IN201 : Corrigé de l'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 [5]. 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.

   Un diagramme de classes est proposé sur la figure 1.

   Quelques remarques sur cette proposition de corrigé :

   - j'ai choisi d'avoir une classe `Cellule` abstraite, car une cellule est toujours une cellule procaryote ou eucaryote. On n'utilise pas d'interface, car il y a des associations existant vers d'autres classes. De la même façon, les classes `Procaryote` et `Eucaryote` sont abstraites, il n'existe pas de cellule « générique », on spécialise toujours en bactéries ou neurones par exemple.
   - à chaque fois que j'avais le choix entre agrégation et composition, j'ai choisi une composition pour signifier que les cycles de vie sont liés (ce qui me paraît intuitif ici).
   - la classe `Cellule` est composée d'une instance de type `ADN` et d'une instance de la classe `Membrane`. Il y a une relation de dépendance vers les classes `ARN` et `Protéine` d'après les fonctionnalités d'une cellule (synthèse des protéines par exemple).
   - les protéines se spécialisent en protéines fibreuses ou globulaires. Elles sont composées d'un ensemble d'acides aminés. J'ai utilisé la contrainte `ordered` pour préciser que l'on avait une séquence. Les liaisons peptidiques sont représentés par une association entre acides aminés.
   - `ADN` est une interface, car les objets de type `ADN` seront des instances des classes `ADNEucaryote` et `ADNProcaryote` réalisant l'interface. Les compositions vers les molécules étaient claires.
   - pour préciser qu'une cellule de type `Procaryote` ne pouvait avoir qu'un ADN de type `ADNProcaryote`, j'utilise une contrainte sur la classe. On pouvait faire de même avec

