Document de synthèse du Projet Informatique : « Application de calcul d'itinéraires à partir des données de la RATP »

Aelaïg Cournez et Fanny Vignolles

17 mai 2018

Table des matières

1 Bilan du projet

1.1 Organisation de travail

Pour ce projet, nous avions réalisé un diagramme de GANTT prévisionnel dont les délais ont été modifiés au fur et à mesure de notre avancée. (cf. *Annexes 3.1*)

Nous nous sommes aussi réparties les tâches, de ce fait à chaque séance (encadrées ou en autonomie) nous effectuions des bilans provisoires des objectifs remplis et nous fixions les objectifs suivants pour chacune d'entre nous (cf. *Carnet-de-Bord_ ProjetJAVA.pdf*). La répartition grossière des tâches peut se résumer comme ceci : (il est à noter que nous avons eu un regard et une action sur les tâches de l'autre)

Aelaïg	Fanny
Implémentation du réseau en Java	Génération de la base de données
Création de la Table Horaire	Rédaction Rapport d'Analyse
Implémentation CSA [1]	Implémentation Dijkstra
Implémentation CSA modifié	Gestion requête utilisateur
Gestion requête gestionnaire	Rédaction Document de Synthèse

TABLE 1 – Répartition des tâches

La documentation utilisateur a été rédigée au fur et à mesure de notre avancée lors de l'implémentation à chaque création d'une nouvelle méthode ou classe.

1.2 Bilan des tâches accomplies et difficultés rencontrées

1.2.1 Base de données et réseau

Création de la base de données :

Le premier objectif par rapport à la base de données (BDD) était de comprendre la modélisation du système et des fichiers de la RATP (cf. *Rapport d'Analyse*). Après cette étape, nous avons pu créer la base de données avec les tables et colonnes adaptées.

Importation des données :

Le deuxième objectif était d'y insérer les différentes données présentes dans les fichiers RATP. Des traitements supplémentaires ont été effectués à l'aide de requêtes SQL, tels que la création d'une colonne ligne dans la table *station* permettant de faire le lien entre la station et sa ligne de rattachement, le tri des correspondances et des arrêts qui concernaient uniquement le réseau ferré auquel nous nous étions cloisonnées.

Nous avons rencontré un problème lors de l'importation des données de la ligne de métro 6 à cause d'une mauvaise gestion des entiers *id_ routes* qui, étant trop grands, se retrouvaient en écriture scientifique lors de la conversion au format *.csv* des fichiers *.txt* de la RATP (ex : 10025336351193702 devenait $1,0025336351193702 \cdot 10^{16}$), nous empêchant par la suite d'importer ces données. Nous avons donc décidé de ne pas prendre en compte la ligne 6 avec l'idée éventuelle de gérer ce problème une fois l'application optimisée.

1. *Connection Scan Algorithm*

Modélisation du réseau en Java :

Pour finir, nous avons modélisé notre réseau sur Java par le biais de différentes classes qui seront détaillées dans la partie *Développement et exécution du programme*.

1.2.2 Algorithmes

Table Horaire :

La table horaire est le socle du traitement des données sans lequel nos algorithmes de calcul d'itinéraires ne pourraient fonctionner. La générer était donc notre objectif prioritaire dans l'implémentation des algorithmes. Il a fallu créer d'abord une table horaire recensant uniquement les courses et une autre recensant uniquement les correspondances, puis fusionner les deux en triant le résultat ainsi obtenu par horaire de départ et d'arrivée.

L'inconvénient majeur de cette table horaire est qu'elle contient énormément de données et est donc trop lourde pour être calculée à chaque demande d'itinéraire. Nous avons dans un premier temps souhaité l'enregistrer dans notre base de données de manière à simplement la consulter aux besoins avec la possibilité de la calculer à nouveau en cas de changement des horaires du réseau. Cependant, cet enregistrement prenait du temps (nos tests duraient plus de cinq heures) donc nous avons dû nous adapter et avons décidé de ne pas calculer la table horaire en entier mais seulement la partie qui nous intéressait lors d'une demande d'itinéraire (i.e. celle de la tranche horaire entre l'heure de départ et un intervalle de temps défini).

Algorithmes de calcul d'itinéraire

Nous avions choisi d'implémenter trois algorithmes de calcul d'itinéraires différents pour notre application. Ainsi nous avons réussi à implémenter les trois, même s'ils ne sont pas tous optimisés (par exemple le calcul de la table horaire s'effectue à chaque recherche d'itinéraire et l'algorithme pour le moins de correspondance peut être très long).

