Liens entre classes dans les langages à objets

Robert Chignoli, Pierre Crescenzo et Philippe Lahire

Rapport de recherche N° 97-22
Juillet 1997

Liens entre classes dans les langages à objets

Robert Chignoli, Pierre Crescenzo et Philippe Lahire

chignoli@unice.fr, Pierre.Crescenzo@i3s.unice.fr, lahire@unice.fr

Laboratoire I3S – CNRS : UPRES-A 6070 & Université de Nice-Sophia Antipolis (UNSA)
Les Algorithmes / Bâtiment Euclide – 2000, route des Lucioles - Sophia-Antipolis -
F-06410 Biot - France

Tél. : +33 (0)4 92 94 27 01 - Fax +33 (0)4 92 94 28 98

Mots-clés :

Technologie objet, langages à classes, liens inter-classes, héritage, méta-programmation, Eiffel

Résumé :

Ce rapport de recherche présente l'état d'avancement du projet *OFL* en insistant tout particulièrement sur les motivations de cette étude.

Le premier objectif d'*OFL* (pour *Object Flexible Language*) est de contribuer à l'étude des fondements de la technologie objet – plus particulièrement, dans les langages à classes – en vue de mieux prendre en compte les besoins des concepteurs et développeurs de logiciels (maîtrise des mécanismes d'héritage et de clientèle, mutations de type, inter-opérabilité, gestion de versions, persistance, etc.). Notre approche conduit à proposer des mécanismes de paramétrisation des liens entre classes via un modèle *OFL/M*.

Le deuxième objectif d'*OFL*, non développé dans ce rapport, est de proposer à terme un prototype (*OFL/VM* pour *OFL/Virtual Machine*) en Eiffel qui permettra de mieux satisfaire les besoins exprimés ci-dessus en offrant la possibilité (par méta-programmation) de paramétrer et de généraliser les liens entre classes.

Table des matières

1 Introduction **3**

2 Usages et limites des liens d'héritage et de clientèle dans les LC **4**
2.1 Choisir entre clientèle et héritage 4
2.2 Un panorama des usages des liens d'héritage dans les LC 6
2.3 Une limite : versions de classes et mutations d'objets 8
2.4 Bilan : paramétrer et généraliser les mécanismes objet dans les LC 9

3 Quelques LC supportant le paramétrage ou la généralisation de liens **11**
3.1 Langages ouverts et Génie logiciel 11
3.2 CLOS/MOP 12
3.3 *NeoClassTalk* 13
3.4 *Iguana* 13
3.5 *OpenC++* 14
3.6 CodA 15
3.7 *VBOOL* 16

4 Paramétrage pour des liens d'héritage et de clientèle **16**
4.1 Pourquoi paramétrer ? 16
4.1.1 Augmenter l'expressivité 18
4.1.2 Clarifier les usages et promouvoir le génie logiciel 20
4.1.3 Conserver ses habitudes 21
4.2 Les axes de paramètrage de l'héritage 21
4.2.1 Lien entre classes ou entre types 21
4.2.2 Extension 22
4.2.3 Assertions 22
4.2.4 Généricité 22
4.2.5 Polymorphisme 23
4.2.6 Migration 23
4.2.7 Clauses d'adaptation 23
4.2.8 Opérateurs de base 24
4.2.9 Influences inter-liens 24

5 Conclusion **25**

1 Introduction

L'émergence de la technologie objet dans le domaine du génie logiciel [Mey97] a marqué, marque et marquera probablement profondément de son empreinte les domaines de la conception du logiciel, de la réutilisation de composants et du rétro-développement.

Cette technologie permet en effet d'envisager une automatisation plus poussée et moins coûteuse des activités liées à l'évolution des applications et par là, pourrait conduire à une meilleure maîtrise des processus d'informatisation.

En particulier, l'arrivée de techniques de persistance "objet" sophistiquées[1] ou le développement d'environnements distibués ou parallèles vont permettre d'élargir le champ d'application de cette nouvelle technologie qui devrait à terme se généraliser dans l'industrie du logiciel.

Nous nous intéressons ici spécifiquement aux langages à classes (notés LC dans la suite), c'est-à-dire à la grande famille des langages objets qui, de Smalltalk à Java en passant par C++ et Eiffel procurent un typage fort aux entités manipulées.

Bien que déjà largement utilisés, les LC sont toujours l'objet de débats passionnés qui divisent la communauté (comment présenter *formellement* les concepts centraux d'héritage et d'agrégation, comment *bien se servir* de la technologie, etc.)... Et ces débats de base sont sans cesse relancés par la prise en compte des nouveaux besoins (intégration de la persistance, contrôle fin de la réutilisation, rétro-développement, gestion de versions d'objets et/ou de types et/ou de classes, parallélisme et distribution).

Le projet *OFL*, dont nous présentons ici les motivations et les grandes lignes, se situe directement dans ce champ d'investigations. Il prend pour base le langage Eiffel, à la fois en tant que langage de développement du projet, mais aussi et surtout en tant que base conceptuelle de l'environnement que nous proposons.

L'environnement *OFL* repose sur trois postulats :

- Les relations d'héritage et d'agrégation (nous parlerons de *clientèle* dans la suite) ne sont que deux types de liens possibles entre classes. L'environnement doit être suffisamment ouvert pour permettre de paramétrer (à la demande, en fonction des besoins) ces mécanismes et même d'ajouter de nouveaux types de liens.

- L'environnement doit avoir des qualités de *réflexion* pour permettre deux niveaux de programmation : l'un classique pour le programmeur d'applications et l'autre pour le méta-programmeur de liens.

- Les types (et les classes) doivent être des objets de première classe pour favoriser un couplage fort de tout objet d'exécution (persistant ou non) avec son type.

Après la présente introduction, la section 2 présente et discute les principaux usages – actuels ou potentiels – des mécanismes objet, en insistant plus particulièrement sur l'héritage. La section 3 décrit quelques langages représentatifs des différentes approches pour mieux maîtriser les usages de la technologie objet. La section 4 développe notre approche pour un paramétrage/généralisation des mécanismes objet pour Eiffel.

1. Via des extensions à des langages existants où par des SGBDOO.

2 Usages et limites des liens d'héritage et de clientèle dans les LC

Cette section présente un panorama des liens dans les LC en insistant sur trois problèmes : un premier paragraphe discute du choix entre héritage et clientèle pour modéliser un concept ; un second paragraphe présente plus généralement les usages (actuels et potentiels) des principes objets dans les LC ; et le paragraphe suivant évoque les limitations de la technologie objet pour prendre en compte les problèmes de mutation de types.

2.1 Choisir entre clientèle et héritage

Il s'agit là certainement du premier problème que se pose tout concepteur lorsqu'il entame la phase qui cherche à définir l'architecture du modèle de son application.
Pour mener à bien cette tâche, le concepteur doit connaitre et maîtriser les critères de choix entre les deux mécanismes, c'est-à-dire connaitre et appliquer certaines règles. Bertrand Meyer a proposé dans [Mey96b] plusieurs règles qui doivent aider le concepteur à faire son *bon* choix. Nous les reprenons ci-dessous en les commentant.

Règle fondamentale de l'héritage

> La **règle de l'héritage** (ou **règle « est-un »**) est : une classe `B` ne peut hériter d'une classe `A` que si, d'une manière ou d'une autre, il est possible de considérer toute instance de `B` aussi comme une instance de `A`.
> Le contre-exemple donné par Bertrand Meyer est celui de la classe `AUTOMOBILISTE` qui hérite de `PERSONNE` (ce qui est bon) et de `AUTOMOBILE` (ce qui est une aberration : un automobiliste ne peut être une automobile). La règle, qui est nécessaire mais non suffisante, est relativement lâche : en effet il suffit qu'on puisse raisonnablement dire qu'une instance de `B` « est-une » instance de `A` sans que cela soit nécessairement prouvé ou universel. Il faut seulement que ce soit défendable.

Claire *a priori*, cette définition devient problématique si on l'érige en condition indispensable... D'une part, savoir si un choix est défendable ou pas nécessite une expertise importante et, d'autre part cette définition semble exclure certains besoins d'héritage (pour réutiliser du code, par exemple).

Règle du changement

> La **règle du changement** est : il ne faut pas utiliser l'héritage pour décrire une relation « est-un » si les objets liés peuvent être changés lors de l'exécution. L'héritage ne pourra donc être utilisé que si la relation est permanente, sinon il faudra choisir la clientèle.

