

# Rapport de Développement

**SAé 2.01-02**

**MULLIER Mathys - Giorgio UTZERI - Paul MACQUET (A1)**

BUT Informatique - IUT de Lille



## Lancement de l'application

Notre application se lance à partir de la classe **LaunchApplication** ne contenant uniquement une méthode **main** permettant l'exécution de l'application. La commande d'exécution, à exécuter depuis la racine du répertoire A1 (répertoire de ce projet) est la suivante :

```
# Commande de compilation :
javac -d bin -cp lib/sae_s2_2024.jar:src src/EAP/LaunchApplication.java

# Commande d'exécution :
java -cp bin:lib/sae_s2_2024.jar:lib/jgrapht-core-1.5.1.jar:lib/jheaps-0.14.jar EAP.LaunchApplication
```

## Exécution de l'application

La méthode **main** (exécutée lors du lancement de l'application) contient une variable de type **String[]** contenant donc un tableau de chaîne de caractères. Chaque chaîne correspond à une ligne de transport. Pour chaque chaîne, on retrouve en quelque sorte la structure d'un fichier CSV avec la structure suivante :

```
villeDépart;villeArrivée;modalitéTransport;prix(e);pollution(kgCO2e);durée(min)
```

De ce fait, chaque ligne est caractérisée par une ville de départ, une d'arrivée, du moyen de transport la caractérisant ainsi qu'aux valeurs relative à son coût financier, à la pollution qu'elle dégage et à la durée de transport. Ainsi, la méthode contient une variable :

```
String[] data = new String[] {
    "villeA;villeB;Train;60;1.7;80",
    "villeB;villeD;Train;22;2.4;40",
    "villeA;villeC;Train;42;1.4;50",
    "villeB;villeC;Train;14;1.4;60",
    "villeC;villeD;Avion;110;150;22",
    "villeC;villeD;Train;65;1.2;90",
    "villeC;villeD;Train;150;0.5;90",
    "villeC;villeA;Train;10;0.2;20",
    "villeA;villeD;Train;40;0.6;100"
};
```

Pour la suite, on crée une nouvelle instance de **Voyageur** caractérisée par son nom, son moyen de transport favori, la modalité que chercher à optimiser l'utilisateur ainsi qu'un tableau des valeurs maximales admises par l'utilisateur pour les modalités de son voyage.

Dans notre exemple, on instancie deux voyageurs différents avec une modalité favorite différente afin d'observer la différence des résultats sur un même voyage.

```
Voyageur premVoyageur = new Voyageur("priceTraveller", ModaliteTransport.TRAIN, TypeCout.PRIX);
Voyageur deuxVoyageur = new Voyageur("ecoloTraveller", ModaliteTransport.TRAIN, TypeCout.CO2);
```

Ici, le constructeur utilisé ne prend pas en paramètre les valeurs maximales pour les modalités, toutes les fonctionnalités

n'étant pas encore implémentées. Les valeurs par défaut sont alors utilisées, à savoir les valeurs maximales stockables dans une variable de type Double.

Une fois ces deux variables instanciées, nous pouvons maintenant passer la création du réseau par l'instanciation d'une variable Plateforme. C'est cette instanciation qui va permettre, par le biais des mécanismes objets, l'instanciation des principaux attributs de l'application :

```
Plateforme reseau = new Plateforme(data, premVoyageur);
```

Le réseau est donc maintenant prêt à l'utilisation pour le calcul des itinéraires optimisés, selon une modalité choisie, entre deux points. On peut alors récupérer ces plus courts chemins avec :

```
reseau.getKpcc("villeC", "villeD", 3)
```

La commande suivante lance la recherche des plus courts chemins pour le premier voyageur, entre la station *villeC* et la station *villeD* et affiche les 3 meilleurs résultats. Le résultat obtenu est de type List<Chemin>. Le reste du contenu de la méthode main n'est utile qu'à des fins d'affichage (en console pour cette première version). Pour notre première exemple, on obtient, pour le premier voyageur (favorisant le tarif), le résultat suivant :