1.2.3 Interface

Etant l'interaction directe entre l'application et l'utilisateur (ou le gestionnaire), c'est une partie importante de l'implémentation. Nous avons rencontré des problèmes avec la gestion des différents *Scanner* que nous avons pu partiellement résoudre à l'aide de la création d'un scanner *scann* attribut static de notre classe *ApplictionRATP* propre à l'interface. Cependant, nous n'avons pas réussi à corriger le beugue ne prenant pas en compte les stations dont le nom comporte des espaces (ex : station *« Champs Elysées »*).

Le descriptif de l'interface sera approfondi dans un document *READ_ME* fournit qui explique l'utilisation de l'application.

1.3 Améliorations

Nous avons envisagé certaines améliorations de notre application : établir un choix aléatoire de l'identifiant de station au lieu de demander à l'utilisateur de le choisir dans une liste (pour gérer les quais opposés d'une même ligne dans une gare), gestion des formats erronés des paramètres entrés en console, gestion du problème de saturation de la mémoire suite à de multiples requêtes successives.

1.4 Compétences acquises

Par le biais de ce projet, nous avons pu découvrir l'algorithme du CSA [2] et les particularités d'une table horaire. Nous avons aussi découvert les principes de *course* et *route* de la RATP.

Grâce aux nombreuses données que nous avions à traiter dans ce projet, nous avons pu développer nos compétences en requête *SQL* ainsi qu'en gestion de bases de données depuis une interface de SGBD (notamment *SQLiteman*) ou même en interaction direct avec Java. Nous avons aussi appris à utiliser le logiciel DIA pour dessiner les diagrammes UML.

2. *Connection Scan Algorithm*

2 Développement et exécution du programme

2.1 Organisation du développement

Nous décrirons dans cette partie la hiérarchie de notre projet découpé en différents *packages* et *sous-packages* des différentes structures de l'application (cf. *Annexes 3.2* pour les diagrammes de classe Java). Pour approfondir le fonctionnement des différentes méthodes de classe, nous vous invitons à lire la *Javadoc* fournit avec le code. (Suite à un problème de compilation de la *Javadoc*, elle est accessible seulement dans le code Java)

2.1.1 Package *bdd*

(cf. Annexes page 12)

Ce package rassemble l'ensemble des classes ayant permis de créer la base de données et de la gérer, réparties dans les sous-packages suivants en fonction de leur visée :

***connexion* :** classe *Connecter*.

La classe *Connecter* permet d'établir la connexion plus simplement avec la base de données. Elle possède un attribut *connection* qui portera le lien de la connexion. Ensuite, deux méthodes ont été implémentées pour établir la connexion vers le bon driver (*seConnecter() : Connection*) et pour la fermer (*seDeconnecter() : void*).

***creationTable* :** classes *TableArret, TableCorrespondance, TableCourse, TableRoute*, *TableStation*.

Ensuite, nous avons dû créer nos différentes tables dans la base de données avant de pouvoir y importer les données RATP. Ainsi ces quatre classes contiennent les requêtes SQL permettant d'initialiser notre base de données (*creerTable() : void*).

***modificationTable* :** classe *LienStationBDD*.

Nous avons eu besoin au fil de notre projet, d'établir dans la base de données le lien entre une station et sa ligne de rattachement. Au vue du nombre important de stations du réseau (1259), nous avons choisi d'implémenter cette modification de la base de données par le biais de l'interpréteur Java et de ses boucles itératives, simplifiant les requêtes SQL (*addNumLinesToStation() : void*).

2.1.2 Package *calcul*

(cf. Annexes page 13)

Ce package comprend l'ensemble des classes qui permettent d'effectuer les différents calcul d'itinéraire. Elles sont réparties dans les sous-packages suivants :

tablesHoraire classe *TableCorrespondance*, *TableCourses*, *TableHoraire*, *Troncon*.

La table horaire parcourt les arrêts à étudier qui sont répertoriés dans la liste *ArretsTempo*, puis elle recense, par la création de *Troncon*, les différentes connections possibles à l'heure établie et comprises dans l'intervalle d'étude défini. Comme expliqué dans la partie *1.2.2 page 4*, elle se crée par fusion d'une table des courses et d'une table de correspondances. Les méthodes comprises dans ces classes permettent de lister les *Troncon* correspondant aux paramètres d'heure de départ et d'intervalle de temps défini pour générer la table horaire.

csa classe *Al*gorithmeCSA.