L'exemple donné par Bertrand Meyer est : si on décide qu'un ingénieur en logiciel est toujours un ingénieur, on peut dire que la classe `INGÉNIEUR_LOGICIEL` hérite de la classe `INGÉNIEUR`. Mais si l'on veut pouvoir modifier ce fait (par exemple, l'ingénieur en logiciel se considère désormais comme un savant), il faudra faire de `INGÉNIEUR_LOGICIEL` une cliente de `VOCATION` (par exemple, `VOCATION` pourrait être héritée par `INGÉNIEUR` et `SAVANT`).

Un cas intéressant est celui où l'on utilise l'héritage en souhaitant tout de même pouvoir modifier la partie ingénieur de l'ingénieur en logiciel. Il suffit pour cela de prévoir une méthode qui modifie tous les attributs liés à l'état d'ingénieur.

Cette deuxième règle, tout en apportant un critère clair pour choisir, place néammoins le concepteur dans une situation difficile. En effet, si on relie l'exemple donné à la première règle, l'héritage de `VOCATION` par `INGÉNIEUR` revient à dire qu'être ingénieur « est-une » vocation ! Prouvé ? Universel ? Défendable ?

Règle du polymorphisme

La **règle du polymorphisme** est : l'héritage est approprié pour décrire une relation « est-un » si les entités du type le plus général peuvent être attachées à des objets du type le plus spécialisé.

Cette règle dit simplement que l'héritage est adapté aux cas où le polymorphisme est utile. Mais comme nous le verrons plus loin, il existe plusieurs situations d'héritage communément admises qui sortent de ce cadre (héritage pour rendre abstrait, héritage de constantes, etc.).

Règle de choix entre clientèle et héritage

Pour exprimer la dépendance d'une classe `B` par rapport à une classe `A` :

1. Si toute instance de `B` possède un composant de type `A` mais que ce composant peut être remplacé durant l'exécution par un objet de type différent, alors faire de `B` une cliente de `A`.

2. Si les entités de type `A` doit pouvoir décrire un objet de type `B` ou si des conteneurs d'objets de type `A` peuvent aussi contenir des objets de type `B`, alors faire de `B` une héritière de `A`.

Compilation des deux précédentes règles, cette règle de choix entre héritage et clientèle réduit en définitive le recours à l'héritage au cas du polymorphisme, ce qui est en pratique assez éloigné des multiples usages (pas toujours abusifs) faits par les concepteurs « objet ».

C'est d'ailleurs l'objet de la section suivante que de présenter brièvement les différents usages du mécanisme d'héritage.

2.2 Un panorama des usages des liens d'héritage dans les LC

Cette section a un double objectif :

- montrer la complexité d'une mise en œuvre *bien controlée* de la technologie objet. On s'appuie pour cela sur la récente proposition de classification des usages potentiels de l'héritage faite par B. Meyer qui, en s'éloignant un peu des règles décrites précédemment, décrit l'ensemble des usages réels de l'héritage.
- montrer l'intérêt de l'introduction de moyens de contrôle de la mise en œuvre de l'héritage dans un langage donné. On discute pour cela du mécanisme d'héritage proposé par le langage Java.

Les usages de l'héritage selon B. Meyer

Les usages valides de l'héritage sont :

- héritage de modélisation :
  - héritage de sous-typage
  - héritage de vue
  - héritage de restriction
  - héritage d'extension
- héritage de programmation :
  - héritage de réification
  - héritage de structure
  - héritage d'implantation
  - héritage de facilité :
    - héritage de constantes
    - héritage de machine
- héritage de variation :
  - héritage de variation fonctionnelle
  - héritage de variation de type
  - héritage pour rendre abstrait

L'héritage en Java

Dans le paragraphe précédent, nous avons donné une liste d'usages potentiels du mécanisme d'héritage, liste dans laquelle un concepteur peut (ou devrait) "piocher" pour définir la structure de son application. Et cette application pourrait être alors écrite, par exemple, en *C++* et *Eiffel*, deux langages qui offrent l'héritage multiple. Dans ce contexte, le mécanisme d'héritage est livré "brut" et son bon usage repose exclusivement sur la méthodologie de conception. Concrètement, le mot-clé `inherit` d'*Eiffel* sert à implémenter tous les usages décrit précédemment.

Mais une autre approche est possible, où le langage pourrait directement offrir un cadre méthodologique au concepteur, en mettant à disposition des clauses spécifiques pour tout ou partie des usages.

Pratiquement, cela reviendrait à éliminer le mot-clé multi-usages `inherit` et à introduire une série de mots-clés à la sémantique plus précise.

Par exemple :

- `extends`, pour compléter une classe existante
- `modifies`, pour redéfinir en partie une classe existante
- `specializes`, pour affiner (spécialiser) une classe existante
- `implements`, pour concrétiser une classe existante
- `re-uses`, pour simplement réutiliser le code d'une classe existante

Ce qui, bien-sûr fait apparaître d'importants problèmes de sémantique. On peut en particulier noter les suivants :

- Quelle sémantique précise donne-t-on à chaque mot-clé ? Par exemple, `implements` s'applique-t-il uniquement pour finaliser une classe abstraite ou permet-t-il de conserver dans la nouvelle classe des parties abstraites ?
- Comment régler les cas qui relèvent de plusieurs usages ? avec des règles de précédence ? avec des règles d'interdiction mutuelle ?
- Doit-on relacher la substitution polymorphique dans le cas d'une simple réutilisation de code par `re-uses` ?
- Comment traiter l'héritage multiple ?

Bien que largement incomplète, la série de questions ci-dessus est déja en soi une explication de l'absence (à notre connaissance) d'un langage reposant sur une classification précise de l'héritage ! Règlementer l'héritage oblige à fixer le fonctionnement du mécanisme à plusieurs niveaux (des problèmes de modélisation du monde réel aux problèmes de récupération de code existant), et il n'est même pas sûr qu'une telle généralisation reste praticable !

Si la généralisation de la règlementation de l'héritage dans un langage donné reste à l'heure actuelle un sujet de recherche, l'introduction partielle de cette règlementation est en revanche une réalité et *Java* en est un exemple particulièrement réussi (ou du moins visiblement apprécié).

Java propose deux sortes d'héritage distincte autour de deux mots-clés qui affine l'usage de l'héritage. L'héritage au sens classique se décrit par le mot-clé `extends`, et le mot-clé `implements` a été introduit spécialement pour faire de l'héritage d'implémentation à partir de classes entièrement abstraites (`interface`).

D'autre part, les règles de mise en œuvre de l'héritage multiple diffèrent selon la sorte d'héritage : héritage simple pour `extends`, mais héritage multiple pour `implements`.

Du point de vue de l'héritage multiple, en *Java*, la différence entre `implements` et `extends` peut être décrite ainsi : *Un seul mot-clé suffirait mais il faudrait alors vérifier la condition suivante : toutes les classes héritées, sauf éventuellement une, doivent être totalement abstraites.*

Que retenir de l'initiative *Java* ?

- *Java* propose une règlementation de l'usage du mécanisme d'héritage en distinguant deux cas.
- *Java* n'autorise l'héritage multiple que pour les seuls besoins d'implémentation.

Ces deux points seront discutés dans la dernière partie de cette section. Auparavant, la suite présente la problématique de l'évolution des classes et des objets dans le cadre des LC, en tentant de montrer la proximité de ces problèmes avec ceux traités par les mécanismes objet.

2.3 Une limite : versions de classes et mutations d'objets

Les problèmes de gestion de versions et de mutation de types sont des problèmes récurents du domaine informatique. L'évolution fonctionnelle des applications entraîne des besoins d'évolution des modèles mis en jeu et un problème de gestion de versions se pose lorsque, par exemple, tel modèle doit *désormais* disposer d'une nouveau champ rendu obligatoire par une quelconque nouvelle réglementation. Si des entités persistantes décrites par le modèle de départ doivent suivre l'évolution du modèle se posera alors le problème de la mutation des entités.

Pour compléter le panorama des usages de la technologie objet, nous nous intéressons ici aux reponses qu'apportent ou pourrait apporter cette technolgie à ces problèmes.

Pour le premier problème (évolution fonctionnelle), la technique objet apporte une réponse : il est possible, par exemple à l'aide de l'héritage, de définir un nouveau modèle qui introduit le champ souhaité. C'est d'ailleurs un des *plus* de la technologie objet que d'apporter une notable amélioration dans le processus de conception d'une nouvelle version d'une application, la relation « est-un » se transformant intuitivement en « est-une version de ».

