PROJET JAVA ECLIPSE

Simulateur de Contrôle Aérien

Colin Mourard <colin.mourard@ensica.isae.fr>
Anthony Pagliai <anthony.pagliai@ensica.isae.fr>

Formation ISAE-ENSICA - Année Scolaire 2014-2015
Remise du Projet : 06.02.2015

Table des matières

1 Travail préparatoire

En premier lieu, nous avons consacré une grande partie de notre temps à bien lire le sujet et essayer de comprendre quelle architecture et quelle stratégie nous pouvions mettre en place. Pour cela, nous avons essayé de développer notre programme dans une logique de Conception Orientée Objet, c'est à dire dans une logique où l'architecture de nos packages/classes/méthodes s'appuie sur l'architecture réelle d'un tel système : structures/sous-structures/procédures.

En mettant en relation les éléments du cours avec les spécifications attendues dans ce sujet, nous sommes arrivés au tableau suivant (qui est non exhaustif) :

SPECIFICATIONS	NOTIONS DE COURS
Avion	Thread
Aéroport	Réseaux
Circuit d'attente Piste	Sémaphores - Exclusion mutuelle
Données de vol	Lecture - Ecriture dans un fichier
Traitement des erreurs	Exceptions

Tableau 1 : Réflexion initiale

2 Développement du programme

2.1 Architecture du programme

L'enjeu d'un programme de cette envergure réside dans la bonne séparation des spécifications en différents packages et classes afin d'avoir un programme doté d'une bonne modularité. Il sera ainsi plus clair et la maintenance de celui-ci s'en trouvera facilitée. Nous avons donc opté pour une architecture en 4 packages :

- Aéroport : ce package permet de centraliser toutes les fonctions réalisées par l'aéroport, à savoir la gestion du trafic par l'APP et le CCR et la gestion des différentes ressources telles que la piste, le circuit d'attente, le parking, etc.
- Avion : physiquement ce package représente la vie d'un avion avant, pendant et après le vol. Nous avons fait la distinction dans ce package entre l'avion et le vol auquel il est associé. En effet, dans la réalité, un avion et un vol ne sont pas deux unités indissociables, il nous a donc paru légitime de faire cette distinction.
- Réseau : ce package nous servira à établir la communication entre les différents aéroports.
- Exécutable : utilisation de la classe `Main` pour lancer le programme.

2.2 Etapes du développement

La stratégie que nous avons adopté pour le développement de notre logiciel est la suivante : nous faisons des hypothèses très simplificatrices afin de rendre le programme facile à réaliser. Peu à peu, nous levons une à une chacune de ces hypothèses simplificatrices afin que notre programme puisse réaliser chacune des spécifications attendues.

Ainsi, notre développement est construit autour des étapes suivantes :

- Faire décoller et atterrir un avion : à ce stade, l'aéroport n'est pas encore développé. Le développement consiste ici à mettre à jour les paramètres de vol de l'avion et faire s'endormir le thread pendant une durée choisie (assimilable au temps de vol). De plus, nous avons développé ici la simulation temporelle du programme (par exemple, 1 minute réalisée en 1 seconde dans le programme).
- Exclusion mutuelle (Piste) : à partir de cette étape, il est indispensable d'avoir développé l'aéroport. C'est ce que nous avons fait. En particulier nous nous sommes attardés sur l'APP qui est le garant de la sûreté de la piste. A ce stade, nous n'avons qu'un seul aéroport.
- Réseau : avant tout, il faut commencer par se demander quelle méthode utiliser pour échanger les données au sein du réseau. De part la fiabilité du transfert des données qu'il apporte, le protocole TCP nous a semblé être la meilleure solution. Ce point clé du développement nous amène naturellement vers le point suivant qui est « ajouter un deuxième aéroport ». Le deuxième enjeu de cette étape est de comprendre comment lire et écrire dans un fichier CSV qui contiendra tous les vols de notre simulation.
- Ajouter un deuxième aéroport : la principale difficulté à laquelle nous faisons face est de savoir comment gérer le fait que chaque aéroport peut à la fois envoyer et recevoir des données.

Les points suivants concernent toutes les spécifications liées à la régulation du trafic :

- Paliers / Circuit d'attente : d'abord un cercle, puis deux, etc. Dans un premier temps, nous ne limitons pas le nombre maximum d'avions sur chaque circuit, ni l'espacement angulaire qui est imposé. Il s'agit ici de commencer le développement du CCR.
- Nombre maximum d'avions par palier : il s'agit de complexifier notre circuit d'attente (5 avions maximum par palier).
- Espacement angulaire : la spécification attendue est d'espacer deux avions sur un même cercle d'attente d'un angle au moins égal à $\Pi/3$.
- Parking de l'aéroport : chaque aéroport est limité à 30 places de parking. Si un avion ne peut pas atterrir (nous comprendrons : ne peut pas se parker) alors il faut prévoir une solution de déroutement vers un autre aéroport.

