Rapport POO

SAE

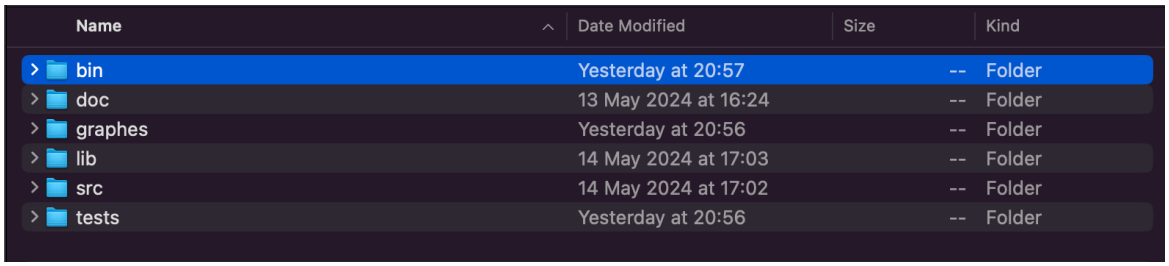
Dylan Lecocq
Amaury Vanhoutte
18/05/2024

Sommaire

I - Version 1

A. Structuration du projet

Le projet est structuré sous la forme d'une arborescence des fichiers, comme suit :



Les librairies nécessaires au bon fonctionnement de l'application se trouvent dans le repertoire lib/ ,les ressources de l'application dans le repertoire src/ tandis que les classes de tests se trouvent dans le repertoire tests. Le repertoire graphes contient le rendu pour la partie graphe du projet. Les repertoires doc/ et bin/ servent respectivement à stocker la documentation des différentes classes du projet, ainsi que les fichiers compilés de ces même classes.

B. Lancement de l'application

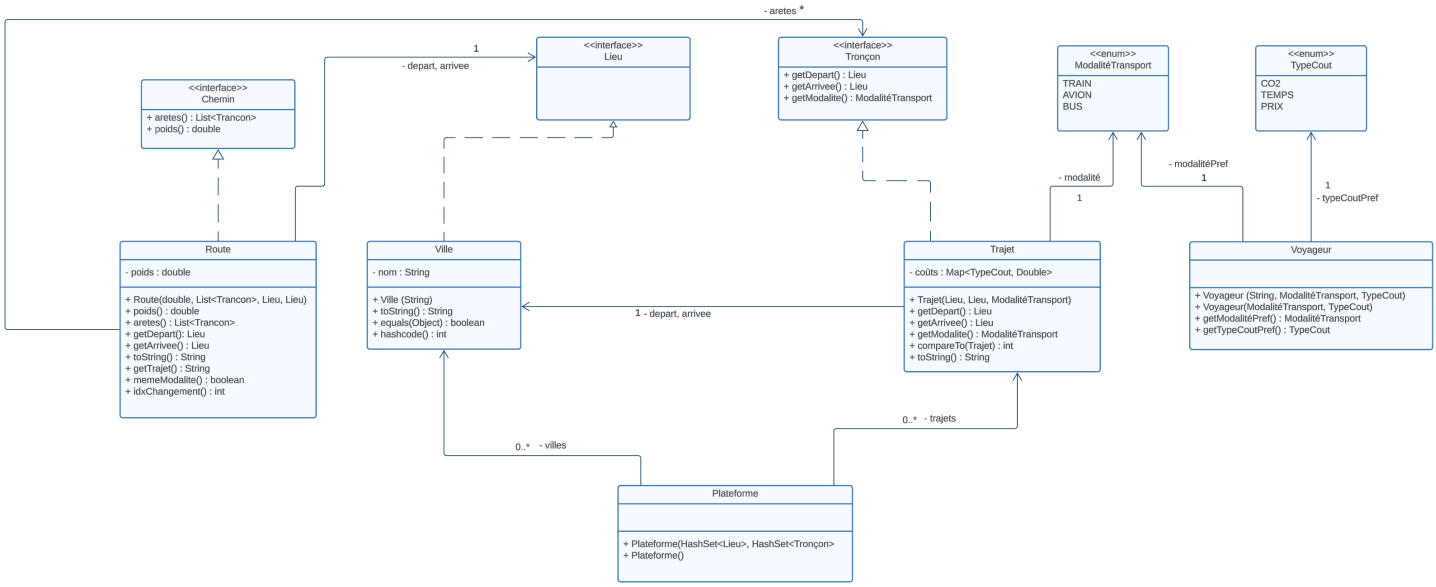
Comme indiqué précédemment, l'application se trouve dans le repertoire src/. Une fois placé à la racine du projet, pour pouvoir démarrer l'application il faut, dans un premier temps compiler l'ensemble des classes, ce qui peut être fait avec la commande suivante :

```
javac -cp ".:lib/*" -d bin src/*.java
```

