Examen de Conception/Programmation OO SUPAERO 2A

Christophe Garion <garion@supaero.fr> 1 mars 2005

Ce document est un corrigé possible de l'examen de conception et programmation orientées objet.

Exercice 1

1.1 Présentation du problème

PROLOG est un langage déclaratif permettant de faire de la programmation logique (on ne s'étendra pas plus sur le sujet). Tout comme JAVA, PROLOG est *interprété*, c'est-à-dire que l'on ne produit pas du code directement exécutable sur une machine, mais du « code » qui sera exécuté par un programme particulier que l'on appelle *interpréteur*. Nous souhaitons créer un interpréteur PROLOG dans un langage orienté objet, par exemple en JAVA pour pouvoir faire de la programmation logique directement dans ce langage.

Un programme PROLOG manipule différentes structures :
- des termes. Un terme peut être un atome, une variable ou un terme structuré qui est en fait l'application d'une fonction (au sens mathématique) à un ensemble de termes ;
- des littéraux PROLOG composés d'un nom de prédicat et d'un ensemble non vide de termes ;
- des clauses PROLOG qui sont composées d'un ensemble de littéraux appelé *corps* de la clause et d'un littéral appelé *tête* de la clause ;

Un atome est simplement une chaine de caractère, une variable est une chaîne de caractères commençant obligatoirement par « * ».

Un programme PROLOG est un ensemble de clauses PROLOG et éventuellement un ensemble de littéraux appelé but.

On peut charger un ou plusieurs programmes PROLOG dans notre interpréteur. L'interpréteur fait appel à un algorithme non déterministe particulier, appelé algorithme de Résolution pour « exécuter » le programme PROLOG. Un utilisateur extérieur peut alors poser des requêtes à l'interpréteur. Ces requêtes sont en fait des buts PROLOG. L'interpréteur fournit une réponse sous forme d'un ensemble de littéraux.

Il existe plusieurs stratégies pour appliquer la méthode de Résolution. Par exemple, la plupart des programmes PROLOG utilisent une stratégie dite *Linear Resolution with Selection Function*, mais il en existe d'autres : *Linear Resolution*, *Input Resolution* etc. Toutes ces stratégies ont une partie commune qui est une fonction que l'on appelle fonction d'unification.

Un utilisateur peut choisir d'introduire un programme PROLOG dans l'interpréteur grâce à un système interactif ou de charger un fichier texte contenant le programme. Dans ce dernier cas, le fichier est parcouru par un *parser* qui se charge de construire le programme PROLOG correspondant au texte du fichier.

1. proposer un diagramme *UML* de conception préliminaire (analyse, donc sans attributs ni méthodes) de l'ensemble des éléments décrits dans l'énoncé présentant les classes, les relations entre les classes, les éventuels rôles et multiplicités (ou cardinalités).

   Vous pourrez justifier par écrit les relations utilisées et modifier de façon mineure l'énoncé si celui-ci vous paraît ambigu ;

   Un diagramme d'analyse possible est présenté sur la figure 1. Quelques remarques particulières :
   - j'ai tout de suite précisé que la classe `Stratégie` était abstraite. En effet, le texte nous précisait qu'il existait différentes stratégies possibles (représentées par les classes concrètes héritant de

