Université Nice Sophia Antipolis

CNRS — Centre National de la Recherche Scientifique

LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

LAMP : VERS UN LANGAGE DE DÉFINITION DE MÉCANISMES DE PROTECTION POUR LES LANGAGES DE PROGRAMMATION À OBJETS

Gilles Ardourel, Pierre Crescenzo, Philippe Lahire

Projet OCL

Rapport de recherche
I3S/RR–2002-48–FR

Septembre 2002

LABORATOIRE I3S: Les Algorithmes / Euclide B – 2000 route des Lucioles – B.P. 121 –
06903 Sophia-Antipolis Cedex, France – Tél. (33) 492 942 701 – Télécopie : (33) 492 942 898
`http://www.i3s.unice.fr/I3S/FR/`

RÉSUMÉ :
L'encapsulation et la modularité jouent un grand rôle dans le succès des langages à objets. Ces notions s'accompagnent naturellement de mécanismes de protection, souvent particulièrement complexes, qui définissent ou limitent l'accès à certaines entités décrites par le langages telles les objets, les méthodes, les classes... L'objectif de ce rapport est de proposer un langage simple, mais très expressif et facilement extensible, de définition de mécanismes de protection dans les langages à classes.

MOTS CLÉS :
protection, contrôle d'accès, encapsulation, langage à classes

ABSTRACT:
Encapsulation and modularity play a prominent role in the success of object-oriented languages. These concepts are supported by access control mechanisms which are often complex and, define or restrict access to entities like objects, methods, classes... This paper aims at describing a simple, expressive and extensible language for defining access control mechanisms in class-based languages.

KEY WORDS :
protection, access control, encapsulation, class-based language

LAMP : vers un langage de définition de mécanismes de protection pour les langages de programmation à objets

Gilles Ardourel* — Pierre Crescenzo — Philippe Lahire****

** LIRMM (UM2/CNRS)*
Équipe Représentation Par Objets
161, rue Ada
F-34392 Montpellier cedex 5
ardourel@lirmm.fr
** Laboratoire I3S (UNSA/CNRS)*
Projet Objets et Composants Logiciels
2000, route des lucioles
Les Algorithmes, Bâtiment Euclide B
BP 121 F-06903 Sophia-Antipolis cedex
{Pierre.Crescenzo,Philippe.Lahire}@unice.fr

RÉSUMÉ. *L'encapsulation et la modularité jouent un grand rôle dans le succès des langages à objets. Ces notions s'accompagnent naturellement de mécanismes de protection, souvent particulièrement complexes, qui définissent ou limitent l'accès à certaines entités décrites par les langages telles les objets, les méthodes, les classes... L'objectif de cet article est de proposer un langage simple, mais très expressif et facilement extensible, de définition de mécanismes de protection dans les langages à classes.*

ABSTRACT. *Encapsulation and modularity play a prominent role in the success of object-oriented languages. These concepts are supported by access-control mechanisms, which are often complex, and define or restrict access to entities like objects, methods, classes... This paper aims at describing a simple, expressive, and extensible language for defining access control mechanisms in class-based languages.*

MOTS-CLÉS : *protection, contrôle d'accès, encapsulation, langage à classes.*

KEYWORDS: *protection, access control, encapsulation, class-based language.*

1. Introduction

L'encapsulation constitue l'un des principes fondamentaux des langages de programmation à objets. Elle offre la capacité de localiser une information là où elle est pertinente et peut être traitée. Elle participe donc à la modularité mais également à une plus grande séparation entre l'interface et l'implémentation des classes.

Les langages à objets proposent, pour mettre en œuvre l'encapsulation, des techniques qui consistent à associer à certaines entités du programme (tels les objets, les méthodes ou les classes) des directives de protection (comme `public` ou `protected` en Java [FLA 99, GOS 00] ou `export` en Modula [HAR 92]) que les mécanismes du langage interprètent pour permettre ou limiter l'accès à certains aspects des objets suivant le profil de l'entité qui y accède. Une utilisation correcte de ces mécanismes de protection permet de réduire le couplage entre les composants et assure ainsi de bonnes conditions de maintenance ou de réutilisation [COA 91, BOO 98]. Les moyens d'expression de la protection dans les langages existants sont bien souvent trop faibles pour les concepteurs, programmeurs et documenteurs de logiciels. Ceux-ci ne disposent trop souvent que de `public`, `protected` ou `private` et de leurs contreparties UML [LAI 97, Obj 01] `+`, `#` et `-` qui n'ont pas toujours la même signification d'un langage à l'autre, voire d'une version d'un même langage à la suivante.

