IN201 : Corrigé de l'examen

SUPAERO 2A

Christophe Garion <garion@supaero.fr>

19 mars 2007

Cet examen est composé de trois parties indépendantes. Vous avez 2h30 pour le faire. Les documents autorisés sont les photocopies distribuées en cours et les notes manuscrites que vous avez prises lors du cours. Il sera tenu compte de la rédaction.

Remarque importante : dans tous les exercices, il sera tenu compte de la syntaxe UML lors de l'écriture de diagrammes.

1 Modélisation objet d'un portail pour SUPAERO

On désire réaliser un serveur interactif pour SUPAERO. Les fonctions principales requises pour ce serveur sont la gestion des élèves et de leurs notes, la gestion de vacataires et de leur paye etc. Pour l'instant, on veut réaliser un premier prototype. On dispose d'un certain nombre d'informations sur le fonctionnement de SUPAERO. Ces informations vont permettre de modéliser le *domaine* de l'application.

SUPAERO est composée d'un certain nombres de personnels et d'étudiants. Tout étudiant est caractérisé par son nom, son prénom, sa date de naissance, sa nationalité et un numéro. Les étudiants ne peuvent être que des élèves ingénieurs, des étudiants de M2R, des doctorants ou des étudiants ERASMUS.

Le personnel est partitionné en trois niveaux : personnel de niveau I, de niveau II et de niveau III. Parmi les personnels de niveau III, on trouve les Professeurs. Ceux-ci appartiennent à un département d'enseignement.

Les modules sont composés d'un certain nombre de séances. Chaque séance se déroule à une date donnée et dans une salle donnée. Les séances sont classées en différents types : Cours, PC, BE, TP, Examen. Les modules sont regroupés dans des parcours pédagogiques qui peuvent être des majeures, des programmes de tronc commun, des options et des approfondissements. Les modules sont caractérisés par un nom, un synopsis, un coefficient et un volume horaire.

Chaque module est associé à un professeur que l'on appelle correspondant. Les séances des modules sont assurées par des intervenants qui sont soit des vacataires extérieurs, identifiés par un matricule, soit des professeurs. Une note relie un étudiant à un cours qu'il/elle a suivi.

Du point de vue de l'application, toutes ces informations seront contenues dans un serveur de stockage. Les utilisateurs devront s'identifier grâce à un *login* et un mot de passe également stockés dans ce serveur. L'accès à l'application se fait via un portail qui permet aux utilisateurs de s'authentifier et d'émettre des requêtes. Les utilisateurs peuvent être des inspecteurs des études, des professeurs ou des élèves. Un scénario d'entrée de notes pour un élève par un inspecteur des études est le suivant :

- l'inspecteur se connecte au portail et celui-ci lui demande son *login* et son mot de passe ;
- l'inspecteur envoie son *login* et son mot de passe ;
- le portail vérifie l'identité grâce au serveur de stockage ;
- le portail ouvre une session pour l'inspecteur ;
- celui-ci demande une visualisation des notes pour l'élève X ;
- le portail, après interrogation du serveur de stockage lui renvoie les notes ;
- l'inspecteur entre la note de 15 pour la matière XX200 pour l'élève X ;
- le portail appelle la méthode correspondante du serveur de stockage pour modifier les notes de l'élève.

1. proposer un diagramme de séquence représentant le scénario précédent. Pour les méthodes pour lesquelles cela est utile, représenter les paramètres et les valeurs de retour.

   Un diagramme de séquence est proposé sur la figure 1. Quelques remarques :

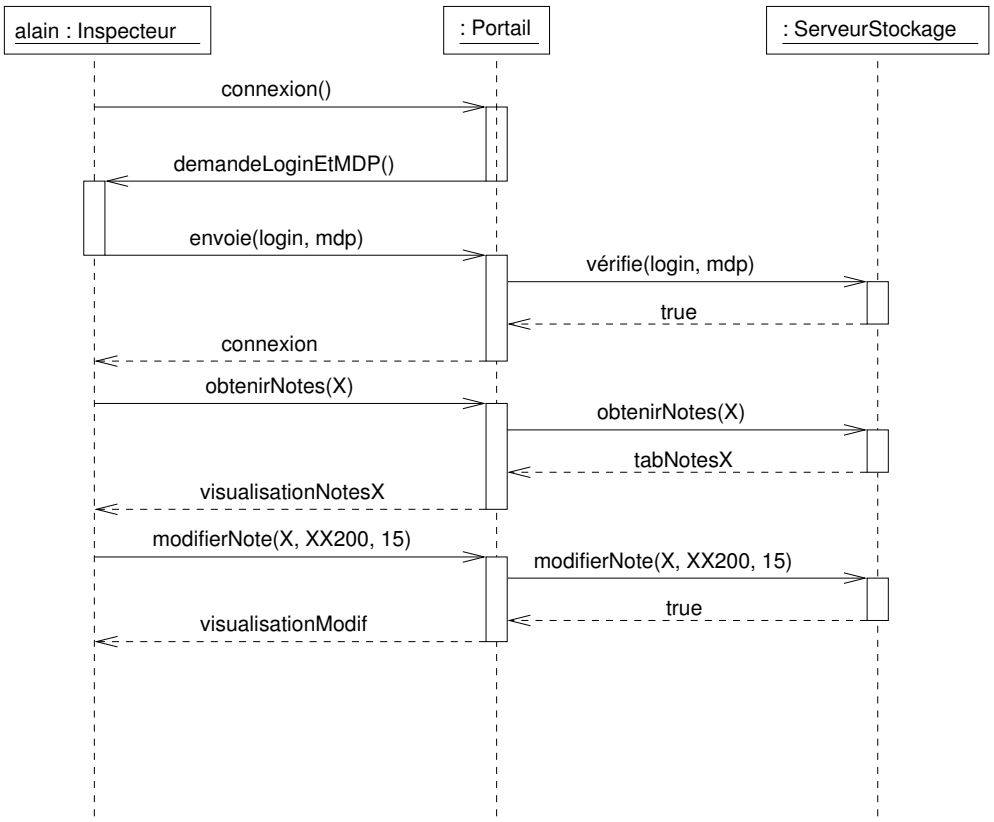

FIG. 1 – Diagramme de séquence modélisant le scénario

- j'ai choisi des noms de méthodes explicites. Il ne fallait pas oublier les paramètres des méthodes lorsque l'on en avait besoin (exemple de `envoie`) ;
- les retours des méthodes vers l'inspecteur des études sont plutôt informelles : les objets `connexion`, `visualisationNotesX` et `visualisationModif` sont là pour représenter les interactions entre l'inspecteur et le portail.