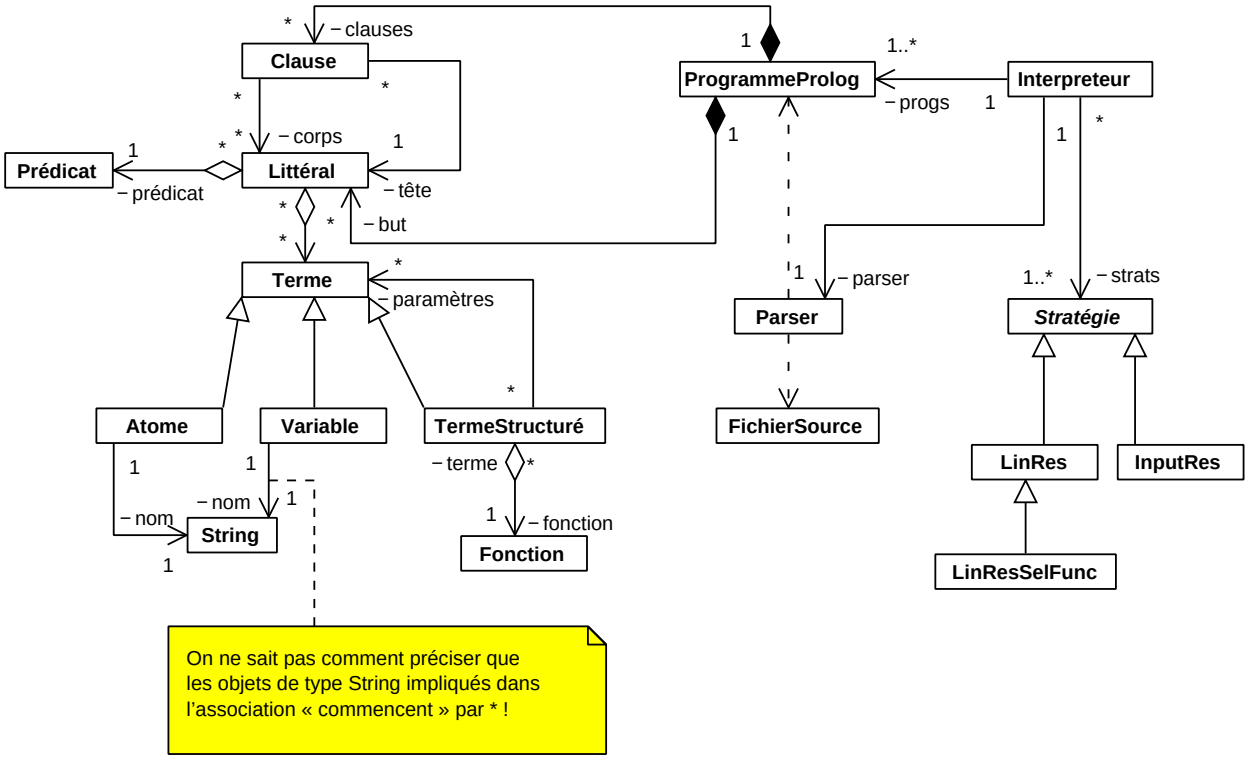

FIG. 1 – Diagramme d'analyse du problème posé

`Stratégie`), mais que celles-ci possédaient une opération commune. J'ai donc fait le choix d'une classe abstraite plutôt que d'une interface. Il est également à noter que j'ai supposé que la stratégie *Linear Resolution with Selection Function* était une stratégie dérivée de *Linear Resolution* ;
- j'ai essayé de mettre le plus de noms de rôles possibles, même lorsque ceux-ci étaient « redondants » avec le nom de la classe à laquelle ils se référaient dans l'association ;
- j'ai choisi de mettre une composition entre la classe `ProgrammeProlog` et les classes `Clause` et `Littéral`. Je suppose donc que si je détruis le programme, je détruis ses composants ;
- dans l'état de nos connaissances, on ne peut pas préciser sur ce diagramme de classes quelles sont les contraintes imposées aux objets de type `String` intervenant dans les classes `Variable` et `Atome` ;
- un semblant de « récursivité » était nécessaire pour représenter la notion de terme structuré, mais cela ne devait pas poser de problème particulier ;
- pour le *parser*, j'ai choisi de ne pas faire d'associations avec les classes `ProgrammeProlog` et `FichierSource`, mais des relations de dépendance. Je suppose ainsi que le *parser* ne « stocke » par forcément ces objets de façon « durable » mais les utilise en paramètre de méthodes et en retour de méthodes. La solution utilisant des associations était bien évidemment parfaitement valable.

2. écrire un diagramme de séquence représentant le scénario suivant :
   - un utilisateur charge un fichier dans l'interpréteur ;
   - l'interpréteur utilise le *parser* et celui-ci créé le programme contenant les clauses suivantes (on

représente une clause par corps → tête) :
– $predicat1(*X) \rightarrow predicat2(*X)$;
– $predicat1(atome1)$
et le but $predicat2(*Y)$. Le programme ainsi créé est retourné à l'interpréteur (vous pourrez abréger $predicat1$ et $predicat2$ respectivement par $p1$ et $p2$ pour alléger le diagramme. Idem pour $atome1$ que vous pourrez abréger en $a1$) ;
– utilisation de la méthode de résolution par l'interpréteur via une stratégie *Linear Resolution with Selection Function*. Cette méthode renvoie un résultat sous forme d'un atome et elle utilise la méthode d'unification.

