IN201 : Corrigé de l'examen

SUPAERO 2A

Christophe Garion <garion@isae.fr>

1er mars 2010

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. L'exercice 1 est un exercice plutôt technique et devient difficile à partir de la question 3. L'exercice 2 étudie un patron de conception particulier. L'exercice 3 est un exercice de modélisation avec UML. Chaque exercice sera noté sur 8 points, mais le barème final peut être soumis à de légères modifications.

Remarque importante : dans tous les exercices, il sera tenu compte de la syntaxe UML lors de l'écriture de diagrammes. Ne perdez pas des points bêtement.

1 Ajout dynamique de fonctionnalités à un objet via un proxy

Cet exercice est inspiré de [5].

La date de remise de votre BE approche. Obnubilé(e) par votre grande passion, une tentative de résolution analytique des équations de Navier-Stokes en 3D (cf. figure 1), vous avez laissé votre binôme s'occuper de l'implantation du BE.

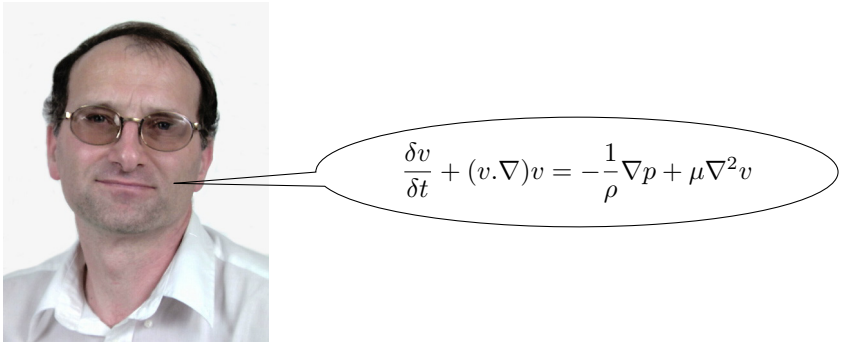

FIGURE 1 : Un aérodynamicien sévissant à SUPAERO

Malheureusement, ce dernier a profité des vacances de février pour faire une mauvaise chute à ski et se trouve actuellement dans l'incapacité d'utiliser ses bras ou de communiquer avec vous, sa machoire étant immobilisée. Un malheur n'arrivant jamais seul, il avait perdu son ordinateur portable dans le train et il ne le récupérera que deux jours avant la date de remise du BE.

Il faut donc que vous vous débrouillez seul(e) pour avancer au maximum le BE avant son retour. Vous savez que le BE était pratiquement fini, mais que votre binôme avait détecté un problème : il semblerait que l'algorithme de plus court chemin soit appelé plus que de raison lors du déroulement de la simulation et vous vous souvenez vaguement que votre binôme souhaitait vérifier quand l'algorithme était appelé et quelle était sa durée d'exécution. Malheureusement (ter), votre ~~andouille de~~ binôme ne vous a laissé qu'une archive contenant des *bytecodes* et vous ne possédez pas les sources du BE. La tâche s'annonce ardue...

Vous avez toutefois pu récupérer un programme de test (cf. listing 1) et un jeu de fichiers de configuration qui devaient lui servir à tester la simulation (sans lancer l'interface graphique).

Listing 1 : Le programme de test

```
1  package fr.supaero.simu;
2  import fr.supaero.pathfinding.*;
3
4  public class TestSimulateurModeTexte {
5      public static void main(String[] args) {
6          // lors du lancement, l'argument en ligne de commande est le
7          // nom du fichier de configuration du test
8
9          Simulateur s = new Simulateur(new PCCDijkstra(), args[0]);
10
11         System.out.println("hey ho, let's go");
12         s.go();
13         System.out.println("this is the end...");
14     }
15 }
```

Vous pouvez donc lancer la simulation en donnant un des fichiers de configuration de test en paramètre et préciser l'algorithme de plus court chemin utilisé en modifiant le programme de test. En fouillant la javadoc de l'API de Java sur le site de Sun, vous avez également trouvé que la classe `System` possède une méthode statique `currentTimeMillis` qui renvoie un entier représentant la date courante. Votre tâche va donc consister à trouver un moyen d'afficher (via `System.out.println`) un message du type « `appel a PCCDijkstra : XXXXX` » avant chaque appel à l'algorithme de plus court chemin et un message du type « `fin d'appel a PCCDijkstra : XXXXX` » après chaque appel à l'algorithme de plus court chemin (`XXXXX` représentant à chaque fois le résultat de l'appel à `currentTimeMillis`). On appelera ces deux affichages un *log*.

1. on ne considère que la classe `PCCDijkstra`. Cette classe possède une méthode `plusCourtChemin()` qui ne renvoie rien et calcule le plus court chemin d'un point à un autre d'un graphe. Proposer en Java une solution simple permettant d'avoir un log pour `PCCDijkstra`.

   On cherche ici à modifier le comportement d'une classe dont on ne peut pas modifier le code. Même si le fait de ne pas disposer du code de `PCCDijkstra` était particulier, nous avons vu en cours que la *spécialisation* permettait de modifier le comportement d'une classe en redéfinissant les méthodes que l'on souhaite modifier. On peut bien entendu spécialiser une classe pour laquelle on ne dispose pas du code source. La classe `PCCDijkstraLog` est présentée sur le listing 2.

   Il suffisait ensuite d'utiliser une instance de `PCCDijkstraLog` dans le programme de test.

   Pour modifier le comportement d'une méthode d'un objet, on peut également encapsuler cet objet dans un autre objet avec la même interface et utiliser l'objet englobant pour ajouter les logs par exemple. Cette solution est proposée par le patron de conception Décorateur présenté dans la question suivante, mais on pouvait l'utiliser dès cette question si vous en aviez eu l'idée.

2. en retrouvant dans vos documents votre rapport de conception, vous vous rendez compte que vous aviez prévu d'utiliser plusieurs algorithmes de plus court chemin. Et il semble que votre binôme a effectivement construit plusieurs classes représentant des algorithmes de plus court chemin et a pensé à utiliser une interface.

   Vous obtenez ainsi le diagramme UML présenté sur la figure 2.

   Il faut donc maintenant pouvoir générer des logs pour les autres classes de plus court chemin développées.

Listing 2 : La classe `PCCDijkstraLog`

```
1  package fr.supaero.pathfinding;
2
3  public class PCCDijkstraLog extends PCCDijkstra {
4
5      public void plusCourtChemin() {
6          System.out.println("Appel a PCCDijkstra : " + System.currentTimeMillis());
7          super.plusCourtChemin();
8          System.out.println("Fin d'appel a PCCDijkstra : " + System.currentTimeMillis());
9      }
10 }
```

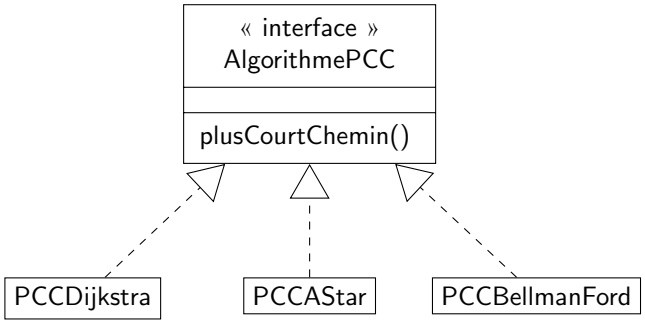

FIGURE 2 : Les différents algorithmes de plus court chemin utilisés

(a) est ce que la solution développée précédemment peut encore convenir ? On réfléchira en particulier à la possibilité d'avoir de nombreuses classes implantant des algorithmes de plus court chemin.

La solution développée précédemment convenait bien lorsque l'on travaillait avec une seule classe. Avec plusieurs classes représentant des algorithmes de plus court chemin, on va être obligé de créer autant de sous classes qu'il y a de classes. De plus, le code écrit dans chaque sous-classe est pratiquement le même. La solution précédente n'est donc pas forcément très judicieuse.

