IN201 : Examen

SUPAERO 2A

Christophe Garion <garion@supaero.fr>

20 mars 2006

Cet examen est composé de deux parties indépendantes. Vous avez 2h30 pour le faire. Tous les documents sont autorisés sauf [3] et [2]. Il sera tenu compte de la rédaction.

Le barême est donné ici à titre indicatif. En particulier la question 2.4 du second exercice apporte des points supplémentaires vu sa difficulté.

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

1 Modélisation objet d'une cellule biologique (10 points)

Cet exercice a été réalisé avec l'aide des références [1] et [4]. Il ne prétend pas être une référence en biologie cellulaire. Certaines imprécisions ont été faites pour simplifier la modélisation.

On cherche à réaliser un logiciel de simulation pour un centre de recherche en biologie. En particulier, ce logiciel devra permettre de simuler la vie d'une cellule. Nous ne nous intéresserons pas ici à la modélisation du simulateur à proprement parler. On cherche donc dans un premier temps à modéliser les notions propres au domaine de la biologie cellulaire avec le paradigme objet. Après divers entretiens avec les chercheurs du centre, on a pu regrouper un ensemble de définitions et de faits.

Une cellule est l'unité structurelle fondamentale constituant un être vivant. Toutes les cellules possèdent un ADN et une membrane cytoplasmique. Les cellules possèdent également des mécanismes communs :

- la mitose, appelée également division cellulaire, qui crée une nouvelle cellule ;
- le métabolisme cellulaire, permettant de construire la cellule et rejetant des produits dérivés ;
- la synthèse des protéines, par la transcription de l'ADN en ARN, puis la traduction de l'ARN en protéines.

Une protéine est une séquence d'acides aminés liés par des liaisons peptidiques. Elles sont séparées en deux grandes classes, les protéines fibreuses (kératine, collagène etc.) et les protéines globulaires (hémoglobine, certaines enzymes etc.).

Les cellules sont organisées en deux grandes familles, les procaryotes et les eucaryotes. Par exemple, les bactéries sont des cellules procaryotes et les neurones des cellules eucaryotes.

Les procaryotes peuvent posséder un flagelle leur permettant de se déplacer. Leur ADN se compose d'une molécule circulaire.

Les eucaryotes peuvent également posséder un flagelle ou des cils, mais pas les deux. Leur ADN est composé de plusieurs molécules linéaires et est contenu dans un noyau. Les eucaryotes possèdent également des organites ayant plusieurs fonctions spécifiques : le réticulum endoplasmique qui permet la transformation et le transport des protéines, l'appareil de Golgi qui transporte les protéines vers la membrane plasmique, les mitochondries qui assurent la

production d'énergie et le cytosquelette qui permet le mouvement des différents organites. Les chloroplastes sont des organites présents dans les plantes et les algues.

Le cycle cellulaire représente les différentes phases par lesquelles passe une cellule entre deux divisions successives. Ces phases sont les suivantes :

- G0, phase de quiescence. La cellule a quitté le cycle cellulaire. Les cellules peuvent passer en phase G1 depuis G0 sous l'effet d'antigènes, d'hormones ou de facteurs de croissance.
- G1, première phase de croissance cellulaire. Cette phase a une certaine durée caractéristique de la cellule.
- S pour synthèse, pendant laquelle l'ADN est dupliqué. Elle dure environ 8 heures.
- G2, seconde phase de croissance cellulaire. Cette phase a une durée de 3 heures environ. On passe en phase M à condition que le MPF (*Mitosis Promoting Factor*) soit actif.
- M pour mitose (ou méiose pour les gamètes) qui est la phase de division à proprement parler.

Le cycle peut être interrompu en cas de problèmes (par exemple si l'ADN est endommagé). Dans ce cas, la cellule peut recevoir un signal (que nous appellerons pour simplifier apoptose) d'« auto-destruction », ou passer par une phase de réparation avant un retour dans le cycle. La cellule sait si elle est dans un état réparable. En cas de réparation, le cycle reprend au même endroit.

1. proposer un diagramme de classes d'analyse représentant le domaine. 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.
2. proposer un diagramme de classe détaillé de la classe `Cellule`. On y fera apparaître les attributs, constantes et méthodes pertinents.
3. proposer un diagramme de machines d'états représentant le cycle de vie d'un objet de type `Cellule`.

