Raffinement de classes dans les langages à objets statiquement typés

Jean Privat — Roland Ducournau

LIRMM – Université Montpellier II
161 rue Ada
F-34392 Montpellier cedex 5
{privat, ducour}@lirmm.fr

RÉSUMÉ. *Les classes et la spécialisation apportent simultanément structuration et souplesse aux programmes objets. Ces deux qualités sont, malgré tout, souvent considérées comme insuffisantes, comme en témoignent les nombreuses propositions d'extension du modèle (aspects, modules, etc.). Cet article propose deux notions couplées de raffinement de classes et de modules, la première apportant la souplesse et la seconde, la structuration. Les modules regroupent un ensemble cohérent de définitions de classes et ont également la capacité de modifier les classes définies dans les modules dont ils dépendent.Cette proposition se place dans un cadre de typage statique, où les modules sont compilables séparément. Elle est basée sur un méta-modèle des modules analogue à celui des classes et les problèmes liés à la spécialisation ou au raffinement multiple sont traités de façon similaire à l'héritage multiple.*

ABSTRACT. *Classes and specialisation bring simultaneously structure and flexibility to object oriented programs. However, numerous model extension propositions (aspects, modules, etc.) prove these qualities are often considered insufficient. This article propose two binded notions of class refinement and modules, the first one bring flexibility and the second one, the structuration. Modules contains a coherent set of class definitions and can modify classes defined in modules they depend. This proposition aims statically typed languages where module can be separately compiled. It is based on a module metamodel analogue to the class one and problems with multiple specialisation and refinement are managed like those in multiple inheritance.*

MOTS-CLÉS : *raffinement de classes, langages de programmation orienté objets, héritage multiple, modules, compilation séparée.*

KEYWORDS: *class refinement, object-oriented programming languages, multiple inheritance, modules, separate compilation.*

1. Introduction

Les langages de programmation orientés objet offrent aux programmeurs un puissant cadre de développement qui permet à la fois de construire des entités stables et cohérentes (les classes) tout en autorisant une réutilisabilité et une évolutivité grâce à la spécialisation et au mécanisme d'héritage [MEY 88].

Malgré tout, ces qualités ne sont pas toujours suffisantes et de nombreuses approches ont été développées, allant de la réflexion permettant de modifier à la compilation, voire à l'exécution, le fonctionnement d'un programme jusqu'aux aspects permettant de « tisser » de nouveaux comportements à des classes existantes.

Le modèle de raffinement de classes que nous proposons fait partie de ces approches mais se focalise sur les langages à typage statique en héritage multiple. Il se caractérise par une décomposition modulaire orthogonale aux classes dans laquelle chaque module est autonome, entendre « compilable séparément », mais peut *raffiner*, c'est-à-dire étendre, une ou plusieurs classes existantes de modules parents en ajoutant de nouvelles propriétés, en modifiant des propriétés existantes, en ajoutant des relations de spécialisation, en unifiant les classes ou en généralisant des propriétés.

Notre proposition se base sur une analogie structurelle entre les modules et les classes où le méta-modèle des modules est semblable à celui des classes, les mécanismes et difficultés liés aux dépendances multiples entre modules devenant analogues à ceux que l'on traite en héritage multiple.

La section 2 expose la problématique du raffinement et les motivations. La section 3 résume rapidement le méta-modèle objet considéré sans raffinement en insistant sur l'héritage multiple, en reprenant l'analyse faite dans [DUC 95]. La section 4 introduit le raffinement de classes au niveau du modèle puis la section 5 s'intéresse aux conflits d'héritage et d'importation multiple et à leur résolution. La section 6 couvre l'implémentation et les techniques de compilation. La section 7 regroupe des travaux connexes. Enfin, la section 8 présente conclusion et perspectives.

2. Problématique du raffinement

2.1. *Besoins de souplesse et de structuration*

De nombreux travaux proposent des schémas et des implémentations de techniques de raffinement de classes qui permettent de définir une classe à un endroit du programme puis d'amender cette définition à divers autres endroits. La souplesse apportée par ces techniques va en général de pair avec une structuration des programmes non seulement sous forme de classes mais également sous forme de fonctionnalités : un ensemble de fonctionnalités fortement liées ensemble seront ainsi regroupées plutôt que d'être dispersées à travers toutes les classes d'un programme. [AND 92] évoque déjà ce besoin d'orthogonalité entre objets et fonctionnalités qui fait défaut aux modèles à classes et [SZY 92] montre le besoin de modules regroupant des ensembles de

classes. La motivation de ce genre d'approche peut être par exemple de définir plusieurs versions d'une même hiérarchie de classes, différentes suivant les applications visées [ERN 03].

Un second intérêt est de pouvoir faire évoluer un ensemble de classes (programme ou bibliothèque) *a posteriori* sans devoir modifier les classes existantes (les classes pouvant être partagées, donc devant continuer à exister sous leur ancienne forme ou le code source pouvant tout simplement être indisponible). Les langages réflexifs (CLOS [STE 90], SMALLTALK [GOL 83]) ou orientés aspects (ASPECTJ [KIC 01]) permettent déjà cela via un *meta-object protocol* (MOP) ou un tisseur. Notre approche diffère par la proposition d'une technique applicable aux langages à typage statique et compilés séparément et ne nécessitant pas l'utilisation d'un mécanisme externe comme un MOP ou un tisseur.