Nous avons choisi pour calculer l'itinéraire le plus rapide, d'implémenter un CSA [3] qui se base sur la table horaire précédemment décrite (*1.2.2 page 4*). Ainsi cette classe possède des attributs *static* qui sont modifiés au fil de ses trois méthodes permettant de calculer un itinéraire avec le CSA. Tout d'abord, une méthode *calculerTrajetOpti(stationDep, heureDep)* permet de calculer le temps optimal pour accéder à n'importe quelle station du réseau depuis la station de départ à l'heure de départ choisie. Ensuite, une méthode *retrouverIti(stationArr)* retourne la liste des stations composant l'itinéraire du trajet optimal. Pour finir, *calculerItineraire(stationDep, stationArr, heureDep)* permet de faire appel aux deux méthodes précédemment décrites pour retourner l'itinéraire le plus rapide.

csaTempsMarche classe *AlgorithmeCSACorrespondance.*

Pour l'itinéraire avec le moins de marche, c'est un CSA légèrement modifié que nous avons souhaité implémenter. Ainsi une première méthode *modifTable()* modifie la table horaire pour que seuls les temps de correspondances soient pris en compte dans le calcul du CSA et non plus ceux du temps de course. Puis la méthode *calculerItineraire(stationDep, stationArr, heureDep)* permet de lancer le calcul d'itinéraire par l'algorithme du CSA mais sur la table précédemment modifiée.

dijkstra classe *Dijkstra.*

Nous avons choisi pour calculer l'itinéraire avec le moins de correspondance, d'implémenter un algorithme de Dijkstra. Ainsi l'itinéraire retourné par l'algorithme est celui de poids minimal donc avec le moins de correspondances.

L'algorithme de Dijkstra s'appuie sur la recherche du plus court chemin d'une station de départ vers toutes les autres du réseau (*calculerDijkstra(id_statDep, id_statArr, heureDepart)*), selon un principe d'attribution d'un poids nul pour un tronçon d'une même course et d'un poids 1 pour un tronçon en correspondance. De ce fait, tant que la station considérée de poids minimal (*statPoidMin*) n'est pas la station d'arrivée, on la recherche à chaque itération, on l'ajoute à la liste *stationsVisitees* des stations vues puis on met à jour les nouveaux poids de chemin depuis la station de départ vers toutes les autres non encore vues (présentes dans la liste *stationsNonVisitees* et nommée *statTraitee*) selon le principe suivant :

$$poids^{new}_{statTraitee} = min(poids^{old}_{statTraitee}, poids_{statPoidMin} + poids^{Troncon}_{statPoidMin \rightarrow statTraitee})$$

A chaque mise à jour du poids d'une station, on met aussi à jour l'identifiant de la station qui précède, selon le plus court chemin, la station traitée dans la liste *stationPrecedente*. Ainsi crée-t-on un équivalent de liste « chaînée » entre les différentes stations qui permet de savoir par quelles stations passer pour atteindre n'importe quelle station depuis la station de départ *statDep*. C'est ensuite grâce à la méthode *retracerChemin(statDep, statArr, listStation, stationPrecedente),* parcourant la liste « chaînée » *stationPrecedente* depuis la station d'arrivée vers *statDep*, que l'itinéraire final entre *statDep* et *statArr* est généré.

La méthode *chercherMin(statDep, poidsStationNonVisitees, stationsNonVisitees, listStation)* permet de trouver l'identifiant de la station de poids minimal en comparant les éléments de la liste des poids à un compteur de poids initialisé à 0 et qui s'incrémente si aucun élément de la liste lui est égal. (Cette méthode de l'algorithme doit largement pouvoir être optimisée).

2.1.3 Package *application*

(cf. Annexes page 14)

3. *Connection Scan Algorithm : http ://www.dericbourg.net/2015/12/10/calcul-ditineraire-a-partir-des-donnees-ratp/*

main classes *ApplicationRATP*, *Main.*

Ce package est celui de lancement de l'interface. Il contient la classe *RunProject* (classe permettant à l'utilisateur de simplement lancer le programme directement), ainsi que la classe *ApplicationRATP* (gérant les méthodes de l'interface). Cette dernière possède les les méthodes *lancement_ Appli_ RATP()* qui permet de maintenir l'interface tant que l'utilisateur souhaite continuer à l'utiliser, et la méthode *relance_ Appli_ RATP()* appelée pour démarrer l'interaction avec l'utilisateur.

utilisateur classe *Requete.*

La méthode principale *Requete()* permet d'acquérir par le biais d'un scanner, les différents paramètres de l'itinéraire souhaité par l'utilisateur. Dans celle-ci, le nom des stations de départ et d'arrivée sont demandées, cependant avec la modélisation de nos stations (un quai de ligne = une station), cette information n'est pas assez précise. De ce fait, la méthode *acquerirIdStation(nomStation)* permet d'afficher les différents identifiants de stations de même noms avec leur numéro de ligne propre afin que l'utilisateur puisse choisir son quai de départ.