2. proposer un diagramme de classes d'analyse représentant le domaine. **On ne s'intéressera pas ici à la modélisation de l'application (serveur de stockage, *login* etc).** On fera apparaître les classes, les relations entre les classes, les noms de rôles et les multiplicités des associations et on justifiera l'utilisation de classes abstraites et d'interfaces. On pourra faire apparaître quelques attributs si nécessaire.

   Un diagramme de classes est proposé sur la figure 2. Quelques remarques :
   - certaines caractéristiques des classes sont représentées sous la forme d'attributs (quand elles ont des types simples, comme `Date` ou `String`), mais on pouvait bien évidemment les représenter en utilisant des associations avec d'autres classes.
   - SUPAERO est un établissement et sera donc une instance de la classe `Etablissement`. Ceci permet de dire par exemple qu'un étudiant appartient à plusieurs établissements (3A + M2R par exemple) ou qu'un personnel est à mi-temps dans deux établissements.
   - les associations entre la classe `E1tablissement` et les classes `Etudiant` et `Personnel` sont des agrégations : les durées de vie des objets de type `Etudiant` et `Personnel` ne sont pas liées à celle de l'objet de type `Etablissement`. Je considère également qu'un établissement possède au moins

un étudiant et un personnel.
- la classe `Etudiant` est abstraite : un étudiant ne peut être qu'un élève ingénieur, un élève de M2R, un doctorant ou un étudiant ERASMUS. Cette classification est représentée par des sous-classes de `Etudiant`. `Etudiant` est abstraite, car tous les étudiants ont des caractéristiques communes, représentées ici par des attributs.
- de même, la classe `Personnel` est abstraite. Elle possède trois sous-classes pour chacun des niveaux. Le lien avec la classe `Professeur` et le lien entre `Professeur` et `DptEns` était évident (on pourra remarquer qu'un département d'enseignement n'existe pas sans professeurs).
- `Séance` est une classe abstraite, sous-typée avec cinq classes. Une séance se déroule dans une seule salle a priori. Un module est composée d'au moins une séance et si on « détruit » le module, on détruit la séance associée.
- j'ai choisi de faire une interface `Intervenant` réalisée par les classes `Professeur` et `Vacataire`. Une classe abstraite était également possible (l'héritage multiple est autorisé en UML, donc il n'y pas de problème avec le fait que `Professeur` hérite déjà de `NivIII`).
- enfin, la classe `Note` permet de relier `Etudiant` et `Module` (on aurait également pu utiliser une *classe-association*, mais cela n'a pas été abordé en cours).

2 Création d'une collection typée avec Java

Remarque importante : dans tout cet exercice, on considère que l'on n'a pas à disposition le mécanisme de types génériques vus en cours.

Nous avons vu en cours que les collections en `Java` permettaient de manipuler des ensembles d'objets via différentes implantations : listes, listes doublement chaînées, ensembles, arbres etc. Ces différentes implantations sont manipulées via des classes et des interfaces situées dans le paquetage `java.util`. Les principales classes et interfaces disponibles sont représentées sur la figure 3.

Pour utiliser une collection, on utilise les méthodes `add(Object e)` et `remove(Object e)` qui permettent d'ajouter et de retirer une instance de `Object` de la collection.

Pour pouvoir parcourir la collection, on récupère un itérateur sur cette collection via la méthode `iterator()` et on utilise les méthodes de l'itérateur, i.e. :
- `hasNext()` qui renvoie **`true`** si il reste des éléments à parcourir ;
- `next()` qui renvoie le prochain élément de l'itération avec le type `Object` ;
- `remove()` qui retire le dernier élément renvoyé de la collection.

1. écrire une méthode statique `afficherLongueur(java.util.Collection)` qui prend en paramètre une collection contenant des objets de type `String` et qui affiche leurs longueurs (on utilisera la méthode `length()` de `String`) ;

   Voici la méthode. Attention, ici on est suppose que la collection contient effectivement des objets de type `String` (on le précise d'ailleurs dans la documentation Javadoc) :

```
/**
 * <code>afficherLongueur</code> affiche la longueur des chaines
 *  de caracteres contenues dans la collection.
 *
 * @param c une instance de <code>java.util.Collection</code> ne
 *          contenant que des objets de type <code>String</code>
 */

public static void afficherLongueur(java.util.Collection c) {
    java.util.Iterator i = c.iterator();
    while (i.hasNext()) {
```

