HAL
open science

Le modèle OFL au service du métaprogrammeur

Adeline Capouillez, Pierre Crescenzo, Philippe Lahire

To cite this version:

Adeline Capouillez, Pierre Crescenzo, Philippe Lahire. Le modèle OFL au service du métaprogrammeur. Conférence sur les langages et modèles à objets 2002 (LMO'2002), Jan 2002, Montpellier, France. pp.11-24. hal-00484442

HAL Id: hal-00484442

https://hal.science/hal-00484442

Submitted on 18 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Le modèle OFL au service du métaprogrammeur

Application à Java

Adeline Capouillez — Pierre Crescenzo — Philippe Lahire

Laboratoire I3S (UNSA/CNRS)
Projet OCL
Les Algorithmes, bâtiment Euclide B
2000, route des lucioles
B.P. 121
F-06903 Sophia Antipolis Cedex
{Adeline.Capouillez,Pierre.Crescenzo,Philippe.Lahire}@unice.fr

RÉSUMÉ. *OFL est le sigle de Open Flexible Languages et le nom d'un métamodèle des langages de programmation à classes. Il repose sur trois concepts essentiels de ces langages : les descriptions qui sont une généralisation de la notion de classe, les relations telles l'héritage ou l'agrégation et les langages eux-mêmes. OFL offre un paramétrage de ces trois concepts dans le but d'adapter leur sémantique opérationnelle aux besoins du programmeur. Ce document résume les principales caractéristiques du modèle OFL, montre comment créer une application à l'aide de ce modèle et décrit le langage Java en OFL.*

ABSTRACT. *OFL is the acronym for Open Flexible Languages and the name of a metamodel for object programming languages based on classes. It relies on three essential concepts of these languages: the descriptions which are a generalisation of the notion of class, the relationships such as inheritance or aggregation and the languages themselves. OFL provides a customisable definition of these three concepts in order to adapt their operational semantics to the programmer's needs. This paper summarises the main characteristics of the OFL model, shows how to create an application using this model and describes the Java language according to OFL.*

MOTS-CLÉS : *métaprogrammation, relation interclasses, paramétrage.*

KEYWORDS: *Metaprogramming, Relationship between Classes, Customisation.*

1. Introduction

L'importance du coût engendré par les besoins de faire évoluer les logiciels est un problème récurrent. Un grand nombre d'approches ont été et sont encore étudiées. On peut citer par exemple *i)* l'ajout de mécanismes d'assertions [COL 96, MEY 97], *ii)* l'introduction de mécanismes pour la génération automatique de documentations ou de tests, *iii)* l'amélioration de l'expressivité des langages afin de rapprocher les phases de conception et de programmation, *iv)* le développement de bibliothèques réutilisables et éprouvées, *v)* l'utilisation de patrons de conception [GAM 99] qui offrent des modèles d'architecture s'appliquant à des problèmes particuliers de programmation, *vi)* l'utilisation de techniques basées sur la rétroconception. On peut citer aussi un ensemble d'approches qui reposent sur la séparation des préoccupations et qui s'intéressent à la programmation par sujets [HAR 93] ou par aspects [KIC 97].

Notre approche souhaite dans ce contexte apporter une contribution notamment à l'amélioration des capacités à évoluer d'une classe, d'une hiérarchie de classes ou d'un composant logiciel. Elle part de l'idée que les relations entre classes dans les langages à objets, et notamment l'héritage, sont des mécanismes de trop bas niveau qu'il serait intéressant de mieux spécifier [MEY 97]. Cette approche se matérialise par la définition du modèle OFL (*Open Flexible Languages*) [CRE 01]. OFL fut tout d'abord pensé sous la forme d'un protocole métaobjet tel celui de CLOS [KEE 89]. Cependant, plus ouvert et plus ambitieux que ce dernier, il est apparu à la fois très difficile et fastidieux à programmer, et plus grave, complexe à utiliser. Nous nous sommes alors orientés vers une approche hypergénérique [DES 94], c'est-à-dire selon nous vers la définition d'un modèle de paramétrage de la sémantique opérationnelle des classes et des relations entre classes. Les principaux éléments de cette approche qui sont décrits dans [CAP 01] sont rappelés dans la section 2.

2. L'approche OFL

L'approche OFL peut, en première lecture, se résumer à la recherche d'un ensemble de paramètres dont la valeur détermine la sémantique opérationnelle d'un langage à classes. Dans [CAP 01] nous avons présenté de manière détaillée la signification de chaque paramètre et nous en avons proposé une classification qui tient compte de leurs objectifs, puis nous avons proposé une micro-application qui montre le bénéfice que peut en tirer le programmeur. Dans le présent article, c'est au niveau du métaprogrammeur que nous nous situons avec une description OFL du langage Java [ARN 00]. Par ailleurs nous préférons montrer la manière dont les différentes entités du modèle sont reliées les unes aux autres, plutôt que de donner une liste exhaustive des valeurs de paramètre. Rappelons, tout de même, deux points essentiels. Le modèle OFL est formé de deux parties principales : un système de paramètres valuables qui permet la création d'éléments de langage et un système d'actions qui sont des algorithmes représentant un *morceau* d'un compilateur, interprète ou exécutif dont l'exécution dépend des valeurs des paramètres.

