Conception/Programmation OO - Corrigé 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.

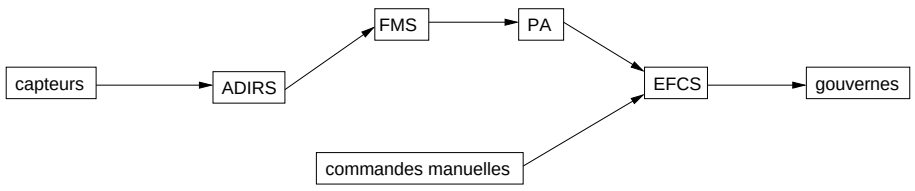

FIG. 1 – Interaction entre les différents composants du système de commande de gouvernes

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) ;

   Un diagramme de conception possible est représenté sur la figure 2.

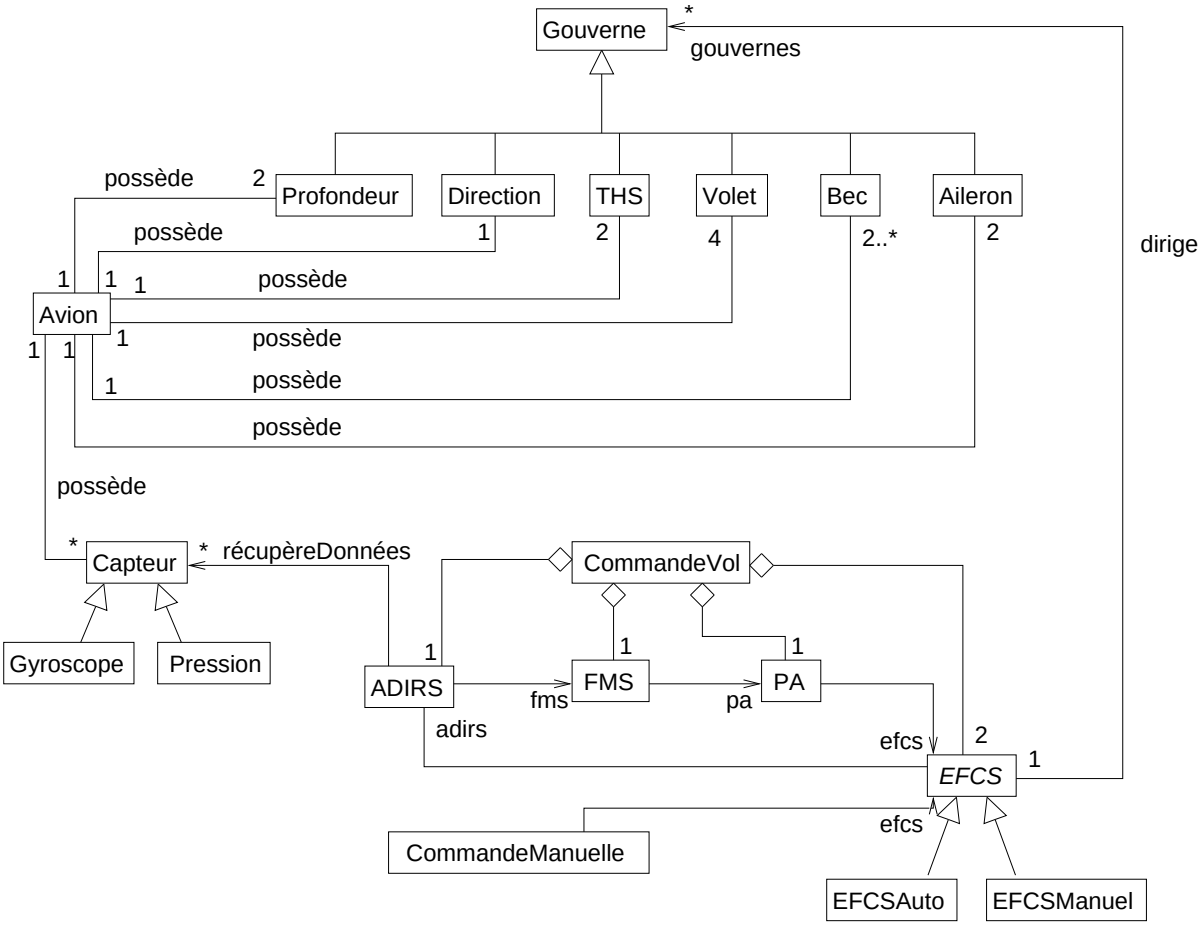

FIG. 2 – Diagramme de conception préliminaire

Quelques remarques :