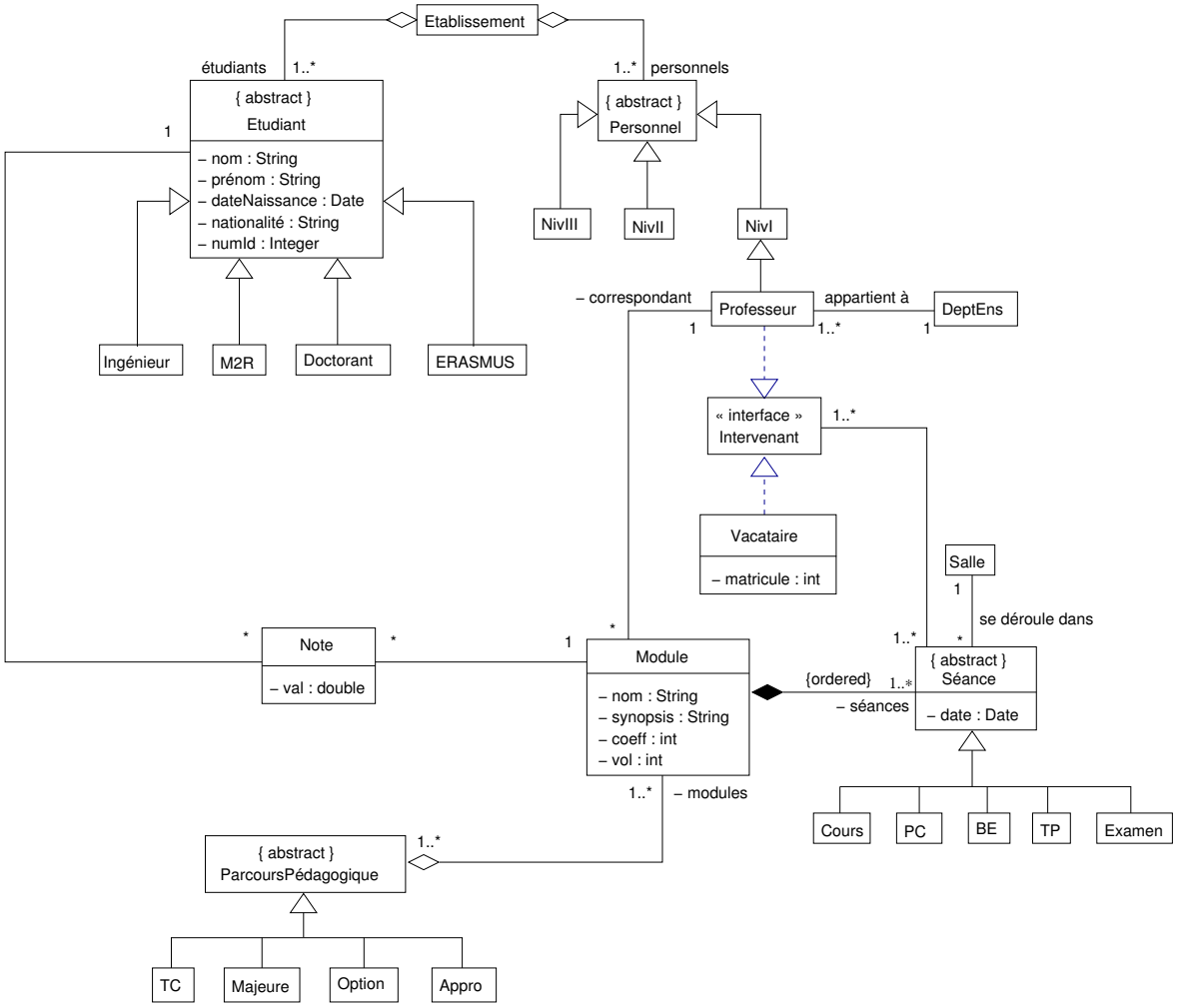

FIG. 2 – Diagramme de classes du domaine

```
            System.out.println("Longueur de la chaine " +
                               ((String) i.next()).length());
        }
    }
```

2. que se passe-t-il si un des objets contenu dans la collection n'est pas de type `String` ? Comment y remédier ?

   Si un des objets dans la collection n'est pas de type `String`, une exception de type `ClassCastException` est alors levée à l'exécution. Par contre, à la compilation, aucune erreur n'est détectée. Plusieurs solutions sont possibles pour éviter cela[1] :

   – la solution la plus « propre » est de spécifier par un *contrat* que la collection ne doit contenir que des objets de type `String`.

[1] Dans le corrigé de cette question, les noms des méthodes changent en fonction des solutions

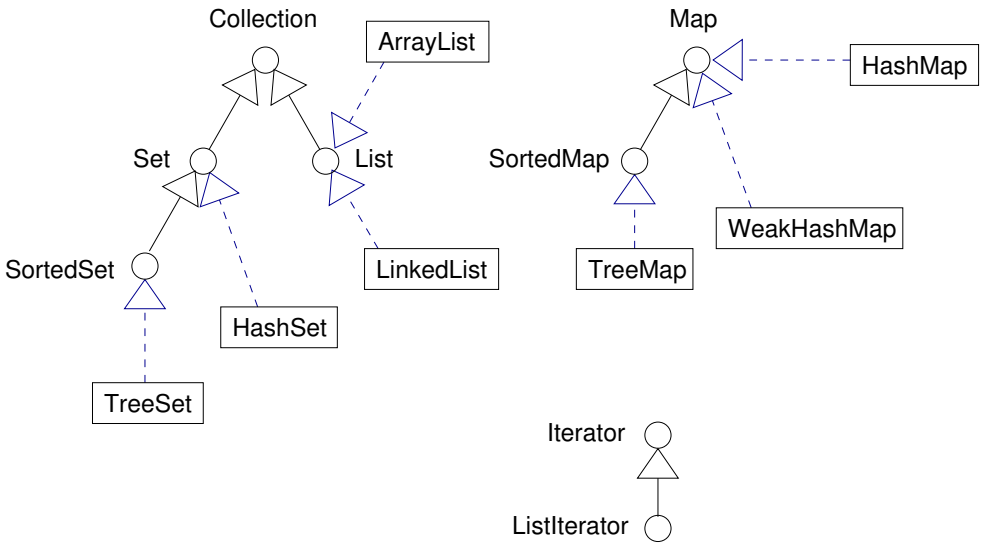

FIG. 3 – Diagramme de classes présentant les classes et interfaces principales du paquetage `java.util`

Cette approche est malheureusement très difficile à mettre en œuvre : il faut en effet pouvoir spécifier avec JML par exemple que chaque élément de la collection est une instance de `String`. Une solution serait la suivante :

```
/* @requires c != null &&
            (\forall int i; 0 <= i < c.toArray().length;
                    c.toArray[i] instanceof String);
*/
```

J'utilise alors la méthode `toArray` de `Collection` qui permet de « transformer » la collection en tableau d'`Object`. On voit bien que cette solution est extrêmement lourde à mettre en œuvre (en particulier, on appelle `toArray` un grand nombre de fois).

