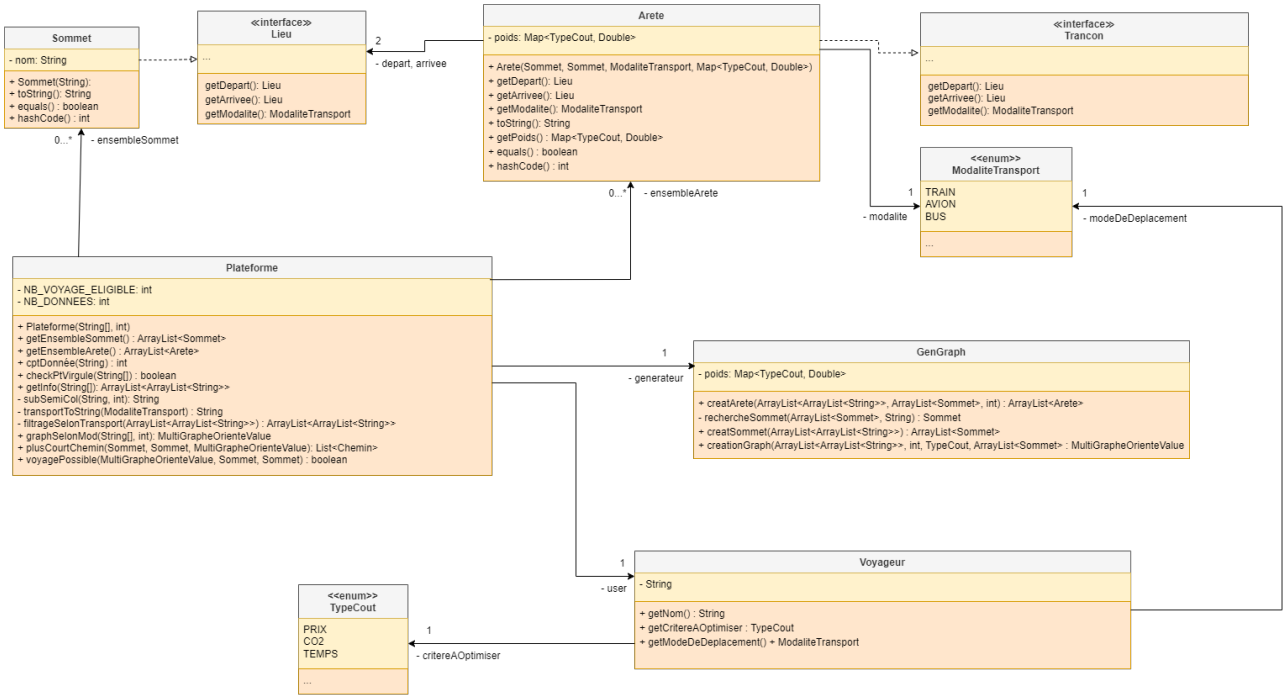
GAMBIRASIO Florian, DEHAINE Alexandre, PINTO Cristobal

Rapport POO S2.01/S2.02

Version 1 :

Pour utiliser l'application « Plateforme » il faut instancier un Objet plateforme, avec en paramètre un tableau de String qui correspond aux données représentant le réseau, le nombre de Voyage maximum que vous voulez afficher en résultat de la recherche, un Voyageur (utilisateur)
L'utilisateur est de la classe Voyageur défini par un nom, un mode de transport et le critère qu'il cherche a optimiser pour son trajet (prix, pollution, temps).
Ensuite la méthode « graphSelonMode » crée un graphe respectant les critère de l'utilisateur, en prenant en paramètre les donnée et le nombre de poids de chaque arêtes de ces donnée. La méthode « voyagePossible » détermine si il existe un chemin entre 2 ville avec les critère de l'utilisateur et la méthode « plusCourtChemin » revoie la liste des chemin entre 2 ville respectant les critère de l'utilisateur.

UML :



Analyse technique :

Lecture des arêtes

Afin de pouvoir lire les données fournis sous la forme:
"villeA;villeB;Train;60;1.7;80",
"villeB;villeD;Train;22;2.4;40",
"villeA;villeC;Train;42;1.4;50"

Alexandre a créé la fonction "public ArrayList<ArrayList<String>> getInfo(String[] data)" qui prend en paramètre un tableau de string, afin de pouvoir prendres plusieurs arêtes en même temps,

et qui, à l'aide de sa fonction “private String subSemiCol(String chaine, int prevIndex)” retourne une ArrayList d'ArrayList de String, la 1ère ArrayList contient plusieurs ArrayList qui eux même contiennent les String fournis décomposées et exploitable. Par exemple, si on prend le tableau de String fourni plus haut, si l'on veut la ville d'arrivé de la deuxième arête, cet à dire villeD, on écrit : variableArrayList.get(2).get(1), ce qui est égale à "villeD".

Pour faire ça, il utilise donc la fonction subSemiCol qui renvoie une fragementation de la chaine fournis, jusqu'au prochain ; en partant de l'index fournis en paramètre.