Nous pensons que les mécanismes de protection devraient être plus expressifs, et qu'un langage de définition de mécanismes de protection pourrait bénéficier à la fois aux concepteurs et programmeurs d'applications et aux métaprogrammeurs qui définissent des langages, en décrivant clairement une sémantique et en assurant la continuité des spécifications de protection entre la conception et l'implémentation. Une telle explicitation de ces mécanismes entraînerait aussi une meilleure compréhension des concepteurs, qui les utiliseraient donc plus souvent et à meilleur escient, et des programmeurs qui traduiraient mieux les spécifications des concepteurs.

Cet article a pour objectif de proposer un langage simple, nommé *Lamp*, de définition de mécanismes de protection. Dans la section 2, nous proposons un ensemble de métainformations ainsi que les contraintes d'utilisation associées, qu'il faut recenser dans le langage de programmation cible ; nous nous appuyons pour cela sur les besoins de langages existants, en particulier Java. La section 3 définit la grammaire de Lamp pour déclarer des protections et montre comment simuler des mécanismes existants. Enfin, nous concluons dans la section 5 en donnant les perspectives de nos travaux.

2. Métainformations pour la mise en œuvre des protections

Dans notre approche la *protection*, consiste, pour une entité donnée E (une entité est un des éléments qui constituent un programme), à définir le(s) types(s) d'action que d'autres entités (y compris éventuellement E) peuvent effectuer sur elle. Dans les langages de programmation, cette protection est mise en œuvre de manière implicite, par les règles propres au langage lui-même (exemple : l'accès à un attribut redéfini en

Java) et de manière explicite par des déclarations de protection (exemple : les modificateurs `private`, `protected` ou `final` en Java). Nous appelons **accédée** l'entité qui définit sa protection et **accédante** celle qui réclame un service à l'accédée. Le mécanisme de protection a pour charge de contrôler que l'accédante n'outrepasse pas les droits définis par l'accédée.

Lamp est un langage de définition de la protection pour les langages à objets. Une première étape consiste à lui associer les métainformations qui seront nécessaires, plus tard, à son utilisation par le programmeur ; c'est le rôle du métaprogrammeur. En effet, la flexibilité de Lamp provient de sa capacité à s'appuyer sur une définition précise des entités manipulées par les langages à objets, sur les opérations qu'il est possible de réaliser sur celles-ci et sur les relations qui sont susceptibles d'être placées entre ces entités. Avant de pouvoir définir une protection, il faut donc définir les types d'entité, d'opération et de relation ainsi que les règles qui régissent leur utilisation.

2.1. *Types d'entité et d'opération avec leurs contraintes*

2.1.1. *Types d'entité et d'opération*

Les types d'entités qui constituent un langage de programmation, et que le programmeur veut pouvoir protéger les unes des autres, varient selon les langages même si certains comme les classes, les attributs ou les méthodes sont généralement présents ; il faut donc que le langage de protection proposé supporte un ensemble de types d'entité évolutif. Il en va de même pour les opérations que l'on veut pouvoir contrôler et qui ne sont pas identiques suivant le langage ciblé.

Dans la figure 1, nous proposons deux classifications : une pour les entités et une autre pour les opérations. Ces classifications sont constituées d'une partie générique suffisamment générale pour s'adapter aux langages à classes industriels classiques (Java [GOS 00], C++ [STR 97], Eiffel [MEY 94]...), et d'une partie spécifique qui étend par spécialisation la partie générique et qui dépend du langage de programmation ciblé ainsi que des protections à mettre en œuvre[1]. Dans la figure, cette partie est représentée par les feuilles des deux hiérarchies. Par ailleurs, chaque opération spécialisant la catégorie `Comportement` spécifie (à l'aide d'une information supplémentaire appelée *localisation* qui n'est pas spécifiée dans la figure) si elle ne peut s'appliquer que sur le receveur courant[2] ou seulement sur un objet distinct du receveur courant, ou bien sur les deux[3]. Notons que les opérations, par opposition aux entités, auront une seule instance, ou plus exactement une instance par localisation.

La partie encadrée de la figure représente des exemples de types d'entité (Monde, Paquet, Classe, Méthode et Attribut) et d'opération (Typage, Héritage, Redéfinition,

1. Il n'est pas nécessaire de capturer la sémantique des entités et opérations du langage qui ne seront pas exploitées pour définir des protections.
2. C'est-à-dire, pour une classe, sur l'objet courant.
3. L'ajout de cette propriété à un type d'opération nous permet d'éviter de compliquer inutilement l'arbre de classification de la figure 1.

