

Master mention IMS  
Spécialités Informatique Recherche et Professionnelle

UMINM 202	Langages à Objets à Typage Statique
UMINM 208	Méta-programmation et réflexivité

Programmation Par Objets  
Les concepts fondamentaux

Roland Ducournau  
Marianne Huchard

—  
Département d'Informatique

—  
UFR des Sciences  
Université Montpellier II

20 janvier 2005

# Table des matières

<b>1</b>	<b>La programmation par objets</b>	<b>3</b>
1.1	Caractérisation . . . . .	3
1.2	Problématiques . . . . .	4
1.3	Motivations des objets et de leur succès . . . . .	5
1.4	Classification des langages de programmation par objets . . . . .	5
1.5	Bibliographie commentée . . . . .	5
1.6	Plan . . . . .	6
<b>2</b>	<b>Classes, spécialisation et héritage</b>	<b>8</b>
2.1	Classes, instances et propriétés . . . . .	8
2.2	Sémantique de la spécialisation . . . . .	8
2.3	Spécialisation multiple . . . . .	10
2.4	Mono- et multi-instanciation . . . . .	10
<b>3</b>	<b>Typage et sélection de méthodes</b>	<b>11</b>
3.1	Typage et sous-typage . . . . .	11
3.2	Sous-typage et redéfinition. . . . .	13
3.3	Types paramétrés et généricité . . . . .	14
3.4	Variations sur C++ et JAVA . . . . .	16
3.5	Coercition . . . . .	16
3.6	Contradiction entre typage et sémantique de spécialisation . . . . .	18
3.7	Variance sur les exceptions ou les droits d'accès . . . . .	18
3.8	Sélection multiple et surcharge statique . . . . .	19
3.9	Problèmes divers . . . . .	23
3.10	Méta-modèle pour les propriétés . . . . .	23
<b>4</b>	<b>Héritage multiple</b>	<b>25</b>
4.1	La redéfinition et le masquage . . . . .	25
4.2	Héritage multiple et conflits de noms . . . . .	25
4.3	Héritage multiple et conflit de valeurs . . . . .	27
4.4	Les solutions proposées par les différents langages . . . . .	27
4.5	Techniques de linéarisation . . . . .	29
4.6	De l'héritage multiple à l'héritage simple . . . . .	31
4.7	Méta-modèle pour les propriétés . . . . .	32
<b>5</b>	<b>Réflexivité</b>	<b>33</b>
5.1	Introduction . . . . .	33
5.2	La réflexivité dans les langages interprétés . . . . .	36
5.3	La méta-programmation dans les langages compilés . . . . .	39
<b>6</b>	<b>Méta-modèle pour les propriétés</b>	<b>42</b>
6.1	Le méta-modèle, version minimale . . . . .	42
6.2	Influence du typage . . . . .	44
6.3	Héritage multiple . . . . .	46

<b>7</b>	<b>Implémentation</b>	<b>47</b>
7.1	En héritage et sous-typage simples . . . . .	47
7.2	En héritage multiple . . . . .	48
7.3	En héritage simple et sous-typage multiple . . . . .	51
<b>8</b>	<b>Bref manuel CLOS</b>	<b>53</b>
8.1	Définition de classes et méthodes . . . . .	53
8.2	Accès aux attributs . . . . .	55
8.3	Fonctions utiles . . . . .	55
8.4	Lancement . . . . .	56
<b>9</b>	<b>Exercices et problèmes</b>	<b>57</b>
9.1	Les polygones . . . . .	57
9.2	Classes qui mémorisent leurs instances . . . . .	58
9.3	Inspecteur d'objets . . . . .	59
9.4	Types paramétrés en CLOS . . . . .	59
9.5	Variables de classes, en C++ ou JAVA . . . . .	60
9.6	Modélisation d'animaux et d'espèces animales . . . . .	60
9.7	Simulation de la redéfinition covariante avec la surcharge statique . . . . .	61
<b>10</b>	<b>Glossaire</b>	<b>62</b>

# Chapitre 1

## La programmation par objets

### 1.1 Caractérisation

La programmation par objets présente, dans son modèle standard, celui des *langages de classes*, quatre caractéristiques originales.

**L'objet = données + procédures.** Alors que les langages de programmation traditionnels distinguent données et procédures, les secondes agissant sur les premières, les langages de programmation par objets mettent en avant la notion d'*objet*, qui associe étroitement des données et des procédures (appelées *méthodes*), les unes et les autres étant propres à l'objet. Dans une approche orthodoxe des objets, les méthodes constituent la seule interface de l'objet avec le monde extérieur, les données de l'objet ne pouvant être accédées qu'au travers de ses méthodes : c'est l'*encapsulation*.

Enfin, cette définition de l'objet comme un tout relativement *autonome* conduit à prendre en compte quelque-chose qui relève de l'*identité* de l'objet, qui se distingue d'une *valeur* : un objet ne se réduit pas à une quelconque suite de 0 et de 1 : c'est celle-ci et non celle-là.

**L'envoi de message,** qui remplace l'appel de procédure, est nécessité par le fait que plusieurs objets peuvent avoir des procédures de même nom, ce que l'on appelle, improprement, le *polymorphisme*. Les procédures de l'objet sont activées par un « envoi de message », le nom, ou sélecteur, du message étant le nom de la procédure de l'objet. Une vision anthropomorphe des objets (agents) conduit à considérer que c'est l'objet *receveur du message* qui décide de son comportement, suivant le message qu'il reçoit.

On peut remplacer cette vision très « Intelligence Artificielle », par un point de vue plus classique, celui de la compilation par exemple. On parlera alors de *liaison tardive* (*late binding* en anglais), caractérisée par le fait que le compilateur ne peut pas déterminer statiquement la procédure à activer : il faut attendre l'exécution pour connaître l'objet receveur et la procédure exacte (l'adresse à laquelle se brancher).

Dans tous les cas, le *receveur* est une notion primordiale, soulignée dans les langages par une pseudo-variable spécifique pour le désigner : *self* en SMALLTALK, *this* en C++ et JAVA ou *current* en EIFFEL. L'*encapsulation* idéale à la SMALLTALK ou EIFFEL réserve au receveur le droit de certaines actions sur lui-même (par ex. accès aux attributs).

Ces deux premières caractéristiques sont étroitement corrélées : l'une ne va pas sans l'autre. Elles constituent la base d'une famille de langages, dits de *prototypes* ou d'*objets sans classes*. Ils présentent un inconvénient majeur vis-à-vis des grands principes du génie logiciel (modularité, réutilisabilité, extensibilité), leur manque de structuration. Aussi les deux caractéristiques suivantes introduisent des abstractions fortement structurantes, qui sont responsables du succès des objets en génie logiciel.

**La classe** regroupe des objets similaires, ses *instances*, et factorise leurs propriétés communes. Une fois que l'on a de nombreux objets, dotés de leurs données et procédures propres, il est vite nécessaire de chercher à factoriser ce qu'ils ont de commun. Cela peut être fait de différentes façons, la plus commune étant la classe, qui regroupe tous les objets de même *structure* (données) et de même *comportement* (procédures). Dans ce modèle, que l'on appelle *langages de classes*, les procédures sont regroupées dans la classe, les objets n'ayant plus que leurs données en propre. L'encapsulation de l'objet et l'abstraction de la classe en font une implémentation des *types abstraits de données*.

**La spécialisation de classes et l'héritage de propriétés.** L'opération d'abstraction qui a consisté à passer des objets aux classes se répète ici, sous une autre forme, pour passer de plusieurs classes à une nouvelle classe qui factorise leurs propriétés communes. Les classes sont alors structurées en une hiérarchie de spécialisation-généralisation et les propriétés des classes les plus générales sont héritées par les classes les plus spécialisées.

En pratique, cela s'utilise en sens inverse, une nouvelle classe étant définie comme *spécialisation* d'une ou plusieurs classes préexistantes. On parlera alors de *sous-classes* et *super-classes*, directes et indirectes. La sous-classe *hérite* des propriétés de ses super-classes, qu'elle peut étendre et redéfinir.

**Langage à objets = objet + message + classe + héritage.** Ces 4 caractéristiques sont essentielles pour constituer un « langage à objets », dans l'acception la plus classique du terme. Certains langages ne satisfont à ces critères que de justesse — ainsi C++ et son mot-clef `virtual` — ou marginalement (ADA95).

## 1.2 Problématiques

L'approche objet offre plusieurs problématiques importantes. Dans chacune, les divers langages présentent des comportements assez variés : certains enfreignent allègrement les contraintes théoriques, d'autres augmentent ces contraintes de façon artificielle.

### 1.2.1 Problématiques abordées ici

**Le typage.** Dans les langages compilés et typés statiquement (C++, JAVA, EIFFEL, etc.) la notion de *type* se confond généralement avec celle de classe. Cependant, le *sous-typage* pose des problèmes délicats et contraint assez strictement la possibilité de redéfinition des propriétés dans une sous-classe. Les *types paramétrés* posent eux aussi divers problèmes.

**L'héritage** présente des problèmes variés, de sémantique et dans sa relation avec le typage. Lorsqu'il est *multiple*, il peut présenter des conflits ou des ambiguïtés, qui sont en général traités assez différemment suivant que le langage est interprété ou compilé. Lorsque l'héritage est simple, le sous-typage peut être multiple : c'est le cas de JAVA et de C# [Mic01], le langage de la plate-forme DOTNET de Microsoft.

**La méta-modélisation et la réflexivité.** Dans les langages interprétés, à typage dynamique, comme SMALL-TALK ou CLOS, les classes doivent exister à l'exécution et il est naturel de procéder à un double mouvement, d'abstraction en regroupant les classes en méta-classe, et de réification en faisant des classes des objets « de première classe ». On aura donc des langages avec un méta-niveau, permettant de manipuler les classes. Ces langages seront appelés *réflexifs* lorsque ce méta-niveau sera plongé dans le niveau inférieur, faisant des classes des objets (presque) comme les autres.

Dans les langages compilés, la classe disparaît en général à l'exécution. Certains langages ou systèmes proposent néanmoins des niveaux méta concernés par la compilation — OPENC++, OPENJAVA ou JAVASSIST— voire une simple fonctionnalité d'*introspection* comme le `package reflect` de JAVA.

Dans tous les cas, on parle de « protocole de méta-objets » (*Meta-Object Protocol* ou MOP).

De façon plus générale, la méta-modélisation est une technique très utile pour spécifier le modèle objet lui-même : on l'utilisera par exemple pour analyser la problématique de l'*héritage multiple*.

**L'appel de méthode** tel que présenté plus haut avec la métaphore de l'envoi de message peut se compliquer de deux façons orthogonales. La *surcharge statique* permet d'utiliser le même nom pour des méthodes qui se distinguent par leurs signatures, essentiellement par le nombre et le type des paramètres. Dans ce cas, la sélection entre les diverses méthodes surchargées se fait statiquement, suivant le nombre et le type des arguments.

En revanche, certains langages comme CLOS, CECIL ou CLAIRE proposent un mécanisme de liaison tardive généralisée, où la sélection se fait, dynamiquement, suivant le type de tous les arguments. La métaphore de l'envoi de message devient alors inappropriée et l'on parle dans ces langages de *fonctions génériques*.

### 1.2.2 Autres problématiques

Parmi les nombreuses problématiques qui sont plus ou moins au cœur de l'approche objet et qui ne seront pas abordées ici, par manque de place et de temps, citons :

- la question de l'*identité* des objets qui est au centre de la problématique des bases de données ; dans un langage de programmation, cette identité peut se réduire à une adresse : mais elle exclut l'usage d'objets connus directement par leur « contenu » comme en C++ (cf. paragraphe 3.5).
- la question de la protection ou de l'encapsulation des objets, qui joue sur le fait que certaines propriétés peuvent être déclarées plus ou moins `public` ou `private`, avec des sémantiques assez variables.
- de façon générale, les problématiques de protection et d'envoi de message ont tendance à se mélanger, dans de nombreux langages, dont C++, à des problématiques plus classiques de portée des noms.

### 1.3 Motivations des objets et de leur succès

L'approche objet a rencontré un grand succès que l'on peut expliquer par deux catégories de raisons.

**Fiabilité et réutilisabilité.** Le génie logiciel aura tendance à privilégier les propriétés favorisant la fiabilité et la réutilisabilité : l'objet et la classe favorisent la modularité, l'envoi de message la sûreté par encapsulation, et l'héritage est un facteur clé de partage du code, donc de réutilisabilité.

**Représentation.** Les méthodes d'analyse et de conception en génie logiciel, ou la représentation des connaissances en intelligence artificielle privilégieront en revanche l'adéquation du modèle objet à la représentation des objets du monde.

Dans cette optique, le modèle objet procure un modèle de classification très proche de la classification utilisée pour les espèces naturelles.

**Contradictions.** On verra, au paragraphe 3.6, que ces deux motivations conduisent à une contradiction un peu insurmontable : en effet, la fiabilité impose une sûreté du typage dont la théorie impose malheureusement qu'elle soit incompatible avec la sémantique de la spécialisation.

### 1.4 Classification des langages de programmation par objets

On peut classer les langages de programmation par objets, suivant différents critères, par exemple :

- les langages interprétés et les langages compilés ;
- les langages à typage statique ou dynamique ;
- les langages sans méta-niveau, avec méta-niveau ou réflexifs ;
- les langages avec héritage simple ou multiple ;
- les langages avec sélection simple ou multiple pour l'appel de méthode.

