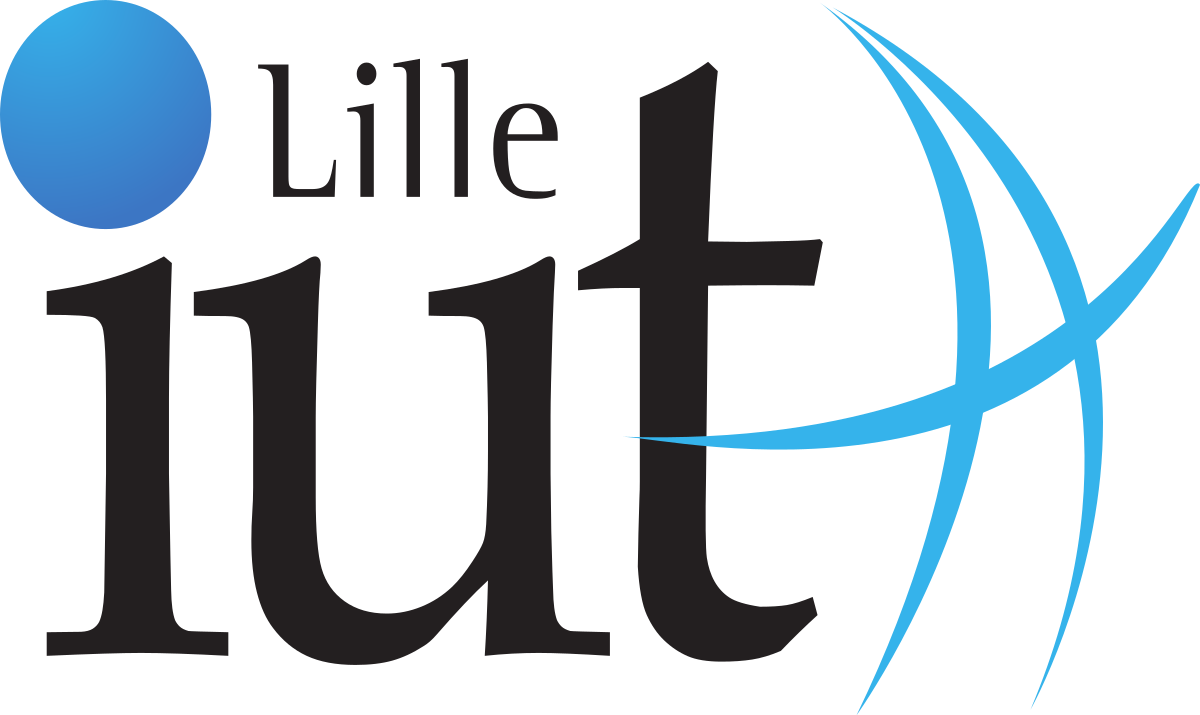