La méthode *nouvelItineraire()* permet de lancer la demande de requête de l'utilisateur et de faire appel à l'algorithme de calcul d'itinéraire adéquat en fonction du critère sélectionné. Elle fait notamment appel à une méthode *afficherItineraire(itineraire)* qui affiche les noms et lignes des stations de l'itinéraire.

gestionnaire classes *Parametres*, *Travaux.*

La classe *Parametres* est une classe ne contenant que un paramètre fixe représentant la fourchette de temps à considérée lors des calculs d'itinéraires. Nous n'avons pas eu le temps d'implémenter la possibilité de modification de ce paramètre, ainsi il faut le modifier directement dans le code Java.

La classe *Travaux* permet de gérer les travaux sur les stations et lignes par le biais d'une méthode principale *gestionReseau()* interagissant avec le gestionnaire et faisant appel aux méthodes adaptées à ses demandes.

2.1.4 Package *outils*

(cf. Annexes page 15)

Ce package ne contient qu'une seule classe : *Time*. Nous avons dû créer cette classe de manière à pouvoir gérer l'heure comme nous le souhaitions. Ainsi, un attribut *horaireMinute* permet de stocker tout horaire sous la forme d'un nombre entier de minute facilitant ensuite les calculs temporels tels que la somme (*somme(t1, t2) : Time*) ou la durée (*calculDuree(t1, t2) : Time*). D'autres méthodes ont été implémentées afin de simplifier les affichages d'heure et leurs traitements.

2.1.5 Package *reseau*

(cf. Annexes page 15)

Les six classes du package réseau ne sont rien d'autres que la modélisation en Java du réseau RATP avec ses *lignes*, *routes*, *courses*, *arrêts*, *stations* et *correspondances.*

Nous avons crée des méthodes *isEnTravaux()* et *setEnTravaux(enTravaux)* pour les classes *Ligne* et *Station* de manière à gérer la possibilité du gestionnaire du réseau de déclarer des lignes en travaux ou de les réhabiliter.

La fonction *listerStation(statDep, ArretsTempo)* de la classe *Station* permet de créer une liste de l'ensemble des stations viables du réseau en positionnant la station de départ *statDep* en premier élément de la liste. Cette fonction est ensuite largement utilisée dans l'ensemble des calculs de tables horaires et d'itinéraires.

2.2 Exécution

Pour décrire l'exécution de notre application, nous avons rédigé un document *READ_ME.txt*, auquel nous vous renvoyons, et qui explique la démarche à suivre pour l'utilisateur lors de l'utilisation de notre application.