Les premiers critères se résument dans la figure 1.1. La figure 1.2 résume les différentes propriétés des langages.

Ces oppositions binaires sont en général à pondérer. La frontière est souvent beaucoup moins nette. Ainsi, entre la compilation et l'interprétation, la compilation dans une machine virtuelle (*bytecode*), comme JAVA, est un intermédiaire. Un langage apparemment interprété comme SMALLTALK peut très bien être parfaitement compilé. Le typage peut être statique sans annotations explicites de type : c'est le cas des langages de la famille ML où les types statiques sont inférés. L'héritage simple des classes peut être complété par une héritage (ou plutôt sous-typage) multiple des interfaces, comme en JAVA. Enfin, la surcharge statique de C++ et JAVA peut déguiser leur sélection simple en une fausse sélection multiple.

### 1.5 Bibliographie commentée

#### Bibliographie générale

[MNC<sup>+</sup>89] et [DEMN98] sont des ouvrages de synthèse sur l'approche objet. [GMP96] est, comme son nom l'indique, un cours de programmation par objets, prenant comme support les langages C++ et EIFFEL.

#### Bibliothèque en ligne

Le livre de B. Meyer OOSC\_2 [Mey97] est disponible en intranet sur le campus de l'UM2 (file : /usr/doc/OOSC\_2) aussi bien qu'au LIRMM ([http://www.lirmm.fr/~ducour/Doc-objets/OOSC\\_2](http://www.lirmm.fr/~ducour/Doc-objets/OOSC_2)). C'est un très bon ouvrage d'introduction à l'approche objet et de méthodologie de programmation en EIFFEL.

	langages interprétés typage dynamique	langages compilés typage statique
langage sans méta-niveau		C++, JAVA EIFFEL
langage avec méta-niveau		OPENC++ OPENJAVA JAVASSIST
langage réflexif	SMALLTALK CLOS	

FIG. 1.1 – Classification des langages à objets

langage	exécution	typage	sous-typage	héritage	méta-niveau	réflexivité	sélection	généricité
C++	comp.	statique	multiple	multiple	OPENC++	non	simple	<i>templates</i>
JAVA	comp./mv	statique	multiple	simple	OPENJAVA	non	simple	PIZZA, GENERIC JAVA
C#	comp./mv	statique	multiple	simple	?	non	simple	?
EIFFEL	comp.	statique	multiple	multiple	?	non	simple	oui
SMALLTALK	interp/comp	dynamique	—	simple	oui	oui	simple	—
CLOS	interp/comp	dynamique	—	multiple	oui	oui	multiple	—
OBJECTIVE-CAML	interp/comp	infér. type	multiple	simple	?	?	simple	?

FIG. 1.2 – Caractéristiques des différents langages

Le manuel de référence de COMMON LISP et CLOS est lui aussi disponible en ligne sur le réseau du campus (file : /usr/doc/cltl/clm/clm.html).

### Bibliographie par langage

La littérature sur les langages décrits ou cités dans le cours est résumée dans le tableau de la figure 1.3, avec les sites Web où des logiciels ou de la documentation sont disponibles.

Enfin, le site <http://www.dmoz.org/Computers/Programming/Languages/> est un point d'entrée sur tous les langages de programmation du monde! A plus petite échelle, le site <http://www.lirmm.fr/~ducour/Doc-objets/> propose des pointeurs sur des documents concernant la plupart des langages abordés dans ce cours.

C++ est un langage à utiliser avec des pincettes car sa compatibilité avec la philosophie objet est douteuse : s'il faut vraiment s'en servir, l'étude de son implémentation est indispensable [ES90, Lip96].

Enfin, ceux qui désirent approfondir les notions de réflexivité, sur un plan aussi bien logique que philosophique ou artistique, trouveront leur bonheur dans le mytique [Hof85].

## 1.6 Plan

Le chapitre 2 traite de la spécialisation de classes de façon abstraite. Le chapitre 3 s'intéresse au typage statique et à son rapport avec la spécialisation de classes, ainsi que de la sélection de méthodes. Le chapitre 4 s'intéresse à l'héritage multiple. Ces trois chapitres peuvent être utilement éclairés par la lecture du chapitre 6 qui propose un méta-modèle pour les propriétés. Entre temps, le chapitre 5 a introduit les notions liées à la méta-modélisation, à la méta-programmation et à la réflexivité. Le chapitre 7 décrit schématiquement l'implémentation des langages en typage statique, ce qui peut éclairer les spécifications des langages comme C++ ou JAVA. Le chapitre 8 est une rapide présentation du langage CLOS. Le dernier chapitre propose quelques exercices.

langage	référence	autres	Web
SMALLTALK	[GR83]		<a href="http://www.objectshare.com/">http://www.objectshare.com/</a> <a href="http://www.oti.com/links/smaltalk.htm">http://www.oti.com/links/smaltalk.htm</a>
COMMON LISP	[Ste90]		<a href="file:///usr/doc/cltl/clm/clm.html">file:///usr/doc/cltl/clm/clm.html</a> <a href="http://clisp.cons.org/">http://clisp.cons.org/</a>
CLOS	—	[Kee89, KdB91, Hab96]	<a href="http://ftp.sunet.se/ftp/pub/lang/lisp/oop/clos/">http://ftp.sunet.se/ftp/pub/lang/lisp/oop/clos/</a>
EIFFEL	[Mey94]	[Mey97]	
SMALL EIFFEL			<a href="http://www.loria.fr/projets/SmallEiffel/">http://www.loria.fr/projets/SmallEiffel/</a>
C++	[Str98, Str03]	[ES90, Lip96]	
OPENC++			<a href="http://www.hlla.is.tsukuba.ac.jp/~chiba/openc++.html">http://www.hlla.is.tsukuba.ac.jp/~chiba/openc++.html</a>
JAVA	[AG97]	[Gra97, GJS96] [HC99]	
OPENJAVA			<a href="http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/">http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/</a>
JAVASSIST			<a href="http://www.csg.is.titech.ac.jp/~chiba/javassist/">http://www.csg.is.titech.ac.jp/~chiba/javassist/</a>
PIZZA			<a href="http://www.cs.bell-labs.com/who/wadler/pizza/">http://www.cs.bell-labs.com/who/wadler/pizza/</a>
GENERIC JAVA			<a href="http://www.cs.bell-labs.com/who/wadler/pizza/gj/">http://www.cs.bell-labs.com/who/wadler/pizza/gj/</a>
C#	[Mic01]		

FIG. 1.3 – Bibliographie et sites Web par langage.

## Chapitre 2

# Classes, spécialisation et héritage

La spécialisation de classes et l'héritage de propriétés constituent certainement la caractéristique la plus originale de l'approche objet, ainsi que la cause des principales difficultés.

### 2.1 Classes, instances et propriétés

On en dira pour l'instant le moins possible, les choses se dévoilant progressivement en analysant la spécialisation. Disons simplement que une classe a des instances, une instance a (appartient à) une ou plusieurs classes : on parle alors de relation d'instanciation entre les instances et les classes. Par ailleurs, une classe a aussi des propriétés, qu'il faut comprendre comme des propriétés de ses instances, et non pas de la classe elle-même : le rôle de la classe est de décrire et stocker ces propriétés pour ses instances.

Ces propriétés sont de deux sortes — attributs (données) et méthodes (procédures ou fonctions) — pour lesquelles la classe intervient de façon asymétrique : la classe se contente de déclarer l'existence de l'attribut (la valeur étant stockée par l'objet), alors que la classe implémente la méthode, c'est-à-dire qu'elle en stocke elle-même la valeur fonctionnelle.

### 2.2 Sémantique de la spécialisation

Les trois entités de base du modèle objet — classes, objets et propriétés — sont concernées par la spécialisation. Deux notions sont à distinguer : la spécialisation de classes et l'héritage de propriétés. Le programmeur-concepteur définit des classes par spécialisation — c'est-à-dire des sous-classes de classes existantes — qui héritent des propriétés de leurs super-classes. Les objets sont créés par instanciation d'une classe.

#### 2.2.1 Inclusion des extensions

Une bonne utilisation de l'héritage consiste à lui conserver la sémantique de spécialisation qui remonte au moins à Aristote (ou plutôt à la tradition aristotélicienne) et qui s'illustre par le syllogisme classique : *Socrate est un homme, les hommes sont mortels, donc Socrate est mortel*. Ici, *Socrate* est un objet, *homme* et *mortel* sont des classes<sup>1</sup>. On généralisera en disant que :

**Axiome 2.1** *Les instances d'une classe sont aussi des instances de ses super-classes.*

Il faudra donc faire la différence entre les instances *directes* (ou *propres*) et les instances *indirectes*, de même qu'entre *une* classe de l'objet et *sa* classe, qui est l'unique classe minimum (suivant  $\prec$ ) dont il est instance directe.

Plus formellement, si l'on introduit la relation de spécialisation  $\prec$  ( $B$  sous-classe de  $A$  s'écrit  $B \prec A$ ) et une fonction *Ext* qui associe à chaque classe son *extension*, c'est-à-dire l'ensemble de ses instances<sup>2</sup>, alors :

$$B \prec A \implies \text{Ext}(B) \subseteq \text{Ext}(A) \tag{2.1}$$

L'implication n'est pas une équivalence : en effet, l'extension d'une classe est égale à l'union des extensions de ses sous-classes, à laquelle il faut ajouter ses instances propres. Si  $B$  est la seule sous-classe de  $A$  et que  $A$

<sup>1</sup> On remarquera que, dans la langue française et dans beaucoup d'autres, le verbe *être* — la *copule* pour les linguistes — désigne aussi bien la relation d'instance à classe, que la relation de classe à super-classe.

<sup>2</sup> On ne précisera pas plus cette notion d'extension, qui nécessiterait une formalisation plus poussée : il faut en effet pouvoir considérer toutes les instances potentielles, dans toutes les exécutions de tous les programmes possibles ...

est une classe *abstraite*, c'est-à-dire non instanciable, alors  $Ext(A) = Ext(B)$  mais  $A$  n'est évidemment pas une sous-classe de  $B$ . Une telle configuration n'est cependant pas très sensée : si  $A$  est une classe abstraite, il doit bien exister un programme dans lequel  $A$  a une autre sous-classe que  $B$ . Il est donc raisonnable de considérer que  $Ext$  est injective :  $Ext(A) = Ext(B) \Rightarrow A = B$ . Sous cette condition, une conséquence formelle de l'inclusion des extensions est :

**Propriété 2.1** *La relation de spécialisation  $\prec$  est transitive, anti-réflexive et antisymétrique : c'est un ordre partiel strict.*

La littérature parle souvent « d'héritage d'implémentation » pour désigner un héritage *ad hoc* ne respectant pas cette sémantique de la spécialisation : on ferait par exemple hériter la classe `Personne` de la classe `string` sous prétexte que les personnes ont un nom, de même que la `Tortue` de `LOGO` est une sous-classe de la classe `Point`. Cet usage est à proscrire, la bonne modélisation consistant à remplacer la spécialisation par une association ou une agrégation, en définissant un attribut `nom` de type `string`, ou un attribut `position` de type `Point`. En langue naturelle, une `Tortue`  $a$  un `Point`, elle n'en est pas un !

## 2.2.2 Héritage des propriétés

La syllogistique d'Aristote continue à s'appliquer lorsque l'on considère les propriétés des classes, qui sont, n'oublions pas, les propriétés des objets dont la description est factorisée dans les classes. Si  $B$  est une sous-classe de  $A$ , les instances de  $B$ , étant des instances de  $A$ , ont toutes les propriétés des instances de  $A$ . On dit donc que  $B$  hérite des propriétés de  $A$ .

**Propriété 2.2** *La sous-classe hérite des propriétés de ses super-classes.*

Plus formellement, si l'on introduit une fonction  $Int$  qui associe à chaque classe son *intension*, c'est-à-dire, en première approximation, l'ensemble de ses propriétés, alors :

$$B \prec A \implies Int(A) \subseteq Int(B) \quad (2.2)$$

C'est une conséquence formelle de l'inclusion des extensions (2.1) et du fait que toutes les instances ont les propriétés définies dans la classe. On notera que l'inclusion des intensions se fait en sens inverse de l'inclusion des extensions.

Au total, on voit que *héritage* et *spécialisation* sont des notions distinctes mais voisines : ce sont les deux faces d'une même médaille, l'un(e) ne pouvant pas aller sans l'autre.

## 2.2.3 Spécialisation des domaines

Ces propriétés peuvent être évaluées : dans ce cas, la *valeur* de la propriété dans une instance de la classe est contrainte à appartenir à un *domaine*<sup>3</sup>.

Ainsi, par exemple, les personnes ont un âge, dont le domaine est l'intervalle entier  $[0, 120]$ .

Lorsque l'on spécialise une classe, on spécialise aussi le domaine des propriétés de la classe :

$$B \prec A \implies Dom(B, p) \subseteq Dom(A, p) \quad (2.3)$$

Pour que ce soit une conséquence formelle de l'inclusion des extensions, il est nécessaire de définir le domaine comme l'ensemble des valeurs prises par la propriété sur l'extension de la classe<sup>4</sup>.

