Examen de Conception/Programmation OO SUPAERO 2A

4 mars 2003

Cet examen est composé de deux parties indépendantes. Nous vous conseillons d'y consacrer la même durée. Tous les documents sont autorisés.

Exercice 1

1.1 Présentation du problème

Dans cet exercice, nous nous intéressons à la modélisation d'un système ayant pour objectif la commande des gouvernes d'un avion en fonction de l'état courant de l'avion et des ordres du pilote et du copilote.

Nous considérons que l'avion possède un certain nombre de surfaces (gouvernes) qui sont contrôlées par le système de commandes de vol. Ces surfaces peuvent être les suivantes (nous prenons pour exemple un avion de type A320) :

- deux gouvernes de profondeur ;
- une gouverne de direction ;
- deux THS (*trimmable horizontal stabilizer*) qui sont des plans horizontaux réglables ;
- quatre volets ;
- plus de deux becs de bord d'attaque ;
- deux ailerons.

Le système de commande de vol est composé quant à lui des éléments suivants :

- un ADIRS (*Air Data and Inertial Reference System*) qui calcule les données décrivant l'état de l'avion. Ces données sont calculées à partir de capteurs situés sur l'avion et qui peuvent être soit des capteurs de pression, soit des gyroscopes permettant de calculer des accélérations angulaires ;
- un FMS (*Flight Management System*) qui élabore les consignes de vol à destination du pilote automatique ;
- un PA (pilote automatique) qui calcule les ordres à destination des gouvernes de l'avion en fonction des données fournies par le FMS ;
- un EFCS (*Electrical Flight Control System*) qui calcule les angles à appliquer aux gouvernes. L'EFCS est associé à un ADIRS pour pouvoir connaître l'état de l'avion. En réalité, il n'existe pas d'EFCS « général », mais deux EFCS spécialisés :
  - un EFCS pour le mode automatique, qui calcule les angles en fonction des ordres du PA ;
  - un EFCS pour le mode manuel, qui calcule les angles à partir des ordres de l'équipage de l'avion.

  Les interactions entre ces différents éléments peuvent être résumées sur le schéma suivant (une flèche représente un « flux de données ») présenté sur la figure 1.

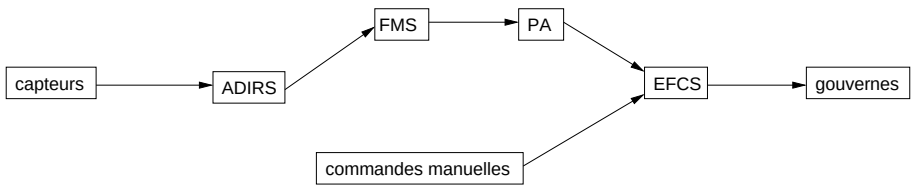

FIG. 1 – Interaction entre les différents composants du système de commande de gouvernes

1.2 Questions

1. proposer un diagramme *UML* de conception préliminaire (analyse) de l'ensemble des éléments décrits dans l'énoncé présentant les classes, les relations entre les classes et les éventuelles multiplicités (ou cardinalités).

   Vous pourrez justifier par écrit les relations utilisées et modifier de façon mineure l'énoncé si celui-ci vous paraît ambigü ;

2. proposer un diagramme de conception détaillée (attributs et opérations) de la classe *EFCS*[1]. Vous vous limiterez à la construction d'un petit nombre d'opérations sur cette classe ;

3. à partir du schéma présenté sur la figure 1, représenter sous forme d'un diagramme de séquence le scénario correspondant au changement de la valeur fournie par un des gyroscopes de l'avion. Pour cela, on supposera que l'avion est en mode de pilotage automatique et que le gyroscope « connaît » l'ADIRS.

   Vous considérerez également que vous disposez dans chaque classe d'opérations au nom explicite (ex : `calculeValeur` etc).

4. nous nous intéressons maintenant au comportement dynamique du pilote automatique. Nous allons le représenter grâce à un diagramme d'états-transitions.

   Au démarrage de l'appareil, celui-ci est inactif. Lorsque le pilote passe en mode de pilotage automatique (représenté par un état extérieur `PilotageAuto`), l'appareil met deux unités de temps pour devenir actif. En mode actif, le pilote automatique envoie des consignes de pilotage toutes les deux unités de temps et recalcule les consignes à partir d'éléments extérieurs toutes les unités de temps. Ces deux activités se déroulent de façon concurrente. Si le FMS est inactif (symbolisé par un état extérieur `FMS.Inactif`), le pilote automatique redevient inactif.

   Modéliser le comportement du pilote automatique grâce à un diagramme d'états-transitions.