3 Annexes

3.1 Diagramme de GANTT

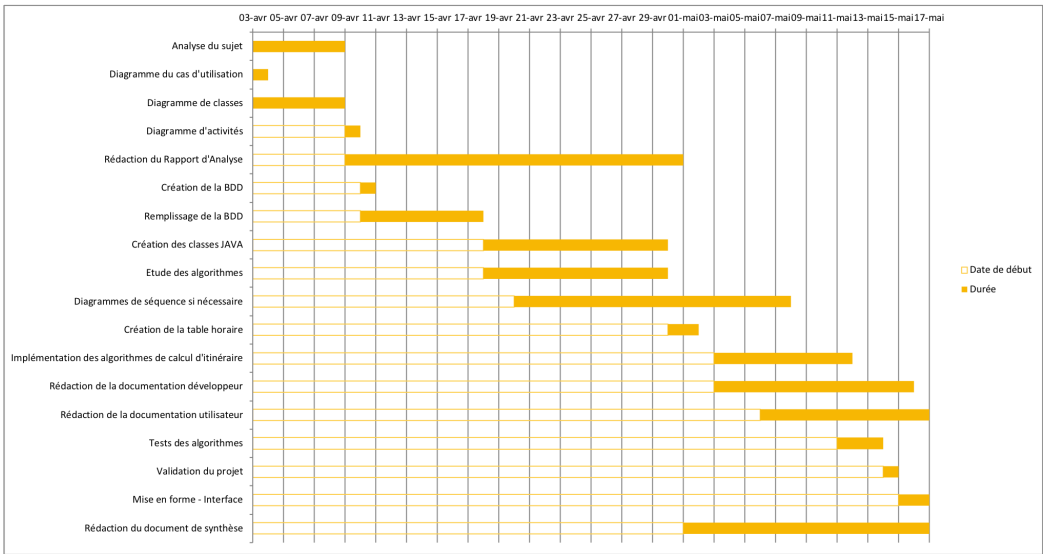

FIGURE 1 – Diagramme de GANTT prévisionnel

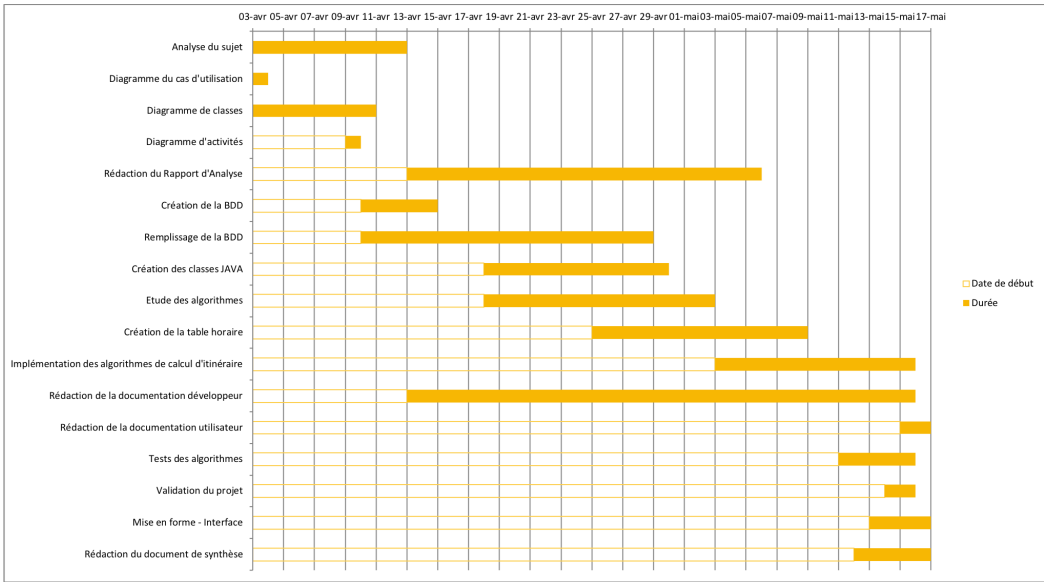