Mais il n'en est pas de même au niveau de la phase d'exploitation de la nouvelle application pour la mutation des entités. Parce qu'on s'intéresse alors à l'évolution des objets (un client donné, une facture particulière, etc.) créés à partir de l'ancienne version. À l'heure actuelle, ce problème est souvent partiellement éludé par le fait que la gestion des objets persistants est faiblement couplée avec le langage de programmation des applications. Par exemple, l'application sera écrite en C++, mais fera appel à un SGBD relationnel pour un stockage *ad hoc* des "objets" [2]. Traditionnellement, la migration fera alors l'objet d'une application spécialement écrite pour convertir les anciens objets sous leur nouvelle forme. Si on envisage maintenant le cas d'un langage objet intégrant un service de persistance *minimal*, tel, par exemple, que les objets persistants maintiennent un lien avec leur modèle de création et peuvent être indifféremment accédés sous leurs formes volatile ou persistante, le problème se pose alors avec plus d'acuité. Il devient indispensable d'envisager plus d'autonomie des objets.
L'application *invite* un objet à migrer vers sa nouvelle version, en lui fournissant, lorsque c'est possible, les informations indispensables à sa migration. Le cas le plus trivial, consistera à fournir la même valeur par défaut (ou l'algorithme de calcul de la valeur initiale) à attribuer au champ apparu dans la nouvelle version, alors que des cas plus complexes conduiront à des solutions moins automatiques (par exemple, si la valeur à placer n'est pas calculable (numéro de téléphone, etc.) [3]).

Pour permettre de tels comportements des objets, il faut disposer d'un mécanisme de mutation de type. En d'autres termes, il faut permettre de relâcher les règles de typage sous une certaine condition qui pourrait être décrite par une relation spécifique « est-une version de » entre le modèle-source et le modèle-destination.

2.4 Bilan : paramétrer et généraliser les mécanismes objet dans les LC

Dans les parties précédentes de cette section, nous avons présenté différents points de vues sur la technologie objet, en essayant de mettre l'accent sur les problèmes auquels il nous semble prioritaire de tenter d'apporter des solutions.

En résumé, qu'en retirons-nous ?

- le mécanisme d'héritage est, au départ, plûtot un mécanisme de sous-typage, mais son usage réel (utile) dépasse largement ce cadre ;
- les mécanismes objet sont des mécanismes de bas niveau multi-usages. En particulier, l'héritage fonctionne en mode "tout-ou-rien" ; par exemple, le polymorphisme est toujours activé alors qu'il n'a parfois aucun sens réel et n'est d'aucune utilité pour le concepteur ;
- les mécanismes objets, sous réserve d'être étendus (et règlementés) peuvent offrir un cadre pour la gestion de versions et la mutation de types ;

2. Ce problème de versions est aussi au cœur des recherches sur les SGBDOO.
3. Ou encore, quand il s'agit de construire un nouveau modèle à partir de plusieurs modèles existant.

– du point de vue de leur réglementation par le programmeur, les mécanismes objet d'agrégation (clientèle) et d'héritage ne sont pas au même niveau. Par exemple, *Java* (comme beaucoup de langages) offrent des services complets de règlementation des mécanismes d'encapsulation, mais ne propose que deux sortes d'héritage.

Réglementer l'héritage = paramétrer l'héritage

– *Java* propose une règlementation de l'usage du mécanisme d'héritage en ne distinguant que deux cas (`extends` et `implements`) et en limitant l'héritage multiple. En d'autres termes, le programmeur peut choisir la forme d'héritage qu'il souhaite : *Java* lui permet de paramétrer l'usage de l'héritage selon la nature conceptuelle des classes en jeu.

– *Java* réglemente (paramètre) l'héritage selon une dimension conceptuelle, mais d'autres dimensions pourraient être elles-aussi prises en compte. Citons deux exemples simples :

  – la possibilité d'interdire la substitution polymorphique pour un héritage réalisé en vue de simplement réutiliser du code et

  – la possibilité de limiter les droits de renommage/redéfinition dans le cadre d'un sous-typage.

Paramétrer l'héritage = généraliser l'héritage ?

Introduire des moyens de paramétrer l'héritage – dans son sens classique – implique des modifications notables des règles sémantiques du langage concerné. Et si on se place dans les cas limites de mutation de types ou de gestion de versions, le terme même d'"héritage" perd le sens commun qu'il a dans les langages objets.

De même, peut-on encore appeler "héritagé" le fait d'inventer une classe qui en "chapeauté" (l'inverse de "inherits") plusieurs autres ? Il s'agit pourtant d'un besoin réel, fondé sur l'observation que la programmation à grande échelle est une activité continue qui se réalise en descendant *et* en remontant. Schématiquement, des mots-clés tels que `abstracts` ou `generalizes` pourraient qualifier cet usage.

Finalement, il ressort que l'idée de paramétrer l'héritage contient en elle-même la notion de généralisation de l'héritage pour des usages nouveaux (mutation de types) ou simplement utiles (`generalizes`).

Paramétrer/généraliser les liens entre classes ?

Si nous intégrons maintenant dans notre discours le mécanisme de clientèle, ce sont donc deux types de liens entre classes (et entre instances) qui sont susceptibles de paramétrage/généralisation. En *Java*, par exemple, les mots-clés `public`, `private` et `protected` permettent directement au programmeur de paramétrer ses liens de clientèle, et les mots-clés `extends` et `implements` ses liens d'héritage.

Mais *Java* n'est qu'un exemple parmi d'autres. Dans la section suivante, nous résumons brièvement ce qui nous semble être les principales propositions visant à permettre le paramétrage/généralisation des liens entre classes. Lors de ce survol sera en particulier évoqué le problème crucial du niveau d'accès au mécanisme de paramétrage/généralisation autour du concept de langage ouvert.

3 Quelques LC supportant le paramétrage ou la généralisation de liens

Il est bon de rappeler ici que le thème qui nous intéresse dans ce rapport concerne, d'une facon générale, l'*évolution des langages à classes*, vaste thème qui, des aspects théoriques aux aspects expérimentaux, couvre de nombreuses directions. Dans ce contexte, existent de nombreuses propositions d'environnement de programmation qui visent à modifier, faciliter, assouplir, étendre ou règlementer, etc. la mise en œuvre de la technologie objet. Nous nous intéressons ici à *CLOS*, néo*ClassTalk*, *Iguana*, *OpenC++*, CodA et *VBOOL*.

Après un bref rappel de la problématique et des concepts sous-jacents, sont résumées les caractéristiques majeures de chacun de ces environnements.

3.1 Langages ouverts et Génie logiciel

Le terme d'environnement de programmation employé ci-dessus recouvre plusieurs situations qu'il y a lieu d'éclaicir au préalable selon deux axes :

Langage fermé et langage ouvert

Un langage fermé est un langage dans lequel la sémantique des opérations de base est entièrement figée. Dans le cas des LC, *Eiffel*, *C++* et *Java* sont de parfaits exemples d'une telle situation, dans le sens que, par exemple, l'opération d'accès "." (dot, la notation pointée) dispose d'une sémantique srictement définie par le langage et câblée par les compilateurs associés, et ceci, indépendamment du mode d'exécution retenu (interprétation ou production de binaire).
À l'inverse, un langage ouvert est fondée sur le principe qu'il est irréaliste de vouloir définir *le* langage universel qui contient tout ce qu'il faut (et qu'il faudra demain). En conséquence, il doit être possible de régler, selon ses besoins la sémantique des opérations de base du langage.
Conceptuellement. un langage ouvert représente une famille de langages, famille dont la taille et la cohérence est en définitive du ressort du "metteur au point" des opérateurs de base.

Concepts de base des langages ouverts

Un environnement de programmation pour un langage ouvert doit permettre de modifier, pour une application donné ou pour un site donné, la sémantique du langage.

Pour ce faire, plusieurs approches sont possibles et notablement différentes. Par exemple, sur l'échelle du "dynamique" au "statique", cela va du protocole MOP de Clos à la génération préalable du compilateur associé à un réglage donné de la sémantique du langage ouvert (*OpenC++*).

Pour autant, ces approches utilisent toutes, plus ou moins, les mêmes concepts de base pour le fonctionnement du noyau d'exécution. Nous les présentons brièvement ci-après :

- Réflexion : désigne la capacité d'une application à manipuler, au cours de son exécution, une description de son propre état.
- Introspection : faculté d'une application réflexive à s'interroger sur son propre état.
- Intercession : faculté d'une application réflexive à modifier sa sémantique.
- Réification : désigne la technique permettant à une application réflexive de gérer la description de son propre état (structures de données internes). L'introspection nécessite au minimum la réification des types de données manipulés par l'application. L'intercession implique la réification de la sémantique du langage, (afin de pouvoir modifier le code implémentant la sémantique du langage).

Le lecteur trouvera dans [Duc97], un chapitre présentant ces concepts et une bibliographie sur le thème des langages ouverts.