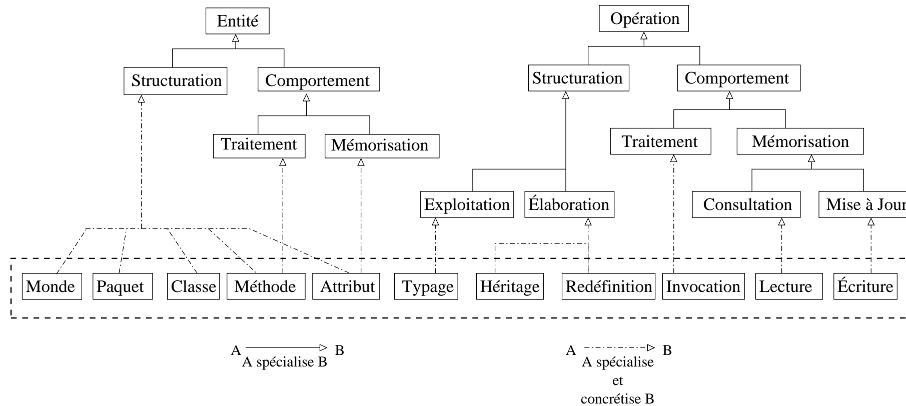

Figure 1. *Des types d'entité et d'opération*

Invocation, Lecture et Écriture) que peut mettre en œuvre un langage. Nous les avons choisies parce qu'elles sont des entités et opérations bien connues des langages à objets les plus courants. Le type d'entité Monde est un singleton fort utile qui inclut toutes les autres entités. Il est bien sûr possible d'envisager d'autres langages en étendant d'une manière différente les feuilles de cet arbre.

Nous devons aussi définir les contraintes du modèle. Il nous faut préciser qu'il est possible pour une entité de spécialiser deux branches de l'arbre ; par exemple une méthode peut être considérée comme une entité de structuration et de comportement. D'autre part, l'affectation des opérations possibles à une entité doit respecter plusieurs règles. Ainsi, une opération de structuration ne peut être appliquée que si l'entité est aussi de structuration. Une règle identique concerne les entités et opérations de comportement, de traitement et de mémorisation.

2.1.2. *Contraintes sur les opérations et entités*

Après la définition des types d'entité et d'opération, l'étape suivante consiste donc à associer les opérations qu'il est possible d'appliquer sur chaque type d'entité. Pour les entités proposées comme exemple dans la figure 1, nous pouvons définir les associations suivantes :

Type d'entité	**Type(s) d'opération possible(s)**
Classe	Structuration
Méthode	Héritage, Redéfinition et Invocation
Attribut	Héritage, Redéfinition, Lecture et Écriture

Examinons la ligne `Méthode`. Elle exprime la possibilité qu'une méthode (en tant qu'entité de structuration) soit héritée et redéfinie mais aussi invoquée (car elle est aussi une entité de comportement). Nous constatons, par contre, qu'il n'y a aucune entrée pour `Paquet`. En effet, c'est une entité sur laquelle aucune opération ne peut

être exercée et qui ne peut donc être vue que comme accédante et non comme accédée. En d'autres termes, les paquets ne peuvent pas être protégés dans notre exemple.

2.2. *Relations*

Un langage de programmation permet de lier des entités entre elles ; c'est par exemple le cas lorsque deux classes sont liées par un *héritage* ou par une *agrégation*, ou bien lorsqu'il est spécifié qu'une méthode *appartient* à une classe ou lorsque une classe *appartient* à un paquet ou encore quand un paquet *est inclus* dans un autre. Il est évidemment intéressant de pouvoir définir les protections entre des entités en tenant compte des relations qui peuvent lier celles-ci. Ainsi, de même que pour les entités et les opérations, il est souhaitable que l'ensemble des types de relation soit évolutif. Il nous faut donc là encore proposer un formalisme pour définir un type de relation. Nous recensons trois types de relation génériques :

– L'`Héritage` permet de spécifier toutes les formes d'héritage en allant du `inherit` d'Eiffel au `extends` ou à l'`implements` de Java en passant par le `:` de C++, mais aussi bien d'autres usages plus précis comme la spécialisation, la généralisation, voire les relations de *vue* ou de *version*.

– L'`Utilisation` permet de spécifier comment une entité en utilise une autre : nous pouvons citer comme exemples significatifs d'utilisation le lien de *composition* et celui d'*agrégation* entre classes ou encore l'`import` de Java.

– L'`Inclusion` permet de spécifier l'appartenance d'une méthode ou d'un attribut à une classe ou bien l'appartenance d'une classe à un paquet ou bien encore l'appartenance d'un paquet à un autre paquet, etc.