Les chemins optimisés pour le premier voyageur :

```
Chemin(Arêtes: [[villeC -> villeA | TRAIN | {PRIX: 10.0€, TEMPS: 20.0, CO2: 0.2}], [villeA -> villeD  
| TRAIN | {PRIX: 40.0€, TEMPS: 100.0, CO2: 0.6}]], Poids: 50,000000)  
Chemin(Arêtes: [[villeC -> villeD | TRAIN | {PRIX: 65.0€, TEMPS: 90.0, CO2: 1.2}]], Poids: 65,000000)  
Chemin(Arêtes: [[villeC -> villeA | TRAIN | {PRIX: 10.0€, TEMPS: 20.0, CO2: 0.2}], [villeA -> villeB  
| TRAIN | {PRIX: 60.0€, TEMPS: 80.0, CO2: 1.7}], [villeB -> villeD | TRAIN | {PRIX: 22.0€, TEMPS: 40.0,  
CO2: 2.4}]], Poids: 92,000000)
```

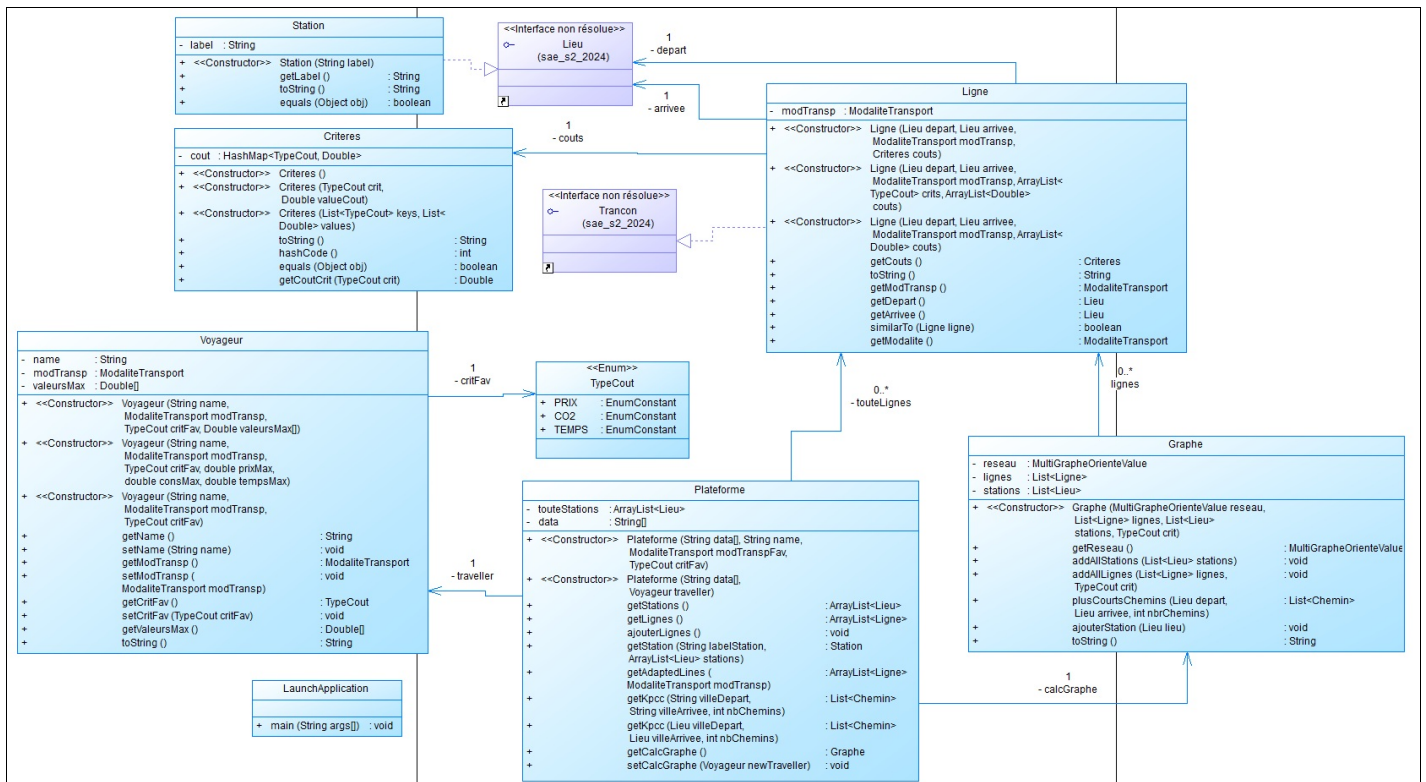
Pour le deuxième voyageur, favorisant lui un trajet à la consommation faible, on obtient :

