IN201 : Examen

SUPAERO 2A

Christophe Garion <garion@supaero.fr>

3 mars 2008

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 en 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 du système immunitaire humain

Cet exercice a été élaboré en utilisant les références [3, 1]. Il ne prétend pas être une référence en médecine ou en biologie cellulaire. Certaines approximations ont été faites pour simplifier le sujet.

On cherche à fournir à un centre de recherche en biologie un logiciel de simulation de réaction du système immunitaire humain à différents virus et bactéries. Dans un premier temps, on cherche donc à proposer un modèle objet des notions « métier » nécessaires aux biologistes pour modéliser le système immunitaire.

Le système immunitaire est une collection de mécanismes permettant à un organisme de se protéger de l'infection en détectant et en tuant des agents pathogènes. Il faut donc qu'il puisse distinguer les molécules du corps des molécules étrangères. Les agents pathogènes peuvent être des bactéries (comme l'anthrax), des virus (comme l'herpès), des protozoaires (comme la malaria), des champignons (comme le candidiasis), des parasites (comme le ver solitaire) ou des protéines (comme les prions). Les différents pathogènes ont pour effet d'attaquer ou de modifier les cellules du corps.

Le système immunitaire d'un être humain est constitué d'un ensemble de protéines, de cellules, d'organes et de tissus. Il est décomposé en plusieurs couches : une couche physique, le système inné et le système adaptatif. La couche physique empêche certains pathogènes, comme les virus et les bactéries d'entrer dans le corps. Si un pathogène réussit à rentrer, le système immunitaire inné déclenche une première réponse non spécifique. Si celle-ci n'est pas efficace, le système immunitaire adaptatif permet d'adapter la réponse du corps au pathogène et de se souvenir de cette réponse appropriée via la mémoire immunologique.

La couche physique comporte par exemple la peau, les poumons, les larmes produites par les yeux, la flore intestinale ou des barrières chimiques, par exemple contenues dans la salive (enzymes) ou produites par la peau (peptides).

Le système immunitaire inné entre en action lorsqu'un organisme étranger entre dans le corps. Il propose une réponse générique au pathogène sous plusieurs formes : inflammation, système du complément (ensemble d'une vingtaine de protéines permettant de détruire la membrane des cellules étrangères), leucocytes (globules blancs). Ces derniers ont plusieurs formes, la plus connue étant les phagocytes, tous capables de phagocyter un pathogène. On trouve parmi eux les macrophages, les neutrophiles et les cellules dendritiques.

Le système immunitaire adaptatif fournit lui une réponse spécialisée par type de pathogène. Ses principales fonctions sont : la reconnaissance des antigènes n'appartenant pas au corps, la génération d'une réponse adaptée et le développement d'une mémoire immunologique. Il utilise un type particulier de leucocytes, les lymphocytes. Les lymphocytes peuvent reconnaître des cibles pathogènes particulières. Ils se divisent en deux catégories : les lymphocytes T, et les lymphocytes B, qui peuvent créer des plasmocytes fabriquant des anticorps.

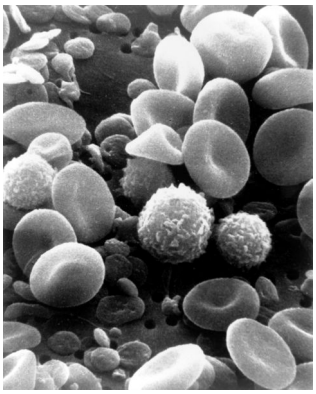

FIG. 1 – Une photo prise au MEB d'un échantillon de sang circulant humain. On distingue de nombreux globules rouges sous forme de disques applatis biconcaves, ainsi que de nombreux types de globules blancs : des lymphocytes, un monocyte, un neutrophile, et de nombreuses plaquettes (source NCA, 1982).

