IN201 : Corrigé de l'examen

SUPAERO 2A

Christophe Garion <garion@isae.fr>

16 février 2011

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. L'exercice 1 est un exercice orienté Java. L'exercice 2 étudie un patron de conception particulier. L'exercice 3 est un exercice de modélisation avec UML. Il sera tenu compte de la rédaction. Chaque exercice sera noté sur 10 points, mais le barème final peut être soumis à de légères modifications.

Remarques 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.

Dans certains exercices, 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).

1 Un système de *logging* évolué

Il est 1h14 du matin, votre BE est à rendre pour tout à l'heure et cela fait 2 heures que vous examinez tous vos fichiers sources pour enlever les `System.out.println` que vous avez mis un peu partout pour déboguer votre application. Les commentaires qui s'affichaient sur la console pour vous aider à avancer dans le BE feraient mauvaise impression si votre PC(wo)man les voyait...

En tout cas, ce travail d'effacement est fastidieux et peu intéressant. Afin d'être plus efficace dans votre prochain projet en Java [1], vous décidez de regarder ce « problème » de plus près. Ce que l'on cherche à faire lorsque l'on imprime sur la console des messages pour savoir ce qu'une application est en train de faire est appelé *logging* (historique des événements en français). Le *logging* d'une application peut se faire sur la console, mais également dans un fichier, une base de données etc. On peut normalement configurer le *logging* de façon simple (par un fichier de configuration par exemple) et donc facilement empêcher par exemple l'affichage de commentaires sur la console. Ça tombe bien, il y a justement une API Java qui fait cela, log4j [1]. Vous avez tout juste le temps de lire le tutoriel en ligne de log4j que crac, le réseau de l'ISAE tombe en rade, les 1A ayant eu la merveilleuse idée de télécharger tous en même temps la version PDF de [2]. Il va falloir recoder tout cela sans l'aide du réseau...

1. dans un premier temps, on ne s'intéresse qu'à du *logging* sur la console.

   D'après ce que vous avez pu comprendre, un *logger* est un objet sur lequel on peut écrire des messages avec différents niveaux de priorité. On peut écrire des messages informatifs (niveau INFO), des messages d'avertissement (niveau WARNING) ou des messages de débogage (niveau DEBUG). Le niveau le plus important est DEBUG, puis vient WARNING et enfin INFO. Lorsque l'on a un *logger*, on dispose donc de trois méthodes correspondant chacune à l'écriture d'un message avec une priorité particulière.

   Le *logger* lui-même a un *masque* qui est le niveau à partir duquel le *logger* va effectivement écrire les messages. Par exemple, un logger de niveau WARNING va effectivement écrire les messages de niveau WARNING et INFO qu'on lui a confiés, mais pas ceux de niveau DEBUG. Ainsi, un message de niveau DEBUG écrit sur un *logger* de niveau WARNING ne sera pas affiché sur la console. Par contre, si on change le niveau du même *logger* en DEBUG, le message sera affiché (ainsi que tous les messages de niveau WARNING ou INFO). On peut configurer le masque du *logger* à sa création ou via une méthode.

1. Si si, il y en aura sûrement un.

(a) donner une représentation UML de la classe `Logger`. On considérera que les niveaux de priorité peuvent être représentés par des attributs statiques de type entier dans la classe `Logger`. La classe `Logger` comportera, entre autres, une méthode pour chaque appel de *log* suivant son niveau.

Un diagramme UML de la classe est proposé sur la figure 1. Rien de bien compliqué, j'ai choisi ici d'utiliser des attributs statiques de type entier pour représenter les niveaux comme conseillé par le corrigé. On aurait pu utiliser une énumération, mais comme il faut pouvoir comparer les niveaux, l'utilisation d'entiers simplifiait les choses. J'ai choisi d'appeler les méthodes permettant d'écrire les messages de *logging* par le nom de la priorité associée (c'est ce qui se fait dans log4j par exemple).

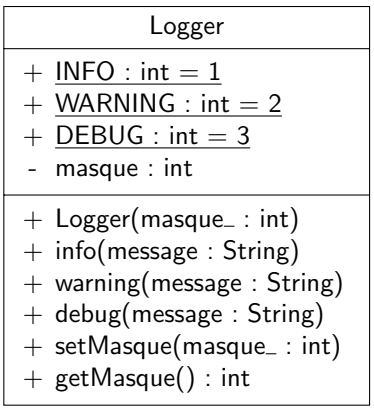

FIGURE 1 : La classe `Logger`

(b) écrire un programme d'exemple (classe possédant une méthode `main`) qui créé un *logger* de niveau WARNING et écrit trois messages respectivement de niveau INFO, WARNING et DEBUG sur ce *logger*.

Rien de bien compliqué, il suffisait d'utiliser la classe précédemment décrite en UML. Le listing 1 présente la classe.

Listing 1 : La classe `TestLogger`

```
1  package fr.supaero.log.first;
2
3  public class TestLogger {
4      public static final void main(final String[] args) {
5          Logger logger = new Logger(Logger.WARNING);
6          logger.info("Message de type INFO, devrait s'afficher");
7          logger.warning("Message de type WARNING, devrait s'afficher");
8          logger.debug("Message de type DEBUG, ne devrait pas s'afficher");
9      }
10 }
```

(c) écrire la classe `Logger` en Java.

La classe est présentée sur le listing 2. Rien de bien particulier pour l'écriture de la classe en Java, le plus gros du travail était fait via le diagramme UML. Vous remarquerez que j'ai utilisé

une méthode privée qui est appelée par les méthodes `info`, `warning` et `debug`. En effet, dans ces trois méthodes, il faut vérifier que le masque actuel du *logger* autorise l'affichage puis afficher le message si c'est le cas. On pouvait donc factoriser facilement ce code, cela évitait des erreurs potentielles.

Listing 2 : La classe `Logger`

```
1  package fr.supaero.log.first;
2
3  public class Logger {
4
5      public static int INFO = 1;
6      public static int WARNING = 2;
7      public static int DEBUG = 3;
8
9      private int masque;
10
11     public Logger(int masque_) {
12         this.masque=masque_;
13     }
14
15     public void setMasque(int masque_) {
16         this.masque=masque_;
17     }
18
19     public int getMasque() {
20         return this.masque;
21     }
22
23     public void info(String message) {
24         this.ecrire(message, INFO);
25     }
26
27     public void warning(String message) {
28         this.ecrire(message, WARNING);
29     }
30
31     public void debug(String message) {
32         this.ecrire(message, DEBUG);
33     }
34
35     public void ecrire(String message, int priorite) {
36         if (priorite <= this.masque) {
37             System.out.println(message);
38         }
39     }
40 }
```

2. on s'intéresse maintenant à plusieurs types de *loggers*. On pourra par exemple *logger* sur la console ou dans un fichier, mais rien ne devrait empêcher la création plus tard de *loggers* vers un fichier

XML, une base de données ou de *loggers* qui envoient des mails.

(a) en essayant d'avoir une solution permettant l'ajout facile de nouveaux types de *loggers*, quelle(s) modification(s) proposez-vous sur la classe `Logger` précédemment développée ?

Les différents types de *loggers* proposés n'ont qu'une seule différence : ce sur quoi ils écrivent réellement (console, fichier, fichier XML, base de données, etc). En particulier, la gestion des priorités reste la même pour toutes ces classes. Je propose de créer une méthode abstraite `ecrire` qui prend en paramètre un message à écrire et une priorité. Cette méthode sera appelée par les méthodes `info`, `warning` et `debug`. La méthode `ecrire` devra être implantée par les différentes sous-classes de `Logger` pour effectivement ecrire le message (sur la console, dans un fichier etc). Ce patron de conception est appelé « *template method* ». La classe `Logger` possédant maintenant une méthode abstraite, elle devient elle-même abstraite. Un diagramme UML de la nouvelle classe est donné sur la figure2. Sur ce diagramme, j'ai juste précisé le contenu de la méthode `info`, mais le principe est le même pour `warning` et `debug`.

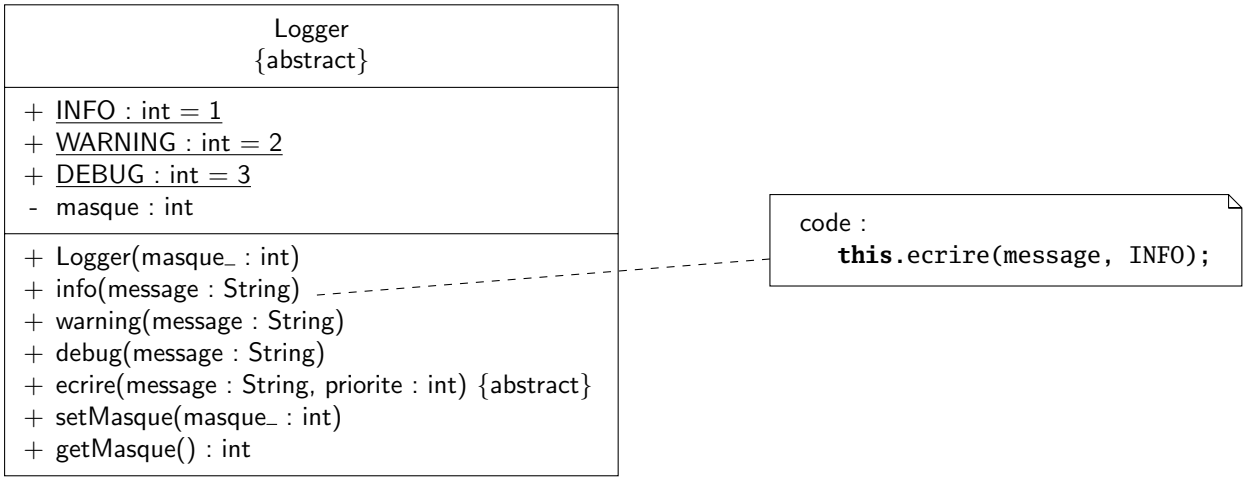

FIGURE 2 : La nouvelle classe `Logger`

(b) réécrire la classe `Logger` en Java.