(b) pour pallier le problème soulevé dans la question précédente, vous décidez après lecture de [4] d'utiliser le patron de conception décorateur. Ce patron permet de changer le comportement d'un objet en l'encapsulant dans un autre objet. Un diagramme de classe présentant le décorateur est donné sur la figure 3 et un diagramme de séquence expliquant son fonctionnement est donné sur la figure 4.

i. instancier le diagramme de classes précédent pour les classes représentant des algorithmes de plus court chemin et un décorateur particulier, `DecorateurLog` ;

Rien de bien difficile ici, il suffisait de bien comprendre que le décorateur que l'on cherche à faire va permettre d'ajouter les logs aux classes d'algorithme de plus court chemin. Le diagramme de classes est présenté sur la figure 5.

ii. écrire en Java la classe `DecorateurLog` ;

La classe `DecorateurLog` est présentée sur le listing 3. Il ne fallait pas oublier d'introduire l'algorithme de plus court chemin à décorer comme attribut et de réaliser l'interface

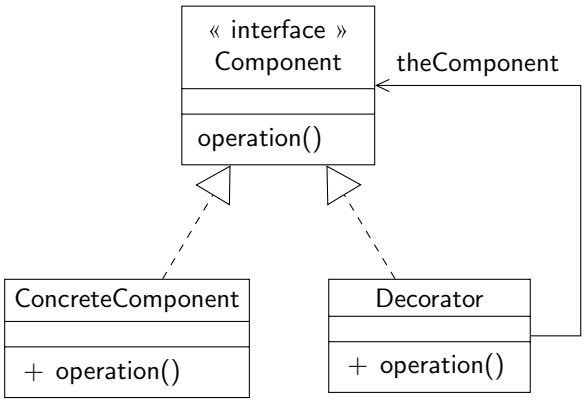

FIGURE 3 : Le patron de conception Décorateur

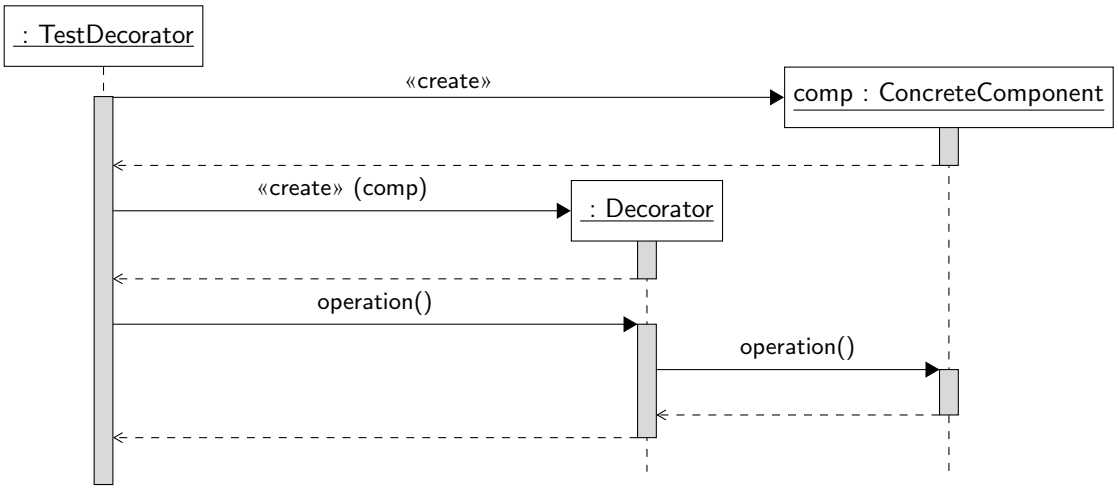

FIGURE 4 : Un scénario d'utilisation du décorateur

`AlgorithmePCC`. On remarquera que l'on ne sait pas (pour l'instant) comment afficher dans le décorateur quelle est la classe de l'algorithme utilisé.

iii. réécrire le programme de test en utilisant le décorateur (on ne réécrira pas les lignes 10 à 13 du programme de test) ;

Rien de bien difficile, il suffisait d'encapsuler l'instance de `PCCDijkstra` qui est passée au simulateur dans un décorateur. Le code source est présenté sur le listing 4.

3. en fouillant un peu sur le site d'IBM DeveloperWorks, vous tombez sur un article de Brian Goetz sur les *proxies* dynamiques. Un *proxy* est un représentant d'un objet : il permet d'« intercepter » les appels à une ou plusieurs méthodes d'un objet, d'effectuer un traitement puis d'appeler la méthode sur l'objet et de récupérer le résultat avant de le renvoyer. Vous découvrez dans l'article que Java propose déjà une classe permettant de faire des *proxies* dynamiques. On pourrait donc créer un *proxy* pour un algorithme de plus court chemin, créer les logs grâce au *proxy* tout en effectuant réellement le calcul du plus court chemin.

   Dans ce qui suit, on appelera objet original l'objet à partir duquel on a créé le *proxy* et méthode

Listing 3 : La classe `DecorateurLog`

```
1  package fr.supaero.pathfinding;
2
3  public class DecorateurLog implements AlgorithmePCC {
4
5      private AlgorithmePCC theAlgorithm;
6
7      public DecorateurLog(AlgorithmePCC theAlgorithm) {
8          this.theAlgorithm = theAlgorithm;
9      }
10
11     public void plusCourtChemin() {
12         System.out.println("Appel a l'algorithme : " + System.currentTimeMillis());
13         this.theAlgorithm.plusCourtChemin();
14         System.out.println("Fin d'appel a l'algorithme : " + System.currentTimeMillis());
15     }
16 }
```

Listing 4 : Le programme de test modifié

```
1  package fr.supaero.simu;
2  import fr.supaero.pathfinding.*;
3
4  public class TestSimulateurDecorateur {
5      public static void main(String[] args) {
6          // lors du lancement, l'argument en ligne de commande est le
7          // nom du fichier de configuration du test
8
9          Simulateur s = new Simulateur(new DecorateurLog(new PCCDijkstra()), args[0]);
10
11         System.out.println("hey ho, let's go");
12         s.go();
13         System.out.println("this is the end...");
14     }
15 }
```

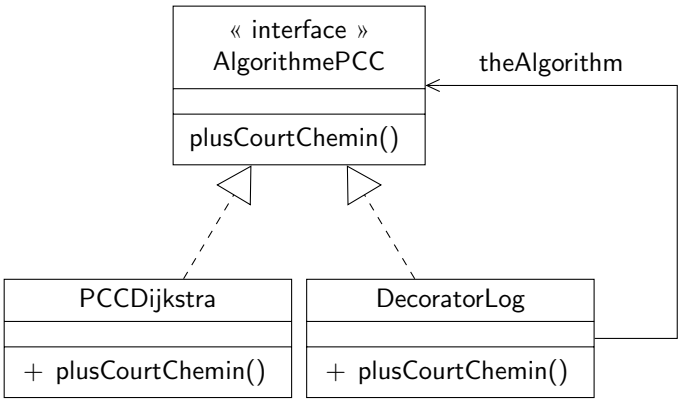

FIGURE 5 : Le patron de conception Decorateur adapté à notre problème

originale une méthode que l'on appelle sur le *proxy* (et qui doit donc nécessairement être une méthode de l'objet original).

Toutes les nouvelles classes et interfaces présentées dans cette question appartiennent au paquetage `java.lang.reflect`.

(a) en vous plongeant plus en détail dans l'article, vous découvrez que le mécanisme de *proxy* dynamique repose sur la notion d'*invocation handler*. Un *invocation handler* est une classe réalisant l'interface `InvocationHandler` qui possède une seule méthode dont la signature est la suivante :

```
Object invoke(Object proxy, Method method, Object[] args) throws Throwable
```

À chaque fois qu'on appelle une méthode sur le *proxy*, la méthode `invoke` de l'*invocation handler* associé au *proxy* est appelée. C'est donc dans cette méthode que l'on va faire les traitements supplémentaires à ajouter à la méthode de l'objet original. Les paramètres de la méthode `invoke` sont créés lors de l'appel de la méthode sur le *proxy* et ont la signification suivante :

- `proxy` représente le *proxy* sur lequel on a appelé la méthode originale
- `method` est un objet de type `Method` représentant la méthode originale [1]. `Method` est une classe dont les instances sont des objets représentant des méthodes. Elle possède une méthode permettant d'exécuter la méthode représentée sur un objet particulier :

  ```
  public Object invoke(Object obj, Object[] args) throws Exception
  ```

  où `obj` est l'objet sur lequel appeler la méthode et `args` les arguments de la méthode. Les exceptions qui peuvent être levées le sont lorsque par exemple l'objet passé en paramètre n'est pas du bon type et n'a pas la méthode représentée par l'objet de type `Method`.
  Un appel d'une méthode `m` sur un objet `o` avec les paramètres `arg1`, ..., `argn` devient donc `oM.invoke(o, args)` si `oM` est un objet de type `Method` représentant `m` et `args` un tableau contenant `arg1`, ..., `argn`.
- `args` représente les arguments de la méthode
- un objet de type `Throwable` peut être propagé par la méthode en cas de problème.

`Method` possède également une méthode `getName` qui renvoie le nom de la méthode sous forme d'un objet de type `String`.

1. On peut donc en Java représenter les méthodes par des objets. On verra que l'on peut également représenter les classes par des objets. C'est ce que l'on appelle la *réflexion*.