2.1. *Les objectifs d'OFL*

Nous pouvons ici distinguer d'une part le modèle et d'autre part les outils logiciels que l'on peut construire pour en tirer parti. Ce dernier point sera discuté dans la section 5. Concernant le modèle lui-même nous croyons qu'en spécifiant des relations entre classes dont la sémantique est plus précise qu'un héritage ou qu'une agrégation nous contribuons à l'amélioration de la lisibilité du code. De même, cela permet d'envisager la génération d'une documentation automatique pertinente et la réalisation de contrôles plus appropriés. Cette documentation et ces contrôles devraient en outre permettre de réduire le fossé qui existe entre l'expressivité des méthodes de conception et celle des langages de programmation.

Nous sommes cependant persuadés que, comme une utilisation excessive de l'héritage [LAH 95] et de l'agrégation, un excès de spécialisation de ces mécanismes peut également nuire à la lisibilité. Un compromis est à trouver dans ce domaine. Une solution pourrait être de ne préciser l'usage de l'héritage ou de l'agrégation qu'à certains endroits du programme où l'effet sera particulièrement bénéfique : par exemple l'utilisation de la généralisation (en lieu et place de la spécialisation) pour ajouter une classe au milieu d'une hiérarchie sans modifier le code des classes déjà existantes [CAP 00]. Toujours dans l'idée d'une utilisation réaliste d'OFL, nous avons aussi pour objectif de mettre à la disposition du programmeur des bibliothèques de composants-relations et composants-descriptions — la description est une généralisation de la notion de classe — dans lesquelles il pourra sélectionner ceux qu'il souhaite utiliser. Cette démarche s'apparente à celle visant à fournir des composants logiciels réutilisables voire des composants métiers [BOU 97].

2.2. *OFL et l'état de l'art*

Dans [CAP 01] nous avons cité un certain nombre de travaux connexes avec pour objectif principal de montrer différentes approches visant à mieux décrire les relations entre descriptions. Ici nous préférons répondre à quelques unes des questions que le lecteur pourrait être amené à se poser. Comment notre approche se positionne-t-elle par rapport à des MOP ou des langages réflexifs existants comme CLOS et ClassTalk [COI 89] ? Quelle est la différence avec des extensions comme OpenC++ [CHI 95], OpenJava [TAT 00] ou Javassist [CHI 98] ? MOF [Obj 01a] a été proposé par l'OMG, pourquoi proposer un autre métamodèle ?

Tout d'abord, les langages construits grâce à OFL ne sont pas réflexifs : OFL permet de définir des langages qu'il représente par l'intermédiaire de métainformations. Le travail doit être réalisé en deux phases. Premièrement, le métaprogrammeur intervient en créant des composants de langage et en les assemblant pour créer un langage. Ensuite, dans une seconde phase indépendante de la première, le programmeur peut tirer parti de l'œuvre du métaprogrammeur. Le travail de métaprogrammation est plus difficile et plus sensible que celui de programmation mais plus stable et moins fréquent. Cela nous différencie d'une approche MOP réflexive.

À la différence d'OpenC++, OpenJava et Javassist, nous avons voulu que le modèle OFL ne soit pas attaché à un langage particulier et cela dès sa création. OpenC++, OpenJava et Javassist sont donc moins généraux qu'OFL et ne mettent notamment pas l'accent sur la gestion des relations entre descriptions, les descriptions elles-mêmes tenant la place principale dans ces approches. De plus, notre système de paramètres est piloté par des actions — donc le volume de code à écrire est limité — et contrôlé par des assertions — ce qui offre au métaprogrammeur un cadre sémantique.

Autant par les approches MOP (CLOS et ClassTalk par exemple) que par celle proposées par OpenC++, OpenJava et Javassist, il est possible de réaliser tout ce que fournit OFL. Cependant, OFL, en plus de n'être pas attaché à un langage, dresse un cadre de génie logiciel qui facilite, contrôle et guide la difficile tâche du métaprogrammeur.

La spécificité d'OFL est double : *1)* donner une place prépondérante à la modélisation et à la flexibilité des relations entre descriptions et *2)* offrir un système de paramètres qui facilitent la génération de composants de langage. MOF ou UML [Obj 01b] ne permettent pas directement ou peu de telles choses. Par contre, MOF et UML sont des supports naturels pour décrire OFL. Une description UML est d'ailleurs en cours de réalisation, une partie MOF s'y adjoindra sans aucun doute.

Enfin, citons FLO [DUC 97] qui s'intéresse, comme OFL, aux relations mais qui est un système réflexif et qui se concentre sur les relations interobjets là où notre intérêt va plutôt aux relations interdescriptions.

3. Le modèle OFL

Nous présentons tout d'abord, dans la figure 1, la légende qui s'applique à tous les schémas de ce document. La notation choisie s'apparente à celle d'UML.