Les cinq amendements de classes que nous visons sont l'ajout et la redéfinition de méthodes et d'attributs, l'ajout de super-classes, l'unification de classes (exprimer que deux classes définies à deux endroits différents sont les mêmes, les instances de l'une étant aussi les instances de l'autre) et la généralisation de propriétés (remonter l'introduction d'une propriété dans une super-classe). Nous nous plaçons dans un cadre d'héritage multiple en étendant l'analyse de [DUC 95] au raffinement multiple.

2.2. *Idée intuitive du raffinement*

Intuitivement, on peut présenter le raffinement d'une classe c_1 par une classe c_2 comme une définition incrémentale des classes dans laquelle les propriétés définies dans c_2 se rajoutent ou remplacent celles de c_1 et les instances de c_1 et de c_2 se cumulent. Un tel procédé se rencontre plus ou moins fréquemment dans les langages à typage dynamique. C'était le cas en YAFOOL [DUC 91] où la définition incrémentale de classes était d'usage standard. C'est le cas, pour les méthodes au moins, dans les surcouches objets de LISP où les méthodes sont définies à l'extérieur des classes. En toute généralité, c'est possible dans tous les langages dotés d'un *meta-object protocol* [KIC 91] comme CLOS, même si des expériences montrent que ces protocoles ne sont pas toujours très adaptés à la modification des classes [PAV 99].

Le caractère intuitif du raffinement se perd un peu dès que l'on rencontre un raffinement multiple d'une même classe ou que l'on combine raffinement et spécialisation : l'ordre des raffinements et de la spécialisation sous-jacente n'est plus innocent. Si les langages dynamiques peuvent se reposer sur l'ordre chronologique, cela n'est plus possible dans le cadre des langages à typage statique que nous visons.

Aussi, pour donner un cadre plus structuré à notre proposition, nous y ajoutons une notion de *module*, dans son plus simple appareil, assez classique. Les modules sont dépendant entre eux et contiennent les définitions et raffinements de classes L'ordre des raffinements se déduit alors de l'ordre sur les modules. Il n'y a pas de notion de visibilité (ou exportation), ni d'espace de noms associés aux modules : ce n'est pas nécessaire pour l'instant.

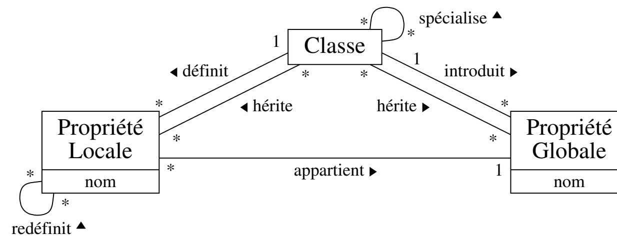


Figure 1. *Méta-modèle des propriétés*

3. Le méta-modèle objet

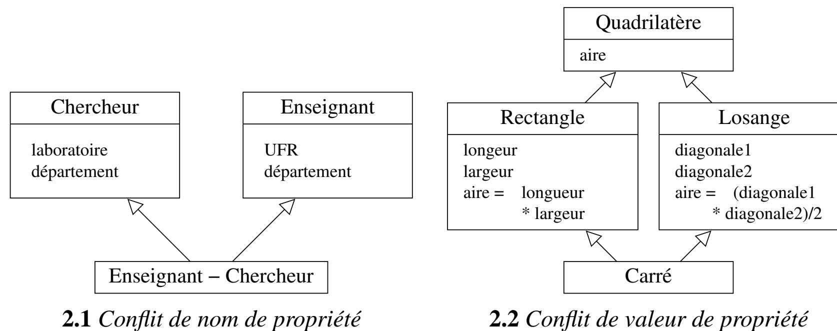
Le méta-modèle de départ est composé de trois entités principales : les classes, les propriétés locales et les propriétés globales (figure 1). Bien qu'il ne soit explicite dans aucun langage, ni même dans UML, nous pensons qu'il a vocation à l'universalité : c'est le méta-modèle implicite de JAVA et d'EIFFEL (à condition, dans ce dernier, de faire un usage limité du renommage).

3.1. *Entités et relations*

Pour méta-modéliser les propriétés, la liaison tardive (envoi de messages) impose de définir deux catégories d'entités. Les *propriétés locales* correspondent aux attributs et aux méthodes tels qu'ils sont définis dans une *classe*, indépendemment de toute éventuelle redéfinition. Les *propriétés globales*[1] correspondent aux messages auxquels les instances d'une classe peuvent répondre, cette réponse étant alors la mise en œuvre d'une propriété locale de la classe. Chaque propriété locale appartient à une unique propriété globale et dans chaque classe il y a une bijection entre ses propriétés locales (définies ou héritées) et ses propriétés globales. La correspondance entre ces deux entités se fait par leurs noms.

La relation de spécialisation permet aux sous-classes d'hériter les propriétés de leurs super-classes. Cet héritage s'opère à deux niveaux [DUC 95]. Au niveau des propriétés globales, appelé *héritage de nom*, toute classe possède les propriétés globale de ses super-classes. Au niveau des propriétés locales, appelé *héritage de valeur*, si une classe apporte sa propre propriété locale pour une propriété globale donnée, il s'agit de (re)définition ; sinon la classe hérite la propriété locale la plus spécifique définie dans ses super-classes, c'est-à-dire la propriété locale définie dans la super-classe la plus spécifique par la relation de spécialisation. Lorsqu'une classe définit une propriété locale dont le nom ne correspond à aucune propriété globale héritée,

1. Dans les travaux antérieurs, le terme utilisé était *propriété générique*, par analogie avec les *fonctions génériques* de CLOS. Le passage aux modules (section 4) nous a contraint à remplacer *générique* par *global*, le terme de « classe générique » ayant un autre sens.