Cette fonction ne vérifie pas si les données fournies sont correctes, Cristobal a donc créé la fonction “checkPtVirgule(String[] data)” qui vérifie si les données sont au format attendu, il faut donc l'appeller avant d'utiliser “getInfo” pour éviter des erreurs si les données fournies sont invalide.

Transformation des données en graphes

Pour définir les sommets et les arêtes d'un graphe, Florian a écrit les classes **Arete**, qui implémente l'interface **Trancon**, et **Sommet**, qui implémente l'interface **Lieu**.
Une Arete est définie par un départ et une arrivée, tous deux de type **Lieu**, un mode de déplacement de type **ModaliteTransport**, et les différents poids possibles de l'arête dans une HashMap de type **HashMap<TypeCout,Double>**.

Pour obtenir le graphe souhaité, Florian a écrit la classe **GenGraph** qui permet :
- La création d'une liste de sommets à partir de données de type String (en vérifiant qu'il n'y ait pas de doublons car il ne sait pas utiliser de Set), via la fonction “public ArrayList<Sommet> creatSommet(ArrayList<ArrayList<String>> data)” qui parcourt une liste de String, vérifie qu'il n'y a pas de doublons, puis transforme les String sélectionnées en Sommet.
- La création d'une liste d'arêtes à partir de données de type String, une liste de Sommet et le nombre de "poids" que les arêtes auront, via la fonction “public ArrayList<Arete> creatArete(ArrayList<ArrayList<String>> data, ArrayList<Sommet> listSommets, int nbModalite)”. Pour fonctionner, cette fonction nécessite que les données en paramètre contiennent au minimum 4 éléments, dont 3 avant les "poids". De plus, pour que les poids soient correctement placés dans la HashMap de l'arête, les valeurs des poids doivent être dans le même ordre que les valeurs de l'énum TypeCout (ici PRIX, CO2, TEMPS).
- La création de graphes de type MultiGrapheOrienteValue via la fonction “public MultiGrapheOrienteValue creationGraph(ArrayList<ArrayList<String>> data, int nbModalite, TypeCout modalite, ArrayList<Sommet> listeTotalSommet)” qui utilise la fonction de création de liste d'Arêtes la liste complète des Sommets (pour palier à leur suppression suite au filtrage).

Représentation des utilisateurs

Pour représenter les utilisateurs, Florian a écrit la classe **Voyageur** qui définit un voyageur par un nom, le critère qu'il cherche à optimiser en se déplaçant (Critère de type TypeCout) et son mode de déplacement de type ModaliteTransport.

Filtrage des données

Pour filtrer les données afin de sélectionner un mode de déplacement choisi par l'utilisateur, Florian a écrit la fonction “private ArrayList<ArrayList<String>> filtreSelonTransport(ArrayList<ArrayList<String>> data)” dans la classe **Plateforme** qui renvoie des données en excluant les lignes ne contenant pas le mode de transport spécifié en paramètre.
Le défaut de cette fonction est qu'après son exécution, les nouvelles données obtenues peuvent perdre des sommets s'ils sont uniquement reliés par un mode de transport différent de celui spécifié

en paramètre, d'où la présence de la liste des sommet en attribut de la plateforme afin de remédier a ce problème.

La fonction "public MultiGrapheOrienteValue graphSelonMod(String[] data, int nbModalite)" permet de créer un graphe en utilisant un tableau de String qui sert de données, le nombre de "poids" que les arêtes auront, le critère priorisé par l'utilisateur, et son mode de déplacement.

Possibilité d'effectuer un voyage

La fonction "public boolean voyagePossible(MultiGrapheOrienteValue graphe, Sommet depart, Sommet arrivee)" permet de vérifier si un voyage est possible ou non entre 2 villes avec un moyen de transport sélectionné. Le choix du moyen de transport est effectué lors de la création du graphe passé en paramètre de cette fonction. Étant donné que le filtrage va supprimer les arêtes ne correspondant pas au moyen de transport de l'utilisateur, cette fonction va donc vérifier si les villes de départ et d'arrivée sont dans le graphe, et s'il existe ou non un plus court chemin allant du départ à l'arrivée.

Version 2 :

Utilisation :

L'utilisation de cette 2e version est la même que pour la première version sauf qu'il faut désormais lors de l'instanciation de la Plateforme ajouter en plus en paramètre une String correspondant au nom du fichier CSV qui contient les données du changement de transport.

