Un modèle de paramétrage des liens entre classes

Robert Chignoli, Pierre Crescenzo, Philippe Lahire[1]

Résumé : Cet article décrit les principaux aspects d'un modèle de paramétrage des relations entre classes dans les langages à objets. Ce modèle a pour objectif de généraliser, de façon contrôlée, les types de relation entre classes. Les trois notions centrales du modèle sont présentées : lien, classe et langage et les principes du paramétrage sont décrits : chaque concept dispose de paramètres et ceux-ci sont utilisés par les algorithmes (opérateurs) du modèle. Un exemple de mise en œuvre est donné : l'ajout d'un lien de réutilisation de code au langage *Java*.

Mots-clés : programmation objets, liens entre classes, paramétrage

Abstract: This paper describes the main aspects of a model dedicated to the customization of relationships between classes in the object-oriented languages. The aims of this model is to generalize the types of relationship between classes but also to control them. The three main entities of the model are presented: link, class and language, and the main customization aspects are described: each concept is associated to a set of parameters which are used by the algorithms (operators) of the model. An example of link implementation is proposed: the extension of the *Java* language with a link for code reuse.

Keywords: Object-Oriented Programming, Links between Classes, Customization

[1] *Pour les trois auteurs :* Laboratoire I3S (UNSA/CNRS), Projet OCL, Sophia Antipolis, France – *E-Mails* : {Robert.Chignoli | Pierre.Crescenzo | Philippe.Lahire}@unice.fr

1. Introduction

La problématique de notre étude est posée dans le cadre des langages à objets [MAS 89, DUC 98] proposant les notions de *classe* et d'*héritage*, plus particulièrement dans une optique de génie logiciel. Dans ces langages, il est fréquent d'utiliser le mécanisme d'héritage dont on dispose pour mettre en œuvre des liens différents. On n'hésite pas, par exemple, à utiliser l'héritage autant pour réaliser un sous-typage strict que pour effectuer une simple opération de réutilisation de code source [MEY 97]. Cette diversité dans les usages de l'héritage démontre l'intérêt très général de ce mécanisme mais laisse aussi apparaître un défaut immédiat : il est impossible pour un programmeur de spécifier quel usage il souhaite faire de ce mécanisme avec toutes les conséquences que cela peut avoir sur le contrôle, la lisibilité, la documentation, la maintenabilité et l'évolution des programmes. Notre démarche permet donc de tendre vers une amélioration de l'expressivité de l'héritage en mettant à la disposition des programmeurs des paramètres pour ce mécanisme et à proposer des combinaisons de ces paramètres pour les cas les plus courants d'utilisation.

Cette motivation dépasse en fait le cadre de la programmation pour déborder sur celui de la conception. En effet, un décalage existe, et a même parfois tendance à s'accentuer, entre la précision et la richesse sémantique décrites durant les dernières étapes conceptuelles et les possibilités offertes par les langages de programmation en terme de leur expressivité. C'est donc également dans cette démarche de réconciliation et de rapprochement que nous nous inscrivons en conservant toujours à l'esprit que cela ne peut se faire qu'en proposant un système ouvert. Il permet à chacun de spécifier les comportements souhaités tout en restant compatible avec l'existant et les autres extensions.

Dans la suite de cet article, la section 2 présente les trois entités de base du modèle (les liens, classes et langages). La section 3 évoque un aspect essentiel (les paramètres et opérateurs) et aborde les problèmes de niveaux de paramétrage et de composition de liens. Puis, la section 4 décrit un exemple d'ajout d'un lien d'importation (réutilisation de code) dans *Java* [ARN 96, CLA 97, GOL 98]. Enfin, la dernière section conclut et cite les perspectives d'évolution de ce travail.