2 Modification du comportement d'un objet (10 points)

Remarques importantes : 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

Nous avons vu en cours que l'héritage permet de modifier le comportement des objets d'une classe donnée en la spécialisant et en redéfinissant une ou plusieurs de ses méthodes. Cette redéfinition de comportement est statique, car réalisée à la compilation des classes. Nous allons nous intéresser dans ce problème à d'autres moyens pour redéfinir le comportement d'une classe : l'utilisation du *design pattern* Décorateur et du *pattern* d'extension inverse.

Nous nous intéresserons ici à des composants graphiques. On considérera la classe publique `JComponent` du paquetage `javax.swing` et une classe `MyPanel` du paquetage `fr.supaero.gui`

qui étend cette classe (cf. figure 1). `MyPanel` représente une fenêtre graphique simple. La seule méthode considérée *a priori* sera la méthode `paintComponent(Graphics g)` qui permet de dessiner le composant. Elle est redéfinie dans `MyPanel`.

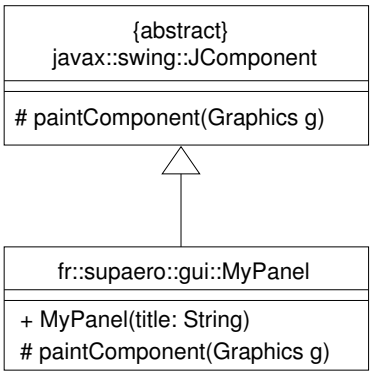

FIG. 1 – La classe `fr.supaero.gui.MyPanel`

On souhaite créer de nouveaux types de fenêtre à partir de `MyPanel` : fenêtre avec bordures, fenêtres avec « ascenseurs » etc. On supposera que l'on ne connaît de l'API Swing de Java que le fonctionnement de la méthode `paintComponent`. Si d'autres connaissances sont nécessaires, elles vous seront rappelées au fur et à mesure du problème.

2.2 Questions et propositions de solutions préliminaires

1. La classe `JComponent` appartient au paquetage `javax.swing` et la classe `MyPanel` appartient au paquetage `fr.supaero.gui`. L'extension de `JComponent` par `MyPanel` et la redéfinition de la méthode `paintComponent` sont-elles possibles ?
2. pour pouvoir créer plusieurs types de fenêtres graphiques à partir de `MyPanel`, on propose dans un premier temps d'étendre directement la classe `MyPanel` et de spécialiser ainsi `MyPanel` selon les besoins. Par exemple, on souhaite pouvoir créer :
   - une fenêtre avec bordure représentée par la classe `MyBorderPanel` ;
   - une fenêtre avec un ascenseur représentée par la classe `MyScrollPanel` ;
   - une fenêtre avec bordure et un ascenseur représentée par la classe `MyBorderAndScrollPanel` ;

   Que pensez-vous de cette solution (on réfléchira au problème de la maintenance et de l'ajout d'autres types de fenêtres) ?
3. dans un second temps, on propose d'inclure directement dans la classe `MyPanel` les caractéristiques (couleur de la bordure etc.). Que pensez-vous de cette solution ?

2.3 Utilisation du *design pattern* Décorateur

Pour pallier les problèmes évoqués dans la section 2.2, nous allons utiliser un *design pattern*, le Décorateur [3]. Les Décorateurs permettent d'attacher des responsabilités de façon dynamique

à des objets[1].

Le diagramme de classes présentant le principe du Décorateur est présenté sur la figure 2.

