Airline Management System

Cet énoncé comporte 8 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. Chaque compagnie Airline est associé à un ensemble de vols (Flights). Un vol a un aéroport de départ (origin) et un aéroport d'arrivée (destination). L'origine et la destination ne peuvent être les mêmes.

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. Vous devez également 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 le caractère critique (pas de tests sur les getters/setters !).

Vous utiliserez les collections les plus appropriées pour la gestion des différentes listes. Comme par exemple, il serait bien d'utiliser l'interface Map avec les HashMap ou HashTable, si vous pouvez faire du O1.

Vous utiliserez les flux pour parcourir les différentes listes.

Vous utiliserez le design pattern builder pour la construction d'un vol (section + seat).
Vous utiliserez le design pattern de la factory pour la création des différents objets.

Vous n'oublierez pas de tester chaque méthode de votre programme. Pour cela vous pouvez modifier les types de retour void en boolean pour pouvoir tester les résultats.

Vous êtes libres de rajouter des fonctionnalités (ex : annulation).

Les test sont classé par priorité (de 1 à 5 : 1 priorité maximal, 5 priorité minimal).

Vous n'oublierez pas de générer les méthodes equals() et hashCode partout où cela est nécessaire.
Bonus : Ajouter la persistance des données en utilisant les .properties ou une base de données.

2. SystemManager

C'est le point d'entrée de l'application. Les clients interagissent avec l'application en appelant les opérations offertes par SystemManager. Ce dernier est relié aux aéroports 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 invoquées.

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().

Pour cette classe vous utiliserez le design pattern du Singleton, pour garantir l'unicité du point d'entrée.

— 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.

Pour cette méthode vous vous créerez les tests suivant :

- Le nombre de caractères doit être strictement égal à 3. (ex de test : 3 caractères, 2 caractères et 4 caractères). 3
- Le nom ne doit pas déjà exister. (Testé deux nom différents et deux fois le même nom). 1