Ces trois catégories de relation définies représentent une classification minimale des relations pour notre approche. Celle-ci sera en général étendue en fonction du langage cible. La figure 2 donne un exemple d'extension possible de cette classification. L'approche est la même que pour les entités et les opérations. Les feuilles (partie encadrée) correspondent à quelques relations dont nous avons besoin si nous considérons le langage Java.

Les relations du langage cible nous servent notamment à définir des ensembles d'entités accédantes. Par exemple, étant donnée une entité $Classe1$, l'ensemble des classes héritant de $Classe1$ sera défini comme $Classe1_{\texttt{héritage}}$. On pourra aussi définir l'ensemble des classes héritant de $Classe1$ et des méthodes de ces dernières par : $Classe1_{\texttt{Héritage},\texttt{Inclusion}}$.

L'utilisation explicite de relations, notamment l'inclusion et l'héritage, pour déterminer les accédantes est à notre avis un des points forts de Lamp. En effet, l'interaction mal prise en compte par les langages de programmation, de ces deux relations et de la protection est à la base de la complexité de nombreux mécanismes [SNY 86].

Nous n'exploiterons pas ici la relation d'utilisation pour définir des ensembles d'accédantes. Dans les langages à objets les plus courants, la définition des droits

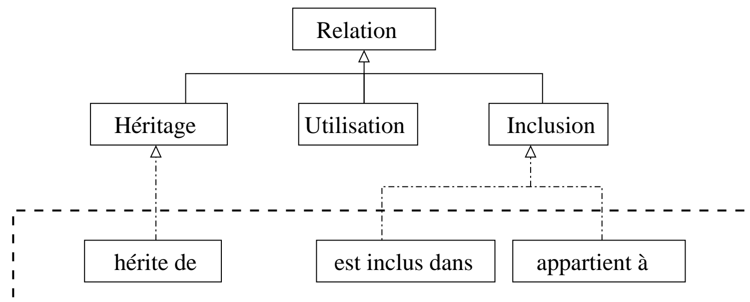

Figure 2. *Des types de relation*

prend traditionnellement appui sur le graphe d'héritage plutôt que de suivre le graphe d'utilisation[4]. Nous pourrions cependant envisager, à l'instar de certains systèmes de bases de données existants, d'autoriser un transfert de droits amoindris via la relation d'utilisation. Par exemple, si une classe `A` possède un attribut de type `B`, `A` pourrait alors transférer à ses utilisatrices une partie des droits qu'elle possède sur `B`. Notez cependant qu'une solution simple existe dans les langages à objets pour cela : il s'agit de définir des méthodes de `A` qui utilisent l'attribut de type `B` et qui sont rendues accessibles aux utilisatrices de `A`.

2.3. *Contraintes sur les relations et entités*

Une fois l'ensemble des types de relation connu, une autre étape consiste à expliciter les relations possibles entre chaque entité ; pour cela il s'agit de définir un ensemble de triplets : type d'entité servant de source, type de relation et type d'entité utilisé comme cible. Par exemple, à partir des feuilles définies dans les figures 1 et 2, nous pouvons préciser les règles suivantes :

Type d'entité source	Type de relation autorisée	Type d'entité cible
Paquet	est inclus dans	Monde
Classe	est incluse dans	Paquet
Classe	hérite de	Classe
Classe	est incluse dans	Classe
Méthode	appartient à	Classe
Classe	est incluse dans	Méthode
Attribut	appartient à	Classe

4. Quelques exceptions, spécifiques à leur langage, existent comme les membres `friends` de C++.

2.4. *Bilan*

Les métainformations concernant le langage de programmation cible que Lamp doit connaître sont :

– l'ensemble des types d'entité, de relation et d'opération,

– l'ensemble des couples (type d'entité et type d'opération) qui définissent, pour chaque entité, les opérations qu'il est possible de lui appliquer dans le langage et

– l'ensemble des triplets (type d'entité-source, type de la relation et type d'entité-cible) qui définissent comment les entités peuvent être liées entres elles dans le langage.

Intuitivement, l'expressivité des types d'entité, de relation et d'opération doit permettre de gérer des situations très diverses. Par exemple, soient une classe `C1` qui contient une méthode `M1` et un attribut `A1` et une classe `C2`. Il doit être possible d'exprimer que toute classe peut *hériter de* `C1`, que toute classe qui *hérite de* `C1` peut modifier le contenu de `A1` ou encore que toute classe qui *hérite de* `C2` peut invoquer la méthode `M1` de `C1`.

3. Lamp

Une fois fixés pour le langage cible, les types d'entité, de relation et d'opération, nous devons définir des protections dans le cadre de l'écriture d'un programme.