Listing 5 : L'*invocation handler* pour les algorithmes de plus court chemin

```java
package fr.supaero.pathfinding;
import java.lang.reflect.*;

public class LogInvocationHandler implements InvocationHandler {

    private AlgorithmePCC algo;

    public LogInvocationHandler(AlgorithmePCC algo) {
        this.algo = algo;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        if (method.getName().equals("plusCourtChemin")) {
            System.out.println("Appel a plusCourtChemin : " + System.currentTimeMillis());
            method.invoke(this.algo, args);
            System.out.println("Fin d'appel a plusCourtChemin : " + System.currentTimeMillis());
        }

        return null;
    }
}
```

Écrire une classe `LogInvocationHandler` représentant un *invocation handler* pour un algorithme de plus court chemin. On réfléchira en particulier aux attributs nécessaires à la classe.

Le code source de l'*invocation handler* est présenté sur le listing 5.

Avant de se précipiter sur le code, il fallait effectivement réfléchir à ce dont on avait besoin lors de l'écriture de la classe. L'*invocation handler* permet d'intercepter les appels sur le *proxy* et donc de réaliser le *log*. Par contre, il faut connaître l'objet original pour pouvoir appeler la méthode `plusCourtChemin` sur ce dernier. C'est donc le seul attribut dont on a besoin.

J'ai choisi de placer la classe dans le paquetage `fr.supaero.pathfinding` car elle ne concerne a priori que ces algorithmes.

L'écriture de la méthode `invoke` pour l'*invocation handler* devait être faite pas à pas. Tout d'abord, on peut afficher le *log*, cela ne pose pas de problème particulier. Reste à appeler la méthode `plusCourtChemin` sur l'objet original. Pour cela, on utilise le paramètre `method` de `invoke`. L'appel se fait en utilisant les indications données dans l'énoncé. Enfin, on remarque que la méthode `invoke` de l'*invocation handler* doit renvoyer un objet représentant le résultat de l'appel à la méthode. Comme la méthode qui nous intéresse ici, `plusCourtChemin`, ne renvoie rien, je renvoie `null`.

On remarquera finalement que je teste si la méthode appelée est bien celle qui m'intéresse, i.e. `plusCourtChemin` (ceci pour éviter que des *logs* soient faits pour d'autres méthodes). Cette vérification n'était pas forcément nécessaire.

(b) on va maintenant chercher à construire un *proxy* dynamique pour un algorithme de plus court chemin en utilisant le *handler* précédemment développé. Pour cela, on va utiliser la méthode *statique* de la classe `Proxy` suivante :

Listing 6 : Le programme de test utilisant l'*invocation handler* pour les algorithmes de plus court chemin

```
1  package fr.supaero.simu;
2  import fr.supaero.pathfinding.*;
3  import java.lang.reflect.*;
4
5  public class TestSimulateurLogInvocationHandler {
6      public static void main(String[] args) {
7          // lors du lancement, l'argument en ligne de commande est le
8          // nom du fichier de configuration du test
9
10         PCCDijkstra algo = new PCCDijkstra();
11
12         Simulateur s = new Simulateur((AlgorithmePCC)
13                                 Proxy.newProxyInstance(algo.getClass().getClassLoader(),
14                                                        new Class<?>[] {fr.supaero.pathfinding.A
15                                                        new LogInvocationHandler(algo)),
16                                 args[0]);
17
18         System.out.println("hey ho, let's go");
19         s.go();
20         System.out.println("this is the end...");
21     }
22 }
```

```
Object newProxyInstance(ClassLoader loader,
                        Class<?>[] interfaces,
                        InvocationHandler h)
```

où

- `loader` est ce que l'on appelle un *class loader*. On peut l'obtenir en appelant `getClass().getClassLoader()` sur un objet. *Vous n'avez pas besoin d'en savoir plus.*
- `interfaces` est une tableau d'objets de type `Class` représentant les interfaces que l'on veut que le *proxy* réalise. Dans notre cas, on ne s'intéresse qu'à une interface, `AlgorithmePCC`.
  `Class` est une classe générique dont les instances représentent des classes. Ainsi, `Class<AlgorithmePCC>` est une classe dont les instances représentent l'interface `AlgorithmePCC`.
  On rappelle que pour créer et instancier un tableau dans une même expression, on peut écrire `new int[] {1, 2 , 3 , 4}` (l'exemple pris ici est celui d'un tableau d'entiers contenant les valeurs 1, 2, 3 et 4 dans cet ordre).
- le type de retour de `newProxyInstance` est `Object` (attention !)

Réécrire le programme de test pour `PCCDijkstra` en créant un *proxy* dynamique permettant la création de logs.

Le code source de la classe `TestSimulateurLogInvocationHandler` est présenté sur le listing 6.

Rien de bien particulier sur cette question qui était toutefois assez technique. J'ai utilisé un objet de type `PCCDijkstra` pour pouvoir appeler `getClass` pour obtenir un *class loader*. La seule interface qui m'intéresse est `AlgorithmePCC` que l'on représente par

Listing 7 : Un *invocation handler* générique

```
1  package fr.supaero.proxy;
2  import java.lang.reflect.*;
3
4  public class LogInvocationHandler<T> implements InvocationHandler {
5
6      private T originalObj;
7
8      public LogInvocationHandler(T originalObj) {
9          this.originalObj = originalObj;
10     }
11
12     public Object invoke(Object proxy, Method method, Object[] args)
13         throws Throwable {
14         System.out.println("Appel a " + method.getName() + " : " + System.currentTimeMillis());
15         Object ret = method.invoke(this.originalObj, args);
16         System.out.println("Fin d'appel a " + method.getName() + " : " + System.currentTimeMillis());
17
18         return ret;
19     }
20 }
```

`fr.supaero.pathfinding.AlgorithmePCC.class`.

(c) après tout, ce mécanisme de *proxy* dynamique est bien intéressant. Pourquoi le limiter aux classes réalisant l'interface `AlgorithmePCC` ?

i. écrire un *invocation handler* générique permettant de générer des logs pour n'importe quel type d'objet ;

J'ai choisi de placer toutes les classes dans un paquetage `fr.supaero.proxy`.

Pour créer un *invocation handler* générique, on va s'inspirer de la classe `LogInvocationHandler` développée précédemment. Il faut comme pour `LogInvocationHandler` que l'on ait comme attribut l'objet original sur lequel appeler la méthode. Cette fois-ci par contre, on utilisera un paramètre de type générique pour typer cet objet. De la même façon, la liste d'interfaces qui nous intéressera ne sera constituée que du type générique.

Le code source de la classe `LogInvocationHandler` est présenté sur le listing 7.

ii. écrire une méthode statique générique `creerLogProxy` qui prend en paramètre :
- un objet dont on veut créer un *proxy* permettant de générer des logs ;
- un objet de type `Class` correctement paramétré représentant l'interface que l'on veut considérer pour le *proxy*.

Le code source de la classe `LogProxyFactory` est présenté sur le listing 8. Rien de bien particulier, on pourra juste remarquer que le transtypage du retour de `newProxyInstance` n'est pas sûr et on a un *warning* à la compilation.

Le lecteur intéressé par ce genre de problématique (ajout de fonctionnalités à la volée en particulier) pourra se référer à la programmation orientée aspects [6, 1].

Listing 8 : Une *factory* pour *invocation handler*

```
1  package fr.supaero.proxy;
2  import java.lang.reflect.*;
3
4  public class LogProxyFactory {
5
6      public static <T> T creerLogProxy(T originalObj, Class<T> intf) {
7          return (T) Proxy.newProxyInstance(originalObj.getClass().getClassLoader(),
8                                            new Class[] { intf },
9                                            new LogInvocationHandler<T>(originalObj));
10     }
11 }
```

2 Patron de conception Commande

Cet exercice est largement inspiré d'un chapitre de [3].

On cherche ici à construire une application domotique qui permette à un utilisateur de contrôler différents services pour sa maison ou son appartement. Par exemple, il pourra, à partir de cette application, contrôler les lumières de la maison, la chaîne HiFi, la porte du garage, l'alarme, le thermostat etc. L'application à concevoir sera sous la forme d'une interface graphique qui possède les propriétés suivantes :

- elle possède un certain nombre de couples de boutons On/Off qui permettent de mettre en marche ou d'arrêter un service ;
- l'utilisateur peut choisir quel service associer à quel couple de boutons ;
- il y a un bouton « Undo » qui permet d'annuler la dernière action (appui sur un bouton) réalisée. On ne peut annuler qu'une seule action.

L'aspect rudimentaire de l'interface graphique [2] (cf. figure 6) est justifié par le fait que l'on cherche également à produire un contrôleur physique qui ne pourra posséder que ces boutons On/Off.