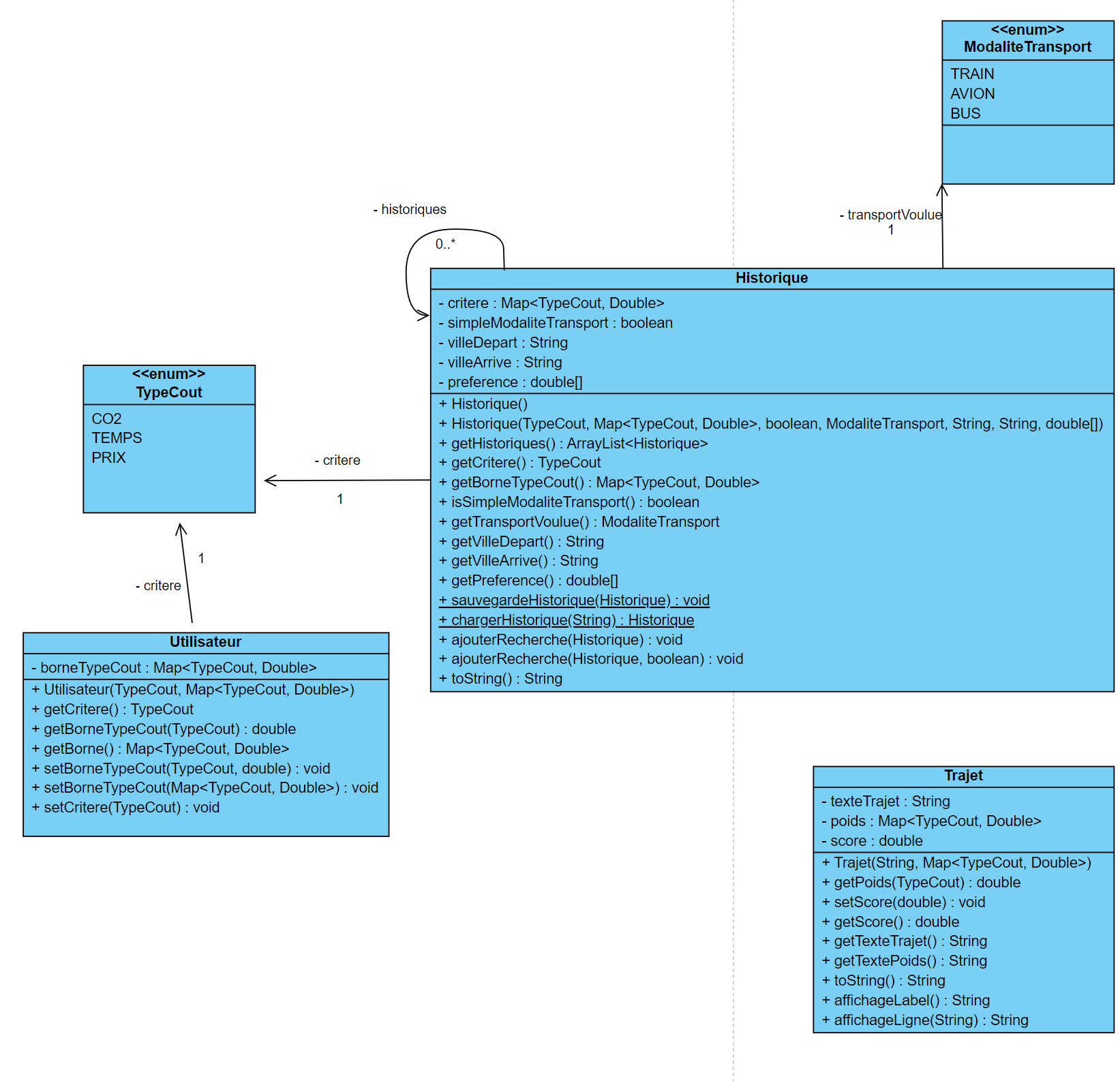
****

