

# Airline Management System

Cet énoncé comporte 4 pages.

## 1. Objectif

L'objectif de ce devoir est de mettre en oeuvre les bonnes pratiques et patrons de conceptions en programmation orientée-objet. Il s'agit de réaliser une application de gestion de système de gestion aéroportuaire, dont le comportement global est réparti dans un ensemble de classes, présentées dans la figure 1.

L'application doit permettre à un client de créer des aéroports, des compagnies aériennes ainsi que des vols. Le point d'entrée de l'application est le **SystemManager** *aucune instance ne sera créé sans le SystemManager c'est lui qui gère le programme*. Chaque compagnie Airline est associé à un ensemble de vols (**Flights**) *forte composition*. Un vol a un aéroport de départ (origin) et un aéroport d'arrivée (destination). L'origine et la destination ne peuvent être les mêmes. Il faudra donc penser à vérifier au moment de l'instanciation de l'objet chaque départ et chaque arrivée.

Chaque vol comprend des classes tarifaires (e.g., première classe, business) appelées **flight section**.

Chaque classe tarifaire comprend des sièges organisés en rangs (ligne et colonne).

### 1.1. Consigne

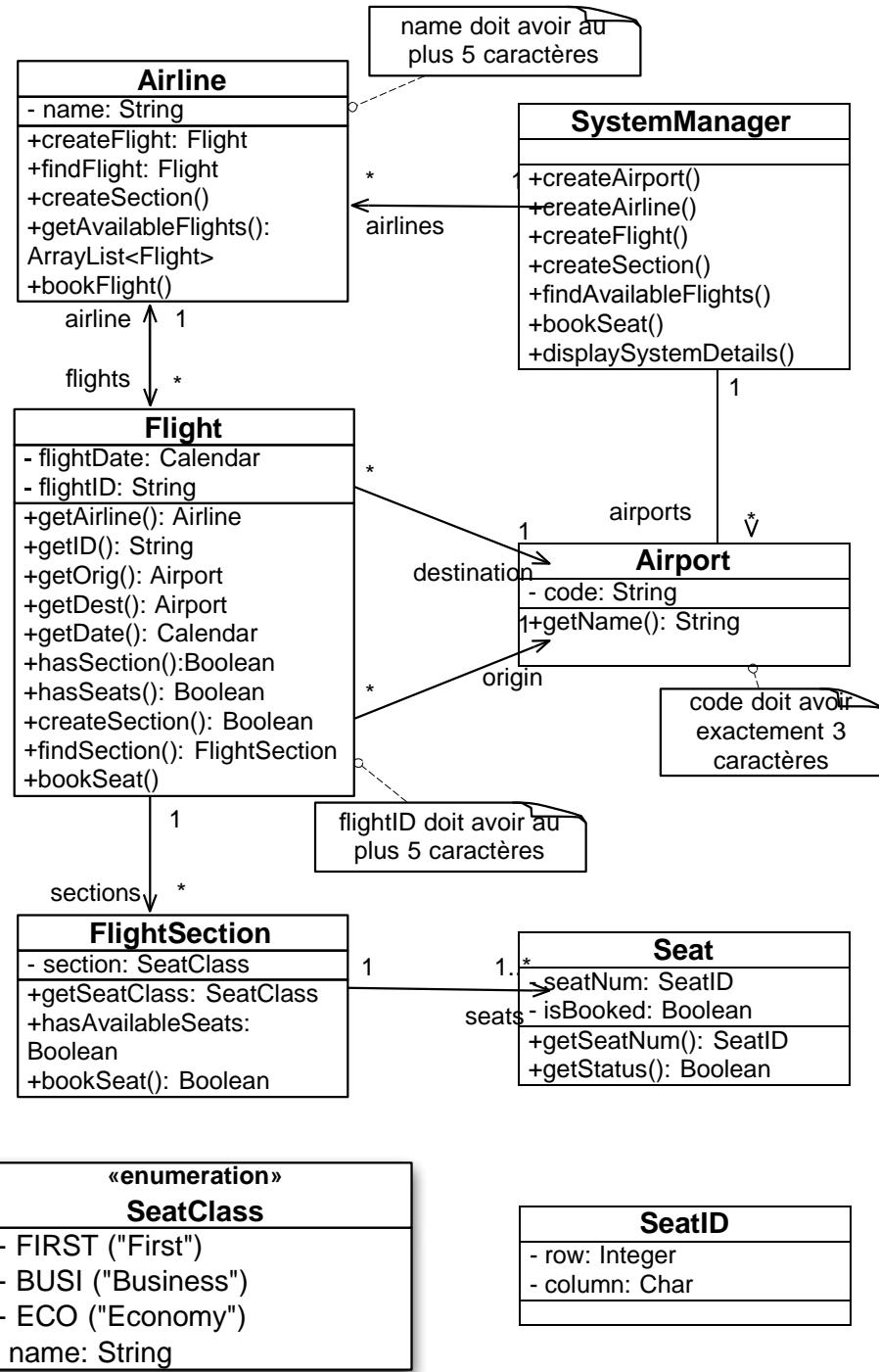
Vous devez faire fonctionner votre application sur le jeu minimal d'instructions fourni dans le programme principal à la fin de ce document. *Dans un premier temps penser à réaliser vos tests qui répondront à vos règles de gestion ensuite* utiliser un framework de test (JUnit, TestNG), pour fournir des jeux de test pour les fonctions critiques de l'application que vous identifierez et dont vous justifierez les caractères critiques (pas de tests sur les getters/setters!).