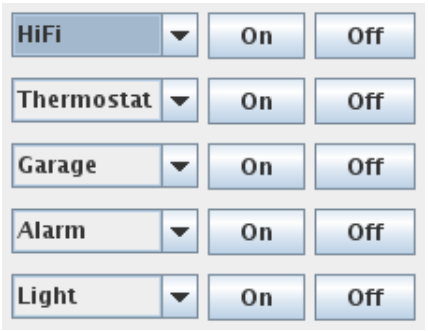

FIGURE 6 : L'interface graphique développée

On suppose que l'on a défini un modèle pour cette interface graphique représenté par une classe `RemoteControl` qui est chargée de lier les boutons aux différents services. C'est cette classe que nous allons étudier plus en détail.

On nous fournit un certain nombre de classes permettant de contrôler différents services. Ces classes ont été développées par différents fournisseurs et quelques-unes d'entre elles sont représentées sur la

2. On pourrait par exemple avoir une interface graphique qui ait des contrôles plus complexes : *slider* permettant de régler la température si le service choisi concerne la thermostat etc.

figure 7. On suppose également que l'on va pouvoir ajouter d'autres classes provenant de fournisseurs extérieurs dans le futur.

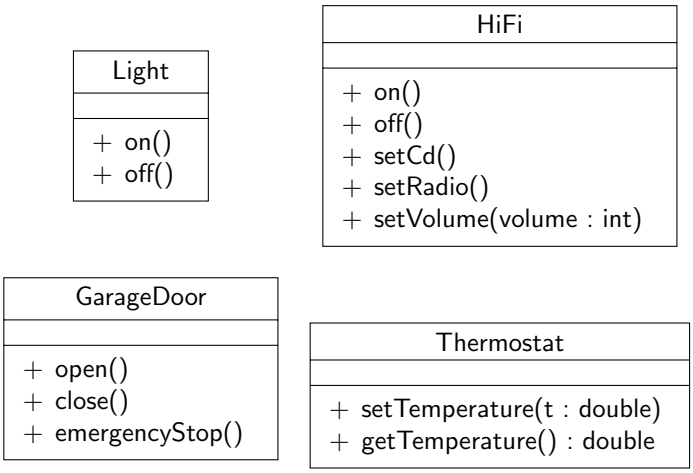

FIGURE 7 : Quelques-unes des classes fournies

Les objectifs de l'exercice sont donc :

1. de pouvoir placer des services « quelconques » sur les boutons
2. de gérer un bouton d'« undo »

1. si ces classes étaient modifiables, quelle solution simple proposeriez-vous pour résoudre le problème d'affectation des services aux boutons ?

   Si les classes étaient modifiables, il suffirait de créer une interface `ServiceOn` avec une méthode `on` et une interface `ServiceOff` avec une méthode `off` et de faire réaliser ces deux interfaces par tous les services. On disposerait donc pour chaque service d'une méthode `on` et d'une méthode `off` que l'on pourrait utiliser sur les boutons de la télécommande.

2. on suppose dans toute la suite que les classes sont fournies sous forme de *bytecode* et donc *non modifiables*. Il nous faut donc trouver un moyen de pouvoir assigner des actions à chaque couple de boutons de l'interface graphique sans avoir un ancêtre commun à toutes les classes représentant les services. Pour cela, nous allons utiliser un patron de conception particulier : Commande [4].

   Le patron de conception Commande permet d'encapsuler dans un objet une requête sur un objet *récepteur*, c'est-à-dire un ou plusieurs appels de méthode(s) sur cet objet récepteur, et d'appeler ensuite cette requête au moment opportun. Il est présenté sur la figure 10.

   L'idée du patron est la suivante : on veut pouvoir « charger » un *invoker* avec une requête sur un objet dit *receiver*. L'*invoker* n'est pas obligé de connaître le *receiver* réel qu'il va utiliser. Le chargement de la requête est fait par un *client*. Attention, le client est juste là pour charger l'*invoker* avec la requête. Ce n'est pas forcément lui qui activera la requête via l'*invoker*.

   Le fonctionnement du patron est le suivant : le client crée un objet de type `Command` qui utilise une instance particulière de `Receiver`. L'objet de type `Command` contient donc toutes les informations pour exécuter un certain nombre d'actions sur l'instance de `Receiver`. Le client appelle alors `setCommand` sur une instance d'`Invoker` pour lier cette dernière avec la commande créée. L'instance d'`Invoker` appelle alors *à un certain moment*[3] la méthode `execute` de la commande.

3. Ce moment peut être déterminé par l'instance d'`Invoker` elle-même ou par un appel extérieur à une méthode de cette même instance, comme par exemple l'appui sur un bouton d'une interface graphique...

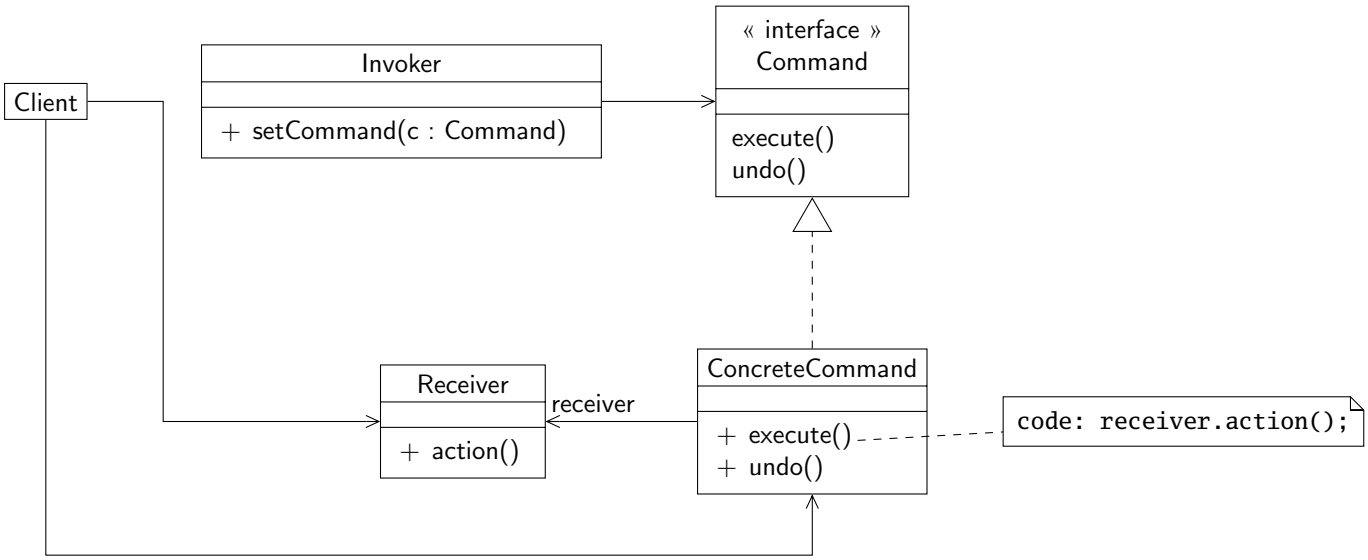

FIGURE 8 : Le patron de conception Commande

Écrire un diagramme de séquence représentant la séquence d'actions précédente.

Le diagramme de séquence est représenté sur la figure 9. Il ne fallait pas oublier que `Command` est une interface et donc qu'il fallait utiliser `ConcreteCommand` pour instancier un objet.

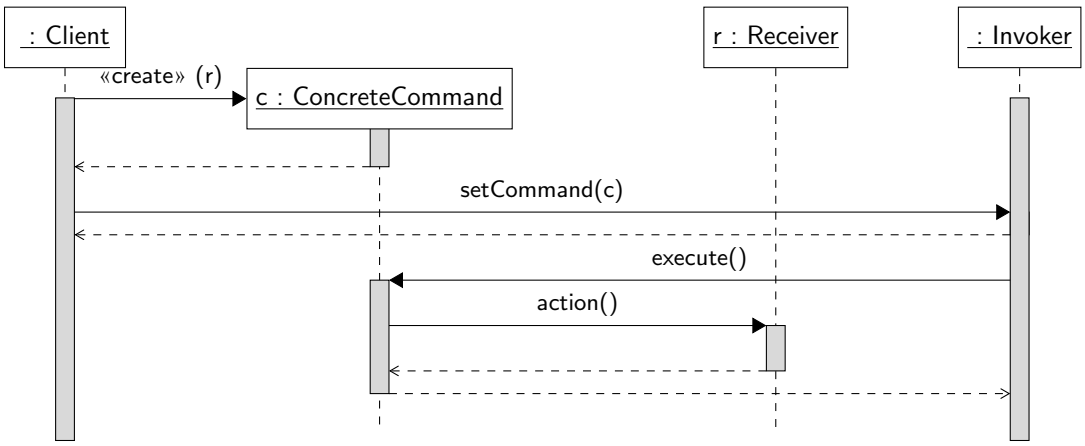

FIGURE 9 : Diagramme de séquence représentant le scénario d'utilisation du patron Commande

3. adapter le diagramme de classes pour une commande qui permet d'allumer la lampe. On suppose que l'on ne possède pas de classe représentant un modèle des boutons. On réfléchira en particulier au rôle joué par la classe `RemoteControl` dans ce patron. Qui joue le rôle d'*invoker* ici ?