Les chemins optimisés pour le deuxième voyageur :

```
Chemin(Arêtes: [[villeC -> villeD | TRAIN | {PRIX: 150.0€, TEMPS: 90.0, CO2: 0.5}]], Poids: 0,500000)  
Chemin(Arêtes: [[villeC -> villeA | TRAIN | {PRIX: 10.0€, TEMPS: 20.0, CO2: 0.2}], [villeA -> villeD  
| TRAIN | {PRIX: 40.0€, TEMPS: 100.0, CO2: 0.6}]], Poids: 0,800000)  
Chemin(Arêtes: [[villeC -> villeD | TRAIN | {PRIX: 65.0€, TEMPS: 90.0, CO2: 1.2}]], Poids: 1,200000)
```

## Diagramme UML et mécanismes objets

Vous trouverez ci-dessous une capture d'écran de notre diagramme UML (*également trouvable dans ./rapportDev/html/assets*) préalablement établi, d'un premier jet avec de se lancer développement de l'application puis enrichi au fur et à mesure de l'avancement du travail. À l'aide de cette représentation de l'application, il est possible de comprendre les différents mécanismes objets composants cette application.



## Analyse technique

Pour réussir l'implémentation des différentes fonctionnalités exigées pour cette SAé, nous avons dû implémenter les interfaces **Lieu** et **Trancon**, respectivement avec les classes **Station** et **Ligne**. Station a comme unique attribut **label**, de type **String**, correspondant à son nom.

Ligne implémente Trancon et a comme attribut une **HashMap** nommée **criteres**, qui associe à chaque type de coût un poids (de type **Double**, les types primitifs n'étant pas utilisable pour une HashMap), un lieu de depart, un lieu d'arrivee, et une modalité de transport.

La classe **Graphe** est caractérisée par une variable instance de l'objet **MultiGrapheOrienteeValue**, d'une liste de **Stations** et d'une liste de **Lignes**. Toutes les Stations et Lignes sont ajoutées dans le MultiGraphe via les méthodes **addAllStations** et **addAllLignes**.

La classe **Plateforme** stocke toutes les lignes et stations du réseau, celle-ci seront utilisées pour faire les calculs des plus courts chemins.

## Analyse des tests

Une fois l'implémentation terminée, il est important de vérifier le bon fonctionnement des méthodes implémentées. Nous avons donc réaliser une série de tests sur nos diifférentes classes :

**Criteres** : Sur Criteres, nous testons le constructeur, la méthode **toString** et la méthode **equals**.

**Ligne** : Sur Ligne, nous testons l'initialisation, la méthode constructeur et la méthode **similarTo**.

**Plateforme** : Sur Plateforme, nous testons le constructeur, la méthode **AjouterLigne** et la méthode **getKpcc**.

**Station** : Sur Station, nous testons le constructeur, la méthode **toString** et la méthode **equals**.

**Troncon** : Sur Troncon, nous testons le constructeur.

**Voyageur** : Sur Voyageur, nous testons le constructeur et les setters.

Le fait d'effectuer un bon nombre de tests sur chaque classe indépendemment les unes des autres, nous permet d'être sûrs de la bonne exécution de notre code par la suite.

# Rapport - Deuxième Version : Gestion des correspondances

Pour cette deuxième version les commandes d'exécutions restent les mêmes. Toutefois, quelques méthodes et signatures ont dû être modifiées pour s'adapter à la prise en compte des correspondances. Nous allons donc maintenant passer à la description de la nouvelle exécution de l'application avant d'expliquer dans une partie suivante quelles ont été les modifications et les nouvelles implémentations.