La classe est présentée sur le listing 3. Rien de particulier ici.

Listing 3 : La nouvelle classe `Logger`

```
1  package fr.supaero.log.second;
2
3  public abstract class Logger {
4
5      public static int INFO = 1;
6      public static int WARNING = 2;
7      public static int DEBUG = 3;
8
9      private int masque;
10
11     public Logger(int masque_) {
12         this.masque=masque_;
```

```
13      }
14
15      public void setMasque(int masque_) {
16          this.masque=masque_;
17      }
18
19      public int getMasque() {
20          return this.masque;
21      }
22
23      public void info(String message) {
24          this.ecrire(message, INFO);
25      }
26
27      public void warning(String message) {
28          this.ecrire(message, WARNING);
29      }
30
31      public void debug(String message) {
32          this.ecrire(message, DEBUG);
33      }
34
35      public abstract void ecrire(String message, int priorite);
36  }
```

(c) écrire en Java un *logger* spécifique pour écriture sur la console.

La classe est présentée sur le listing 4. Rien de particulier encore, il ne fallait pas oublier l'appel à **super** dans le constructeur. Seule la méthode `ecrire` était à définir dans cette classe.

Listing 4 : La classe `ConsoleLogger`

```
1   package fr.supaero.log.second;
2
3   public class ConsoleLogger extends Logger {
4
5       public ConsoleLogger(int masque_) {
6           super(masque_);
7       }
8
9       public void ecrire(String message, int priorite) {
10          if (priorite <= this.getMasque()) {
11              System.out.println(message);
12          }
13      }
14  }
```

3. maintenant, supposons que l'on souhaite utiliser plusieurs *loggers* dans un même programme. Ces différents *loggers* peuvent avoir des niveaux de priorité différents. Par exemple, on peut souhaiter avoir un *logger* de priorité INFO sur la console pour vérifier que le déroulement du programme est

correct et un *logger* de priorité WARNING ou DEBUG dans un fichier pour analyser plus finement les problèmes qui peuvent se poser.

Évidemment, il paraît difficilement concevable que le développeur fasse deux appels, un pour le *logger* sur la console et un pour le *logger* dans le fichier, à chaque fois qu'il veut *logger* un message dans son application. Il faut donc trouver un moyen d'éviter que le développeur effectue de multiples appels.

(a) en cherchant dans [4], vous trouvez le patron de conception qui vous conviendrait : « chaîne de responsabilité ».

Le patron de conception « chaîne de responsabilité » permet de définir un ensemble de classes pouvant répondre à une requête sans se connaître les unes les autres. Un diagramme de classes présentant plus précisément le patron de conception est présenté sur la figure 3. En utilisant ce diagramme, on peut présenter un scénario d'utilisation de ce patron de conception comme suit :

- le client veut traiter une requête. Pour cela, il appelle `traiterRequete` sur un *handler*
- le *handler* vérifie qu'il peut traiter la requête. Si ce n'est pas le cas et qu'il a un *handler* successeur (représenté par le rôle `suivant` dans le diagramme), il lui transmet la requête
- la requête est traitée par le *handler* adéquat

On peut remarquer que dans notre cas, les *handlers* traiteront tous la requête. L'utilisation du patron de conception « chaîne de responsabilité » est ici un moyen de cacher une chaîne de traitement au développeur, pas forcément de trouver le bon *handler* pour traiter la requête.

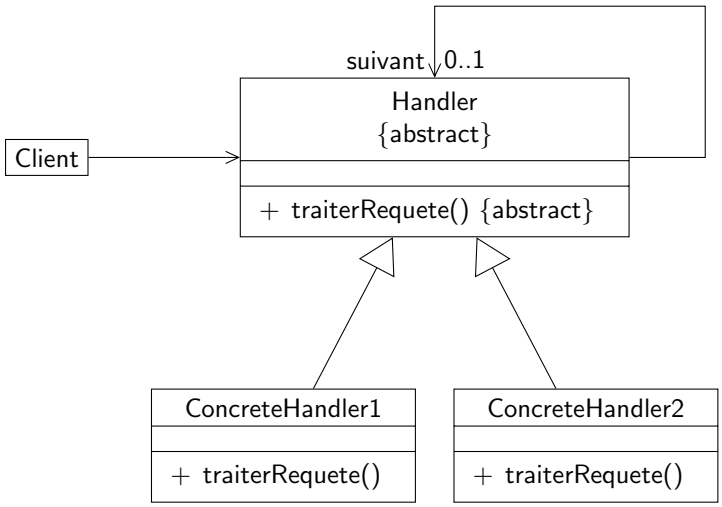

FIGURE 3 : Le patron de conception « chaîne de responsabilité »

Adapter le patron de conception à notre problème en ne considérant que deux *loggers* particuliers, `ConsoleLogger` et `FileLogger`, et représenter la solution obtenue par un diagramme UML.

Le diagramme de classes proposé est représenté sur la figure 4. Rien de bien difficile ici, la classe `Logger` est le *handler* abstrait et les deux classes de *logging* sont les *handlers* concrets. J'ai enlevé l'association entre *handlers* pour la faire apparaître explicitement comme un attribut de la classe `Logger`.

La méthode `traiter` de `Logger` est appelée par les méthodes `info`, `warning` et `debug` avec le niveau de priorité correspondant. `traiter` appelle la méthode `ecrire` pour écrire effectivement le message et ensuite transfert le traitement vers le prochain *logger* s'il existe.

En ce qui concerne le *logger* vers un fichier (classe `FileLogger`), il fallait ajouter un attribut représentant le fichier.

On peut remarquer que l'on aurait pu appliquer ce patron dès le départ, en créant un *handler* par niveau de priorité (c'est-à-dire un *handler* pour INFO, un pour WARNING et un pour DEBUG). Cependant, lorsque l'on aurait voulu ajouter différents types de *loggers* (vers la console, vers un fichier etc.), la solution aurait été plus difficile à trouver.

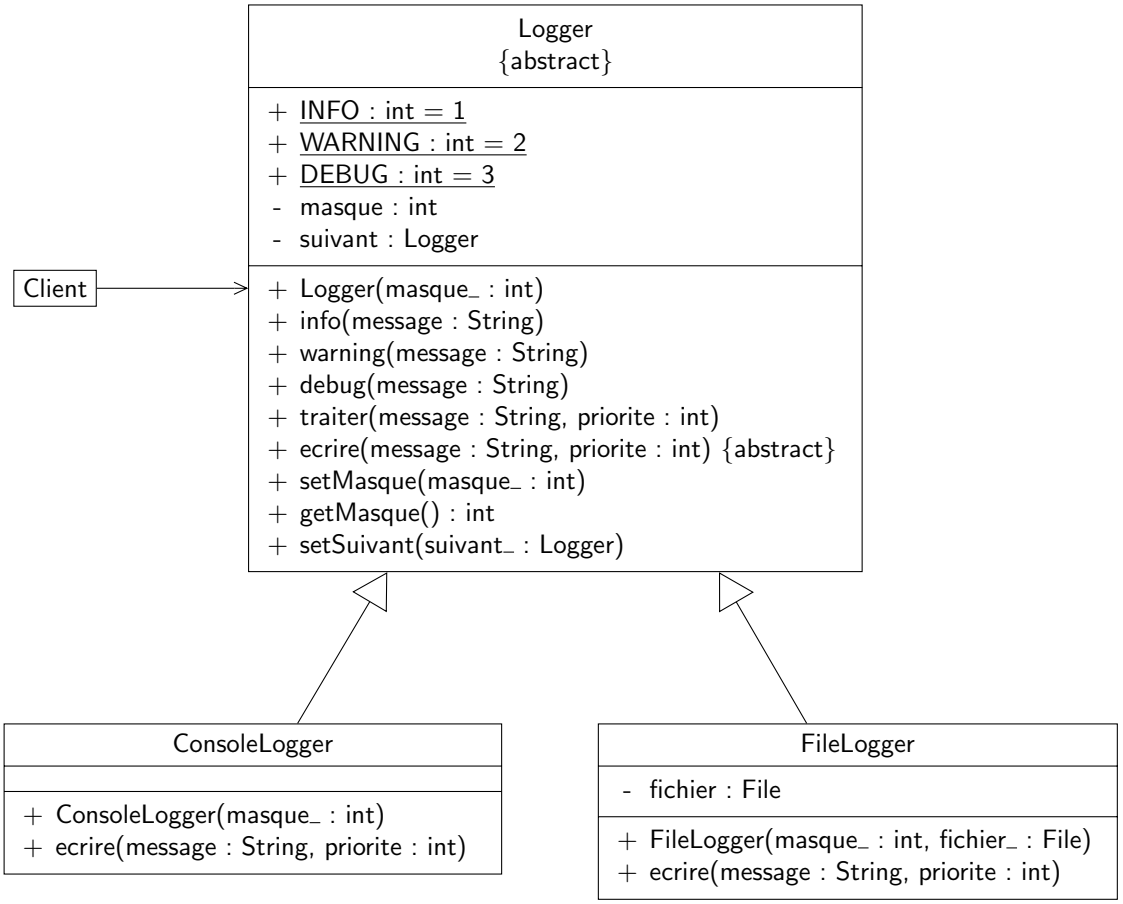

FIGURE 4 : Le patron « chaîne de responsabilité » adapté au problème

(b) réécrire en Java la classe `Logger`.

La classe `Logger` est présentée sur le listing 5. Rien de particulier ici, le travail a été fait à la question précédente.