Un diagramme solution est proposé sur la figure 2. Ce diagramme est très (trop ?) complet. Je n'attendais pas à ce que vous me précisiez par exemple qu'à la création d'un objet on récupère une référence vers cet objet. Dans la solution proposée, le *parser* créé les objets par type, ce qui est intuitivement faux. Une solution plus correcte aurait été de créer d'abord $*X$, puis $p1$, puis $p2$, puis les deux littéraux correspondant, puis la clause $c1$ et l'inclure dans le programme avant de passer aux créations concernant la « deuxième » ligne du fichier. J'ai émis quelques hypothèses de base :
– la construction d'une clause demande de fournir un littéral (la tête de la clause), cela était impliqué par la multiplicité proposée sur le diagramme de classe. J'ai ajouté une méthode `addCorps` qui permet d'ajouter les clauses servant de corps à la clause ;
– de la me façon, il existe des méthodes `addClause` et `addBut` dans la classe `ProgrammeProlog`
– pour être plus rigoureux, j'aurais du préciser que la réponse fourni par la résolution résulte de la création d'un atome particulier.

3. proposer un diagramme de conception détaillée (attributs et opérations typés) des classes `Stratégie` et `Interpréteur`. Ce diagramme devra faire apparaître l'implantation des relations existant avec les autres classes. Vous vous limiterez à la construction d'un petit nombre d'opérations sur ces classes.

   Une solution est proposée sur la figure 3. Rien de bien particulier, la classe `Stratégie` est abstraite, et sa méthode `résoudre` également, car son implantation est déterminée dans les sous-classes de `Stratégie`. La méthode `unifier` par contre est concrète, car on précisait bien que c'est une méthode commune à toutes les formes de stratégies.

   Je me suis appuyé sur le diagramme 2 pour construire la classe `InterpréteurProlog`, en particulier pour déterminer certaines de ses méthodes. Les types de retour éventuels des méthodes étaient facilement déterminables en utilisant l'énoncé.

   Lorsqu'un paramètre d'une méthode, un attribut ou un type de retour possédaient une multiplicité supérieure à 1, j'ai utilisé un « tableau » pour le représenter. Évidemment, cette solution d'analyse pourrait ensuite être implantée en utilisant non pas un tableau, mais par exemple une classe du type `java.util.ArrayList`.

Exercice 2

Remarques importantes : dans cet exercice, vous allez devoir écrire du code JAVA. Il est évident que les petites erreurs de syntaxe ne seront pas pénalisantes (qui peut se vanter d'écrire directement du code correct à chaque fois ?). De même, vous n'écrirez la documentation Javadoc d'une méthode ou d'une classe que si vous le jugez nécessaire (préciser la signification d'un paramètre ou du retour d'une méthode par exemple).

Enfin, le code à écrire n'est pas long, mais comprend quelques subtilités. Réfléchissez bien avant de vous lancer dans l'écriture.