   On ne s'intéresse ici qu'à une commande qui permet d'allumer la lampe. L'objet *receiver* sera donc une instance de la classe `Light` qui possède l'action concrète permettant d'allumer la lampe. Le client est en fait le client (humain ou non) qui va associer la commande à un bouton donné. Donc l'*invoker* va être un des boutons de la télécommande. Comme on n'a pas a priori accès à une classe

Listing 9 : L'interface `Command`

```
1  package fr.supaero.command;
2
3  public interface Command {
4
5      void execute();
6
7      void undo();
8  }
```

représentant les boutons, on propose de mettre `RemoteControl` comme *invoker* ici. Le diagramme de classes correspondant est présenté sur la figure

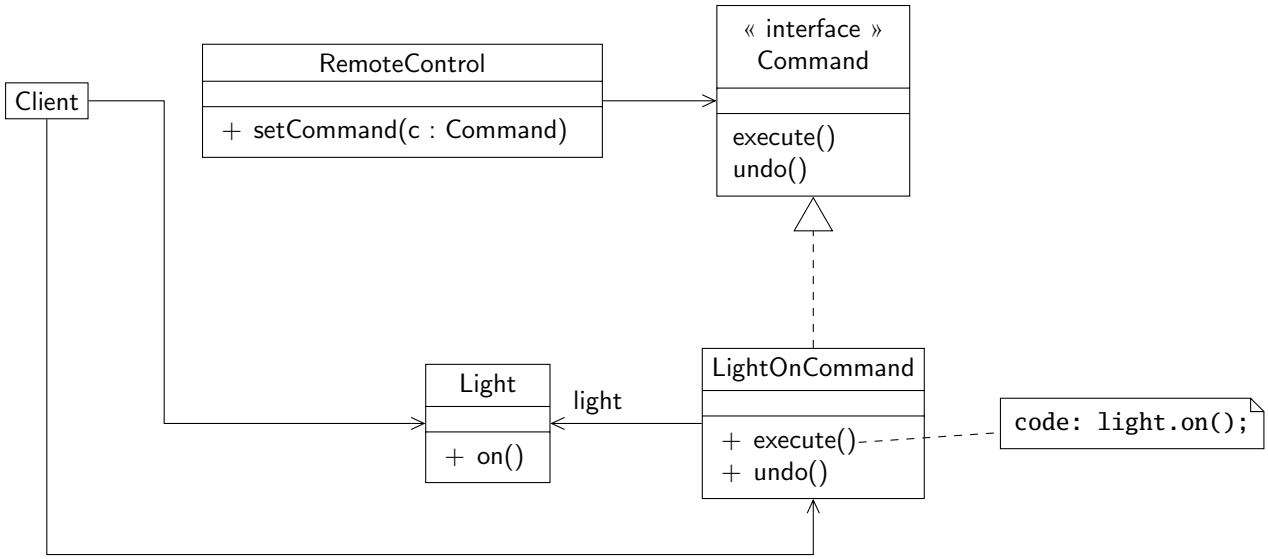

FIGURE 10 : Le patron de conception Commande adapté à notre problème

4. écrire en Java l'interface `Command` et la classe `LightOnCommand`.

   Le code source de l'interface `Command` est présenté sur le listing 9. Rien de particulier ici.
   Le code source de la classe `LightOnCommand` est présenté sur le listing 10. Il ne fallait pas oublier de réaliser l'interface et de définir un constructeur qui permettait de lier la classe avec la classe `Light`. La méthode `undo` ne fait qu'éteindre la lampe.
5. écrire en Java la classe `RemoteControl` pour utiliser le patron de conception. On supposera que l'on a un nombre limité *et connu* (par exemple 5) de modèles des boutons On et Off. On stockera dans deux tableaux (un pour les boutons On, un pour les boutons Off) les commandes associées à chaque bouton. `RemoteControl` permet donc d'encapsuler plusieurs commandes. On supposera également que l'on dispose de méthodes dans cette classe pour simuler l'appui sur un bouton particulier :
   - une méthode `buttonOnPressed(int i)` qui permet de signifier que le ième bouton de type On a été appuyé ;
   - une méthode `buttonOffPressed(int i)` qui permet de signifier que le ième bouton de type Off a été appuyé

Listing 10 : La classe `LightOnCommand`

```
1  package fr.supaero.command;
2
3  public class LightOnCommand implements Command {
4
5      private Light light;
6
7      public LightOnCommand(Light light) {
8          this.light = light;
9      }
10
11     public void execute() {
12         this.light.on();
13     }
14
15     public void undo() {
16         this.light.off();
17     }
18 }
```

On n'oubliera pas de tenir compte du bouton « Undo » et on vérifiera qu'une commande est bien associée avec le bouton sur lequel on appuie.

La classe `RemoteControl` permet en fait d'encapsuler tous les *invokers* qui sont les boutons de la télécommande. Le code source de `RemoteControl` est présenté sur le listing 11.

Il n'y avait pas trop de difficulté pour écrire la classe, on peut toutefois souligner les points suivants :

- pour vérifier que l'on pouvait bien appuyer sur un bouton sans provoquer d'erreur à l'exécution, il fallait tester que la référence associée au bouton sur lequel on appuyait n'était pas **`null`** ;
- comme on utilise des tableaux, il faut normalement que l'utilisateur n'utilise pas des valeurs d'index non légales (les tableaux ont 5 éléments ici). J'ai choisi d'utiliser des commentaires javadoc pour le préciser, mais l'utilisation de la programmation par contrat et de préconditions permettait de spécifier correctement les méthodes (on pouvait également traiter le problème précédent via la programmation par contrat) ;
- le bouton Undo est géré simplement en se souvenant non pas de la dernière commande exécutée, mais de la commande à exécuter pour annuler la dernière commande exécutée ;
- la visibilité **`protected`** des attributs `onCommands` et `offCommands` sera expliquée dans la dernière question du sujet.

Listing 11 : La classe `RemoteControl`

```
1  package fr.supaero.command;
2
3  public class RemoteControl {
4
5      protected Command[] onCommands;
6
7      protected Command[] offCommands;
8
```

```
 9      private Command lastCommand;
10
11      public RemoteControl() {
12          this.onCommands = new Command[5];
13          this.offCommands = new Command[5];
14      }
15
16
17      /**
18       * <code>setOnCommand</code> affecte une commande a un bouton On.
19       *
20       * @param i un entier compris entre 0 et 4 representant
21       *          le bouton
22       * @param c la commande que l'on veut affecter
23       */
24      public void setOnCommand(int i, Command c) {
25          this.onCommands[i] = c;
26      }
27
28      /**
29       * <code>setOffCommand</code> affecte une commande a un bouton Off.
30       *
31       * @param i un entier compris entre 0 et 4 representant
32       *          le bouton
33       * @param c la commande que l'on veut affecter
34       */
35      public void setOffCommand(int i, Command c) {
36          this.offCommands[i] = c;
37      }
38
39      /**
40       * <code>buttonOnPressed</code> presse sur un bouton On.
41       *
42       * @param i un entier compris entre 0 et 4 representant
43       *          le bouton
44       */
45      public void buttonOnPressed(int i) {
46          if (this.onCommands[i] != null) {
47              this.onCommands[i].execute();
48              this.lastCommand = this.onCommands[i];
49          }
50      }
51
52      /**
53       * <code>buttonOffPressed</code> presse sur un bouton Off.
54       *
55       * @param i un entier compris entre 0 et 4 representant
56       *          le bouton
57       */
58      public void buttonOffPressed(int i) {
```

```
59          if (this.offCommands[i] != null) {
60              this.onCommands[i].execute();
61              this.lastCommand = this.offCommands[i];
62          }
63      }
64
65      /**
66       * <code>buttonUndoPressed</code> presse sur le bouton Undo.
67       *
68       */
69      public void buttonUndoPressed() {
70          if (this.lastCommand != null) {
71              this.lastCommand.undo();
72          }
73      }
74  }
```

6. proposer une solution simple pour éviter de faire la vérification précédente.

   Pour éviter de faire la vérification précédente, il suffit de créer une commande qui ne fait rien et de l'affecter par défaut à tous les boutons dans le constructeur de `RemoteControl`. Une telle commande est présentée sur le listing 12.