2. Les entités de base du modèle

Pour atteindre les objectifs décrits dans l'introduction, nous proposons un modèle [CHI 97, CHI 99a, CHI 99b] qui décrit les principaux concepts et sémantiques des langages à objets à classes. Celui-ci repose sur trois notions

fondamentales : les liens (tels l'agrégation[2] et l'héritage), les classes et les langages[3]. Notre ambition est double, d'une part être suffisamment général pour décrire les concepts de classe et de lien de langages à objets à classes industriels (*Java* [ARN 96, CLA 97, GOL 98], *C++* [COP 92, STR 94, CHI 95], *Eiffel* [MEY 94], ...) ; d'autre part, cela permet d'expérimenter de nouveaux concepts (exemple : lien de version de classe) afin de tester leur validité dans le but de faire évoluer les langages de programmation. Les trois entités de base du modèle sont décrites dans cette section. Elles disposent d'une couche méta-programmation et d'une interface de paramétrage présentées dans la section 3.

2.1. *Les liens*

Le premier concept fondamental du modèle est constitué par les concepts-liens. Nous utilisons la terminologie suivante : un concept-lien est une abstraction d'une sorte de lien dans les langages à objets (exemple de concept-lien : l'héritage `implements` de *Java* ; exemple de lien : la relation `implements` entre une interface et une classe *Java*). On trouve ainsi décrit dans le modèle, sous la forme d'un arbre de spécialisation, les liens courants des langages à objets (cf. figure 1 page 4 qui montre quelques-uns de ces liens). Pour préciser encore, chaque concept-lien pourrait être, par exemple dans une implémentation du modèle, représenté par une classe, les instances de cette classe seraient donc des liens. Dans la figure 1, l'arbre de spécialisation pourrait correspondre alors à un arbre d'héritage, les feuilles, adaptées au langage décrit, seraient les seules classes concrètes ; par exemple, la feuille `HÉRITAGE` disparaîtrait, pour *Java*, au profit de `IMPLEMENTS` et `EXTENDS`. Il en va de même pour les concept-classe et concept-langage présentés plus loin. Nous nous sommes particulièrement intéressés aux liens entre classes, et principalement à l'héritage que nous aimerions décliner. Pour chaque lien, nous avons défini les notions de *classe-source* (celle qui déclare le lien) et de *classe-cible*. Par exemple, pour l'héritage, la classe-source est l'héritière, la classe-cible l'héritée ; pour l'agrégation, la classe-source est celle qui déclare un attribut (ou un paramètre de méthode ou ...) et la classe-cible celle qui décrit le type de cet attribut. L'agrégation, accompagnée de la composition[4] sont des liens d'utilisation. L'héritage et la réutilisation de code sont considérés comme des liens d'importation.

[2] Nous nommons *agrégation* (cf. *UML* [LAI 97]) le lien client-fournisseur défini généralement à l'aide d'un attribut. Ce lien est aussi appelé *clientèle* dans la terminologie *Eiffel*.

[3] Le modèle décrit aussi les attributs, méthodes, messages, types de base, instructions, structures de contrôle, ... mais cela dépasse le cadre de cet article.

[4] Le lien de *composition* est à prendre au sens d'*UML*. Il décrit une *utilisation* spécifique dans laquelle la classe-source déclare que ses instances *contiennent* (et non *référencent* comme pour l'agrégation) une instance de la classe-cible.

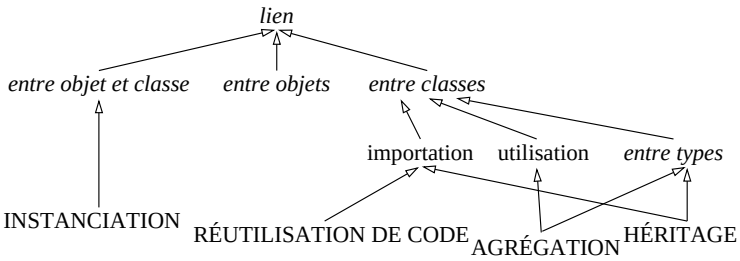

Figure 1. *Des concepts-liens*

Le modèle décrit également les liens entre classes et objets mais notre étude s'est limitée ici à l'instanciation (avec son opposé : l'extension). En revanche, les liens entre objets, bien qu'ils puissent être décrits, ne rentrent pas dans notre champ de recherche.