3.2 CLOS/MOP

CLOS [Kee89] est un système de programmation pour "programmer objet" en Lisp (Common Lisp Object System). Avec *CLOS*, le programmeur a à sa disposition une batterie de primitives qui lui permet de définir les différentes entités dont il a besoin dans une application (classes, attributs, méthodes).

Du point de vue de son implémentation, *CLOS* est un exemple parfait de langage ouvert. Fondamentalement réflexif de part son langage support (Lisp), il repose de plus sur un méta-protocole qui permet à un (méta)programmeur de modifier la sémantique des liens entre les entités d'une application.

La généralisation de ce protocole (écrit lui-même en *CLOS*) a finalement donné naissance à MOP (pour Meta-Object Protocol) [Gal95] dont l'ensemble des concepts et primitives permet de définir (ou d'adapter) un langage aux besoins spécifiques d'une application.

L'environnement *CLOS*/MOP représente certainement le niveau le plus avancé dans le domaine des langages ouverts. L'inconvénient principal que l'on peut lui reprocher est qu'il est situé strictement dans l'univers lispien, qui nous semble assez éloigné du monde du génie logiciel.

3.3 *NeoClassTalk*

NeoClassTalk est un langage de programmation proposé par F. Rivard dans le cadre d'un thèse de doctorat qu'il vient de terminer à l'École des Mines de Nantes. *NeoClassTalk* étend le langage *ClassTalk* en permettant de "relâcher la contrainte d'immutabilité" habituellement posée sur les liens d'instanciation et d'héritage, et ceci dans le cadre des langages à classes réflexifs.

L'objectif principal est de réduire la frontière entre les problèmes de conception et d'implémentation d'une application. En d'autres termes (ceux de la deuxième partie de ce rapport) l'objectif est d'implémenter de nouveaux usages de la technologie objet.

Techniquement, *NeoClassTalk* est est muni d'un protocole d'adoption d'instance autorisant tout objet à changer dynamiquement de classe, et d'un protocole permettant la modification dynamique de l'arbre d'héritage d'une classe. Cette approche permet d'apporter des solutions aux problèmes généraux d'évolution. Et plus particulièrement au changement dynamique de classe où l'ancienne et la nouvelle classe sont dans une relation d'héritage.

Selon l'auteur, «le changement de classe *en largeur* traduit alors plus spécifiquement des changements d'état tels qu'ils apparaissent dans les modélisations par graphe de transition d'état, alors que le changement *en profondeur* traduit des spécialisations du comportement. Généralisant ce dernier type d'évolution, le pattern de *spécialisation dynamique* autorise l'ajout/le retrait dynamique d'un comportement. Appliqué aux classes, il offre un cadre pour la composition dynamique de leurs propriétés, elles-mêmes représentées par d'autres classes».

Le résultat de cette étude est matérialisé par une nouvelle version de CLASSTALK (NEOCLASSTALK). NEOCLASSTALK intègre les protocoles de changement dynamique de classe et de superclasse. De plus, il supporte la réification de l'envoi de messages ou plus exactement, de l'application des méthodes. La réalisation de NEOCLASSTALK est faite en SMALLTALK [LP90, LP91, GR83] en se basant sur ses principaux aspects réflexifs.

NEOCLASSTALK est avant tout un outil de recherche pour l'étude des langages à classes réflexifs, ce qui cadre parfaitement avec nos objectifs. Résultant directement d'un travail de doctorat qui vient de s'achever, cet outil fera l'objet d'une étude plus approfondie dès que sa diffusion (dont le mémoire de thèse) sera réalisée.

3.4 *Iguana*

Iguana est un environnement de programmation pour *C++* qui vise à ouvrir ce langage en vue de faciliter, en particulier, la programmation d'application distribuées.

L'approche retenue consiste à réifier tous les comportements de classes *C++* (appel de méthode, accès aux données membres, etc.). Tous ces comportements sont transformés en objets de première classe, directement manipulables par le langage. Il est ainsi possible de changer la façon dont les données membres sont accédées, mais aussi de stocker les appels de méthodes dans une liste (pour une trace d'exécution par exemple) ou de faire des appels à des méthodes d'objets distants à travers un réseau.

La réflexivité mise en œuvre par *Iguana* est dynamique, c'est-à-dire que les méta-objets

existent pendant l'exécution du programme et qu'il est alors possible de changer, via de nombreux points d'accès, le méta-objet contrôlant un des comportements d'une classe.

À notre connaissance, *Iguana* est à l'état de projet ; il n'existe pas à l'heure actuelle d'environnement d'*Iguana* disponible.

3.5 *OpenC++*

OpenC++ [CM93] est une version de *C++* munie d'un protocole méta-objet. L'environnement fournit une boîte à outils puissante pour construire des extensions au langage *C++*. Par son ouverture, *OpenC++* doit permettre d'étendre le langage standard vers le parallélisme, la distribution et/ou la persistance. Un autre objectif d'*OpenC++* est de permettre le développement de supports d'exécution optimisés pour un usage donné.

La première version développée pour *OpenC++* peut être qualifiée de *dynamique*. Dans cette version, les méta-objets contrôlant l'exécution des appels de méthode (la sémantique d'une classe donnée) sont présents à l'exécution. Combinés avec la réification des appels de méthodes, cette technique permet de modifier dynamiquement la sémantique du code effectuant l'appel de méthode en passant par la méthode du méta-objet de la classe concernée.

En vue d'obtenir un environnement plus performant, les auteurs d'*OpenC++* proposent désormais une deuxième version qui concilie les objectifs d'ouverture de l'environnement et d'efficacité des applications engendrées.

Cette version 2, que l'on peut qualifier de *statique*, sépare distinctement les phases de programmation et de méta-programmation et permet au programmeur d'obtenir, au final, du code *C++* standard qu'il ne reste plus qu'à compiler.
Dans un premier temps, le méta-programmeur définit les méta-classes qui serviront aux programmeurs. Il s'agit ici d'implémenter les méthodes du MOP (`TranslateBody`, `TranslateMethodBody`, etc.) en donnant les règles de traduction à effectuer sur les textes (les arbres abstraits) des classes d'application. La phase *méta* s'achève par la compilation de ces méta-classes (par *OpenC++*), ce qui produit un nouveau compilateur pour un *nouveau* langage qui dispose d'une sémantique propre. Le niveau méta permet aussi d'étendre la syntaxe (mots-clés) du langage produit.
C'est ce nouveau compilateur qu'utilisera alors le programmeur pour développer ses applications. Techniquement, le compilateur appelera les (méta) méthodes de traduction définies dans chaque méta-classe pour produire le code *C++* adéquat.

L'environnement *OpenC++* est opérationnel. Il s'agit d'un exemple complet (voire deux) de développement d'un système de programmation ouvert à partir d'un langage industriel fortement typé.

Pour compléter cette section, notons qu'un projet probablement similaire, au moins dans ses objectifs, concerne le langage *Java*. Son étude sera effectuée dès la disponibilté d'articles le présentant [KG96].

3.6 CodA

Afin de compléter ce survol de différents environnements de langages ouverts, et avant de conclure par une proposition fondée sur le langage *Eiffel*, nous développons ci-dessous l'approche particulière retenu par J. McAFFER pour la définition d'un MOP (CodA) [McA95] susceptible d'être utilisé pour concevoir un langage ouvert.

Cette description est directement tirée du mémoire de thèse de Stéphane Ducasse [Duc97].