- les relations de spécialisation se trouvaient aisément dans le texte ;
- l'association qui lie l'avion aux gouvernes se nomme **possède**. On aurait également pu utiliser une relation d'agrégation ou de composition ;
- la relation liant le système de commande de vol à ses composants est une relation d'agrégation. On considère donc ici que les composants ont une durée de vie indépendante du système de commande ;
- les associations unidirectionnelles (représentées par des associations avec une flèche indiquant dans quel sens l'association se fait) se déduisent du diagramme présentant les interactions entre les différents composants ;

– la classe *EFCS* est une classe abstraite, car on ne fait effectivement les calculs que pour un mode particulier.

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 ;

   Un diagramme de conception détaillée de la classe *EFCS* est représenté sur la figure 3.

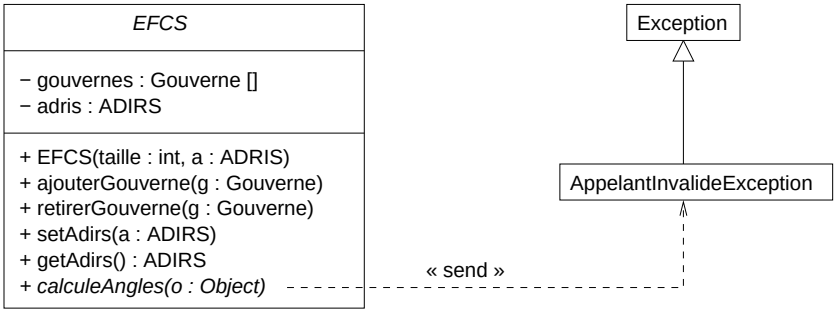

FIG. 3 – Diagramme de conception détaillée de la classe *EFCS*

   Il n'y a que deux attributs dans la classe *EFCS*, car les associations entre *EFCS* et *PA* ou *Commande-Manuelle* sont unidirectionnelles. Les méthodes `ajouterGouverne` et `retirerGouverne` permettent d'ajouter et de retirer une gouverne particulière à la liste de gouvernes contrôlées par l'EFCS. Les méthodes `getAdirs` et `setAdirs` permettent de recupérer l'ADIRS associé à l'EFCS et de positionner l'EFCS sur un ADIRS particulier.

   La seule difficulté provenait de la méthode appelée ici `calculeAngles` qui permet de calculer les angles des gouvernes. Cette méthode est abstraite, car selon que l'on soit en mode automatique ou manuel, les calculs ne seront pas effectués de la même façon. Il faut par contre savoir si cette méthode a été appelée par le PA ou par une commande manuelle. Pour cela, on passe en paramètre de la méthode un objet et on testera dans les classes spécialisant *EFCS* la nature de cet objet (PA ou commande manuelle).

   On a également précisé sur le diagramme que cette méthode pouvait lever une exception de type `AppelantInvalideException` si l'appelant n'est pas le PA ou une commande manuelle.

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).

   Un diagramme de séquence possible est présenté sur la figure 4.

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.

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

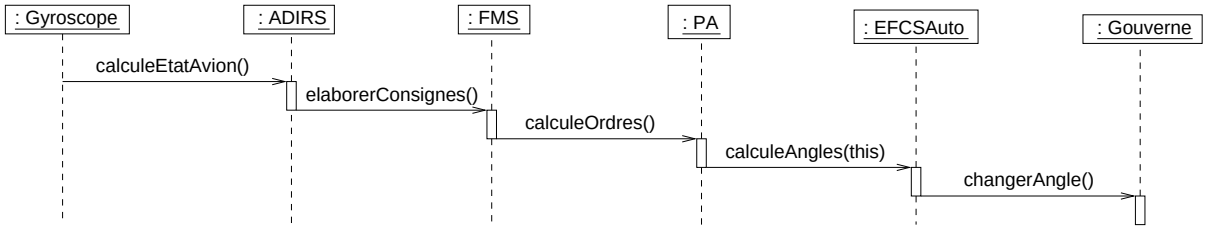

FIG. 4 – Diagramme de séquence

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.