2.2. *Les classes*

La modélisation de la notion de classe est plus traditionnelle [GOL 89, KIC 91]. Notre objectif est de pouvoir faire coexister plusieurs concepts-classes à la sémantique différente. Il s'agit ici aussi bien de gérer une certaine interopérabilité (imaginons un langage possédant à la fois les concepts-classes d'*Eiffel* et de *C++*) mais aussi de prendre en compte les langages comme *Java* (où nous décrivons les notions d'interface et de classe comme deux concepts-classes).

Chaque concept-classe définit un ensemble de concepts-liens avec lesquels il est compatible et gère leurs interactions. C'est donc à ce niveau que sont traités les éventuels problèmes de composition entre liens issus de concept-lien différent. L'exemple le plus connu de problème de composition de liens est constitué par les célèbres difficultés d'usage de l'héritage multiple et répété. Par exemple : *Eiffel* serait modélisé par un concept-classe et trois concepts-liens représentant l'agrégation, l'expansion et l'héritage (multiple).

Il faut enfin noter que les concepts-classes intègrent la notion de généricité. La notion de type est également modélisée. Chaque classe décrit plusieurs ou un seul type, selon qu'elle est ou non générique. Certains liens, comme l'héritage, agissent en fait entre les types (par l'intermédiaire des classes qui décrivent ces types), alors que d'autres, comme la réutilisation de code ne prennent effet que sur les classes.

2.3. *Les langages*

Le concept-langage est une notion importante et simple. Il décrit, comme son nom l'indique à l'évidence, un langage modélisé. Chacun de ces langages est constitué de deux composants :

1. un ensemble de concepts-classes et
2. un ensemble de concepts-liens, chacun étant compatible avec au moins un des concepts-classes sélectionnés.

L'ensemble des concepts-liens peut, soit être déduit de l'ensemble de concepts-classes, soit être donné explicitement, ce qui permet éventuellement de réduire, à ce niveau, les spécifications de compatibilité décrites dans chaque concept-classe. Par exemple, un langage peut souhaiter utiliser le concept-classe *Eiffel* en éliminant la possibilité d'usage du lien d'expansion. Évidemment, seule une restriction des spécifications peut être effectuée, pas une extension.

À terme, cette modélisation des langages permettra de mettre en œuvre et de contrôler l'interopérabilité entre objets engendrés par des concepts-langages différents.

3. Les paramètres et les opérateurs

Nous venons de décrire les entités de base du modèle. Comme notre but est de permettre un paramétrage de ces concepts de telle façon qu'il soit possible de les adapter à des contextes particuliers ou à des expériences de génie logiciel, il est nécessaire de décrire leur sémantique. Notre approche consiste à développer une sémantique opérationnelle adaptable, pour chaque concept, grâce à un ensemble de paramètres qui pilote l'exécution d'un système d'opérateurs. Chaque concept (concept-lien, concept-classe, concept-langage) possèdent des paramètres. Seuls les concepts-liens et concepts-classes disposent d'opérateurs et d'un protocole pour les méta-programmer. Le travail du méta-programmeur peut prendre deux formes : soit il attribue une valeur aux paramètres[5] des concepts, soit il souhaite modifier la sémantique associée à au moins un de ces paramètres. Dans le premier cas, son travail est terminé et c'est là l'un des intérêts de notre approche. Dans le second cas, il lui faut redéfinir les opérateurs dont la sémantique ne lui convient pas. Nous présentons d'abord le protocole de méta-programmation des opérateurs puis quelques exemples de paramètres et d'opérateurs de classe ou de lien avant de discuter des niveaux de paramétrage et de composition de liens.

[5] Un paramètre est une valeur, exemples : un entier, un booléen, une table, ...

3.1. *La méta-programmation des opérateurs*

La gestion des opérateurs suit un protocole systématique instauré dans le but de faciliter leur usage et leur manipulation. Tout opérateur *OP* (exemples : `lookup`, `assign`) est donc défini par les éléments suivants.