Nous nous plaçons dans l'hypothèse classique où chaque entité ne peut spécifier des droits que sur elle-même, assurant ainsi sa protection. Nous pourrions cependant envisager de décrire des protections sur des entités désignées en intention, ce qui permettrait de factoriser un certain nombre de déclarations de protection, mais avec les inconvénients suivants :

– La connaissance de la protection n'est plus directement associée à une entité, ce qui augmente la complexité du traitement pour un traducteur (interprète ou compilateur) et la difficulté de compréhension pour un humain. L'encapsulation de la protection est en effet perdue.

– Si nous décidons que les déclarations ne peuvent qu'ajouter des droits, les possibilités de se protéger seront plus réduites, par exemple dans le cas où un droit trop lâche a déjà été exprimé comme « tout le monde peut accéder à tout le monde ».

– Si nous autorisons les déclarations à retirer des droits, alors des règles de priorité des déclarations doivent être introduites aux dépens de la clarté et de la facilité de traitement.

C'est pourquoi, au moins dans un premier temps, nous considérons que, sans une déclaration explicite, aucun droit n'est donné.

3.1. *Déclaration de protection*

Une règle, nommée *déclaration de protection*, permet à une accédée d'octroyer des droits la concernant à ses accédantes.

La grammaire d'une déclaration de protection est donnée ci-dessous. Les tubes (|) signifient « ou », les symboles plus (+) signifient « de 1 à n fois » et les guillemets anglais (“ et ”) entourent les terminaux.

```
                      :- Accédée ( EnsembleDAccédantes ( Opération
                         ( “(C)” | “(O)” | “(A)” ) )+ )+
Accédée               :- Entité
EnsembleDAccédantes   :- Accédante | Accédante_Relation+ |
                         EnsembleDAccédantes “∪” EnsembleDAccédantes
Accédante             :- Entité
```

Cette `déclaration de protection` peut se traduire en langage naturel par : « l'entité `Accédée` confère le droit d'effectuer sur elle les opérations `Opération` aux entités appartenant à l'`EnsembleDAccédantes`. » Les mentions `(C)`, `(O)` ou `(A)` après une opération permettent de préciser si l'opération peut s'appliquer, respectivement, à l'objet courant[5] seulement, aux autres objets seulement, ou à tous les objets.

Les terminaux dépendent de l'application considérée, et sont constitués des noms des entités, opérations et relations. Les non-terminaux `Entité`, `Opération` et `Relation` dépendent du langage cible. En reprenant les définitions de notre exemple de langage cible, nous obtiendrions ainsi pour `Entité` :

```
Entité                :- Monde | Paquet | Classe | Méthode | Attribut
```

Nous pouvons bien sûr utiliser des catégories plus générales que celles du langage cible, par exemple `Inclusion` (cf. figure 2) si nous ne souhaitons pas distinguer la relation liant un paquet à une classe de celle liant une classe à une méthode. Ce sera le cas dans notre exemple qui se base sur Java. Mais si le langage cible choisi avait été C++, la relation d'inclusion entre classe et méthode aurait du être distinguée de celle concernant l'inclusion d'une classe dans une autre, pour décrire l'utilisation du mécanisme `friend`. Ce mécanisme peut conférer des droits à une classe et aux méthodes incluses dans cette classe, mais pas aux classes internes de cette classe.

3.2. *Exemples de définition de protection dans une application*

Voyons maintenant quelques exemples d'utilisation de Lamp. La figure 3 représente le diagramme de classes UML qui servira de base à ces exemples. Les annotations de *visibilité* d'UML (`+`, `#` et `-`) ne sont pas utilisées ici. Il conviendrait de posséder un vocabulaire plus riche que celui d'UML pour représenter les nouvelles protections définies avec Lamp, mais ce n'est pas l'objet de cet article.

5. Si le type d'entité accédé est `Classe`, alors on pourra parler de Receveur Courant.

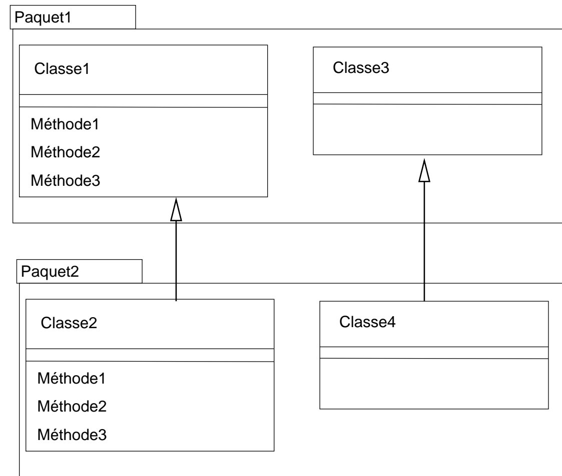

Figure 3. *Diagramme de classes UML sans annotation de* visibilité

Nous disposons de quatre classes réparties dans deux paquets, `Classe2` hérite de `Classe1` et `Classe4` hérite de `Classe3`. Voici un exemple de déclarations de protection correspondant à ce diagramme. Nous n'avons détaillé que le niveau des méthodes pour des raisons de simplicité et nous ne tenons donc pas compte ici des droits associés aux paquets et classes.