– une seconde solution est d'utiliser bien sûr l'opérateur **`instanceof`** pour vérifier que les éléments du tableau sont bien des instances de `String` :

```
public static void afficherLongueurVerif(java.util.Collection c) {
    java.util.Iterator i = c.iterator();
    while (i.hasNext()) {
        Object obj = i.next();
        if (obj instanceof String) {
            System.out.println("Longueur de la chaine " +
                               ((String) obj).length());
        }
    }
}
```

Cette fois, il n'y aura plus d'exception levée. Cette solution a un défaut important : l'utilisateur ne se rend pas compte que certains éléments de son tableau n'ont pas été pris en compte.

– une dernière solution est d'utiliser le mécanisme des exceptions. Il ne s'agit pas ici de traiter localement les exceptions pouvant être levées à cause du transtypage impossible, mais de les propager. On les déclarera donc dans la signature de la méthode : l'utilisateur de `afficherLongueurEx` de-

vra soit déclarer ces exceptions dans la signature de sa méthode, soit utiliser un bloc **try/catch** pour les traiter localement.
Un point de détail technique : on ne peut pas utiliser `ClassCastException` pour cela, car celle-ci hérite de `RuntimeException`. Or, on a vu en cours que les exceptions de ce type étaient hors-contrôle. Si on utilise `ClassCastException` et qu'on la propage, l'utilisateur de `afficherLongueurEx` *n'est pas obligé par le compilateur* de propager ou de traiter localement cette exception. Il faut donc créer sa propre exception (dans la solution proposée, elle s'appelle `PasUneChaineException`).
Voici une proposition de méthode :

```
/**
 * <code>afficherLongueurEx</code> affiche la longueur des chaines
 *  de caracteres contenues dans la collection.
 *
 * @param c une instance de <code>java.util.Collection</code> ne
 *          contenant que des objets de type <code>String</code>
 * @exception PasUneChaineException si un des elements de la
 *            collection n'est pas une instance de <code>String</code>
 */
public static void afficherLongueurEx(java.util.Collection c)
                    throws PasUneChaineException {
    java.util.Iterator i = c.iterator();
    while (i.hasNext()) {
        Object obj = i.next();
        if (! obj instanceof String) {
            throw new PasUneChaineException("Un element n'est pas une chaine");
        }
        System.out.println("Longueur de la chaine " +
                           ((String) obj).length());
    }
}
```

Supposons que l'on souhaite manipuler une collection ne contenant que des instances de la classe `String`. Les méthodes offertes par les collections, en particulier `add`, ne nous garantissent pas qu'un utilisateur ne puisse insérer que des objets de type `String` dans cette collection, car les collections manipulent des instances d'`Object` qui est la classe la plus « générale » de JAVA

On souhaite pallier ce problème en ajoutant à une collection la donnée d'un type qui permettrait d'imposer à tous les éléments de la collection d'être d'un même type.

Pour simplifier le problème, nous nous intéresserons ici à une collection particulière, `java.util.ArrayList`, et nous chercherons donc à écrire une classe `java.util.ArrayListTypee` qui est une `ArrayList` à laquelle on a attaché un type caractérisant ses éléments.

3. deux solutions s'offrent à nous : soit on *délègue* les services fournis par `ArrayListTypee` à `ArrayList` (association entre les deux classes), soit `ArrayListTypee` étend `ArrayList`. Donner les avantages et inconvénients de chaque solution.

   Les deux solutions sont évidemment possibles et sont « duales ». Si on choisit de déléguer les services fournis par `ArrayListTypee` à `ArrayList`, on évite d'utiliser une relation d'héritage. De plus, on peut choisir d'utiliser éventuellement une sous-classe de `ArrayList` si on en a besoin. Si on choisit de faire hériter `ArrayListeTypee` de `ArrayList`, dans ce cas `ArrayListTypee` ne peut plus hériter d'une autre classe, mais la relation d'héritage nous permet de garantir qu'une instance

d'`ArrayListTypee` est également une instance d'`ArrayList`. On peut ainsi utiliser une instance d'`ArrayListTypee` à la place d'une instance d'`ArrayList`. De plus, on peut récupérer par héritage les méthodes d'`ArrayList` alors qu'avec la délégation on est obligé de les réécrire même si leur code est le même que celui de celles d'`ArrayList`.

On choisit dans la suite d'étendre `ArrayList` avec `ArrayListTypee`. Pour pouvoir représenter le type des objets contenu dans la liste, on peut utiliser l'API de JAVA qui nous fournit une classe appelée `java.lang.reflect.Class` représentant les classes. Cette classe possède en particulier une méthode `isInstance(Object o)` qui renvoie **`true`** si `o` est bien du type représenté par l'objet `Class` manipulé.

La classe `Object` possède une méthode `getClass` permettant de récupérer un objet de type `Class` représentant la classe de l'objet considéré.

Par exemple, le code suivant récupére un objet de type `Class` représentant la classe `java.lang.String` (les chaînes de caractères) et vérifie qu'un objet de type `String` est bien instance de cette classe :

```
String s1 = "Coucou";
java.lang.reflect.Class classe = s1.getClass();

String s2 = "Blabla";
if (! classe.isInstance(s2)) {
  System.out.println("On ne doit pas arriver ici !");
}
```

Nous ne considérerons que les caractéristiques suivantes de la classe `ArrayList` :
- elle possède un constructeur prenant en paramètre un entier représentant la capacité initiale de la liste ;
- une méthode `add` prenant une instance d'`Object` en paramètre et l'ajoutant à la fin de la liste. Cette méthode renvoie **`true`** si l'objet a été ajouté ;
- une méthode `remove` prenant une instance d'`Object` en paramètre et enlevant l'objet considéré si celui-ci est dans la liste. Cette méthode renvoie **`true`** si l'objet a été effectivement enlevé ;
- une méthode `iterator` renvoyant un objet de type `Iterator` permettant de parcourir la liste.