1. proposer un diagramme de classes d'analyse représentant le domaine. On fera apparaître les classes, les relations entre classes, les noms de rôles et les multiplicités des associations, et on justifiera l'utilisation de classes abstraites et d'interfaces. On pourra utiliser quelques attributs et représenter quelques méthodes si nécessaire.
2. représenter par un diagramme de séquence le scénario suivant :
   (a) le virus de la grippe entre dans un corps humain ;
   (b) le corps réagit dans un premier temps en produisant une inflammation ;
   (c) un lymphocyte B a le bon récepteur antigène pour le virus. Il inactive le virus ;
   (d) le lymphocyte phagocyte alors le virus ;
   (e) le lymphocyte produit alors un complexe peptidique ;
   (f) ce complexe est reconnu par un lymphocyte T qui stimule le lymphocyte B par l'envoi de cytokines ;
   (g) le lymphocyte B se réplique alors en plasmocytes qui secrètent des anticorps permettant de neutraliser le virus.

   On supposera que l'on dispose d'une instance de la classe `SystemeImmunitaire`. On réfléchira en particulier aux appels de méthodes : dans quel sens se font-ils ? Avec quels arguments ?
3. on s'intéresse maintenant au cycle de vie d'un lymphocyte B. Un lymphocyte « naît » dans la moelle osseuse. Il est alors en mode « naïf » et passe dans le système lymphatique. Lorsqu'il rencontre un de ses antigènes de prédilection, il engloutit l'antigène et le digère. Il affiche ensuite des fragments de l'antigène. La combinaison de l'antigène attire alors un lymphocyte auxiliaire T mature. Ce dernier sécrète des cytokines qui enclenchent un processus de division du lymphocyte B. Celle-ci produit alors des plasmocytes qui peuvent produire des anticorps. Les lymphocytes meurent au bout de 2-3 jours, sauf environ 10% des cellules qui deviennent des lymphocytes B à mémoire pour pouvoir réagir plus rapidement à la prochaine infection.

   Proposer un diagramme de machine d'états modélisant le cycle de vie d'un lymphocyte B.

2 Étude du *design pattern* Adaptateur

Lors du cours, lorsque nous voulions manipuler un ensemble d'objets, nous avons fait appel à des *collections*. En Java, Les collections sont un ensemble de classes appartenant au paquetage `java.util` et représentant différentes façons de stocker un ensemble d'objets : listes, tas, arbres etc. Nous considérerons dans toute la suite que toutes les classes et interfaces appartiennent à `java.util`. Il faudra en tenir compte lors de l'écriture du code.

Pour pouvoir accéder au contenu d'une collection, nous avons utilisé des itérateurs : un itérateur est un objet permettant d'accéder de façon uniforme aux objets d'une collection, sans s'occuper de la façon dont ces objets sont stockés. Un itérateur est un objet implantant l'interface `java.util.Iterator`. Cette interface possède trois méthodes :

- `hasNext()` qui renvoie un booléen. Elle renvoie **`true`** si la collection possède encore des éléments non parcourus ;
- `next()` qui renvoie le prochain élément de la collection sous la forme d'une instance d'`Object` et qui avance d'un élément dans la collection ;
- `remove()` qui enlève l'élément courant.

1. représenter l'interface `Iterator` sur un diagramme UML ;
2. écrire en Java une méthode statique `affiche` qui prend un itérateur en paramètre et affiche toutes les chaînes de caractères se trouvant dans la collection correspondante ;
3. les anciennes classes représentant les collections en Java (par exemple `Vector` ou `Stack`) implantent une méthode `elements` qui renvoie un objet de type `Enumeration`. L'interface `Enumeration` permet également de parcourir ces collections, mais ne permet pas d'effacer un élément de la collection. Ses méthodes sont :
   - `hasMoreElements()` qui renvoie **`true`** si il y a encore des éléments à parcourir dans la collection ;
   - `nextElement()` qui renvoie le prochain élément sous la forme d'une instance d'`Object`.