2.1 *Conflit de nom de propriété* **2.2** *Conflit de valeur de propriété*

Figure 2. *Conflits en héritage multiple*

une nouvelle propriété globale est implicitement introduite dans la classe et liée à la propriété locale.

Enfin, lors d'un envoi de message `x.foo(args)`, `foo` désigne la propriété globale de nom `foo` du type statique du receveur `x`. Notons que le typage dynamique rend ce méta-modèle inopérant, et que la surcharge statique de C++, JAVA, C#, etc. impose, dans ces langages, d'ajouter au nom des méthodes le type des paramètres pour identifier les propriétés, aussi bien locales que globales.

3.2. *Héritage multiple*

En héritage multiple, la source de difficulté principale est l'existence de conflits. La définition qui en est donnée ici est équivalente à celle de [DUC 95].

3.2.1. *Héritage de nom*

Un *conflit de nom de propriété* survient lorsqu'une classe spécialise deux classes possédant des propriétés globales distinctes mais homonymes. L'exemple de la figure 2.1 montre deux classes (`Chercheur` et `Enseignant`) possédant toutes deux une propriété globale nommée `département`, l'une désignant un département dans un laboratoire de recherche, l'autre un département d'enseignement dans une université. Il est alors attendu que la sous-classe commune hérite des propriétés globales de ses super-classes, or le nom `département` ne peut en désigner qu'une seule.

Dans tous les cas, un conflit de nom n'est qu'un problème de vocabulaire et un renommage systématique garantirait l'absence de conflits de noms. Ce renommage peut être libre ou spécifié, local ou global.

Interdiction La spécification du langage proscrit les conflits de noms. Cette situation force le programmeur à renommer au moins l'une des deux propriétés globales dans tous les programmes qui l'utilisent ce qui peut impliquer la modification

d'un grand nombre de classes (avec les erreurs potentielles inhérentes) voire s'avérer impossible si ces classes ne sont pas modifiables (code source indisponible par exemple).

Désignation explicite Les noms de propriétés litigieux doivent être préfixés d'une classe où le nom n'est pas ambigu, par exemple la classe qui introduit la propriété globale. Dans la spécification de la classe `Enseignant-Chercheur` de l'exemple, `Enseignant.département` désignerait la propriété globale connue sous le nom `département` dans la classe `Enseignant`. Cette solution est utilisée dans C++ [STR 86] pour les attributs[2].

Renommage local Afin de garantir que, dans une classe, un nom ne désigne qu'une seule propriété globale, le renommage local permet de changer la désignation des propriétés globales en fonction de la classe considérée. Dans la classe problématique de l'exemple, nous pouvons renommer `département` hérité de `Enseignant` en `dépt-ens` et `département` hérité de `Chercheur` en `dépt-rech`. Ainsi `département` dans `Chercheur` et `dépt-rech` dans `Enseignant-Chercheur` désignent la même propriété globale et comme attendu, dans la classe `Enseignant-Chercheur`, `dépt-rech` et `dépt-ens` désignent deux propriétés globales distinctes. Cette solution est utilisée dans EIFFEL [MEY 92].

Unification Les langages dynamiques (CLOS...), JAVA via ses interfaces et C++ pour les fonctions considèrent que si deux propriétés globales sont homonymes (pour C++ et JAVA, il faut intégrer au nom le nombre et le type des paramètres) alors elles ne sont pas distinctes. Il n'y a donc pas de conflit de nom possible et les ambiguïtés de l'héritage multiple sont reportées sur l'héritage de valeur.

3.2.2. *Héritage de valeur*

Un *conflit de valeur de propriété* survient lorsqu'une classe hérite deux propriétés locales de la même propriété globale, aucune n'étant plus spécifique que l'autre. L'exemple de la figure 2.2 montre deux classes (`Rectangle` et `Losange`), toutes deux redéfinissant la méthode `aire` dont la propriété globale a été introduite dans la classe `Quadrilatère`. Dans la sous-classe commune `Carré`, laquelle est la plus spécifique ? Contrairement au conflit de nom, il n'y a pas de solution intrinsèque à ce problème (comme un renommage massif dans l'héritage de nom) et le programmeur ou le langage doit apporter une sémantique additionnelle pour s'affranchir de ce problème.

Interdiction La spécification du langage proscrit les conflits de valeur. Le programmeur doit redéfinir la propriété globale par une nouvelle propriété locale dans la classe où le conflit apparaît. Dans la redéfinition, une désignation explicite, comme en C++, peut permettre de choisir parmi les propriétés locales en conflit.

Combinaison Pour certaines valeurs ou propriétés particulières, la résolution du conflit doit se faire par combinaison des valeurs : c'est le cas, par exemple, pour le type en cas de redéfinition covariante (la borne inférieure des types en conflit,

2. Pour les méthodes, l'opérateur :: correspond à un appel statique : c'est donc une propriété locale qui est désignée et non la propriété globale.

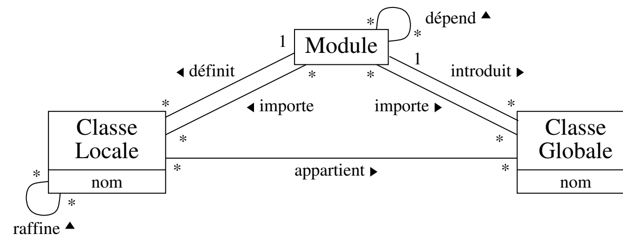


Figure 3. *Méta-modèle des classes et des modules*

si elle existe) ou pour des propriétés dont les valeurs se cumulent. On retrouve également la combinaison pour les contrats en EIFFEL avec la disjonction des pré-conditions et la conjonction des post-conditions.