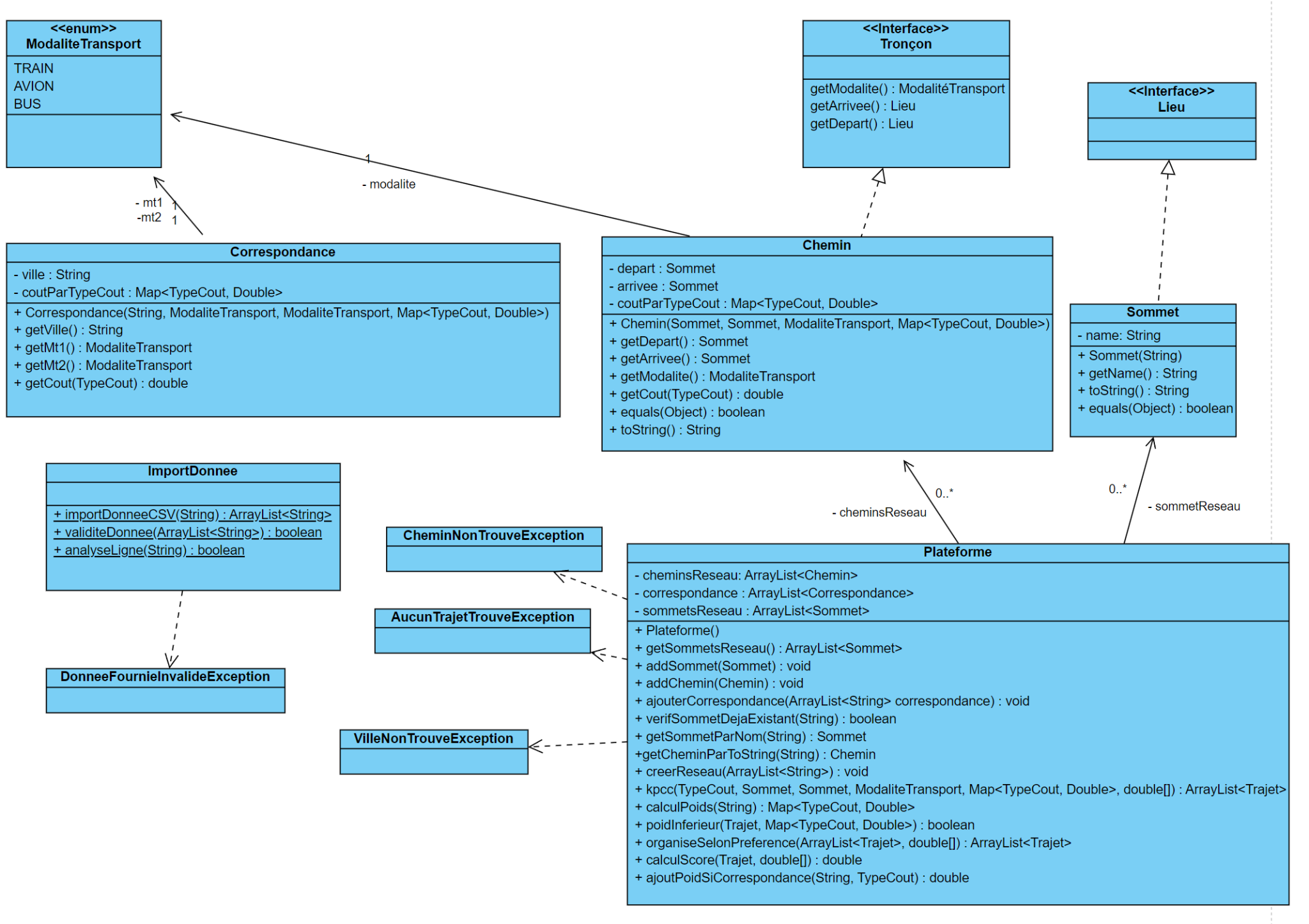
**Rapport SAE 2.01 & 2.02 Version 3**

**RANDOUX Martin, DESPREZ Mathéo, LENCEL Antoine**

**Groupe : A2**

**Notre diagramme UML :**

****

****

**Chemin** :

* La classe “Chemin” implémente l'interface “Trancon”.
* Elle représente un trajet entre deux sommets (“depart” et “arrive”) avec une modalité de transport (“modalite”) et un ensemble de coûts (“coutParTypeCout”).
* La méthode “equals(Object obj)” est surchargée pour comparer deux objets Chemin en vérifiant l'égalité de leurs attributs (“depart”,”arrive”, “modalite” et “coutParTypeCout”).
* La méthode “getCout(TypeCout typeCout)” retourne le coût associé à un type de coût spécifique.
* Les méthodes “getDepart()”, “getArrivee()” et “getModalite()” sont simplement les accesseurs.
* La méthode “toString()” retourne une représentation décrivant la modalité de transport et les sommets de départ et d'arrivée du chemin.

**Correspondance** :

* La classe “Correspondance” représente une relation entre deux modalités de transport (“mt1” et “mt2”) dans une (“ville”) spécifique .
* Elle inclut différents types de coûts (“TypeCout”) associés à cette correspondance (“coutParTypeCout”).
* Les méthodes “getVille()”, “getMt1()” et “getMt2()” sont des accesseurs pour récupérer respectivement le nom de la ville, la première et la deuxième modalité de transport.
* La méthode “getCout(TypeCout typeCout)” retourne le coût associé à un type de coût spécifique dans cette correspondance.