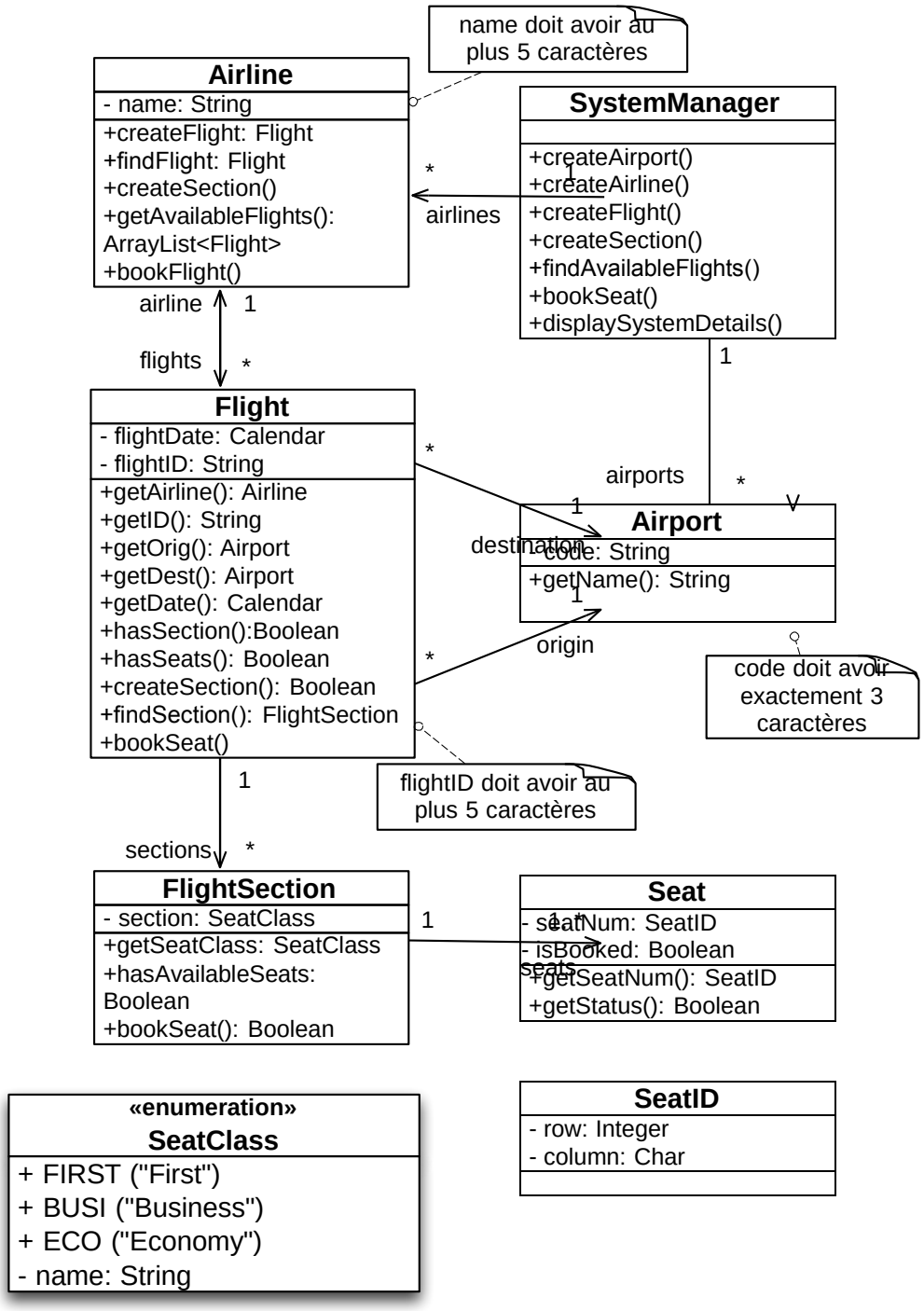

Figure 1 – Description architecturale de Airline Management System

— 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.

Test à faire :

- Le nombre de caractères doit être inférieur ou égal à 5. (ex de test : 5 caractères, 6 caractères et 4 caractère). 4
- Le nom ne doit pas déjà exister. (Testé deux nom différents et deux fois le même nom). 2

— 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.

Test à faire :

- La destination doit exister. (ex de test : La destination existe et la destination n'existe pas) 2
- L'origine doit exister. (ex de test : L'origine existe et l'origine n'existe pas) 2
- L'id ne doit pas déjà exister. (ex de test : L'id existe et l'id n'existe pas) 1
- La compagnie doit exister. 3

— createSection(String air, String flID, int rows, int cols, SeatClass s) : crée une section tarifaire, de classe s, pour un vol identifié par flID, associé à une compagnie air, en invoquant l'opération createSection() de la classe Airline. La section contiendra le nombre de lignes et de colonnes.

Test à faire :

- Le flightId doit exister. 1
- La compagnie doit exister. 2
- Le nombre de siège d'une section doit être au minimum égal à 1. 5
- Le nombre de rangées ne doit pas dépassées 100. 3
- Le nombre de siège par rangées ne doit pas dépassées par 10. 4

— findAvailableFlights(String orig, Strin dest) : trouve tous les vols pour lesquels il existe encore des sièges disponibles, entre les aéroports de départ et d'arrivée. Cette méthode invoquere getAvailableFlights de la classe Airline.

Test à faire :

- Crée des vols avec des places disponibles est regardé le résultat retourné. 3

-bookSeat(String air,String fl,SeatClass s) : réserver le premier siège disponible dans la class s.

— 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.

Test à faire :

- Le siège doit exister. 1
- La SeatClass doit exister. 2
- La compagnie doit exister. 2
- Le flightId doit exister. 2
- Vérifier que le siège a bien été réservé en utilisant la méthode getStatus de la classe Seat. 3

— 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.

Test à faire :

- Deux aéroports ne peuvent pas avoir le même nom. 2
- Le nombre de caractères du nom de l'aéroport doit être strictement égal 3. 1

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 n, String orig, String dest, Calendar date, String id) : crée un vol pour une compagnie aérienne.

Test à faire :

Même test que ceux décrit dans 2. SystemManager createFlight.

- La destination doit exister. (ex de test : La destination existe et la destination n'existe pas) 2
- L'origine doit exister. (ex de test : L'origine existe et l'origine n'existe pas) 2
- L'id ne doit pas déjà exister. (ex de test : L'id existe et l'id n'existe pas) 1
- La compagnie doit exister. 3

— Flight findFlight(String n) : trouve un vol dont l'identifiant est n.

Test à faire :

- Créer des vols est testé le résultat le résultat de la récupération. 3

— 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.