## Exécution de l'application

La méthode main (exécutée lors du lancement de l'application) contient des variables de type Voyageur. Pour chaque voyageur, on crée une plateforme qui se base sur le Voyageur et donc sur ses critères. On finit par afficher le réseau créé. Pour la suite, on crée une nouvelle instance de Voyageur caractérisée par son nom, son moyen de transport favori (ou non, dans le cas où il accepte de prendre des correspondances), le type de coût que cherchait à optimiser l'utilisateur, la ville de départ, la ville d'arrivée ainsi que le nombre de chemins qui lui seront proposés.

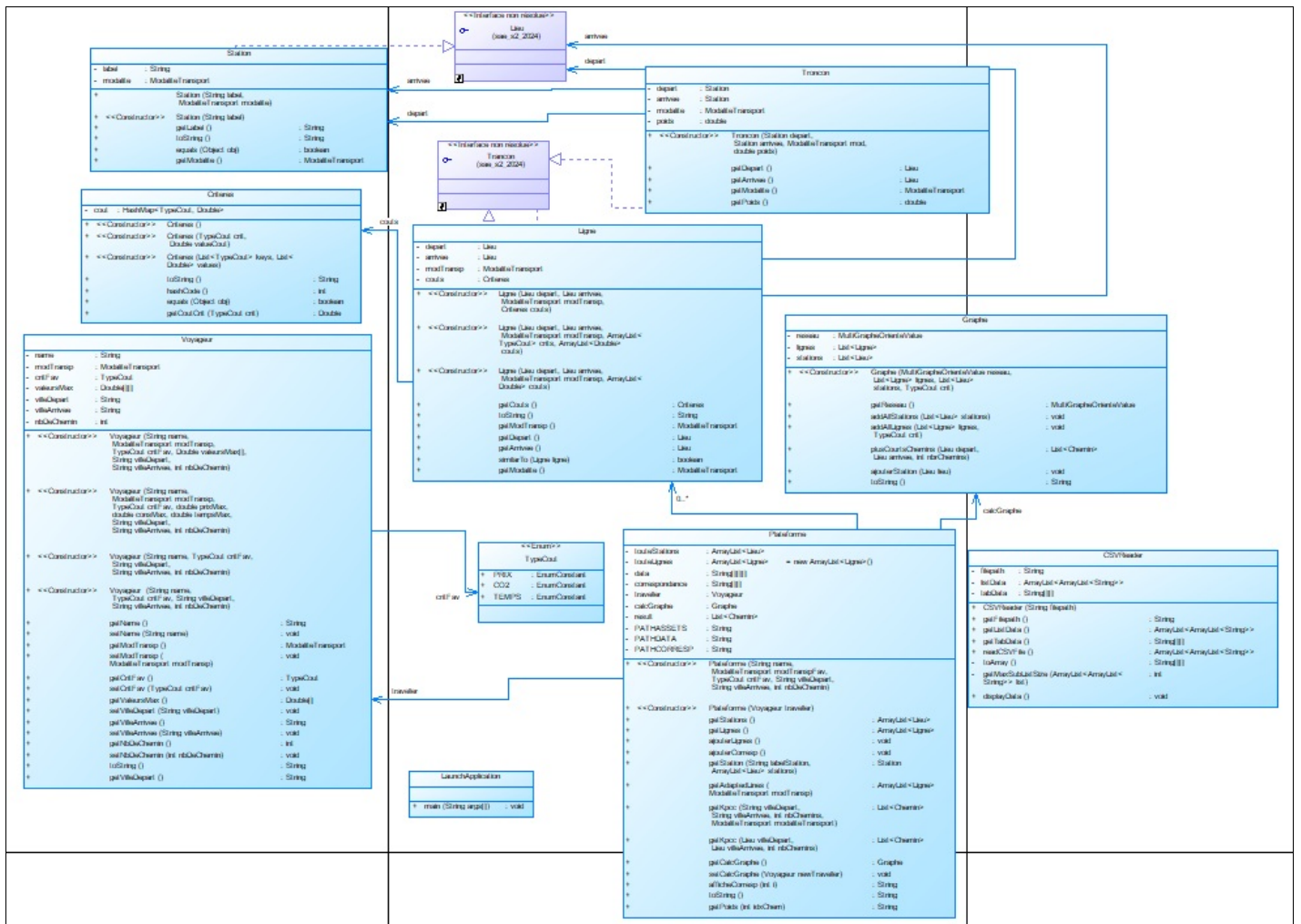
Dans notre exemple, on instancie deux voyageurs différents qui acceptent ou non les correspondances afin d'observer la différence des résultats.

```
Voyageur premVoyageur = new Voyageur("priceTraveller", TypeCout.C02, "Lille", "Toulouse", 2);
Voyageur deuxVoyageur = new Voyageur("ecoloTraveller", ModaliteTransport.TRAIN, TypeCout.C02,
                                     "Lille", "Marseille", 1);
```