Puis pour démarrer l'application on utilise : `java -cp "bin:lib/*" App`

C. Diagramme UML

Voici le diagramme UML représentant l'ensemble des classes du projets :



Dans notre implémentation, on retrouve deux principes de POO, l'implémentation, qui nous permets d'implémenter les diverses interfaces présentes dans les librairies fournies, qui définit des comportements pour les classes les implémentent.

Puis on retrouve de l'encapsulation, en effet plusieurs classes contiennent des attributs du type d'autre classes qui sont en visibilité `private`, ce qui les rends inaccessible par l'utilisateurs, excepté en utilisant les méthodes fournies.

D. Implémentation des fonctionnalités

Pour cette version 1, les fonctionnalités demandées sont les suivantes :

- Vérification de la validité des données fournies
- Récupérer les informations pour reconstituer le réseau ;
- Filtrer les données sur un moyen de transport spécifique ;
- Déterminer si, selon la modalité choisie, il existe un voyage possible entre les villes sélectionnées par l'utilisateur ;
- afficher les meilleurs voyages possibles, ordonnés selon le critère le plus important pour l'utilisateur ;
- exclure des alternatives qui excèdent les bornes définies par l'utilisateur.

La vérification des données se fait à l'aide d'une méthode verifyData(), qui vérifie si l'ensemble des champs attendus sont présents et s'ils sont au bon format grâce à une autre méthode isNumeric() qui vérifie si une chaîne de caractère passé en paramètre est numérique.

La récupération des données se fait après la vérification de celles-ci, grâce à une méthode retrieveData() qui prends en paramètre le tableau des données ainsi que la plateforme à laquelle on veut ajouter ces données.

Le filtre selon un moyen de transport se fait quant à lui au moment de l'ajout des données au graphe, grâce à une méthode ajouterVillesEtTrajets() qui prends en paramètre le graphe auquel on souhaite ajouter les données, la plateforme les contenant, le type de coût et le moyen de transport souhaités afin de n'obtenir un graphe n'ayant que les arêtes respectant les modalités définies par l'utilisateur.

L'affichage des meilleurs voyages est rendu possible grâce à une méthode dédiée afficherPCC() qui prends en paramètre une liste de chemins, le type de coût préféré de l'utilisateur.

E. Tests réalisés

Nous avons réalisé une classe de test destinée à vérifier le bon fonctionnement des méthodes principales de l'application, et ainsi améliorer la robustesse de l'application en général. Elle permet actuellement de tester 6 méthodes différentes. Chacune d'entre elle teste différents cas, d'exception ou non, et contrôle le résultat renvoyé par la fonction.