   Représenter l'interface sur le diagramme UML précédent.
4. on dispose d'un objet de type `Enumeration` sur une instance de `Vector` par exemple et on souhaiterait pouvoir utiliser dessus la méthode `affiche` développée précédemment sans modifier les interfaces `Enumeration` et `Iterator`, ni la méthode `affiche(Iterator)` développée précédemment. Est-ce possible ? Pourquoi ? Quel « mécanisme » intervient ?
5. pour résoudre ce problème, nous allons utiliser un *design pattern*[2] : l'adaptateur. Il s'agit de réaliser l'interface `Iterator` par une classe `EnumerationIterator` (qui possédera donc toutes les méthodes de l'interface). `EnumerationIterator` utilisera l'instance de `Enumeration` que nous voulions utiliser : les méthodes de `EnumerationIterator` utiliseront celles de `Enumeration`.

   Compléter le diagramme UML avec la classe `EnumerationIterator`.
6. écrire la classe `EnumerationIterator` en Java. On constate que la méthode `remove` n'existe pas dans l'interface `Enumeration`. Indiquer comment écrire la méthode `remove` de `EnumerationIterator`.
7. proposer un diagramme de classe général pour le *design pattern*. L'objectif général de ce *pattern* est d'adapter une interface existante pour l'utiliser dans un nouveau contexte, c'est-à-dire avec une nouvelle interface.

3 Création d'objets et Factory Method

On suppose que l'on souhaite faire une machine automatique cuisinant des tartes aux fruits[1]. Pour cela, on développe dans un premier temps une application Java permettant de simuler le comporte-

[1] Il vaut mieux bien évidemment les faire soi-même, c'est meilleur...

ment de la machine. La classe principale de l'application est `MachineTarte` et dispose d'une méthode `commanderTarte` qui aura l'allure suivante :

```
public Tarte commanderTarte() {
  Tarte tarte = new Tarte();

  tarte.preparer();
  tarte.cuire();
  tarte.emballer();

  return tarte;
}
```

Les méthodes `preparer`, `cuire` et `emballer` sont des méthodes simulant le travail effectif de la machine.

1. on suppose que l'on a une machine évoluée qui peut faire plusieurs types de tartes aux fruits. On va donc rendre `Tarte` abstraite et créer une hiérarchie de classes représentée sur la figure 2.

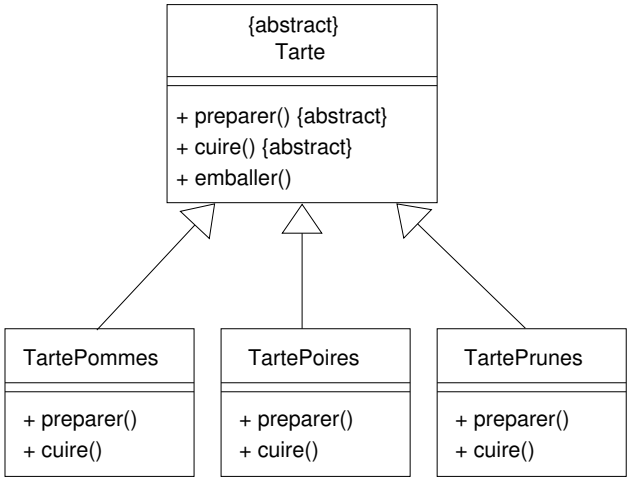

FIG. 2 – Hiérarchie de classes représentant les tartes

   On va donc modifier la méthode `commanderTarte` pour utiliser les nouveaux types de tartes : on va passer en paramètre de `commanderTarte` une chaîne de caractères précisant le type de tarte choisi (par exemple `"pommes"` ou `"poires"`). En fonction de cette chaîne de caractères, on construira la tarte correspondante.

   Modifier la méthode `commanderTarte` en conséquence.

2. si l'on souhaite ajouter de nouvelles tartes, il va falloir modifier la méthode `commanderTarte`. Or on souhaite fermer `commanderTarte` à la modification[2].

   Pour résoudre ce problème, on va déléguer la création des tartes à une classe `FabriqueTarte` via une méthode `creerTarte`.

   (a) représenter sur un diagramme de classes les classes `MachineTarte`, `FabriqueTarte` et la hiérarchie des tartes.

[2] C'est en effet un bon principe de conception objet : lorsqu'une méthode est correcte, on n'« autorise » pas les modifications directes de la méthode, mais on permet de la redéfinir dans une sous-classe.

