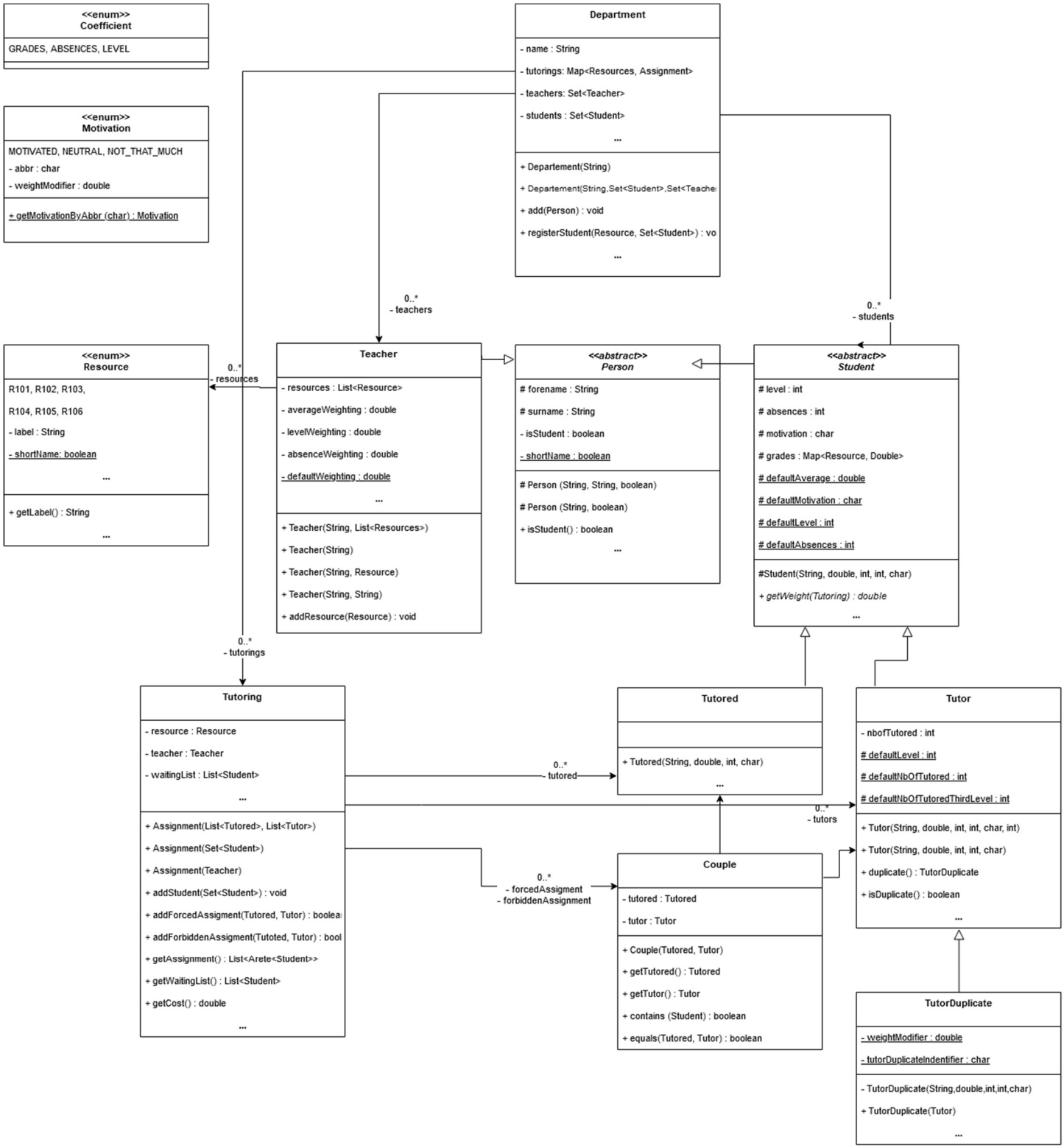
Le tutorat est créé obligatoirement à partir d'une ressource mais peut être plus spécifié comme avec une liste d'étudiants ou de tuteurs et tutorés et ou avec un enseignant renseigné.