II - Version 2

A - Structuration du projet

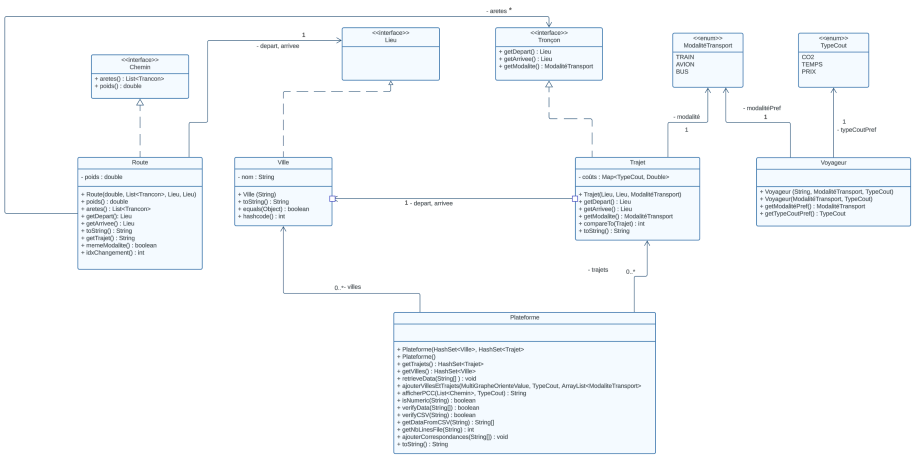
La structuration du projet est identique à la version 1.

B - Lancement de l'application

Les commandes pour compiler et exécuter cette version sont identiques à celles de la version 1. C'est à dire **`javac -cp ".:lib/*" -d bin src/*.java`** pour la compilation et **`java -cp "bin:lib/*" App`** pour l'exécution.

C - Diagramme UML

Pour la version 1, certaines méthodes se trouvaient dans la classe principale du projet et n'apparaissaient donc pas sur le diagramme UML, pour la version 2, ces méthodes ont toutes étaient déplacées dans la classe Plateforme, ce qui explique les méthodes ajoutées sur le diagramme UML.



D - Implémentation des fonctionnalités

Pour cette version 2, les fonctionnalités demandées sont les suivantes :

- Fonctionnalités de la version 1
- Import des données depuis des fichiers CSV
- Signaler les problèmes via un mécanisme d'exception
- Filtrer l'affichage d'une solution, pour n'afficher que les points d'intérêt

La plupart des méthodes s'occupant des fonctionnalités de la version 1 n'ayant pas été modifiées, celles-ci garde le même comportement dans cette version 2. Cependant, certaines ont dû être adaptée afin de prendre en charge le mécanisme d'exception et le nouvel affichage, c'est le cas des méthodes suivantes :

- afficherPCC(List<Chemin>, TypeCout) qui lance une exception de type NoTripException si aucun voyage respectant les critères n'a été trouvé. Et qui n'affiche plus que les points d'intérêts des Trajets.

- verifyData(String[]) qui lance une exception de type InvalidStructureException si les fichiers CSV ne respectent pas la structure requise.

- toString(), la méthode toString() de la classe Route a été modifiée afin de prendre en charge le nouvel affichage requis.

De nouvelles méthodes sont disponibles dans cette version 2 :

- verifyCSV(String) et getDataFromCSV(String) qui permettent respectivement de verifier l'existence et la structure du fichier passé en paramètre puis de récupérer les données de celui-ci si il est valide. Elles lancent une exception de type FileNotFoundException si le chemin passé en paramètre ne correspond à aucun fichier existant, et une exception de type InvalidStructureException si la structure de celui-ci n'est pas conforme.

- getNbLinesFile(String) qui permet de récupérer le nombre de lignes dans un fichier, elle lance une exception de type FileNotFoundException si le fichier n'existe pas.

E - Tests réalisés

En plus des tests réalisés pour la version 1, nous avons ajouté au sein de la classe de test, de tests pour chacune des 3 nouvelles méthodes ajoutées pour cette version, qui sont verifyCSV(String), getDataFromCSV(String) et getNbLinesFile(String). Chacun des nouveaux tests vérifie, comme celles de la version 1, des cas d'exceptions et des cas ordinaires afin de s'assurer que le programme continue de s'exécuter même s'il rencontre un problème