FIGURE 2 – Diagramme de GANTT réel

3.2 Diagramme de classe Java

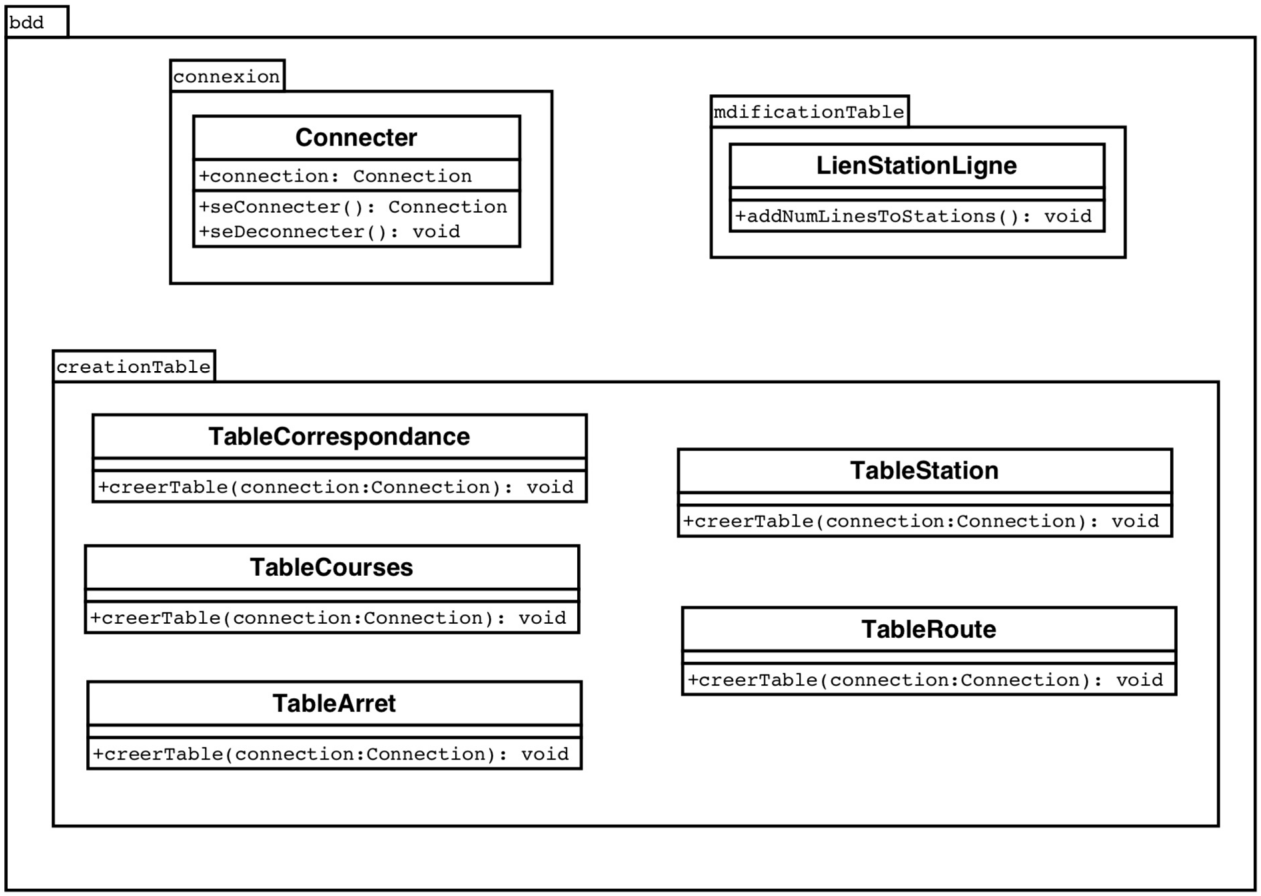

FIGURE 3 – Package *bdd*