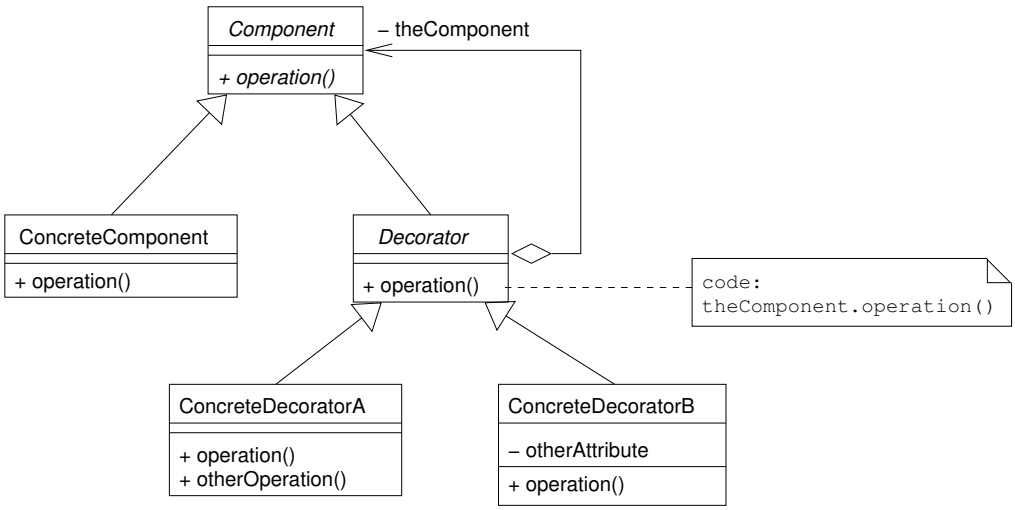

FIG. 2 – Le *pattern* Décorateur

Le diagramme de séquence 3 décrit la création d'un décorateur et l'appel à l'opération depuis un programme de test.

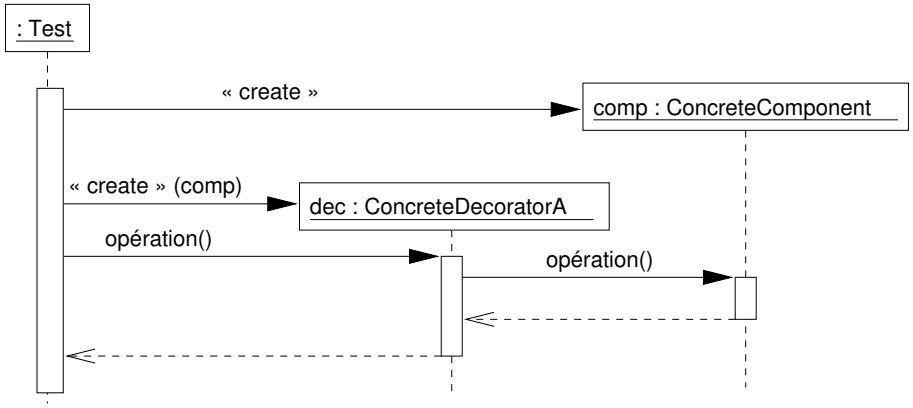

FIG. 3 – Diagramme de séquence présentant le fonctionnement du Décorateur

1. expliquer le fonctionnement du Décorateur en vous appuyant sur le diagramme de séquence 3.
2. proposer un diagramme de classes instanciant ce *pattern* avec les classes `JComponent`, `MyPanel` et deux décorateurs :

[1]On remarquera que le terme « décorateur » s'applique intuitivement dans notre cas...

- `BorderDecorator` qui permet de dessiner un rectangle autour de la fenêtre. On peut fixer la couleur du rectangle à la création et la changer dynamiquement via une instance de la classe `java.awt.Color`. On détaillera la classe ;
- `ScrollBarDecorator` qui ajoute un « ascenseur » sur un composant (on ne détaillera pas la classe).

3. la solution proposée ici permet-elle de résoudre les problèmes soulevés en section 2.2 ?
4. que pensez-vous de la relation d'héritage entre `Component` et `Decorator` ? Aurait-on pu utiliser une relation de réalisation en déclarant `Component` comme une interface ?
5. pourquoi `Decorator` est-elle abstraite ?
6. écrire en JAVA une méthode statique `addBorderAndScroll` qui prend une instance de `JComponent` en paramètre et renvoie un composant décorant l'objet passé en paramètre avec une bordure de couleur rouge et un ascenseur.
7. écrire les classes `Decorator` et `BorderDecorator` en JAVA. On utilisera les méthodes `getHeight()` et `getWidth()` de `JComponent` qui renvoient les dimensions du composant et la méthode `drawRect(int x, int y, int width, int height)` de `Graphics`.
8. si la méthode `paintComponent` de `JComponent` peut lever une exception, doit-on la traiter ? Si oui, comment ?