Enfin, l'étape finale de notre développement consiste à traiter les exceptions et à réaliser les tests unitaires afin d'assurer la conformité de notre programme et de facilité sa maintenance.

2.3 Codage de chaque spécification du programme

L'enjeu de cette sous-section est de donner quelques précisions sur la manière dont sont codées nos classes dans ce projet.

2.3.1 Gestion des fichiers CSV

Notre package `Réseau` contient une seule classe : la classe `Communication`. Cette classe nous permet de gérer tout ce qui est lié de près ou de loin à la communication, que ce soit entre plusieurs aéroports ou pour la lecture et l'écriture des fichiers CSV. Avant de lancer la simulation, il faut placer le fichier CSV dans le dossier parent du projet. Ensuite, nous nous sommes inspirés de codes trouvés sur Internet pour lire et écrire efficacement dans ces fameux fichiers CSV. Pour ce faire, le plus simple est de créer des objets de type `BufferedReader` (méthode `readLine()`) pour la lecture et de type `FileWriter` (méthode `write(String info)`) pour l'écriture. En définissant bien qu'à chaque ligne correspond un vol et que chaque information d'un vol est séparée de la suivante par un « ; », notre programme répond aux spécifications attendues.

2.3.2 Mise en place du réseau

Rappelons que nous avons choisi d'utiliser le protocole TCP pour notre réseau puisqu'il permet un transfert fiable et bilatéral des données. Cette partie a cependant mis du temps pour nous à se concrétiser puisqu'avec notre unique classe `Main` comme classe exécutable, nous ne voyions pas comment faire...