Choix Le programmeur ou le langage choisit arbitrairement la propriété locale à hériter. Dans les langages dynamiques comme CLOS ou YAFOOL, une technique de linéarisation permet de la déterminer ; en EIFFEL, le programmeur peut sélectionner la propriété voulue à l'aide de la clause d'héritage `undefine`.

4. Classes et modules

Nous présentons maintenant plus formellement les modules et le raffinement, par une analogie très forte avec le méta-modèle objet présenté ci-dessus. Les problèmes liés aux conflits d'héritage et de raffinement multiple seront traités dans la section suivante.

4.1. *Méta-modèle des modules*

Une approche plus rigoureuse se base sur le fait que, dans une certaine mesure, les classes sont aux modules ce que les propriétés sont aux classes, la relation de dépendance entre modules étant analogue à la spécialisation, et l'importation à l'héritage. Il est ainsi nécessaire de définir deux entités associées au concept de classe (figure 3).

Les *classes locales* (analogues aux propriétés locales) sont définies dans les modules. Une classe locale est décrite par un nom, des noms de super-classes et des définitions de propriétés locales.

Les *classes globales* (analogues aux propriétés globales sont orthogonales aux modules. Chaque module possède des classes globales qui correspondent aux classes qu'il est statiquement capable de manipuler. Les classes globales regroupent des classes locales. La correspondance entre ces deux entités se fait par leur nom.

Au total, les classes locales se définissent dans un module exactement comme dans un langage à objets ordinaire, les noms de classes désignant aussi bien les classes locales que globales.

4.2. *Définitions et notations*

Un *module* est un ensemble de définitions de *classes locales*, qui *importe* (ou *dépend de*) zéro, un ou plusieurs modules. On note $m, n, \ldots$ les modules et $<$ la relation de dépendance entre modules : $m < n$ signifie que m dépend de n (m est un *sous-module* de n, qui est un *super-module* de m). La relation de dépendance est un ordre partiel strict.

Une *classe globale* est un ensemble de classes locales de même nom définies dans des modules différents. De façon analogue à l'héritage de nom de propriétés, un module importe toutes les classes globales de ses super-modules. Lorsque l'on définit une classe locale dans un module m, deux cas se présentent. Si une *classe globale* de même nom est importée de l'un des super-modules : il s'agit alors d'un *raffinement*. Sinon, une nouvelle classe globale de ce nom est *introduite* dans le module m. Dans les deux cas, la nouvelle classe locale est rajoutée à la classe globale.

On note A_m la classe locale de m appartenant à la classe globale A et on utilise par abus de notation l'appartenance ensembliste entre toutes ces entités : $A_m \in m$, $A_m \in A$, aussi bien que $A \in m$ si A est importée ou introduite dans m. Pour toute classe globale A d'un module m, on supposera l'existence de la classe locale correspondante A_m : il s'agit soit d'une définition explicite, soit d'un raffinement implicite, A_m étant alors appelée *classe implicite* puisque sa description est vide. En corollaire, raffiner explicitement une classe par une classe locale vide équivaut à ne pas la raffiner explicitement.

Lorsque deux modules sont en relation de dépendance, les classes locales de leurs classes globales communes sont en relation de raffinement, également noté $<$:

$$A_m < A_n \Leftrightarrow_{\text{def}} m < n \text{ et } A \in n \tag{1}$$

Dans un module m, la définition d'une classe locale C_m peut en faire explicitement une sous-classe d'une classe D : nous notons cette relation de *spécialisation explicite* $C_m \prec_m D_m$. Elle est licite si le module m introduit ou importe la classe globale D. La relation de *spécialisation* entre classes locales est un ordre partiel strict notée $\prec$ et se construit par l'importation des spécialisations explicites des super-modules. Il se définit par la fermeture transitive de la relation $\prec'$ suivante :

$$C_m \prec' D_m \Leftrightarrow_{\text{def}} \exists n \text{ tel que } m \leq n \text{ et } C_n \prec_n D_n \tag{2}$$

Signalons qu'une relation de spécialisation déclarée entre classes locales en relation de spécialisation par importation est sans effet.

Enfin, un *programme* est la donnée d'un ensemble de modules fermé par la relation d'importation : il correspond à un module (éventuellement virtuel) important tous les

modules dont il est composé. On appellera *classes locales d'un programme* l'union des classes locales de tous ses modules et *quasi-spécialisation* la fermeture transitive de l'union de la relation de raffinement $<$ et de la relation de spécialisation $\prec$ sur les classes locales du programme.

Dans les figures, nous adoptons la convention suivante : les classes locales figurent comme des petites boites nommées, à l'intérieur de grandes boites éventuellement numérotées, les modules. Seules les relations de spécialisation dans un module ($\prec$) et d'importation entre modules ($<$) sont explicitées. Les classes locales d'une même classe globale portent le même nom : la relation de raffinement entre classes ($<$) reste donc implicite. De plus, les classes et les spécialisation implicites sont figurées en pointillés.

Intuitivement, le programme correspondant au module m se comporte comme la hiérarchie de ses classes locales, munie de la quasi-spécialisation comme relation de spécialisation, et dans le code de laquelle tout nom de classe A, utilisé comme type ou pour l'instanciation, serait interprété comme la classe locale A_m. Toutes les autres classes locales des modules importés se comportent comme des classes abstraites. De façon alternative, on peut voir le programme comme la hiérarchie des classes locales du module m, ordonnée par la relation $\prec$, le contenu de chaque classe résultant de ses raffinements successifs dans les modules importés.