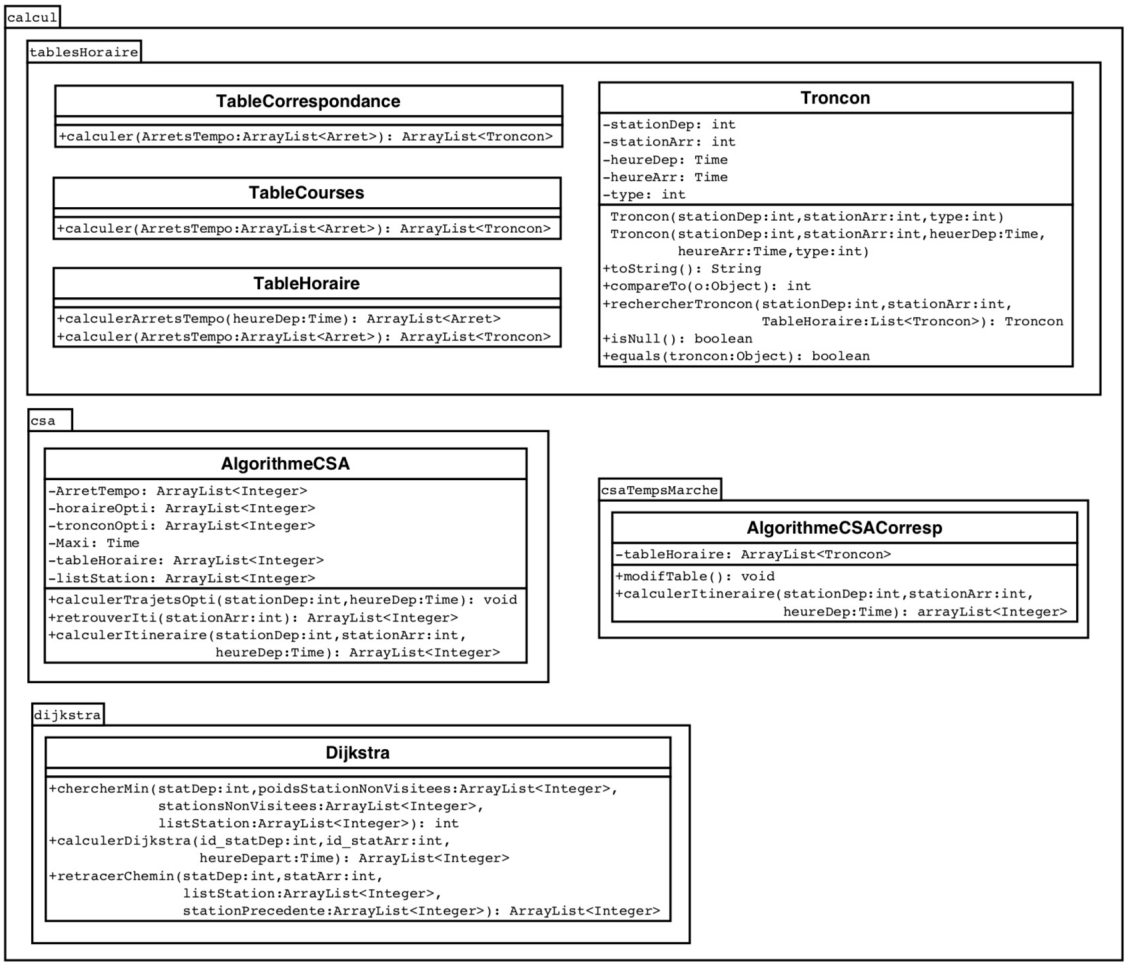

FIGURE 4 – Package *calcul*

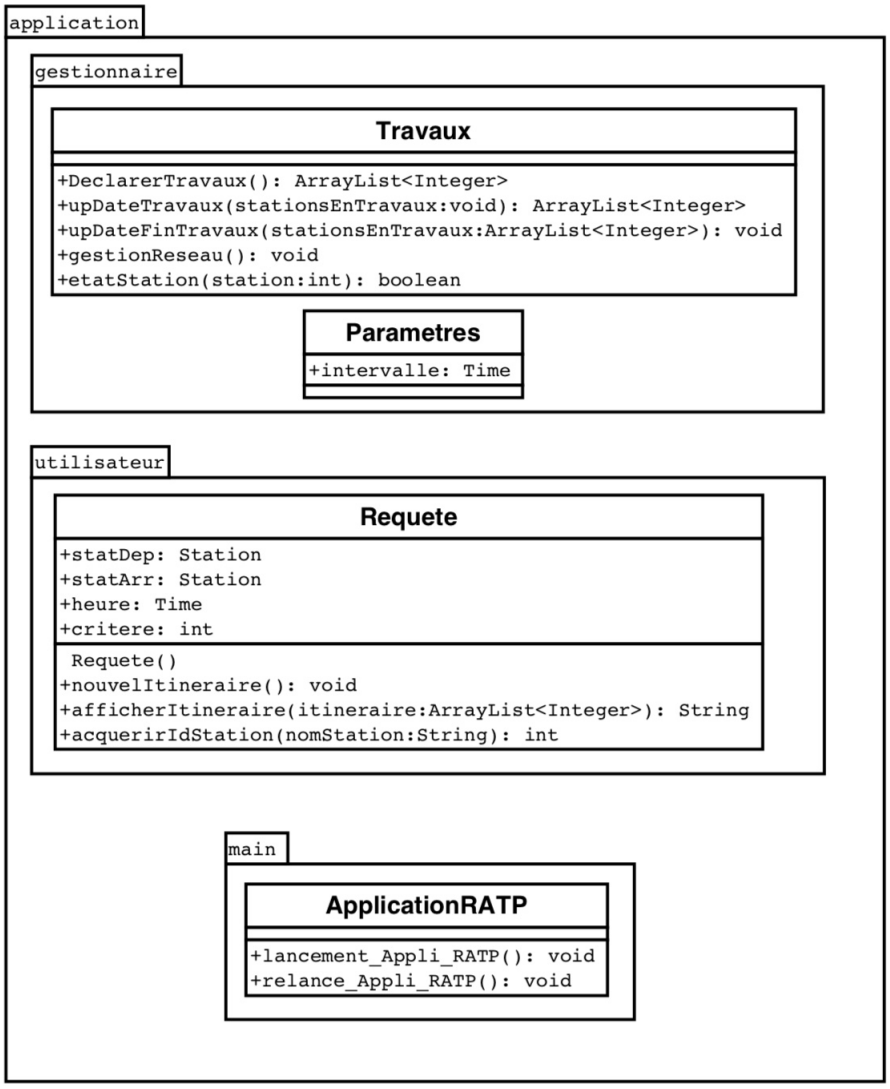

FIGURE 5 – Package *application*