Dans la Classe1 :

`Méthode1`	$\mathtt{Classe1} \cup \mathtt{Classe1}_{\mathtt{inclusion,héritage}}$	`invocation(C)`
	`Monde`	`invocation(O)`
`Méthode2`	$\mathtt{Classe1} \cup \mathtt{Classe1}_{\mathtt{inclusion,héritage}}$	`invocation(C)`
	$\mathtt{Paquet1} \cup \mathtt{Paquet1}_{\mathtt{inclusion}}$	`invocation(O)`
`Méthode3`	$\mathtt{Classe1} \cup \mathtt{Classe1}_{\mathtt{inclusion}}$	`invocation(A)`

Dans la Classe2 :

`Méthode1`	$\mathtt{Classe2} \cup \mathtt{Classe2}_{\mathtt{inclusion,héritage}}$	`invocation(C)`
	`Monde`	`invocation(O)`
`Méthode2`	$\mathtt{Classe1} \cup \mathtt{Classe1}_{\mathtt{inclusion,héritage}}$	`invocation(C)`
	$\mathtt{Paquet2} \cup \mathtt{Paquet2}_{\mathtt{inclusion}}$	`invocation(O)`

Expliquons maintenant chaque déclaration une par une.

La `Méthode1` de `Classe1` confère :

– au travers de l'objet courant (de `Classe1`), le droit d'effectuer son `invocation` à l'entité `Classe1` et à toutes celles atteignables par `inclusion` et `héritage` depuis `Classe1`, c'est-à-dire toutes les méthodes de `Classe1` ainsi que toutes celles des sous-classes de `Classe1` et

– au travers d'autres références (de `Classe1`), le droit d'effectuer son `invocation` à toutes les entités atteignables par `inclusion` depuis `Monde`, c'est-à-dire par toute entité.

La `Méthode2` de `Classe1` confère :

– au travers de l'objet courant (de `Classe1`), les mêmes droits aux mêmes entités que la `Méthode1` de `Classe1` et

– au travers d'autres références (de `Classe1`), le droit d'effectuer son `invocation` à toute entité incluse dans `Paquet1`.

La `Méthode3` de `Classe1` confère, que ce soit au travers de l'objet courant (de `Classe1`), ou au travers d'autres références (de `Classe1`), le droit d'effectuer son `invocation`, à l'entité `Classe1` et à toutes celles atteignables par `inclusion` depuis `Classe1`, c'est-à-dire toutes les méthodes de `Classe1`.

3.3. *Déclaration de mots-clés*

Une simplification de ces déclarations consiste à recenser les déclarations les plus courantes, à les généraliser si nécessaire, et à leur associer un mot-clé. Pour illustrer cette idée, précisons tout d'abord que nous n'avons pas choisi au hasard les déclarations de notre exemple figure 3, nous les avons faites en pensant au langage Java.

La première déclaration, dans `Classe1`, est :

Méthode1	$\mathtt{Classe1} \cup \mathtt{Classe1}_{\mathtt{inclusion,h\acute{e}ritage}}$	invocation(C)
	Monde	invocation(O)

Elle peut être rapprochée d'une définition de primitive *publique*, c'est-à-dire qu'il est possible d'appeler cette méthode de n'importe quel point de l'application. Aussi pourrions-nous choisir d'associer à l'ensemble de ces deux déclarations de protection le mot-clé `public`.

La troisième déclaration est constituée de :

Méthode3	$\mathtt{Classe1} \cup \mathtt{Classe1}_{\mathtt{inclusion}}$	invocation(A)

La méthode concernée peut être appelée partout dans sa classe mais seulement dans sa classe. Cela correspond à une primitive *privée*, nous pourrions donc choisir le mot-clé `private`.