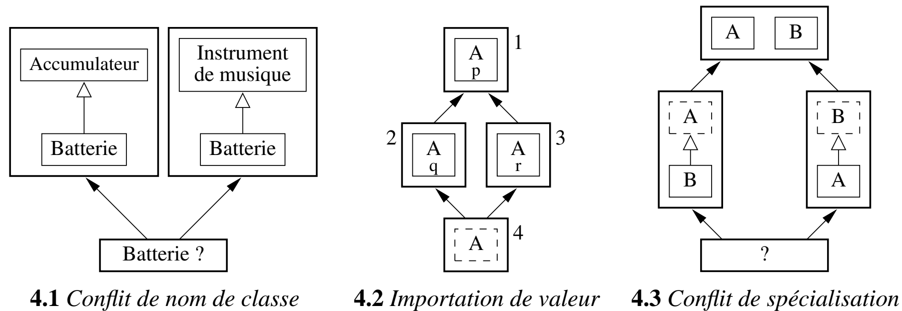
4.3. *Unification de classes*

Dans la définition d'une classe locale, le programmeur peut choisir d'unifier explicitement cette classe avec d'autres classes. Par exemple via une écriture syntaxique analogue à la déclaration de spécialisation.

L'unification de classes consiste à lier ensembles les classes globales : dans le module et dans tous les sous-modules, les deux classes globales partageront une unique classe locale. Nous notons $\widehat{AB}_m = A_m = B_m$ la classe locale associée à l'union dans le module m des classes globales A et B. Comme la classe locale $\widehat{AB}_m$ appartient à la fois à A et à B, elle raffine les classes A_n et B_n pour tout super-module n. De plus comme la relation de spécialisation est anti-symétrique, l'équation (2), devient :

$$C_m \prec' D_m \Leftrightarrow_{\text{def}} C_m \neq D_m \text{ et } \exists n \text{ tel que } m \leq n \text{ et } C_n \prec_n D_n \quad (3)$$

La question de la dénomination de telles classes locales peut se résoudre de deux manières : soit en considérant que les noms des classes globales sont des alias et peuvent être indépendemment utilisés pour désigner la classe locale, soit en choisissant un nom unique pour l'identifier.



4.1 *Conflit de nom de classe* **4.2** *Importation de valeur* **4.3** *Conflit de spécialisation*

Figure 4. *Importation multiple et conflits.*

4.4. *Cas des constructeurs*

Dans les langages statiques sans raffinement, les constructions (c'est-à-dire initialisations) d'instances ne sont pas soumises au polymorphisme : le type dynamique des instances qui seront créées est déjà déterminé par le type statique utilisé dans le constructeur. Ainsi au niveau des langages, soit les méthodes particulières qui ont un rôle de constructeur d'instance ne sont pas héritées (JAVA ou C++), soit c'est leur rôle de constructeur qui ne l'est pas (EIFFEL).

Avec le raffinement, les choses sont un peu différentes. Certes le type dynamique des instances qui seront créées est statiquement prévisible mais il demeure globalement inconnu, la classe locale que le module manipule statiquement pouvant être raffinée dans d'éventuels sous-modules. Ainsi, lors d'un raffinement de classes, les constructeurs doivent d'une part être pleinement héritables et d'autre part, les classes doivent s'assurer que les constructeurs introduits dans les classes qu'elles raffinent restent cohérents (en les redéfinissant si nécessaire).

5. Héritage et importation multiple

Il résulte des définitions de la section 4 que le raffinement d'une sous-classe (au sens de la spécialisation) induit automatiquement de l'héritage multiple (au sens de la quasi-spécialisation). Il est donc impossible de faire l'économie des problèmes posés par l'héritage multiple. De plus, la relation de dépendance entre module peut elle-même être multiple. Le traitement des conflits va être plus inhabituel, tout en restant pour l'essentiel dans le schéma de [DUC 95].

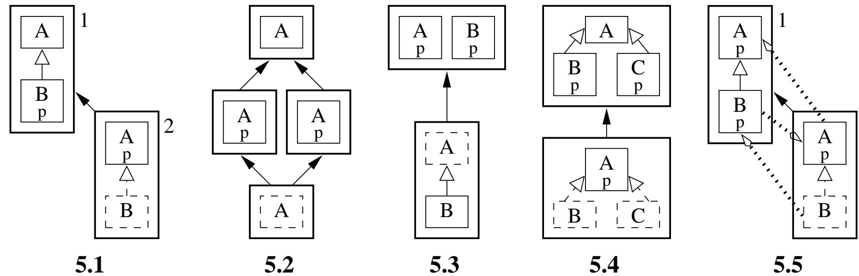


Figure 5. *Propriétés et héritage.*

5.1. *Importation multiple*

Un *conflit de nom de classe* survient lorsqu'un module importe deux classes globales de même nom de deux super-modules différents (figure 4.1). Analogues aux conflits de noms de propriétés, ils peuvent être résolus de la même manière : les modules étant généralement des espaces de noms, la désignation explicite est ici la solution la plus naturelle. Toutefois, un mécanisme de renommage des classes à la EIFFEL et son langage de configuration LACE pourrait également résoudre le problème.

La figure 4.2 illustre la configuration analogue au conflit de valeur : le module 4 importe A introduite par le module 1 et raffine implicitement A_2 et A_3 par une classe locale A_4. Dans ce cas, le conflit de valeur va être traité par combinaison des classes locales en conflit, par héritage de leurs propriétés au travers de la relation de spécialisation et de raffinement.