Listing 5 : La classe `Logger` utilisant le patron « chaîne de responsabilité »

```
1  package fr.supaero.log;
2
3  public abstract class Logger {
4
5      public static int INFO = 1;
```

```
6       public static int WARNING = 2;
7       public static int DEBUG = 3;
8
9       private int masque;
10      private Logger suivant;
11
12      public Logger(int masque_) {
13          this.masque=masque_;
14      }
15
16      public void setMasque(int masque_) {
17          this.masque=masque_;
18      }
19
20      public int getMasque() {
21          return this.masque;
22      }
23
24      public void setSuivant(Logger suivant_) {
25          this.suivant = suivant_;
26      }
27
28      public void info(String message) {
29          this.traiter(message, INFO);
30      }
31
32      public void warning(String message) {
33          this.traiter(message, WARNING);
34      }
35
36      public void debug(String message) {
37          this.traiter(message, DEBUG);
38      }
39
40      public void traiter(String message, int priorite) {
41          this.ecrire(message, priorite);
42
43          if (this.suivant != null) {
44              this.suivant.traiter(message, priorite);
45          }
46      }
47
48      public abstract void ecrire(String message, int priorite);
49  }
```

(c) écrire un programme d'exemple (classe possédant une méthode `main`) utilisant deux *loggers* :
- un premier *logger* de type `ConsoleLogger` de priorité INFO ;
- un deuxième *logger* de type `FileLogger` de priorité DEBUG.

Le programme écrira un message pour chaque niveau de priorité.

Là encore, pas de difficulté particulière, le listing 6 présente la classe.

Listing 6 : La classe `TestChainLogger`

```
1  package fr.supaero.log;
2  import java.io.File;
3
4  public class TestChainLogger {
5      public static final void main(final String[] args) {
6          Logger logger = new ConsoleLogger(Logger.INFO);
7          logger.setSuivant(new FileLogger(Logger.DEBUG, new File("./log.txt")));
8
9          logger.info("Message de type INFO");
10         logger.warning("Message de type WARNING");
11         logger.debug("Message de type DEBUG");
12     }
13 }
```

4. il serait également intéressant de pouvoir ajouter des informations dans les messages que l'utilisateur veut *logger*. Par exemple, on pourrait ajouter automatiquement la date à laquelle le message a été écrit sur le *logger*. Ce problème vous rappelle quelque chose... En révisant votre examen, vous avez aperçu dans l'examen 2009/2010 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 5 et un diagramme de séquence expliquant son fonctionnement est donné sur la figure 6.

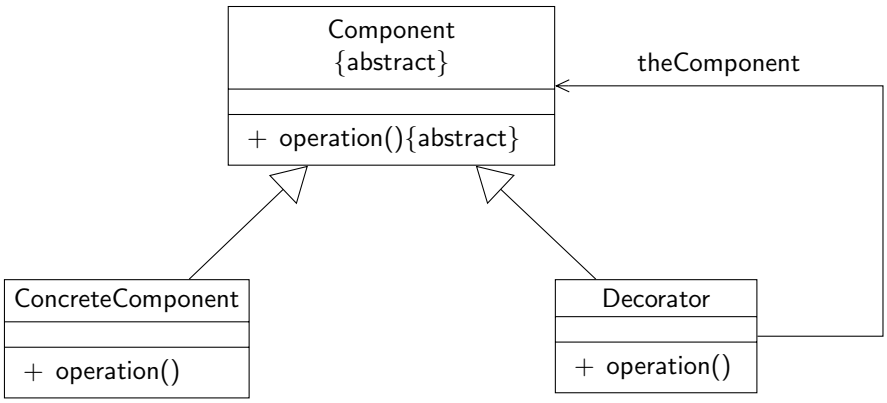

FIGURE 5 : Le patron de conception « décorateur »

(a) instancier le patron de conception « décorateur » sur le problème. On considérera que l'on cherche à pouvoir ajouter une date à chaque *log* de l'utilisateur s'il le souhaite via un décorateur particulier, `DateAppender`.
Le diagramme de classe correspondant est donné sur la figure 7. Rien de bien particulier, on considère que les *loggers* développés précédemment peuvent être décorés par `DateAppender`. L'opération concernée était bien sûr `ecrire`.

(b) écrire la classe `DateAppender` en Java. On rappelle que la classe `java.util.Date` permet d'avoir la date courante en instanciant un objet de la classe.

La classe est présentée sur le listing 7. Rien de bien difficile, la classe ne fait rien mis à part appeler la méthode `ecrire` du logger qu'elle décore en lui passant un message modifié par l'ajout

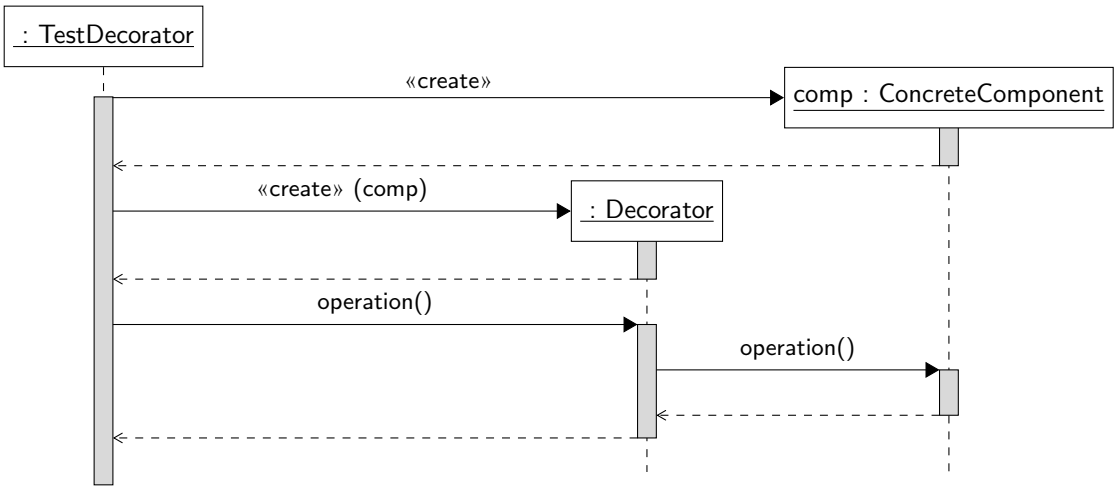

FIGURE 6 : Un scénario d'utilisation du décorateur

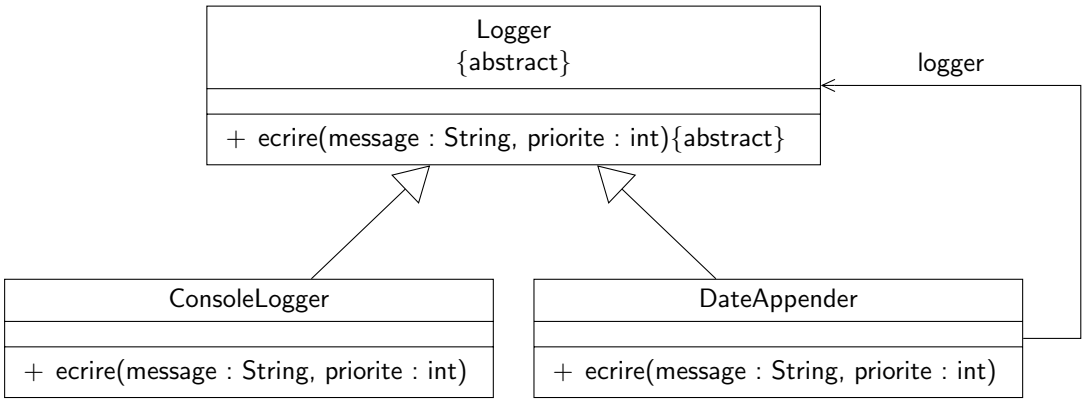

FIGURE 7 : Le patron de conception « décorateur » adapté au problème

de la date courante. Il fallait faire attention dans le constructeur de `DateAppender` d'utiliser le masque du *logger* que l'on décore pour conserver une certaine cohérence.

Listing 7 : La classe `DateAppender`

```
1  package fr.supaero.log;
2
3  import java.util.Date;
4
5  public class DateAppender extends Logger {
6
7      private Logger logger;
8
9      public DateAppender(Logger logger_) {
10         super(logger_.getMasque());
11         this.logger = logger_;
```

```
12      }
13
14      public void ecrire(String message, int priorite) {
15          this.logger.ecrire(new Date().toString() + ": " + message, priorite);
16      }
17  }
```

2 Visiteur du soir, espoir

Ça y est, votre binôme a encore eu une idée de génie. Il a déboulé dans votre chambre et vous disant « hé, en fait ce serait super cool de pouvoir faire des recherches évoluées sur les étiquettes du BE, genre tu vois tu cherches les marques-pages étiquetés par « vacances » ou par « mer ». Ou alors ceux étiquetés à la fois par « Java » et « fun » (ça existe ?). Bon c'est pas tout ça, mais j'ai une soirée à préparer, alors vu que je me suis farci le BE de C l'an dernier, au boulot. ». La soirée s'annonce bien.

Reprenons depuis le départ. Vous avez développé la classe `Etiquette` lors du TP récapitulatif et elle contient un ensemble de marque-pages. Un diagramme UML de la classe `Etiquette` est rappelé sur la figure 8.