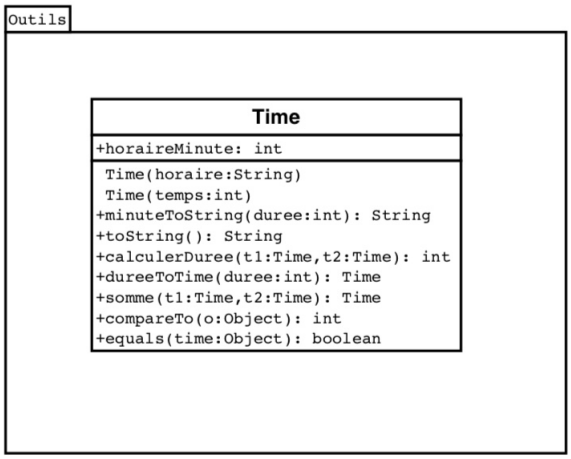

FIGURE 6 – Package *outils*

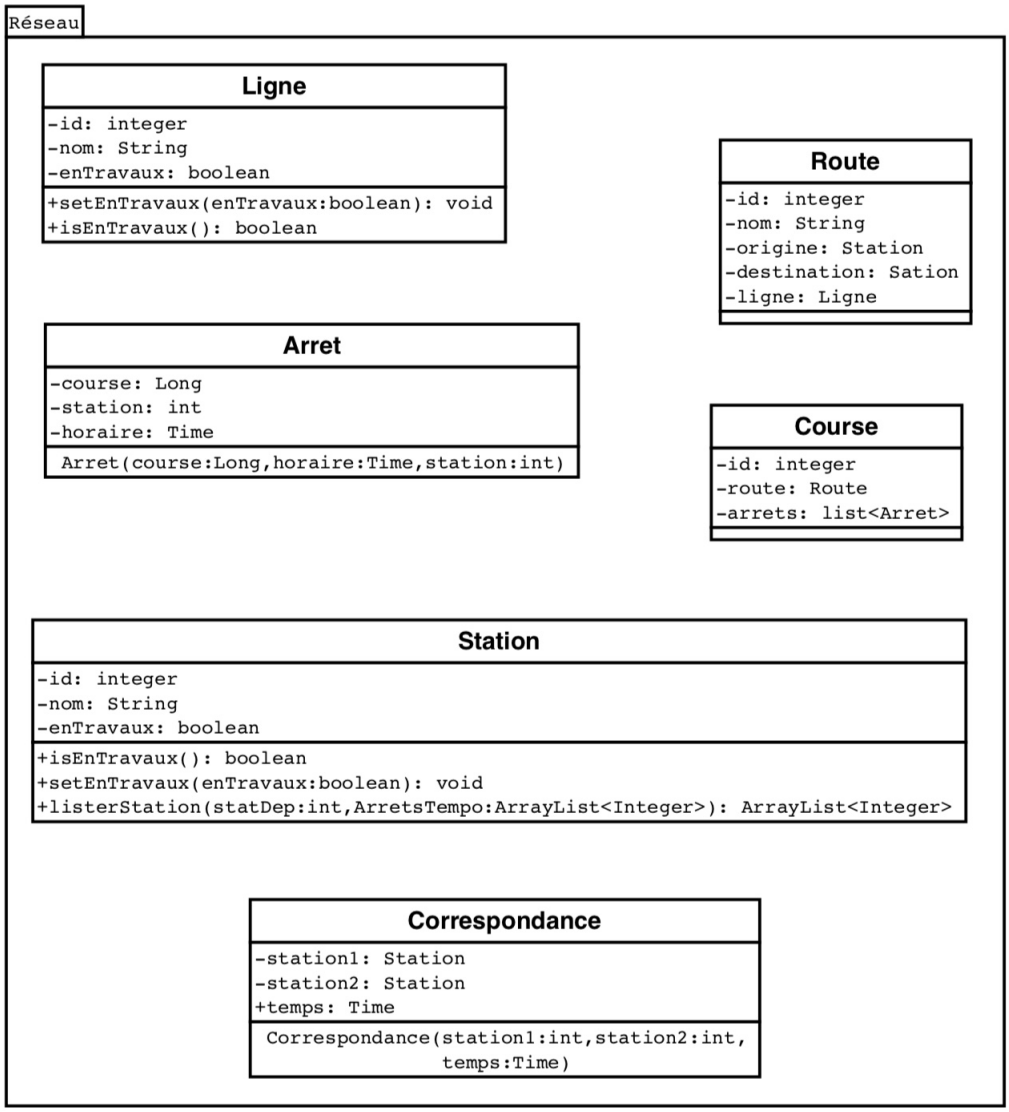

FIGURE 7 – Package *reseau*