Ainsi, l'âge des enfants, spécialisation de personnes, a pour domaine  $[0, 16]$ . L'exemple de l'âge porte sur les attributs. On peut étendre la notion de domaine aux méthodes, à condition de considérer un domaine par paramètre, ainsi que pour la valeur de retour : le domaine est alors l'ensemble des valeurs prises par le paramètre, ou retournée par la méthode, dans toutes les exécutions possibles.

## 2.2.4 Covariance et contravariance

On a donc introduit 3 notions dont la variation est monotone relativement à la spécialisation, mais dans des sens différents. On dira que :

- les extensions se spécialisent de façon *covariante* ;
- les intensions se spécialisent de façon *contravariante* ;
- les domaines se spécialisent de façon *covariante*.

<sup>3</sup>On peut donc voir la propriété comme une fonction dont le *domaine* (au sens mathématique) est l'extension de la classe et le *co-domaine* est ce que nous appelons ici *domaine*.

<sup>4</sup>Cela revient à considérer la propriété comme une fonction *surjective* sur son co-domaine.

## 2.3 Spécialisation multiple

Deux classes  $A$  et  $B$  sont *incomparables* lorsqu'aucune des deux n'est spécialisation de l'autre, c'est-à-dire que l'on n'a ni  $A \prec B$ , ni  $B \prec A$ .

On parle de *spécialisation multiple* lorsqu'une classe  $C$  est la spécialisation d'au moins deux classes incomparables,  $A$  et  $B$  :  $C \prec A$  et  $C \prec B$ . Il vient alors naturellement, pour les extensions, les intensions et les domaines :

$$Ext(C) \subseteq Ext(A) \cap Ext(B) \quad (2.4)$$

$$Int(C) \supseteq Int(A) \cup Int(B) \quad (2.5)$$

$$Dom(C, p) \subseteq Dom(A, p) \cap Dom(B, p) \quad (2.6)$$

La spécialisation multiple est donc tout à fait naturelle. Cependant, en cas de spécialisation multiple, l'héritage de propriétés conduit à des problèmes d'ambiguïtés ou de conflits, assez classiques mais souvent maltraités, que nous examinerons au chapitre 4, en nous aidant du méta-modèle introduit au chapitre 6. C'est la problématique connue sous l'appellation d'*héritage multiple*.

## 2.4 Mono- et multi-instanciation

La spécialisation multiple a son pendant du côté de l'instanciation. La *mono-instanciation* se définit par le fait que, si un objet est instance de deux classes incomparables, alors il est instance de l'une de leurs sous-classes communes. Tout objet  $a$  a une classe directe unique. On parle de *multi-instanciation* lorsque ce n'est pas le cas.

La multi-instanciation n'existe que dans quelques langages de laboratoires, ou dans des modèles plus lointainement apparentés au modèle objet standard sur lequel nous nous appuyons. Nous resterons donc dans un cadre strict de mono-instanciation.

## Chapitre 3

# Typage et sélection de méthodes

Le typage est une notion fondamentale des langages de programmation, dont l'approche objet a, si l'on peut dire, hérité. La prise en compte du typage dans l'approche objet conduit à divers problèmes. D'une part, la proximité entre les types et les classes fait que la spécialisation induit un sous-typage qui lui est pour partie contraire. D'autre part, le typage peut introduire des ambiguïtés dans la sélection de méthodes.

### 3.1 Typage et sous-typage

**Typage statique et typage dynamique** Les langages de programmation se distinguent suivant que leur typage est statique ou dynamique. Dans tous les langages, les valeurs ont un type, soit que ce type soit codé dans la chaîne de bit représentant la valeur, soit qu'elle se déduise simplement de l'usage de cette valeur.

Le typage statique se caractérise par le fait que les diverses entités du programme (variables, fonctions, etc.) sont explicitement typées par des annotations de type : c'est le cas de C, C++, JAVA, EIFFEL, PASCAL, etc. : dans ce contexte, le type des valeurs n'a pas besoin d'être codé dans la valeur.

Dans le typage dynamique, il n'y a aucune annotation de type dans les programmes : c'est le cas de LISP, SCHEME, CLOS, SMALLTALK. L'absence d'annotation de type rend indispensable un codage du type dans la valeur et des vérifications dynamique de type. En pratique, dans ces langages, il n'est pas nécessaire de faire une différence entre type et classe.

Entre ces deux positions extrêmes, on trouve des langages dont le prototype est ML, qui ne dispose pas d'annotations de type mais dont des types statiques sont déduits par une inférence de types.

**Type et classe.** Un type est souvent vu comme un ensemble de valeurs et un ensemble d'opérateurs qui s'appliquent à ces valeurs, bref comme une extension et une intension. Il est donc assez naturel de chercher à faire des classes des types, à identifier les deux, ou encore à associer un type à chaque classe.

Il est alors tout à fait naturel de faire coïncider la spécialisation avec une notion de *sous-typage* : intuitivement, un sous-type serait défini par un sous-ensemble des valeurs du type et un sur-ensemble de ses opérateurs, un peu comme une sous-classe. Inversement, le *domaine* des propriétés serait un *type*.

En typage dynamique, cette identification des classes et des types ne pose pas de problème. C'est en typage statique que tout se complique.

Dans la suite de cette section, sauf mention contraire, nous ne considérerons que le typage statique.

**Type statique et type dynamique.** Dans les langages à typage statique, il faut distinguer le type *statique* des *expressions* du programme, qui concernent le compilateur et le programmeur, et le type dynamique des *valeurs* que ces expressions prennent à l'exécution. Avec le polymorphisme lié au sous-typage<sup>1</sup>, le type dynamique des valeurs peut être un sous-type du type statique des expressions.

Dans les langages à objets à typage dynamique, les expressions du langage ne sont pas typées. Il y a pourtant une exception pour le receveur du message (*self*, *this* ou *current*, suivant les langages), qui a toujours implicitement le type statique de la classe qui définit la méthode.

**Sémantique du modèle objet en typage statique.** Quand on considère le typage statique, le critère principal pour analyser un trait de langage est que

---

<sup>1</sup> Dit *polymorphisme d'inclusion*.

**Axiome 3.1** *Seul le type dynamique d'un objet, c'est-à-dire la classe qui l'a instancié, détermine la sémantique de l'objet, c'est-à-dire son comportement.*

La liaison tardive veut que ce soit *toujours* le type dynamique qui soit pris en compte, et non le type statique, dont le rôle se réduit à la vérification qu'il n'y aura pas d'erreur de type. Le type statique ne doit donc avoir aucune influence, autre que syntaxique, sur le comportement des valeurs. C'est une indication au compilateur qui, idéalement, doit permettre des vérifications sans avoir d'influence sur le code généré. Cela signifie en pratique que les effets apparents du typage statique sur le comportement des programmes peuvent se ramener à des manipulations syntaxiques simples comme un renommage.

**Typage statique sûr.** Pour bien comprendre la suite, il est indispensable de préciser à quoi sert le typage statique. Il a essentiellement un rôle de commentaire, adressé aussi bien au(x) programmeur(s) qu'au compilateur.

Pour le programmeur, c'est un commentaire qui a l'avantage d'être formel : si le commentaire est obsolète, le compilateur devrait s'en apercevoir. C'est aussi une contrainte, la programmation rapide étant plus agréable en typage dynamique.

Pour le compilateur, le typage statique sert à vérifier la correction du programme et à générer le code approprié. Son rôle essentiel est d'assurer l'absence d'*erreur de type* à l'exécution. C'est ce que l'on appelle le *typage sûr*.

**Erreurs de type** Deux erreurs de type sont possibles, à l'exécution, dans un envoi de message de la forme  $x.foo(arg)$  (où  $foo$  peut être aussi bien un attribut qu'une méthode) :

- l'erreur message *inconnu*, quand le type dynamique du receveur ( $x$ ) ne connaît pas la méthode  $foo$  ;
- l'erreur mauvais type d'argument, lorsque le type dynamique de l'argument  $arg$  n'est pas conforme au type du paramètre de  $foo$  dans le type dynamique de  $x$ .

Ces erreurs de type sont définies à partir des types dynamiques, or le compilateur ne dispose que des types statiques pour vérifier la correction.

Le typage sûr va donc se baser sur une version pessimiste de ces erreurs de type, où les types dynamiques sont remplacés par les types statiques de façon à imposer une quantification universelle sur tous les types dynamiques possibles (le type dynamique devant toujours être un sous-type du type statique) :

- l'erreur message *inconnu* survient quand le type *statique* du receveur ( $x$ ) ne connaît pas la méthode  $foo$  ;
- l'erreur mauvais type d'argument, lorsque le type *statique* de l'argument  $arg$  n'est pas *conforme* au (c'est-à-dire sous-type de) type du paramètre de  $foo$  dans le type *statique* de  $x$ .

**Sous-typage et substituabilité.** Le sous-typage se définit alors à partir du typage sûr, en passant par la notion de *substituabilité*.

**Définition 3.1** *Un type  $t_1$  est un sous-type d'un type  $t_2$  (noté  $t_1 <: t_2$ ) si toute valeur du type  $t_1$  peut être substituée, à l'exécution, à toute expression du type  $t_2$  sans déclencher d'erreur de type.*

La quantification universelle implicite considère la totalité des programmes corrects qui peuvent être écrits en utilisant les types  $t_1$  et  $t_2$ .

**Affectation et passage de paramètre polymorphe** On parlera d'affectation (resp. de passage de paramètre) polymorphe lorsque l'on affecte à une variable de type statique  $A$  une valeur désignée par une entité (variable ou valeur de retour de fonction) de type statique  $B$ , sous-type de  $A$  (resp. en cas de passage d'un argument de type statique  $B$  à un paramètre formel de type statique  $A$ ).

**Sous-type et sous-classe.** Pour qu'une sous-classe puisse être considérée comme un sous-type (on peut alors identifier  $<$  et  $<:$ ), il faut que la définition de la sous-classe respecte les contraintes imposées par la substituabilité.

En revanche, inversement, une classe peut être un sous-type sans être une sous-classe : il suffit d'imaginer deux classes de même définition, à part leur nom, qui seront parfaitement substituables l'une à l'autre.

De fait, si la relation de spécialisation est un ordre partiel, la relation de sous-typage, au sens strict de la substituabilité, n'est qu'un préordre<sup>2</sup>. En pratique, les langages prennent en général la relation de spécialisation comme relation de sous-typage et les substituabilités qui ne coïncident pas avec la spécialisation ne sont pas considérées.

<sup>2</sup> Rappel : un préordre est une relation *réflexive* et *transitive*. Un ordre partiel est un préordre *antisymétrique*.

type	$A$	$B$	règle
de retour	$m_A() : t$	$m_B() : u$	$u <: t$
des paramètres	$m_A(t)$	$m_B(u)$	$t <: u$
des attributs	$m_A : t$	$m_B : u$	$t = u$

FIG. 3.1 – Résumé des règles de redéfinition, pour 2 classes  $A$  et  $B$ , avec  $B <: A$

## 3.2 Sous-typage et redéfinition.

Pour qu’une sous-classe puisse être considérée comme un sous-type, il est nécessaire de contraindre la façon dont un attribut ou une méthode définie dans une classe peut être redéfinie dans une sous-classe.

En effet, l’introduction de nouvelles propriétés<sup>3</sup> dans la sous-classe ne pose aucun problème. En revanche, la redéfinition peut poser des problèmes lorsqu’elle concerne le type d’un attribut ou la signature d’une méthode.

### 3.2.1 Redéfinition et surcharge statique

On évitera, d’abord, de confondre *redéfinition* (ou *surcharge dynamique*) et *surcharge statique*. La redéfinition et la surcharge consistent, l’une et l’autre, en une description, dans le corps de la sous-classe, d’une propriété déjà définie dans la super-classe (ou d’une propriété de même nom qu’une propriété déjà définie). Cette description constitue une modification de la description dans la sous-classe, qui peut concerner le type d’un attribut, la signature (type des paramètres et de la valeur retournée) d’une méthode, ou le corps de la méthode.

C’est une *redéfinition* si tout se passe comme si la propriété décrite dans la sous-classe remplaçait celle de la super-classe dans tous les contextes où une propriété de ce nom est concernée, sur une instance de la sous-classe : on appelle ce phénomène le *masquage*.

- Dans le cas d’un attribut, un seul attribut de ce nom est implémenté physiquement dans l’objet.
- Dans le cas d’une méthode, c’est la *liaison tardive* qui va permettre de faire appel à la procédure appropriée, suivant le type de l’objet receveur.

C’est une *surcharge statique* si tout se passe comme si les deux propriétés coexistaient dans les instances de la sous-classe, et que l’une ou l’autre était désignée suivant le contexte dans lequel on utilise le nom de la propriété. Dans ce cas, la discrimination se fait statiquement, à la compilation, d’après le nombre et le type des paramètres.

### 3.2.2 Redéfinition sans surcharge statique

Nous aborderons plus en détail la surcharge statique en section 3.8.4 et nous intéressons ici à la redéfinition, dans un contexte où il n’y a pas de surcharge statique.

NOTATIONS. — Dans la suite, on notera  $m_A(u) : v$  une méthode de nom  $m$ , définie dans la classe  $B$  avec un type de paramètre  $u$  et un type de retour  $v$ . Dans le cas de plusieurs paramètres, on se ramène à un paramètre unique dans le produit des types (cf. section 3.8.1). En cas d’absence de paramètres, on la notera  $m_A() : v$ . Pour un attribut, on utilisera la notation  $m_A : v$ . Suivant le contexte, les annotations de type ou de classe pourront être omises.