Une fois ces deux variables instanciées, nous pouvons maintenant passer la création du réseau par l'instanciation d'une variable Plateforme (pour chaque voyageur). C'est cette instanciation qui va permettre, par le biais des mécanismes objets, l'instanciation des principaux attributs de l'application :

```
Plateforme reseau = new Plateforme(premVoyageur);
reseau = new Plateforme(deuxVoyageur);
```

Enfin, nous n'avons plus qu'à afficher les 2 réseaux, via un `System.out.println` qui va chercher le `toString` de la classe Plateforme. Diagramme UML et mécanismes objets Vous trouverez ci-dessous une capture d'écran de notre diagramme UML (également trouvable dans `./rapportDev/html/assets`) préalablement établi afin de résoudre les problèmes liés à la version 2, du développement de l'application puis enrichi au fur et à mesure de l'avancement du travail. À l'aide de cette représentation de l'application, il est possible de comprendre les différents mécanismes objets composants cette application.



## Analyse Technique

Pour réussir l'implémentation des différentes fonctionnalités exigées pour cette version de la SAÉ, nous avons dû créer CSVReader pour pouvoir lire le fichier csv, cette classe a comme attribut un String filepath qui correspond au chemin du fichier, une liste de liste de String qui correspond au fichier lu par la méthode readCSVFile et un tableau à 2 dimensions de String qui contient les données de listData convertit en tableau à 2 dimensions afin de s'adapter aux méthodes de la classe Plateforme. Nous avons dû rajouter à Station un attribut de modalité de Transport à Station, car dans la version 2, on nous demande de prendre en compte les correspondance. Notre approche est la suivante :

Nous créons pour chaque Station trois points dans le graphe qui contient chacune une modalité de transport (Par exemple : pour une Station Lille, nous créons un point "Lille Train", un autre "Lille Avion" et un autre "Lille Bus"). Nous avons aussi dû modifier la classe Plateforme pour qu'elle gère les correspondances pour la nouvelle version. Plateforme contient toujours tous les lieux et toutes les lignes du reseau, un tableau à deux dimensions correspondant aux données des Stations du reseau. Nous avons ajouter un tableau à deux dimensions de String contenant les correspondances (plus tôt récupéré grâce à CSVReader), un voyageur afin d'adapter le reseau à ses préférences, un graphe qui servira à calculer les plus courts chemins et enfin une liste de Chemin qui correspond aux résultats des plus courts chemins.

Dans le constructeur de Plateforme ne prenant qu'un voyageur en paramètre (celui utilisé dans LaunchApplication), on récupère et on stocke les données qui serviront à calculer les plus courts chemins. On récupère le voyageur et ses critères. On ajoute les lignes dans le Graphe via la méthode ajouterLignes, celle-ci fonctionne de la manière suivante :

On fait une boucle sur data, où chaque boucle boucle sur une ligne, la ligne courante est utilisée dans un forEach basé sur chaque ModaliteDeTransport de l'enum. On regarde si la Station de départ et la Station d'arrivée sont dans le Graphe, sinon on les ajoute. Ensuite, on récupère les coûts de la ligne dans une ArrayList de Double. Puis on teste si la Station d'arrivée et la Station de départ sont basées sur la même modalité, si oui on ajoute la ligne. Dans le constructeur, ensuite on ajoute les correspondances avec la méthode ajouterCorresp, dans celle-ci on utilise les correspondance lu grâce au CSVReader, on ajoute les lignes entre les stations ayant le même nom et une modalité différentes dans la liste des chemins, celle-ci sera ajoutée dans le graphe quand on instancie calcGraphe. Enfin, dans le constructeur on applique la méthode getKpcc sur le graphe et on le récupère dans result.