(b) représenter sur un diagramme de séquence les interactions entre les classes lors de la commande d'une tarte aux pommes.

(c) donner le code Java des classes `FabriqueTarte` et `MachineTarte`.

3. on suppose maintenant que l'on veut pouvoir créer deux grands types de tartes : des tartes normales et des tartes sans gluten (pour les personnes allergiques). La hiérarchie de classes représentant les tartes va donc s'en trouver modifiée. L'utilisateur choisira au départ la « famille » de tarte qu'il veut (sans gluten ou normale), puis le type de la tarte (aux pommes, aux poires etc.). Dans notre simulation, nous voulons donc avoir deux types de `MachineTarte` : un pour les tartes classiques et l'autre pour les tartes sans gluten.

   Nous voulons également rester cohérents : la méthode emballer par exemple devra rester la même pour les deux machines (on utilise un emballage spécial que l'on ne veut pas changer).

   On pourrait donc spécialiser `FabriqueTarte` en `FabriqueTarteNormale` et `FabriqueTarteSansGluten`.

   Que pensez-vous de cette solution ? En particulier, comment créer une machine pour un type de tarte particulier ? Contrôle-t-on l'utilisation qui sera faite des fabriques de tarte ?

4. pour pallier les problèmes soulevés précédemment, nous décidons de placer la méthode permettant de créer les tartes non pas dans une classe extérieure, mais dans les classes `MachineTarte`, `MachineTarteSimple` et `MachineTarteSansGluten`.

   Nous décidons donc d'utiliser le *design pattern*[2] Factory Method. Celui-ci est représenté sur la figure 3.

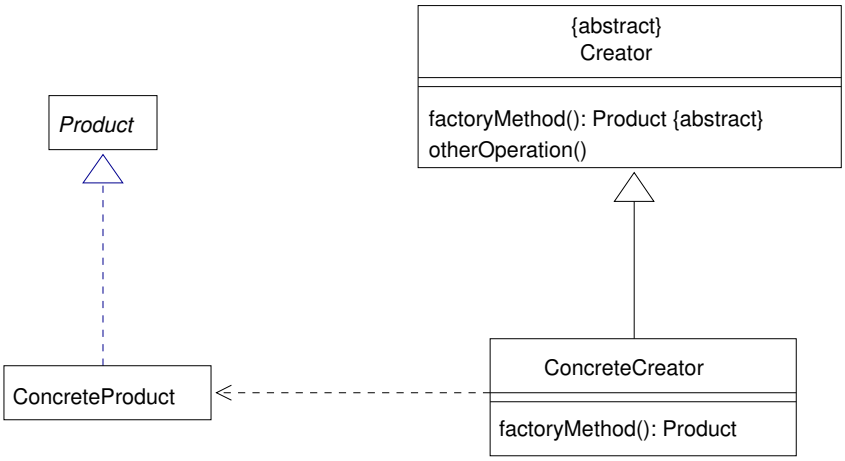

FIG. 3 – Le *Design Pattern* Factory Method

Dans ce *pattern*, la classe `Creator` permet de créer des objets typés par `Product` grâce à la méthode `factoryMethod`. On spécialise ensuite cette classe dans des classes concrètes permettant de créer un ou plusieurs produits concrets en spécialisant la méthode `factoryMethod`.

(a) pourquoi utilise-t-on une interface (ou une classe abstraite) pour `Product` ?

(b) peut-on mettre `ConcreteProduct` comme type de retour de `factoryMethod` dans `ConcreteCreator` ?

(c) proposer une adaptation de ce *pattern* à notre problème via un diagramme de classes.

(d) écrire les classes `MachineTarte` et `MachineTarteSimple`.

Références

[1] Microbiology Department of Pathology and School of Medicine Immunology at the University of South Carolina. Microbiology and immunology on-line. `http://pathmicro.med.sc.edu/book/welcome.htm`.

[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.

[3] Wikipedia. Immune system. `http://en.wikipedia.org/wiki/Immune_System`.