Le tutorat tient un registre des moyennes des notes ou absences des tuteurs et tutorés distinctement, contient des affectations forcées ou interdites que l'on peut ajouter ou supprimer.

Il est possible d'ajouter un ou plusieurs étudiants MAIS qui ne sont pas forcément inscrits dans le département et ce dernier n'en aura donc aucune trace, nous n'avons pas eu le temps d'implémenter cette vérification / ajout automatique.

Le tutorat contient un calcul d'affectation mis à jour dès qu'il est requêté avec une liste d'attente des étudiants non affectés.

II. Diagramme UML



III. Mécanismes

A. Encapsulation

Mécanisme le plus utilisé dans notre projet, il nous semble le plus adapté pour doter nos classes des différents attributs nécessaires à la modélisation d'un système de tutorat comme des collections d'étudiants (*Student*), de professeurs (*Teacher*) et de tutorats (*Tutoring*) du département (*Department*), celles de tutorés (*Tutored*), tuteurs (*Tutors*) et d'affectation forcées ou interdites sous forme d'un set de *Couple (Tutored, Tutor)* d'un tutorat.

B. Héritage

1. Extension de classes

C'est la solution que nous avons trouvé la plus naturelle afin de modéliser les différents types de personnes (*Person*) qu'il s'agisse de professeurs comme d'étudiants qui peuvent être des tuteurs ou tutorés et partageant un certain nombre d'attributs en commun comme le prénom et nom sur les personnes, puis les notes, la motivation ou encore les absences pour les étudiants, les professeurs en étant dépourvus il était logique de les faire prendre un autre chemin en les dotant des matières qu'ils enseignent.

2. Implémentation d'interfaces

Tous nos objets devant être traités de manière semblable héritent soit de *Person* soit de *Student*, nous avons donc préféré l'utilisation de méthodes abstraites et de méthodes définies par les parents utilisant les attributs des descendants.
La seule implémentation effectuée est celle de *Comparator<Student>* utilisée pour les deux classes de comparaison des étudiants (*StudentComparator* et *StudentPriorityComparator*).
Nous aurions pu créer un interface *Duplicable* définissant *isDuplicate() : boolean* qui aurait été implémenté sur *TutorDuplicate* mais même si Tutor dispose déjà de cette méthode, nous n'avons pas, à notre état de développement, été confrontés au besoin de l'utiliser.
De même pour les fonctionnalités de tutorat comme *getWeight(…) : double* mais nous avons également préféré la définir dans *Student*.

C. Abstraction

1. Classes

Comme expliqué un peu plus haut, ayant privilégié l'héritage de classes plutôt que d'interfaces, nous avons eu recours à des classes abstraites, *Person* et *Student* qui ne peuvent donc pas être instanciées mais sont pratiques afin de gérer les attributs communs à tous les descendants.

2. Méthodes

Unique méthode abstraite utilisée, *getWeight(…) : double* permet le calcul du poids d'un étudiant pour un tutorat donné, comme le calcul est différent pour les tuteurs et tutorés mais indispensable à l'exécution de notre programme.
Nous avons le plus souvent préféré la redéfinition des méthodes parentes comme par exemple pour les *toString() : String* qui s'appuient toutes sur une structure commune mais sont plus riches en attributs plus l'on descend dans l'héritage.

D. Statisme

1. Attributs

Les différents attributs statiques nous permettent la définition de valeurs par défaut comme par exemple le niveau d'un étudiant, sa note, son nombre d'absences quand l'un des paramètres est manquant ou erroné. Ils nous permettent aussi de modifier le comportement de méthodes comme les *toString() : String* qui affichent soit l'instance en détail soit sous forme réduite *(<Prénom> <Nom>)* grâce à l'attribut *Person.shortName* qui peut être modifié à l'aide de la méthode *Person.setShortName(boolean) : void*.