La figure 2 illustre l'utilisation du modèle OFL pour décrire une application ; elle représente à la fois une hiérarchie méta représentée par des liens d'instanciation mais aussi une hiérarchie de descriptions liées par spécialisations. Dans l'arbre d'instanciation, chaque niveau n'est relié directement qu'au niveau supérieur, par contre ce n'est pas le cas pour l'arbre de spécialisation dont l'objectif est structurel : modéliser la réification des informations qu'elles soit méta ou pas[1].

Nous montrons dans cette figure les trois niveaux de modélisation nécessaires : *1)* le niveau application regroupe les descriptions et les objets du programme (*OFL-instances* et *OFL-données*), il est donc créé par le programmeur, *2)* le niveau langage décrit les composants du langage de programmation (*OFL-composants*), il est sous la responsabilité du métaprogrammeur et *3)* le niveau OFL représente la réification de ces composants (*OFL-concepts* et *OFL-atomes*), c'est-à-dire notre modélisation de la sémantique opérationnelle des descriptions et relations. L'introduction d'un vocabulaire nouveau a pour objectif de minimiser les quiproquos concernant les termes

1. Nous revenons sur cette distinction au fur et à mesure de l'examen de la figure 2.

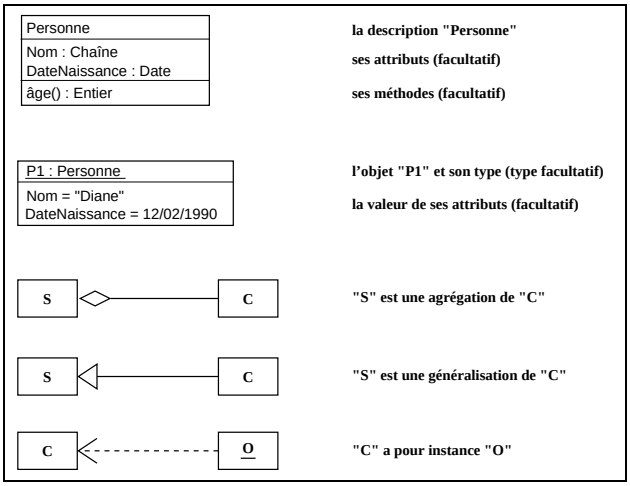

Figure 1. *Légende générale*

employés. Le lecteur pourra trouver par exemple qu'une *description* s'apparente à une classe et qu'un *composant-description* se rapproche de la notion de métaclasse.

3.1. *Le niveau application*

Pour décrire une application, le programmeur utilise les services offerts par le niveau langage. Il crée, au niveau application, des OFL-instances, qui sont les descriptions et les relations de son application, par instanciation des OFL-composants. À l'exécution, les objets de l'application, nommés OFL-données, sont des instances des OFL-instances représentant les descriptions.

Les OFL-instances. Chaque description ou relation décrite par le programmeur est modélisée par une OFL-instance. La figure 2 propose un exemple d'application qui comprend cinq OFL-instances : *i)* trois descriptions : `Véhicule`, `Automobile` et `Couleur`, *ii)* une relation de généralisation : `Automobile` hérite de `Véhicule` et *iii)* une relation d'agrégation : `Automobile` a un attribut de type `Couleur`.

Les OFL-données. Dans l'application, chaque instance de description est réifiée à l'exécution par une OFL-donnée. La figure 2 en présente deux : *i)* `MaFerrari`, instance de la description `Automobile` et *ii)* `Rouge`, instance de la description `Couleur`. Remarquons que les OFL-instances qui représentent des descriptions spécialisent l'OFL-atome `objet`. En effet, `objet` est la réification des données de l'application (OFL-données) et constitue donc la racine de l'arbre de spécialisation des OFL-instances représentant des descriptions. `objet` contient par exemple la collection des attributs d'une instance de description.