- Une primitive booléenne `IS_OP_VALID` donne `vrai` si *OP* possède une sémantique dans le concept-langage courant, `faux` sinon.
- Une primitive `OP`[6] décrit l'implantation de la sémantique associée à *OP*.
- Pour les opérateurs de lien seulement : un ensemble de primitives `OP_`*L* (où *L* est une lettre) qui définit des étapes de *sous-comportement* pour `OP`.
- Un ensemble de primitives `DEFAULT_OP_`*X* (où *X* est un entier naturel) qui fournit des comportements standards pour *OP*, `DEFAULT_OP_0` décrivant généralement le comportement par défaut de `OP`. Ce système est extensible aux sous-comportements des opérateurs de lien.

3.2. *Les paramètres et opérateurs de lien*

On appelle donc *opérateur de concept-lien*, toute primitive définissant un comportement essentiel au niveau des concepts-liens. Nous avons actuellement dégagé quarante-huit opérateurs de concept-lien paramétrables. Dès maintenant, comme nous l'avons fait dans le titre de cette section, nous allégerons un peu le discours en parlant d'*opérateur de lien* en lieu et place d'*opérateur de concept-lien*. Les sept thèmes des paramètres et opérateurs de liens sont la recherche de primitive, le contrôle sémantique, l'exécution de primitive, la gestion des instances de classes (création, suppression et affectation), la gestion d'extensions de classe, les opérations de base (copie, égalité, ...) et les clauses d'adaptation (renommage, redéfinition, suppression, ... détaillés dans l'exemple, section 4). L'algorithme de ces opérateurs tient compte de la valeur des paramètres de lien pour exécuter une action sémantiquement valide. Par exemple, l'opérateur `lookup`, qui recherche la primitive à exécuter en fonction d'un appel, vérifie entre autres si le lien permet le polymorphisme ou pas et dans quel(s) sens.

Les paramètres de lien fournissent l'information essentielle pour permettre une exécution correcte des opérateurs, en voici quelques-uns :

`cardinality` Il exprime la cardinalité du lien sous la forme $1 - n$ qui signifie que le lien peut être créé entre 1 classe-source et n classes-cibles (n est l'infini ou un entier naturel constant supérieur ou égal à 1). Par exemple,

[6] Cette primitive est une routine (fonction ou procédure) munie ou non de paramètres.

un lien d'héritage simple aura une cardinalité $1-1$ (chaque classe ne peut hériter que d'une seule classe), alors que pour un héritage multiple, elle sera de $1-\infty$ (chaque classe peut hériter de plusieurs classes). On pourra limiter la multiplicité d'un héritage $(1-3)$ ce qui revient à normaliser la programmation au sein d'une équipe de développement ou d'une entreprise.[7]

`circularity` Il exprime la possibilité de créer des cycles avec le lien : `vrai` signifie que des cycles sont permis, `faux` qu'ils sont interdits. Par exemple, l'héritage et l'expansion interdisent les cycles alors que l'agrégation les permet souvent.

`polymorphism` Il indique si le polymorphisme est autorisé par le lien et si oui, dans quel(s) sens il est possible. Prenons plusieurs exemples : l'agrégation interdit le polymorphisme, la spécialisation le permet dans le sens source-vers-cible, la généralisation dans le sens inverse et on peut imaginer un lien de version qui l'autorise dans les deux sens.

`repetition` Il indique si une répétition (directe ou indirecte) de classe est permise dans les classes-sources et dans les classes-cibles.

`symmetry` Il indique si le lien est symétrique.

`opposite` Il décrit, s'il existe, le lien inverse. Il y a pour nous une différence entre symétrie et inverse. Un lien est symétrique s'il fournit lui-même une relation sémantique symétrique. On peut imaginer ainsi un lien *est-une-sorte-de* pour lequel la classe-cible et la classe-source sont les termes d'un tel lien symétrique. Un lien inverse définit, par ailleurs, un *autre* lien décrivant une sémantique inverse. Par exemple : un lien de *spécialisation* est inverse d'un lien de *généralisation*. Ces définitions entraînent le fait suivant : un lien symétrique est son propre inverse.