« *CodA n'est pas un langage réflexif, mais plutôt un protocole qui n'est pas lié à un langage particulier [...] CodA applique une décomposition logicielle au méta-niveau.*
Un méta-niveau est composé de méta-composants, des objets représentant des comportements spécifiques de la gestion des objets de la base. Chaque objet possède alors un méta-niveau conceptuel qui regroupe plusieurs méta-composants spécialisés. CodA définit sept méta-composants qui peuvent être à leur tour spécialisés ou étendus. Nous les présentons rapidement :

- « – *le composant* ***send*** *gère les interactions entre l'expéditeur et le receveur d'un message. Il achemine un message jusqu'au composant* ***accept*** *du receveur.*

- « – *le composant* ***accept*** *précise si le message est accepté, ce dernier pouvant alors être stocké par le composant* ***queue*** *pour traitement ultérieur.*

- « – *le composant* ***queue*** *gère les messages en attente de traitement.*

- « – *le composant* ***receive*** *correspond à la phase finale de réception d'un message qui est déclenchée lorsqu'un objet recherche un message à exécuter.*

- « – *le composant* ***protocol*** *a en charge la recherche de la méthode à exécuter.*

- « – *le composant* ***execution*** *a en charge l'application de la méthode trouvée.*

- « – *le composant* ***state*** *décrit la gestion de l'état interne d'un objet et les modes d'accès.*

J. Mc AFFER centre son MOP sur les activités des objets et non sur le langage l'implémentant. La décomposition (en méta-composants) d'un méta-iveau offre la possibilité de particulariser différemment le comportement des objets en offrant une vision plus fine du dit comportement [...] CodA décompose l'envoi d'un message à la fois du point de vue de l'objet émetteur et de l'objet receveur. CodA offre une description du comportement des objets au travers de méta-composants distincts (émission, réception des messages, ...) contrairement à CLOS ou ClassTalk dans lesquels la description du comportement des objets passe par une description des éléments du langage (classes, méthodes, variables, ...). La différence réside dans le choix de factorisation des comportements : en CLOS, les méta-comportements sont plus ou moins répartis dasn les cinq méta-objets de base ; dans CodA, un méta-comportement est regroupé dans un même méta-objet : un méta-composant. »

3.7 *VBOOL*

VBOOL est un nouveau langage qui propose d'étendre le langage *Eiffel* pour y intégrer un concept de *point de vue*. *VBOOL* (pour View Based Object Oriented Language) apporte à Eiffel une fonctionnalité de *visibilité répétée* et un mécanisme qui permet l'*évolution dynamique d'un point de vue* tout en garantissant la cohérence du modèle de base.

La *visibilité* est supportée par des constructeurs de points de vues et de vues. Elle permet d'abord de définir les vues d'une classe. On appelle vue une abstraction partielle du modèle (c'est à dire un sous-modèle). On appelle point de vue la vision qu'a un utilisateur du modèle ; un point de vue est une combinaison de n vues ($n >= 1$). La *visibilité* permet ensuite de filtrer l'information provenant d'une classe et d'en préciser les statuts d'*exportation en visibilité* vers ses classes dérivées ou ses instances. La présence de ces filtres procure ainsi une sorte d'héritage sélectif (droit de visibilité en fonction du point de vue courant).

L'intégration pratique de ces concepts dans le langage *Eiffel* est faite sous la forme d'une extension du concept de classe. *VBOOL* définit la notion de *classe flexible* comme étant une classe *Eiffel* augmentée d'un ensemble de clauses de visibilité dont la principale (nommée `seen_as`) permet de désigner plusieurs classes (éventuellement une) qui seront les vues de la classe flexible, avec des possibilités de renommage et de redéfinition d'Eiffel controlées par *VBOOL*. Définir un point de vue consiste alors, dans une classe cliente (flexible ou non) à définir un attribut citant à la fois la classe flexible et les vues choisies pour cet attribut. Par exemple, la déclaration d'attribut `X : AUTOMOBILE (ACHETEUR, MECANO)` rend visible à travers `X` les vues (classes) `ACHETEUR` et `MECANO`, sous réserve que `ACHETEUR` et `MECANO` aient été désignées par des clauses `seen_as` de la classe flexible `AUTOMOBILE`.

Enfin, pour apporter des services d'évolution dynamique de points de vues, *VBOOL* offre des primitives de gestion qui permettent à toute application de modifier un point de vue en ajoutant des vues (`X.etablish_view (CONCESSIONNAIRE, ADMINISTRATION)`), en supprimant des vues (`X.suppress_view (ACHETEUR)`) ou bien encore en échangeant des vues (`X.exchange_view (MECANO, ASSURANCES)`), sous le contrôle statique et dynamique de l'environnement *VBOOL*.

A notre connaissance, seul un environnement partiel sous `Centaur` a été réalisé (sans généricité, assertions, ancres, et création typée) et aucun environnement de programmation n'est défini (système ouvert, précompilateur, etc.).

4 Paramétrage pour des liens d'héritage et de clientèle

4.1 Pourquoi paramétrer?

Après la modélisation, avant la programmation, il existe une phase de « conception logicielle » qui ne doit pas être négligée. Elle consiste notamment, dans une approche à objets, à étudier précisément l'usage qui sera fait de l'héritage, de la clientèle et, en général,

des liens entre classes [Lie86].

L'héritage est, sans nul doute, le plus puissant mais aussi le plus controversé de ces liens [Car84, CW85, Coo89]. Est-ce un lien entre classes ou entre types ? Peut-il être utilisé uniquement pour « récupérer » du code existant ? Doit-il se limiter au cadre stricte du sous-typage ? Et bien d'autres questions encore.

Nous n'avons pas la prétention de répondre ici à ces questions qui n'ont d'ailleurs probablement aucune réponse définitive. Notre but est plutôt de travailler sur l'existant. Nous avons l'héritage, il a des défauts et des qualités et il est utilisé selon plusieurs objectifs parfois peu compatibles entre eux. La question que nous nous sommes posés est donc : voyons comment nous pouvons analyser, comprendre puis structurer la manière dont nous utilisons ce lien ?

Pour cela, nous avons choisi de déterminer des paramètres de liens qui nous permettent de décrire l'héritage et la clientèle[4], tout d'abord, puis d'autres liens par la suite. Nous avons réifié, au sein de notre système *OFL*, le concept de lien entre classes[5]. Nous ne remettons pas en cause, par cette modélisation, les programmes existants en *Eiffel* car notre première tâche est de décrire le plus fidèlement possible les liens déjà existants.

Notre travail se poursuit ensuite dans l'extension d'*OFL*[6] avec des liens spécialisés, le plus souvent dérivés de l'héritage (exemples : abstraction, réutilisation, ...). Nous nous rapprochons ici des travaux de [Duc97] qui, dans son système *FLO*, réintroduit le lien d'héritage comme une dépendance entre classes. À la différence de ce concept de dépendance qui peut être placé entre deux objets quelconques, nos liens ne peuvent être placés qu'entre des objets de classe `M_CLASS` (pour *meta-class*). Notre démarche est donc plus spécifique.

La section 4.1 page 16 présente le problème et donne les arguments qui nous ont incité à y trouver une solution. La section 4.2 page 21 décrit les paramètres que nous avons sélectionnés.

Pour entamer cette partie, précisons une chose. Les paramètres que nous définissons sont accessibles au niveau méta (*OFL*) directement pour générer de nouveaux liens. Dans l'état actuel des choses, nous préférons ne pas les offrir au niveau de la programmation (*OFL-Eiffel* qui représente un *Eiffel* étendu), bien que cela soit *a priori* possible, pour bien séparer les activité de méta-programmation des activités de programmation. Nous avons donc *OFL* qui permet de générer de nouvelles sortes de liens grâce à une série de paramétrages et *OFL-Eiffel* qui permet d'utiliser ces liens en plus de l'héritage et de la clientèle.

Pour que notre discours soit plus clair, nous prendrons, à chaque fois que cela sera né-

4. Nous avons choisi l'héritage et la clientèle du langage *Eiffel* comme liens typiques. De manière générale, *Eiffel* nous servira constamment de langage à objets typique.

5. Nous prenons pour axiome qu'une classe décrit un ou plusieurs types (seules les classes génériques pouvant, dans les faits, décrire plusieurs types). Clarifions donc tout de suite une possible ambiguïté : nos liens portent sur les classes. Quand une classe décrit plusieurs types, chaque lien décrit au niveau de cette classe s'applique de manière identique sur chacun des types qu'elle décrit (ou ne s'applique sur aucun s'il s'agit d'un lien n'affectant pas les types).

6. *OFL* se présente sous la forme d'une application *Eiffel* réifiant les concepts d'*Eiffel* (exemples de concepts réifiés : classes, liens entre classes, primitives, instructions, ...) en en généralisant certains, permettant ainsi de les faire évoluer selon nos besoins.

cessaire, un exemple appuyant notre démarche. Dans la mesure du possible, nous essaierons de présenter des exemples les plus simples et significatifs qu'il est possible. Comme nous le rappellerons plus loin, précisons que l'intégration de notre système *OFL* n'entraîne pas de conséquences sur les codes *Eiffel* existants puisqu'il s'agit de gérer uniquement des informations additionnelles certes utiles mais non indispensables (comme le sont les assertions). Tous les comportements par défaut sont maintenus.

4.1.1 Augmenter l'expressivité

La paramétrisation des liens entre classes permet d'enrichir le mécanisme (qui, actuellement, repose uniquement sur la clientèle ou l'héritage), aussi bien sur les aspects purement pratiques que d'un point de vue modélisation. Dans [Mey96a, Mey96c], Bertrand Meyer définit plusieurs catégories d'usage de l'héritage. Nous en reprenons un certain nombre afin de montrer les problèmes, souvent bien connus, associés à l'utilisation de l'héritage.