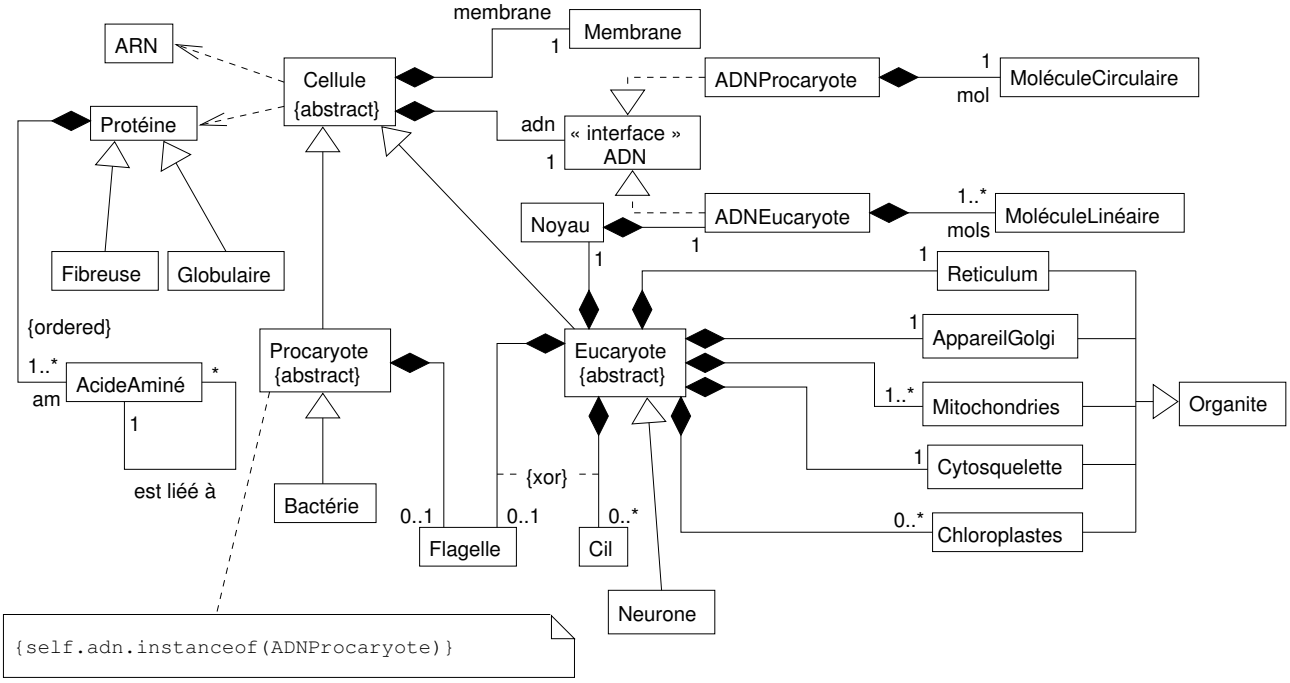

FIG. 1 – Proposition de diagramme de classes d'analyse du problème

`Eucaryote`. Si on réfléchit à l'implantation de cette classe, on contraindra le type du paramètre fourni au constructeur de `Procaryote` pour initialiser l'ADN de l'objet.

– de la même façon, j'utilise une contrainte sur les associations entre `Eucaryote`, `Flagelle` et `Cil` pour préciser qu'une cellule eucaryote ne peut pas avoir à la fois une flagelle et un cil.

On aurait pu détailler encore plus le diagramme (classe représentant des molécules générales, liens entre les protéines et le réticulum, les différents types de protéines etc.), mais j'ai choisi de ne pas alourdir le diagramme.

2. proposer un diagramme de classe détaillé de la classe `Cellule`. On y fera apparaître les attributs, constantes et méthodes pertinents.

Un diagramme de classe détaillé de `Cellule` est proposé sur la figure 2.

Quelques remarques :

– les attributs `membrane` et `adn` se déduisaient des relations de composition du diagramme 1. Je propose des accesseurs à ces deux attributs, mais un modifieur seulement pour `adn` (on ne peut pas modifier la membrane d'une cellule *a priori*). L'attribut `timeG1` représente la durée de la phase G1 et n'est initialisé qu'avec le constructeur.

– l'attribut `phase` représente l'état dans lequel est la cellule sous la forme d'un entier. Je propose également des attributs statiques permettant d'avoir des entiers spécifiques pour chaque phase. Il n'y a pas de modifieur de `phase`, cet attribut est modifié à travers les