Les concepts-liens peuvent être décrits dans deux contextes différents : un monde fermé où ils ont connaissance des concepts-classes mis en œuvre (c'est le cas lors de la modélisation d'un langage) ; un monde ouvert où les concepts-classes ne sont pas *a priori* connus (par exemple, lorsqu'on implante une bibliothèque de concepts-liens). Plusieurs paramètres tiennent compte de cette distinction.

3.3. *Les paramètres et opérateurs de classe*

Les paramètres et opérateurs de concept-classe sont le pendant, pour les classes, des paramètres et opérateurs de lien. On appelle *opérateur de classe* (il s'agit en fait d'opérateurs de concept-classe), toute primitive définissant

[7] Nous avons conçu les liens avec une cardinalité $1-n$ mais le modèle permet une généralisation à $m-n$.

un comportement essentiel au niveau des classes. Les opérateurs de classe paramétrables sont actuellement au nombre de quarante-et-un. Les principales situations prises en compte sont naturellement très corrélées avec celles des opérateurs de lien, si ce n'est les clauses d'adaptation qui disparaissent. La capacité d'une classe à créer des instances, à gérer son extension et l'ensemble des concepts-liens compatibles avec elle constituent trois exemples significatifs de paramètres de classe.

3.4. *Les niveaux de paramétrage et la composition de liens*

À propos des niveaux de paramétrage, chaque paramètre de lien peut être redéfini au niveau du concept-classe et du concept-langage pour favoriser la réutilisabilité. Dans ce cas, cette redéfinition ne peut que restreindre, bien évidemment, celle donnée par le lien lui-même. Par exemple : un lien d'héritage multiple peut être utilisé pour décrire un héritage simple (en limitant sa cardinalité), mais pas le contraire. Il ne s'agirait pas ici d'une redéfinition au sens de l'héritage, mais plutôt d'une mise sous contrainte qui respecte la règle suivante. Les paramètres du langage prime sur ceux de la classe qui ont priorité sur ceux du lien. Par exemple : un concept-langage donné utilise un concept-classe donné en réduisant l'héritage multiple accepté par ce dernier à un héritage simple. Il est également possible de retrouver ces paramètres au niveau des liens (qui sont des instances des concepts-liens). En effet, si la cardinalité d'un concept-lien est $1-\infty$, celle d'une de ses instances peut être $1-3$. Cette information supplémentaire permet de faciliter l'interopérabilité (exemple : un lien d'héritage $1-1$ issu d'un concept-lien multiple est potentiellement compatible avec un concept-lien d'héritage simple) et de réaliser des contrôles d'intégrité et de cohérence.

Le problème de la composition de liens se pose lorsqu'un concept-classe permet l'utilisation de plusieurs concepts-liens. Nous présentons ci-après quatre exemples qui peuvent générer des conflits et quelques éléments de solutions. La méthode de résolution de tels conflits est toujours identique : les problèmes entre concepts-liens sont résolus par les opérateurs de classe alors que les problèmes internes à un concept-lien le sont par les opérateurs de lien.

1. **Héritage et agrégation** : Par exemple, dans le langage *Eiffel*, il est possible de masquer des primitives dans les descendantes d'une classe qui les déclare exportées. C'est un exemple démontrant que des possibilités d'interaction existent même entre liens d'utilisation (l'agrégation) et d'importation (l'héritage). La résolution de cette influence est réalisée, au travers du lien d'importation par l'usage d'un opérateur de lien permettant de cacher une primitive (`hide`).

2. **Héritage multiple** : Notre modèle permet deux types de modélisation possible d'un héritage multiple. Le premier et le plus simple est l'usage direct d'un lien d'*héritage multiple* (opérateur de lien `cardinality` à $1-\infty$) qui se charge en interne de résoudre les problèmes de conflits éventuels. La seconde solution consiste à utiliser plusieurs liens d'*héritage simple* (`cardinality` à $1-1$) et à résoudre les conflits au niveau de l'opérateur de classe `lookup` grâce aux opérateurs de lien (clauses d'adaptation). Nous préconisons l'usage exclusif de la première solution car elle est plus claire du point de vue modélisation et plus simple à mettre en œuvre.