2. Méthodes

La plupart de nos méthodes statiques sont des *getters* et *setters* d'attributs statiques, cependant nous disposons de classes utilitaires qui regroupent des méthodes statiques permettant le traitement de listes de nos objets par exemple, nous en parlerons un peu plus bas.

3. Énumérations

Notre programme implémente actuellement deux énumérations *Resource* qui définit les différentes matières cruciales enseignées sous la forme *<Resource>(<libellé>)* par exemple R101(« Initiation au développement ») et également *Coefficient* qui définit les différents critères utilisés pour le calcul de l'affectation, un enseignant conserve ses coefficients dans une table de type *EnumMap<Coefficient, Double>*.

Nous aurions également souhaité implémenter l'énumération *Motivation* afin de remplacer l'implémentation actuelle par caractère de '*A*' à '*C*' qui dépend d'une boîte à outil pour retourner la bonne valeur.

4. Classes utilitaires

Notre programme dispose de classes utilitaires qui permettent, comme décrit plus haut, d'effectuer des actions sur des collections de nos objets. La classe *Couples* permet de réaliser des actions sur des collections de *Couple* comme par exemple *containsCouple(Set<Couple>, Tutored, Tutor) : boolean* qui permet de nous dire si un set contient un couple des deux étudiants passés en paramètres ou encore *containsStudent(Set<Couple>, Student) : boolean*. Cette méthode est utile lors du calcul du poids d'une arête pour une affectation forcée ou interdite.

La classe *ToolsCSV* permet la gestion des fichiers CSV pour la persistance des données et l'import des étudiants avec leurs notes grâce à la méthode *importStudent() : Set<Student>*.

Quant à *Persons*, elle ne contient actuellement qu'une méthode *getPerson(String, List< ? extends Person>) : Person* qui sert à retrouver une personne en fonction de son nom dans une collection de personnes.

E. Surcharge et chaînage

Certaines de nos fonctions sont surchargées comment par exemple *addStudent(Student) : boolean* et *addStudent(Collection<Student>) : boolean* dans *Department* ce qui permet à l'aide d'un même appel soit d'ajouter un unique étudiant ou bien une collection, la méthode *add(Person) : boolean* toujours dans *Departement* fait un appel à *addStudent(Student)* ou bien à *add(Teacher)* en fonction de la nature de la personne passée en paramètres.
Le chaînage nous permet d'éviter la redondance de code et de gérer les cas par défaut d'exécution du programme.
Un autre exemple est celui de la surcharge et du chaînage des constructeurs qui font soit appel à un autre constructeur de l'instance soit au constructeur parent.

F. Contrôle d'accès

L'encapsulation représentant une grande partie de notre programme, nous avons défini la plupart des *getters* et *setters* des attributs afin de pouvoir les consulter et modifier peu importe le contexte. Certains attributs et méthodes sont *protected* comme par exemple les constructeurs des classes abstraites afin de limiter leur accès, d'autres encore sont *private* pour des questions de sécurité, par exemple le *setter* de l'année d'un étudiant qui ne peut avoir lieu que lors de l'instanciation, ou encore l'appel au constructeur *TutorDuplicate(String, int, int, motivation)* qui n'est accessible que par chaînage du constructeur *TutorDuplicate(Tutor)*.

G. Gestion des entrées et sorties

Notre programme ne demande pas d'entrée utilisateur, la seule gestion d'entrée / sortie concerne les fichiers CSV des étudiants, tant pour leur enrichissement que pour leur lecture à la création de la liste d'étudiants du département.
Nous avons également implémenté une lecture de fichier JSON pour permettre aux professeurs de définir des filtres à appliquer sur les listes d'étudiants. Nous avons fait le choix du JSON pour sa meilleure lisibilité quand le nombre de données à entrer est restreint. La lecture du fichier est faite grâce à la librairie *JSONObject* fournie en TP.