4. écrire la classe `ArrayListTypee`. Cette classe possédera le constructeur et les méthodes suivants :
   - un constructeur prenant en paramètre :
     - la capacité initiale de la liste
     - un objet dont le type sera celui des éléments contenus dans la liste
   - `ajouter` permettant d'ajouter un objet du type associé à la liste. Elle renverra **`true`** si l'objet est ajouté. Il faudra vérifier que le type de l'objet à ajouter est correct ;
   - `remove` permettant de retirer le dernier élément de la liste ;
   - `iterator`, qui renvoie un itérateur sur la liste.

   On réfléchira à la necessité de redéfinir ces méthodes.

   En cas d'incompatibilité de type, une exception de type `TypeException` sera levée et propagée. On écrira la classe correspondante.

   Voici tout d'abord le source de la classe `TypeException` :

```
/**
 * <code>TypeException</code> represente une exception levee en cas
 * d'incompatibilite de type.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class TypeException extends Exception {
```

```
    public TypeException(String message) {
        super(message);
    }
}
```

Pour la classe `ArrayListTypee`, il fallait réfléchir un peu avant de la coder :

- la classe doit posséder un attribut de type `Class` permettant de représenter le type des objets contenus dans la liste ;
- le constructeur d'`ArrayListTypee` fait bien évidemment appel au constructeur d'`ArrayList`. Cet appel devait être la première instruction du constructeur. On utilise ensuite la méthode `getClass` sur l'objet passé en paramètre pour obtenir le type des éléments de la liste et le stocker dans l'attribut ;
- la méthode `ajouter` vérifie simplement que l'objet passé en paramètre est du bon type (grâce à la méthode `isInstance`), lève une exception si le type n'est pas correct (il faut donc spécifier cette exception dans la signature de la méthode) et appeler la méthode `add` de `ArrayList` si le type est correct ;
- en ce qui concerne les méthodes `remove` et `iterator`, il n'est pas nécessaire de les redéfinir (même `remove`, car si le type de l'objet passé en paramètre n'est pas le bon, l'objet ne sera pas enlevé de la liste et la méthode `remove` d'`ArrayList` renverra **`false`**) ;
- on pourra remarquer que ces indications étaient « données » par l'énoncé... En effet, on ne redéfinit pas la méthode `add` dans `ArrayListTypee`, car on ne pourra pas ajouter à sa signature une exception (le principe d'héritage impose que les exceptions levées par une méthode redéfinissant une méthode de la classe mère soient compatibles avec celles définies dans la signature de la méthode de la classe mère). C'est pour cela que la méthode s'appelait `ajouter`.

Voici le code source de la classe `ArrayListTypee` :

```
import java.util.*;
import java.lang.reflect.*;

/**
 * <code>ArrayListTypee</code> represente une ArrayList dont les
 * elements ont un type precis.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class ArrayListTypee extends ArrayList {

    private Class type;

    /**
     * Cree une instance d'<code>ArrayListTypee</code>.
     *
     * @param capacite un <code>int</code> qui est la capacite
     *                 initiale de la liste
     * @param o un <code>Object</code> dont le type sera celui des
     *          elements de la liste
     */
    public ArrayListTypee(int capacite, Object o) {
        super(capacite);
```

```
        this.type = o.getClass();
    }

    /**
     * <code>ajouter</code> permet d'ajouter un objet dans la liste.
     *
     * @param o l'instance d'<code>Object</code> a ajouter
     * @return un <code>boolean</code> qui est <code>true</code> si
     *         l'objet est ajoute
     * @exception TypeException si le type de <code>o</code> n'est pas
     *                          le bon
     */
    public boolean ajouter(Object o) throws TypeException {
        if (! type.isInstance(o)) {
            throw new TypeException("Le type de l'objet n'est pas correct");
        }

        return(add(o));
    }

    @Override public String get(int index) { return ""; }
}
```

5. la classe `ArrayList` possède une méthode `get(int index)` qui renvoie l'instance d'`Object` stockée à la position `index` de la liste. Serait-il possible de redéfinir cette méthode dans `ArrayListTypee` de telle sorte que son type de retour soit le type associé à la liste ?

   Non, ce n'est pas possible : même si on peut redéfinir la méthode `get` en mettant une sous-classe d'`Object` comme type de retour de la méthode, on est incapable de définir un type de retour de la méthode avec l'objet de type `Class` stocké en attribut.

6. écrire une classe de test `TestListeChaines` qui crée un objet de type `ArrayListTypee` de capacité initiale 2 et ne contenant que des objets de type `String` et ajoutant deux objets de type `String` dedans.

   Pas de problème particulier, il fallait utiliser un bloc **`try`**/**`catch`** pour gérer les éventuelles exceptions de type `TypeException` pouvant être levées. On remarquera que cela ne peut pas arriver avec le programme proposé.

```
/**
 * <code>TestListeChaines</code> permet de test la classe
 * <code>ArrayListTypee</code>.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class TestListeChaines {

    public static void main(String[] args) {
        ArrayListTypee array = new ArrayListTypee(2, "");

        try {
```

```
            array.ajouter("Coucou");
            array.ajouter("Blabla");
        } catch (TypeException e) {
            System.out.println("Ca ne doit pas arriver !");
        }
    }
}
```

3 Étude du *design pattern State*

On souhaite étudier le comportement d'une machine à café fonctionnant avec des jetons : pour obtenir une boisson, il faut utiliser un jeton préacheté.

À l'état initial, la machine est en attente d'un jeton. Lorsque l'on introduit un jeton, elle passe dans un état où elle attend l'appui sur le bouton de demande de café. Si on appuie sur « Annulation », elle rend le jeton et attend un jeton de nouveau. Si on appuie sur le bouton de demande de café, elle passe dans un état « Café commandé ». Dans ce cas, elle fait le café et le sert et s'il reste des cafés revient en état d'attente de jeton, sinon va dans un état indiquant qu'il faut la recharger.

1. représenter le comportement de la machine par un diagramme de machines d'états.

   Pas de problème particulier ici, le diagramme de machines d'états correspondant est proposé sur la figure 4.