Enfin, pour les types statique et dynamique du receveur, on utilisera les notations respectives  $\tau_s$  et  $\tau_d$ , avec  $\tau_d \leq \tau_s$ .

**Redéfinition covariante du type de retour.** Soit une méthode  $m$  définie dans la classe  $A$  avec un type de retour  $t$  et redéfinie dans la classe  $B$ , sous-classe de  $A$  avec un type de retour  $u$ . Dans tout contexte comme :

```
x : A
y : t
x := new B
y := x.m( . . . )
```

$x$  doit être substituable par toute instance de  $A$ , donc en particulier de  $B$ , ce qui impose que  $u$  soit un sous-type de  $t$ .

On dira que la redéfinition est *covariante* sur le type de retour, puisque les types varient dans le même sens.

**NB.** L’affectation polymorphe de l’exemple est déraisonnable. Plutôt que de faire  $x : A ; x := new B$ , autant typer par  $x : B$ , sinon il sera nécessaire de faire une coercition sur  $x$  pour pouvoir lui appliquer les méthodes introduites dans  $B$ . Cependant, cette formulation a l’avantage de la concision par rapport à une formulation équivalente faisant appel à un passage de paramètre polymorphe  $f_{\circ\circ}(new B)$  où  $f_{\circ\circ}(x : A)$  contient l’exemple précédent.

<sup>3</sup> C’est-à-dire des propriétés dont le nom ne correspond à aucune propriété des super-classes.

**Redéfinition contravariante du type des paramètres.** Soit une méthode  $m$  définie dans la classe  $A$  avec un type de paramètre  $t$  et redéfinie dans la classe  $B$ , sous-classe de  $A$  avec un type de paramètre  $u$ . Dans tout contexte comme :

```
x : A
y : t
x := new B
x.m(y)
```

$x$  doit être substituable par toute instance de  $A$ , en particulier une instance de  $B$ , et donc la méthode  $m$  de  $A$  doit avoir le même nombre de paramètres que celle de  $B$  et accepter toute valeur du type  $t$  comme paramètre, ce qui impose que  $t$  soit un sous-type de  $u$ .

On dira que la redéfinition est *contravariante* sur le type du paramètre, puisque les types varient en sens contraire.

**Redéfinition invariante du type d'un attribut.** Le cas d'un attribut se ramène à celui d'une méthode :

- lorsqu'on accède en lecture à l'attribut, on peut le considérer comme une méthode sans paramètre dont le type de l'attribut est le type de retour : la redéfinition doit être covariante ;
- lorsqu'on accède en écriture (affectation) à l'attribut, on peut le considérer comme une méthode avec un paramètre, la valeur à affecter, dont le type est le type de l'attribut : la redéfinition doit être contravariante.

On en conclut qu'il n'est pas possible de redéfinir le type d'un attribut dans une sous-classe : la redéfinition est *invariante*.

**Règle de contravariance.** Elle est définie par la redéfinition covariante des types de retour, contravariante des types de paramètres et invariante des types d'attributs.

### 3.3 Types paramétrés et généricité

La *généricité*, aussi appelée *polymorphisme paramétrique*, consiste en la possibilité de paramétrer des types par d'autres types. Il s'agit donc de ce que l'on appelle *templates*, en C++.

**Instanciation d'un type paramétré.** Le type (la classe) paramétré(e) est un type ou une classe normale, sauf qu'il est paramétré par un ou plusieurs paramètres formels qui peuvent être utilisés comme des types dans sa définition. C'est donc un générateur de types (classes), par instanciation de son(s) type(s) formel(s).

Soit un type (ou une classe) `vaisselle` et un type `pile<T>` paramétré par le type formel  $T$ . Le principe de l'instanciation est de substituer toutes les occurrences du type formel  $T$ , par un type (ou classe), par exemple ici `vaisselle`, pour obtenir le type non paramétré `pile<vaisselle>`. La sémantique de la classe ainsi obtenue est exactement celle qu'aurait eu le programme résultant d'une substitution textuelle de  $T$  par `vaisselle`.

**Généricité et sous-typage.** Notons d'abord qu'il n'y a aucune relation de sous-typage entre `pile<vaisselle>` et `pile<T>`. A proprement parler, le dernier n'est d'ailleurs pas un type : ou plutôt, ce n'est un type que dans la définition d'un autre type paramétré par le même type formel  $T$ .

Par ailleurs, les types paramétrés posent des problèmes de sous-typage et de spécialisation à cause de la règle de contravariance. En effet, si `assiette` et `verre` sont des sous-types de `vaisselle`, il semblerait raisonnable de penser que `pile<assiette>` est un sous-type de `pile<vaisselle>` : une pile d'assiette *est* bien une pile de vaisselle. Pourtant si c'en est bien une spécialisation, ce n'en est pourtant pas un sous-type : elles ne sont pas substituables.

En effet, les deux opérations `pop` et `push` de `pile<T>` doivent respectivement retourner une valeur et prendre un paramètre de type  $T$ . Par le même raisonnement que précédemment, il s'ensuit que, si `pile<assiette>` est un sous-type de `pile<vaisselle>`, le type du paramètre de `push` devrait varier de façon contravariante. Les 2 types paramétrés sont donc incomparables.

**Type paramétré et méta-classe.** En fait, `pile<assiette>` doit plutôt être considéré comme une instance de `pile<T>` qui serait alors une méta-classe (cf. chapitre 5 sur la réflexivité). En effet, considérer `pile<vaisselle>` comme un sous-type de `pile<T>` revient à confondre `pile<T>` et `pile<object>`, où `object` serait la classe de tous les objets, racine du graphe de la hiérarchie de classes.

De fait, `pile<T>` n'est pas à proprement parler un type puisqu'il est impossible de s'en servir pour annoter un élément de programme, du moins en dehors de la définition d'un autre type paramétré par  $T$ , c'est-à-dire lorsque la variable  $T$  est libre : c'est un type d'ordre supérieur.

**Généricité bornée.** Le code de la classe paramétrée peut se servir d'opérations spécifiques de son type formel, ce qui n'était pas le cas de la classe `pile<T>`. Les instantiations de la classe paramétrée ne sont donc correctes que si le type formel est instancié par un type qui possède bien les opérations utilisées. Pour que le compilateur puisse vérifier la correction de la classe paramétrée, il faut donc donner des informations sur le type formel.

La généricité bornée (ou contrainte) consiste à contraindre le type formel à être un sous-type d'un type donné. Par exemple, le type « ensemble ordonné »<sup>4</sup> pourrait être typé comme `OrderedSet<T <: Comparable>`, où le type `Comparable` doit posséder un opérateur de comparaison (ordre).

La généricité bornée permet au compilateur de vérifier qu'un type paramétré est correct du point de vue des types, sans attendre son instantiation sur un type particulier. La généricité bornée existe en Eiffel ou Pizza, mais pas en C++ : c'est ce qui explique que ce dernier ne compile pas réellement les *templates*.

La généricité *F-bornée* est une généralisation où la borne peut être elle-même paramétrée par le type formel. On définira ainsi les ensembles ordonnés par `OrderedSet<T <: Comparable<T>>`, ce qui permet de spécifier que le type  $T$  doit être `Comparable` avec lui-même.

**Spécialisation de types paramétrés.** Nous avons vu que la règle de contravariance interdisait le sous-typage entre, par exemple, `Set<integer>` et `Set<number>` (avec le sous-typage naturel entre `integer` et `number`).

En revanche, la spécialisation est possible entre 2 types paramétrés : `OrderedSet<T>` peut être défini comme un sous-type de `Set<T>` (sous-réserve de le faire proprement).

En cas de généricité bornée, le sous-typage entre  $A'<T <: B'>$  et  $A<T <: B>$  est possible, mais il impose une relation de sous-typage entre les bornes.

$A'<T <: B'> <: A<T <: B>$  signifie que  $A'<T> <: A<T>$ , quelque soit  $T <: B'$ , et il faut que  $B' <: B$ . Le super-type peut ne pas être borné : cela revient à prendre pour  $B$  le type le plus général (`Object` en Java).

`OrderedSet<T <: Comparable>` peut donc être un sous-type de `Set<T>` aussi bien que de `Set<T <: Comparable>`, à condition que `Comparable` soit un sous-type de `Comparable`<sup>5</sup>.

### 3.3.1 Attributs immutables

Les attributs considérés jusqu'ici sont *mutables*, au sens où il est possible d'en modifier la valeur dynamiquement, par une affectation, après la création de l'objet. Lorsque l'attribut est *immutable*, la nécessité de la contravariance disparaît, en même temps que la possibilité d'y accéder en écriture.

Ainsi, lorsque l'attribut est *immutable*, ou lorsque le type paramétré est immuable — par exemple, une liste qu'il n'est pas possible de modifier —, on peut envisager une redéfinition *covariante*.

**Méthodes et types fonctionnels.** Une méthode peut être considérée comme un attribut immuable dont le type est fonctionnel. Si l'on note  $t_1 \rightarrow t_2$  le type des fonctions qui prennent un argument de type  $t_1$  et retournent une valeur de type  $t_2$ , on constate, par les mêmes arguments que précédemment, que :

$$t_1 \rightarrow t_2 <: u_1 \rightarrow u_2 \text{ ssi } t_2 <: u_2 \ \& \ u_1 <: t_1 \quad (3.1)$$

La méthode étant immuable, sa redéfinition est covariante, ce qui ramène bien aux conclusions du paragraphe 3.2.

En corollaire, on voit qu'un système de types pour les objets inclut un système de type fonctionnel. Ce n'est pas réellement étonnant : pour avoir des valeurs fonctionnelles de première classe, il suffit de définir une classe avec une méthode `apply`. Inversement, la présence de valeur fonctionnelle de première classe, et d'un équivalent de `apply` provoque un phénomène de liaison tardive : la compilation de `apply` est aussi difficile que celle d'un appel de méthode.

**Type d'un objet.** Le type ou la classe d'un objet peut lui-même être considéré comme un attribut immuable, cette immutabilité permettant d'en assurer une redéfinition covariante, ce qui est le moins : le type des instances du sous-type doit bien être un sous-type !

<sup>4</sup> Le type (ou plutôt la classe) « ensemble ordonné » est une implémentation des ensembles optimisée par le fait qu'une relation d'ordre est donnée sur les éléments : on peut donc rechercher ou insérer dans l'ordre (complexité divisée de moitié) ou par dichotomie (complexité logarithmique).

<sup>5</sup> En toute généralité, `Set<T>` nécessite que le type  $T$  soit muni de l'égalité.

## 3.4 Variations sur C++ et JAVA

### 3.4.1 Programmation objet et/en C++.

L'essence de la programmation objet est que l'objet (c'est-à-dire son type dynamique) détermine son propre comportement. Il s'ensuit que seul l'objet peut se manipuler lui-même : c'est l'*encapsulation*, telle que pratiquée en SMALLTALK et EIFFEL.

A l'inverse, en C, chaque structure de données (chaîne de 0 et de 1) peut être manipulée par n'importe quelle procédure. C et la programmation par objets sont donc parfaitement antagonistes et cela explique bien des défauts de C++ qui essaie de concilier les 2.

Par ailleurs, l'ensemble de ce qui précède ne s'applique à un sous-ensemble de C++, car par défaut le langage est inadapté à la programmation par objets :

- le mot-clef `virtual` doit être utilisé partout, aussi bien pour les méthodes que pour l'héritage (cf. chapitre 4) ;
- le comportement des classes dépend de leur implémentation qui elle-même dépend de ce qu'elles contiennent : pour avoir un comportement normal, il faut que toute classe comporte au moins une méthode `virtual` ;
- le mode par défaut de désignation des valeurs et des types ne permet pas le polymorphisme : il est donc nécessaire de se servir explicitement de pointeurs ou de références, avec les risques inhérents à leur plein usage : tout ce qui précède s'applique en revanche au mode de désignation des valeurs à la LISP ou JAVA, par leurs adresses sans en faire un pointeur.

Enfin, les *templates* de C++ correspondent bien à la généricité, mais cette généricité n'est pas bornée, ce qui explique que le compilateur n'en fasse pas grand chose.

### 3.4.2 Et JAVA

**Interfaces de JAVA.** La notion d'*interface* de JAVA est un très bon exemple d'entité intermédiaire entre classe et type. La relation de spécialisation entre classes ou entre interfaces, ainsi que la relation dite d'implémentation, entre classes et interfaces, sont des relations de sous-typage.

Sur la notion d'interface, voir aussi les sections 3.8.6 et 4.6.

**Les tableaux en JAVA.** Ils sont en parfaite contradiction avec la théorie des types : si  $X$  est une sous-classe de  $Y$ ,  $X[]$  est considéré comme un sous-type de  $Y[]$ . Aux risques et périls du programmeur, du programme et des passagers.

**La généricité en JAVA.** A part les tableaux, il n'y a pas de généricité en JAVA. Deux extensions de JAVA introduisent des types paramétrés (et d'autres constructions comme les fonctions de « première classe ») : PIZZA et GENERIC JAVA.

## 3.5 Coercition

La coercition ou conversion de type — en anglais *coercion*, en C++ et JAVA *cast* (en français attribuer un rôle) — permet de convertir une valeur d'un type (source) dans un autre (cible). *A priori*, la coercition garantit l'absence de sûreté du typage et la présence d'un mécanisme de coercition est contradictoire avec la revendication d'un typage sûr.