Le diagramme d'états-transitions représentant le comportement du PA est présenté sur la figure 5.

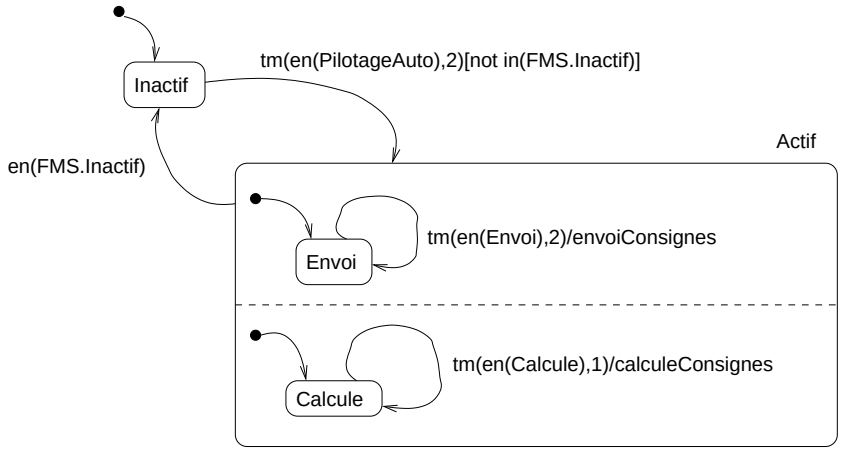

FIG. 5 – Diagramme d'états-transitions représentant le comportement du PA

Exercice 2

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 6.

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.

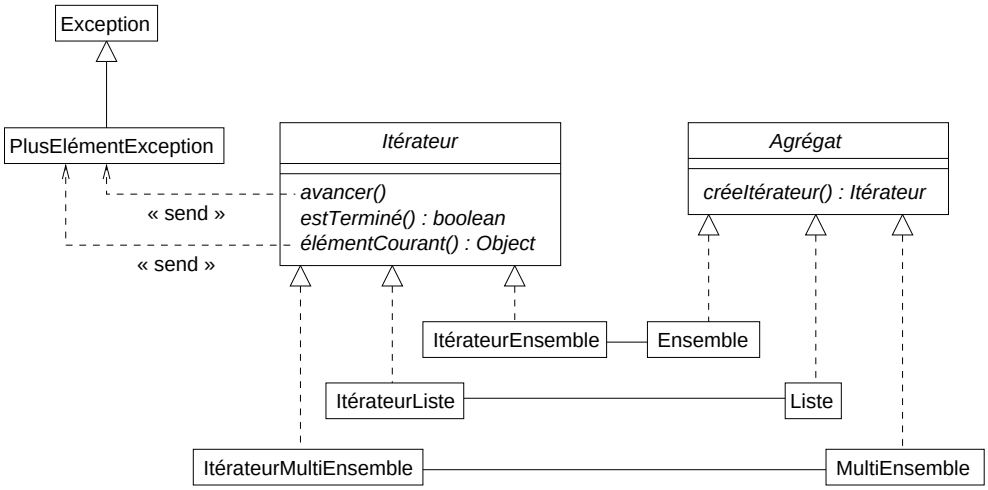

FIG. 6 – Diagramme de classe présentant les interfaces *Iterateur* et *Agregat* et quelques agrégats et leurs itérateurs associés

2.2 Questions

1. le diagramme *UML* de la figure 6 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 ?

   La question posée ici est classique : il s'agit de « débattre » des avantages et inconvénients respectifs des classes abstraites et des interfaces.

   Dans notre cas, toutes les méthodes de `Itérateur` et de `Agrégat` sont abstraites (on ne peut pas écrire de code pour ces méthodes). De plus, les itérateurs et les agrégats ne possèdent pas d'attributs. Il vaut mieux donc utiliser deux interfaces pour conserver la possibilité d'utiliser une relation d'héritage pour les classes réalisant `Itérateur` ou `Agrégat`.