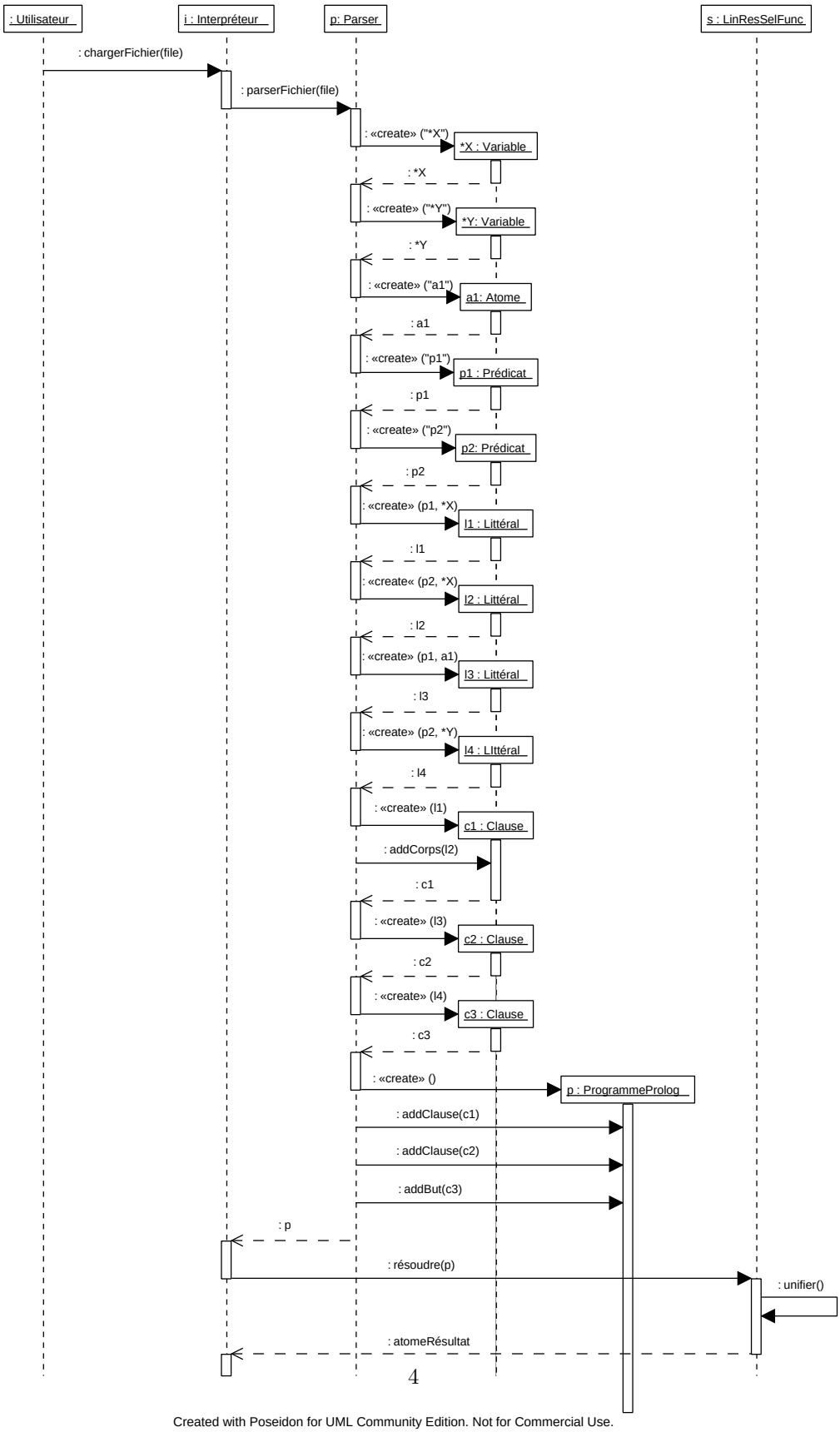

FIG. 2 – Diagramme de séquence représentant le scénario proposé en 1.2

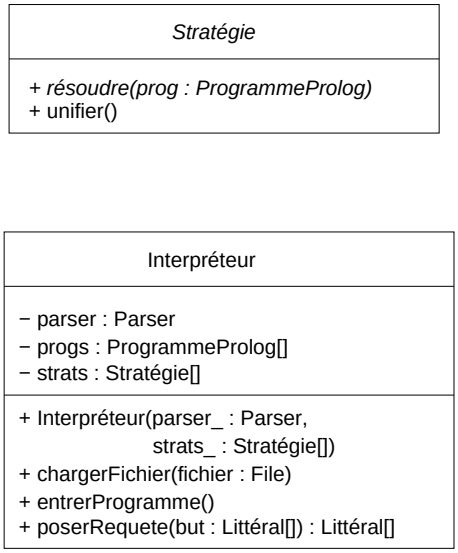

FIG. 3 – Diagramme de classe détaillé des classes `Stratégie` et `InterpréteurProlog`

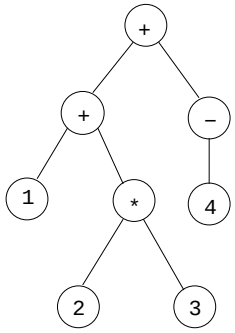

FIG. 4 – Représentation sous forme d'arbre de l'expression $1 + (2 * 3) + (-4)$

2.1 Présentation du problème

On souhaite développer un ensemble de classes permettant de *calculer* et d'*afficher* des expressions mathématiques. On se restreindra ici aux opérations habituelles de l'arithmétique sur les entiers à savoir l'addition, la multiplication, l'inversion du signe d'un entier. Une expression mathématique que l'on considérera par exemple sera $1 + (2 * 3) + (-4)$. On remarque tout d'abord que les constituants de l'expression peuvent être :

- des opérateurs binaires (qui prennent deux opérandes), comme $+$ et $*$;
- des opérateurs unaires comme $-$;
- des constantes qui sont des entiers.

Pour représenter une expression mathématique, on peut utiliser une structure d'arbre. 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 d'expressions, chaque nœud de l'arbre est soit un entier, soit un opérateur. Chaque nœud représentant un opérateur possède un nombre de nœuds fils égal à son arité. Par exemple, l'expression $1 + (2 * 3) + (-4)$ peut être représentée par l'arbre représenté sur la figure 4.