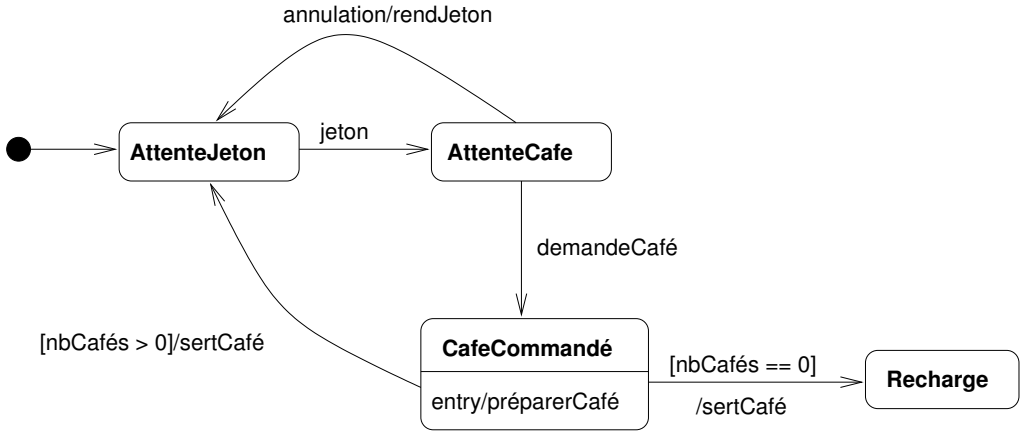

FIG. 4 – Diagramme de machine d'état de la machine à café

On souhaite représenter la machine à café par une classe `MachineCafe`. Dans un premier temps, on souhaite utiliser la modélisation suivante :

- la classe aura un attribut entier représentant l'état dans lequel la machine se trouve ;
- la classe aura un attribut entier représentant le nombre de cafés se trouvant dans la machine ;
- la classe aura un ensemble de constantes entières représentant les différents états possibles de la machine ;
- chaque événement ou action associé aux transitions du diagramme sera représenté par une méthode de la classe. À l'intérieur de ces méthodes, on affichera les actions effectuées par la machine *suivant l'état dans lequel la machine à café se trouve à l'appel de la méthode* : « ce n'est pas possible de

faire cette action, vous avez déjà mis un jeton », « Je fais votre café » etc. Pour simplifier l'écriture de la classe, on pourra utiliser `println` au lieu de `System.out.println`.

2. écrire la classe `MachineCafe` en Java. En ce qui concerne les méthodes correspondant aux actions ou événements associés aux transitions du diagramme, on n'en détaillera qu'une seule et on précisera la signature des autres.

   Voici une proposition de corrigé pour la classe `MachineCafe`. Je n'ai pas mis de documentation Javadoc pour éviter de surcharger le code. J'ai utilisé un **`switch`** dans chacune des méthodes, mais une succession de **`if...elseif`** était tout à fait possible.

```
public class MachineCafe {

    public static final int ATTENTE_JETON = 1;
    public static final int ATTENTE_CAFE = 2;
    public static final int CAFE_COMMANDE = 3;
    public static final int RECHARGE = 4;

    private int etat = RECHARGE;
    private int nbCafe;

    public MachineCafe(int nbCafe) {
        System.out.println("Je démarre...");
        this.nbCafe = nbCafe;
        if (this.nbCafe > 0) {
            this.etat = ATTENTE_JETON;
            System.out.println("Je suis en attente de jeton");
        }
    }

    public void jeton() {
        switch (this.etat) {
        case ATTENTE_JETON:
            this.etat = ATTENTE_CAFE;
            System.out.println("Vous venez d'inserer un jeton");
            System.out.println("J'attends maintenant le choix d'un cafe");
            break;
        case ATTENTE_CAFE:
            System.out.println("Vous venez déjà de mettre un jeton," +
                               " il faut choisir votre cafe maintenant !");
            break;
        case CAFE_COMMANDE:
            System.out.println("Attendez un peu, je sers votre cafe");
            break;
        case RECHARGE:
            System.out.println("Ca ne sert a rien de mettre un jeton," +
                               " il n'y a plus de cafe !");
            break;
        }
    }
```

```
public void annulation() {
    switch (this.etat) {
    case ATTENTE_JETON:
        System.out.println("Vous n'avez pas encore mis de jeton...");
        break;
    case ATTENTE_CAFE:
        this.etat = ATTENTE_JETON;
        System.out.println("OK, je vous rends votre jeton");
        break;
    case CAFE_COMMANDE:
        System.out.println("Ce n'est plus possible de vous rendre votre" +
                           " jeton, le cafe est en train d'etre fait");
        break;
    case RECHARGE:
        System.out.println("Vous ne pouvez pas recuperer votre jeton," +
                           " vous n'en avez pas introduit !");
        break;
    }
}

public void demandeCafe() {
    switch (this.etat) {
    case ATTENTE_JETON:
        System.out.println("Vous devez d'abord inserer un jeton");
        break;
    case ATTENTE_CAFE:
        this.etat = CAFE_COMMANDE;
        System.out.println("OK, je prepare votre cafe...");
        break;
    case CAFE_COMMANDE:
        System.out.println("Vous avez deja demande un cafe !");
        break;
    case RECHARGE:
        System.out.println("Ca ne sert a rien de demander un cafe," +
                           " il n'y en a plus !");
        break;
    }
}

public void sertCafe() {
    switch (this.etat) {
    case ATTENTE_JETON:
        System.out.println("Vous devez d'abord inserer un jeton");
        break;
    case ATTENTE_CAFE:
        System.out.println("Pas de cafe servi...");
        break;
    case CAFE_COMMANDE:
        System.out.println("Le cafe est servi !");
        this.nbCafe--;
```