Test à faire :

- Le flightID doit exister. 1
- La SeatClass doit exister. 2
- Le nombre de siège d'une section doit être au minimum égal à 1. 5
- Le nombre de rangées ne doit pas dépassées 100. 3
- Le nombre de siège par rangées ne doit pas dépassées par 10. 4

— 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.

Test à faire :

- L'origine doit exister. 1
- La destination doit exister. 1
- Créer des vols avec des places disponibles et non disponibles et comparer le résultat obtenu avec celui prévue. 3

-bookFlight(String **fl**, SeatClass s) : réserve un siège un siège disponible.

— 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.

Test à faire :

- Le fl (flightID) doit exister. 2
- La SeatClass doit exister. 2
- Le numéro de la colonne doit être compris entre 1 et 10. 3
- Le numéro de ligne doit être compris entre 1 et 100. 3
- Le siège doit exister. 1
- Le siège ne doit pas déjà être reservé. 1

5. Flight

Cette classe définit un vol. Un vol possède un identifiant, (flightID) ainsi qu'une date de vol (flightDate).

hasSection(Flight fID) : Cette méthode retourne un booléen qui indique si le flightID comporte une section ou non.

Test à faire :

- Créer un vol sans section est vérifié que hasSection retourne bien false. 4
- Créer un vol avec une section est vérifé que hasSection retourne bien true. 4
- Le flightID doit exister. 4

hasSeat(Flight fid) : Cette méthode retourne un booléen qui indique si le flightID comporte des sièges ou non.

Test à faire :

- Créer un vol avec une section avec des sièges est vérifié que hasSeat retourne true. 4
- Le flightID doit exister. 3

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.

Test à faire :

- Le flightID doit exister. 1
- La SeatClass doit exister. 2
- Le nombre de siège d'une section doit être au minimum égal à 1. 5
- Le nombre de rangées ne doit pas dépassées 100. 3
- Le nombre de siège par rangées ne doit pas dépassées par 10. 4

findSection(flight fID, SeatClass sClass) : Cette méthode permet de récupérer une section en fonction d'un flightID et d'une SeatClass.

Test à faire :

- Le flightID doit exister. 2
- Le SeatClass doit exister. 2
- Créer un vol avec une section et vérifier que findSection la trouve bien. 3

-bookSeat(SeatClass s,int row,char col) reserve le siege donné en parametre.

-bookSeat(SeatClass s) réserver le premier siège disponible. Son utilisation est conditionnée à celle de hasAvai- lableSeats().

Test à faire :

- Créer une section avec un siège disponible et vérifier en utilisant hasAvailableSeats() retourne false et que bookSeat() retourne true. 2
- Créer une section ne comportant aucun siège de disponible et vérifier en utilisant hasAvailableSeats() retourne false et que bookSeat() retourne false. 2

6. 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).

Test à faire :

- Créer une section avec des sièges disponibles et vérifier qu'on récupère bien true. 2

- Créer une section ne comportant aucun siège de disponible et vérifier que l'on récupère bien false. 2

— bookSeat() réserver le premier siège disponible. Son utilisation est conditionnée à celle de hasAvailableSeats().

Test à faire :

- Créer une section avec un siège disponible et vérifier en utilisant hasAvailableSeats() retourne false et que bookSeat() retourne true. 2

- Créer une section ne comportant aucun siège de disponible et vérifier en utilisant hasAvailableSeats() retourne false et que bookSeat() retourne false. 2

— boolean bookSeat(SeatID sId) réserver le siège à l'emplacement désigné par le paramètre sID, si ce siège est disponible.

Test à faire :
- Le sID doit exister. 1
- Créer une section avec des sièges disponibles et un numéro qui n'existe pas et vérifier que bookSeat retourne false. 2
- Créer une section avec un siège disponible et un numéro qui existe et vérifier que bookSeat retourne true et que hasAvailableSeat retourne false après. 2
- Créer une section ne comportant aucun siège de disponible et vérifier que bookSeat retourne false. 3

7. 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.

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("SWEST");
```

```
23                  res.createAirline("FRONTIER");    // Invalide
24                  res.createAirline("SWEST");       // Invalide

25
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
                    res.createFlight("DELTA", "ABC", "NCE", 2010, 9, 8, "567"); //
                      Invalide
                    res.createFlight("DELTA", "DEN", "DEF", 2010, 9, 8, "567"); //
                      Invalide
                    res.createFlight("ABCDE", "DEN", "DEH", 2009, 8, 9, "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  }
```