On peut distinguer trois sortes de coercition, suivant la relation de sous-typage existant entre les types source et cible.

### 3.5.1 Conversion entre types incomparables

C'est la coercition classique des langages de programmation. On veut par exemple convertir un entier en flottant (2 donnera 2.) ou en `string` (2 donnera "2"), etc. Voir par exemple la fonction `coerce` de COMMON LISP, qui permet de convertir n'importe quoi en n'importe quoi, ou encore la coercition implicite ou explicite des types primitifs en C.

Dans ce cas, aucun principe général ne s'applique et il n'y a que des cas d'espèces, à traiter pour chaque paire de types source-cible. Les conversions implicites sont très dangereuses en elles-mêmes. Couplées avec la surcharge statique elles peuvent rendre fous les meilleurs. Aussi évitez de prendre comme exemple, pour l'étude de la surcharge statique, des types primitifs comme `int`, `float` ou `char`.

### 3.5.2 Coercition ascendante

La coercition « ascendante » permet de prendre une instance de la sous-classe pour une instance directe de la super-classe. C'est un procédé *ad hoc* dont le sens échappe un peu : cela peut servir à empêcher d'utiliser une méthode *m* spécifique à la sous-classe *B*, pour la remplacer par la méthode *m* d'une super-classe *A*. Ce procédé peut ne s'appliquer qu'à la méthode *m*, l'objet receveur conservant son type *B* pour les méthodes appelées par *m*. Il peut aussi, mais pas forcément — ce doute est à lui seul un problème —, s'appliquer récursivement à toutes les méthodes appelées par *m*, ce qui revient à modifier le type dynamique de l'objet.

Ce mécanisme dual est souvent appelé « coercition implicite » en C++.

On obtiendra une vraie conversion, en C++, lors d'une affectation polymorphe dont les types ne sont pas des références : il s'agit d'une copie où tout ce qui dépasse est tronqué.

On obtiendra une conversion apparente en cas d'affectation ou de passage de paramètre polymorphe (en C++, uniquement si les types sont bien des références). Dans ce cas, « il ne se passe » strictement rien — on applique seulement le polymorphisme et le principe de *substituabilité* —, et il n'y aurait pas lieu de nommer ce rien si divers mécanismes du langage n'en faisait pas un élément de contrôle des programmes. La raison d'une telle notion, *a priori* vide de sens, provient des diverses aberrations du langage, en particulier de la surcharge statique qui fait jouer un rôle exorbitant aux types statiques des paramètres (pour les méthodes `virtual`) mais aussi au receveur pour les “méthodes” non `virtual`, à liaison statique, qui font que le type dynamique n'est plus considéré, ce qui revient à une conversion momentanée.

La coercition ascendante peut donc être nécessaire, en JAVA et surtout en C++, pour résoudre la surcharge statique, soit lorsqu'elle est ambiguë directement, soit lorsque la méthode visée est masquée (C++), soit pour résoudre la sélection autrement.

Une raison supplémentaire est implémentatoire : avec l'héritage multiple, les pointeurs de types statiques différents sur le même objet ne sont pas égaux, car ils ne pointent pas au même endroit dans l'objet [ES90, Lip96] (cf. chapitre 7). Mais les détails d'implémentation ne doivent jamais remonter au niveau conceptuel.

### 3.5.3 Coercition descendante

La coercition « descendante » a un rôle plus compréhensible, si ce n'est plus avouable. Elle suppose que le type dynamique de la valeur à convertir est un sous-type du type cible. Elle permet de faire réapparaître statiquement le type dynamique de l'objet, ou, si l'on préfère, de faire réapparaître la spécialisation là où le sous-typage est interdit.

Ainsi, puisque l'on ne peut pas spécialiser de façon covariante un paramètre, on cherchera à le « convertir » dans le sous-type avec lequel on aurait voulu le typer. Ou bien, comme un type paramétré par un sous-type n'est pas un sous-type, on pourra chercher à convertir un élément d'une `pile(vaisselle)` pour en faire une `assiette`. Si l'on voit bien l'utilité d'un tel procédé, son existence même est un sérieux argument en faveur des infractions à la théorie des types (cf. *infra*).

Le bon usage de la coercition descendante consistera à l'accompagner d'un test dynamique de type pour vérifier qu'il n'y a pas d'erreur, c'est-à-dire que les prévisions sont bien réalisées : on se comporte ainsi comme avec un typage dynamique, mais seulement sur un sous-ensemble des types, les sous-types du type source.

**Coercition descendante en C++, JAVA et EIFFEL.** Le recours à la coercition descendante est malheureusement souvent nécessaire, ne serait-ce que pour implémenter les modèles qui ont besoin de covariance. Cette coercition doit impérativement se faire au moyen d'un test de sous-typage, pour vérifier que l'objet considéré est bien une instance du type cible.

En C++, il est impératif d'utiliser l'opérateur `dynamic_cast` qui effectue effectivement ce test et lève une exception s'il n'est pas vérifié : `dynamic_cast<target>(valeur)`. La syntaxe parenthésée héritée de C, `(target)valeur` ne doit en aucun cas être utilisée : elle correspond à l'opérateur `static_cast`, qui ne fait aucun test.

En revanche, en JAVA, la syntaxe parenthésée correspond assez exactement au `dynamic_cast` de C++. Par ailleurs, l'absence de généricité en JAVA conduit à un usage assez immodéré de la coercition descendante : cette malheureuse habitude ne doit en aucun cas être conservée avec d'autres langages.

Enfin, en EIFFEL, la coercition descendante est peu utile puisque le langage dispose à la fois de redéfinition covariante et de généricité. Le mécanisme de *tentative d'affectation* (opérateur `?=`) la permet néanmoins.

### 3.5.4 Coercition et conversion

La coercition entre types incomparables et l'affectation polymorphe par copie tronquée (en C++ avec des types non référence) sont les seuls cas pour lequel on peut vraiment parler de conversion.

En revanche, les coercitions ascendante ou descendante, ne sont en général pas une réelle conversion de type, laquelle nécessiterait une transformation physique de la valeur. Les coercitions ascendante ou descendante se contentent de faire « comme si » la valeur était du type cible : c'est juste une indication donnée au compilateur sur la façon dont il doit compiler un certain envoi de message. L'influence de cette transformation n'est que syntaxique.

### 3.5.5 Pièges de C++ et JAVA

Dans vos tests, il est impératif d'éviter les types primitifs usuels (`int`, `float`, `string`) car ces langages pratiquent des conversions implicites qui peuvent fausser toutes vos observations, en particulier pour la surcharge.

En C++, il faut aussi que les classes aient au moins une méthode virtuelle : à défaut, des coercitions apparemment impossibles pourraient se révéler possibles.

## 3.6 Contradiction entre typage et sémantique de spécialisation

Les conclusions qui précèdent sur les règles de redéfinition imposées par le sous-typage sont en parfaite contradiction avec l'intuition et avec ce que voudrait exprimer le concepteur qui doit modéliser un domaine d'application : la règle de contravariance pose un problème. Ainsi, les `personnes` ont un `age`, inclus dans un intervalle, `[0, 120]` par exemple, les `enfants` (sous-classe de la précédente) ont leur `age` inclus dans l'intervalle `[0, 17]`. Ou bien, les `animaux` mangent des `matières organiques`, les `carnivores` mangent des `animaux` et les `phoques` mangent des `poissons` : `manger` peut être considéré comme une méthode avec un paramètre, et les relations de spécialisation sont clairement covariantes sur le type du paramètre<sup>6</sup>.

L'opposition réside dans le constat suivant. Le typage et la règle de substituabilité font une quantification universelle : tous les cas doivent être considérés pour assurer l'absence d'erreur de type. En revanche, la conception ou la représentation ne font qu'une hypothèse existentielle : une `personne` a un `age`, un `phoque` mange un `poisson`. Et non pas tous les âges, ou tous les poissons !

**Erreur de type en programmation ou en conception.** C'est une contradiction irréductible : la sûreté du typage et l'expressivité sont inconciliables. Soit une erreur de type est possible, soit un `phoque` risque de manger autre chose que du `poisson`, ce qui est aussi, d'une certaine manière, une erreur de type.

Dit autrement, le risque d'erreur de type est inévitable : s'il est exclu d'un langage de programmation, il se produira au niveau de la conception. L'exemple de la coercition descendante montre bien que, si l'on chasse les erreurs de type par la porte, ils rentrent par la fenêtre.

**Langages covariants ou contravariants.** La plupart des langages privilégient la sûreté du typage, sans forcément réussir à éviter toute erreur de type : c'est la position de C++ ou de JAVA : notons que JAVA ne permet pas une redéfinition covariante du type de retour, ce qui revient à être plus strict que la théorie ne l'impose.

D'autres langages privilégient l'expressivité de la représentation, quitte à imposer des contraintes pour réduire les risques : c'est le cas de EIFFEL — qui impose des restrictions sur les « *catcalls* » — et du SGBD O<sub>2</sub>.

## 3.7 Variance sur les exceptions ou les droits d'accès

Le problème de la co/contra-variance se pose aussi pour des caractéristiques des propriétés qui ne sont pas liées directement aux types : par exemple, les exceptions et les droits d'accès aux propriétés.

### 3.7.1 Les exceptions

Les langages de programmation demandent ou permettent au programmeur d'indiquer les exceptions qu'une classe ou une méthode peut signaler, directement ou indirectement. Suivant le point de vue, cet ensemble d'exceptions peut varier de différentes façons avec la spécialisation :

- si l'on considère que ces exceptions constituent toutes les exceptions signalables par une instance de la classe, elles devraient varier de façon covariante, les instances de la sous-classe signalant moins d'exceptions ;
- si l'on adopte un point de vue plus intensionnel, il y aurait plutôt contravariance car la sous-classe introduit de nouvelles propriétés qui peuvent signaler de nouvelles exceptions ;
- la substituabilité est ici d'accord avec la spécialisation en exigeant la covariance mais l'extensibilité voudrait plutôt la contravariance !

---

<sup>6</sup>Et la crise de la vache folle ne résulte ainsi que d'une banale erreur de type !

### 3.7.2 Les droits d'accès statiques

De façon générale, les droits d'accès statiques permettent de dire si une classe  $A$  a le droit d'accéder aux propriétés des instances d'une classe  $A'$ . Ces droits d'accès sont statiques. Si  $B' \prec A'$ , et que  $A$  a les droits d'accès sur une propriété  $p$  de  $A'$ , il peut y accéder sur des instances de  $B'$  référencées par une entité de type  $A'$ . Il est donc raisonnable d'exiger que  $A$  a aussi les droits sur  $p$  dans  $B'$ .

Donc, quand on spécialise une classe, on ne peut qu'élargir l'ensemble des classes qui ont le droit d'accéder à une propriété donnée : cet ensemble est donc contravariant.

Bien entendu, par la sémantique de la spécialisation, l'ensemble des classes qui accéderont effectivement à la propriété évoluerait plutôt de façon covariante. La redéfinition covariante des types s'accorde bien à la redéfinition covariante de l'ensemble des classes qui ont un droit d'accès. Eiffel pratique d'ailleurs les 2 redéfinitions covariantes, qui donnent les deux cas possibles de *catcalls*<sup>7</sup>.

## 3.8 Sélection multiple et surcharge statique

### 3.8.1 Sélection multiple : une vraie liaison tardive

**Envoi de message et objet receveur.** Dans la plupart des langages à objets, l'appel de méthode est régi par la métaphore de l'envoi de message, ou liaison tardive. Dans un appel de procédure ou de fonction classique, tous les paramètres de la procédure ou fonction jouent sensiblement le même rôle. Avec l'envoi de message, un paramètre est privilégié : le receveur du message. C'est suivant son type dynamique (sa classe) que se fera la sélection de la méthode effective à appliquer. Les autres paramètres sont secondaires, et ne supportent qu'une redéfinition contravariante de leur type, pour respecter le sous-typage.

#### Sélection multiple

Le *Common Lisp Object System* (CLOS) a divergé de cette approche par envoi de message en permettant de faire la sélection de la méthode — la liaison tardive — suivant le type de plusieurs arguments, en fait de tous les arguments obligatoires.

Plusieurs arguments pouvant participer à la sélection, le privilège de l'objet receveur disparaît. Tous les arguments pouvant participer à la sélection, le problème de la contravariance de la redéfinition peut être évité. Mais bien entendu, ce problème de contravariance ne se pose pas en CLOS qui n'a pas de typage statique.

La sélection multiple a été reprise par plusieurs langages comme ILOGTALK, CECIL ou CLAIRE.

**Produit des types.** Techniquement, la sélection multiple se ramène à la sélection simple : il suffit de considérer le type produit. Ainsi, la sélection sur 2 paramètres  $(x : t_1, y : t_2)$  est équivalente à la sélection sur un seul paramètre  $(x, y) : t_1 \times t_2$ , le tuple  $(x, y)$  devenant le receveur de message effectif.

On notera donc  $m_{t_1 \times \dots \times t_n}() : u$ , une méthode de nom  $m$ , définie sur les types  $t_i, i \in [1, n]$ , avec un type de retour  $u$ .