# Analyse des tests

Pour nous assurer du bon fonctionnement de cette nouvelle version, nous avons rajouté et modifié des tests des classes :  
**Graphe - Ligne - Plateforme - Station - Voyageur**



# Rapport - Troisième version : Coûts multiples

Pour cette deuxième version les commandes d'exécutions restent les mêmes. Toutefois, quelques méthodes et signatures ont du être modifiées pour s'adapter à la prise en compte des correspondances. Nous allons donc maintenant passer à la description de la nouvelle exécution de l'application avant d'expliquer dans une partie suivante quelles ont été les modifications et les nouvelles implémentations.

## Exécution de l'application

La méthode main (exécutée lors du lancement de l'application) contient des variable de type Voyageur. Pour chaque voyageur, on crée une plateforme qui se base sur le Voyageur et donc sur ses critères. On finit par afficher le réseau créé.

L'exécution de l'application ne change pas beaucoup de la précédente version. Il y a juste deux paramètres en plus pour instancier les voyageurs (ceux-ci seront expliqués dans la partie Analyse Technique). On les instancie de la manière suivante :

```
Voyageur premVoyageur = new Voyageur("priceTraveller", TypeCout.PRIX, 0.75, TypeCout.TEMPS, "J", "K", 10);
Voyageur deuxVoyageur = new Voyageur("ecoloTraveller", ModaliteTransport.TRAIN, TypeCout.CO2, 0.75,
                                     TypeCout.PRIX, "A", "K", 1);
```

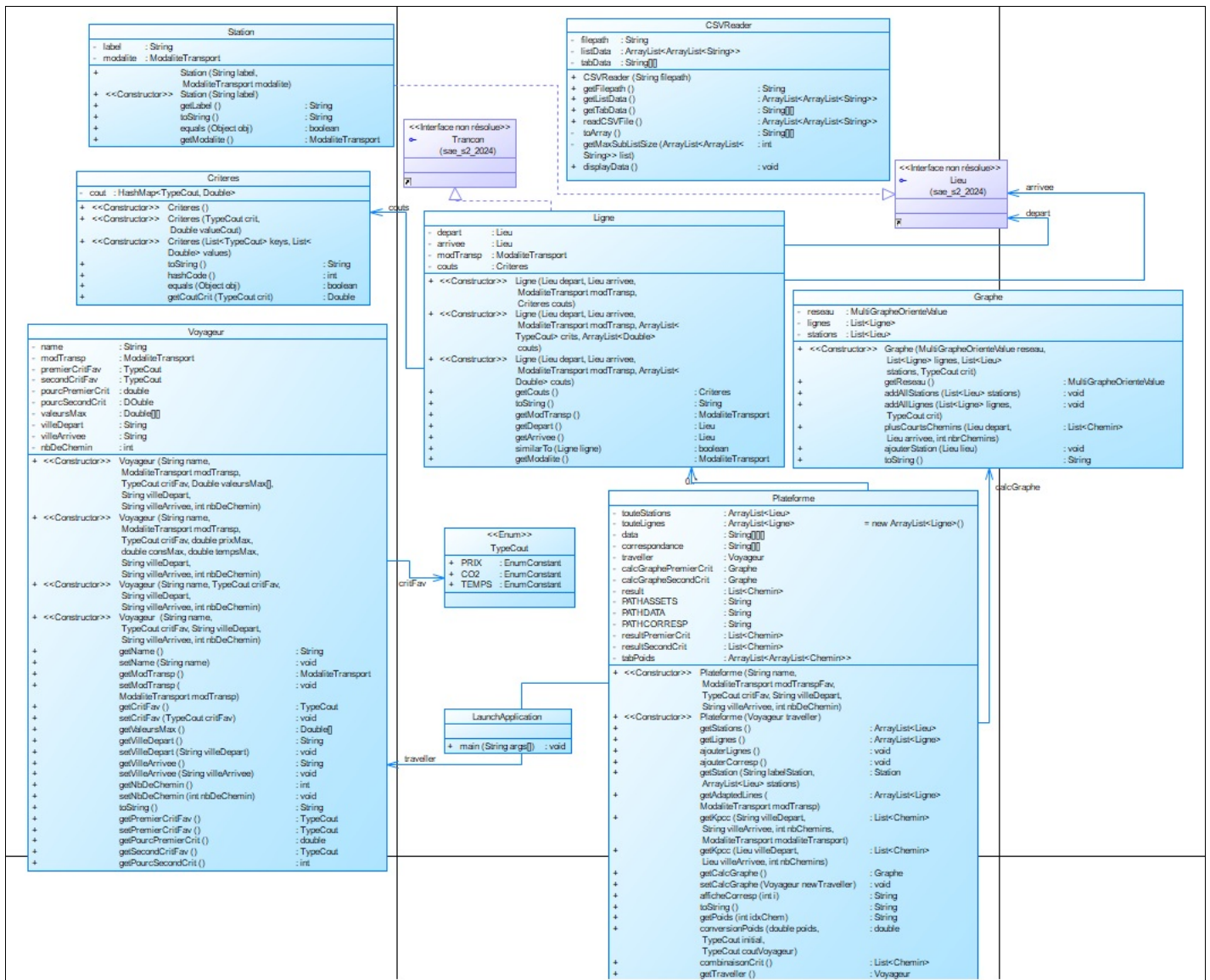
Puis on crée deux plateformes sur chacun :

```
Plateforme reseau = new Plateforme(premVoyageur) par exemple, puis on a juste à afficher la plateforme créée:
System.out.println(reseau);
```