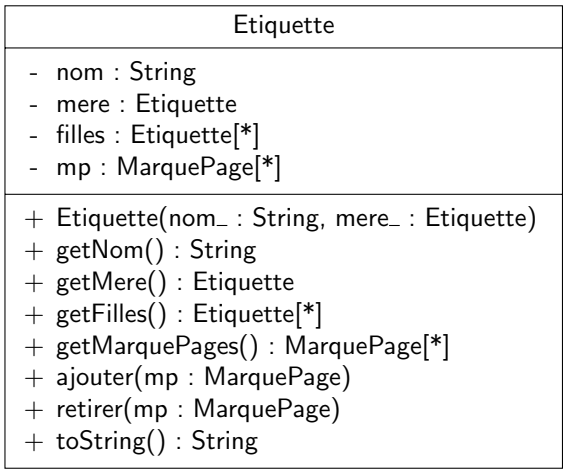

FIGURE 8 : Diagramme détaillé de la classe `Etiquette`

L'objectif est donc de pouvoir poser des requêtes complexes pour récupérer de façon plus fine des marque-pages. On va manipuler des expressions permettant de combiner avec des opérateurs logiques différentes étiquettes. On va se restreindre ici aux opérateurs logiques OU, ET et NOT. Par exemple, on pourra considérer l'expression « vacances ET (mer OU montagne) ET NOT (camping) » qui représentera une recherche de tous les marque-pages associés à l'étiquette « vacances » et à l'étiquette « mer » ou à l'étiquette « montagne », mais pas à l'étiquette « camping ». On remarque tout d'abord que les constituants de l'expression peuvent être :

- des opérateurs binaires (qui prennent deux opérandes), comme ET et OU
- des opérateurs unaires comme NOT
- des étiquettes

Pour représenter un critère de recherche complexe, on peut utiliser une structure d'arbre. On rappelle qu'un arbre est une structure comportant un ensemble de nœuds reliés entre eux par des arcs. Un nœud accessible depuis un nœud A par un arc est dit nœud fils de A. Dans notre problématique de représentation

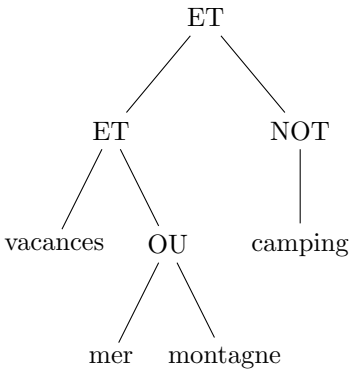

FIGURE 9 : Représentation sous forme d'arbre du critère de recherche « vacances ET (mer OU montagne) ET NOT (camping) »

de critères de recherche, chaque nœud de l'arbre est soit une étiquette, soit un opérateur. Chaque nœud représentant un opérateur possède un nombre de nœuds fils égal à son arité (le nombre d'arguments de l'opérateur). Par exemple, le critère « vacances ET (mer OU montagne) ET NOT (camping) » peut être représenté par l'arbre représenté sur la figure 9.

Enfin, on souhaite également pouvoir afficher sur la console une représentation d'un critère de recherche en utilisant sa représentation sous forme d'arbre.

1. proposer un diagramme de classes simple mais complet permettant de représenter les différents nœuds que l'on peut représenter. On introduira les opérations nécessaires à l'affichage et à l'évaluation d'une requête dans les nœuds et on veillera à construire un diagramme suffisamment générique (via l'héritage ou la réalisation d'interfaces). On fera ainsi apparaître :
   – un type de nœud générique sous forme d'une classe ou d'une interface
   – les différents types d'opérations (unaire ou binaire) sous forme de classes ou d'interfaces
   – les opérations « ET », « OU » et « NOT » sous forme de classes
   – les nœuds ne contenant qu'une étiquette que l'on représentera par une classe `NoeudEtiquette`

   On ne représentera pas les éventuelles associations entre classes, mais on introduira des attributs permettant de les coder.

   Une solution est proposée sur la figure 10.

   Tous les nœuds représentant des expressions arithmétiques pouvaient rendre les mêmes services, i.e. afficher le critère de recherche et renvoyer le résultat de l'évaluation de la requête sous forme d'un ensemble de marque-pages. Évidemment, ces méthodes ne pouvaient pas être implantées à ce niveau, donc elles devaient être abstraites. J'ai choisi ici d'utiliser une interface, car il n'y avait pas d'attributs représentatifs d'un nœud générique. Les classes `OperateurUnaire`, `OperateurBinaire` permettaient d'introduire les différents types d'opérateurs. Un opérateur unaire ne possède qu'un fils d'où l'attribut unique de la classe `OperateurUnaire`. Un opérateur binaire possède deux fils que j'ai appelé opérandes gauche et droit. Ces deux classes sont abstraites, car l'implantation des méthodes d'interprétation et d'affichage de la requête dépend de l'opérateur (on aurait toutefois pu embarquer une représentation de l'opérateur sous forme d'attribut pour l'affichage). Vous remarquerez que je n'ai pas répété les méthodes de l'interface dans ces classes abstraites car on les récupère par réalisation de l'interface et on ne sait pas les implanter dans les classes abstraites. On spécialise ensuite ces classes avec les opérateurs nécessaires. Ces classes possèdent des méthodes concrètes, car on sait alors calculer et afficher les expressions.

   Enfin, la classe `NoeudEtiquette` représente les nœuds terminaux et est caractérisée par l'étiquette

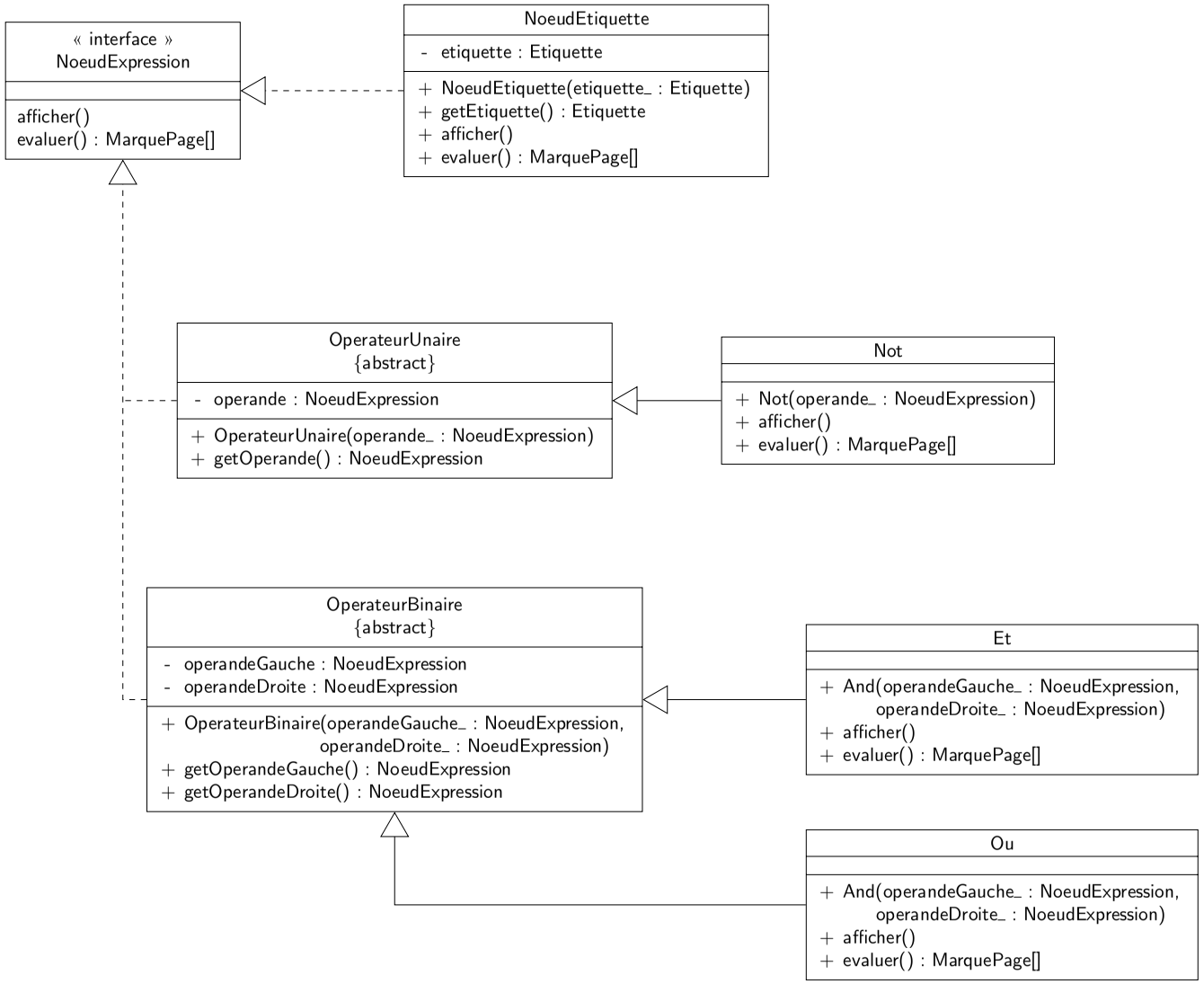

FIGURE 10 : Diagramme de classes représentant les nœuds

embarquée.

2. écrire la classe `NoeudEtiquette` en Java. Cette classe ne contient qu'une étiquette et possédera une méthode d'affichage et une méthode permettant d'évaluer le nœud.

   La classe `NoeudEtiquette` est présentée sur le listing 8. Il n'y avait rien de particulier à signaler pour l'écriture de cette classe, il suffisait d'utiliser les méthodes de `Etiquette`. L'évaluation d'un nœud de type `NoeudEtiquette` est assez simple : comme on est sur un nœud terminal de l'arbre, il suffit de renvoyer les marque-pages contenus dans l'étiquette du nœud.