   Listing 12 : La classe `NullCommand`

```
1  package fr.supaero.command;
2
3  public class NullCommand implements Command {
4
5      public void execute() { }
6
7      public void undo() { }
8  }
```

7. on s'intéresse maintenant à la classe `Thermostat`. Quelles sont les deux actions associées à un couple de boutons On/Off qui utilisent le thermostat ? Quel est le problème qui va se poser ? Comment le pallier ?

   Pour la classe `Thermostat`, on peut considérer que l'action associée au bouton On est d'augmenter la température du thermostat et que l'action associée au bouton Off est de diminuer la température du thermostat. Or la classe `Thermostat` n'a pas de méthodes permettant d'augmenter la température ou de la diminuer, mais une méthode permettant d'affecter une température donnée. La solution la plus simple est donc d'utiliser la méthode `getTemperature` pour pouvoir augmenter ou diminuer la température. On stocke également la température du thermostat avant application de la commande pour l'undo. Par exemple, le code source de la classe `ThermostatOnCommand` est donné sur le listing 13.

8. on souhaite pouvoir créer des macros, i.e. des séquences de commandes pour les associer à un couple de boutons. Par exemple, une macro « Party » permettrait d'allumer les lumières et la chaîne HiFi en un seul clic.

   (a) proposer un diagramme de classe avec une classe permettant d'implanter des macros de commandes. Écrire ensuite le code de cette classe.

Listing 13 : La classe `ThermostatOnCommand`

```
1  package fr.supaero.command;
2
3  public class ThermostatOnCommand implements Command {
4
5      private Thermostat thermostat;
6
7      private double lastTemperature;
8
9      public ThermostatOnCommand(Thermostat thermostat) {
10         this.thermostat = thermostat;
11     }
12
13     public void execute() {
14         this.lastTemperature = this.thermostat.getTemperature();
15         this.thermostat.setTemperature(this.lastTemperature + 1);
16     }
17
18     public void undo() {
19         this.thermostat.setTemperature(this.lastTemperature);
20     }
21 }
```

Pour répondre à cette question, nous allons créer une commande particulière qui contient elle même plusieurs commandes. Le diagramme de classes est présenté sur la figure 11. J'ai ajouté une méthode `add` permettant d'ajouter une commande dans la macro. Il ne fallait pas oublier la relation de réalisation entre l'interface `Command` et la classe `MacroCommand`.

Le code source de la classe `MacroCommand` est présenté sur le listing 14. Rien de bien difficile, j'ai choisi de stocker les commandes dans une instance de `Vector` pour respecter l'ordre des commandes. Pour la méthode `undo`, j'ai utilisé la méthode statique `reverse` de la classe `Collections` pour inverser l'ordre des commandes.

(b) supposons que l'on dispose de deux boutons « StartRecordingMacro » et « StopRecordingMacro » qui permettent d'enregistrer les commandes effectuées. Spécialiser la classe `RemoteControl` pour ajouter cette fonctionnalité. On supposera que l'on dispose des méthodes `startRecordingMacroPressed` et `stopRecordingMacroPressed`.

Le code source de la classe `RemoteControlWithMacro` est présenté sur le listing 15. L'idée est bien sûr de spécialiser les méthodes `buttonOnPressed` et `buttonOffPressed` pour pouvoir non pas exécuter la commande mais l'enregistrer dans la macro quand on est en mode enregistrement. J'ai écrit ici les méthode `startRecordingMacro` et `stopRecordingMacro` qui sont triviales et je modélise le fait que l'on soit ou pas en mode d'enregistrement par un booléen. J'ai utilisé le fait que les attributs `onCommands` et `offCommands` de `RemoteControl` sont protégés pour pouvoir y accéder directement.

Listing 14 : La classe `MacroCommand`

```
1  package fr.supaero.command;
2  import java.util.Vector;
3  import java.util.Collections;
4
5  public class MacroCommand implements Command {
6
7      private Vector<Command> commands;
8
9      public MacroCommand() {
10         this.commands = new Vector<Command> ();
11     }
12
13     public void add(Command c) {
14         this.commands.add(c);
15     }
16
17     public void execute() {
18         for (Command c : this.commands) {
19             c.execute();
20         }
21     }
22
23     public void undo() {
24         Collections.reverse(this.commands);
25         for (Command c : this.commands) {
26             c.undo();
27         }
28     }
29 }
```

Listing 15 : La classe `RemoteControlWithMacro`

```
1  package fr.supaero.command;
2
3  public class RemoteControlWithMacro extends RemoteControl {
4
5      private MacroCommand macro;
6
7      private boolean isRecording;
8
9      public RemoteControlWithMacro() {
10         this.macro = new MacroCommand();
11         this.isRecording = false;
12     }
13
14     public void buttonOnPressed(int i) {
15         if (!this.isRecording) {
16             super.buttonOnPressed(i);
17         } else {
18             this.macro.add(this.onCommands[i]);
19         }
20     }
21
22     public void buttonOffPressed(int i) {
23         if (!this.isRecording) {
24             super.buttonOffPressed(i);
25         } else {
26             this.macro.add(this.offCommands[i]);
27         }
28     }
29
30     public void startRecordingMacro() {
31         this.isRecording = true;
32     }
33
34     public void stopRecordingMacro() {
35         this.isRecording = false;
36     }
37 }
```

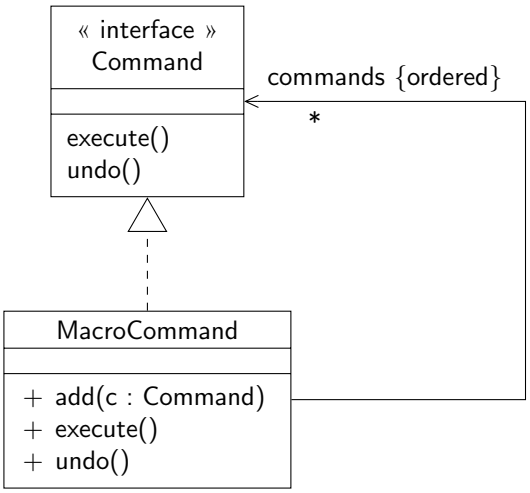

FIGURE 11 : La classe `MacroCommand`

3 Réseaux neuronaux pour l'apprentissage

Cet exercice est inspiré de [2]. **Il est plus que conseillé de lire l'énoncé en entier avant de répondre aux questions.**

On cherche ici à concevoir un logiciel permettant de classifier des données en utilisant un algorithme d'apprentissage. Pour cela, on va s'inspirer du fonctionnement du cerveau, en particulier de ses neurones, pour créer des réseaux de neurones artificiels.

Les réseaux de neurones artificiels permettent de simuler le fontionnement du cerveau. Ce sont de très bons outils permettant de classifier, de détecter des régularités, de travailler sur des images, du son, d'optimiser etc. Il existe plusieurs types de réseaux de neurones (cf. plus loin), mais tous possèdent un composant de base : le neurone.

Un neurone est une unité de traitement qui prend une entrée et renvoie un résultat. Le neurone produit un résultat fonction de ses entrées en utilisant une fonction particulière. On peut supposer que le type des valeurs manipulées en entrée et en sortie par les neurones dépend du type de réseau que l'on utilise.

Les réseaux sont organisés en couches reliées entre elles. Il existe deux couches particulières, la couche d'entrée qui contient les neurones qui reçoivent les valeurs d'entrées du réseau et la couche de sortie qui contient les neurones qui renvoient les valeurs de sortie du réseau. Ces deux dernières peuvent être identiques si le réseau n'a qu'une couche.

Un neurone d'une couche peut être relié à un ou plusieurs neurones :

– des neurones dont les résultats vont servir d'entrées au neurone ;
– des neurones qui se serviront du résultat produit par le neurone comme entrée.

Un poids est associé à chaque connexion entre neurones.

Comme un neurone peut avoir les sorties de plusieurs neurones en entrées, une fonction d'agrégation permet d'agréger les différentes entrées en utilisant les poids associés. Il en existe de plusieurs types : fonction utilisant la valeur maximale, la valeur minimale, le produit, la somme, une moyenne, un « ou » logique etc. des entrées. De la même façon, la valeur de sortie d'un neurone peut être calculée en utilisant différentes fonctions de transfert : gaussienne, rampe, linéaire, trapèze, tangente hyperbolique etc.

Un réseau de neurones est donc composé de couches de neurones. Il est possible d'associer à un réseau une fonction d'apprentissage, qui va permettre de corriger les valeurs des poids des connexions. Pour cela, on dispose d'un ensemble de mesures cibles et on peut connaître l'erreur entre la sortie de chaque neurone et le résultat attendu. Différentes fonctions d'apprentissage sont disponibles : fonctions d'apprentissage

itératives, parmi lesquelles on trouve des fonctions d'apprentissage supervisées (propagation arrière par exemple) ou non supervisées, comme la fonction d'apprentissage de Hopfield, la fonction d'apprentissage de Kohonen etc.