## Diagramme UML et mécanismes objets

Vous trouverez ci-dessous une capture d'écran de notre diagramme UML(également trouvable dans ./rapportDev/html/assets) préalablement établi afin de résoudre les problèmes liés à la version 3, du développement de l'application puis enrichi au fur et à mesure de l'avancement du travail. À l'aide de cette représentation de l'application, il est possible de comprendre les différents mécanismes objets composants cette application.





## Analyse Technique

Pour réussir à implémenter la gestion des multi-critères, nous proposons à l'utilisateur de choisir deux typeCout, c'est-à-dire qu'il peut choisir par exemple un critère de temps et un critère de pollution, puis il choisit le pourcentage de préférence par rapport à un critère.

Si il souhaite un parcours préférant la pollution et le temps à parts égales, il peut choisir ces 2 critères et ajuster le pourcentage à 50%. Mais il peut aussi ajuster le pourcentage à 100% sur un seul critère.

Cette fonctionnalité est implémentée par l'ajout d'attributs dans la classe Voyageur, qui correspondent aux typeCout choisis par le voyageur et à leur pourcentages de préférence associé. Nous avons aussi dû implémenter la conversion de typeCout en un autre. Notre approche de la normalisation est la suivante :

Nous récupérons la totalité des couts de data dans l'attribut tabPoids dans la classe Plateforme, puis dans la méthode conversionPoids qui prend en paramètre un double qui correspond au poids et deux typeCout, on convertie le double poids qui a pour Typecout la variable initial le convertie et le renvoie dans l'autre typeCout. Pour cela, on divise la somme du typeCout voulu par la somme du typeCout initial et on multiplie par le poids. Nous avons choisi cette approche de normalisation car nous avons considéré que si nous prenions les intervalles de chaque typeCout, cela pouvait créer des écarts trop importants dans la normalisation, par exemple pour le cas du CO2, les valeurs minimales et maximales sont extrêmement éloignées par le fait que les modalités ne pollue pas du tout la même quantité. Nous avons donc décidé de fonctionner plus par moyenne.

## Analyse des tests

Pour cette troisième version, les classes testées sont les mêmes que pour la version précédente. Toutefois, des modifications ont été opérées afin d'adapter les tests aux modifications de code.

