IN201 : 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.

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.

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 1.

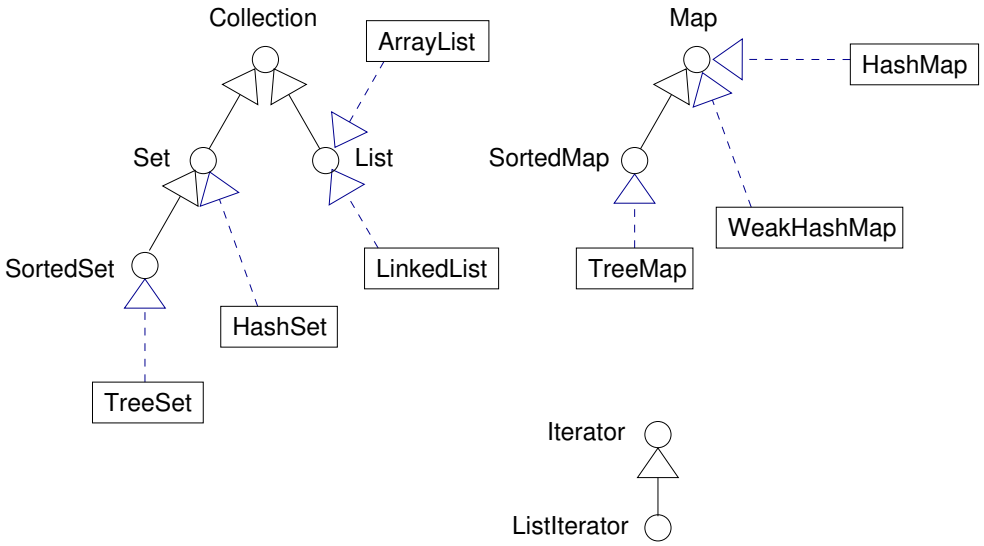

FIG. 1 – Diagramme de classes présentant les classes et interfaces principales du paquetage `java.util`

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

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.

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.

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 ?

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.

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.

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.

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 ?

Pour pouvoir pallier ces problèmes, on décide d'utiliser un *design pattern* appelé *State* [2, 1]. Ce *pattern* est représenté sur la figure 2. 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 3.

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

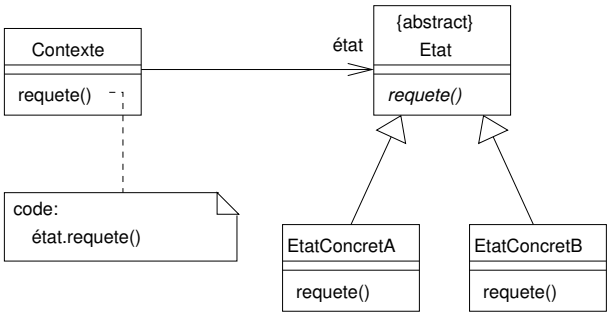

FIG. 2 – Le *design pattern State*

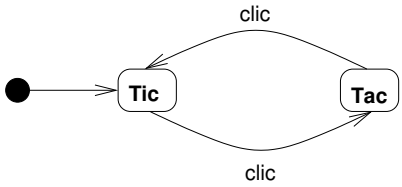

FIG. 3 – Diagramme de machine d'état d'une « horloge »

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` ?

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***).
6. écrire la classe `MachineCafe`.
7. écrire la classe abstraite `EtatMachineCafe`.
8. écrire la classe `AttenteJeton`.

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.