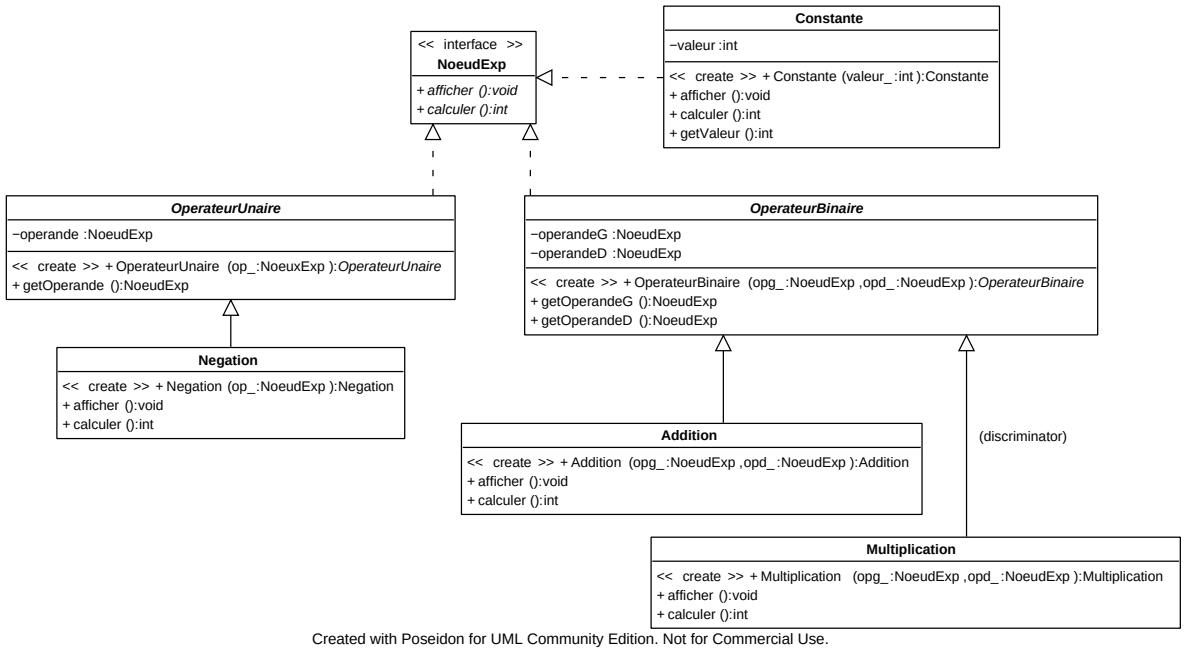

FIG. 5 – Diagramme de classes possible pour représenter les nœuds

2.2 Questions

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 au calcul des expressions et on veillera à construire un diagramme suffisamment générique (via l'héritage ou la réalisation d'interfaces). On fera ainsi apparaître les différents types d'opérations (unaire ou binaire), l'opération de multiplication, d'addition, de « négation », les constantes ainsi qu'un type de noeud générique. 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 5.

   Tous les nœuds représentant des expressions arithmétiques pouvaient rendre les mêmes services, i.e. afficher et renvoyer un résultat de calcul sous forme d'un entier. 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 opérande gauche et droit. Ces deux classes sont abstraites, car l'implantation des méthodes de calcul et d'affichage dépend de l'opérateur (on aurait toutefois pu embarquer une représentation de l'opérateur sous forme d'attribut pour l'affichage). 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 `Constante` représente les nœuds terminaux et est caractérisée par la valeur entière embarquée.

2. écrire la classe `Constante` en JAVA. Cette classe ne contient qu'un entier ;

Voici la classe `Constante`. Il n'y avait rien de particulier à signaler pour l'écriture de cette classe.

```
/**
 * <code>Constante</code> représente un noeud contenant une constante.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class Constante implements NoeudExp {

    private int valeur;

    /**
     * Créer une instance de <code>Constante</code>.
     *
     * @param valeur_ la valeur de la constante
     */
    public Constante(int valeur_) {
        this.valeur = valeur_;
    }

    /**
     * La valeur de la constante.
     *
     * @return un entier qui est la valeur de la constante
     */
    public int getValeur() {
        return this.valeur;
    }

    public void afficher() {
        System.out.print(this.getValeur());
    }

    public int calculer() {
        return this.getValeur();
    }
}
```

3. quelles critiques peut-on émettre sur ce modèle ? Vous réfléchirez en particulier à l'ajout d'une nouvelle opération sur les expressions (autre que le calcul et l'affichage) ;

   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'affichage par exemple. Enfin, supposons que l'on veuille ajouter une nouvelle opération sur les nœuds (en plus de l'affichage et du calcul) : on est obligé de réécrire toutes les classes et de les recompiler. Il serait plus efficace d'ajouter de nouvelles opérations sans modifier les nœuds.