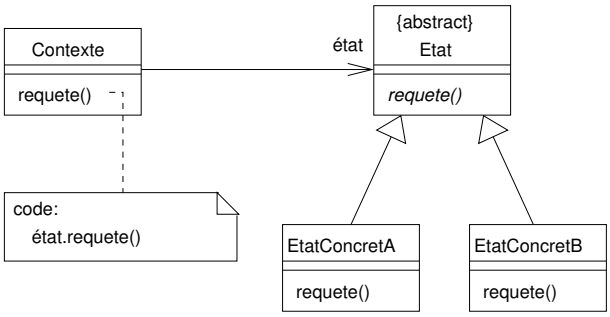

FIG. 5 – Le *design pattern State*

```
            if (this.nbCafe == 0) {
                System.out.println("Il n'y a plus de cafe, rechargez moi !");
                this.etat = RECHARGE;
            } else {
                this.etat = ATTENTE_JETON;
            }
            break;
        case RECHARGE:
            System.out.println("Pas de cafe servi...");
            break;
        }
    }
}
```

Supposons maintenant que le fabricant de la machine décide pour augmenter ses ventes que toutes les 10 demandes en moyenne, il offre un café supplémentaire. On suppose que dans la modélisation, on rajoute un état « Gagnant » et qu'on ne s'occupe pas du calcul de probabilité (une condition `gagnant` permettra de déterminer si l'on a gagné ou pas lors de l'appui sur le bouton de demande de café).

3. quelles sont les modifications imposées par cet ajout ? Qu'en pensez-vous ?

   Évidemment, il va falloir changer toutes les méthodes écrites précédemment dans la classe `MachineCafe` : un nouvel état étant disponible, il faut rajouter un cas dans le **`switch`** de chacune des méthodes.

   En particulier, la méthode `demandeCafe` va devenir compliquée : elle devra gérer les changements d'état en fonction de la valeur de la condition `gagnant`.

Pour pouvoir pallier ces problèmes, on décide d'utiliser un *design pattern* appelé *State* [2, 1]. Ce *pattern* est représenté sur la figure 5. Une classe appelée `Contexte`, dont les objets peuvent posséder plusieurs états comme `EtatA` ou `EtatB`, délégue ses services à des classes représentant ces états. Chaque classe « état » implante donc tous les services de la classe `Contexte` et est responsable du comportement de la classe `Contexte` lorsque celle-ci est dans cet état.

4. on suppose dans un premier temps que l'on considère le diagramme de machines d'états représenté sur la figure 6.

   Ce diagramme représente le fonctionnement d'un objet « horloge » : un événement extérieur, `clic`, change l'état de l'horloge en le faisant passer de `Tic` à `Tac`.

   En supposant que l'on a adapté le diagramme de classes du *pattern* de la façon suivante :

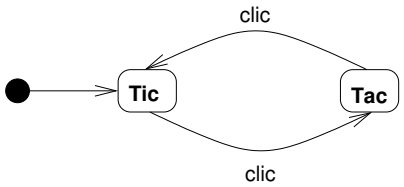

FIG. 6 – Diagramme de machine d'état d'une « horloge »

– la classe abstraite `EtatHorloge` possède une seule méthode, `clic` ;
– les deux classes `Tic` et `Tac` réalisent l'interface ;

écrire un diagramme de séquence représentant trois appels de `clic` par un programme de test sur un objet de type `Horloge` et les appels aux objets qui instancient `EtatHorloge`. Que peut-on en déduire sur la relation existant entre `EtatHorloge` et `Horloge` ?

Un diagramme de séquence possible est représenté sur la figure 7. On peut remarquer que :

– l'objet de type `Horloge` délègue bien son comportement alternativement aux objets de type `Tic` et `Tac` ;
– les objets de type `Tic` et `Tac` changent l'état de l'objet de type `Horloge`. On peut donc supposer qu'il existera une association entre la classe abstraite `EtatHorloge` et la classe `Horloge`, ainsi qu'une méthode `setEtat` dans `Horloge`. On peut également supposer que c'est l'objet de type `Horloge` qui change lui même son état, mais ce n'est pas la solution que j'ai retenue (essentiellement pour découpler complétement l'horloge de ses états).

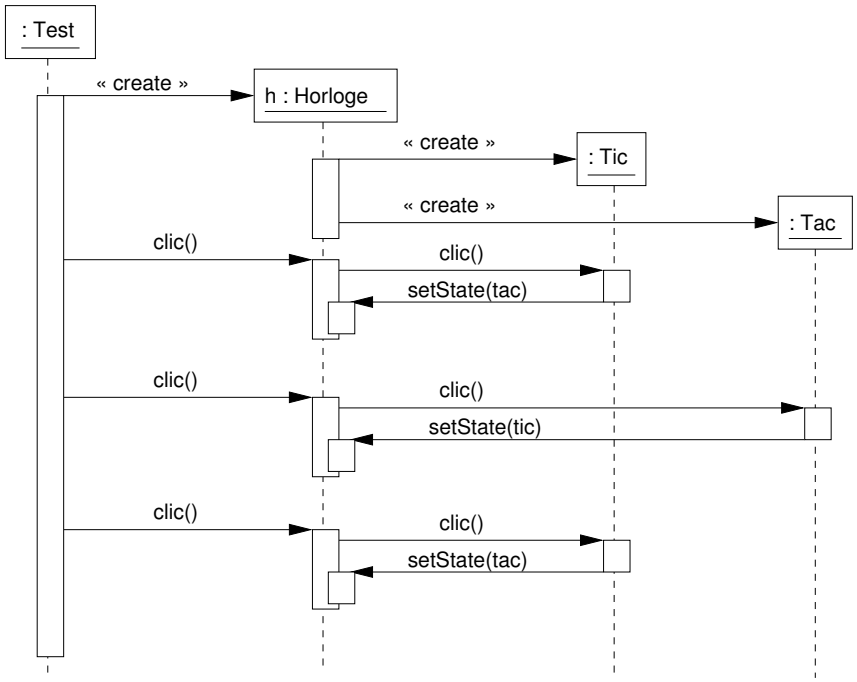

FIG. 7 – Diagramme de séquence représentant des appels à `clic`

5. adapter le *design pattern State* au problème de la machine à café en proposant un diagramme de

classes adapté (**on ne considérera pas l'état supplémentaire `Gagnant` qui nous a amené à utiliser le *pattern***).