Afin de mettre en place la possibilité pour l'utilisateur de définir une borne qu'un trajet ne doit pas excéder (fonction demandée pour la version 1 mais que nous n'avions pas mis en place), nous avons ajouté 2 attributs au Voyageur, un 2e TypeCout définissant le type du critère ne devant être dépassé et un double représentant la valeur ne devant pas être dépassé, nous avons donc ajouté un constructeur plus détaillé pour la classe Voyageur et mis à jour l'ancien (pour permettre la création d'un voyageur qui n'aurait pas envie de définir une borne à ne pas dépasser). La fonction private List<Chemin> filtreSelonCritSec(List<Chemin> lChemin) défini dans Plateforme va parcourir les chemins d'une liste et renvoyer une nouvelle liste de chemin excluant les chemins de la liste en paramètre qui excédent la borne défini pas l'utilisateur.
Cette fonction est ensuite utilisée par la fonction « plusCourtChemin » pour prendre en compte les spécificités de l'utilisateur.

Afin de stocker les coûts supplémentaires en cas de changement de transports, Florian a ajouté à la class **Sommet** une Map de type : Map<ModaliteTransport[], Map<TypeCout, Double>> qui prend comme clé les 2 transports où s'effectue le changement (exemple : [Train, Avion]) et y associe comme valeur une Map contenant le Type du coût en clé, associé à la valeur correspondante.

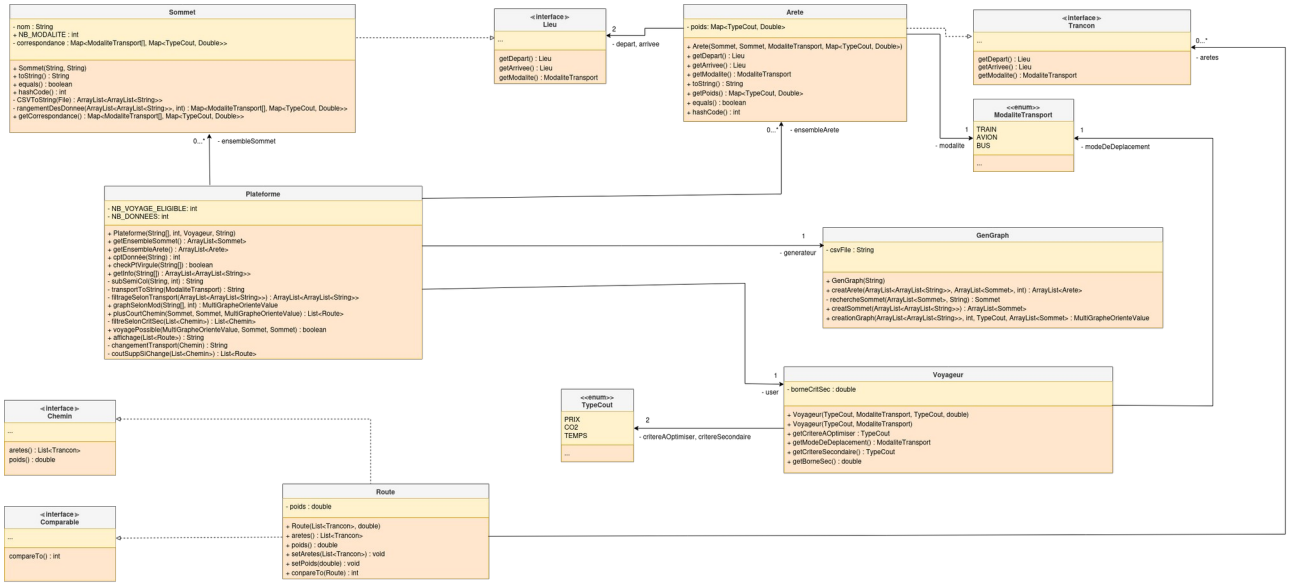
La fonction ArrayList<ArrayList<String>> CSVToString(File csv) de Alexandre, permet la lecture d'un fichier CSV en utilisant un Scanner, et stocke les données qu'il contient dans une ArrayList<ArrayList<String>> afin de pouvoir les exploiter par la suite.

La fonction rangementDesDonnee(ArrayList<ArrayList<String>> data, int nbModalite) va parcourir les données en paramètre et remplir la Map en attribut de Sommet avec celles-ci.

Pour permettre l'utilisation du CSV dans la création de Sommet, Florian a également modifié le constructeur de Sommet afin que, en plus du nom du Sommet, il prenne en paramètre une String

correspondant au nom du fichier CSV ayant les données des coûts de changement de transports, en respectant le format :
<ville>;<transport1>;<transport2>;<coût1>;<coût2>;<coût3>
(l'ordre des coûts doit être l'ordre inverse de de celui de l'énum TypeCout)

UML de la version 2 :



Version 3 :

Utilisation :

L'utilisation de cette 3e version est exactement la même que pour la version 2.

Le seul changement dans le code pour cette version 3 est l'historisation des voyages cherchés par l'utilisateur, pour cela Florian a modifié la fonction d'affichage de Plateforme afin qu'elle écrive dans un fichier historique.txt les trajets obtenus par l'exécution de l'application. Pour cela la fonction utilise un FileWriter qui écrit le résultat créé par l'affichage et l'écrit dans le fichier historique.txt, avant de finalement renvoyer le résultat comme avant pour en permettre l'affichage.

UML de la version 3 :