Une configuration conflictuelle nouvelle, le *conflit de spécialisation*, apparaît lorsque dans un module, deux classes locales se retrouvent spécialisation l'une de l'autre (figure 4.3), ce qui provoque un circuit dans la relation de spécialisation qui n'est plus un ordre partiel. Ce conflit peut être résolu en unifiant les classes du circuit.

5.2. *Héritage de nom de propriété*

Une classe locale possède les propriétés globales de ses super-classes locales au sens de la quasi-spécialisation. Dans la figure 4.2, A_2 hérite la propriété globale `p` introduite dans A_1 et A_4 hérite des propriétés globales `p`, `q` et `r` introduites dans les classes qu'elle raffine.

Toutefois, comme le programmeur connaît les modules qu'il importe, certains conflits de nom apparents peuvent se résoudre par l'identité des propriétés globales. L'ajout de propriétés locales aux propriétés globales n'a alors plus de raison de se limiter à la redéfinition dans des sous-classes. La *généralisation de propriété globale* permet d'étendre la possession de propriétés globales des classes aux super-classes

au sens de la spécialisation. La généralisation d'une propriété globale possédée par une classe locale C_m dans un module n, sous-module de m consiste à définir dans n une propriété locale homonyme à la propriété globale dans une classe D_n telle que $C_n \prec D_n$. L'exemple de la figure 5.1 montre la définition d'une propriété `p` dans une classe locale B_1 et sa généralisation dans la classe A_2 (puisque $B_2 \prec A_2$).

Les conflits de noms de propriétés liés à la spécialisation ou au raffinement sont toujours possibles mais doivent être résolus, au cas par cas, en utilisant les techniques de la section 3.2.1. En plus du conflit de nom lié à la spécialisation multiple (figure 2.1), la figure 5.2 montre un conflit lié au raffinement multiple et les figures 5.3 et 5.4 montrent des conflits impliquant raffinement et spécialisation.

5.3. *Héritage de valeur de propriété*

Soit la propriété globale d'une classe locale est redéfinie, soit la propriété locale définie dans la super-classe la plus spécifique est héritée. Toutefois, il faut combiner les deux ordres sur les classes locales que sont la spécialisation et le raffinement pour produire un ordre partiel strict unique de *spécificité* dans le but de déterminer les propriétés locales héritées. Cette relation de spécificité diffère légèrement de celle de quasi-spécialisation.

La relation de spécificité pour le module s entre les classes locales C_m et D_n (avec $s < m$ et $s < n$) se note $C_m \ll_s D_n$ et se définit par :

$$C_m \ll_s D_n \Leftrightarrow_{\text{def}} C_s \prec D_s \text{ ou } (C_s = D_s \text{ et } m < n) \quad (4)$$

La première partie exprime que, dans le module, la relation de spécialisation entre deux classes locales se répercute sur tout couple de classes des mêmes classes globales. La seconde partie de la définition exprime qu'une classe raffinée est moins spécifique que son raffinement.

L'exemple de la figure 5.5, où la relation de spécificité est représentée par les flèches hachurés à tête blanche, pose le problème de l'héritage de valeur de la propriété globale `p` dans la classe locale B_2. L'équation (4) permet de dire que $B_2 \ll_2 B_1 \ll_2 A_2 \ll_2 A_1$. On en déduit que la propriété locale `p` héritée est celle définie dans B_1.

La relation de spécificité contient strictement la relation de quasi-spécialisation. En effet, cette dernière est insuffisante car en considérant le cas de la figure 5.5, les classes B_1 et A_2 seraient incomparables. Or la vision intuitive qui considère le raffinement comme une modification incrémentale des classes attribuerait à B_2 la méthode p définie dans B_1 comme c'est le cas avec la relation de spécificité $\ll$. Informellement, cela signifie que la relation de raffinement est *plus forte* que celle de spécialisation. Toutefois, lors de l'unification de classes, les relations de spécialisation qui pouvaient exister dans d'éventuels super-modules n'ont plus d'incidence sur la relation de spécificité, seule la relation de raffinement garde une utilité dans la spécificité.

Lors de la redéfinition ou de l'héritage d'une propriété locale l dans une classe C_m, il est nécessaire de vérifier que celle-ci est conforme non seulement avec les propriétés

locales définies dans les super-classes de C_m (au sens de la quasi-spécialisation) mais aussi que les propriétés locales définies ou héritées dans les sous-classes de C_m (au sens de la spécialisation) sont conformes avec l. Dans l'exemple de la figure 5.1, il faut vérifier que la propriété locale p héritée dans B_2 (c'est-à-dire celle définie dans B_1) est bien conforme à celle définie dans A_2. La conformité entre propriétés locales peut prendre plusieurs formes suivant les langages (arité, type de retour, types des paramètres, exceptions déclarées, contrats, etc.)

Notons que, dans un cadre de typage sûr, la règle de contravariance s'applique au raffinement. De plus, comme la sémantique du raffinement n'est pas une sémantique de spécialisation, une politique de raffinement covariant [DUC 02] ne s'applique pas : au contraire, une contravariance stricte pourrait trouver ici son utilité.