H. Gestion des erreurs

Nous avons dû faire un choix en ce qui concerne les erreurs. Par exemple, lorsque nous instancions n'importe quel objet enfant de *Student*, il faut que celui-ci respecte certaines conditions : son niveau doit être compris entre 1 et 3 (BUT1, BUT2 ou BUT3), sa moyenne ne peut pas être en dessous de 0 ou au-dessus de 20, et sa motivation doit être A, B ou C.
Nous avons hésité entre plusieurs manières de forcer le respect de ces contraintes. Nous avons d'abord décidé de lever une exception lorsque l'utilisateur entre une valeur non permise. Ces erreurs étaient des *RuntimeException*, ce qui nous n'obligeait pas de les traiter ou de les propager : elles arrêtaient simplement l'exécution dans la plupart des cas, pour éviter toutes erreurs dans les calculs.
En revanche, lorsque nous développions les tests, nous avons trouvé que la levée d'exceptions rendait les tests beaucoup plus complexes et diminuait notre possibilité de tester : nous avons donc opté pour une autre solution.
Cette solution est la mise en place de valeurs par défaut : la majorité des classes comportent des attributs statiques par défaut, qui sont modifiables au bon vouloir du professeur ou du chef de département. Si l'utilisateur venait à créer un tuteur avec un nombre d'absence négatif, ou avec une motivation évaluée à L, alors la valeur par défaut définie s'appliquera.
Au final, nous avons donc limité la levée d'exception, et nous avons privilégié le remplacement par des valeurs par défaut lorsque cela était possible.

IV. Qualité

A. Tests

Nous nous sommes efforcés de tester un maximum de fonctionnalité grâce à des classes de tests unitaires et la librairie JUnit. Nous avons utilisé la même méthodologie que les tests fournis dans le cadre de la ressource R2.01 et R2.03, avec la majorité des tests qui couvrent une seule méthode, sauf dans le cas du test de notre tutorat : la très grande partie des méthodes étant privées, nous avons décidé de tester les différents scénarios possibles.

B. Outils de qualité

Afin d'assurer la qualité de notre code, nous nous sommes servis de PMD avec comme fichier de règles celui utilisé actuellement les étudiants de DUT Informatique S3 nommé « Ensemble des règles considérées en M3301 DUT Info S3 de Lille », auquel nous avons retiré certaines règles que nous jugions inutiles ou trop avancées pour notre projet.
Parmi les règles que nous n'avons pas suivies, on peut citer la documentation qui n'a pas été écrite pour toutes les méthodes, classes & attributs, et les tailles des noms de variables et paramètres (nous avons préféré des noms longs et explicites lorsque c'était nécessaire).
De plus, certaines règles sur le nombre de méthode par classe ont été ignorées, car elles nous empêchaient de surcharger des méthodes.
Les règles concernant des concepts que nous n'avons pas vus cette année et qui nous semblaient compliquées à régler à notre niveau n'ont également pas été suivies, comme la complexité cyclomatique, la loi de Déméter, ou encore les « God Object », et, en général, les règles que nous n'avons pas suivi pendant les TPs et les cours de cette année.
Enfin, certaines règles n'ont pas été suivies pour les méthodes de tests, car nous avons privilégié la couverture et l'exhaustivité de nos tests.

C. Git

Le répertoire git étant disponible en permanence, nous partagions nos idées lors des autonomies et dans un groupe de discussion afin d'organiser le travail puis chacun travaillait en fonction de ses disponibilités.

Cette disponibilité était particulièrement pratique comme nous travaillions tant à l'IUT que depuis chez nous.

Quand un changement de philosophie important devait être opéré, une nouvelle branche était créée (*ex : abandonner les exceptions pour des valeurs par défaut lors de l'instanciation de descendants de Person afin de pouvoir plus complètement tester les fonctionnalités*). Si le changement était approuvé, alors cette branche et main étaient fusionnées.

L'utilisation de l'historique nous a été utile lorsque nous avons dû récupérer une version antérieure du code pour vérifier qu'aucune bêtise n'avait été commise.