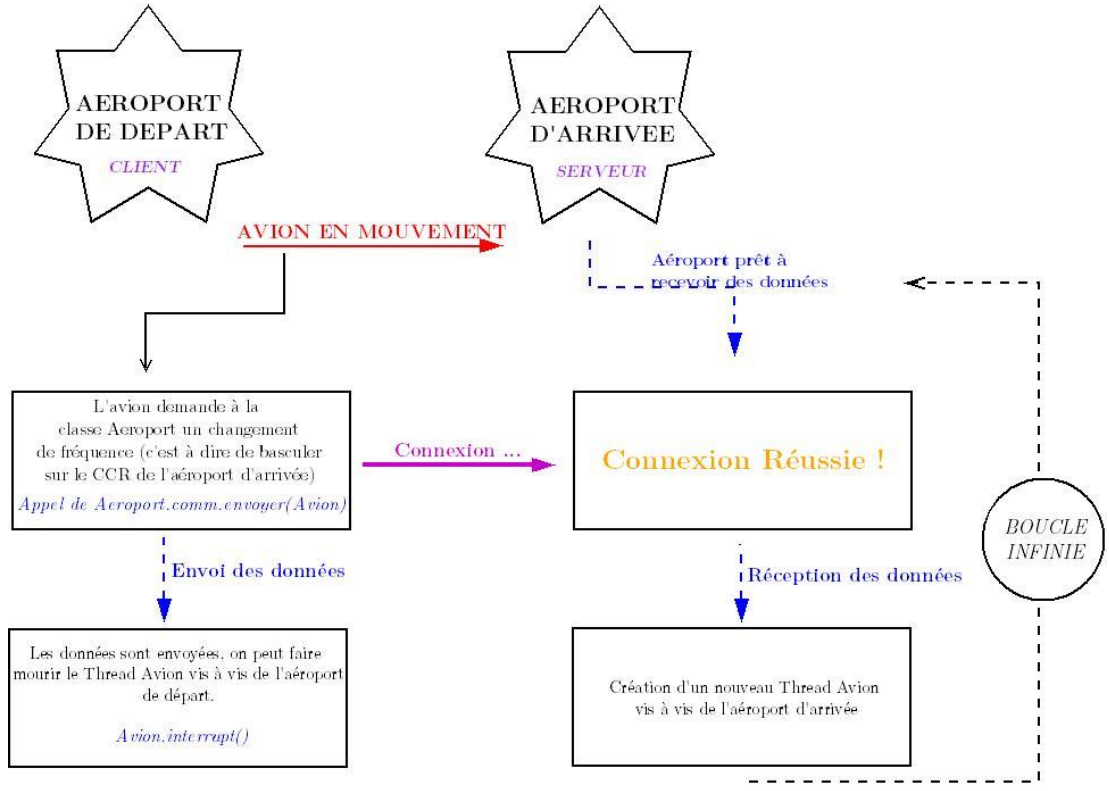

Figure 1 : Schéma de principe de notre réseau

L'idée que nous avons eu pour développer notre réseau est de créer une méthode `run()` dans notre classe `Aeroport`. Notre aéroport se comporte alors comme un thread. Au début de notre classe `Main`, on lance ce thread aéroport qui s'exécute en tâche de fond. Il écoute en permanence le port qu'on lui a attribué en attendant qu'un aéroport client s'y connecte. Cette méthode tourne donc en boucle et fait office de classe serveur (du point de vue du protocole TCP) pour les autres aéroports qui se comporteront comme des clients lorsqu'ils voudront envoyer des données. Pour assurer le bon déroulement de notre simulation, nous imposons à notre réseau de ne pouvoir envoyer les données que d'un seul vol à la fois (à l'aide d'une Sémaphore initialisée en mutex). Le schéma de fonctionnement de notre réseau est présenté sur la Figure 1.

Enfin, au terme de notre simulation, nous fermons automatiquement le module réseau de la classe `Aeroport` en envoyant un avion qui déclenche le critère d'arrêt. Nous préférons faire ce choix plutôt que d'interrompre le thread aéroport car avec un critère d'arrêt, nous sommes sûr de fermer proprement nos `Socket` (et notamment la `ServerSocket` ...).

2.3.3 Ressources partagées

Nous sommes partis sur une stratégie simple : garder une seule manière de traiter les exclusions mutuelles. Nous avons choisi d'utiliser à chaque fois une sémaphore (initialisée en mutex ou pas suivant les cas de figures).