Listing 8 : La classe `NoeudEtiquette`

```
1  package fr.supaero.tags.first;
2
3  import fr.supaero.tags.*;
4  import java.util.*;
5
6  public class NoeudEtiquette implements NoeudExpression {
7
8      private Etiquette etiquette;
9
10     public NoeudEtiquette(Etiquette etiquette_) {
11         this.etiquette = etiquette_;
12     }
13
14     public Etiquette getEtiquette() {
15         return this.etiquette;
16     }
17
18     public void afficher() {
19         System.out.println(this.etiquette.getNom());
20     }
21
22     public Vector<MarquePage> evaluer() {
23         return this.etiquette.getMarquePages();
24     }
25 }
```

3. quelles critiques peut-on émettre sur ce modèle ? Vous réfléchirez en particulier à l'ajout d'une nouvelle opération (autre que l'évaluation et l'affichage) sur les expressions. Par exemple, on pourrait souhaiter avoir une opération qui permette de sauvegarder l'expression de recherche dans un fichier texte.

   On remarque dans un premier temps que toutes les opérations de traitement sont embarquées dans chacun des types de nœuds possibles, ce qui ne rend pas le code très lisible (imaginez que l'on ait une cinquantaine d'opérateurs...). De plus, comme les traitements sont embarqués dans chacune des classes, on ne dispose pas d'une seule classes avec toutes les opérations d'évaluation par exemple.

   Enfin, supposons que l'on veuille ajouter une nouvelle opération sur les nœuds comme par exemple la sauvegarde de l'expression dans un fichier texte : on est obligé de réécrire toutes les classes et de les recompiler. Il serait plus efficace et plus sûr d'ajouter de nouvelles opérations sans modifier les nœuds.

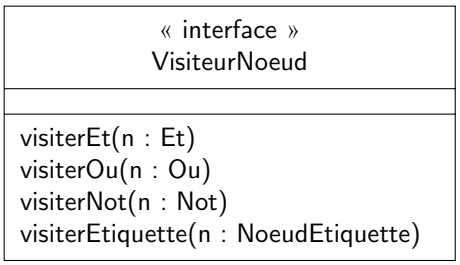

FIGURE 11 : L'interface `VisiteurNoeud`

4. pour pallier les problèmes évoqués dans la question précédente, on décide d'appliquer un patron de conception particulier, le *visiteur* [4, 3]. L'idée de ce patron est d'encapsuler une opération dans un objet appelé visiteur et de passer cet objet aux nœuds de l'arbre. Lorsqu'un nœud *accepte* un visiteur pour une opération particulière, il appelle une méthode du visiteur correspondant à son type et se passe en paramètre de cette méthode.

   Par exemple, supposons que l'on dispose d'un visiteur pour une opération `op` sur les nœuds. Cet objet aura donc une méthode de visite correspondant à ET, une à OU, une à NOT et une aux nœuds représentant les étiquettes. Ce sont ces méthodes qui « effectueront » `op` sur les différents nœuds. On les nommera par convention `visiterEt`, `visiterOu`, `visiterNot`, `visiterEtiquette`[2]. On dispose ainsi des méthodes pour chaque type de nœud. Comme tous les visiteurs devront disposer de ces méthodes, on peut créer une interface `VisiteurNoeud` les possédant (cf. figure 11).

   Les nœuds n'auront plus à coder les différentes opérations qui peuvent s'effectuer sur eux, celles-ci seront « stockées » dans un visiteur particulier. Il suffit alors pour chaque nœud de disposer d'une méthode `accepter(v : Visiteur)` qui appelle la méthode du visiteur correspondant au type de nœud. Un exemple d'utilisation d'un visiteur par un nœud de type `Et` est donné sur la figure 12. Évidemment, dans l'appel à `visiterEt`, on risque d'appeler une méthode particulière sur les opérandes de l'opérateur ET...

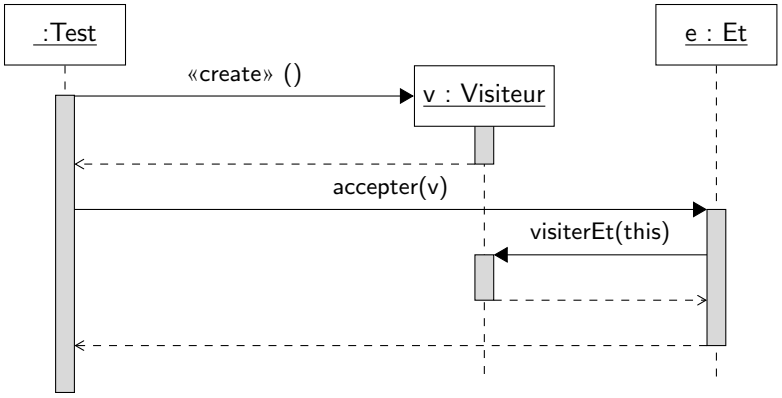

FIGURE 12 : Diagramme de séquence présentant une utilisation de `Visiteur`

Modifier les classes que vous aviez proposées en question 1 et introduire les deux classes de visiteurs nécessaires à la réalisation des opérations d'affichage et de calcul de l'expression (attention pour

2. On remarquera qu'en Java, on pourrait utiliser plusieurs méthodes s'appelant `visiter` mais avec des arguments différents (nœud de type ET, de type OU etc). C'est le principe de la *surcharge* de méthodes

cette dernière, il faut trouver un moyen de conserver le résultat de l'évaluation de la requête).

Le diagramme de classes est présenté sur la figure 13. Rien de bien particulier, il fallait juste penser à utiliser un attribut dans `VisiteurEvaluation` pour stocker le résultat intermédiaire de l'évaluation de la requête. Vous remarquerez également que j'ai utilisé la surcharge des méthodes dans l'interface `VisiteurNoeud` pour éviter d'avoir des méthodes avec des noms comportant à chaque fois le type de nœud concerné (le type du paramètre suffit).

5. écrire les classes et interfaces correspondant au diagramme construit à la question précédente.
   On rappelle que la classe `HashSet` du paquetage `java.util` représente un ensemble d'objets (sans répétition d'occurences possible donc). Elle contient une méthode `boolean contains(E element)` qui permet de savoir si un élément appartient ou non à l'ensemble et une méthode `add` qui permet d'ajouter un élément à l'ensemble (si l'élément est déjà dans l'ensemble, il n'est pas ajouté). On supposera que la classe `MarquePage` est correctement implantée et que la méthode `contains` de `HashSet` fonctionnera avec cette classe [3]. Enfin, `HashSet` a un constructeur qui prend une instance de `Vector` en paramètre et permet de construire un ensemble à partir d'un vecteur d'objets.
   Si plusieurs classes sont quasiment identiques (au nom près par exemple), vous pouvez n'en écrire qu'une seule, préciser que les autres classes ont le même code et donner les différences entre les classes.
   Enfin, l'implantation de la méthode permettant de visiter un nœud de type Not pour l'évaluation est compliquée, le traitement de ce problème sera considéré comme un bonus.

   Les classes et interfaces sont présentées sur les listings 9, 10, 11, 12, 13, 14, 15, 16, 17 et 18.

Listing 9 : L'interface `NoeudExpression`

```
1  package fr.supaero.tags;
2
3  public interface NoeudExpression {
4
5      void accepter(VisiteurNoeud v);
6  }
```

Listing 10 : La classe abstraite `OperationUnaire`

```
1  package fr.supaero.tags;
2
3  public abstract class OperationUnaire implements NoeudExpression {
4
5      private NoeudExpression operande;
6
7      public OperationUnaire(NoeudExpression operande_) {
8          this.operande = operande_;
9      }
10
11     public NoeudExpression getOperande() {
12         return this.operande;
13     }
14 }
```

