Roget Benjamin
Fryson Adrien

Groupe: **B6**

Rapport POO

Dépot GitLab

Sommaire

Pour lancer le programme, il est nécessaire de se baser sur les exemples des fichiers settings.json et launch.json fournis dans le readme du projet Gitlab.

Lancer le programme

Il faut faudra créer le répertoire .vscode et y créer les fichiers settings.json et launch.json comme suit:

Avec interface graphique

launch.json: (si on veut lancer la version graphique)

```
{
      "version": "0.2.0",
      "configurations": [
            { "type": "java",
            "name": "Accueil",
            "request": "launch",
            "mainClass": "src.ihm.Accueil",
            "vmArgs" : "--module-path "JAVAFX/LIB"
      --add-modules=javafx.controls,javafx.fxml --add-exports
      javafx.base/com.sun.javafx.event=ALL-UNNAMED" }
             ]
}
```

Il faut modifier le chemin JAVAFX/LIB vers votre répertoire lib de votre installation javafx

settings.json

```
{
      "java.project.referencedLibraries": [ "JAVAFX/lib/*.jar",
      "lib/*.jar" ], "java.project.sourcePaths": [ "." ],
      "java.project.outputPath": "bin",
      "java.debug.settings.console": "internalConsole"
}
```

Il faut modifier le chemin JAVAFX/lib vers votre répertoire lib de votre installation javafx

Sans interface graphique

vous avez juste à utiliser les classes fournies dans B6.jar, la classe Voyageur et celle en héritant permettent d'expérimenter facilement l'entièreté du programme.

Il est préférable d'exécuter le programme à la racine du projet.

Utilisation des CSV

Pour importer des fichiers CSV différents, il est nécessaire de préciser leur chemin lors de l'instanciation de votre Voyageur. La seule exception est avec VoyageurIHM où vous devez fournir le fichier CSV sous forme d'une ArrayList. Vous pouvez le faire facilement avec la fonction getCSV de ToolsCorrespondance.

Réflexion sur les mécanismes Objet:

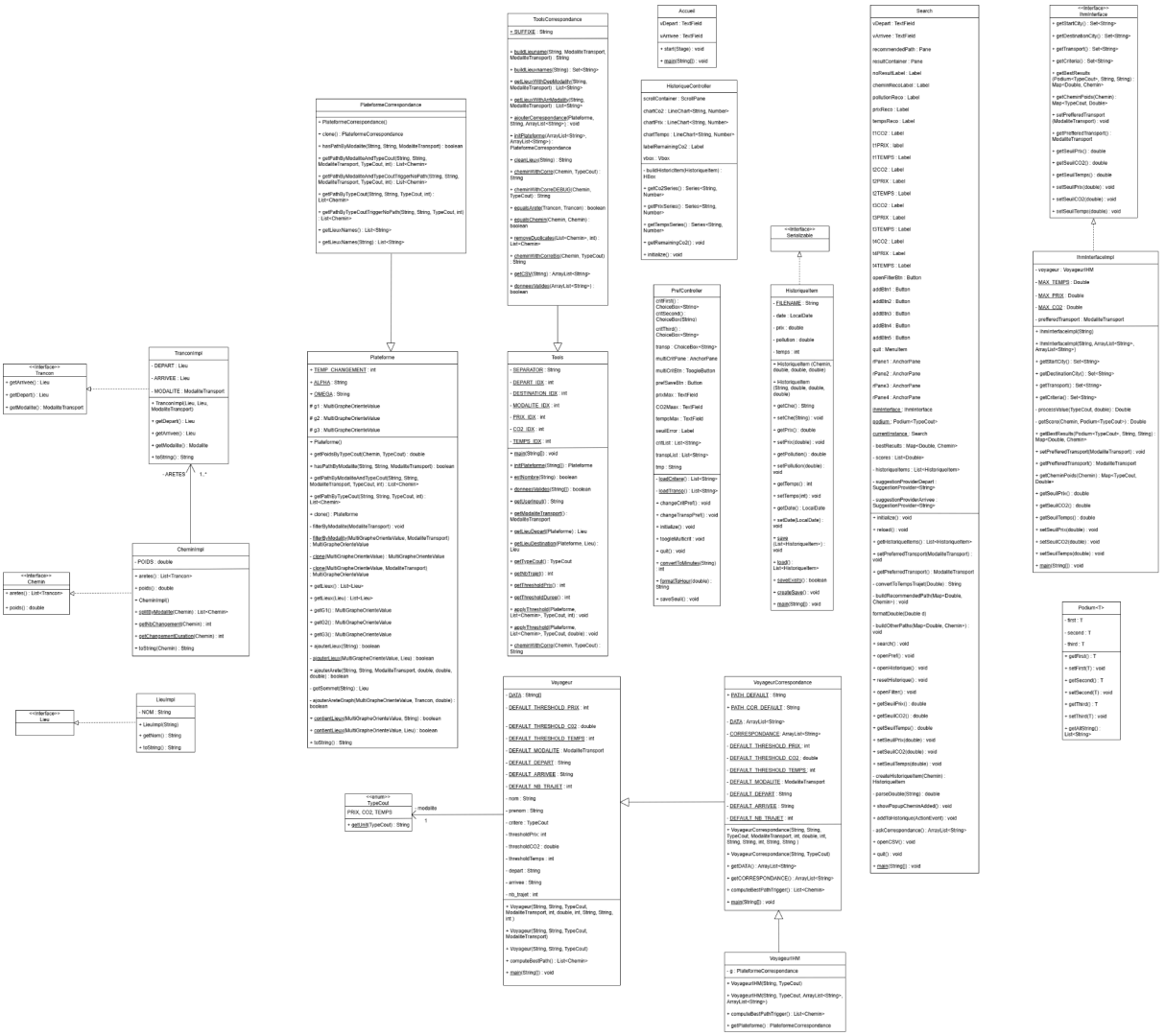

Diagramme UML V3

(l'UML est disponible sur le repo GitLab)

Principe Open-closed

Nous avons utilisé le principe open-closed pour développer ce projet. Cette manière de développer nous a permis d'assurer une base stable au code et de pouvoir l'étendre facilement.