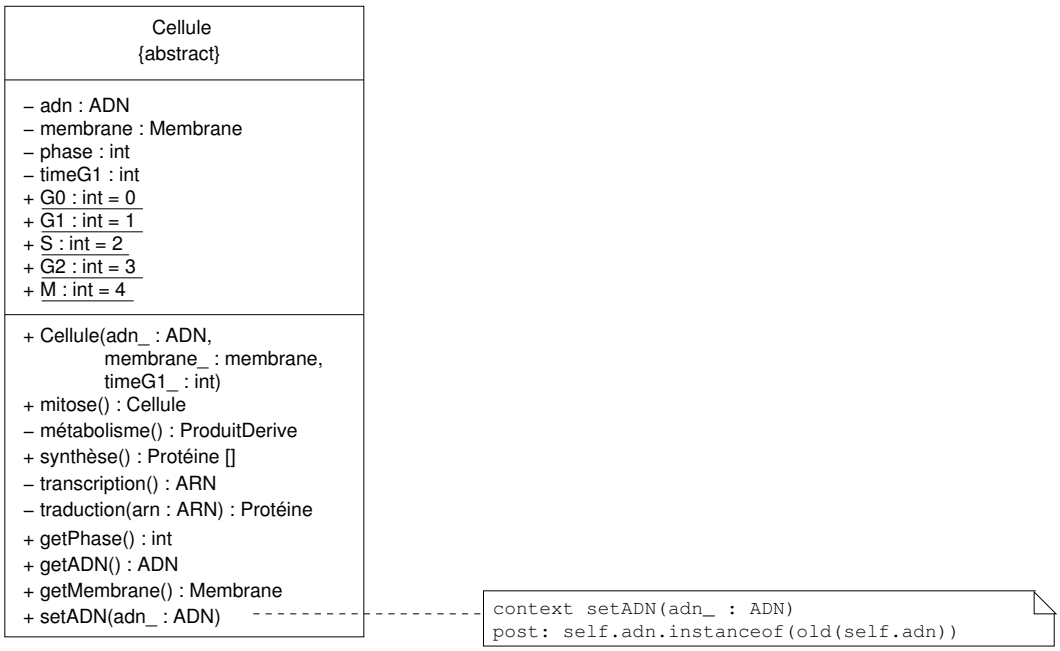

FIG. 2 – Une proposition de diagramme pour la classe `Cellule`

méthodes de la classe.
- le constructeur de la classe permet d'initialiser les attributs `membrane` et `adn`. Je suppose que l'état d'une cellule à sa création est toujours le même.
- il y deux méthodes publiques représentant les mécanismes de la cellule : mitose, métabolisme cellulaire et la synthèse des protéines. Les détails du processus de synthèse sont représentés par deux méthodes privées.
- j'ai utilisé une postcondition pour vérifier que si l'on modifie l'ADN, le type reste correct.

3. proposer un diagramme de machines d'états représentant le cycle de vie d'un objet de type `Cellule`.

   Une proposition de diagramme est présentée sur la figure 3.

   Rien de bien particulier, à part l'utilisation d'un état composite pour mieux modéliser le problème. J'ai également choisi d'utiliser des activités internes pour les activités prenant du temps.

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).

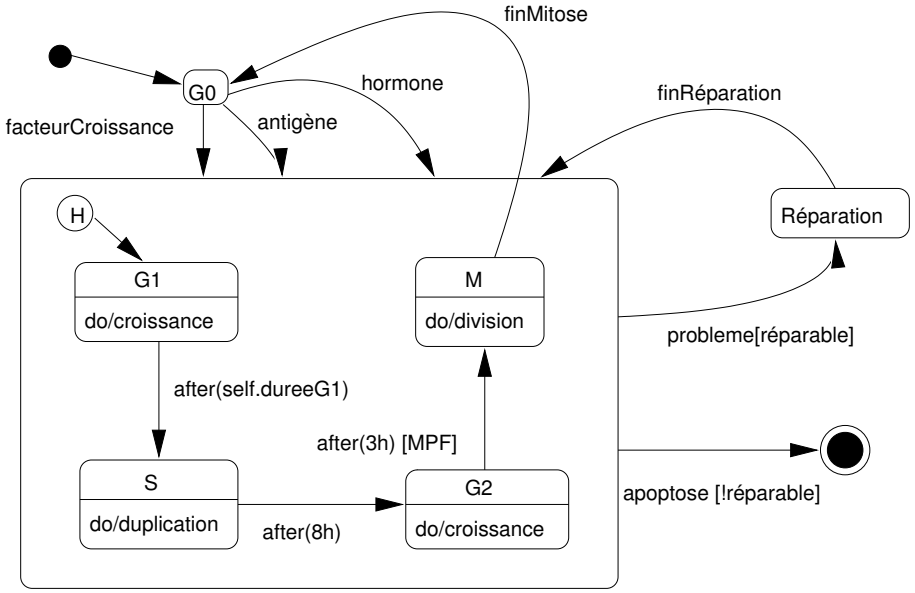

FIG. 3 – Proposition de diagramme de machines d'états pour le cycle cellulaire

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 4). `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`.

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 ?

   Les classes `JComponent` et `MyPanel` appartiennent à deux paquetages différents. Comme `JComponent` est *publique* et n'est pas *finale*, `MyPanel` peut étendre `JComponent`.

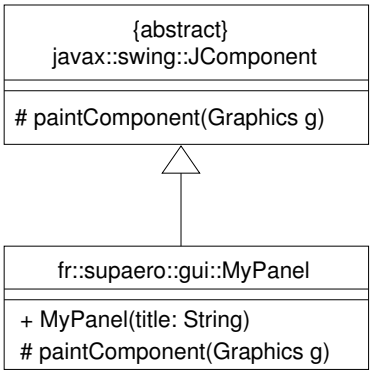

FIG. 4 – La classe `fr.supaero.gui.MyPanel`

La méthode `paintComponent` est protégée dans `JComponent`, elle est donc accessible à `MyPanel` qui étend `JComponent`. `MyPanel` peut donc la redéfinir.