A ce jour, Léopold est à 59 commits (>40%), Alexandre à 70 (>50%) et Théo à 9 (~6%).

V. Bilan

A. Alexandre

Concernant ma contribution au projet, j'ai développé les énumérations, Department, Teacher et Person et les articulations avec ses descendants.
Je suis également à l'origine de l'implémentation actuelle de TutorDuplicate, des attributs de classe par défaut et d'une grande partie de l'UML.
Anecdotiquement, j'ai réalisé la classe qui enrichit les données pour tester et les stocke dans un fichier csv ainsi que celle qui permet de lire ces données et de générer des étudiants.
Léopold ayant réalisé une grosse partie du développement, je l'aidais le cas échéant à mettre au clair ses solutions, je lui suggérais également les implémentations qui me semblaient les plus adaptées.
Faisant la chasse à la redondance et voulant augmenter le confort d'utilisation, j'ai participé activement à la surcharge, ainsi qu'au chaînage des méthodes.
Quant aux tests, j'ai réalisé un peu moins de la moitié de ceux existant. La réalisation de ces derniers m'a permis d'être confronté aux lacunes du programme et donc de les implémenter à la volée, de même quand des erreurs était découvertes pour leur correction.
Rétrospectivement j'aurais aimé pousser davantage le développement, de le compléter avec des fonctionnalités absentes comme la sauvegarde d'une affectation et qui est clairement à notre portée ou encore des contrôles et des fonctions de confort comme l'ajout automatique des étudiants d'un tutorat à ceux du département s'ils n'y sont pas présents. Cette liste n'est pas exhaustive, et si elle l'était, ce document ferait 50 pages.
Je suis globalement satisfait du travail que Léopold et moi avons réalisé malgré le handicap en termes de force de travail auquel nous avons été confrontés et je suis heureux d'avoir pu compter sur lui pour ce projet.

B. Léopold

En ce qui concerne ma participation au projet, j'ai essayé de travailler sur tous les aspects du projet pour pouvoir m'entraîner sur tous les concepts de développement objet. Plus particulièrement, j'ai écrit une grosse partie des classes *Tutoring*, *Student*, *Tutored* et *Tutor*, ainsi que sur la classe utilitaire *Tools*, même si celles-ci ont changé de design au fil du temps.
En ce qui concerne les tests, j'ai réalisé environ la moitié des classes de test, pour une trentaine de méthodes de tests unitaire. Je préfère les réaliser après avoir développé les fonctionnalités plutôt qu'avant (pas de TDD) car je trouve que la réalisation de test est un bon prétexte pour revenir sur son implémentation, et peut être trouver une meilleure manière de faire en relisant ce que l'on a fait par le passé.
Même si ce n'était pas demandé, j'ai également essayé de documenter un maximum des méthodes que j'ai réalisé. J'aime beaucoup lire des documentations (bizarrement) et donc ce projet était un bon entraînement pour essayer d'en produire une à grande échelle, même si nous savons que notre code ne sera pas réutilisé par quelqu'un d'autre que nous même. Malheureusement, le manque de temps et de moyens humain ne m'ont pas permis de tout commenter.
En conclusion, ce projet m'a permis de mettre en pratique les connaissances que nous avons acquises lors des cours de R2.01 et R2.02 plus concrètement et d'implémenter pour la première fois toutes ses connaissances au sein d'un seul gros projet Java qui utilise avec pertinence les concepts de développement objet. De plus, cela m'a permis d'apprendre à travailler à plusieurs en même temps sur un projet en utilisant des outils qui permettent les résolutions de conflits et une bonne qualité de code.

C. Théo

VI. Information sur le code source

Dépôt git :	https://gitlab.univ-lille.fr/sae2.01-2.02/2022/A-G5
Numéro de commit :	851226b1
SHA :	851226b132a933d69060cc9792f9f22fa44a2ea7
Classe principale :	main.Main.java
Packages principaux :	oop, utility, graphs
Packages de tests :	graphstests, ooptests, utilitytests