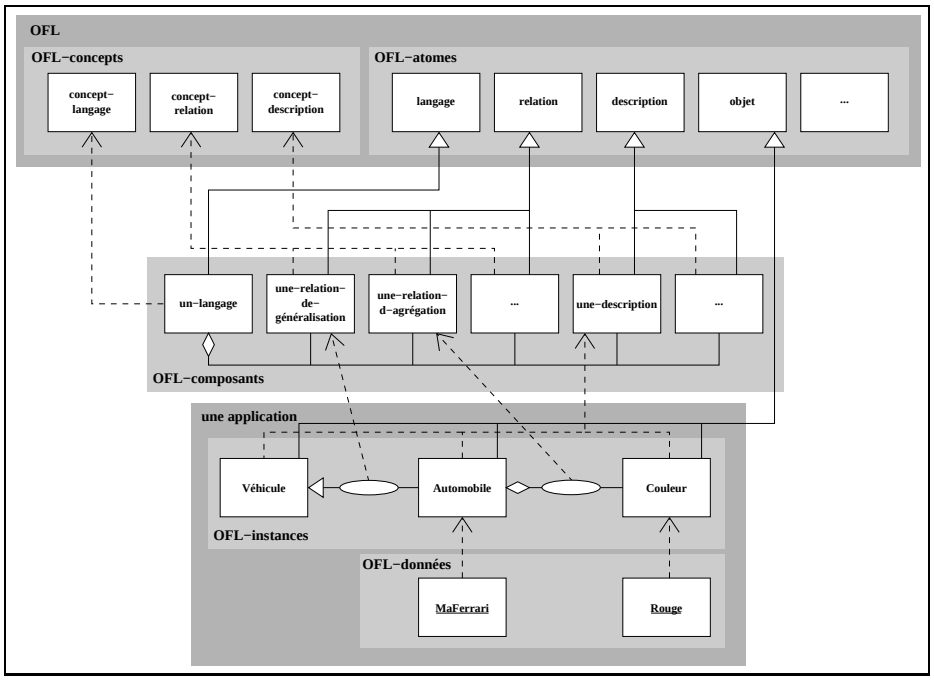

Figure 2. *Architecture d'OFL*

3.2. *Le niveau langage*

Le niveau langage décrit les différentes sortes de relation et de description qu'il est possible d'utiliser dans le langage modélisé. Les relations sont des instances de `concept-relation`, les descriptions des instances de `concept-description`. Le langage lui-même est une instance de `concept-langage`. Il a pour principale fonction de regrouper les relations et descriptions qu'il met à la disposition du programmeur.

Les OFL-composants. La figure 2 recense : *i)* des composants-descriptions dont `une-description`, *ii)* des composants-relations dont `une-relation-de-généralisation` et aussi `une-relation-d-agrégation` et *iii)* un composant-langage `un-langage`. Il est possible de se représenter un composant-description sous la forme d'une métaclasse, un composant-relation comme une métarelation et, de la même manière un composant-langage comme un métalangage. Les entités méta, en plus de l'aspect comportemental qui leur est associé, contiennent un ensemble d'informations fixe. Ces informations sont importées des OFL-atomes au travers d'une spécialisation.

3.3. *Le niveau OFL*

Le niveau OFL constitue un métamodèle pour le langage de programmation (niveau langage) et un métamétamodèle pour les programmes (niveau application). Nous avons choisi de paramétrer trois notions essentielles : les relations, les descriptions et les langages. Cependant, il est nécessaire de réifier bien d'autres composants, tels les objets, les méthodes, les assertions, etc. pour modéliser complètement un langage. Le niveau OFL contient donc deux sortes d'entités : *1)* les OFL-concepts qui décrivent la partie paramétrable (la sémantique opérationnelle) des relations, descriptions et langages et *2)* les OFL-atomes qui décrivent la partie non-paramétrable de ces trois concepts ainsi que tous les autres éléments. Ajoutons enfin que des assertions sont décrites dans chaque OFL-concept et OFL-atome pour garantir la cohérence du modèle.

Les OFL-concepts. La figure 3 montre l'intégralité de la classification des OFL-concepts. La tâche du métaprogrammeur consiste à créer un OFL-composant, instance d'OFL-concept, en donnant une valeur à chacun de ses paramètres. Il définit par ce biais le comportement de chaque future instance de l'OFL-composant. Si les actions prévues n'offrent pas au métaprogrammeur la sémantique opérationnelle qu'il souhaite associer à un OFL-composant, il doit alors modifier le code de ces actions. Cette possibilité laisse le modèle OFL ouvert mais ne doit être utilisée que dans des contextes très spécifiques. En effet, le travail du métaprogrammeur est dans ce cas beaucoup plus lourd que la simple valuation de paramètres.

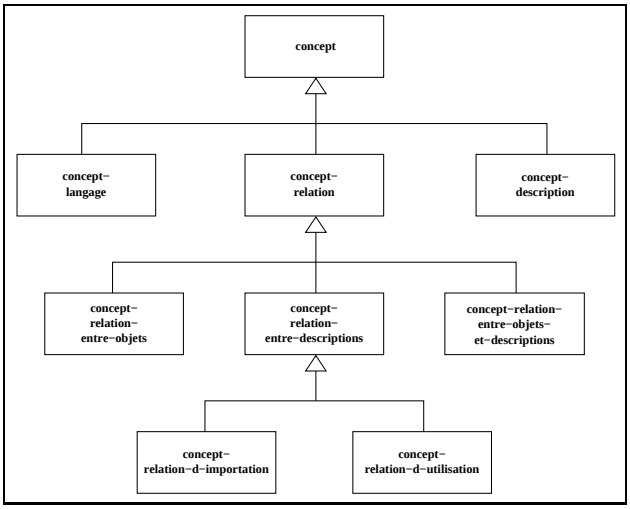

Figure 3. *Les OFL-concepts*

Les concepts-relations. Nous appelons concept-relation l'entité représentant une sorte de relation. Un concept-relation est donc une métamétarelation. Parmi les sortes de relation présentes dans de nombreux langages à classes et méthodes de concep-

tion à objets, nous pouvons citer par exemple l'héritage, l'agrégation, la composition, la généralisation, ... Cependant une méthode ou un langage donné possède rarement toutes ces relations et en utilise certaines pour en simuler d'autres. Par exemple la généralisation en UML décrit aussi bien une généralisation, qu'un héritage, qu'un sous-typage strict[2], ... Une trentaine de paramètres définissent la sémantique de chaque concept-relation du modèle OFL. Vous trouverez dans la partie 4 la liste des composants-relations représentant les relations du langage Java. La figure 3 propose notre classification des concepts-relations. Au sein des relations interdescriptions, nous distinguons les relations d'importation (généralisation du mécanisme d'héritage) de celles d'utilisation (généralisation du mécanisme d'agrégation). La figure 2 montre un exemple d'instance de concept-relation d'importation (`une-relation-de-généralisation`) et un exemple d'instance de concept-relation d'utilisation (`une-relation-d-agrégation`). OFL prend également en compte les relations entre objets et descriptions qui permettent notamment de modéliser le lien d'instanciation qui existe entre un objet et sa description. Nous pouvons aussi modéliser les relations entre objets. Cependant, notre principale préoccupation reste les relations interdescriptions.