4. pour pallier les problèmes évoqués dans la question précédente, on décide d'appliquer un patron

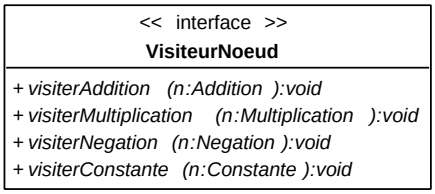

FIG. 6 – Interface `VisiteurNoeud`

de conception particulier, le *visiteur* [1]. 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 la multiplication, une à l'addition, une à la négation et une aux constantes. Ce sont ces méthodes qui « effectueront » `op` sur les différents nœuds. On les nommera par convention `visiterMultiplication`, `visiterAddition`, `visiterNegation`, `visiterConstante`. 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 6).

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 `Multiplication` est donné sur la figure 7. Évidemment, dans l'appel à `visiterMultiplication`, on risque d'appeler une méthode particulière sur les opérandes de la multiplication ...

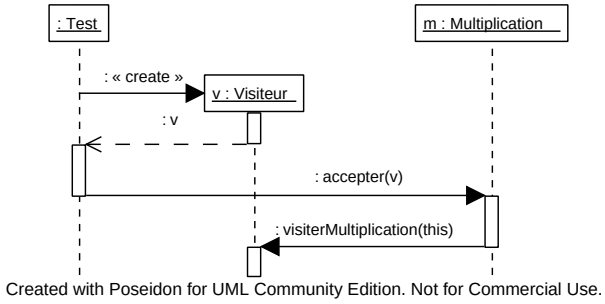

FIG. 7 – Diagramme de séquence présentant une utilisation de `Visiteur`

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 cette dernière, il faut trouver un moyen de conserver la valeur de l'expression) ;

Le diagramme de classes est présenté sur la figure 8. Rien de bien particulier, il fallait juste penser à utiliser un attribut dans `VisiteurCalcul` pour stocker le résultat de l'évaluation de l'expression.

5. écrire les classes et interfaces correspondant au diagramme construit à la question précédente ;