Enfin, voici les grands types de réseaux :

- le plus simple est le perceptron, qui utilise des valeurs binaires et possède deux couches ;
- le perceptron multi-couches est une évolution du perceptron qui accèpte plusieurs couches « cachées » entre la couche d'entrée et la couche de sortie. Il utilise un apprentissage par propagation arrière ;
- le réseau de Hopfield, qui ne possède pas de couche cachée, qui utilise la fonction d'apprentissage de même nom et dans lequel tous les neurones sont connectés entre eux ;
- le réseau de Kohonen, qui utilise la fonction du même nom et des valeurs réelles.

1. proposer un diagramme de classes d'analyse représentant le domaine étudié. 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. Enfin, on cherchera à avoir une solution la plus générique possible, i.e. permettant l'ajout de nouvelles fonctionnalités facilement (fonctions d'apprentissage par exemple).

   Si vous souhaitez contraindre votre diagramme, n'hésitez pas à ajouter des notes UML aux éléments que vous voulez contraindre ou du texte en plus du diagramme.

   Un diagramme de classes est proposé sur la figure 12. Quelques remarques sur la solution proposée :
   - la classe `Neurone` est générique : on ne sait pas en fait quel est le type des données manipulées en entrée et en sortie par le neurone. J'ai donc choisi de la rendre générique.
   - j'ai également choisi de représenter le lien entre le réseau et les différentes couches par deux associations de multiplicité 1 : une pour représenter la couche d'entrée du réseau et une pour représenter la couche de sortie du réseau. Les autres couches du réseau peuvent être découvertes via l'association qui existe entre les couches (on n'en sait pas plus en lisant le sujet). La modélisation des réseaux à une seule couche est bien réalisée ici : les deux associations « entrée » et « sortie » peuvent en fait utiliser la même couche, cela ne pose pas de problème ;
   - les associations entre neurones font apparaître un poids. Pour modéliser cela, on pourraît utiliser la notion de *classe association*, mais qui n'est pas abordée en cours. J'ai donc créé une classe `Poids` qui représente les poids des connexions entre neurones ;
   - la fonction d'agrégation associée à chaque neurone est représentée par une interface (on aurait pu choisir une classe abstraite). Cette modélisation est classique (il s'agit du patron de conception Stratégie) et permet d'avoir facilement des alternatives à une fonction d'agrégation via le sous-typage. J'ai également considéré que cette classe est générique, et on devrait normalement avoir une contrainte indiquant que le type utilisé doit être le même que celui du neurone auquel la fonction est associée. Le même raisonnement est appliquée pour les fonctions de sortie et la fonction d'apprentissage. Je n'ai écrit une contrainte que pour la fonction de sortie ;
   - un problème se posait pour les différents types de réseaux : il fallait pouvoir éventuellement lier chaque classe à une instanciation particulière du paramètre formel de type du neurone et à une fonction d'apprentissage particulière. Il n'y a pas de moyen facile de faire cela avec UML... J'ai donc choisi d'utiliser des contraintes informelles et je ne le montre que sur le réseau de Kohonen pour gagner de la place (la « formalisation » de la note n'est pas forcément très heureuse...).
2. on suppose que l'on s'intéresse ici au comportement d'un neurone de la couche d'entrée d'un réseau à deux neurones. On suppose donc que ce neurone n'a qu'un seul neurone successeur (i.e. que sa sortie ne sert d'entrée qu'à un seul neurone). Lorsque l'on utilise l'apprentissage, une séquence pouvant se produire est la suivante :

   (a) le réseau demande à la couche contenant le neurone de calculer les sorties de neurones ;

   (b) la couche demande au neurone de calculer sa sortie ;

(c) ce dernier appelle la fonction d'entrée qui lui est associée. Lors de cet appel, il récupérera sa valeur d'entrée en utilisant la couche ;

(d) le neurone utilise sa fonction de transfert. Après cet appel, il demande au neurone auquel il est connecté de calculer sa valeur de sortie (mécanisme de propagation) ;

(e) le module d'apprentissage va récupérer la valeur de sortie du deuxième neurone, la compare avec une valeur de référence ;

(f) le module d'apprentissage va demander au neurone la valeur du poids de sa liaison avec son neurone successeur et va la modifier.

Représenter le scénario précédent par un diagramme de séquence.

Un diagramme de séquence est présenté sur la figure 13. Le scénario proposé était bien sûr simplifié. En particulier, l'appel de propragation est mal représenté ici.

3. le cycle de vie d'un réseau de neurone peut être distingué en deux grands cas : soit le réseau de neurone est en mode d'apprentissage, soit il est en mode « normal ».
   En mode d'apprentissage, le réseau attend un exemple, calcule la sortie correspondant à l'entrée de l'exemple, compare cette sortie à la sortie de l'exemple et ajuste les poids en conséquence.
   En mode « normal », le réseau attend une entrée, calcule la sortie et la fournit.
   On passe d'un mode à l'autre grâce à un signal particulier.

   Représenter ce qui précède par un diagramme de machines d'états.

   Un diagramme de machines d'états représentant le cycle de vie d'un réseau est présenté sur la figure 14. Rien de bien particulier ici.

Références

[1] Aspectj. http://www.eclipse.org/aspectj/.

[2] Neuroph, Java neural network framework. http://neuroph.sourceforge.net.

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

[4] 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.

[5] B. Goetz. Decorating with dynamic proxies - IBM DeveloperWorks Java theory and practice series. http://www.ibm.com/developerworks/java/library/j-jtp08305.html, 2005.

[6] Wikipedia. Aspect-oriented programming. http://en.wikipedia.org/wiki/Aspect-oriented_programming.

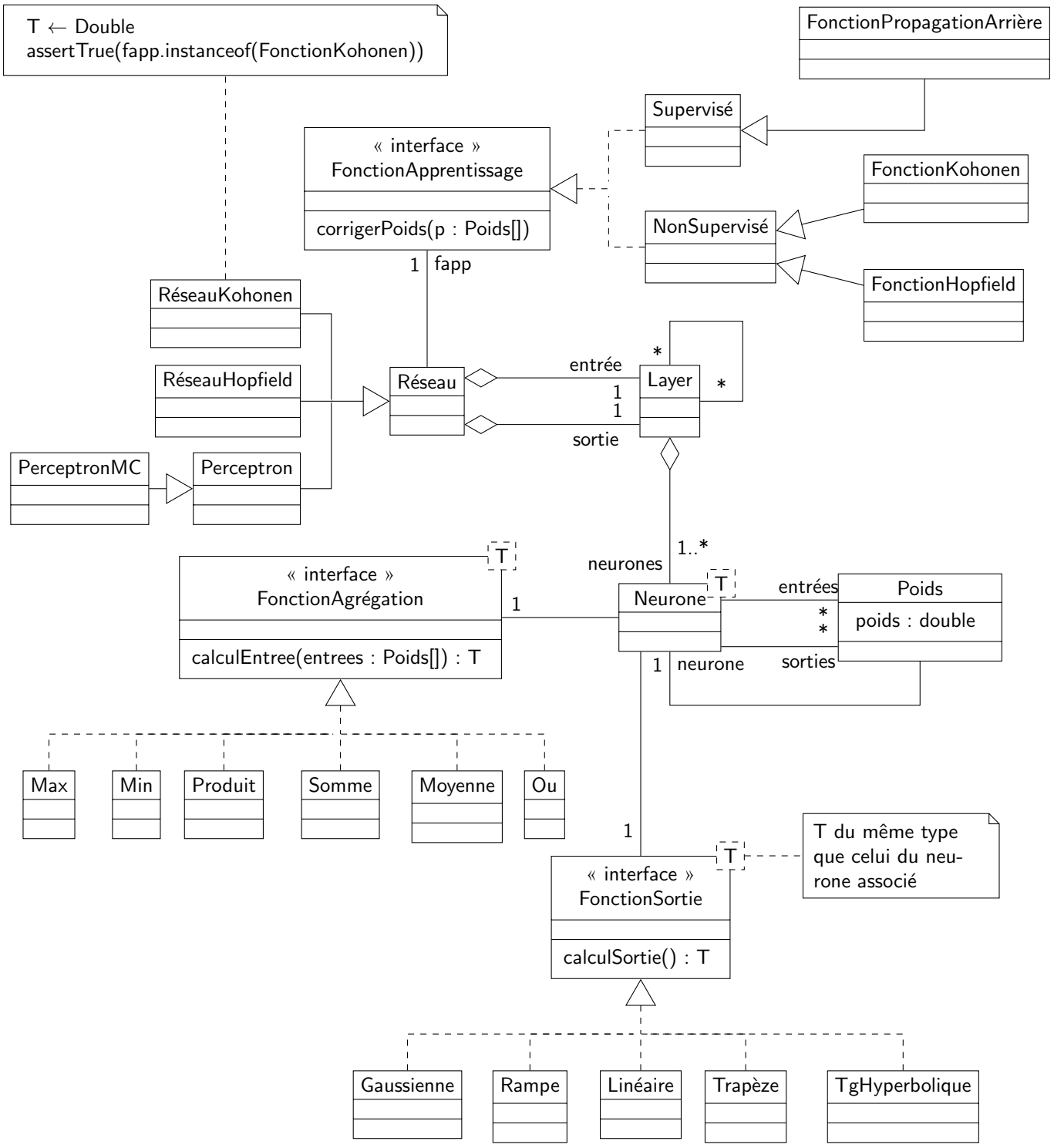

FIGURE 12 : Un diagramme de classes représentant l'application permettant de gérer des réseaux de neurons pour l'apprentissage