Les concepts-descriptions. Un concept-description permet de définir la notion de classe et de tout ce qui ressemble à une classe, comme les interfaces en Java. Un concept-description est donc une sorte de métamétaclasse. Nous pouvons remarquer, par exemple, que les classes d'Eiffel [MEY 97], de C++ [STR 97] ou de Java, même si elles se ressemblent, présentent des différences notables. La figure 2 donne, à titre d'exemple, une seule instance de concept-description appelée `une-description`. Une vingtaine de paramètres sont nécessaires pour décrire le comportement d'une description dans le modèle OFL. Chaque description peut initier ou être la cible d'une ou plusieurs sortes de relation selon la sémantique qu'on veut lui donner ; les sortes de relations qui sont acceptées par (autrement dit, compatibles avec) un composant-description font partie des informations qui lui sont associées. Par exemple en Java, l'instance de concept-description `interface` peut être la cible d'une relation d'`implémentation` mais pas d'`héritage interclasses`.

Les concepts-langages. Le concept-langage est une notion importante et simple. Il modélise, comme son nom l'indique à l'évidence, un langage. Chaque langage est constitué en particulier d'un ensemble de composants-descriptions et d'un ensemble de composants-relations, chacun étant compatible avec au moins un des composants-descriptions sélectionnés. Dans la figure 2 nous avons une seule instance de concept-langage (`un-langage`) qui représente le langage modélisé. Les concepts-langages sont très peu paramétrés et leur principale fonction est de fédérer des composants-relations et des composants-descriptions compatibles entre eux.

Les OFL-atomes. Les OFL-atomes représentent la réification des entités non paramétrées du modèle. Les relations, descriptions et langages possèdent également

2. Ces trois relations ont une sémantique différente même si elles sont suffisamment proches pour être parfois confondues.

leur OFL-atome qui décrit la partie de leur structure et de leur comportement qui n'est pas paramétrée[3]. Sur la figure 2 nous pouvons noter que l'OFL-composant `une-relation-d-agrégation` est une spécialisation de l'OFL-atome `relation` — qui spécifie par exemple la liste des descriptions-sources et la liste des descriptions-cibles. Dans une application, toutes les primitives d'une description sont instances d'un descendant de `primitive`, toutes les expressions sont instances d'`expression` ou d'un de ses descendants et toutes les instances de descriptions sont aussi instances d'un descendant d'`objet`. OFL offre donc une réification complète des entités présentes à l'exécution d'une application.

4. La mise en œuvre d'OFL : application à Java

Nous venons de décrire les différents éléments qui constituent le modèle OFL. Nous proposons maintenant d'utiliser OFL pour modéliser les descriptions et les relations du langage Java. Nous recensons ici les différentes sémantiques de descriptions et de relations entre descriptions du langage Java. Chacune de ces sémantiques est représentée par un OFL-composant. Le lecteur peut se référer à la figure 4 pour avoir la liste complète des OFL-composants de Java.

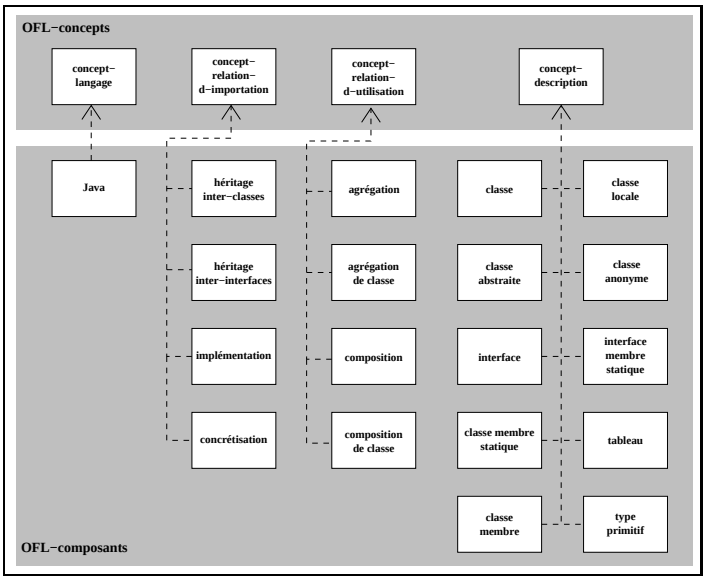

Figure 4. *Les OFL-composants de Java*

Nous avons ainsi dénombré pour Java : *i)* un seul composant-langage (évidemment !), *ii)* huit composants-relations où il est possible d'en retrouver quatre d'impor-