## 2. SystemManager

C'est le point d'entrée de l'application il faut donc s'assurer de l'unicité de l'instance qui a pour but de créer un point d'entrée universel pour les autres instances pensez donc à utiliser un **Design patern singleton**. Les clients interagissent avec l'application en appelant les opérations offertes par **SystemManager**. Ce dernier est relié au aéroport et compagnies aériennes dans l'application. A sa création, le **SystemManager** ne possède aucun aéroport ou compagnie aérienne. Pour les créer les opérations `createAirport()` et `createAirline()` doivent être invoqués.

Le **SystemManager** contient également les opérations pour créer les classes tarifaires, trouver les vols disponibles entre deux aéroports, et réserver des sièges sur un vol. Pour afficher toute l'information concernant les aéroports les compagnies et les vols, classes tarifaires et sièges, on invoque l'opération `displaySystemDetails()`.

- `createAirport(String n)` : crée un objet de type `Airport` et le lie au `SystemManager`. L'aéroport doit avoir un code `n`, dont la longueur est exactement égale à 3. Deux aéroports différents ne peuvent avoir le même code. Pensez donc à vérifier et à rechercher une structure de données efficace qui permettra de répondre à cette problématique



**Figure 1** - Description architecturale de Airline Management System

- `createAirline(String n)` : crée une compagnie aérienne et la lie au SystemManager. Le nom n d'une compagnie doit être de longueur au plus égale à 5. Deux compagnies différentes ne peuvent avoir le même nom. **Pensez donc à vérifier et à rechercher une structure de données efficace qui permettra de répondre à cette problématique**
  - `createFlight(String n, String orig, String dest, int year, int month, int day, String id)` : crée un vol pour une compagnie n, en invoquant l'opération `createFlight` sur la classe `Airline`.
  - `createSection(String air, String fIID, int rows, int cols, SeatClass s)` : crée une section tarifaire, de classe s, pour un vol identifié par fIID, associé à une compagnie air, en invoquant l'opération `createSection()` de la classe `Airline`. La section contiendra le nombre de lignes et de colonnes.

- `findAvailableFlights(String orig, String dest)` : trouve tous les vols pour lesquels il existe encore des sièges disponibles, entre les aéroports de départ et d'arrivée.
- `bookSeat(String air, String fl, SeatClass s, int row, char col)` : réserve le siège dont la position est indiquée par row et col (e.g. 15A), sur le vol fl de la compagnie air.
- `displaySystemDetails()` : affiche toute l'information concernant tous les objets (e.g. aéroports, compagnies, vols, sièges, ...) dans le système.

## 3. Airport

Un objet de cette classe représente un aéroport. Il possède un nom, de longueur 3 caractères.

## 4. Airline

Cette classe définit une compagnie aérienne. Une compagnie possède zéro ou plusieurs vols en cours. A la création d'un objet de ce type, il n'y a initialement aucun vol. Chaque vol doit avoir un identifiant unique.

- `Flight createFlight(String orig, String dest, Calendar date, String id)` : crée un vol pour une compagnie aérienne.
- `Flight findFlight(String n)` : trouve un vol dont l'identifiant est n.
- `createSection(String flID, int rows, int cols, SeatClass s)` : crée une section tarifaire de classe s, pour un vol dont l'identifiant est flID. La section contiendra le nombre de lignes et de colonnes.
- `ArrayList<Flight> getAvailableFlights(Airport orig, Airport dest)` : trouve tous les vols sur lesquels il existe encore des sièges disponibles, entre les aéroports de départ et d'arrivée. Attention voir si la structure List n'est pas plus efficace
- `bookFlight(String fl, SeatClass s, int row, char col)` : réserve un siège dont la position est indiquée par row et col (e.g. 15A) dans la section tarifaire s, sur le vol fl.

