Examen de Conception/Programmation OO SUPAERO 2A

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

Cet examen est composé de deux parties indépendantes. Nous vous conseillons d'y consacrer la même durée. Tous les documents sont autorisés.

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.

1.2 Questions

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 ;
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.

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.

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.

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 1.

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

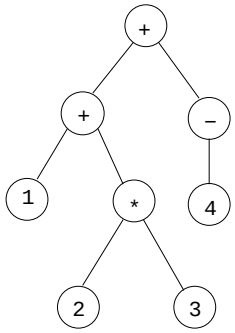

FIG. 1 – Représentation sous forme d'arbre de l'expression $1 + (2 * 3) + (-4)$

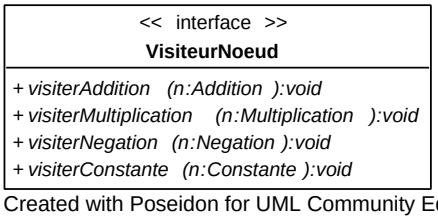

FIG. 2 – Interface `VisiteurNoeud`

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

2. écrire la classe `Constante` en JAVA. Cette classe ne contient qu'un entier ;
3. quelles critiques peut-on émettre sur ce modèle ? Vous réfléchirez en particulier à l'ajout d'une nouvelle opération (autre que le calcul et l'affichage) sur les expressions ;
4. pour pallier les problèmes évoqués dans la question précédente, on décide d'appliquer un patron 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 2).
   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 3. Évidemment, dans l'appel à `visiterMultiplication`, on risque d'appeler une méthode particulière sur les opérandes de la multiplication ...

   Modifier les classes que vous aviez proposées en question 1 et introduire les deux classes de visiteurs

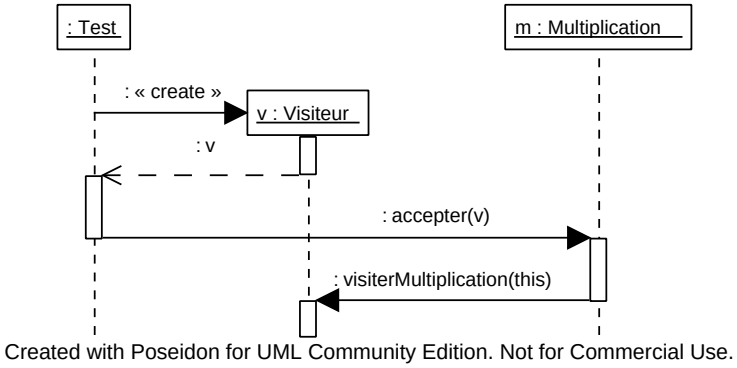

FIG. 3 – Diagramme de séquence présentant une utilisation de `Visiteur`

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) ;

5. écrire les classes et interfaces correspondant au diagramme construit à la question précédente ;
6. écrire un programme de test construisant l'expression $1+(2*3)+(-4)$, la calculant, puis l'affichant ;
7. on souhaite introduire une opération qui peut lever une exception. Est-ce possible avec la solution développée précédemment ?

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.