Exercice 2

Remarque importante : dans cet exercice, vous allez devoir écrire du code JAVA. Il est évident que les petites erreurs de syntaxe ne seront pas pénalisantes (qui peut se vanter d'écrire directement du code correct à chaque fois ?). De même, vous n'écrirez la documentation Javadoc d'une méthode ou d'une classe que si vous le jugez nécessaire (préciser la signification d'un paramètre ou du retour d'une méthode par exemple).

2.1 Présentation du problème

Lorsque l'on dispose d'un agrégat d'objets (comme par exemple une liste, un ensemble ou un multi-ensemble), il est utile de disposer d'un mécanisme d'accès à ses éléments (pour les afficher par exemple). Nous allons mettre en œuvre le mécanisme d'*itérateur*. Un itérateur permet de réaliser un parcours de tous les éléments de l'agrégat.

Des exemple d'agrégats et les itérateurs associés sont présentés sur la figure 2.

Un itérateur sur un agrégat permet donc d'accéder aux éléments de l'agrégat suivant un parcours déterminé en protégeant la structure interne de l'agrégat. On peut ainsi disposer d'un mécanisme commun à tous les types d'agrégats et à tous les parcours possibles sur un même type d'agrégat.

[1] en particulier, ce diagramme devra faire apparaître l'implantation des relations

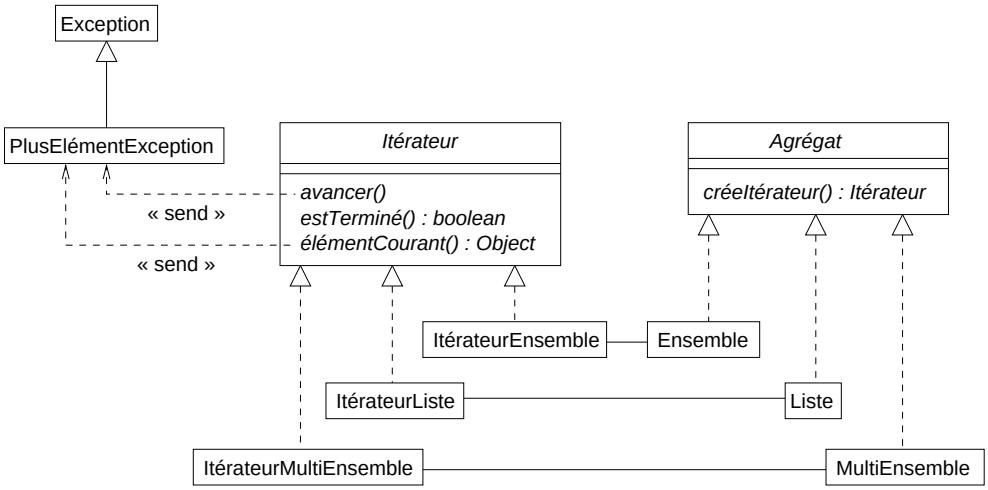

FIG. 2 – Diagramme de classe présentant les interfaces *Iterateur* et *Agregat* et quelques agrégats et leurs itérateurs associés

2.2 Questions

1. le diagramme *UML* de la figure 2 présente deux interfaces spécifiant les services rendus par deux catégories d'objets :
   - les itérateurs, qui fournissent les services suivants :
     - `avancer()` qui positionne l'élément courant sur l'élément suivant de l'agrégat. Cette méthode lève une exception de type `PlusElémentException` s'il n'y a plus d'éléments dans le parcours ;
     - `estTerminé()` qui renvoie un booléen qui est vrai si on a terminé le parcours de l'agrégat. Cela signifie que `estTerminé()` sera vraie ssi on est au delà du dernier élément de l'agrégat ;
     - `élémentCourant()` qui renvoie un `Object` qui est l'élément courant dans l'agrégat. Cette méthode lève une exception de type `PlusElémentException` s'il n'y a plus d'éléments dans le parcours.
   - les agrégats, qui fournissent un seul service, `créeItérateur()` qui renvoie un `Itérateur` sur l'agrégat.