Exemple sur l'« héritage de machine » Penchons-nous un instant sur celle appelée héritage de machine. L'exemple donné est celui de la classe `EXCEPTIONS` qui permet la gestion d'un mécanisme d'exceptions. Dans l'usage de l'héritage, le fait qu'une classe `RECTANGLE` hérite d'`EXCEPTIONS` pour l'utiliser[7] entraîne deux conséquences fondamentales :

1. On peut utiliser dans `RECTANGLE` des fonctionnalités d'`EXCEPTIONS` (l'héritage agit sur les classes).

2. Tout `Rectangle`, instance de `RECTANGLE`, est, de plus, une `Exception`, instance d'`EXCEPTIONS` (l'héritage agit sur les types).

Si la première conséquence est exactement celle que l'on veut obtenir en réalisant cet héritage, la deuxième n'est qu'un effet secondaire pour le moins malheureux et difficilement défendable du point de vue modélisation (Peut-on raisonnablement argumenter qu'un rectangle est une exception ? Si oui, pourquoi un bateau ne serait-il pas un tableau ?) et programmation (Rien n'interdit de faire jouer le polymorphisme entre `RECTANGLE` et `EXCEPTIONS`, mais cela a-t-il un sens ?).

Précisons que l'usage de l'héritage traditionnel sur `EXCEPTIONS` peut tout à fait être légitime dans certains cas. Prenons pour simple exemple une classe `MON_EXCEPTION` qui modifierait certains comportements pour des actions plus spécifiques des exceptions dans certains composants. Enfin, rien dans l'activité de modélisation n'empêche d'utiliser une clientèle (selon Bertrand Meyer, les seuls cas où l'héritage est valide sont ceux où l'on peut raisonnablement argumenter de l'existence du lien « est-un »).

Nous pensons ici qu'un lien d'« utilisation » (assez proche de la clientèle en conservant cependant l'idée d'importation des primitives) pourrait faire l'affaire au lieu de l'héritage.

7. En *Eiffel*, il est indispensable d'hériter d'`EXCEPTIONS` pour pouvoir utiliser un mécanisme d'exceptions.

Exemple sur l'« héritage d'abstraction » Voyons maintenant ce qui est présenté comme l'héritage d'abstraction qui consiste à rendre abstraites certaines primitives d'une classe. Il peut être utilisé, entre autres et comme spécifié dans [Mey96a], quand on rencontre deux conditions :

1. Une hiérarchie de classes fournit une classe pour laquelle on a besoin d'une abstraction (dans le sens généralisation).

2. On ne peut ou ne veut pas modifier cette hiérarchie.

Disons que `CARRÉ` appartienne à une telle hiérarchie mais pas `RECTANGLE` (imaginons, pour simplifier mais sans nuire au raisonnement, que `CARRÉ` n'hérite d'aucune classe exprimant des propriétés de figures géométriques telles `QUADRILATÈRE` ou `LOSANGE`). L'héritage d'abstraction consiste à abstraire (en rendant abstraites les primitives qui n'ont pas de sens, en généralisant celles qui en gardent un et en ajoutant celles qui en prennent un) des primitives dans `RECTANGLE`.

Comme précédemment, cet héritage entraîne deux conséquences fondamentales :

1. On peut utiliser dans `RECTANGLE` des fonctionnalités de `CARRÉ`.

2. Tout `Rectangle`, instance de `RECTANGLE`, est aussi un `Carré`, instance de `CARRÉ`.

Ici aussi, la première conséquence est compatible avec nos désirs. La deuxième, quant à elle, pose un vrai problème car elle décrit exactement l'inverse de ce que l'on veut obtenir. Nous voulons qu'un `Carré` soit un `Rectangle` et nous avons le contraire. Ce résultat est bien sûr dû à la volonté de ne pas remettre en cause la hiérarchie existante : cela conduit à ajouter une « verrue » (la classe `RECTANGLE`) en négligeant les conséquences de cet ajout. Il serait beaucoup plus profitable, de notre point de vue, de pouvoir réaliser cette adaptation en ayant la possibilité d'ajuster les paramètres du lien entre `CARRÉ` et `RECTANGLE` pour clarifier la situation (par exemple, en autorisant ici un polymorphisme ascendant au lieu de descendant et en précisant que tout `Carré` est un `Rectangle` du point de vue des extensions[8]).

Qu'il soit bien clair que la manière « propre » de créer `RECTANGLE` serait de la placer au-dessus de `CARRÉ` dans la hiérarchie. Nous étudions ici le cas où cela n'est pas possible (par exemple, si l'on ne possède pas les sources de `CARRÉ` ou si cette modification entraînerait un surcoût prohibitif dans la modification des classes dépendant de `CARRÉ`). Notre propos est bien ici, au contraire, de montrer que l'on pourrait « proprement » faire évoluer cette hiérarchie, *a priori* figée, grâce à l'usage de paramètres pour ce lien (appelons-le « abstraction ») qui, comme l'héritage, agit sur les classes et sur les types.

Exemple sur l'« héritage d'implémentation » Basons nous toujours sur la taxinomie de l'héritage donnée dans [Mey96a] et étudions, pour ce dernier exemple, l'héritage

8. Ces deux modifications conduisent à faire de `CARRÉ` un héritier de `RECTANGLE`, ce qui est naturel.

d'implémentation. Un héritage d'implémentation est utilisé quand une classe a besoin d'hériter d'une autre classe uniquement pour des impératifs de programmation et à l'exclusion d'une volonté de sous-typage. Cette catégorie d'héritage se rencontre souvent dans ce que Bertrand Meyer appelle un « mariage de raison ». Un exemple donné est celui de la classe `ARRAYED_STACK` qui hérite de `STACK` (par un « héritage de réification ») et d'`ARRAY` par un héritage d'implémentation.

Nous sommes d'accord sur le fait que cette catégorie d'héritage est conceptuellement et pratiquement valide. Mais il signifie clairement une chose : les programmeurs de la classe `ARRAYED_STACK` ont voulu faire d'une `Arrayed_Stack` une `Stack`. Pour des raisons pratiques, il leur était commode d'hériter aussi d'`ARRAY` bien qu'une `Arrayed_Stack` ne soit en rien un `Array`. N'aurait donc-t-il pas été naturel de spécifier, entre autres, que l'usage du polymorphisme était proscrit entre `ARRAYED_STACK` et `ARRAY` car le présent lien (disons « implémentation ») ne doit avoir d'effet que sur les classes et non sur les types ? Il nous semble que cette précision, toute optionnelle qu'elle puisse être, n'en aurait pas moins été très utile. Elle permettrait de réellement séparer la classe qui sert d'implémentation (ici, `ARRAYED_STACK`) de celle qui ne définit en fait pour cette classe d'implémentation que l'interface (ici, `ARRAY`) [PI92, MR95].

Nous nous rapprochons ici, de manière évidente, du lien d'utilisation que nous suggérions dans le premier exemple.

4.1.2 Clarifier les usages et promouvoir le génie logiciel

La déclaration de fonctionnalités souhaitées exige et engendre une meilleure compréhension du programme par le programmeur et facilite notablement la maintenance.

En effet, en prenant pour exemple les assertions, on peut remarquer que l'explicitation d'une information[9] nécessite inévitablement une bonne compréhension des concepts programmés et entraîne donc une qualité plus grande du logiciel produit. L'effort supplémentaire demandé au programmeur est donc largement compensé, non seulement par la qualité, mais aussi par une plus grande maintenabilité et évolutivité.

La possibilité d'interdire ou de contrôler certains mécanismes de base permet d'assurer qu'un mauvais[10] usage de la classe pourra être évité. Ce type de spécifications encouragées mais optionnelles ne peut donc, à notre sens, qu'améliorer la qualité du logiciel produit.

Reprenons un exemple cité plus haut : `ARRAYED_STACK` hérite de `STACK` et d'`ARRAY`. Le lien d'implémentation qui existe entre `ARRAYED_STACK` (classe-source[11]) et `ARRAY` (classe-cible) déclare clairement qu'`ARRAYED_STACK` a été implémentée à partir d'`ARRAY` sans pour autant être un sous-type d'`ARRAY`. Ainsi la situation ne peut plus prêter à confusion.

9. Il s'agit bien d'un information additionnelle puisque les comportements de la clientèle et de l'héritage sont inchangés.

10. « Mauvais » est ici à prendre au sens de « non conforme à l'objectif déterminé par le programmeur de la classe » : cela se rapproche une fois encore des habitudes d'usage des assertions.