3. Il est donc naturel que les trois niveaux de notre architecture fassent référence directement aux OFL-atomes.

tation et quatre d'utilisation et *iii)* dix composants-descriptions. Le nombre d'OFL-composants peut sembler élevé au programmeur Java. Il est dû à la précision de notre système de paramètres qui offre une granularité relativement fine. Les différences sémantiques entre relations ou descriptions sont souvent masquées au programmeur par l'usage d'un même mot-clé dans un contexte différent. La présentation que nous donnons des OFL-composants de Java ne donne pas la valeur de chacun des paramètres mais plutôt une présentation de leurs principales caractéristiques. Nous signalons entre parenthèses les mots-clés associés à chaque OFL-composants.

Les composants-relations de Java. Les quatre premiers composants-relations sont des importations, les quatre suivants des utilisations.

L'héritage interclasses (`extends`). Cette relation est utilisée pour *affiner l'implémentation de la spécification d'un type de données*. L'implémentation d'une spécification est réalisée dans une classe. L'héritage interclasses spécialise donc une classe. Il s'agit d'un héritage simple pour lequel les cycles sont interdits. Les primitives de la classe héritée sont importées dans la classe héritière. Il est possible de remplacer les attributs et de redéfinir les méthodes. Le polymorphisme s'applique de manière ascendante, c'est-à-dire que toute instance de l'héritière peut être vue comme instance de l'héritée.

L'héritage interinterfaces (`extends`). Cette relation permet d'*affiner la spécification d'un type de données*. Elle est posée entre interfaces, l'héritière spécialisant les héritées. Au contraire de l'héritage interclasses, cette relation est en effet multiple. Le polymorphisme fonctionne de manière similaire à l'héritage interclasses.

La concrétisation (`extends`). Cette relation permet de *concrétiser l'implémentation de la spécification d'un type de données*. Elle est posée entre une classe abstraite (héritée) et une classe non abstraite (héritière). Cette relation est identique à l'héritage interclasses si ce n'est qu'elle impose de donner, dans l'héritière, un corps aux méthodes abstraites de l'héritée et éventuellement à celles de ses super-classes lorsque celles-ci n'en possèdent toujours pas.

L'implémentation (`implements`). Cette relation modélise *l'implémentation de la spécification d'un type de données*. Une classe peut ainsi implémenter une ou plusieurs interfaces. L'implémentation est donc une relation multiple. Si la classe est concrète, elle doit donner un corps à toutes les méthodes spécifiées dans les interfaces. Si elle est abstraite, elle peut donner un corps à certaines méthodes et conserver les autres abstraites. Le polymorphisme est identique à celui des relations d'héritage.

L'agrégation. Cette relation modélise *l'utilisation des services d'une description*. Pour réaliser une telle utilisation, il suffit de déclarer et d'initialiser un attribut du type de la description utilisée[4]. Contrairement aux quatre relations d'importation précédentes, les cycles sont autorisés pour l'agrégation. L'accès aux attributs de la description utilisée est direct, c'est-à-dire que la présence d'accesseurs n'est en rien obliga-

4. L'usage d'un paramètre de méthode, d'un résultat de fonction ou d'une variable locale du type de la description utilisée s'apparente à une agrégation.

toire[5]. La durée de vie de l'objet utilisé est indépendante de celle de l'objet utilisateur et le même objet peut être utilisé par plusieurs objets utilisateurs.

L'agrégation de classe (`static`). Cette relation modélise la notion bien connue dans les langages à objets d'*attribut de classe*. Elle est identique à l'agrégation si ce n'est le fait que l'objet utilisé est associé à la classe utilisatrice et non à ses instances.

La composition. Cette relation modélise *l'utilisation forte des services d'une description*. Dans la littérature on emploie le terme *composition* entre deux classes au lieu d'*agrégation* pour dire qu'une instance d'une des deux classes est incluse dans l'instance qui l'utilise, et donc sa durée de vie est dépendante de celle de l'objet qui la contient/l'utilise. En Java cette notion de composition ne s'applique qu'aux descriptions qui utilisent un type primitif (par exemple : `int`).

La composition de classe (`static`). La composition de classe est similaire à la composition mais défini un *attribut de classe* comme le fait l'agrégation de classe.

Les composants-descriptions de Java.[6] De manière générale, nous nommons *classes internes* les classes membres statiques ainsi que les classes membres, les classes locales, les classes anonymes et les interfaces membres statiques. Nous avons recensé dix composants-descriptions[7].

La classe (`class`). La classe est une *implémentation concrète d'un type de données*. C'est une description non générique qui peut contenir des méthodes[8] et des attributs. Elle est visible au sein de son paquetage mais cette visibilité peut être étendue ou restreinte par un qualifieur (`public` ou `private` par exemple). Elle a la capacité de créer des instances mais pas de les détruire explicitement. Enfin, la classe autorise la surcharge sans prendre en compte le type du retour des fonctions.

La classe abstraite (`abstract class`). La classe abstraite est une *implémentation abstraite d'un type de données*. Cette description possède les mêmes propriétés qu'une classe mais elle peut décrire des méthodes abstraites (sans corps) et ne peut pas posséder d'instance propre.