**Historique** :

* La classe “Historique” représente une collection d'objets “Historique” contenant des informations sur des recherches précédentes.
* Elle utilise une liste (ArrayList) pour stocker les divers objets.
* Les attributs incluent :
  + “critere”: Un type de coût (“TypeCout”) utilisé comme critère dans la recherche.
  + “borneTypeCout”: Une carte (“Map”) associant des types de coûts (“TypeCout”) à leurs valeurs.
  + “simpleModaliteTransport”: Un booléen indiquant si la recherche utilise une seule modalité de transport.
  + “transportVoulue”: La modalité de transport spécifique recherchée.
  + “villeDepart” et “villeArrive”: Les villes de départ et d'arrivée.
  + “preference”: Un tableau de doubles représentant les préférences.
* Le constructeur initialise l'attribut “historiques” comme une nouvelle liste.
* Les méthodes d'accès (get...) permettent de récupérer les valeurs des attributs.
* La méthode “sauvegardeHistorique” enregistre les objets “Historique” dans un fichier CSV, où chaque ligne représente un objet “Historique”.
* La méthode “chargerHistorique” charge les objets “Historique” à partir d'un fichier CSV.
* Les méthodes “ajouterRecherche” ajoutent un nouvel objet “Historique” à la liste, avec des options pour ajouter en tête ou en queue.
* La méthode “toString” fournit une représentation sous forme de chaîne de caractères de l'objet “Historique”, affichant la ville de départ et la ville d'arrivée.

**ImportDonnee** :

* La classe “ImportDonnee” est utilisée pour importer des données à partir d'un fichier CSV.
* La méthode “importDonneCSV” prend en paramètre le chemin d'accès d'un fichier CSV et retourne une liste (ArrayList) de String.
* À l'intérieur de cette méthode :
  + Un objet “BufferedReader” est utilisé pour lire le fichier ligne par ligne.
  + Chaque ligne est ajoutée à l'ArrayList donnee.
  + Après avoir lu toutes les lignes, la méthode validiteDonnee est appelée pour vérifier la validité des données importées.
  + En cas d'exception IOException, l'erreur est affichée avec e.printStackTrace().
* La méthode validiteDonnee parcourt chaque ligne dans la liste data et appelle analyseLigne pour vérifier si chaque ligne contient exactement cinq points-virgules (;).
* La méthode analyseLigne prend une chaîne de caractères ligne en paramètre et compte le nombre de points-virgules.
* Si le nombre de points-virgules dans une ligne n'est pas égal à cinq, une exception DonneeFournieInvalideException est levée avec un message d'erreur approprié.
* La méthode analyseLigne retourne true si le nombre de points-virgules est correct, sinon elle lève une exception.

**Main** :

* La classe Main contient la méthode main, qui est la méthode d'entrée principale du programme.
* Dans cette méthode main, plusieurs actions sont effectuées :
  + **Chargement de l'historique** : La méthode statique chargerHistorique de la classe Historique est utilisée pour charger les données historiques à partir du fichier CSV res/historique.csv.
  + **Importation des données** : Les données de chemins (data.csv) et de correspondances (donneeCorrespondance.csv) sont importées à l'aide de la méthode importDonneCSV de la classe ImportDonnee, qui retourne des ArrayList de chaînes de caractères contenant les lignes de chaque fichier CSV.
  + **Initialisation des bornes de coût** : Un HashMap nommé borneTypeCout est créé pour stocker les bornes de coût, avec des valeurs arbitraires (200.0 pour TEMPS et PRIX).
  + **Création de l'utilisateur** : Un objet Utilisateur est instancié avec un critère de coût (TypeCout.PRIX) et les bornes de coût spécifiées.
  + **Création de la plateforme** : Un objet Plateforme est instancié.
  + **Création des trajets** :
    - La méthode creerReseau de Plateforme est appelée pour créer le réseau à partir des données de chemin importées.
    - La méthode ajouterCorrespondance de Plateforme est appelée pour ajouter les correspondances à partir des données importées.
  + **Sélection du mode de transport** : Selon la variable simpleModaliteTransport, si elle est vraie, transportVoulue est défini sur ModaliteTransport.TRAIN.
  + **Recherche de trajets** :
    - Les noms des sommets s1 (L) et s2 (M) sont récupérés à partir de la plateforme à l'aide de getSommetParNom.
    - La méthode kpcc de Plateforme est appelée pour rechercher les trajets possibles entre s1 et s2 en fonction des critères de l'utilisateur, du mode de transport voulu et des préférences.
    - Les résultats sont stockés dans une liste d'objets Trajet et sont affichés à l'écran à l'aide de leur méthode toString.
  + **Gestion des exceptions** : Les exceptions VilleNonTrouveException et AucunTrajetTouveException peuvent être levées lors de la recherche de trajets et sont capturées pour afficher un message d'erreur approprié.
  + **Enregistrement de l'historique** : Un nouvel objet Historique est créé pour enregistrer les détails de la recherche effectuée par l'utilisateur, puis sauvegardeHistorique est appelé pour sauvegarder cet historique dans le fichier CSV res/historique.csv.