2.4 Retour sur l'héritage : l'extension inverse (+3 points)

Nous avons vu précédemment que le *pattern* Décorateur nous permettait d'ajouter ou de modifier dynamiquement un comportement d'un objet d'une classe donnée. Nous allons maintenant revenir sur l'héritage. L'héritage nous permet de redéfinir statiquement le comportement d'une classe en redéfinissant une de ses méthodes. La classe fille est responsable dans la méthode redéfinie de l'éventuel appel à la méthode de la classe mère et du traitement de son retour. Cela peut poser plusieurs problèmes :

- comment être sûr que la méthode de la classe fille appelle la méthode de la classe mère ?
- si l'ordre d'appel de la méthode de la classe mère est important, comment le garantir ?

Dans notre cas, nous devons garantir ces deux propriétés :

- on doit appeler `paintComponent` de `MyPanel` pour dessiner la fenêtre ;
- on doit d'abord dessiner la fenêtre avant d'ajouter les décorations.

On peut remarquer dans l'utilisation du *pattern* Décorateur (cf. figure 2) que la méthode `paintComponent` de `Decorator` fait appel à la méthode `paintComponent` de `MyPanel`. D'après le principe de substitution, toute instance d'une sous-classe de `Decorator` devrait avoir au moins le même comportement et donc faire appel à la méthode de `MyPanel` dans sa redéfinition de `paintComponent`. Mais rien ne le garantit *a priori*...

Il faudrait donc « inverser » le principe d'extension pour laisser la classe mère responsable du moment de l'appel des méthodes redéfinies de la classe fille (ces méthodes ne devraient donc pas comporter d'appel à **super**, sinon on se trouve dans un cas de récursion non terminale !). Malheureusement, le principe de liaison tardive impose que la méthode de la classe fille soit appelée en premier. Il nous faudrait donc redéfinir un mécanisme d'appel de méthodes. Pour

cela, il nous faut pouvoir travailler sur les différents types d'un objet directement, i.e. disposer d'objets représentant les classes, les méthodes etc[2].

1. proposer un diagramme de classes ne faisant apparaître que les notions suivantes : classes, héritage , attributs, méthodes, visibilité. On ne s'intéressera donc pas ici aux interfaces, ni aux associations entre classes.
2. l'API de JAVA nous fournit une classe appelée `Class` représentant les classes et un paquetage permettant de travailler avec les notions objets, `java.lang.reflect`. Nous allons les présenter rapidement dans ce qui suit.

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

   La classe `Class` possède un certain nombre de méthodes intéressantes :
   - `getSuperClass()` qui renvoie un objet représentant la super-classe de la classe considérée. Elle renvoie **`null`** si on l'appelle sur un objet représentant la classe `Object` ;
   - `getMethod(String methodName, Class[] parameterTypes)` **`throws`** `NoSuchMethodException` qui permet de récupérer un objet de type `java.lang.reflect.Method` correspondant à la méthode dont le nom est `methodName` et les types de ses paramètres (dans l'ordre) sont représentés par le tableau `parameterTypes`. Si la méthode n'existe pas, une `NoSuchMethodException` est levée. Attention, si la méthode n'est pas trouvée dans la classe, on essaye de la trouver en « remontant » la hiérarchie.

   Écrire une méthode statique `findMethods` prenant en paramètre un objet `obj`, un nom de méthode valide pour cet objet `name` et un tableau d'objets `args` de type `Object` représentant les paramètres de l'appel à cette méthode[3] et qui renvoie une `ArrayList` contenant tous les objets de type `Method` correspondant aux différentes versions de `name` dans la hiérarchie de classe de `obj`.

Pour la suite de l'exercice, vous pourrez vous référer à l'article [2] qui propose une implantation de l'extension inverse (avec beaucoup de restrictions...) et une discussion intéressante sur le principe même d'utilisation de ce « *pattern* ».

Références

[1] The virtual library of biochemistry, moleculer biology and cell biology - cell cycle and cytokinesis. `http://www.biochemweb.org/cell_cycle.shtml`.

[2] S. Behrens. The inverse extension design pattern. `http://www.linuxjournal.com/node/8747`, December 2005.

[3] 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.

[4] Wikipedia. Cell (biology). `http://en.wikipedia.org/wiki/Cell_%28biology%29`.

[2] On fait donc un exercice de méta-modélisation : on représente les concepts objets en objet !

[3] Pour les types primitifs, on utilise les classes *wrapper* comme `Integer`.