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) ?

   Si l'on prend l'exemple de l'ajout possible d'une bordure et d'ascenseurs pour une fenêtre, on aurait donc trois classes qui étendrait `MyPanel` : `MyBorderPanel`, `MyScrollPanel` et `MyBorderAndScrollPanel` qui peut hériter de l'une ou l'autre des classes. La maintenance est problématique dans ce cas : il se peut qu'une modification pour l'affichage des bordures induise des modifications dans deux classes.

   Si l'on veut ajouter un nouveau type de fenêtre, il ne suffit pas de créer une classe, mais il faut également créer de nouvelles classes pour combiner les différents types de fenêtres. On risque de se trouver rapidement avec un très grand nombre de classes.

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 ?

   Dans ce cas, on se retrouve avec une classe qui comprend toutes les fonctionnalités. Cela pose évidemment problème. Lors de l'ajout d'un nouveau type de fenêtre, il faut modifier la classe `MyPanel` pour lui rajouter des attributs par exemple. Cela suppose un accès à la classe. De plus, l'ajout d'une fonctionnalité implique la modification de la méthode `paintComponent`. Celle-ci va comporter un grand nombre de tests pour savoir s'il faut dessiner la bordure, les ascenseurs etc. Enfin, il problème classique est de pouvoir avoir une double bordure sur une fenêtre, comment faire ici ?

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 5.

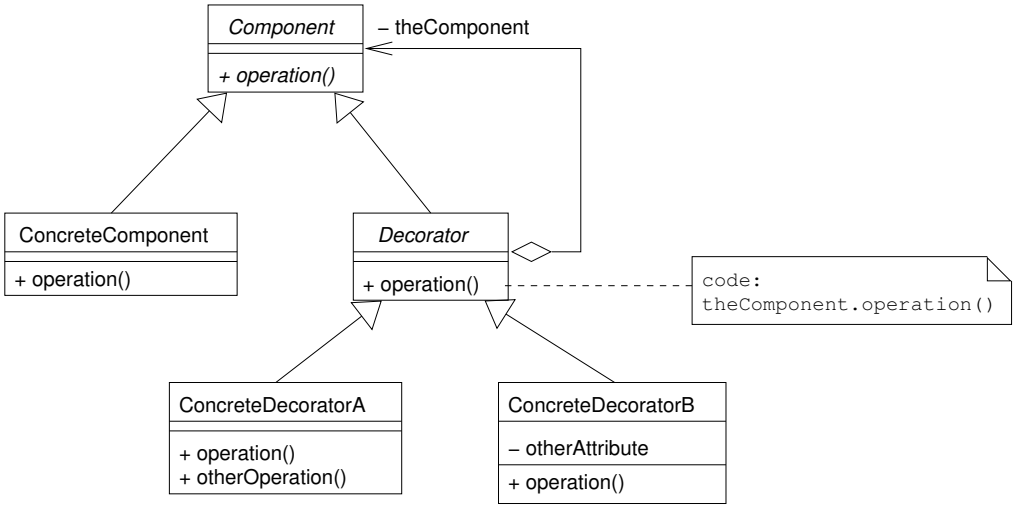

FIG. 5 – Le *pattern* Décorateur

Le diagramme de séquence 6 décrit la création d'un décorateur et l'appel à l'opération depuis un programme de test.