5.4. *Construction de programmes*

Lors de la construction d'un programme, des conflits liés à l'importation multiple de modules peuvent apparaître. Si nécessaire, ceux-ci peuvent être levés en construisant un sous-module dédié à la résolution des conflits. Toutefois, l'éditeur de liens ayant la connaissance globale du programme, seuls les conflits ayant un impact réel sur le programme ont à être signalés. Par exemple les conflits de noms (de classes ou de propriétés) qui peuvent apparaître dans un module virtuel n'ont pas d'influence sur le comportement du programme, à moins que le langage permette l'introspection.

6. Implémentation

L'approche proposée ici, qui considère des modules indépendants combinés entre eux pour former un programme, est compatible avec les schémas de compilation séparée du moment qu'ils permettent de ne prendre connaissance du méta-modèle complet qu'à l'édition de liens. C'est le cas du schéma de compilation proposé dans [PRI 04].

Après l'édition de liens, les différentes classes locales d'une classe globale ont disparu : seules restent les classes locales du module virtuel qu'est le programme. Le fait de retrouver des classes sans modules et sans raffinement permet de prévoir que l'implémentation peut se réaliser sans surcoût spatial ou temporel à l'exécution par rapport au même langage sans raffinement.

Toutefois, cette approche est difficilement compatible avec le chargement dynamique, c'est-à-dire le raffinement de classes pendant l'exécution d'un programme. En effet, la difficulté de l'implémentation de l'héritage multiple en chargement dynamique est ici exacerbée : bien qu'il soit envisageable de modifier pendant l'exécution les valeurs des pointeurs dans les tables de méthodes (redéfinition de méthodes), il est bien plus coûteux de changer profondément ces tables pour refléter l'ajout de méthodes ou de liens de spécialisation. Le cas extrême peut être la nécessité de modifier les instances pour refléter l'ajout d'attributs, ce qui impose des techniques d'implémentation comparables à la migration d'instance.

7. Autres travaux

MULTIJAVA [CLI 00] propose des unités de compilation, analogues aux modules présentés ici, qui sont munis d'une relation de dépendance via le mot-clé `import`. Il permet d'étendre des classes existantes en ajoutant des fonctions aux classes par une syntaxe *ad hoc*. Par contre la redéfinition de méthodes, l'ajout d'attributs ou la déclaration d'implémentation d'interfaces JAVA ne sont pas permis. Néanmoins, MULTIJAVA est compatible avec la compilation séparée et le chargement dynamique. Il propose également une mise en œuvre des multi-méthodes.

Les *classboxes* [BER 03] permettent d'étendre les classes en SMALLTALK par des ajouts ou des redéfinitions de méthodes et d'attributs tout en contrôlant la visibilité des ajouts puisque ces amendements n'ont que des impacts locaux, les réponses aux envois de messages étant déterminés à la fois par le receveur et par la *classbox*. Ainsi, contrairement à la contribution de cet article, les amendements de classes apportés par une *classbox* ne sont appliqués qu'à cette *classbox* et aux *classboxes* qui l'importent, les envois de messages provenant d'autres *classboxes* ne seront donc pas affectés par la modification.

MIXJUICE [ICH 02], basé sur le langage JAVA, propose des modules en relation de dépendance et des amendements de classes autorisant la (re)définition de méthode, l'ajout d'attributs et la déclaration supplémentaire d'implémentation d'interfaces JAVA. En cas de dépendance multiple entre modules, les conflits de noms de propriétés sont résolus par désignation explicite, les conflits de valeur sont résolus par une linéarisation à la CLOS. L'approche est compatible avec la compilation séparée mais ne permet pas le chargement dynamique. MIXJUICE ne fait pas apparaître l'analogie entre les classes et les modules ni n'analyse les différentes configurations conflictuelles et leurs résolutions. C'est avant tout une implémentation fonctionnelle du mécanisme de raffinement (dans les limites imposée par JAVA d'invariance des types et d'héritage simple), qui permet de montrer la faisabilité d'une telle approche.

Les « hiérarchies d'ordre supérieur » de [ERN 03] ressemblent beaucoup à nos modules. Les motivations sont identiques et l'isomorphisme des méta-modèles des classes (ordre 1) et des modules (ordre 2) permettrait une généralisation immédiate à un ordre quelconque. Ce n'est cependant pas notre objectif et nous ne poussons pas la métaphore des classes aussi loin : finalement, comme le dit [SZY 92], nos modules ne sont pas des classes, au moins en ce sens qu'elles n'ont pas d'instances. Un module est de la dimension d'un programme : on comprend bien qu'il puisse avoir une unique instance, correspondant aux données statiques d'une exécution. Mais si l'on poursuit la métaphore, une classe d'ordre 3 serait un système d'exploitation, et la proposition des hiérarchies d'ordre supérieur devient un peu mystérieuse. Enfin, techniquement, les deux approches diffèrent sensiblement sur le traitement de l'héritage multiple, dont nous avons vu qu'il était inévitable. [ERN 03] propose une combinaison de *mixins* totalement ordonnés alors que notre proposition repose sur l'interprétation de l'héritage multiple dans le méta-modèle, qui nous semble, depuis [DUC 95], la seule bonne façon d'aborder l'héritage multiple.

8. Conclusion et perspectives

Nous avons proposé dans cet article une extension du modèle à classe des langages en héritage multiple et en typage statique qui ajoute une notion de module et un mécanisme de raffinement entre classes qui permet dans un module d'amender les classes importées des modules dont il dépend. Le surcoût syntaxique est faible puisqu'il ne nécessite qu'un langage de module rudimentaire (exprimer qu'un module dépend d'un autre module et qu'une (re)définition de classe appartient à un module).