Le sous-typage du type produit s'exprime ainsi :

$$t_2 \times u_2 \leq t_1 \times u_1 \iff t_2 \leq t_1 \ \& \ u_2 \leq u_1 \quad (3.2)$$

Ce passage aux types produits a un effet important : il introduit nécessairement des problèmes d'héritage multiple. En effet, le produit de deux ordres totaux n'est qu'un ordre partiel, comme le montre la figure 3.2.

**Méthodes binaires.** On appelle méthode binaire, une méthode qui prend un argument du même type que le receveur. Si  $B$  est une sous-classe de  $A$ , on voudrait redéfinir la méthode dans  $B$ , avec une redéfinition covariante du paramètre, pour qu'il soit de type  $B$ . L'exemple classique est le prédicat d'égalité. La sélection multiple est la seule façon de mettre en œuvre ce genre de méthode en évitant toute erreur de type.

### 3.8.2 Principe de la sélection multiple

**Sélection simple.** Le mécanisme de liaison tardive, c'est-à-dire d'envoi de message, consiste à sélectionner la méthode « la plus spécifique » suivant le type *dynamique* de l'objet receveur. Si ce type est  $t$ , le mécanisme va sélectionner une méthode attachée à un type  $u$  tel que :

$$t \leq u \text{ et } \nexists v \text{ tel que } t \leq v < u, \text{ avec une méthode attachée à } v \quad (3.3)$$

<sup>7</sup> Où *cat* signifie *Changing Availability or Type*.

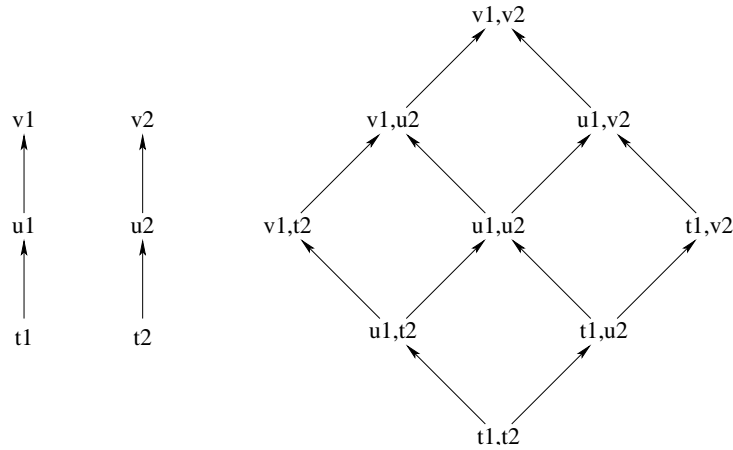


FIG. 3.2 – Produit de deux hiérarchies de types.

En héritage simple et sélection simple, il n’y a pas de problème. En héritage multiple et sélection simple,  $u$  n’est pas forcément unique : il peut donc y avoir des *conflits d’héritage* (cf. chapitre 4).

**Sélection multiple.** En sélection multiple, le principe de sélection est analogue. Si les types dynamiques des « objets receveurs », c’est-à-dire des arguments sur lesquels porte la sélection, sont  $t_1, t_2, \dots, t_n$ , le mécanisme va sélectionner une méthode attachée aux types  $u_1, u_2, \dots, u_n$ , tels que

$$\forall i, t_i \leq u_i \text{ et } \exists v_1, \dots, v_n \text{ tq } \forall i, t_i \leq v_i \leq u_i, \text{ et } \exists i : v_i < u_i, \text{ avec une méthode attachée à } v_1, \dots, v_n \quad (3.4)$$

En passant dans le produit des types, cela s’exprime par un objet receveur de type  $t_1 \times t_2 \cdots \times t_n$  et la sélection d’une méthode attachée à un type  $u_1 \times u_2 \cdots \times u_n$  tel que :

$$\begin{aligned} t_1 \times t_2 \cdots \times t_n &\leq u_1 \times u_2 \cdots \times u_n \text{ et} \\ \exists v_1 \times v_2 \cdots \times v_n \text{ tel que } t_1 \times t_2 \cdots \times t_n &\leq v_1 \times v_2 \cdots \times v_n < u_1 \times u_2 \cdots \times u_n, \\ &\text{avec une méthode attachée à } v_1 \times v_2 \cdots \times v_n \end{aligned}$$

Que l’héritage soit simple ou multiple, la sélection multiple introduit potentiellement tous les problèmes de l’héritage multiple : le type  $u_1 \times u_2 \cdots \times u_n$  n’est pas forcément unique : dans la figure 3.2, on peut par exemple définir deux méthodes, respectivement sur les types  $t_1 \times u_2$  et  $u_1 \times t_2$ .

### 3.8.3 Covariance et sélection multiple.

La sélection multiple, qui peut être vue comme un moyen de contourner le problème de contravariance du type des paramètres en faisant disparaître ces derniers, ne résout en rien la contradiction entre la spécialisation et le sous-typage. D’une part, elle est sans effet sur les attributs mais, d’autre part, son application à l’exemple animalier précédent n’empêcherait pas un phoque de manger de l’herbe : simplement, il le ferait avec la méthode des animaux et non avec celle des phoques.

### 3.8.4 Surcharge statique : une fausse sélection multiple.

Encore une fois, il ne faut pas confondre la surcharge, telle qu’elle peut exister en C++ ou JAVA, avec la sélection multiple. La surcharge conduit bien à une sélection suivant le type des arguments secondaires, mais cette sélection est statique, à la compilation. La liaison tardive ne s’applique qu’après, à l’exécution, sur l’unique receveur du message.

En revanche, dans la sélection multiple, la liaison tardive s’applique à tous les arguments sur lesquels porte la sélection.

**Surcharge statique correcte (JAVA).** Une analyse « conceptuelle » de la spécialisation et de l’héritage conduit à considérer que le nom des propriétés, telles qu’elles sont définies, héritées et utilisées, désigne une « entité générique » sous-jacente, invariante par héritage et redéfinition. C’est le point de vue illustré par CLOS avec la distinction entre *fonction générique* et *méthode*. Voir le chapitre 6 pour plus de détails sur ces entités génériques.

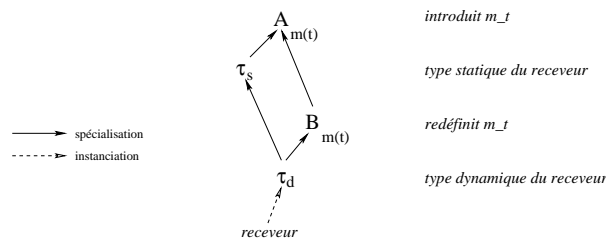


FIG. 3.3 – Surcharge statique et dynamique : les 4 classes impliquées dans la sélection.

Dans cette vision conceptuelle des objets, la surcharge statique est un phénomène d’ambiguïté du nommage des entités génériques, alors que la l’envoi de message et l’héritage sont des phénomènes relatifs à la même entité générique, dont les différentes occurrences (méthodes par exemple) se masquent ou sont à sélectionner. Dans ce cadre, il est donc possible de renommer les méthodes — par exemple en concaténant à leur nom les types des paramètres,  $m(t)$  devenant  $m\_t(t)$  et  $m(u)$   $m\_u(u)$  — pour faire disparaître la surcharge statique sans rien changer au comportement d’ensemble. Ce nouveau nommage étant non ambigu (injectif), il peut alors servir à identifier les entités génériques. Le renommage d’un appel de méthode particulier se fait bien entendu dans l’espace de noms du type statique du receveur.

Au total, on a donc une opération de sélection en 3 temps :

1. sélection de l’ensemble des entités de nom  $m$  du type *statique*  $\tau_s$  du receveur ;
2. sélection, dans cet ensemble, de l’entité générique  $m\_t$  la plus spécifique adaptée au type *statique* des arguments et introduite dans la classe  $A$  ;
3. sélection, à l’exécution et dans l’entité générique  $m\_t$ , de la méthode  $m\_t_B$  la plus spécifique correspondant au type *dynamique*  $\tau_d$  du receveur.

On voit que cette sélection met en jeu 4 types différents, totalement ordonnés,

$$\tau_d \leq B \leq \tau_s \leq A \tag{3.5}$$

avec, dans l’ordre : le type dynamique du receveur, la classe qui possède la méthode héritée, le type statique du receveur et le type qui introduit l’entité générique (figure 3.3).

**Ambiguïté dans la surcharge statique** La surcharge statique est résolue par une sélection statique qui consiste à choisir, comme dans le cas de la sélection dynamique, la propriété générique la plus spécifique compatible avec le type des paramètres.

En cas d’héritage multiple, ce choix de la plus spécifique peut conduire à des conflits (cf. chapitre 4). Mais la surcharge statique conduit à des ambiguïtés spécifiques dans le cas où il y a plusieurs arguments :  $m_A(t, u)$  et  $m_A(u, t)$  sont tous les deux plus spécifiques et ambigus lors d’un appel sur 2 paramètres de type  $u$ . Ce premier cas se ramène à un conflit d’héritage dans le produit des types (cf. 3.8.1). Devant de telles ambiguïtés statiques, le compilateur ne peut que générer une erreur.

Curieusement, JAVA signale une ambiguïté dans un cas similaire mais pourtant assez différent, puisqu’il concerne le receveur et le paramètre secondaire : il considère que  $m_A(u)$  et  $m_B(t)$  sont toutes les deux plus spécifiques pour un receveur de type  $B$  et un paramètre de type  $u$ . Mais cette ambiguïté est contestable : dans notre schéma de sélection,  $B$  associe au nom  $m$  aussi bien  $m\_t$  que  $m\_u$  (étape 1), parmi lesquels le type statique  $u$  d’un argument provoque la sélection de  $m\_u$  (étape 2) : cf. figure 3.6.

Le comportement de JAVA est précisé dans les spécifications du langage [GJS96]. Lors d’un appel de méthode  $x.m(arg)$ , le compilateur

1. collecte, dans le type statique du receveur  $x$  et ses super-types, toutes les méthodes de nom  $m$ , de même nombre de paramètres et dont le type est un super-type des types statiques des arguments,
2. élimine parmi ces méthodes celles qui ne sont pas accessibles (mots-clés `private` ou `protected`)<sup>8</sup>,
3. sélectionne les plus spécifiques, avec la définition suivante de la spécificité :  $m_B(u)$  est plus spécifique que  $m_A(t)$  ssi  $B <: A$  et  $u <: t$  ; il y a alors ambiguïté s’il y a plusieurs méthodes plus spécifiques de signatures différentes (le receveur n’intervenant pas dans la signature).

On voit que les propriétés génériques n’interviennent pas, toutes les méthodes étant mises sur le même plan. La sélection que nous avons proposée repose quant à elle sur la définition suivante de la spécificité :  $m_B(u)$  est plus spécifique que  $m_A(t)$  ssi  $u <: t$ .

<sup>8</sup> Le fait que la protection intervienne dans la sélection est en soi aberrant.

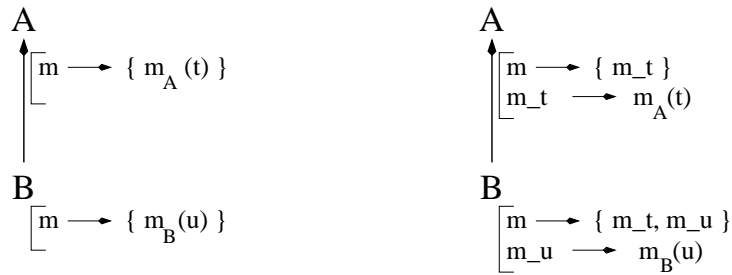


FIG. 3.4 – Surcharge statique. En C++ (à gauche), chaque nom est lié à un ensemble de méthodes surchargées et la spécialisation provoque un masquage des noms. En JAVA (à droite), chaque nom est lié à un ensemble d'entités génériques que la spécialisation cumule, le masquage ne s'effectuant qu'au niveau de chaque entité générique.

**Surcharge statique et `super`** En JAVA, la surcharge statique peut causer des interférences avec `super` : en effet, si  $m_A(t)$  et  $m_B(u)$  sont définis, comment faut-il comprendre `super.m` dans le corps de  $m_B(u)$  ? Faut-il le comprendre comme `super.m_t` ou `super.m_u` ?

La logique voudrait sans doute que `super` serve à étendre le code d'une méthode dans le cadre d'une même entité générique. Il faudrait donc comprendre `super.m_u`.

Mais en réalité, `super` permet de court-circuiter le mécanisme normal d'appel de méthodes, en appelant n'importe quelle méthode de la super-classe. Il faut en fait le comprendre comme `super.m_t` et cela s'explique si l'on considère que `super` a le type statique implicite de la super-classe, ici  $A$ . Notons en particulier que `super` n'a aucun rapport avec une coercition ascendante sur la super-classe.

**Surcharge statique et coercition ascendante** Il a été assuré précédemment que la coercition ascendante n'avait aucun effet, puisque c'était la manifestation naturelle du polymorphisme d'inclusion : une variable peut être liée à n'importe quelle valeur de son type statique, ou d'un sous-type.

Cependant, en cas de surcharge statique, la coercition ascendante n'est pas sans effet. Si  $m_A(t)$  et  $m_B(u)$  sont définis dans les conditions précédentes, le contexte suivant provoquera les sélections statiques indiquées en commentaires :

```
{ x : B
  y : u
  x.m(y)           // m_u
  (A)x.m(y)       // m_t
  x.m((t)y)       // m_t
}
```

On constate en particulier que la coercition du paramètre et celle du receveur ont le même effet.