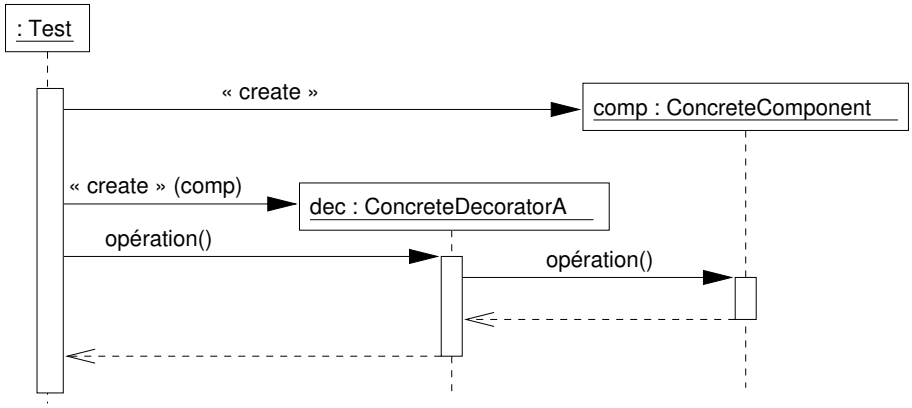

FIG. 6 – Diagramme de séquence présentant le fonctionnement du Décorateur

1. expliquer le fonctionnement du Décorateur en vous appuyant sur le diagramme de séquence 6.

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

Le composant concret est créé indépendamment de son décorateur. C'est la classe de test qui ajoute un comportement au composant concret via le décorateur : on voit bien ici qu'une partie du traitement de l'opération est délégué à l'opération du composant concret.

Le principe de fonctionnement du Décorateur est d'*encapsuler* l'objet dont on veut étendre ou modifier le comportement plutôt que de spécialiser la classe de l'objet.

2. proposer un diagramme de classes instanciant ce *pattern* avec les classes `JComponent`, `MyPanel` et deux décorateurs :
   - `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).

   Un diagramme est proposé sur la figure 7. Rien de bien difficile, en particulier pour la classe `BorderDecorator`. Je n'ai pas mis les paquetages pour ne pas alourdir le diagramme.

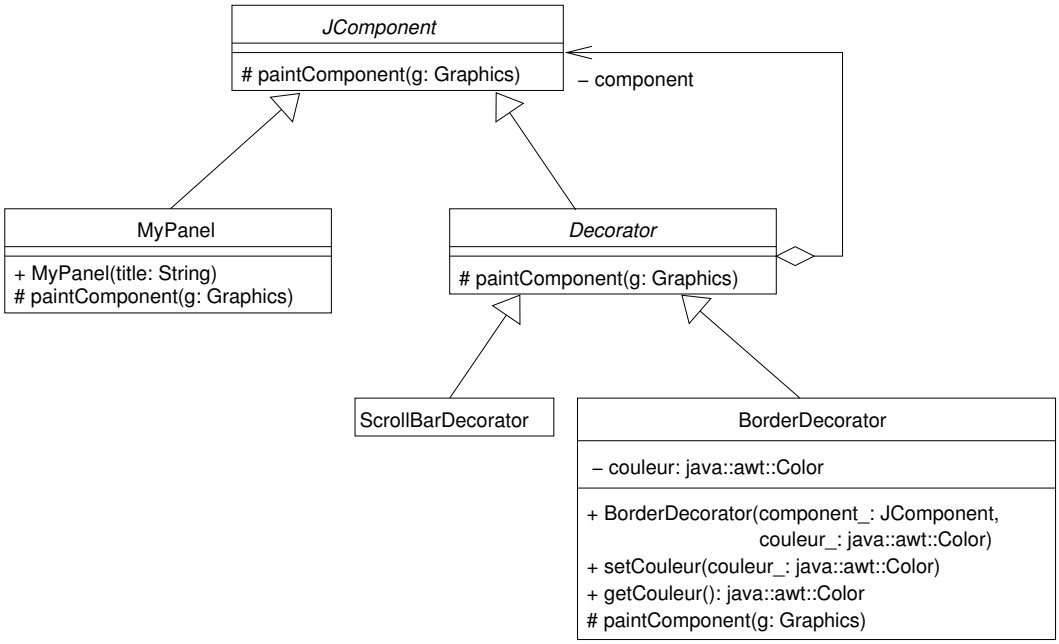

FIG. 7 – Diagramme de classe du décorateur adapté

3. la solution proposée ici permet-elle de résoudre les problèmes soulevés en section 2.2 ?