2. écrire le code JAVA correspondant aux deux interfaces ;

   Voila le code des deux interfaces :

```
/**
 * <code>Itérateur</code> définit une interface pour tous les
 * itérateurs sur des agrégats.
 *
```

```
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public interface Itérateur {

   /**
    * <code>avancer</code> positionne l'élément courant sur
    * l'élément suivant dans l'agrégat.
    *
    * @exception PlusElémentException si on a fini de parcourir l'
    *         agrégat
    */
   public void avancer() throws PlusElémentException;

   /**
    * <code>estTerminé</code> permet de savoir si on est à la fin de
    * l'agrégat.
    *
    * @return un <code>boolean</code> qui est <code>true</code> si
    *       on est à la fin de la liste.
    */
   public boolean estTerminé();

   /**
    * <code>élémentCourant</code> renvoie l'élément courant.
    *
    * @return un <code>Object</code> qui est l'élément courant.
    * @exception PlusElémentException si on a fini de parcourir l'
    *         agrégat
    */
   public Object élémentCourant() throws PlusElémentException;
}
```

```
/**
 * <code>Agrégat</code> représente un agrégat d'objets pouvant
 * être manipulé par l'intermédiaire d'un itérateur.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public interface Agrégat {

   /**
    * <code>créeItérateur</code> crée un itérateur pour l'agrégat.
    *
    * @return une instance d'<code>Itérateur</code> spécifique pour
    *       l'agrégat.
    */
   public Itérateur créeItérateur();
```

```
}
```

Vous remarquerez que l'on doit bien spécifier dans la signature des méthodes de `Itérateur` quelles sont les exceptions qui peuvent être levées à l'intérieur de ces méthodes.

3. écrire une classe `Utilitaire` possédant une méthode `nbEléments` prenant un `Agrégat` en paramètre et renvoyant le nombre d'éléments de l'agrégat ;

```
/**
 * <code>Utilitaire</code> est une classe manipulant des agrégats.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class Utilitaire {

   /**
    * <code>nbEléments</code> permet de déterminer le nombre
    * d'éléments d'un agrégat.
    *
    * @param ag un <code>Agrégat</code>
    * @return le nombre d'éléments de ag
    */
   public static int nbEléments(Agrégat ag) {

      Itérateur i = ag.créeItérateur();
      int nb = 0;

      try {
         while (!(i.estTerminé())) {
            nb++;
            i.avancer();
         } // end of while (!(i.estTerminé()))
      }
      catch(PlusElémentException e) {
         System.out.println("Accès au delà du dernier élément !");
      }
      return nb;
   }
}
```

Le but de cette question était de vous faire manipuler la notion d'itérateur sans avoir d'itérateur particulier. La méthode `nbEléments` est statique, elle ne manipule pas d'instances de la classe `Utilitaire`.

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 7). 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`.

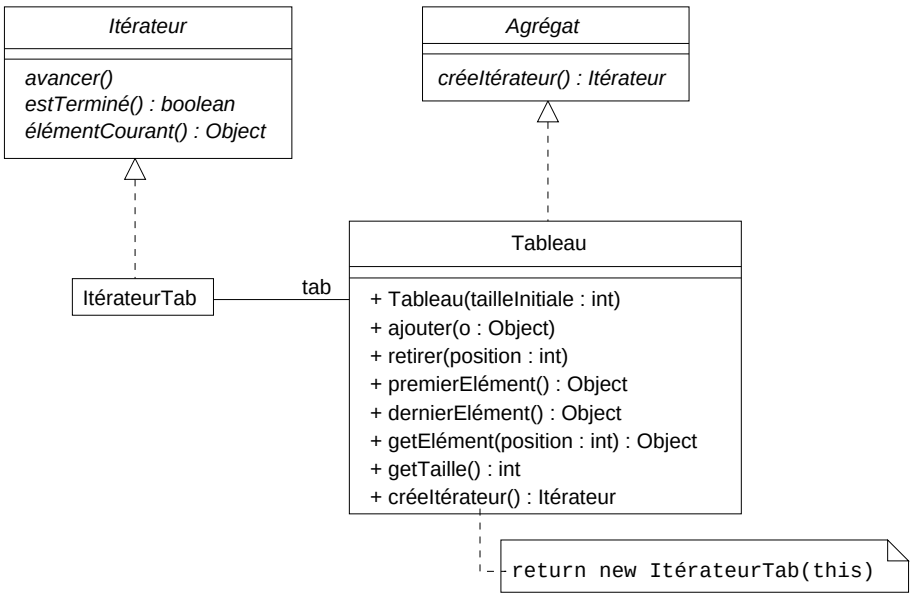

FIG. 7 – Diagramme de classes présentant la classe `ItérateurTab`

(a) quels sont les attributs de `ItérateurTab` ? Doit-on stocker l'élément courant du tableau ?