L'interface (`interface`). Il s'agit de la *spécification d'un type de données*. Au contraire d'une classe, une interface ne peut pas définir d'attribut (sauf les *constantes de classe*). De plus, ses méthodes sont toutes abstraites et elle ne peut donc pas créer d'instance.

La classe membre statique (`static class`). C'est une *implémentation, locale à une classe, d'un type de données*. Sa particularité, par rapport à une classe, est d'être définie à l'intérieur d'une classe et non au plus haut niveau. Elle n'est d'ailleurs accessible qu'au travers de sa classe encapsulante.

---

5. Sauf si la visibilité de l'attribut est restreinte, par exemple en le déclarant `private`.
6. Les *packages* Java ne sont pas des descriptions au sens d'OFL, mais des ensembles de descriptions.
7. Nous ne tenons pas compte des classes internes abstraites dans ce document.
8. Ainsi que des constructeurs, initialiseurs et destructeurs.

La classe membre (`class`). Il s'agit également d'une *implémentation, locale à une classe, d'un type de données*. Mais, à la différence de la classe membre statique, une instance de la classe membre est automatiquement associée à chaque instance de la classe encapsulante.

La classe locale (`class`). Elle représente une *implémentation, locale à une méthode, d'un type de données*. Elle n'est visible qu'à l'intérieur de sa méthode encapsulante. En dehors de cela elle est équivalente aux autres composants-descriptions de classe.

La classe anonyme (`class`). La classe anonyme est une *implémentation, locale à une expression, d'un type de données*. Elle est équivalente à une classe locale mais n'est visible qu'au sein de son expression encapsulante. De plus, n'ayant pas de nom, il n'est pas possible de la référencer et donc d'en hériter. Enfin, de par sa structure syntaxique, si elle implémente une interface, elle ne peut en implémenter qu'une.

L'interface membre statique (`static interface`). Il s'agit d'une *spécification, locale à une classe, d'un type de donnée*. C'est l'équivalent de la classe membre statique sous la forme d'une interface.

Le tableau. Il représente la structure de donnée de même nom, bien connue des informaticiens. C'est donc une *collection indexée et de taille fixe d'entités d'un type défini*. Le tableau est un cas particulier en Java. Chaque tableau est une instance d'une classe virtuelle[9] représentant son type (exemple : un tableau d'entiers est de type `int[]`).

Le type primitif. Le type primitif est la *représentation d'un type de base du langage*. Il permet de décrire les éléments essentiels des applications : booléens, caractères, octets, entiers courts, entiers, entiers longs, flottants et flottants doubles. Remarquons que chaque type primitif décrit une valeur et non un objet mais qu'une classe existe pour représenter chacun d'eux. Par exemple, la classe `Integer` permet de considérer un `int` comme un objet.

Les contraintes entre les OFL-composants de Java. La sémantique des OFL-composants permet de répondre à la question « Quels composants-descriptions sont valides en tant que source ou en tant que cible de quels composants-relations ? ». Pour mieux comprendre le problème, les définitions suivantes sont utiles : *i)* La source d'une relation est la description qui déclare la relation. C'est elle qui requiert le service. Il peut y avoir plusieurs sources, comme dans une association UML, bien que ce cas soit rare. *ii)* La cible d'une relation est la description citée par la source dans la déclaration de la relation. La cible fournit le service. Plusieurs cibles sont possibles, cela étant plus fréquent que pour les sources (exemple : héritage multiple). En ce qui concerne les composants-relations d'importation nous pouvons, par exemple, signaler les contraintes suivantes. *i)* L'héritage interclasses ne peut avoir pour cible ou pour source ni une interface ni une interface membre statique. De plus, les classes anonymes ne peuvent être cible d'un tel héritage. *ii)* L'héritage interinterfaces est

9. Cette classe n'existe pas mais tout se passe comme si elle existait vraiment.

posé entre interfaces (qu'elles soient membres statiques ou pas). *iii)* La concrétisation a forcément lieu entre une classe abstraite et une classe non abstraite. *iv)* Enfin, l'implémentation est une relation qui a toujours comme cible une ou plusieurs interfaces (quelles qu'elles soient) et une classe (quelle qu'elle soit). Le même genre de contrainte s'applique aux composants-relations d'utilisation.

5. Conclusion

Le métamodèle OFL, dont nous avons résumé les caractéristiques dans ce document, décrit les langages à classes. Sa contribution à l'amélioration de l'évolution des composants se fait au travers du paramétrage de la sémantique opérationnelle des descriptions et relations de ces langages.

La concrétisation logicielle d'OFL est constituée de deux parties distinctes d'inégale difficulté. La première s'intéresse à la réalisation d'interfaces graphiques ou syntaxiques pour la métaprogrammation (construction d'OFL-composants) et la programmation (utilisation des OFL-composants dans une application). La seconde, plus complexe, concerne l'implémentation d'un ensemble d'actions répertorié dans [CRE 01] qui prennent en compte la valeur des paramètres. Un large spectre d'utilisation des métainformations est possible. Il s'étend jusqu'à la génération d'un compilateur ouvert qui, par la mise en œuvre de toutes les métainformations, permet la modélisation d'une grande variété de langages.