3. Il faut en effet que la méthode `equals` de la classe soit correctement implantée.

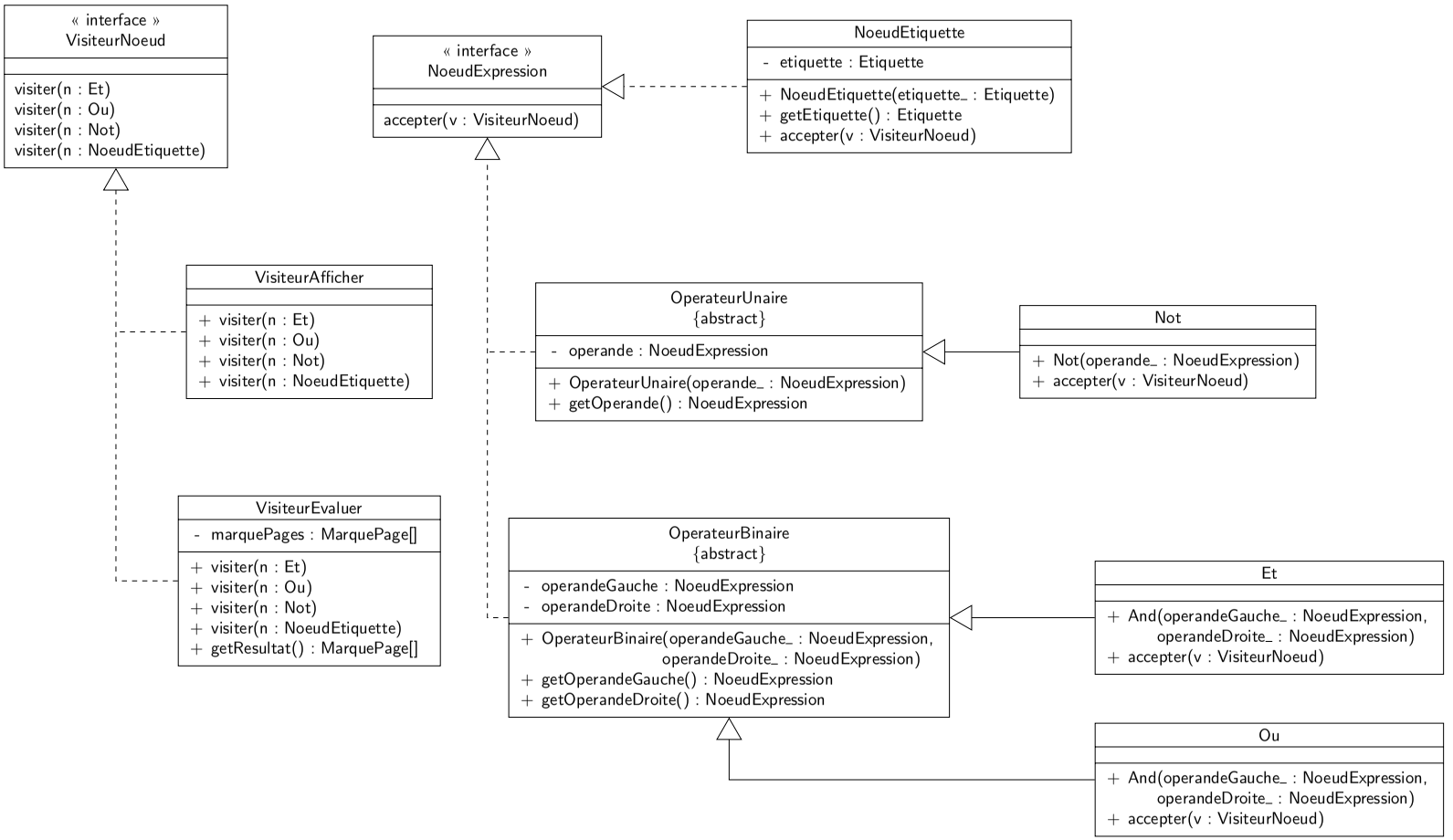

FIGURE 13 : Diagramme de classe avec utilisation des visiteurs

Listing 11 : La classe abstraite `OperationBinaire`

```
1  package fr.supaero.tags;
2
3  public abstract class OperationBinaire implements NoeudExpression {
4
5      private NoeudExpression operandeGauche;
6      private NoeudExpression operandeDroite;
7
8      public OperationBinaire(NoeudExpression operandeGauche_,
9                              NoeudExpression operandeDroite_) {
10         this.operandeGauche = operandeGauche_;
11         this.operandeDroite = operandeDroite_;
12     }
13
14     public NoeudExpression getOperandeGauche() {
15         return this.operandeGauche;
16     }
17
18     public NoeudExpression getOperandeDroite() {
19         return this.operandeDroite;
20     }
21 }
```

Listing 12 : la classe `NoeudEtiquette`

```
1  package fr.supaero.tags;
2
3  public class NoeudEtiquette implements NoeudExpression {
4
5      private Etiquette etiquette;
6
7
8      public NoeudEtiquette(Etiquette etiquette_) {
9          this.etiquette = etiquette_;
10     }
11
12     public Etiquette getEtiquette() {
13         return this.etiquette;
14     }
15
16     public void accepter(VisiteurNoeud v) {
17         v.visiter(this);
18     }
19 }
```

Listing 13 : La classe `Not`

```
1  package fr.supaero.tags;
```

```
2
3  public class Not extends OperationUnaire {
4
5      public Not(NoeudExpression operande_) {
6          super(operande_);
7      }
8
9      public void accepter(VisiteurNoeud v) {
10         v.visiter(this);
11     }
12 }
```

Listing 14 : La classe `Et`

```
1  package fr.supaero.tags;
2
3  public class Et extends OperationBinaire {
4
5      public Et(NoeudExpression operandeGauche_,
6                NoeudExpression operandeDroite_) {
7          super(operandeGauche_, operandeDroite_);
8      }
9
10     public void accepter(VisiteurNoeud v) {
11         v.visiter(this);
12     }
13 }
```

Listing 15 : La classe `Ou`

```
1  package fr.supaero.tags;
2
3  public class Ou extends OperationBinaire {
4
5      public Ou(NoeudExpression operandeGauche_,
6                NoeudExpression operandeDroite_) {
7          super(operandeGauche_, operandeDroite_);
8      }
9
10     public void accepter(VisiteurNoeud v) {
11         v.visiter(this);
12     }
13 }
```

Listing 16 : L'interface `VisiteurNoeud`

```
1  package fr.supaero.tags;
2
3  public interface VisiteurNoeud {
```

```
4       void visiter(Et n);
5       void visiter(Ou n);
6       void visiter(Not n);
7       void visiter(NoeudEtiquette n);
8   }
```

Listing 17 : La classe `VisiteurAfficher`

```
1   package fr.supaero.tags;
2
3   public class VisiteurAfficher implements VisiteurNoeud {
4
5       public void visiter(Et n) {
6           this.afficherOperationBinaire(n, "ET");
7       }
8
9       public void visiter(Ou n) {
10          this.afficherOperationBinaire(n, "OU");
11      }
12
13      public void visiter(Not n) {
14          System.out.print("(");
15          System.out.println("NOT ");
16          n.getOperande().accepter(this);
17          System.out.print(")");
18      }
19
20      public void visiter(NoeudEtiquette n) {
21          System.out.println(n.getEtiquette().getNom());
22      }
23
24      private void afficherOperationBinaire(OperationBinaire n, String nom) {
25          System.out.print("(");
26          n.getOperandeGauche().accepter(this);
27          System.out.println(" " + nom + " ");
28          n.getOperandeDroite().accepter(this);
29          System.out.print(")");
30      }
31  }
```

Listing 18 : la classe `VisiteurEvaluer`

```
1   package fr.supaero.tags;
2   import java.util.Vector;
3   import java.util.HashSet;
4
5   public class VisiteurEvaluer implements VisiteurNoeud {
6
7       private HashSet<MarquePage> marquePages;
```

```
8       private HashSet<MarquePage> allMP;
9
10      public VisiteurEvaluer(Etiquette racine_) {
11          this.marquePages = new HashSet<MarquePage>();
12          allMP = new HashSet(this.getAllMarquePages(racine_));
13      }
14
15      private Vector<MarquePage> getAllMarquePages(Etiquette etiquette) {
16          Vector<MarquePage> vec = new Vector<MarquePage>();
17          this.getAllMarquePagesAux(etiquette, vec);
18
19          return vec;
20      }
21
22      private void getAllMarquePagesAux(Etiquette etiquette,
23                                        Vector<MarquePage> vec) {
24          vec.addAll(etiquette.getMarquePages());
25
26          for (Etiquette e: etiquette.getFilles()) {
27              this.getAllMarquePagesAux(e, vec);
28          }
29      }
30
31      public void visiter(Et n) {
32          n.getOperandeGauche().accepter(this);
33
34          // on stocke le resultat de l'evaluation pour l'operande
35          // gauche dans une variable intermediaire car l'attribut va
36          // etre efface lors de l'appel sur l'operande de droite.
37          HashSet<MarquePage> mpg = this.getResultat();
38
39          n.getOperandeDroite().accepter(this);
40
41          HashSet<MarquePage> aux = new HashSet<MarquePage>();
42
43          for (MarquePage mp: this.marquePages) {
44              if (! mpg.contains(mp)) {
45                  aux.add(mp);
46              }
47          }
48
49          for (MarquePage mp: aux) {
50              this.marquePages.remove(mp);
51          }
52      }
53
54      public void visiter(Ou n) {
55          n.getOperandeGauche().accepter(this);
56
57          // on stocke le resultat de l'evaluation pour l'operande
```

```
58          // gauche dans une variable intermediaire car l'attribut va
59          // etre efface lors de l'appel sur l'operande de droite.
60          HashSet<MarquePage> mpg = this.getResultat();
61
62          n.getOperandeDroite().accepter(this);
63
64          this.marquePages.addAll(mpg);
65      }
66
67      public void visiter(Not n) {
68          n.getOperande().accepter(this);
69
70          // on recupere tous les marque-pages et on enleve ceux du
71          // resultat
72          HashSet<MarquePage> aux = this.marquePages;
73          this.marquePages = allMP;
74
75          HashSet<MarquePage> auxRemove = new HashSet<MarquePage>();
76
77          for (MarquePage mp: this.marquePages) {
78              if (aux.contains(mp)) {
79                  auxRemove.add(mp);
80              }
81          }
82
83          for (MarquePage mp: auxRemove) {
84              this.marquePages.remove(mp);
85          }
86      }
87
88      public void visiter(NoeudEtiquette n) {
89          this.marquePages = new HashSet(n.getEtiquette().getMarquePages());
90      }
91
92      public HashSet<MarquePage> getResultat() {
93          return this.marquePages;
94      }
95  }
```

Il n'y avait rien de particulier à noter. Vous remarquerez que par exemple dans `VisiteurAfficher` j'ai utilisé une méthode privée pour factoriser du code. Ce n'était pas demandé bien sûr, mais c'est une bonne pratique.

La classe la plus difficile à écrire était bien sûr `VisiteurEvaluation`, car le type de retour des méthodes de visite était **void** alors que l'on attend un résultat lors de l'évaluation de la requête.

En ce qui concerne l'opérateur NOT, le problème qui se pose est de trouver le complément des marque-pages de l'opérande de NOT. Il faut donc connaître l'ensemble des marques-pages et donc la racine de l'arbre des étiquettes. J'ai donc choisi d'introduire un attribut dans `VisiteurEvaluer` qui stocke l'ensemble des marque-pages de l'arbre. Pour trouver l'ensemble des marque-pages de l'arbre, il faut parcourir de façon récursive l'arbre.

Enfin, vous remarquerez que j'ai utilisé des instances de `HashSet` intermédiaires lorsque je veux

enlever des éléments d'un `HashSet` existant. En effet, on ne peut pas modifier une collection que l'on est en train de parcourir (levée d'une `ConcurrentModificationException`). Il faut donc utiliser une collection intermédiaire. Bien évidemment, cela ne vous était pas demandé à l'examen.

6. écrire un programme d'exemple (classe contenant une méthode `main`) construisant la requête présentée en figure 9, l'affichant, l'évaluant et affichant le résultat de la requête. On créera des étiquettes directement sous la racine et ne contenant pas de marque-pages.