   La solution proposée par le Décorateur permet de résoudre un certain nombres de problèmes. En particulier, on obtient un système très souple, puisque pour ajouter une fonctionnalité, il suffit d'étendre la classe `Decorator`. On pourrait ainsi avoir un décorateur pour la bordure, un pour les ascenseurs etc. On peut également utiliser une décoration deux fois : par exemple, on peut créer un décorateur avec bordure sur un composant déjà décoré.

Enfin, le fait de ne pas dépendre d'une hiérarchie de classes « statique » permet d'ajouter et d'enlever des comportements dynamiquement.

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 ?

   La relation d'héritage entre `Component` et `Decorator` permet de garantir qu'un composant décoré reste toujours un composant. L'héritage n'est pas ici un héritage de code, mais un héritage de type. On aurait donc très bien pu utiliser une interface pour `Component`.

5. pourquoi `Decorator` est-elle abstraite ?

   `Decorator` est abstraite car on ne doit pas pouvoir créer de décorateur simple. Il faut effectivement que le décorateur ajoute un comportement à la classe décorée pour qu'il soit une classe concrète. De plus, on n'aurait pas pu utiliser une interface, car il faut que `Decorator` soit liée via une association à un composant.

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.

   Voici la méthode statique. On suppose évidemment qu'il faut également un `JComponent` en paramètre du constructeur de `ScrollBarDecorator`.

```
    public static JComponent addBorderAndScroll(JComponent component) {
        return new BorderDecorator(new ScrollBarDecorator(component),
                                   java.awt.Color.RED);
    }
```

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`.

   Voici le code de la classe `Decorator` :

```
package fr.supaero.gui;

import javax.swing.JComponent;
import java.awt.Graphics;

/**
 * <code>Decorator</code> est une classe abstraite permettant de
 * décorer une instance de <code>JComponent</code>.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public abstract class Decorator extends JComponent {
```

```
    private JComponent component;

    /**
     * Initialiser une instance de <code>Decorator</code>.
     *
     * @param component_ le <code>JComponent</code> à décorer
     */
    public Decorator(JComponent component_) {
        this.component = component_;
    }

    protected void paintComponent(Graphics g) {
        this.component.paintComponent(g);
    }
}
```

Rien de bien difficile dans cette classe. Il fallait juste faire attention aux paquetages[2].
Voici le code de la classe `BorderDecorator` :

```
package fr.supaero.gui;

import javax.swing.JComponent;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;

public class BorderDecorator extends Decorator {

    private Color couleur;

    public BorderDecorator(JComponent component_,
                           Color couleur_) {
        super(component_);
        this.couleur = couleur_;
    }

    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2d = (Graphics2D) g.create();

        g.setColor(this.couleur);
```

[2]En réalité, l'utilisation d'une méthode déclarée comme **`protected`** pose problème. On peut utiliser une méthode **`protected`** dans une sous-classe tant que l'appel à cette méthode se fait sur un objet dont la sous-classe définit l'implantation. L'appel à **`this`**`.component.paintComponent(g)` ne fonctionne donc pas !

```
        g.drawRect(0, 0, this.getWidth(), this.getHeight());

        g2d.dispose();
    }

    public void setCouleur(Color couleur_) {
        this.couleur = couleur_;
    }

    public Color getCouleur() {
        return this.couleur;
    }
}
```

8. si la méthode `paintComponent` de `JComponent` peut lever une exception, doit-on la traiter ? Si oui, comment ?

   Si la méthode `paintComponent` peut lever une exception, elle sera signalée dans la signature de la méthode via la clause **`throws`**. On sera donc *obligé* lors de la redéfinition de la méthode de signaler également que l'exception peut être levée.
   Dans notre cas, l'exception peut être levée lors de l'appel de `component.paintComponent(g)`. Si une exception est levée lors de cet appel, on n'a aucun moyen de la traiter localement. On laissera donc le mécanisme de propagation propager l'exception.

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 5) 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[3].

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.

   La figure 8 propose un diagramme de classes d'analyse répondant à la question. Vous trouverez dans [4] une présentation complète du méta-modèle (document sur la superstructure d'UML).