III - Version 3

A - Structuration du projet

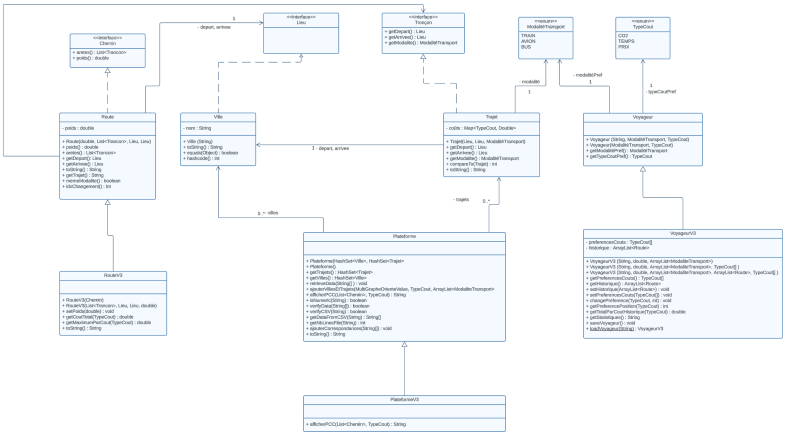
La structuration du projet est identique aux versions précédentes.

B- Lancement de l'application

La commande pour compiler est identique aux versions précédentes, il faut utiliser **`javac -cp ".:lib/*" -d bin src/*.java`**, pour exécuter le programme, la fichier à executer diffère des versions précédentes. IL faut utiliser la commande suivante : **`java -cp "bin:lib/*" AppV3`**.

C - Diagramme UML

Pour cette version 3, on retrouve principalement deux grands principes de POO, premièrement l'héritage, en effet afin de garantir une rétro-compatibilité avec les versions antérieures, les ajouts de cette version ont été ajoutées dans des classes héritant des classes précédentes. Et enfin la gestion d'exceptions, afin de garantir une exécution propre du programme, les fonctionnalités risquant de provoquer des erreurs lèvent des erreurs qui sont gérées par le programme lui-même, tout en expliquant à l'utilisateur le problème rencontré.



D - Implémentation des fonctionnalités

Pour la version finale, les fonctionnalités à ajouter étaient les suivantes :

- Fonctionnalités de la V2
- Calcul multi-critères
- Proposer une IHM ergonomique et efficace
- Enregistrer les voyages de l'utilisateur via sérialisation
- Exploiter l'historique

Afin d'enregistrer les voyages de l'utilisateur, nous avons décidé de rendre la classe VoyageurV3 sérialisable, ce qui permet de récupérer nos données enregistrées lors d'une précédente execution. Pour ce faire, la classe Voyageur dont hérite VoyageurV3 implémente l'interface Serializable. La classe VoyageurV3 contenant une ArrayList de Route, il a également fallu rendre la classe Route sérialisable afin que la classe RouteV3 le soit également, il en va de même pour la classe Ville, à cause des attributs de la classe Route étant de ce type. Pour enregistrer un voyageur, on fait appel à la méthode saveVoyageur() de la classe VoyageurV3, qui enregistre le voyageur courant dans un fichier save, puis pour le récupérer, on utilise la méthode loadVoyageur(String fichier) qui prends en paramètre le fichier dans lequel a été enregistré le voyageur puis le retourne.

Pour exploiter l'historique, nous avons ajouté la méthode getStatistiques() dans la classe VoyageurV3, qui retourne sous forme de chaine de caractères des indicateurs tels que le nombre de voyage, le coût total des voyages, le coût moyen, le temps total passé dans les transports ainsi que le temps moyen par voyage, les émissions totales de CO2 et les émissions moyennes par voyage.

Pour ce qui est de l'IHM, nous avons opté pour une interface simple, qui ne contient pas d'informations superflues tout en affichant les données

importantes pour l'utilisateur, ce qui permet de la rendre beaucoup plus simple d’utilisation et efficace.

E - Tests réalisés

En plus des tests réalisés pour les versions précédentes, nous avons réalisés de nouvelles fonctions de test pour les méthodes ajoutées au sein différentes classes.