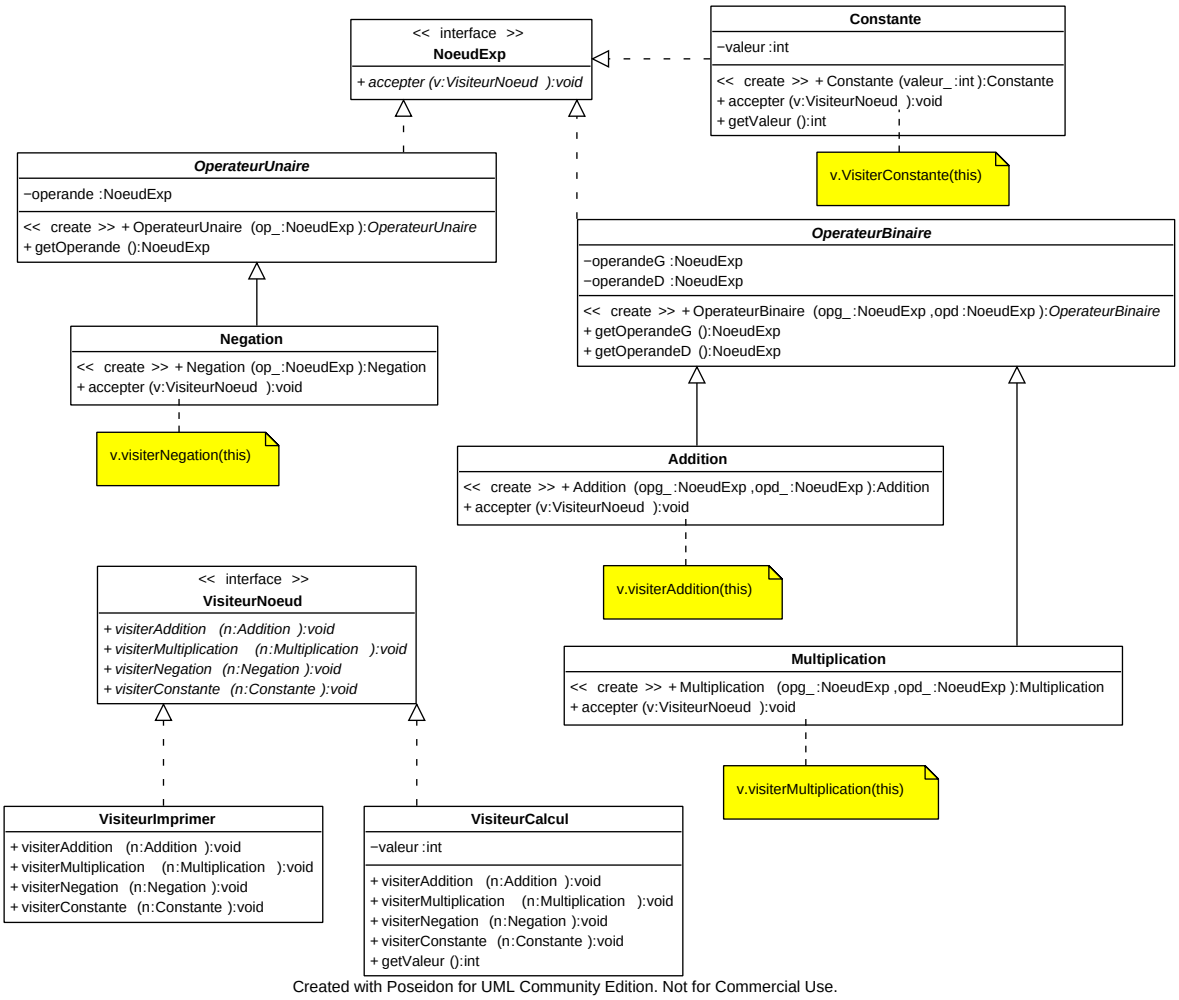

FIG. 8 – Diagramme de classe faisant intervenir les visiteurs

Voici l'interface `NoeudExp` :

```
/**
 * <code>NoeudExp</code> représente un noeud pouvant accepter
 * un visiteur.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public interface NoeudExp {

    /**
     * <code>accepter</code> permet d'accepter un visiteur sur le
     * noeud pour effectuer une opération.
     *
     * @param v le visiteur effectuant l'opération
     */
    public void accepter(VisiteurNoeud v);
}
```

Voici la classe abstraite `OperationUnaire` :

```
/**
 * <code>OperationUnaire</code> représente une opération à un opérande.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public abstract class OperationUnaire implements NoeudExp {

    private NoeudExp operande;

    /**
     * Créer une nouvelle <code>OperationUnaire</code>.
     *
     * @param op_ le noeud successeur
     */
    public OperationUnaire(NoeudExp op_) {
        this.operande = op_;
    }

    /**
     * L'opérande de l'opération.
     *
     * @return un <code>NoeudExp</code> représentant l'opérande
     */
    public NoeudExp getOperande() {
        return this.operande;
    }
}
```

Voici la classe abstraite `OperationBinaire` :

```
/**
 * <code>OperationBinaire</code> représente une opération à deux opérandes.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public abstract class OperationBinaire implements NoeudExp {

    private NoeudExp operandeG;

    private NoeudExp operandeD;

    /**
     * Créer une nouvelle <code>OperationBinaire</code>.
     *
     * @param opg_ le noeud représentant l'opérande gauche
     * @param opd_ le noeud représentant l'opérande droit
     */
    public OperationBinaire(NoeudExp opg_, NoeudExp opd_) {
        this.operandeG = opg_;
        this.operandeD = opd_;
    }

    /**
     * L'opérande gauche de l'opération
     *
     * @return le noeud représentant l'opérande gauche
     */
    public NoeudExp getOperandeG() {
        return this.operandeG;
    }

    /**
     * L'opérande droit de l'opération
     *
     * @return le noeud représentant l'opérande droit
     */
    public NoeudExp getOperandeD() {
        return this.operandeD;
    }
}
```