**Surcharge en C++ (incorrecte).** Ce ne sont plus les entités génériques qui se masquent, mais les ensembles d'entités associés à chaque nom : le fait de définir dans la sous-classe une méthode de nom  $m$ , quel que soit le type de ses paramètres, rend inaccessibles les méthodes de même nom définies dans les super-classes. En C++, une classe n'est guère plus qu'un espace de noms : *Note that dominance* [i. e. masquage] *applies to names not just to functions* [ES90, p. 205]. Ce comportement entraîne une très mauvaise propriété d'extensibilité : le fait de redéfinir une méthode dans une sous-classe peut masquer les méthodes de même nom de la super-classe, rendant le code incompilable, ou pire, compilable mais avec un comportement imprévu.

La substituabilité peut aussi se comprendre comme la possibilité de substituer un sous-type  $t_2$  à un type  $t_1$  dans un programme sans en changer le comportement ou au moins la correction. Ce n'est pas toujours vrai, les contraintes de sous-typage dans les affectations ou passage de paramètres n'étant pas toujours respectées, mais c'est souvent vrai si l'on ne considère que des expressions élémentaires. Ainsi, en JAVA, si `x.foo(arg)` est correct quand le type de  $x$  est  $t_1$ , l'expression reste correcte lorsque le type de  $x$  est un sous-type  $t_2$ . En C++, ce n'est plus vrai, car le `foo` de  $t_1$  a pu être masqué par un autre `foo` dans  $t_2$ .

**Surcharge statique et renommage.** En principe, toute surcharge statique peut se résoudre par un renommage de l'une des deux propriétés en question, par exemple en concaténant le type des paramètres au nom de la fonction. Si l'on effectue ce renommage dans la totalité du programme (en supposant que les nouveaux noms ne sont pas déjà utilisés), sa sémantique doit rester inchangée. C'est bien vrai en JAVA, mais cela ne l'est plus en C++ à cause du masquage de noms que le renommage fait disparaître.

### 3.8.5 Surcharge d'attributs

La surcharge d'attributs consiste à maintenir plusieurs attributs physiquement différents dans une même classe, la sélection se faisant suivant le type statique du receveur. Le type de l'attribut n'a aucun effet dans la sélection : il est donc possible de surcharger un attribut sans changer son type.

### 3.8.6 Surcharge d'introduction

Outre la surcharge statique et la redéfinition (qui est souvent qualifiée de surcharge), on peut distinguer un autre sens de surcharge, commun dans les langages à typage dynamique comme SMALLTALK. Dans un tel langage, lorsqu'une méthode  $m$  est introduite dans 2 classes incomparables  $A$  et  $B$ , l'absence de typage statique ne permet pas de préciser que l'usage de  $m$  dans tel appel de méthode concerne les instances de  $A$  mais pas celles de  $B$ , et réciproquement : en pratique, le même appel de méthode pourra alternativement s'appliquer à un receveur de type  $A$  ou  $B$ . On appellera cela une *surcharge d'introduction*. Si elle est courante et quasiment obligatoire en typage dynamique, elle est aussi pratiquée par JAVA pour les interfaces. Voir aussi l'*héritage de nom*, paragraphe 4.2.

On peut d'ailleurs voir JAVA comme la transposition de SMALLTALK en typage statique : pour que le même appel de la méthode  $m$  puisse s'appliquer aux instances de  $A$  et de  $B$ , il faut que  $A$  et  $B$  ait un super-type commun qui introduise la méthode  $m$  : c'est le rôle des interfaces.

## 3.9 Problèmes divers

**Perte d'information.** Soit une classe  $A$ , possédant une méthode  $m_A()$  :  $A$ , qui retourne l'objet receveur ou une copie de celui-ci. Soit une sous-classe  $B$  de  $A$  qui possède une méthode supplémentaire,  $p_B()$ . Le code suivant va être refusé par un compilateur :

```
x : B
y : A
y := x.m()
y.p()
```

alors qu'il ne provoquera jamais la moindre erreur de type. En effet, le type statique de retour de  $m$  est  $A$  qui ne connaît pas  $p$ . Pourtant, le type dynamique sera toujours celui de  $x$ , donc un sous-type de  $B$ , qui connaît bien  $p$ . Ce problème est connu sous le nom de « perte d'information ».

La solution à ce problème consiste à permettre d'exprimer, dans le langage, le type dynamique d'un paramètre, en particulier du receveur du message. Ainsi, de même qu'il existe différents mots-clefs, suivant les langages, pour désigner le receveur du message — *self* en SMALLTALK, *this* en C++ ou JAVA, *current* en EIFFEL — certains langages rajoutent un mot-clef pour désigner le type du receveur du message — *mytype* par exemple — ou pour désigner le type de n'importe quelle autre entité — *like* en EIFFEL.

Dans l'exemple précédent, il faudrait donc déclarer  $m_A()$  : *mytype* ou, en EIFFEL,  $m_A()$  : *like current*.

## 3.10 Méta-modèle pour les propriétés

Le chapitre 6 peut être d'un grand secours pour comprendre ce qui précède, en particulier la distinction entre redéfinition et surcharge statique.

	$t$	$t[u]$	$u$		
$A$	$m_A(t)$	$m_A(t)$	$m_A(t)$	surcharge statique incorrecte	C++
$A[B]$	$m_A(t)$	$m_A(t)$	$m_A(t)$		
$B$	+−	+−	$m_B(u)$		
$A$	$m_A(t)$	$m_A(t)$	$m_A(t)$	surcharge statique correcte	JAVA
$A[B]$	$m_A(t)$	$m_A(t)$	$m_A(t)$		
$B$	$m_A(t)$	$m_A(t)$	$m_B(u)$		
$A$	$m_A(t)$	$m_A(t)$	$m_A(t)$	covariance avec typage statique	EIFFEL
$A[B]$	+*	$m_B(u)$	$m_B(u)$		
$B$	+−	−*	$m_B(u)$		
$A$	$m_A(t)$	$m_A(t)$	$m_A(t)$	covariance avec typage dynamique	
$A[B]$	+	$m_B(u)$	$m_B(u)$		
$B$	+	$m_B(u)$	$m_B(u)$		
$A$	$m_{A \times t}$	$m_{A \times t}$	$m_{A \times t}$	sélection multiple	CLOS
$A[B]$	$m_{A \times t}$	$m_{B \times u}$	$m_{B \times u}$		
$B$	$m_{A \times t}$	$m_{B \times u}$	$m_{B \times u}$		

FIG. 3.5 – La sélection des méthodes  $m_A(t)$  et  $m_B(u)$  suivant le modèle et les types statique et dynamique (ce dernier entre crochets, lorsqu'il diffère du type statique) du receveur et du paramètre, dans un contexte où  $B <: A$  et  $u <: t$  : un "−" indique une erreur statique (détectable à la compilation) et un "+" une erreur dynamique (qui se produirait si l'erreur n'est pas détectée statiquement). Le cas d'EIFFEL (\*) dépend du traitement des *catcalls* polymorphes.

	$t$	$t[u]$	$u$		
$A$	+−	−	$m_A(u)$	surcharge statique incorrecte	C++
$A[B]$	+−	−	$m_A(u)$		
$B$	$m_B(t)$	$m_B(t)$	$m_B(t)$		
$A$	+−	−	$m_A(u)$	surcharge statique correcte	JAVA
$A[B]$	+−	−	$m_A(u)$		
$B$	$m_B(t)$	$m_B(t)$	$m_A(u)*$		
$A$	+	$m_{A \times u}$	$m_{A \times u}$	sélection multiple	CLOS
$A[B]$	$m_{B \times t}$	$m_{B \times t} **$	$m_{B \times t} **$		
$B$	$m_{B \times t}$	$m_{B \times t} **$	$m_{B \times t} **$		

FIG. 3.6 – La sélection des méthodes  $m_A(u)$  et  $m_B(t)$ , dans le même contexte que la figure précédente : les cas d'EIFFEL et de la covariance ne sont pas décrits car une telle redéfinition non covariante est impossible. Dans le cas de JAVA (\*), une ambiguïté non justifiée est signalée. Dans le cas de CLOS (\*\*), il y a un conflit de valeur réglé par la linéarisation.

	$t$	$t[u]$	$u$		
$A$	$m_A(t)$	$m_A(t)$	$m_A(t)$	surcharge statique	C++ et JAVA
$A[B]$	$m_B(t)$	$m_B(t)$	$m_B(t)$		
$B$	$m_B(t)$	$m_B(t)$	$m_B(u)$		
$A$	$m_{A \times t}$	$m_{A \times t}$	$m_{A \times t}$	sélection multiple	CLOS
$A[B]$	$m_{B \times t}$	$m_{B \times u}$	$m_{B \times u}$		
$B$	$m_{B \times t}$	$m_{B \times u}$	$m_{B \times u}$		

FIG. 3.7 – Surcharge statique et sélection multiple, avec 3 méthodes  $m_A(t)$ ,  $m_B(t)$  et  $m_B(u)$ .

## Chapitre 4

# Héritage multiple

Le degré zéro de l'héritage s'exprime par l'inclusion des intensions de la formule (2.2) : la sous-classe hérite des propriétés de ses super-classes. Cette inclusion apparemment sans problème se complique dès que l'on y rajoute :

- le fait que les propriétés peuvent être redéfinies dans les sous-classes, ce qui provoque un phénomène de masquage ;
- le fait que les propriétés sont désignées par des *noms* qui peuvent être ambigus ;
- le fait que l'héritage multiple peut provoquer un héritage contradictoire de la même propriété ou de propriétés de mêmes noms.

### 4.1 La redéfinition et le masquage

Lorsqu'une classe  $B$  spécialise une classe  $A$  (noté  $B \prec A$ ), il est possible de définir dans  $B$  des propriétés nouvelles, que  $A$  ne possédait pas. Il est aussi possible de redéfinir des propriétés de  $A$ .

Pour reprendre les termes de la section 3.8.4, page 20, nous nous intéressons ici, exclusivement, à la redéfinition d'une propriété, c'est-à-dire d'une même *entité générique*, par opposition à la *surcharge statique* que nous avons déjà discutée.

On supposera donc une méthode  $m()$ , sans paramètre et de type de retour indifférent, définie dans  $A$  par  $m_A()$ , et redéfinie dans  $B$  par  $m_B()$ .

Le *masquage* est le phénomène par lequel la méthode  $m_B()$  de  $B$  semble « masquer » la méthode  $m_A()$ , pour toutes les instances  $B$ , directes ou indirectes : pour ces instances, tout se passe comme si  $m_A()$  n'avait pas été définie.

Dans ce contexte, la *super-méthode* (ou appel à `super` en SMALLTALK ou JAVA) est le mécanisme qui permet à la méthode masquante d'appeler la méthode masquée.

### 4.2 Héritage multiple et conflits de noms

Le fait d'avoir écarté la problématique de la surcharge statique ne libère pas de la question des noms : en effet, avec l'héritage multiple, une classe peut hériter de deux super-classes incomparables deux propriétés de mêmes noms. Ces propriétés sont-elles identiques ou différentes ? C'est la problématique de l'*héritage de nom*.

#### 4.2.1 Le motif du losange

Dans la suite, nous considérerons systématiquement un motif d'héritage traditionnel dit « en losange », dans lequel une classe  $A$  a deux sous-classes directes  $B$  et  $C$ , qui ont elles-mêmes une sous-classe commune  $D$ . On a donc :

$$D \prec B \prec A \quad \text{et} \quad D \prec C \prec A \quad (4.1)$$

Sur ces 4 classes, on peut définir 3 propriétés de nom  $m$ ,  $p$  et  $q$  réparties comme suit :

- $m$  est définie en  $B$  et  $C$  :  $m_B$  et  $m_C$  ;
- $p$  est définie en  $A$  et  $B$  :  $p_A$  et  $p_B$  ;
- $q$  est définie en  $A$ ,  $B$  et  $C$  :  $q_A$ ,  $q_B$  et  $q_C$ .

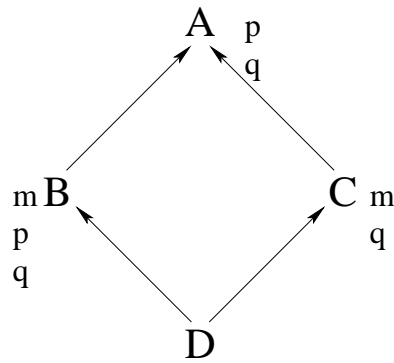


FIG. 4.1 – Héritage multiple : le schéma du losange exhibe une triple situation, de masquage pour  $p$ , de conflit de nom pour  $m$  et de conflit de valeur pour  $q$ .

La figure 4.1 résume la situation.

On pourra, pour fixer les idées, poser les interprétations suivantes :

- $A =$  Personne
- $B =$  Etudiant
- $C =$  Salarié
- $D =$  Etudiant-Salarié

#### 4.2.2 La problématique de l'héritage de nom

La question est alors la suivante : combien  $D$  (et ses instances) ont-ils de propriétés de noms  $m$ ,  $p$ ,  $q$  ? Subsidiatement, comment  $D$  peut-il redéfinir ces propriétés ? comment peut-on y accéder sur des instances de  $D$  ?