Actuellement, l'implémentation des deux interfaces citées ci-dessus est largement avancée et un système de stockage des données au format XML [W3C00] est lui aussi en cours de développement. Nous choisissons, dans un premier temps, de restreindre le spectre d'usage de ces métainformations au langage Java. Nous souhaitons encourager le programmeur à utiliser des relations et descriptions non standard seulement là où, dans le programme, le besoin s'en fait sentir et non à appliquer une politique systématique. Suivant cette approche, un outil logiciel peut extraire du programme Java sa représentation OFL. Puis le programmeur améliore son application en précisant ses relations interdescriptions. Cela permettra, entre autres apports, l'ajout de contrôles particulièrement adaptés et la génération d'une documentation plus pertinente. La même démarche peut être suivie pour modifier ou améliorer un graphe de descriptions existant. Cette seconde option se rapproche des travaux sur OpenC++ ou OpenJava dans le sens où elle s'applique à un langage particulier. Elle en reste cependant distincte car elle conserve le système d'actions qui remplace une grande partie de la tâche de codage méta et les assertions qui contrôlent la validité des usages.

Remerciements

Nous tenons à adresser nos plus sincères remerciements à nos relecteurs qui, par leurs remarques précises et constructives, nous ont permis d'améliorer cet article.

6. Bibliographie

[ARN 00] ARNOLD K., GOSLING J., *The Java Programming Language*, The Java Series... from the Source, Sun Microsystems, 3me édition, 2000.

[BOU 97] BOUZEGHOUB M., GARDARIN G., VALDURIEZ P., *Les objets*, Eyrolles, 1997.

[CAP 00] CAPOUILLEZ A., CHIGNOLI R., CRESCENZO P., LAHIRE P., « Gestion des objets persistants grâce aux liens entre classes », *OCM 2000*, mai 2000.

[CAP 01] CAPOUILLEZ A., CHIGNOLI R., CRESCENZO P., LAHIRE P., « Hyper-généricité pour les langages à objets : le modèle OFL », *LMO 2001*, janvier 2001.

[CHI 95] CHIBA S., « A Metaobject Protocol for C++ », *OOPSLA 95*, octobre 1995.

[CHI 98] CHIBA S., « Javassist — A Reflection-based Programming Wizard for Java », *OOPSLA 98 (Workshop on Reflective Programming in C++ and Java)*, octobre 1998.

[COI 89] COINTE P., BRIOT J.-P., « ClassTalk : une transposition des métaclasses d'Objvlisp à Smalltalk-80 », *RFIA'89*, novembre-décembre 1989.

[COL 96] COLLET P., ROUSSEAU R., « Assertions are Objects too ! », *WOON 96*, juin 1996.

[CRE 01] CRESCENZO P., « OFL : un modèle pour paramétrer la sémantique opérationnelle des langages à objets — Application aux relations inter-classes », PhD thesis, Université de Nice-Sophia Antipolis, décembre 2001.

[DES 94] DESFRAY P., *Object Engineering, the Fourth Dimension*, Addison-Wesley Publishing Co., 1994.

[DUC 97] DUCASSE S., « Intégration réflexive des dépendances dans un modèle à classes », PhD thesis, Université de Nice-Sophia Antipolis, janvier 1997.

[GAM 99] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns : Catalogue de modèles de conception réutilisables*, Addison-Wesley Publishing Co., 1999.

[HAR 93] HARRISON W., OSSHER H., « Subject-Oriented Programming (A Critique of Pure Objects) », *OOPSLA 93*, 1993.

[KEE 89] KEENE S., *Object-Oriented Programming in Common Lisp – A Programmer's Guide to CLOS*, Addison-Wesley Publishing Co., 1989.

[KIC 97] KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., VIDEIRA LOPES C., LOINGTIER J.-M., IRWIN J., « Aspect-Oriented Programming », *ECOOP 97*, 1997.

[LAH 95] LAHIRE P., JUGANT J.-M., « Lessons Learned with Eiffel 3 : the K2 Project », *TOOLS 95*, juillet-août 1995.

[MEY 97] MEYER B., *Object-Oriented Software Construction*, Professional Technical Reference, Prentice Hall, 2me édition, 1997.

[Obj 01a] OBJECT MANAGEMENT GROUP, « Meta Object Facility Specification (MOF) », novembre 2001, Version 1.3.1.

[Obj 01b] OBJECT MANAGEMENT GROUP, « OMG Unified Modeling Language Specification (UML) », septembre 2001, Version 1.4.

[STR 97] STROUSTRUP B., *The C++ Programming Language*, Addison-Wesley Publishing Co., 3me édition, 1997.

[TAT 00] TATSUBORI M., CHIBA S., KILLIJIAN M.-O., ITANO K., « OpenJava : A Class-Based Macro System for Java », *Springer Verlag, LNCS series 1826, Reflection and Software Engineering*, 2000.

[W3C00] « Extensible Markup Language (XML) », 2me édition, octobre 2000, Version 1.0, W3C Recommendation.