Voici la classe `Constante` :

```
/**
 * <code>Constante</code> est un noeud représentant un entier dans
 * une expression.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
```

```
 * @version 1.0
 */
public class Constante implements NoeudExp {

    private int valeur;

    /**
     * Créer une <code>Constante</code>.
     *
     * @param valeur_ la valeur de la constante
     */
    public Constante(int valeur_) {
        this.valeur = valeur_;
    }

    /**
     * La valeur de la constante.
     *
     * @return un entier qui est la valeur de la constante
     */
    public int getValeur() {
        return this.valeur;
    }

    public void accepter(VisiteurNoeud v) {
        v.visiterConstante(this);
    }
}
```

Voici la classe `Negation` :

```
/**
 * <code>Negation</code> permet de trouver l'opposé d'un entier.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class Negation extends OperationUnaire {

    public Negation(NoeudExp op_) {
        super(op_);
    }

    public void accepter(VisiteurNoeud v) {
        v.visiterNegation(this);
    }
}
```

Voici la classe `Multiplication` :

```
/**
```

```
 * <code>Multiplication</code> représente la multiplication de deux entiers.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class Multiplication extends OperationBinaire {

    public Multiplication(NoeudExp opg_, NoeudExp opd_) {
        super(opg_, opd_);
    }

    public void accepter(VisiteurNoeud v) {
        v.visiterMultiplication(this);
    }
}
```

Voici la classe `Addition` :

```
/**
 * <code>Addition</code> représente l'addition de deux entiers.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */

public class Addition extends OperationBinaire {

    public Addition(NoeudExp opg_, NoeudExp opd_) {
        super(opg_, opd_);
    }

    public void accepter(VisiteurNoeud v) {
        v.visiterAddition(this);
    }
}
```

Voici l'interface `VisiteurNoeud` :

```
/**
 * <code>VisiteurNoeud</code> "décrit"" les différents types de noeuds
 * qui peuvent être rencontrés par un visiteur.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public interface VisiteurNoeud {

    public void visiterNegation(Negation n);
    public void visiterAddition(Addition n);
    public void visiterMultiplication(Multiplication n);
    public void visiterConstante(Constante n);
```

```
}
```

Voici la classe `VisiteurAfficher` :

```
public class VisiteurAfficher implements VisiteurNoeud {

    public void visiterNegation(Negation n) {
        System.out.print("(-");
        n.getOperande().accepter(this);
        System.out.print(")");
    }

    public void visiterAddition(Addition n) {
        n.getOperandeG().accepter(this);
        System.out.print(" + ");
        n.getOperandeD().accepter(this);
    }

    public void visiterMultiplication(Multiplication n) {
        n.getOperandeG().accepter(this);
        System.out.print(" * ");
        n.getOperandeD().accepter(this);
    }

    public void visiterConstante(Constante n) {
        System.out.print(n.getValeur());
    }
}
```

Voici la classe `VisiteurCalcul` :

```
public class VisiteurCalcul implements VisiteurNoeud {

    private int valeur;

    public void visiterNegation(Negation n) {
        n.getOperande().accepter(this);
        this.valeur = 0 - this.valeur;
    }

    public void visiterAddition(Addition n) {
        n.getOperandeG().accepter(this);
        int v1 = this.getValeur();
        n.getOperandeD().accepter(this);
        int v2 = this.getValeur();
        this.valeur =  v1 + v2;
    }

    public void visiterMultiplication(Multiplication n) {
        n.getOperandeG().accepter(this);
        int v1 = this.getValeur();
```

```
        n.getOperandeD().accepter(this);
        int v2 = this.getValeur();
        this.valeur =  v1 * v2;
    }

    public void visiterConstante(Constante n) {
        this.valeur = n.getValeur();
    }

    public int getValeur() {
        return this.valeur;
    }
}
```

Il n'y avait rien de particulier à noter. La classe la plus difficile à écrire était bien sûr `VisiteurCalcul`, car le type de retour des méthodes de visite était **void**.

6. écrire un programme de test construisant l'expression $1+(2*3)+(-4)$, la calculant, puis l'affichant ; Voici le programme de test :

```
public class TestVisiteur {
    public static void main(String[] args) {

        Addition exp = new Addition(new Addition(new Constante(1),
                                                 new Multiplication(
                                                                    new Constante(2),
                                                                    new Constante(3))),
                                    new Negation(new Constante(4)));

        VisiteurNoeud v1 = new VisiteurAfficher();
        exp.accepter(v1);

        VisiteurCalcul v2 = new VisiteurCalcul();
        exp.accepter(v2);
        System.out.println("Valeur de l'expression : " + v2.getValeur());
    } // end of main()
}
```

7. on souhaite introduire une opération qui peut lever une exception. 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 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.

Références

[1] 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.