**Plateforme:**

La classe Plateforme joue un rôle central dans la gestion du réseau de transport et des fonctionnalités associées. Voici une description succincte des méthodes et fonctionnalités implémentées dans cette classe :

1. **Attributs** :
   * cheminsReseau, sommetsReseau, correspondance : Ce sont des listes qui stockent respectivement les chemins (instances de Chemin), les sommets (instances de Sommet) et les correspondances (instances de Correspondance) du réseau de transport.
2. **Constructeur** :
   * Plateforme() : Initialise les listes cheminsReseau, sommetsReseau, et correspondance.
3. **Méthodes publiques** :
   * getSommetsReseau(), addSommet(Sommet s), addChemin(Chemin c), ajouterCorrespondance(ArrayList<String> correspondances) : Ces méthodes permettent respectivement de récupérer la liste des sommets du réseau, d'ajouter un sommet au réseau, d'ajouter un chemin au réseau et d'ajouter des correspondances au réseau à partir des données fournies sous forme de chaînes de caractères.
   * verifSommetDejaExistant(String name) : Vérifie si un sommet avec le nom spécifié existe déjà dans sommetsReseau.
   * getSommetParNom(String name) : Retourne le sommet correspondant au nom spécifié. Lève une VilleNonTrouveException si le sommet n'est pas trouvé.
   * getCheminParToString(String text) : Retourne le chemin correspondant à la représentation textuelle spécifiée. Lève une CheminNonTrouveException si le chemin n'est pas trouvé.
   * creerReseau(ArrayList<String> data) : Crée le réseau de transport à partir des données fournies sous forme de chaînes de caractères. Chaque chaîne est divisée en éléments qui représentent les sommets, la modalité de transport et les coûts associés, puis des objets Sommet, Chemin sont créés et ajoutés aux listes correspondantes.
   * kpcc(TypeCout typeCout, Sommet depart, Sommet arrive, ModaliteTransport modaliteTransportUnique, Map<TypeCout, Double> limitePoid, double[] preference) : Utilise l'algorithme KPCC pour rechercher et retourner une liste de trajets (Trajet) entre deux sommets donnés, en respectant les critères de coût, la modalité de transport souhaitée, les limites de poids et les préférences de l'utilisateur. Cette méthode gère également les exceptions AucunTrajetTouveException.
   * calculPoids(String ligne), poidInferieur(Trajet trajet, Map<TypeCout, Double> limitePoid), organiseSelonPreference(ArrayList<Trajet> trajets, double[] preference), calculScore(Trajet trajet, double[] preference), ajoutPoidSiCorrespondance(String ligne, TypeCout typeCout) : Ces méthodes auxiliaires sont utilisées pour calculer le poids d'un trajet, vérifier si le poids d'un trajet est inférieur à une limite donnée, organiser les trajets selon les préférences de l'utilisateur, calculer le score d'un trajet en fonction des préférences, et ajouter un poids si une correspondance est présente entre les trajets.

**Sommet:**

1. **Attributs** :
   * name : C'est le nom du sommet, utilisé comme identifiant unique.
2. **Constructeur** :
   * Sommet(String name) : Initialise un sommet avec un nom spécifié.
3. **Méthodes publiques** :
   * equals(Object obj) : Redéfinit la méthode equals pour comparer deux sommets. Deux sommets sont considérés égaux si leurs noms sont identiques.
   * getName() : Retourne le nom du sommet.
   * toString() : Redéfinit la méthode toString pour retourner une représentation textuelle du sommet, qui est simplement son nom.

La classe Sommet est conçue pour encapsuler les informations relatives à un sommet dans le contexte d'un réseau de transport, en fournissant des méthodes pour la comparaison (equals), l'accès au nom (getName) et la représentation sous forme de chaîne (toString). Cela permet une gestion efficace des sommets dans les opérations telles que la création de chemins, la recherche de sommets par nom, et d'autres opérations liées à l'infrastructure de transport.