Enfin, la deuxième déclaration, la plus fine des trois, est :

Méthode2	$\mathtt{Classe1} \cup \mathtt{Classe1}_{\mathtt{inclusion,h\acute{e}ritage}}$	invocation(C)
	$\mathtt{Paquet1} \cup \mathtt{Paquet1}_{\mathtt{inclusion}}$	invocation(O)

Ici, la méthode peut être invoquée dans toutes les classes du paquet de sa classe ainsi que dans toutes les classes qui héritent de sa classe. Cela ressemble beaucoup à une primitive *protégée*, au moins au sens de Java, et nous pourrions donc associer à ces deux lignes le mot-clé `protected`.

Le mécanisme par défaut de Java utiliserait une déclaration assez proche :

Méthode1 Classe1 ∪ (Classe1$_{\text{inclusion,héritage}}$ ∩ Paquet1$_{\text{inclusion}}$)
invocation(C)
Paquet1 ∪ Paquet1$_{\text{inclusion}}$ invocation(O)

Maintenir un glossaire associant les mots-clés à leur sémantique décrite sous la forme d'un ensemble de déclarations de protection permet d'envisager l'enrichissement simple et progressif du langage de programmation par des mécanismes de protection. Cependant les déclarations associées à un mot-clé ne peuvent contenir de noms d'entité spécifiques comme `Classe1`. Afin de pouvoir exprimer des déclarations valides quelque soit l'entité (notamment l'entité accédée) considérée, on définit un nouveau type d'entité appelé `Current` qui prend son sens dans le contexte d'utilisation du mot-clé ; en d'autre terme on dira qu'il se substitue à toute entité accédée à laquelle le mot-clé est attaché. `Current` accepte un type d'entité `E` en paramètre : `E` appartient à l'ensemble des types d'entité qui a été défini. Grâce à cette adjonction il est possible de spécifier que l'on considère par exemple la `Classe` courante (`Current(Classe)`). Ce nouveau type d'entité est soumis aux mêmes contraintes que le type d'entité auquel il est rattaché.

On peut alors définir `public` comme

Current(Classe) ∪ Current(Classe)$_{\text{inclusion,héritage}}$ invocation(C)
Monde invocation(O)

Si on utilise le mécanisme `public` dans la classe `Classe1` sur la méthode `Méthode1`, le remplacement de `Current(Classe)` par `Classe1` nous permet de retrouver la déclaration de l'exemple.

Le type d'entité `Current` permet également de définir des règles implicites, c'est-à-dire applicables quelles que soient les déclarations de protection. Par exemple, en Eiffel, la règle suivante s'applique systématiquement à chaque attribut :

Current(Attribut) Current(Classe)∪ Current(Classe)$_{\text{inclusion}}$
Lecture(A) Écriture(C)

4. Une perspective de Lamp : la définition des mécanismes

La version de Lamp présentée ici permet de déclarer de manière très flexible la protection sur les entités d'un langage à objets. Cependant, de nombreux mécanismes de protection décrivent, en plus de l'ensemble d'entités accédantes, la façon dont ces protections sont transmissibles à des entités élaborées à partir de celles sur lesquelles ils portent. Par exemple, en Java, déclarer une méthode `public` à des conséquences sur les protections de toutes les classes héritières.

Ainsi, si nous souhaitons indiquer des protections qui porteraient sur des entités *une fois qu'elles seront héritées*, il nous faut définir des règles de transformation d'une déclaration, et les associer à des mécanismes.

Les règles de transformation peuvent être :

– générales et implicites, comme par exemple en Eiffel : en l'absence d'indication explicite du programmeur, les entités héritées héritent aussi de la déclaration de protection ou

– spécifiques à un mécanisme qui précise si une entité héritée hérite aussi de la déclaration de protection et, le cas échéant, si elle peut augmenter ou réduire les listes d'accédantes.

Dans le second cas, qui paraît le plus commun, un mécanisme peut être associé à plusieurs règles de transformation, appliquables selon la relation qui est utilisée pour élaborer une nouvelle entité. Par exemple, en Java, le mécanisme `protected` utilisera des règles de transformations différentes pour suivre la relation de redéfinition et celle d'héritage car une propriété redéfinie `protected` ne confère pas les mêmes droits qu'une propriété `protected` héritée sans être redéfinie.

5. Conclusion

Cet article a pour but de poser les premières pierres d'un langage de définition de mécanismes de protection, nommé Lamp, pour les langages de programmation à objets à classes. Une implémentation de Lamp permettrait aux concepteurs de langages autant qu'aux programmeurs de définir avec précision et surtout souplesse et simplicité les protections associées aux différentes entités mises en jeu. Aux vues de la complexité de ces mécanismes dans des langages pourtant aussi répandus que Java (bien peu de programmeurs Java, mêmes expérimentés, maîtrisent les nombreuses subtilités du `protected`), C++ ou Eiffel, il nous semble utile de proposer des améliorations en ce domaine.