11. Nous appelons « classe-cible » la classe qui décrit le lien et « classe-source » la classe citée dans cette description. Un lien lie donc une (ou plusieurs) classe(s)-source(s) à une (ou plusieurs) classe(s)-cible(s). Par exemple : dans le lien d'héritage, l'héritière est la classe-source et l'héritée la classe-cible ; dans le lien de clientèle, la cliente est la classe-source et le fournisseur la classe-cible.

4.1.3 Conserver ses habitudes

« Cela fonctionne très bien comme ça ! » pourrait-on nous rétorquer. Aussi, par défaut, tous les comportements habituels de l'héritage étant conservés, les habitudes des programmeurs récalcitrants à notre technique ne seront pas perturbées. De la même manière, en *Eiffel*, l'usage des assertions peut être recommandé mais seul le programmeur décidera, en fin de compte, de les utiliser ou non. Seul un ajout d'information (qu'il s'agisse de spécifier une assertion ou de d'utiliser un lien plus spécialisé à la place de l'héritage ou de la clientèle) entraîne des modifications.

Cette condition de continuité est pour nous essentielle. Il nous semble naturel de permettre aux programmeurs d'utiliser une nouvelle technique qui leur semble pertinente. Mais il ne faut pas oublier qu'aucune méthode ne fait jamais l'unanimité et que, quel que soit son champ d'application, elle ne peut couvrir tous les besoins et vœux. De plus, les contraintes d'évolution nous indiquent nettement qu'il est déraisonnable de prétendre pouvoir réécrire la totalité du code existant sous le prétexte d'une avancée technique : il faut donc pouvoir cohabiter non seulement avec de l'ancien code mais aussi avec du nouveau code n'utilisant pas une technologie. C'est pour cela que nous avons souhaité que notre système reste optionnel et compatible avec l'existant.

4.2 Les axes de paramètrage de l'héritage

Cette section décrit les points importants qui nous aideront à déterminer les paramètres que nous sélectionnerons finalement.

4.2.1 Lien entre classes ou entre types

Un lien peut s'appliquer soit entre des classes soit entre des types soit entre des classes et entre les types associés à ces classes. Dissocier ces deux modes d'utilisation permet de mettre en évidence les liens relevant du typage (on parle souvent des nouvelles classes qui sont des sous-types des anciennes) et ceux relevant de l'utilisation de code — soit en important du code que l'on peut éventuellement adapter (héritage), soit en ayant la possibilité d'exploiter le code distant en y faisant référence (clientèle). La combinaison des deux cas est évidemment elle-même tout à fait valide. L'héritage[12] est d'ailleurs dans ce cas : il s'applique aux classes (réutilisation voire factorisation de code) et aux types (sous-typage covariant).

Cette caractéristique peut être décomposée en plusieurs autres. Nous dirons qu'un lien s'applique à un type s'il permet le polymorphisme ou s'il propage les extensions des classes ou les deux. Le sous-typage est un cas particulier de lien sur les types : le polymorphisme est possible en descendant et l'extension des classes-cibles est dépendantes de celle des classes-sources.

Dans notre modèle, tout lien s'applique forcément sur les classes et optionnellement sur les types.

12. Rappelons une dernière fois qu'il s'agit de celui d'*Eiffel*.

4.2.2 Extension

Comme nous venons de le voir, la gestion de l'extension des types influe sur le fait que le lien s'applique aux types ou pas. Dans l'héritage, l'extension d'une classe-cible[13] est l'union des extensions de ses classes-sources. Dans la clientèle, les extensions des classes-cibles et classes-sources sont indépendantes.

Dans *OFL*, la gestion des extensions des classes est faite sur un modèle paresseux. Lorsqu'il est demandé à une classe ou à un type de fournir son extension, celle-ci envoie un message aux classes (ou aux types) dont son extension dépend, calcule son extension en fonction des réponses et de son extension propre[14] et rend le résultat de ce calcul. Ainsi, la gestion de la cohérence des extensions est simplifiée par ce principe.

Une autre méthode simple consiste à stocker l'extension dans chaque type et chaque classe et à propager des messages de création et de destruction pour maintenir la cohérence. Chacune de ces méthodes présentent des inconvénients dans certains cas critiques (en cas de demandes fréquentes des extensions pour la première, en cas de création et destruction en masse pour la seconde).

OFL permet de gérer des dépendances ascendantes (des classes-sources vers les classes-cibles), descendantes (des classes-cibles vers les classes-sources), horizontales (entre classes-sœurs[15]), voire transversales (entre classes quelconques).

Lors de la création d'une classe, son extension propre, vide par défaut, peut être initialisée grâce à une requête (par exemple, pour intégrer des instances persistantes).

4.2.3 Assertions

Le comportement des assertions doit suivre la sémantique que l'on souhaite donner aux liens. Ainsi, pour un lien d'abstraction (qui, rappelons-le, est l'inverse du lien d'héritage), il faut que les assertions soient plus générales dans la classe-source que dans la classe-cible : c'est exactement l'inverse dans l'héritage. *OFL* permet de gérer aussi bien un comportement contravariant (lien d'héritage) que covariant (lien d'abstraction) des assertions.

En ce qui concerne les assertions dans le lien d'exploitation de code distant (exemple typique : la clientèle), le comportement de la clientèle (qui permet de mettre en œuvre la programmation par contrats) est conservé.

Dans tous les cas, il est de toute façon possible de programmer au niveau de la couche méta d'*OFL* pour obtenir un comportement plus spécifique.

4.2.4 Généricité

Dans *OFL*, décrire une classe générique revient à décrire autant de liens de clientèle qu'il y a de paramètres de généricité. Par exemple, une classe `OFL_LIST[T ->OFL_ANY]`

13. L'extension d'une classe est l'union des extensions des types qu'elle décrit.

14. L'extension propre d'une classe est l'ensemble des instances propres de cette classe.

15. Des classes sont dites « classes-sœurs » si et seulement si elles sont classes-sources d'une même classe-cible (pour un lien donné).

décrit un lien de clientèle avec `OFL_LIST` comme classe-source et `OFL_ANY` comme classe-cible. La généricité est donc considérée, au niveau méta-programmation, comme un sucre syntaxique permettant, au niveau programmation, d'écrire plus naturellement certaines clientèles. Tout ceci est bien entendu transparent au niveau de la programmation.

Au niveau du modèle, la généricité permet de décrire plusieurs types avec une même classe (prenons un exemple : une classe `OFL_LIST[T ->OFL_ANY]` peut ainsi décrire, entre autres, les types `OFL_LIST[STATEMENT]` et `OFL_LIST[FUNCTION]`).

4.2.5 Polymorphisme

Le polymorphisme est la capacité d'un objet à être considéré comme une instance d'un type autre que son type propre. Il s'agit donc d'un cas particulier de gestion de points de vue : un objet ne peut avoir qu'un seul type propre, cependant, il peut être, en accord avec la gestion des extensions, instance (directe ou indirecte) de plusieurs types. Conformément à la sémantique des extensions de l'héritage, une instance d'une classe-cible peut être vue (c'est le traditionnel polymorphisme descendant) comme une instance de n'importe laquelle de ses classes-sources (directes ou indirectes).

Le polymorphisme est un concept plus général dans *OFL*. Il permet de mettre en place la notion de points de vue : un point de vue est une perspective qui offre une interface avec un objet, cette interface laissant apparaître cet objet comme instance d'un type différent de son type propre. Dans cette optique, le polymorphisme peut être aussi bien ascendant que descendant voire horizontal ou transversal (selon les liens).

4.2.6 Migration

La migration est la capacité d'un objet à changer de type propre. On peut la voir, du point de vue des types, comme une sorte de métamorphose permanente. La migration est un point crucial de la problématique de l'évolution. Si, pour des objets volatiles, elle présente un intérêt ; elle devient décisive lorsque l'on s'intéresse à des objets persistants. La durée de vie de tels objets peut être très longue et leur capacité à évoluer peut être dépendante de celle à migrer. Rappelons que dans le cadre de l'héritage, la migration n'est pas possible (sauf entre certains types de bases).

Contrairement au simple polymorphisme, la migration ne peut pas être totalement automatisée dans de nombreux cas (à moins d'accepter de fort probable pertes d'informations sémantiques). Notre politique est de l'automatiser dans les cas simples, d'assister le programmeur dans les cas un peu plus ambigus, et de lui laisser franchement la main dans les cas les plus difficiles. Comme le polymorphisme, la migration peut évidemment être ascendante, descendante, horizontale ou transversale. On peut envisager de l'interdire ou de la contrôler entre des classes liés par certains types de liens.