3. **Héritage et sous-typage** : Une classe est déclarée comme héritière (au sens courant de l'héritage) et sous-type (c'est un lien plus restrictif que l'héritage) d'une même autre classe[8]. Ce sont les opérateurs de classe qui doivent lever les ambiguïtés. Le sous-typage étant plus restrictif que l'héritage, c'est la sémantique du sous-typage qui doit être pris en compte dans ces opérateurs.

4. **Spécialisation et généralisation** : Ces deux liens sont *a priori* incompatibles : c'est-à-dire qu'ils ne devraient pas lier dans le même sens, deux classes : ils peuvent d'ailleurs être considérés comme inverses.

4. Un exemple : un lien de *réutilisation de code*

Nous avons choisi de présenter des éléments de solution[9] au problème suivant : nous souhaitons expérimenter l'intégration d'un lien de *réutilisation de code* (par exemple matérialisé par le mot-clé `reuse`) dans le langage *Java*. Il s'agit d'un lien d'importation qui viendra s'ajouter à ceux d'*héritage d'interface* (`implements`) et d'*héritage de classe* (`extends`). Son but est de permettre la récupération de code source. L'intérêt d'un tel lien apparaît clairement dans le cadre du génie logiciel. Il s'agit de pouvoir mieux exprimer les intentions du programmeur dans un souci de contrôle, de lisibilité, de maintenabilité et de documentation afin d'améliorer la qualité du logiciel produit.

Sans `reuse`, la réutilisation de code peut déjà se faire, soit par un héritage, soit par une agrégation, et nous ne remettons pas cela en cause. Nous souhaitons seulement démontrer qu'un lien *ad hoc* serait préférable. Imaginons une classe `A` possédant, entre autres, une primitive `P` que l'on souhaite réutiliser dans une classe `B`.

[8] Ne discutons pas ici de ce choix de programmation *a priori* étonnant.
[9] La solution complète est modélisable.

4.1. *La réutilisation de code par agrégation*

On crée un attribut `Handle` de type `A` dans `B`. On peut citer quelques inconvénients pour cette méthode :

- `Handle` n'a aucun sens du point de vue de la modélisation.
- L'existence de `Handle` crée une indirection systématique pour réutiliser le code de `P`, il faudra toujours faire un appel de la forme `o.Handle.P()`.
- Il faudra soit que toute classe cliente de `B` fasse un appel explicite à `Handle` (implémentation non cachée au client), soit que `B` encapsule l'appel de `Handle.P()` dans une nouvelle primitive (code supplémentaire).
- Il faut gérer `Handle` : création, initialisation, vie, destruction, récupération mémoire, ...

4.2. *La réutilisation de code par héritage*

`B` hérite de `A`. Quelques inconvénients de cette méthode sont les suivants :

- Si `P` est privée (`private`), on ne peut réutiliser son code car les primitives privées ne sont pas héritées[10].
- Le polymorphisme est possible[11] entre `A` et `B` ce qui n'a pas de sens.
- `B` est un sous-type de `A`, ce qui est faux.
- Toutes les instances de `B` (et de ses sous-types) sont également des instances de `A`, ce qui rend la gestion des extensions de classes erronée.
- L'héritage de classe étant simple, `B` ne peut plus hériter.

4.3. *La réutilisation de code par un lien ad hoc*

Le lien que nous décrivons se propose de réaliser cette réutilisation de code sans les inconvénients que nous venons de citer. Aucune composition avec les liens d'utilisation, comme l'agrégation, n'est utile. La composition avec les autres liens d'importation est simpliste car la réutilisation de code n'a aucune influence sur les types et n'est incompatible avec rien. Nous montrons cela en spécifiant quelques exemples de paramètres et d'opérateurs de lien (nous présentons seulement les plus significatifs) que le méta-programmeur devra définir pour réaliser un tel lien. Outre l'intérêt d'un lien spécifique pour la lisibilité,

[10] Décider s'il est légitime de souhaiter réutiliser une primitive privée n'est pas le débat ici.