Le diagramme de la figure 7 nous indique qu'il existe une association entre `ItérateurTab` et `Tableau`. Cette association va se traduire en JAVA par l'introduction d'un attribut de type `Tableau` dans `ItérateurTab`. D'après le nom de rôle donné dans l'association, on peut appeler cet attribut `tab`.

L'itérateur va servir d'intermédiaire entre la liste et l'utilisateur de la liste. Il va en particulier fournir une méthode `élémentCourant` qui renvoie l'élément courant du tableau.

Deux solutions s'offrent à nous :
– soit on stocke explicitement l'élément courant dans `ItérateurTab` ;
– soit on stocke l'indice de l'élément courant dans `ItérateurTab`. C'est cette solution qui est retenue pour des raisons de facilité d'utilisation (manipulation d'un entier, comparaison avec la taille du tableau etc).

(b) quel(s) va(ont) être le(s) constructeur(s) de la classe `ItérateurTab` ? Doit-on garder un constructeur par défaut ?

Le corps de la méthode `créeItérateur` de `Tableau` nous précise qu'il existe un constructeur de `ItérateurTab` qui prend un objet de type `Tableau` en paramètre. On ne doit pas écrire de constructeur par défaut (i.e. sans paramètres), car on a absolument besoin de connaître le tableau associé à l'itérateur pour le construire.

(c) écrire le code JAVA de la classe `ItérateurTableau`. Les méthodes de `ItérateurTableau` ne devront pas propager les éventuelles exceptions lancées dans les méthodes de `Tableau`.

```
/**
 * <code>ItérateurTab</code> est un itérateur pour un tableau.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class ItérateurTab implements Itérateur {

   private Tableau tab;

   private int elementCourant;

   /**
    * Crée une instance de <code>ItérateurTab</code>.
    *
    * @param l la <code>Liste</code> associée à l'itérateur
    */
   public ItérateurTab(Tableau t) {

      tab = t;
      elementCourant = 1;
   }

   /**
    * <code>avancer</code> renvoie l'élément suivant dans le tableau.
    *
    * @exception PlusElementException si on a fini de parcourir le
    *         tableau
    */
   public void avancer() throws PlusElémentException {

      if (estTerminé()) {
         throw new PlusElémentException();
      } // end of if (elementCourant == tab.getTaille())
      elementCourant++;
   }

   /**
    * <code>estTerminé</code> permet de savoir si on a fini de parcourir
    * le tableau.
```

```
    *
    * @return <code>boolean</code> qui est <code>true</code> si on a
    *       fini de parcourir le tableau
    */
   public boolean estTerminé() {

      return (elementCourant > tab.getTaille());
   }

   /**
    * <code>élémentCourant</code> permet de récupérer l'élément
    * courant du parcours.
    *
    * @return un <code>Object</code> qui est l'élément en cours
    * @exception PlusElémentException si on a fini de parcourir le
    *         tableau
    */
   public Object élémentCourant() throws PlusElémentException {

      Object o = null;

      if (estTerminé()) {
         throw new PlusElémentException();
      } // end of if (estTerminé())

      try {
         o = tab.getElément(elementCourant - 1);
      }
      catch (ArgumentInvalideException e) {
      }
      return o;
   }
}
```

Quelques remarques :