4.2.7 Clauses d'adaptation

Nous définissons les clauses d'adaptation comme les mécanismes permettant à une classe-source d'adapter les primitives importées de ses classes-cibles. Nous avons déterminé

un emsemble de quatre catégories de clauses d'adaptation :

- renommage (pour changer le nom d'une primitive importée),
- redéfinition (pour modifier une primitive importée),
- annulation (pour refuser une primitive importée) et
- sélection (pour choisir entre deux primitives importées).

Ces catégories devraient permettre de gérer tous les cas nécessaires. Par exemple, en *Eiffel*, les clauses `redefine` et `undefine` se classent dans notre catégorie redéfinition (en partant du principe que `undefine` sert à redéfinir une primitive concrète en primitive abstraite). Nous avons réduit cette liste à quatre catégories car elles nous semblent couvrir la totalité des besoins. Bien évidemment, nous en ajouterons d'autres si cela s'avère utile.

Signalons que la sémantique des ces primitives reste constante quel que soit les liens sur lesquelles elles s'appliquent. Leur implantation peut varier mais elle ne doivent pas sortir du champ donné par leur sémantique (évoquée ci-dessus).

4.2.8 Opérateurs de base

Un opérateur de base (que nous appelons aussi parfois simplement opérateur) est une méthode qui peut s'appliquer à tout objet, indépendamment de son implantation et de sa sémantique. Prenons comme exemple l'envoi de message ou l'affectation ou encore la création (qui est un méta-opérateur car il ne s'applique en fait qu'aux objets qui sont des classes).

Nous partons du principe que les opérateurs sont des briques de base des langages et ne permettons pas, *a priori*, de les redéfinir. Cependant, cela pourra être remis en cause dans une version ultérieure d'*OFL*.

4.2.9 Influences inter-liens

La gestion de plusieurs types de liens peut produire des conflits. Pour cette raison, nous ne traitons pas directement les liens mais plutôt des compositions de liens de même type. Par exemple, la classe `B` hérite deux fois de la classe `A` et en est cliente une fois. Cette situation, purement démonstrative, est présentée dans la figure 1 page 25.

Ainsi les conflits entre liens d'héritage, par exemple, sont gérés par la composition de ces liens. Quant aux conflits entre liens de type différent, il dénote soit une impossibilité de composition, soit sont gérés simplement dans l'ensemble des liens entre les classes.

Pour les deux liens pré-existant héritage et clientèle, ils n'ont d'influence l'un sur l'autre que par l'intermédiaire du masquage de primitives dans les descendants. Cela montre que des interactions sont possibles sans forcément pouvoir générer des conflits.

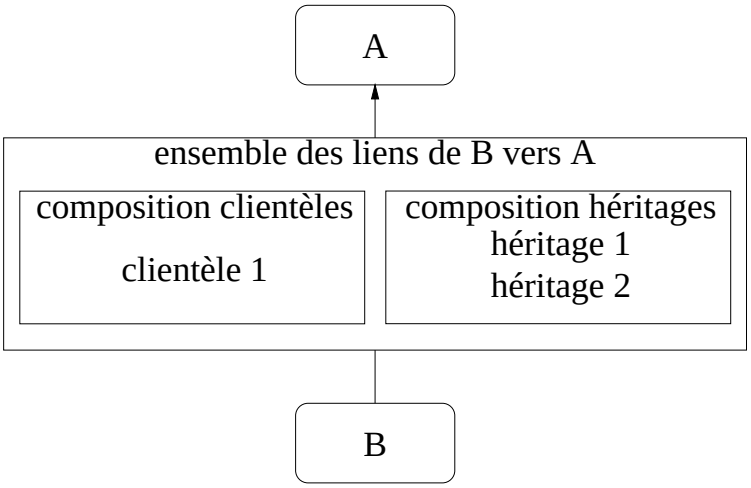

FIG. 1 - *Les compositions de liens de même type*

5 Conclusion

Ce projet en cours (*OFL*) vise à étudier et à améliorer l'usage des liens entre classes dans les langages à objets. Ce rapport de recherche a explicité, dans cet objectif, que l'héritage pourrait être considéré comme un mécanisme de bas niveau utilisé de manière très variée pour exprimer une certain nombre de liens sémantiques et a précisé des champs de pramétrage possible. La modification proposée permettrait ainsi d'améliorer notablement le processus d'évolution des applications. En parallèle de la modélisation, la programmation d'un prototype est engagée, toujours dans le cadre du projet *OFL*.

Références

[Car84] CARDELLI L. "A semantics of Multiple Inheritance". In *Symp. on Semantics Data Types*, num. 173 in Lecture Notes in Computer Science, Sophia Antipolis (F). Springer-Verlag (Berlin), June 1984.

[CCL96] CHIGNOLI R., CRESCENZO P. AND LAHIRE P. "An Extensible Environment for Views in Eiffel". Tech. report 96-53, I3S – UPRESA CNRS 6070 – Université de Nice – Sophia Antipolis, November 1996, 15 pages.

[CM93] CHIBA S. AND MASUDA T. "Designing an Extensible Distributed Language with a Meta-Level Architecture". In NIERSTRASZ O. M. (ed.) , *European Conference on Object-Oriented Programming(ECOOP'93)*, vol. 707 of *Lecture Notes in Computer Science*, pages 483–502, Kaiserlautern, Germany. Springer-Verlag (Berlin), 1993.

[Coo89] COOK W. "A Proposal for Making Eiffel Type-safe". *Computer Journal*, vol. 32, num. 4 (1989), pages 305–311.

[CW85] CARDELLI L. AND WEGNER P. "On understanding Types, Data Abstraction and Polymorphism". *ACM Computing Surveys*, vol. 17, num. 4 (December 1985), pages 471–522.

[Duc97] DUCASSE S. *Intégration réflexive de dépendances dans un modèle à classes.* Thèse de doctorat, spécialité Informatique, Université de Nice – Sophia Antipolis, 10 Janvier 1997, 252 pages.

[Gal95] GALLESIO E. "STklos Reference Manual". Tech. report RT 95-31, I3S CNRS / Université de Nice - Sophia Antipolis, July 1995, 82 pages.

[GR83] GOLDBERG A. AND ROBSON D. *Smalltalk-80 — The Language and its Implementation.* HARRISON M. A. (ed.) . Computer Science. Addison-Wesley Publishing Co. (Reading, MA), 0-201-11371-6, 1983, 718 pages.

[Kee89] KEENE S. E. *Object-Oriented Programming in Common Lisp – A Programmer's Guide to CLOS.* Addison-Wesley Publishing Co. (Reading, MA), 0-201-17589-4, 1989, 266 pages.

[Lie86] LIEBERMAN H. "Delegation and Inheritance : Two Mechnisms for sharing Knowledge in Object-Oriented Systems". In *BIGRE + GLOBULE 1986*, pages 79–89. January 1986.

[LP90] LALONDE W. R. AND PUGH J. R. *Inside Smalltalk – Vol 1.* Prentice Hall Inc. (Englewood Cliffs, NJ), 0-13-468430-3, 1990, 512 pages.

[LP91] LALONDE W. R. AND PUGH J. R. *Inside Smalltalk – Vol 2.* Prentice Hall Inc. (Englewood Cliffs, NJ), 0-13-467309-3, 1991, 553 pages.

[McA95] McAffer J. "Meta-Level Programming with CodA". In Olthoff W. (ed.) , *European Conference on Object-Oriented Programming(ECOOP'95)*, vol. 952 of *Lecture Notes in Computer Science*, pages 190–214, Aarhus, Danemark. Springer-Verlag (Berlin), August 1995.

[Mey96a] Meyer B. "Draft fragments from *Object-Oriented Software construction*, second edition", 1996. from `www.eiffel.com/`.

[Mey96b] Meyer B. "Building Bug-free O-O Software: An Introduction to Design by Contract". *Object Currents, www.sigs.com/objectcurrents/*, vol. 1, num. 3 (March 1996).

[Mey96c] Meyer B. "The many faces of inheritance: A taxonomy of taxonomy". *IEEE Computer*, vol. 29, num. 5 (May 1996), pages 105–108.

[Mey97] Meyer B. *Object-Oriented Software construction*. The O-O series. Prentice Hall Inc. (Englewood Cliffs, NJ), `0-13-629155-4`, 2nd, 1997, 1254 pages.

[MR95] Murray B. and Robson A. "On Behavior, Inheritance, and Evolution". *Journal of Object Oriented Programming*, vol. 8, num. 5 (September 1995), pages 38–42.

[PI92] Porter III H. H. "Separating the subtype hierarchy from the inheritance of implementation". *Journal of Object Oriented Programming*, vol. 4, num. 9 (February 1992), pages 20–29.