Le diagramme de classes est proposé sur la figure 8. Rien de bien particulier ici. Je n'ai pas détaillé les constructeurs des différentes classes, mais il faut bien entendu passer par exemple un objet de type `MachineCafe` en paramètre des constructeurs des objets de type `EtatMachineCafe` pour les initialiser.
J'ai également choisi des accesseurs pour chaque état possible et un modifieur permettant de changer l'état de la machine.

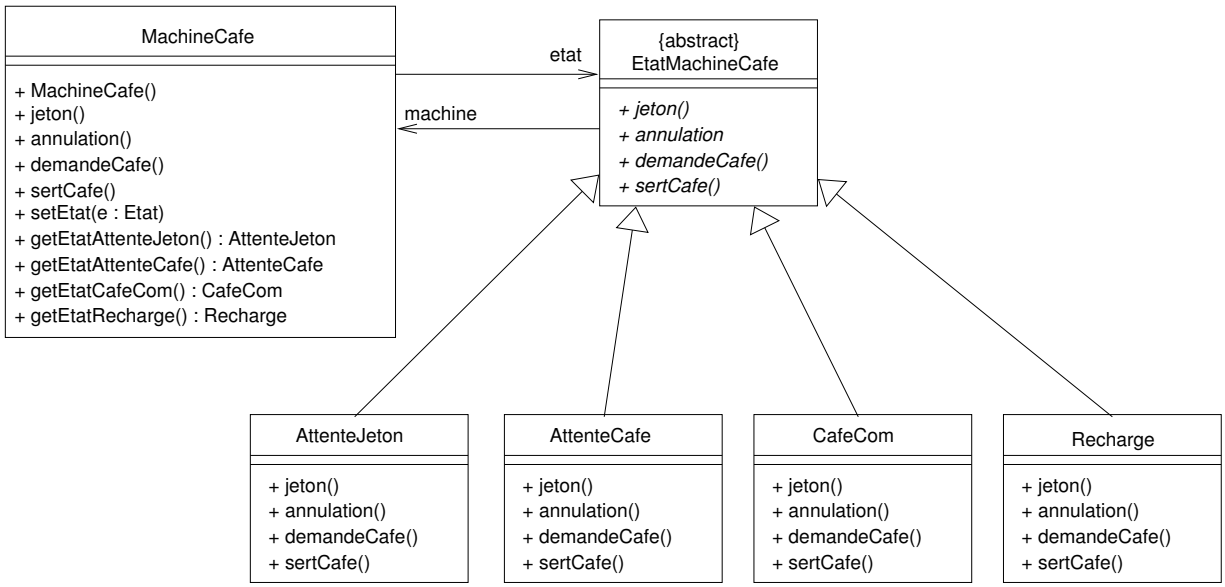

FIG. 8 – Diagramme de classes de la machine à café utilisant le *pattern*

6. écrire la classe `MachineCafe`.

Le source de la classe `MachineCafe` est présenté ci-après. Rien de bien particulier, puisque tout le travail est délégué à l'attribut `etat`.

```
public class MachineCafe {

    private EtatMachineCafe etat;
    private AttenteJeton attenteJeton;
    private AttenteCafe attenteCafe;
    private CafeCom cafeCom;
    private Recharge recharge;

    public MachineCafe() {
        this.attenteJeton = new AttenteJeton(this);
        this.attenteCafe = new AttenteCafe(this);
        this.cafeCom = new CafeCom(this);
        this.recharge = new Recharge(this);
        this.etat = this.attenteJeton;
    }
```

```
    public final void setEtat(EtatMachineCafe etat) {
        this.etat = etat;
    }

    public AttenteJeton getAttenteJeton() {
        return this.attenteJeton;
    }

    public AttenteCafe getAttenteCafe() {
        return this.attenteCafe;
    }

    public CafeCom getCafeCom() {
        return this.cafeCom;
    }

    public Recharge getRecharge() {
        return this.recharge;
    }

    public void jeton() {
        this.etat.jeton();
    }

    public void annulation() {
        this.etat.annulation();
    }

    public void demandeCafe() {
        this.etat.demandeCafe();
    }

    public void sertCafe() {
        this.etat.sertCafe();
    }
}
```

7. écrire la classe abstraite `EtatMachineCafe`.

   Voici le source de la classe `EtatMachineCafe`. Rien de bien particulier, j'ai juste ajouté un constructeur prenant un objet de type `MachineCafe` en paramètre et un accesseur pour cet objet.

```
public abstract class EtatMachineCafe {

    private MachineCafe machine;

    public EtatMachineCafe(MachineCafe machine) {
        this.machine = machine;
    }
```

```
    public MachineCafe getMachineCafe() {
        return this.machine;
    }

    abstract public void jeton();

    abstract public void annulation();

    abstract public void demandeCafe();

    abstract public void sertCafe();
}
```

8. écrire la classe `AttenteJeton`.

   Voici le source de la classe `AttenteJeton`. Cette classe hérite de la classe `EtatMachineCafe` et redéfinit ses méthodes abstraites. Rien de bien particulier sinon, on code bien les méthodes en tenant compte de l'état particulier dans lequel on se trouve.

```
public class AttenteJeton extends EtatMachineCafe {

    public AttenteJeton(MachineCafe machine) {
        super(machine);
    }

    public void jeton() {
        System.out.println("Vous venez d'inserer un jeton");
        this.getMachineCafe().setEtat(this.getMachineCafe().getAttenteCafe());
    }

    public void annulation() {
        System.out.println("Vous n'avez pas encore mis de jeton !");
    }

    public void demandeCafe() {
        System.out.println("Vous n'avez pas encore mis de jeton !");
    }

    public void sertCafe() {
        System.out.println("Pas possible de servir un cafe, " +
                           "vous n'avez pas encore mis de jeton !");
    }
}
```

Références

[1] E. Freeman, E. Freeman, B. Bates, and K. Sierra. *Head first design patterns*. O' Reilly, 2005.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, 1994.