2.4 Diagramme UML

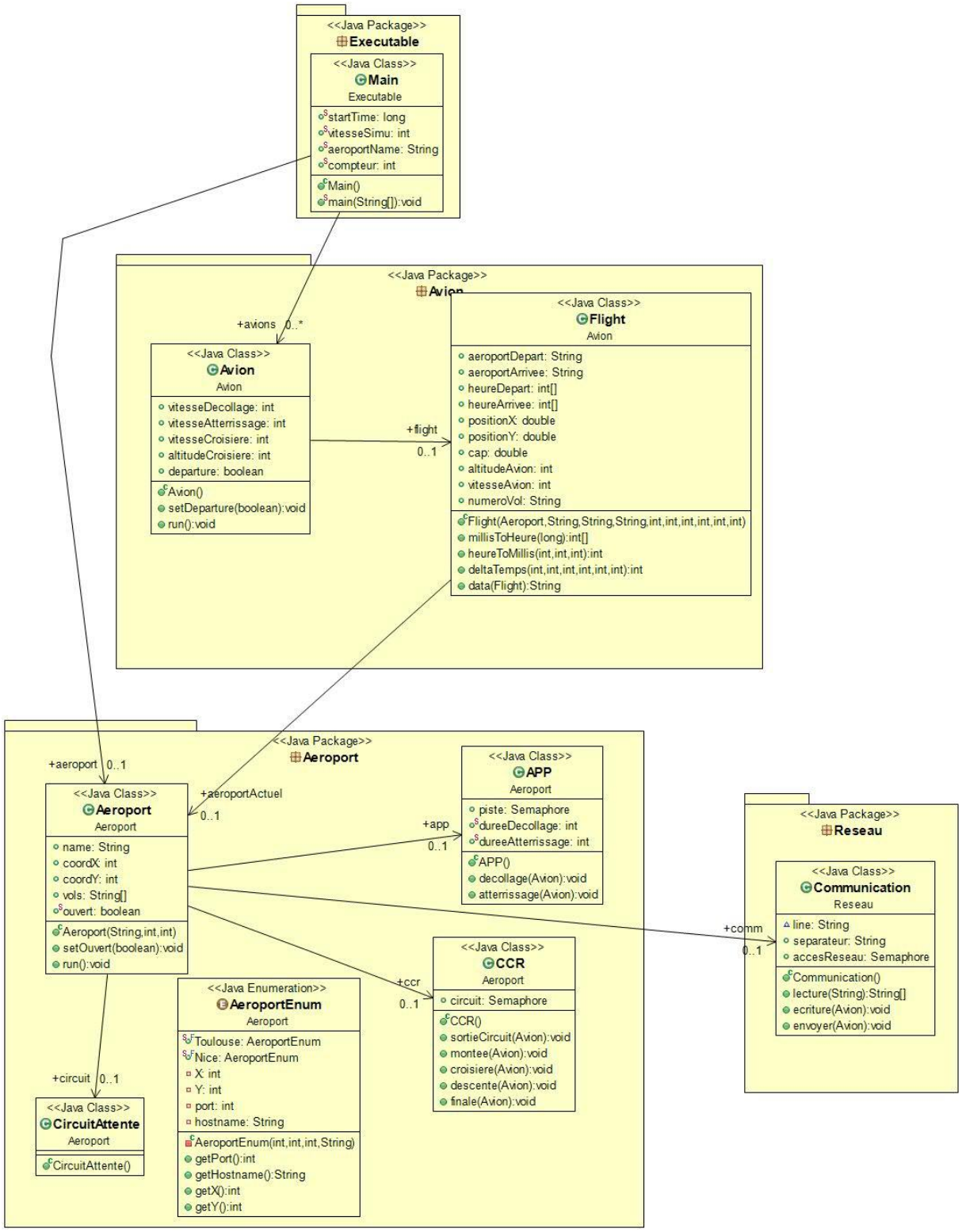

Figure 2 : Diagramme UML de notre projet

3 Limites et axes d'améliorations

3.1 Elégance du code

Pour des soucis de simplifications, lorsque nous lisons les fichiers CSV en entrée de programme, nous les plaçons dans un tableau d'avions. Pour nous simplifier la taille, nous avons initialisé ce tableau avec une taille de 1000, jugée comme infinie à l'échelle de notre simulation. Bien entendu, ce n'est pas la manière la plus élégante de procéder. Il aurait peut être mieux fallu utiliser des tableaux dynamiques ou des listes.

Par ailleurs, dans notre programme, il y a ligne 51 de la classe `Main` un endormissement de thread. Ceci pour permettre au programme de ne pas s'exécuter trop vite (i.e. de s'exécuter entièrement avant de recevoir tous les avions qui veulent atterrir dans cet aéroport). Ce qui n'est pas élégant, c'est la durée arbitraire que nous avons choisie.

Enfin, pour nos paramètres de vol, parfois on arrive sur l'aéroport final avant d'être descendu à 2000m à la vitesse de 250km/h. Cela provoque une descente et une diminution de la vitesse en restant « sur place » : tel un hélicoptère !

3.2 Trajectoire

Le sujet donne comme spécification de sortir des cercles d'attentes tangentiellement pour rejoindre l'aéroport d'arrivée. Nous ne faisons pas exactement comme ça. Nous parcourons le cercle jusqu'au bout cap de sortie puis nous faisons un virage instantané à 90° pour suivre la bonne route. De même pour entrer dans le cercle d'attente de l'aéroport d'arrivée.

3.3 Décalage de l'horloge

Il y a un léger décalage entre les deux heures de démarrage du programme (sur chaque machine) puisqu'il est humainement impossible de lancer exactement en même temps chaque programme. Ce décalage n'a pas de conséquence sur les horaires de décollage mais plutôt sur les horaires d'atterrissages. Il aurait fallu réussir à synchroniser les deux horloges.

3.4 Spécifications non codées

Les spécifications du sujet que nous n'avons pas codées sont les suivantes :

- adaptation de la vitesse d'approche pour respecter l'horaire d'arrivée
- calcul de l'horaire de départ en fonction de l'heure à laquelle on souhaite arriver
- les 4 cercles (à 500, 1000, 1500 et 2000m) du circuit d'attente et l'espacement angulaire sur chacun d'eux.
- les tests unitaires

La non présence de ces spécifications est principalement due à un manque de temps. Il a donc fallu faire des choix, nous avons préféré mettre l'accent sur le réseau, la gestion des threads et des ressources partagées.

4 Mode d'emploi pour l'utilisateur

Pour utiliser correctement notre programme, procédez de la manière suivante :

1. Préparer un fichier CSV pour chaque ordinateur (ie pour chaque aéroport) sous le format : « nom_aeroport.csv ». Par exemple, pour l'aéroport de Toulouse : « Toulouse.csv ». Les noms d'aéroports disponibles se trouvent dans la classe `AeroportEnum`.
2. Changer dans la classe `Main` : l'attribut `aeroportName`, au même nom que le fichier CSV (sans l'extension).
3. Changer dans la classe AeroportEnum : les noms d'hôte des différents ordinateurs qui utilisent le programme. Par exemple, si l'aéroport de Toulouse tourne sur l'ordinateur dont le nom d'hôte est « Mon_Ordi » il faut écrire « Mon_Ordi » comme attribut de Toulouse.
4. Lancer la méthode `Main`.
5. Récupérer le fichier CSV final à la fin. Il est de la forme : « nom_aeroportArrivee.csv »