   Rien de bien particulier, le programme est donné sur le listing 19.

Listing 19 : La classe `Scenario`

```
1  package fr.supaero.tags;
2
3  import java.util.HashSet;
4
5  public class Scenario {
6      public static void main(String[] args) {
7          Etiquette racine = new Etiquette("racine", null);
8          Etiquette vacances = new Etiquette("vacances", racine);
9          Etiquette mer = new Etiquette("mer", racine);
10         Etiquette montagne = new Etiquette("montagne", racine);
11         Etiquette camping = new Etiquette("camping", racine);
12
13         Et expression = new Et(new Et(new NoeudEtiquette(vacances),
14                                       new Ou(new NoeudEtiquette(mer),
15                                              new NoeudEtiquette(montagne))),
16                                new Not(new NoeudEtiquette(camping)));
17
18         VisiteurAfficher va = new VisiteurAfficher();
19         expression.accepter(va);
20
21         VisiteurEvaluer ve = new VisiteurEvaluer(racine);
22         expression.accepter(ve);
23         HashSet<MarquePage> resultat = ve.getResultat();
24
25         for (MarquePage mp: resultat) {
26             System.out.println(mp);
27         }
28     }
29 }
```

7. on souhaite introduire une opération qui peut lever une exception sous contrôle. Est-ce possible avec la solution développée précédemment ?

   A priori ce n'est pas possible, car les opérations définies dans l'interface `VisiteurNoeud` ne lèvent pas d'exception. Or si on redéfinit une méthode, on ne peut que lever des exceptions sous contrôle spécialisant les exceptions levées par la méthode de la classe mère. Dans notre cas, on ne peut donc pas immédiatement écrire des méthodes `visiter` qui lèvent des exceptions.

3 LHC at home

Remarque : cet exercice propose une modélisation de notions de physique théorique. Il ne s'agit pas d'une étude rigoureuse sur le sujet. En particulier, des simplifications ont été faites pour ne pas alourdir le sujet. **Il est plus que conseillé de lire l'énoncé en entier avant de répondre aux questions.**

Lors d'un séjour à Caltech (*California Institute of Technology*), vous rencontrez le Dr. Sheldon Cooper (cf. figure 14), un chercheur en physique théorique. Le Dr. Cooper souhaiterait disposer d'une application permettant de simuler la création de nouvelles particules afin de vérifier ses fascinantes hypothèses sur la théorie des cordes. Comme le travail de conception et de développement d'une application informatique n'est pas assez noble pour lui, il vous demande de lui développer cette application. Après tout cela semble dans vos cordes, vous n'avez qu'un diplôme d'ingénieur.

FIGURE 14 : Sheldon Cooper dans *The Big Bang Theory* ©CBS 2007-2010

Il faut donc que vous vous replongiez dans vos cours de physique de 1A. En physique, selon le modèle standard, toutes les particules possèdent un spin qui est une propriété intrinsèque à chaque particule et une charge électrique qui peut être nulle. Parmi les particules, on distingue les fermions et les bosons. Les fermions ont un spin demi-entier et les bosons ont un spin entier. Les fermions sont associés à la matière et les bosons aux forces fondamentales.

Les fermions sont séparés en deux catégories : les quarks et les leptons. Parmi les quarks, on trouve par exemple les quarks up, down, charm et strange. Ils ont une charge électrique non nulle. Parmi les leptons, on trouve les neutrinos qui n'ont pas de charge électrique et les électrons (électron, muon, tau) qui ont une charge électrique négative.

Parmi les bosons, on trouve le photon, les gluons et les bosons W^-, W^+ et Z^0. La théorie prédit l'existence de deux bosons supplémentaires, le boson de Higgs et le graviton, mais ils n'ont pas encore été observés.

Des fermions peuvent être composés d'autres fermions. Il s'agit par exemple de la famille des hadrons qui sont composés de quarks. Parmi les hadrons, les neutrons sont composés d'un quark up et de deux quarks down et les protons sont composés de deux quarks up et d'un quark down.

Un atome est composé d'un ensemble d'électrons et d'un noyau, lui-même composé d'un ensemble de protons et de neutrons

Les particules peuvent être soumises à des interactions élémentaires qui peuvent être de quatre types : interaction électromagnétique, interaction nucléaire forte, interaction nucléaire faible et gravitation. Ces interactions sont transportées par des bosons.

Le LHC (*Large Hadron Collider*) que vous devez simuler dans votre application a pour objectif de vérifier un certain nombre de théories en faisant entrer en collision des particules, ici des hadrons. Le LHC

dispose également d'un certain nombre de détecteurs qui observent les particules, dont les 4 principaux sont ATLAS, ALICE, CMS et LHCb.

1. proposer un diagramme de classes d'analyse représentant le domaine étudié. On fera apparaître les classes, les relations entre classes, les éventuels 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.
   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 15. Il n'y avait rien de particulier sur le diagramme à construire, c'était assez « classique » par rapport aux examens des années précédentes.
   Quelques remarques :
   – j'ai utilisé des classes abstraites pour représenter les fermions, leptons, quarks etc. car ce sont des entités qui ne sont pas concrètes. Je me suis permis de faire de même pour le boson de Higgs et le graviton ☺
   – seule `Particule` avait des attributs, `spin` et `charge`, qui correspondent aux caractéristiques de toutes les particules.
   – je n'ai pas ajouté sur le diagramme toutes les contraintes (les neutrinos ont une charge électrique nulle par exemple), car je manquais de place.
   – on pourrait remarquer le fait que l'agrégation entre `Proton` et `Up` et `Down` est redondante de l'agrégation entre `Hadron` et `Quark`. Il n'existe pas en effet de moyen en UML de préciser que les premières agrégations sont un raffinement de la seconde.
2. un scénario d'utilisation de votre simulateur est le suivant : le LHC crée deux protons et les accélère en leur donnant des facteurs d'accélération opposés. Au bout d'un certain temps, le LHC va déclencher la collision des deux protons en les injectant dans la boucle principale de l'accélérateur. Lors de la collision, représentée par un appel de méthode du LHC sur le premier proton, les deux protons vont émettre deux bosons Z^0 (ou W, mais nous supposerons que ce sont des bosons Z^0 ici) qui vont se combiner pour donner naissance à un boson de Higgs.

   Représenter le scénario précédent par un diagramme de séquence.

   Un diagramme de séquence est présenté sur la figure 16. Le scénario proposé était bien sûr simplifié. En particulier, ce sont les quarks des protons qui vont interagir via un boson pour produire le boson de Higgs et non pas les protons.
   La question n'était pas si triviale. En particulier, il fallait bien prendre le parti que nous construisons un simulateur et donc que c'est le LHC simulé qui va demander à un proton (ici le premier créé) d'entrer en collision avec le deuxième proton. De la même façon, c'est le premier proton qui est en charge de gérer la collision et de demander aux bosons de se combiner par exemple. Il me semble que c'est la solution la plus simple dans un premier temps.
3. on s'intéresse maintenant au cycle de vie d'un des détecteurs, ALICE. Au démarrage du détecteur, une phase de vérification interne est déclenchée. Durant cette phase, une vérification de tous les composants d'ALICE est effectuée. Si un problème est détecté, ALICE passe dans un état de maintenance durant lequel les composants sont réparés et à la fin de cet état de maintenance on retourne à la phase de vérification interne.
   Si la phase de vérification interne se déroule normalement, ALICE attend d'éventuelles collisions. Lorsqu'une collision se produit, ALICE analyse les observations qu'il a effectué et envoie les résultats au centre de traitement du LHC (cette opération prend beaucoup de temps). Il retourne ensuite en état d'observation s'il reste des expériences à réaliser (modélisé par un booléen `exp`). S'il n'y a

plus d'expérience à réaliser, ALICE s'arrête. Un opérateur peut également à tout moment arrêter le détecteur lors de cette phase de fonctionnement normale.

Construire un diagramme de machines d'états représentant le cycle de vie d'ALICE.

Le cycle de vie proposé a été en grande partie inventé et n'est pas réaliste. Un diagramme d'états-transitions modélisant le cycle de vie d'ALICE est présenté sur la figure 17. Rien de bien particulier ici, il fallait faire attention aux éventuels problèmes de non-déterminisme.

Références

[1] Apache log4j. http://logging.apache.org/log4j/.

[2] A. Bonnet and J. Luneau. *Aérodynamique : théories de la dynamique des fluides*. Cépaduès, 1989.

[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.