## 5. FlightSection

Cette classe définit une classe (ou section) tarifaire. Chaque section possède une classe (première, affaires, ou économique) et au moins 1 siège. Une FlightSection possède des attributs nombre de rows et nombre de columns, afin de savoir combien de sièges elle contient et le calcul du nombre de sièges disponibles.

Une section tarifaire contient au plus 100 rangées de sièges et au plus 10 sièges par rangée.

- `hasAvailableSeats()` renvoie vrai si et seulement si la section possède encore des sièges disponibles (non réservés).
- `bookSeat()` réservé le premier siège disponible. Son utilisation est conditionnée à celle de `hasAvailableSeats()`.
- `boolean bookSeat(SeatID sid)` réservé le siège à l'emplacement désigné par le paramètre sid, si ce siège est disponible.

## 6. Seat

Cette classe définit un siège. Un siège possède un identificateur, qui indique sa rangée et sa colonne (caractère allant de A à J). Il possède également un statut qui indique s'il est réservé ou pas.

## 7. Flight

Cette classe aura pour but de construire un vol, elle récupéra toute les données nécessaires grâce à ses méthodes il faudra donc vérifier avant l'instanciation chaque règle de gestion d'un vol

Voilà les méthodes qui composent cette classe :

```
+getAirline(): Airline  
+getID(): String  
+getOrig(): Airport  
+getDest(): Airport  
+getDate(): Calendar  
+hasSection(): Boolean  
+hasSeats(): Boolean  
+createSection(): Boolean  
+findSection(): FlightSection  
+bookSeat()
```

## 8. Client de l'application

Un exemple de client de cette application est fourni dans la classe ClientAMS. Ce client appelle des opérations de la classe SystemManager.

Vous êtes invités à étendre cette classe client avec d'autres invocations pour tester le comportement attendu de votre application.

```
1 public class ClientAMS {
2     public static void main (String[] args) {
3         SystemManager res = new SystemManager();
4             // Airports
5             res.createAirport("DEN");
6             res.createAirport("DFW");
7             res.createAirport("LON");
8             res.createAirport("DEN");
9             res.createAirport("CDG");
10            res.createAirport("JPN");
11            res.createAirport("DEN"); // Pb d'unicite
12            res.createAirport("DE"); // Invalide
13            res.createAirport("DEH");
14            res.createAirport("DRIrdn3"); // Invalide
15
16             // Airlines
17             res.createAirline("DELTA");
18             res.createAirline("AIRFR");
19             res.createAirline("AMER");
20             res.createAirline("JET");
21             res.createAirline("DELTA");
22             res.createAirline("SWEET");
23             res.createAirline ("FRONTIER"); // Invalide
24
25             // Flights
26             res.createFlight("DELTA", "DEN", "LON", 2008, 11, 12, "123");
27             res.createFlight("DELTA", "DEN", "DEH", 2009, 8, 9, "567");
28             res.createFlight("DELTA", "DEN", "NCE", 2010, 9, 8, "567"); // Invalide
29
30             // Sections
31             res.createSection ("JET", "123", 2, 2, SeatClass.economy);
32             res.createSection ("JET", "123", 1, 3, SeatClass.economy);
33             res.createSection ("JET", "123", 2, 3, SeatClass.first);
34             res.createSection ("DELTA", "123", 1, 1, SeatClass.business);
35             res.createSection ("DELTA", "123", 1, 2, SeatClass.economy);
36             res.createSection ("SWSERTT", "123", 5, 5, SeatClass.economy); // Invalide
37
38             res.displaySystemDetails ();
39
40             res.findAvailableFlights ("DEN", "LON");
41
42             res.bookSeat("DELTA", "123", SeatClass.business, 1, 'A');
43             res.bookSeat("DELTA", "123", SeatClass.economy, 1, 'A');
44             res.bookSeat("DELTA", "123", SeatClass.economy, 1, 'B');
45             res.bookSeat("DELTA", "123", SeatClass.business, 1, 'A'); // Deja reserve
46
47             res.displaySystemDetails ();
48
49             res.findAvailableFlights ("DEN", "LON");
50
51 }
```