Grâce aux modules, cette proposition permet une meilleure structuration des programmes (fonctionnelle ou temporelle) et, grâce au raffinement, autorise plus de souplesse en facilitant entre autre l'évolution des logiciels et le partage de hiérarchies entre applications. Une attention particulière est mise sur la gestion cohérente des conflits visant à respecter le principe de spécialisation entre classes. De plus, cette proposition est pleinement compatible avec la compilation séparée et sans surcoût temporel ou spatial pour des programmes sans chargement dynamique de classes ou de modules.

Bien que les modules et les classes soient d'utilité fondamentalement différente [SZY 92], notre proposition est basée sur une analogie de structure stricte entre ces deux notions puisqu'elles sont décrites par des méta-modèles semblables[3].

Afin d'expérimenter l'utilisation du raffinement sur le développement de logiciel, un prototype de langage de programmation et son compilateur sont en train d'être développés [PRI 04]. L'objectif est à la fois de tester la mise en œuvre du raffinement et son implémentation en compilation séparée.

Toutefois, la limitation principale vient de ce qu'il n'est pas possible *a posteriori* de revenir sur des choix déjà effectués comme par exemple la suppression de propriétés et de relation de spécialisation ou l'unification de propriétés globales. Si la suppression n'est pas forcément utile, l'unification de propriétés serait une réponse à certains conflits, à condition d'arriver à la spécifier proprement.

Remerciements

Les auteurs remercient le relecteur anonyme qui leur a rappelé les travaux d'Erik Ernst.

9. Bibliographie

[AND 92] ANDERSEN E. P., REENSKAUG T., « System Design by Composing Structures of Interacting Objects », MADSEN O. L., Ed., *Proc. ECOOP'92*, LNCS 615, Springer-Verlag, 1992, p. 133–152.

3. « Import is inheritance, why we need both. » pour paraphraser et partiellement contredire [SZY 92].

[BER 03] BERGEL A., DUCASSE S., WUYTS R., « Classboxes : A Minimal Module Model Supporting Local Rebinding », *JMLC 2003 (Joint Modular Languages Conference)*, vol. 2789, 2003, p. 122–131.

[CLI 00] CLIFTON C., LEAVENS G. T., CHAMBERS C., MILLSTEIN T., « MultiJava : Modular open classes and symetric multiple dispatch for Java », *Proc. OOPSLA'00*, SIGPLAN Notices, 35(10), ACM Press, 2000, p. 130–145.

[DUC 91] DUCOURNAU R., « Yet Another Frame-based Object-Oriented Language : YAFOOL Reference Manual », Sema Group, Montrouge, France, 1991.

[DUC 95] DUCOURNAU R., HABIB M., HUCHARD M., MUGNIER M.-L., NAPOLI A., « Le point sur l'héritage multiple », *Technique et Science Informatiques*, vol. 14, n° 3, 1995, p. 309–345, Hermès.

[DUC 02] DUCOURNAU R., « Spécialisation et sous-typage : thème et variations », *Revue des Sciences et Technologies de l'Information, TSI*, vol. 21, n° 10, 2002, p. 1305–1342, Hermès.

[ERN 03] ERNST E., « Higher-Order Hierarchies », CARDELLI L., Ed., *Proc. ECOOP'2003*, LNCS 2743, Springer-Verlag, 2003, p. 303–329.

[GOL 83] GOLDBERG A., ROBSON D., *Smalltalk-80, the Language and its Implementation*, Addison-Wesley, Reading (MA), USA, 1983.

[ICH 02] ICHISUGI Y., TANAKA A., « Difference-Based Modules : A Class-Independant Module Mechanism », *Proc. ECOOP'2002*, LNCS 2374, Springer-Verlag, 2002, p. 62–88.

[KIC 91] KICZALES G., DES RIVIERES J., BOBROW D., *The Art of the Meta-Object Protocol*, MIT Press, Cambridge (MA), USA, 1991.

[KIC 01] KICZALES G., HILSDALE E., HUGUNIN J., KERSTEN M., PALM J., GRISWOLD W. G., « An overview of AspectJ », KNUDSEN J. L., Ed., *Proc. ECOOP'2001*, LNCS 2072, Springer-Verlag, 2001, p. 327–355.

[MEY 88] MEYER B., *Object-Oriented Software Construction*, Prentice Hall International Series in Computer Science, C.A.R. Hoare Series Editor, Prentice Hall International, Hemel Hempstead, UK, 1988.

[MEY 92] MEYER B., *Eiffel : The Language*, Prentice Hall Object-Oriented Series, Prentice Hall International, Hemel Hempstead, UK, 1992.

[PAV 99] PAVILLET G., DUCOURNAU R., « Implémentation des attributs booléens par un Meta Object Protocol », MALENFANT J., ROUSSEAU R., Eds., *Actes LMO'99*, Hermès, 1999, p. 55–68.

[PRI 04] PRIVAT J., DUCOURNAU R., « Intégration d'optimisations globales en compilation séparée des langages à objets », CARRÉ B., EUZENAT J., Eds., *Actes LMO'04* in *L'Objet vol. 10*, Hermès, 2004, p. 61–74.

[STE 90] STEELE G., *Common Lisp : The Language, Second Edition*, Digital Press, Bedford (MA), USA, 1990.

[STR 86] STROUSTRUP B., *The C++ Programming Language*, Addison-Wesley, Reading (MA), USA, 1986.

[SZY 92] SZYPERSKI C. A., « Import is not inheritance — why we need both : Modules and classes », MADSEN O. L., Ed., *Proc. ECOOP'92*, LNCS 615, Springer-Verlag, 1992, p. 19–32.