[11] Dire qu'il est possible, c'est dire qu'il n'est pas contrôlable, sauf avec un surcoût prohibitif (par exemple : du code spécifique nécessitant parfois l'accès à un niveau méta [interprète, compilateur, ...]).

la maintenabilité et le contrôle, on lui trouve d'importants avantages lorsque les classes liées évoluent (pour une nouvelle version d'application ou de bibliothèque, par exemple). En effet, une meilleure spécification permet aux environnements de programmation de mieux adapter leur comportement dans le but d'assister le programmeur.

Des paramètres

`cardinality` La cardinalité de ce lien est $1 - \infty$.

`circularity` Rien n'empêche ce lien d'être circulaire.

`polymorphism` Aucun polymorphisme n'est possible avec ce lien, c'est d'ailleurs une de ses particularités, en tant que lien d'importation.

`repetition` Elle ne pose aucun problème.

`symmetry` Le lien n'est pas symétrique.

`opposite` Dans ce contexte, aucun lien inverse n'est présent.

`rename` Le renommage d'une primitive est autorisé.

`redefine` La redéfinition (de signature, d'assertions ou de corps) est interdite (originalité par rapport à l'héritage).

`remove` L'annulation ne pose aucun problème, grâce à l'absence de polymorphisme (originalité par rapport à l'héritage).

`abstract` Elle consiste à rendre abstraite une routine concrète et elle est interdite : rendre abstraite une primitive (donc *perdre* son corps) dans un lien de réutilisation de code ne présente aucun intérêt (originalité par rapport à l'héritage).

`hide` Le masquage (il s'agit de rendre *privée* et donc de ne plus exporter une primitive au travers de l'interface de la classe — de ses liens d'importation) est permis sans contrainte.

...

Les paramètres ci-dessus dirigent l'exécution des opérateurs suivants pour le concept-lien que nous définissons. Le méta-programmeur peut cependant redéfinir ces opérateurs pour les faire évoluer ou modifier leur sémantique ou pour prendre en compte des situations non résolues par le modèle.

Des opérateurs

`match` Il vérifie la validité d'une primitive par rapport à une recherche sur son nom, les nombre et type de ses paramètres, la présence ou l'absence de valeur de retour, ...

`lookup` Il effectue la recherche de primitive (il peut pour cela utiliser `match`). En l'absence de clauses de redéfinition, d'abstraction et de sélection, il est bien plus simple que celui de l'héritage. Il utilise évidemment la valeur de `polymorphism`.

`type_conformance` La compatibilité de type n'est vérifiée que si les deux types sont strictement identiques (encore un effet de l'absence de polymorphisme).

`assign` Il vérifie la validité d'une affectation puis effectue l'attachement d'objet en plusieurs étapes : évaluation de la partie droite de l'affectation (source), détermination de la partie gauche de l'affectation (cible), contrôle de compatibilité des types (très simplifié en l'absence de polymorphisme), attachement de la source au champ correspondant de la cible (objet ou routine en cas de variable locale), éventuellement prise en compte de la persistance, ...

...

4.4. *Le bilan*

Nous avons montré, dans cet exemple, la faisabilité d'une telle approche. Le nouveau concept-lien défini permet de réaliser, en *Java* et sans les inconvénients de l'agrégation ou de l'héritage, une réutilisation de code source. La tâche du méta-programmeur consiste simplement à créer un concept-lien, à donner une valeur à ses paramètres, puis à intégrer ce nouveau concept-lien au concept-classe `classe_java`. Par défaut, les opérateurs de classe tiennent compte des paramètres des liens, l'intégration d'un nouveau concept-lien est donc facilitée.