Si nous devions tout de même toucher aux classes mères, la correction serait propagée facilement grâce au principe d'héritage.

Héritage

L’héritage nous permet d'étendre facilement le code pour de nouvelles fonctionnalités, par exemple avec les classes Plateforme et PlateformeCorrespondance. De cette manière, on peut réutiliser du code et améliorer la cohérence du code. Ainsi, les mécanismes de développement orienté objet nous permettent de garder une cohérence entre nos différentes classes et de factoriser notre code pour faciliter le débogage.

Polymorphisme

Concernant le polymorphisme, nous l’avons utilisé avec les interfaces, que ce soit Chemin, Tronçon ou IhmInterface. Le polymorphisme nous permet d’avoir un code flexible et de traiter les différents objets de manière uniforme.

Encapsulation

Le mécanisme d’encapsulation que nous utilisons, par exemple dans la classe Plateforme, nous permet de restreindre l’utilisation des attributs encapsulés par l’utilisateur. Ainsi, l’utilisateur ne peut interagir que de la manière que nous avons prédéfinie. Cela permet d’éviter des comportements inattendus. De plus, l’encapsulation nous permet de déléguer la complexité de certaines fonctionnalités à d’autres classes, par exemple l’algorithme KPCC.

Analyse technique et critique de l'implémentation

Pour implémenter les fonctionnalités, nous avons choisi de cacher derrière la classe plateforme trois graphes, puis nous simulons avec les méthodes publiques un unique graphe. Cela nous permet d’ajouter une couche d’abstraction tout en rendant le code plus clair.

Filtrage par modalité de transport

Pour permettre à l'utilisateur de filtrer un un moyen de transport, nous avons ajouté un constructeur à la classe Voyageur permettant de préciser une modalité de transport. La valeur par défaut est null, permettant d’utiliser alors n’importe quel moyen de transport. Ensuite, nous enlevons du graphe les arêtes ne correspondant pas au moyen de transport (filterByModality()). Comme cette fonction modifie directement le graphe, nous la restreignons à un usage private pour éviter que l’utilisateur ne détruise le graphe originel. Ainsi, quand on filtre sur un moyen de transport, la classe Plateforme va créer un nouveau graphe qu’elle va manipuler.

validation des données

Pour vérifier les données, que ce soit avec un fichier CSV ou non, nous nous assurons que chaque champ récupéré soit capable de nous fournir une valeur du bon type. Si c'est le cas, les données sont acceptées.

Gestion de de l'existence d'un chemin

Concernant la gestion de l'existence d'un chemin via un mécanisme d'exception, nous avons dû créer dans PlateformeCorrespondance plusieurs nouvelles fonctions. Chacune reprend le nom d'une fonction de Plateforme mais avec Trigger en suffixe. Nous avons dû procéder ainsi car les fonctions de Plateforme ne sont pas compatibles avec la levée d'exceptions. Par exemple : getPathByTypeCoutTriggerNoPath, getPathByModaliteAndTypeCoutTriggerNoPath.

Affichage selon les points d'intérêt

Pour l'affichage des chemins uniquement par leurs points d'intérêt, cela ne nous a pas posé de problème, car l'algorithme que l'on avait développé pour la première version se basait déjà sur les changements de modalité pour effectuer son affichage. Nous avons simplement dû enlever l'affichage des villes intermédiaires. Voir les méthodes cheminWithCorreBis et cheminWithCorre dans la classe ToolsCorrespondance.

Sérialisation binaire et historique

Pour la gestion de l'historique par sérialisation binaire, nous avons choisi de créer une classe dédiée nommée HistoriqueItem. Un HistoriqueItem contient plusieurs informations sur le chemin enregistré dedans, comme la date, la ville de départ, la ville d'arrivée et d'autres attributs. Un HistoriqueItem a pour but d'être stocké dans une List. La méthode statique Save enregistre une List à l'emplacement défini par la constante FILENAME. La méthode statique Load lit le fichier à ce même emplacement et retourne la liste d'HistoriqueItem. D'autres méthodes ont été implémentées comme saveExist ou createSave.

Multi-critère

Pour la prise de décision multi-critères, nous avons intégré un système de notation dans IhmInterfaceImpl afin de déterminer le trajet recommandé. Cette classe encapsule une instance de VoyageurIHM (qui hérite de VoyageurCorrespondance) et ajoute une couche supplémentaire pour la notation. La méthode utilisée pour calculer cette notation est expliquée en détail dans le rapport de graphes. Une fois la notation déterminée, il suffit de trier les trajets en fonction de cette notation pour obtenir un classement.

IHM

Pour l'IHM, nous avons décidé de créer une interface avec laquelle l'interface graphique interagira. Cela nous permet de rendre l'interface indépendante du code et vice-versa. Cette interface définit des méthodes de base qui ont pour but de faciliter l'interaction avec le backend.

Analyses des tests

Presque toutes les fonction et methodes ont un test unitaire, les méthodes plus complexes bénéficient d'un test avec du contexte plus développé pour correspondre à l'utilisation réelle.

Les tests ne garantissent pas qu'il n'y a pas de bug, néanmoins il devrait suffire à assurer le fonctionnement du code dans une situation simple.