**Trajet:**

La classe Trajet représente un trajet dans un système de transport, caractérisé par plusieurs aspects comme le texte du trajet, les poids associés à différents critères (temps, CO2, prix) et un score éventuel. Voici une explication détaillée de cette classe :

### **Attributs :**

* texteTrajet : Une représentation textuelle du trajet.
* poids : Une carte associant chaque type de coût (TypeCout) à sa valeur respective (temps, CO2, prix).
* score : Le score attribué au trajet.

### **Constructeur :**

* Trajet(String texte, Map<TypeCout, Double> poids) : Initialise un trajet avec le texte du trajet et les poids spécifiés.

### **Méthodes publiques :**

* getPoids(TypeCout typeCout) : Retourne le poids associé au type de coût spécifié.
* setScore(double score) : Définit le score du trajet.
* getScore() : Retourne le score du trajet.
* getTexteTrajet() : Retourne le texte du trajet.
* getTextePoids() : Retourne une représentation textuelle des poids du trajet formatée avec un seul chiffre après la virgule.
* toString() : Redéfinition de la méthode toString() pour retourner une chaîne représentant le trajet, incluant le texte du trajet et les poids formatés.
* affichageLabel() : Retourne une représentation textuelle du trajet formatée avec des étiquettes pour chaque type de coût.
* affichageLigne(String ligne) : Méthode utilitaire pour formater le texte du trajet en s'assurant que les différents segments de transport soient correctement affichés.

### **Explication de affichageLigne(String ligne) :**

Cette méthode prend en paramètre une chaîne ligne représentant un trajet sous forme textuelle. Elle découpe cette chaîne pour extraire chaque segment de transport et les restructure si nécessaire pour s'assurer que le texte du trajet soit correctement formaté et lisible. Elle utilise les indices des espaces pour déterminer où un segment de transport commence et se termine, tout en conservant une gestion des correspondances entre différents moyens de transport.

### **Utilisation :**

La classe Trajet est utilisée pour encapsuler les informations spécifiques à un trajet dans un système de transport, fournissant des méthodes pour accéder aux détails du trajet, les manipuler et les afficher de manière lisible et formatée. Elle est conçue pour être intégrée dans un système plus large de gestion de trajets, de recherche de chemins optimaux et de présentation des résultats au niveau de l'utilisateur.

La classe Utilisateur représente un utilisateur du système avec un critère spécifique et des bornes pour différents types de coûts. Voici une explication détaillée de cette classe :

### **Description des attributs :**

1. **critere** : C'est le critère principal choisi par l'utilisateur pour optimiser ses choix de trajets. Ce critère est de type TypeCout, qui est une énumération définissant les différents types de coûts (CO2, prix, temps).
2. **borneTypeCout** : C'est une Map<TypeCout, Double> qui associe chaque type de coût à sa borne maximale acceptable selon l'utilisateur. Par exemple, pour le type de coût "temps", la borne pourrait indiquer la durée maximale acceptable pour un trajet. Pour le type de coût "prix", la borne pourrait indiquer le prix maximum acceptable.

### **Méthodes de la classe :**

* **Constructeur Utilisateur(TypeCout critere, Map<TypeCout, Double> borneTypeCout)** : Initialise un nouvel utilisateur avec le critère et les bornes spécifiés.
* **getCritere()** : Retourne le critère principal choisi par l'utilisateur.
* **getBorneTypeCout(TypeCout typeCout)** : Retourne la borne maximale acceptable pour le type de coût spécifié.
* **getBorne()** : Retourne la map complète des bornes pour tous les types de coûts.
* **setBorneTypeCout(TypeCout typeCout, double borne)** : Met à jour la borne maximale acceptable pour le type de coût spécifié.
* **setBorneTypeCout(Map<TypeCout, Double> borneTypeCout)** : Remplace toutes les bornes par celles spécifiées dans la nouvelle map.
* **setCritere(TypeCout critere)** : Met à jour le critère principal choisi par l'utilisateur.

La classe Utilisateur est utilisée pour encapsuler les préférences et les paramètres spécifiques à chaque utilisateur, ce qui permet de personnaliser les recommandations de trajets en fonction des critères et des contraintes qu'ils spécifient.

Les méthodes fournies permettent de récupérer et de modifier les critères et les bornes de manière contrôlée, assurant ainsi une gestion flexible et précise des préférences de l'utilisateur dans le système de recommandation de trajets.