   Que pensez-vous du choix des interfaces par rapport à des classes abstraites ?
2. écrire le code JAVA correspondant aux deux interfaces ;
3. écrire une classe `Utilitaire` possédant une méthode `nbEléments` prenant un `Agrégat` en paramètre et renvoyant le nombre d'éléments de l'agrégat ;
4. on va maintenant définir un itérateur particulier sur un agrégat particulier. Pour cela, on dispose d'une classe `Tableau` réalisant l'interface `Agrégat` (cf. figure 3). Cette classe représente un agrégat d'objets sous forme d'un tableau dynamique. Ce tableau est donc extensible : on lui donne une taille initiale, mais on pourra introduire plus d'éléments dans le tableau que la valeur de la taille initiale. Les méthodes de `Tableau` sont les suivantes :
   - `ajouter(o : Object)` permet d'ajouter un élément à la fin du tableau et augmente sa taille si nécessaire ;
   - `getTaille()` renvoie la taille effective du tableau ;
   - `retirer(position : int)` enlève l'élément situé à la position `position` du tableau. `position` doit être compris entre 0 et (`getTaille() - 1`). Cette méthode renvoie une `ArgumentInvalideException` si la position n'est pas correcte ;

– `premierElément()` renvoie le premier élément du tableau. Cette méthode renvoie une `TableauVideException` si le tableau est vide ;
– `dernierElément()` renvoie le dernier élément du tableau. Cette méthode renvoie une `TableauVideException` si le tableau est vide ;
– `getElément(position : int)` renvoie l'élément situé à la position `position`.
`position` doit être compris entre 0 et (`getTaille() - 1`). Cette méthode renvoie une `ArgumentInvalideException` si la position n'est pas correcte ;
– `créeItérateur()` retourne un itérateur sur le tableau.
La note sur le diagramme précise le corps de la méthode `créeIterateur` de `Tableau`.

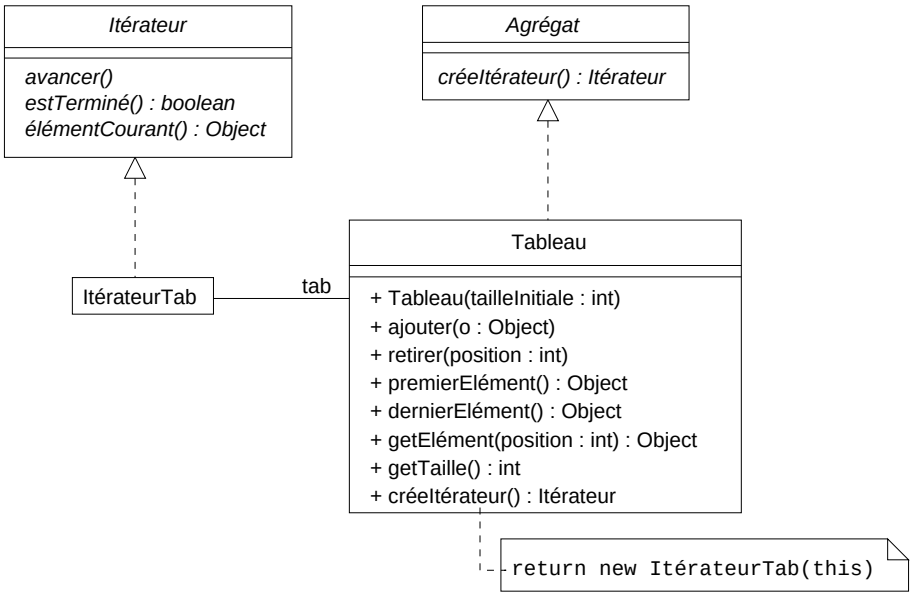

FIG. 3 – Diagramme de classes présentant la classe `ItérateurTab`

(a) quels sont les attributs de `ItérateurTab` ? Doit-on stocker l'élément courant du tableau ?
(b) quel(s) va(ont) être le(s) constructeur(s) de la classe `ItérateurTab` ? Doit-on garder un constructeur par défaut ?
(c) écrire le code JAVA de la classe `ItérateurTab`. Les méthodes de `ItérateurTab` ne devront pas propager les éventuelles exceptions lancées dans les méthodes de `Tab`.

5. écrire une classe de test `TestItérateurTab` dont la méthode principale :
   – crée une instance de `Tableau` de taille 2 ;
   – ajoute trois instances de `Integer` à ce tableau ;
   – crée un itérateur sur le tableau ;
   – utilise cet itérateur dans une boucle (on suppose que l'on ne connaît pas la longueur effective du tableau) et appelle la méthode `intValue()` de la classe `Integer` pour afficher les objets du tableau. La méthode `intValue()` renvoie la valeur de l'entier (donc un `int`) associé à l'objet de type `Integer`.
6. on désire spécialiser la classe `ItérateurTab` en une classe `ItérateurTabFiltre`. La classe `ItérateurTabFiltre` permet d'itérer sur les seuls éléments de type `Filtrable` dans un tableau pouvant comprendre des éléments d'un autre type que `Filtrable`. Le type `Filtrable` a été défini par ailleurs (dans une classe ou une interface).

Écrire en JAVA la classe `ItérateurTabFiltre` en utilisant les méthodes de la classe `ItérateurTab`.
Vous n'écrirez pas la méthode `estTerminé()`.