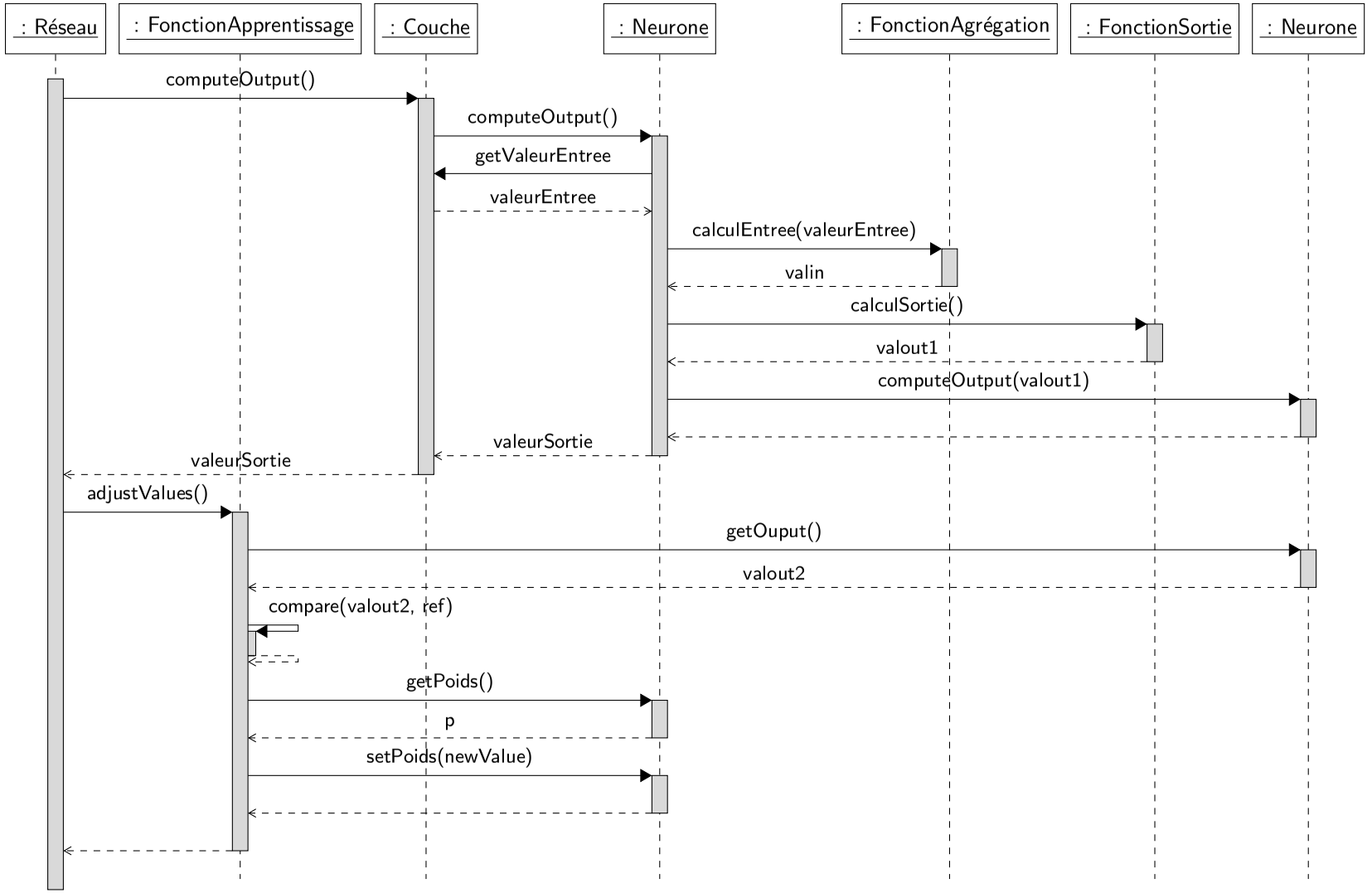

FIGURE 13 : Diagramme de séquence représentant un scénario d'apprentissage

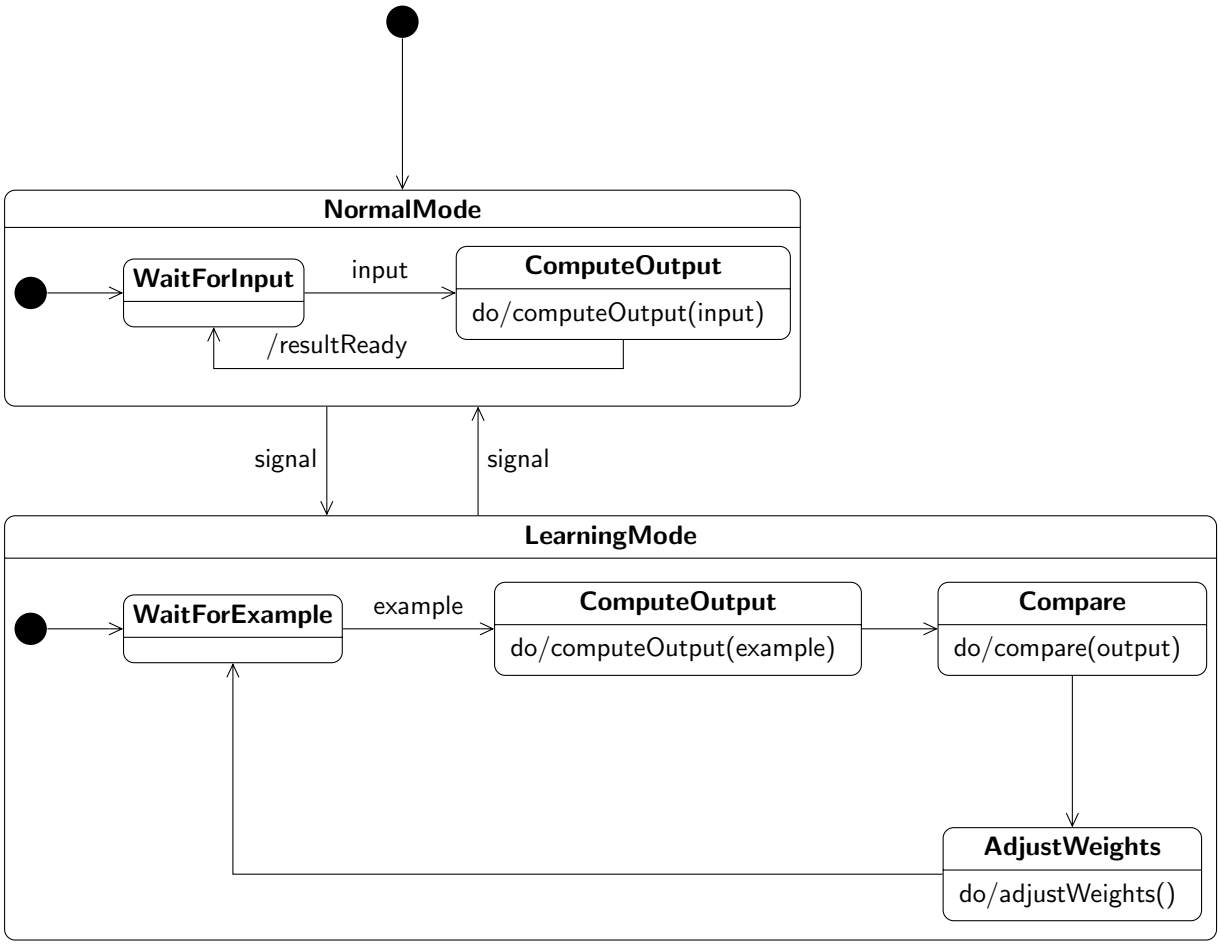

FIGURE 14 : Cycle de vie d'un réseau de neurones