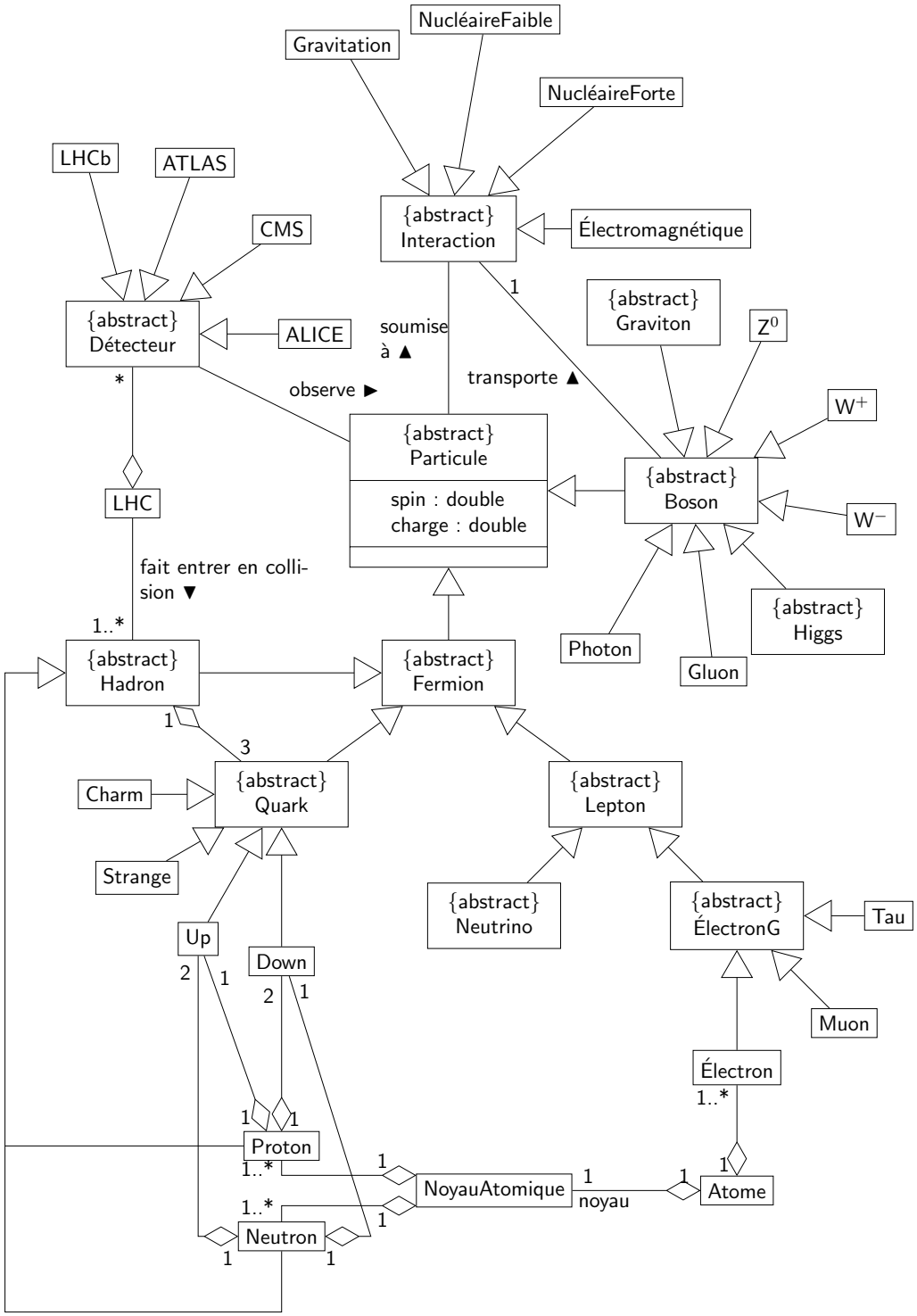

FIGURE 15 : Un diagramme de classes d'analyse du problème

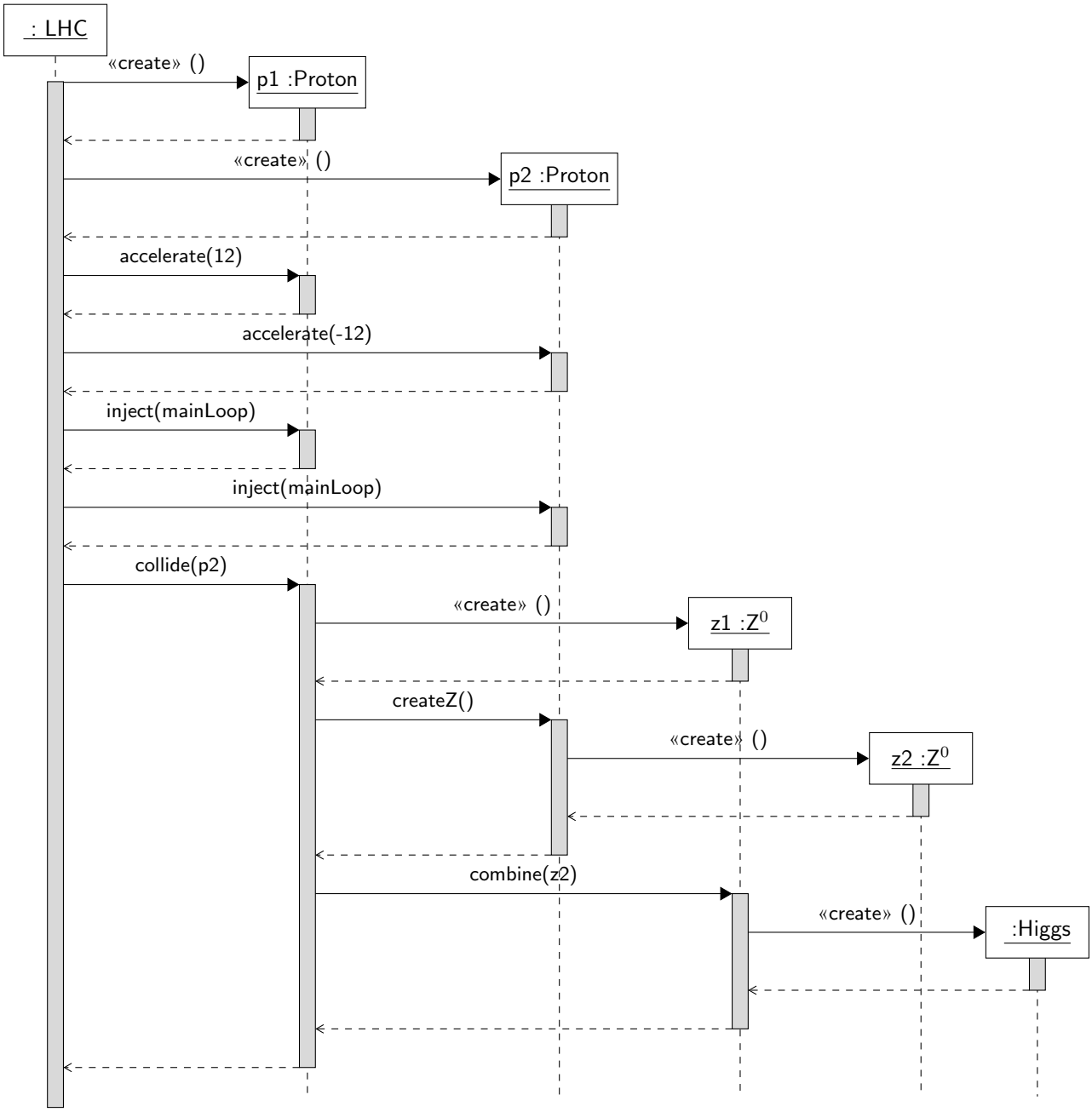

FIGURE 16 : Un diagramme de séquence représentant la collision

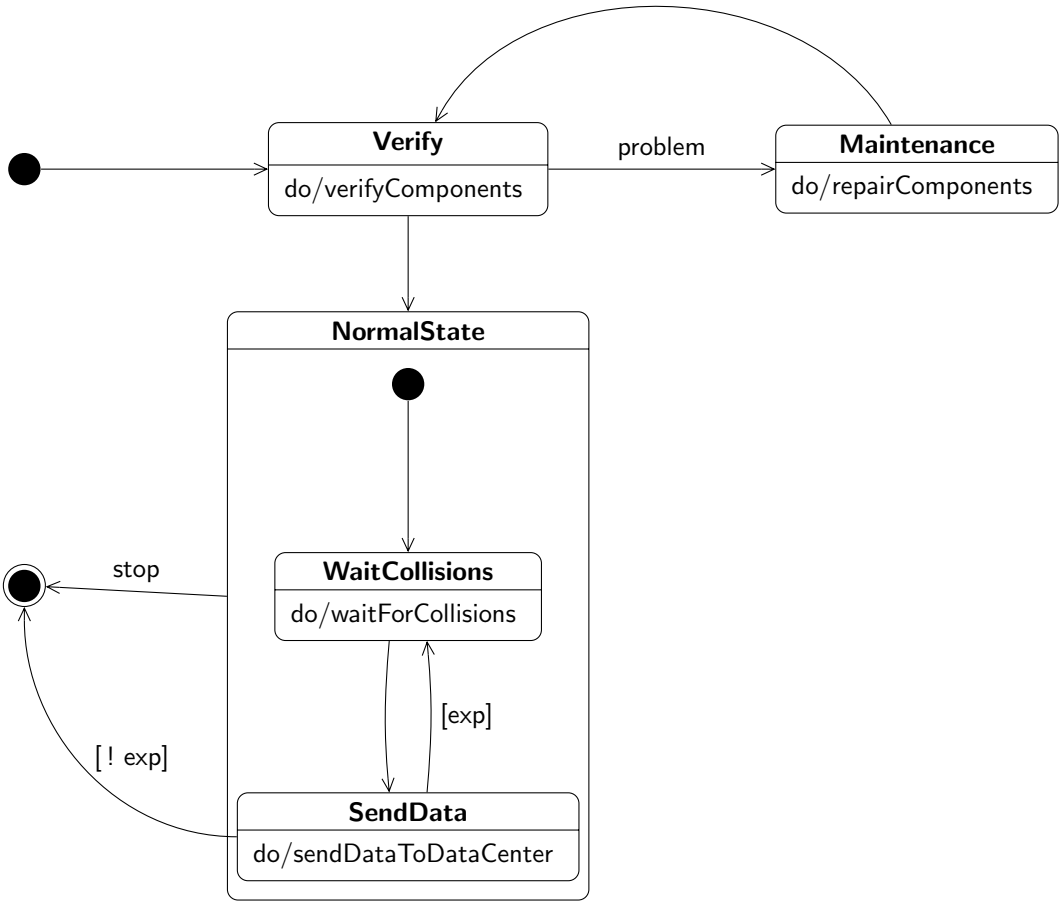

FIGURE 17 : Diagramme d'états-transitions modélisant le cycle de vie d'ALICE