5. Conclusion

Dans cet article, nous avons présenté, dans ses grandes lignes, un modèle de paramétrage des liens entre classes en décrivant les concepts de lien, classe et langage. Le système de paramètres et d'opérateurs, au cœur du modèle, a ensuite été abordé pour enfin laisser la place à un exemple d'utilisation qui montre, selon nous, l'intérêt d'un mécanisme permettant de mieux spécifier les liens entre classes dans les langages à objets à classes.

Nous nous intéressons désormais à la spécification d'un langage de définition des paramètres et d'une extension de *Java* permettant d'utiliser, dans un programme, de nouveaux types de lien (nous commencerons par ajouter le lien de réutilisation de code décrit ci-dessus). Cela conduira au développement d'un système de programmation *ad hoc* offrant ainsi une plate-forme de tests pour la modification et l'ajout de lien entre classes. Enfin, nous étudions un service de persistance (*ROOPS* [CAP 99]) reposant sur ce modèle et utilisant à son profit l'information des liens pour mieux gérer la persistance. Ce service nous

amènera, si nécessaire, à compléter l'ensemble des paramètres que nous avons défini pour prendre en compte ces problèmes de persistance.

Références

[ARN 96] K. Arnold and J. Gosling. The *Java* Programming Language. Sun Microsystems, The Java Series, 1996.

[CAP 99] A. Capouillez. *ROOPS* : un service de persistance paramétrable pour *OFL*. Rapport de stage de DEA, Laboratoire *Informatique, Signaux et Systèmes de Sophia Antipolis*, juillet 1999.

[CHI 95] S. Chiba. A Metaobject Protocol for *C++*. *OOPSLA'95 ACM Conference* volume 30, 1995.

[CHI 97] R. Chignoli, P. Crescenzo et P. Lahire. Liens entre classes dans les langages à objets. Rapport de Recherche 97-22, Laboratoire *Informatique, Signaux et Systèmes de Sophia Antipolis*, juillet 1997.

[CHI 99a] R. Chignoli, P. Crescenzo and P. Lahire. *OFL* : An Open Object Model based on Class and Link Semantics Customization. Research Report 99-08, Laboratory *Informatique, Signaux et Systèmes de Sophia Antipolis*, March 1999.

[CHI 99b] R. Chignoli, P. Crescenzo et P. Lahire. *OFL* : une machine virtuelle objet flexible pour la gestion d'objets persistants et mobiles. Rapport de Recherche 99-09, Laboratoire *Informatique, Signaux et Systèmes de Sophia Antipolis*, mars 1999.

[CLA 97] G. Clavel, N. Mirouze, E. Pichon et M. Soukal. *Java* : la synthèse. Masson, 1997.

[COP 92] J. O. Coplien. Programmation avancée en *C++* : styles et idiomes. Addison-Wesley Publishing Co., 1992.

[DUC 98] R. Ducournau, J. Euzenat, G. Masini et A. Napoli. Langages et modèles à objets : état des recherches et perspectives. INRIA Collection didactique, juillet 1997.

[GOL 89] A. Goldberg and D. Robson. *Smalltalk-80* : the Language. Addison-Wesley, 1989.

[GOL 98] M. Golm and J. Kleinöder. *metaXa* and the Future of Reflexion. UTCCP Report 98-4, 1998.

[KIC 91] G. Kiczales, J. des Rivières, and D.Bobrow The Art of the Metaobject Protocol. MIT Press, 1991.

[LAI 97] M. Lai *UML* : La notation de modélisation objet : Applications en *Java*. Masson, juillet 1997.

[MAS 89] G. Masini, A. Napoli, D. Colnet, D. Léonard et K. Tombre Les langages à objets : langages de classes, langages de frames, langages d'acteurs. MIT Press, 1991.

[MEY 94] B. Meyer. *Eiffel*, le langage. InterEdition, 1994.

[MEY 97] B. Meyer. Object-Oriented Software Construction (second edition). Prentice Hall, 1997.

[STR 94] B. Stroustrup. The Design and Evolution of *C++*. Addison-Wesley Publishing Co., 1994.