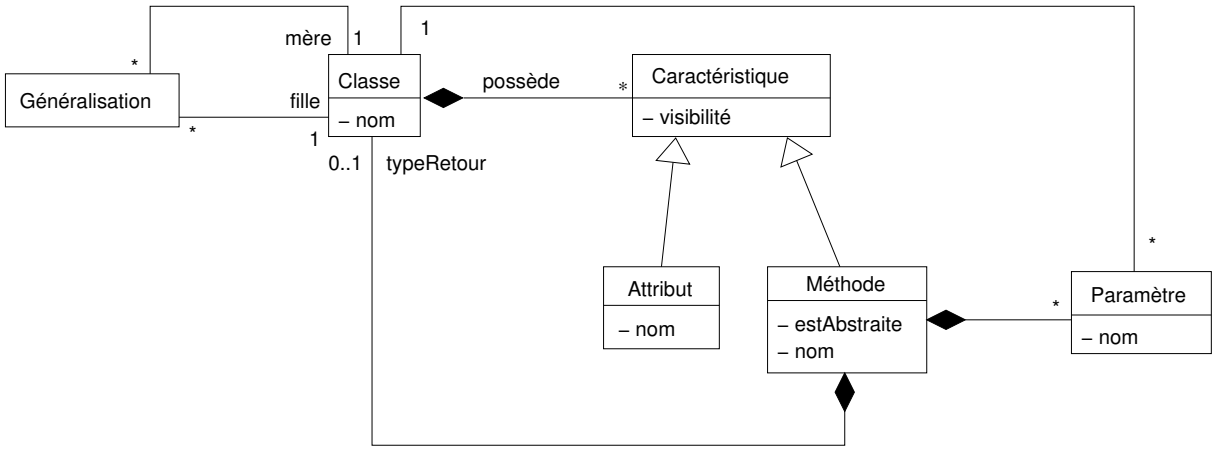

FIG. 8 – Diagramme de classes modélisant les concepts objet

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.

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

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[4] 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`.

C'était la question la plus difficile de l'examen, elle demandait d'être très rigoureux dans la programmation. Voici le code de la méthode `findMethods` (j'ai supposé avoir importé l'ensemble des paquetages `java.util` et `java.lang.reflect`) :

```
public static ArrayList findMethods(Object obj,
                                    String name,
                                    Object[] args) throws Exception {

    // On trouve la classe de obj
    Class c = obj.getClass();

    // Préparation de la liste contenant la hiérarchie de classe
    // de obj
    ArrayList classes = new ArrayList();
    classes.add(c);

    // On trouve les super-classes de c
    while ((c = c.getSuperclass()) != null) {
        classes.add(c);
    }

    // On trouve les types des paramètres dans args. On suppose
    // que args a la bonne longueur...
    Class[] tabArgs = new Class[args.length];
    for (int i = 0; i < args.length; i++) {
        tabArgs[i] = args[i].getClass();
    }


    // On cherche les méthodes possédant la bonne signature dans
    // classes
    ArrayList methods = new ArrayList();
    Method lastMethod = null;
    for (int i = 0; i < classes.size(); i++) {
        c = (Class) classes.get(i);
        try {
            Method m = c.getMethod(name, tabArgs);
```

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

```
                // On vérifie qu'on n'a pas déjà la méthode
                if (!m.equals(lastMethod)) {
                    lastMethod = m;
                    methods.add(lastMethod);
                }
            } catch (NoSuchMethodException e) {
                // Si on est là, puisque les premières classes situées
                // dans la liste sont celles qui sont les plus basses
                // dans la hiérarchie, on peut sortir de la boucle.
                break;
            }
        }

        // La liste de méthodes est vide ? On lève une exception.
        if (methods.size() == 0) {
            throw new Exception("Problème avec la méthode." +
                                "Vérifiez la signature !");
        }

        // On renvoie la liste
        return methods;
    }
```

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] Object Management Group. Unified Modeling Language (UML), version 2.0. http://www.omg.org/technology/documents/formal/uml.htm.

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