- le constructeur ne posait absolument pas de problème dès lors que l'on avait identifié correctement les attributs de la classe ;
- la méthode « centrale » est la méthode `estTerminé`. Il fallait bien comprendre que cette méthode renvoie `true` si on a dépassé le dernier élément du tableau (en particulier, on peut appeler `avancer()` alors que l'on est déjà sur le dernier élément du tableau) ;
- la méthode `avancer` peut lancer une `PlusElementException` si on a fini de parcourir la liste. Il faut donc faire un test sur `elementCourant` avant d'incrémenter la valeur de `elementCourant`. Pour cela, on utilise la méthode `estTerminé` ;
- le problème principal qui se posait était d'utiliser la méthode `getElément` de `Tableau` qui pouvait renvoyer une `ArgumentValideException`. On utilise cette méthode dans la méthode `élémentCourant()` de `ItérateurTab`. Comme le sujet précisait que l'on ne propageait pas les exceptions de `Tab`, il fallait traiter localement cette exception dans un bloc `try/catch`. Remarquez que l'exception si l'exception est levée (parce que l'utilisateur n'a pas vérifié que le tableau n'avait pas été entierement parcouru), la méthode renvoie l'objet `null`.
  Enfin, on utilise un objet intermédiaire `o`, car la méthode doit obligatoirement se terminer

par `return`.

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`.

   Voici la classe `TestItérateurTab` :

```
/**
 * <code>TestItérateurTab</code> est une classe de test pour
 * l'itérateur de tableau.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class TestItérateurTab {

   public static void main (String[] args) {

      Tableau t = new Tableau(2);
      t.ajouter(new Integer(2));
      t.ajouter(new Integer(4));
      t.ajouter(new Integer(6));

      Itérateur i = t.créeItérateur();

      try {
         while (!(i.estTerminé())) {
            System.out.println(((Integer)i.élémentCourant()).intValue());
            i.avancer();
         }
      }
      catch (PlusElémentException e) {
         System.out.println("Il n'y a plus d'éléments dans le tableau !");
      }
   } // end of main ()

}
```

   Là encore, remarquez que l'utilisation des méthodes `avancer` et `élémentCourant` de `ItérateurTab` nous obligeait à utiliser un bloc `try/catch`.

   Petit piège (mais qui n'a pas de conséquences catastrophiques sur la note...), on récupère un `Object` en utilisant `élémentCourant()`. Il faut donc transtyper le résultat pour pouvoir appeler `intValue` dessus.

   L'utilisation d'une boucle `while` est beaucoup plus simple qu'un `for` qui pose énormement de problèmes (initialisation, passage sur le dernier élément etc).

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()`.*

   Voici le code de la classe `ItérateurTabFiltre` :

```
/**
 * <code>ItérateurTabFiltre</code> est un itérateur pour tableau ne
 * prenant en compte que les éléments de type Filtrable.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class ItérateurTabFiltre extends ItérateurTab {
   public ItérateurTabFiltre(Tableau t) {
      super(t);
   }

   /**
    * <code>avancer</code> positionne l'itérateur sur le prochain
    * élément du tableau qui est de type Filtrable.
    *
    * @exception PlusElémentException s'il n'y a plus d'éléments de
    *         type Filtrable dans le tableau
    */
   public void avancer() throws PlusElémentException {
         do {
            super.avancer();
         } while (!(super.élémentCourant() instanceof Filtrable));
   }

   /**
    * <code>estTerminé</code> permet de vérifier s'il existe encore
    * des éléments de type Filtrable dans le tableau.
    *
    * @return un <code>boolean</code> qui est vrai s'il n'y a plus
    *       d'éléments de type Filtrable
    */
   public boolean estTerminé() {
      ItérateurTabFiltre copie = null;

      try {
         copie = (ItérateurTabFiltre)this.clone();
      }
      catch(CloneNotSupportedException e) {
      }
```

```
      try {
         copie.avancer();
      }
      catch(PlusElémentException e) {
         return true;
      }
      return false;
   }

   public Object élémentCourant() {
      return(super.élémentCourant());
   }
}
```

La classe était simple à écrire si on utilisait correctement les méthodes de la classe `ItérateurTab`. La solution proposée pour l'écriture de la méthode `estTerminé` n'est peut être pas la plus élégante que l'on puisse faire, mais elle ne vous était pas demandée...

Remarquez que l'on est obligé d'utiliser une copie du `ItérateurTabFiltre` courant pour vérifier s'il existe encore des éléments filtrables dans la liste (car on utilise la méthode `avancer` qui positionne à chaque fois l'indice sur le nouvel élément).