Au-delà de la perspective de définition des mécanismes de protection au niveau d'un langage (cf. section 4), d'autres nous semblent mériter de l'intérêt. Une manière d'utiliser Lamp est de lui faire jouer le rôle d'un langage intermédiaire pour les protections. Nous avons montré qu'avec quelques métainformations qui proviennent du langage de programmation il était possible de fournir à Lamp les éléments nécessaires à son fonctionnement. L'ajout d'un sucre syntaxique, par exemple sur la base de mots-clés, permettrait de l'intégrer à différents langages de programmation. Lamp peut donc être utilisé pour uniformiser la définition des mécanismes de protection ainsi que leur mise en œuvre.

Moyennant l'intégration des métainformations dans le modèle des graphes d'accès [ARD 02], qui permet de décrire les effets des mécanismes de protection, l'opérationalisation et l'interfaçage de Lamp à AGATE [ARD 01] en tant que langage intermédiaire fournirait une plate-forme de modélisation et d'analyse de la protection, pouvant notamment traduire une protection d'un langage à un autre.

Au travers de la réification des concepts qu'il implémente, Lamp pourrait aussi être intégré dans le modèle OFL [CRE 01], un modèle métaobjet qui se donne pour

vocation d'offrir des moyens aux programmeurs pour adapter leur langage de programmation favori, voire pour créer un nouveau langage de toutes pièces. Cela permettrait d'ajouter une nouvelle facette à ce modèle qui associe des paramètres (qualifiés d'*hypergénériques*) [CRE 02] aux principaux éléments de la sémantique opérationnelle des langages. Une approche complémentaire pourrait avoir pour objectif de tirer de ce travail un certain nombre de paramètres génériques pour la gestion de la protection, actuellement gérée dans OFL à l'aide d'ensembles d'assertions.

6. Bibliographie

[ARD 01] ARDOUREL G., HUCHARD M., « AGATE, Access Graph bAsed Tools for handling Encapsulation », *ASE'2001 (International Conference Automated Software Engineering)*, 2001, p. 311-314.

[ARD 02] ARDOUREL G., HUCHARD M., « Access Graphs : Another View on Static Access Control for a Better Understanding and Use », *Journal of Object Technology*, vol. 1, n° 5, 2002, p. 95-116.

[BOO 98] BOOCH G., JACOBSON I., RUMBAUGH J., *The Unified Modeling Language User Guide*, The Object Technology Series, Addison-Wesley Publishing Co., octobre 1998.

[COA 91] COAD P., YOURDON E., *Object-Oriented Design*, Prentice-Hall, 1991.

[CRE 01] CRESCENZO P., « OFL : un modèle pour paramétrer la sémantique opérationnelle des langages à objets - Application aux relations inter-classes », Thèse de Doctorat, Université de Nice-Sophia Antipolis, décembre 2001, `http://www.crescenzo.nom.fr/`.

[CRE 02] CRESCENZO P., LAHIRE P., « Customisation of Inheritance », *Springer Verlag, LNCS series, ECOOP'2002 (The Inheritance Workshop) et Proceedings of the Inheritance Workshop at ECOOP 2002 par l'University of Jyväskylä, Finlande*, juin 2002, page 7, également Rapport de Recherche I3S/RR–2002-17–FR (Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis), `http://www.i3s.unice.fr/~lahire/`.

[FLA 99] FLANAGAN D., *Java in a Nutshell : a Desktop Quick Reference*, O'Reilly, 3e édition, décembre 1999.

[GOS 00] GOSLING J., JOY B., STEELE G., BRACHA G., *The Java Language Specification*, The Sun Microsystems Press Java Series, Sun Microsystems, juin 2000, `http://java.sun.com/docs/books/jls/`.

[HAR 92] HARBISON S. P., *Modula-3*, Prentice Hall, 1992.

[LAI 97] LAI M., *UML : la notation de modélisation objet - Applications en Java*, InterEditions (Masson), 1997.

[MEY 94] MEYER B., *Eiffel, le langage*, InterEditions, 1994, `http://www.eiffel.com/doc/documentation.html#etl`.

[Obj 01] OBJECT MANAGEMENT GROUP, « UML : Unified Modeling Language Specification », septembre 2001, Version 1.4, `http://www.omg.org/technology/uml/`.

[SNY 86] SNYDER A., « Encapsulation and Inheritance in Object-Oriented Programming Languages », *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'86*, vol. 21, n° 11, 1986, p. 38–45.

[STR 97] STROUSTRUP B., *The C++ Programming Language*, Addison-Wesley Publishing Co., 3e édition, 1997, `http://www.research.att.com/~bs/3rd.html`.