Deux principes apparemment « naturels » peuvent guider une réponse :

- un principe d'*invariance par héritage et redéfinition* : un nom désigne une entité générique et, lorsque l'on redéfinit une entité générique, il s'agit bien de la même : en l'occurrence, le  $p$  de  $B$  est bien le même que le  $p$  de  $A$ , de même que les  $q$  de  $B$  et de  $C$  sont bien les mêmes que ceux de  $A$  ; par conséquent, les  $q$  de  $B$  et de  $C$  sont bien les mêmes ;
- un principe d'*indépendance des premières définitions* : lorsqu'une classe définit une propriété inconnue de ses super-classes, cette propriété est *a priori* sans rapport avec toute autre propriété de même nom déjà définie dans des classes incomparables : en l'occurrence, il n'y a aucune raison que les  $m$  de  $B$  et de  $C$  aient un quelconque rapport.

Dans l'exemple,  $D$  devrait donc avoir deux propriétés de nom  $m$ , mais un seul  $p$  et un seul  $q$ .

#### 4.2.3 Conflit de nom

Pour fixer les idées, on aurait pu prendre pour ces propriétés l'interprétation suivante :

- $m =$  Département
- $p =$  RégimeSécu

Toute personne a un seul régime de sécurité sociale, mais un étudiant-salarié dépend de départements différents suivant qu'il est considéré comme un salarié ou comme un étudiant. Il s'agit là d'un *conflit de nom* : le département des salariés aurait très bien pu s'appeler *service* et le problème ne se serait jamais posé.

Informellement, la définition d'un *conflit de nom* est donc la suivante : il y a un conflit de nom lorsque qu'une classe (ici  $D$ ) hérite de deux propriétés de mêmes noms (ici  $m$ ), définies dans deux classes incomparables (ici  $B$  et  $C$ ).

Formellement, on définit l'*ensemble de conflit de nom* comme l'ensemble des super-classes de la classe considérée ( $D$ ), maximales (au sens de la spécialisation) à posséder une propriété du nom considéré ( $m$ ). Il y a conflit si cet ensemble n'est pas réduit à un singleton.

#### 4.2.4 Solution par désignation explicite ou par renommage

La solution du conflit de nom se trouve dans un renommage non ambigu des entités génériques, similaire à celui qui a été fait pour la surcharge, mais en concaténant la classe de définition et non le type des paramètres<sup>1</sup>

<sup>1</sup> SI l'on veut traiter à la fois la surcharge statique et le conflit de nom, il faut composer les 2 renommages, quelque-chose comme  $m\_A\_t$ .

classe	objet						
A		A					
B		A	B				
C		A	C				
D		A	B	C	D	normal (avec virtual)	
D		A	B	A	C	D	répété (sans virtual)

FIG. 4.2 – L’implémentation des objets avec un héritage normal ou répété.

(section 3.8.4).

Dans l’exemple du losange, on aurait ainsi les 4 entités génériques :  $p_A$ ,  $q_A$ ,  $m_B$  et  $m_C$ . Ce renommage par *désignation explicite* est proche de la notation  $::$  utilisée en C++ pour accéder à une propriété précise en court-circuitant l’héritage. En C++, on aurait ainsi  $A :: p$ ,  $A :: q$ ,  $B :: m$  et  $C :: m$ . Mais la notation de C++ permet aussi  $B :: p$  ou  $C :: q$ , et son sens est fondamentalement différent puisqu’il ne s’agit plus là d’un conflit de nom mais d’un moyen d’éviter la liaison tardive par un appel statique.

La nécessité d’une désambiguïsation intervient alors dans deux cas :

- pour la redéfinition d’une propriété de nom  $m$  en  $D$  (ou dans ses sous-classes),
- pour son utilisation sur une entité de type statique  $D$  (ou un sous-type), ou en typage dynamique.

Dans ces deux cas il faudrait donc préciser s’il s’agit de  $m_C$  ou de  $m_B$  et disposer soit d’une notation de ce type, soit d’un renommage explicite. Dans les autres cas, aucune précision supplémentaire n’est nécessaire : le nom  $m$  suffit.

#### 4.2.5 Arcs de transitivité

Dans la spécialisation, un arc de transitivité serait le fait qu’une classe  $C$  soit déclarée sous-classe directe de deux classes  $A$  et  $B$ , alors que  $B < A$ .

Conceptuellement, un arc de transitivité n’apporte rien. Pourtant, plusieurs langages ont des traits qui leur donnent un sens pratique. En C++, il est possible d’hériter plusieurs fois d’une même classe dès lors que le mot-clé `virtual` n’est pas utilisé. En Eiffel, il est aussi possible d’hériter plusieurs fois d’une même classe, dès lors que tous les renommages nécessaires sont faits. Nous ne considérerons pas ces cas, qui témoignent d’une complexification inutile.

### 4.3 Héritage multiple et conflit de valeurs

Une fois que les éventuels conflits de noms ont été résolus, les problèmes ne sont pas encore éliminés. Ainsi, dans l’exemple de la figure 4.1, pour les propriétés de noms  $p$  et  $q$  qui ne présentent pas d’ambiguïté puisqu’elles sont définies en  $A$ , la question se pose de savoir quelle valeur va être héritée en  $D$  ?

Pour la propriété  $p$ , la première définition en  $A$  est redéfinie (donc masquée) en  $B$ . Pour tous les  $B$  (càd pour toutes les instances de  $B$ ) la redéfinition de  $p$  en  $B$  devrait donc masquer la définition en  $A$ . Ce masquage vaut en particulier pour  $D$ .

Pour la propriété  $q$ , le problème est que la définition en  $A$  est masquée à la fois par une redéfinition en  $B$  et par une autre en  $C$ . Aucune de ces 2 redéfinitions ne masque l’autre, les classes  $B$  et  $C$  étant incomparables. Manifestement, les classes  $B$  et  $C$  présentent ce qu’il faut bien appeler un *conflit de valeur*.

Formellement, on définit l’*ensemble de conflit de valeur* comme l’ensemble des super-classes de la classe considérée ( $D$ ), minimales (au sens de la spécialisation) à posséder la propriété considérée ( $m$ ). Il y a conflit si cet ensemble n’est pas réduit à un singleton.

### 4.4 Les solutions proposées par les différents langages

Aucun langage ne dispose du nommage non ambigu défini plus haut. Nous considérerons juste C++, Eiffel et CLOS.

#### 4.4.1 Héritage multiple en C++

Rappelons d’abord que le conflit de nom repose, en C++, sur une identité des noms et des types de paramètres : si les types de paramètres sont différents, il s’agit d’une surcharge statique.

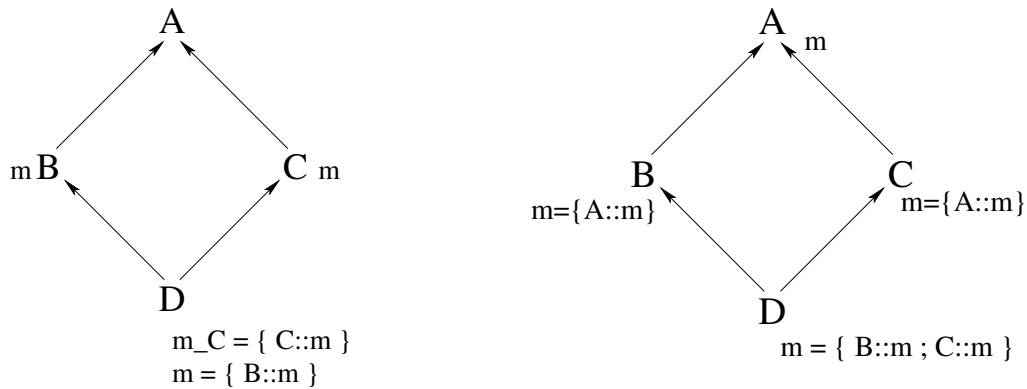


FIG. 4.3 – Héritage multiple en C++ : bricolage pour le conflit de nom (gauche) et héritage répété pour le conflit de valeur (droite).

Par défaut, C++ traite mal l'héritage de nom : en effet, seule la présence du mot-clé `virtual` dans la définition des classes `B` et `C` permet d'éviter un *héritage répété* de `A` par `D`, c'est-à-dire la duplication des attributs de `A` dans une instance de `D` (figure 4.2).

Par ailleurs, la notation `::` permet aussi bien de résoudre un conflit de nom qu'un conflit de valeur, mais aussi de résoudre l'appel à la *super-méthode* (appel à `super` de SMALLTALK ou `call-next-method` de CLOS), et, plus généralement, de faire appel à une propriété précise en court-circuitant tout mécanisme de sélection dynamique.

Lorsqu'elle est utilisée pour la *super-méthode*, la notation `::` a l'inconvénient de provoquer un *héritage répété* pour les méthodes, à cause d'une double évaluation : ainsi, dans la situation du losange de la figure 4.1, si une méthode `r` est définie dans chacune des 4 classes de telle sorte que `rB` et `rC` fassent appel à `A::r` et que `rD` fasse appel à `B::r` et `C::r`, alors le code de `rA` sera exécuté deux fois sur les instances de `D` (figure 4.3 droite).

En pratique, face à un conflit de nom, le programmeur C++ est relativement désarmé. S'il définit la classe `D`, la redéfinition de `m` vaudra autant pour `mB` que pour `mC` : il y a peu de chance que cela satisfasse la sémantique des deux méthodes `m`. Et s'il ne redéfinit pas `m`, le compilateur détectera un conflit. S'il utilise la classe `D` comme type statique, il ne peut pas sélectionner l'une ou l'autre méthode : un `cast` du receveur vers `B` ou `C` n'aura aucun effet puisque les deux méthodes sont redéfinies à l'identique dans `D`.

Une solution partielle consisterait à effectuer un renommage dans `D` de l'une des deux méthodes, par exemple `mC` en `m_C`, qui appellerait `C::m`. La redéfinition de `m` en `D` serait une redéfinition de `mB`, dont le comportement serait normal (figure 4.3 gauche). Pour `mC`, sur un receveur de type statique `D`, on utilise `m_C`. Mais le comportement est incorrect lorsque l'on appelle `mC` sur un receveur de type statique `C` mais de type dynamique `D` : c'est `mB`, redéfinie en `D` qui sera appelée. Et il n'y a aucun moyen de savoir, dans le code de la méthode, sur quel type de receveur elle a été appelée.

#### 4.4.2 Héritage multiple en EIFFEL

Modulo une certaine lourdeur dans les définitions, c'est EIFFEL qui se rapproche le plus du modèle présenté plus haut, en ce qui concerne l'héritage de nom du moins. En cas de conflit de nom, un renommage est imposé dans la déclaration de la classe.

Pour l'héritage de valeur, EIFFEL impose une désignation explicite lors de la redéfinition d'une propriété, même dans le cas de la propriété `p` qui ne présente pas formellement de conflit de valeur : le masquage n'est jamais implicite.

De plus, en cas de motif en losange, EIFFEL autorise un héritage répété sélectif qui est à la fois peu fondé sémantiquement et compliqué à utiliser.

#### 4.4.3 Héritage multiple en JAVA

Le titre peut paraître paradoxal puisqu'il est bien connu que JAVA a un héritage de classes simple. Mais son sous-typage est multiple : une classe ne peut spécialiser qu'une autre classe (mot-clé `extends`) mais elle peut implémenter (mot-clé `implements`), en plus, plusieurs interfaces. Et une interface peut spécialiser (mot-clé `extends`) plusieurs interfaces. Les interfaces ne déclarant que des signatures, seul l'héritage de nom est multiple en JAVA.

Curieusement, l'héritage de nom de JAVA ressemble plus à celui des langages à typage dynamique qu'à celui des langages à typage statique : en cas de conflit de nom, les noms — c'est-à-dire les signatures — désignent tous

la même propriété, ce qui est un cas de *surcharge d'introduction* commun en SMALLTALK (cf. 3.8.6). On peut voir le système de types de JAVA comme le système de types minimal permettant un typage statique de SMALLTALK tout en conservant son héritage simple.

#### 4.4.4 Héritage multiple en CLOS

En CLOS le problème de l'héritage de nom est résolu par une identification des propriétés en conflit : il n'est pas possible de distinguer deux propriétés de même nom (c'est-à-dire attributs, on verra le problème des méthodes au chapitre 8).

Pourtant, les attributs et les méthodes sont réifiés, mais les premiers ne constituent pas *entités génériques* : seul le nom est invariant par héritage. Quant aux méthodes, elles sont regroupées dans des entités génériques appelées *fonctions génériques*, qui restent identifiées à leur nom.

Enfin, pour l'héritage de valeur, CLOS est basé sur un mécanisme original à base de *linéarisation* (section suivante) et la *super-méthode* s'implémente par la fonction `call-next-method` qui fait appel à la méthode suivante dans la linéarisation. Cette approche évite l'héritage répété causé par la notation `::` de C++.

### 4.5 Techniques de linéarisation

Les techniques de linéarisation sont à la base du traitement de l'héritage multiple en CLOS et dans la plupart des langages à objets basés sur LISP. Elles sont aussi utilisées, de façon invisible, en C++, pour les *constructeurs* et *destructeurs*.

#### 4.5.1 Principe des linéarisations

En héritage simple, la recherche de la propriété héritée par une classe — recherche appelée *lookup* en SMALLTALK — est simple : elle consiste à chercher la propriété dans la classe, puis récursivement dans sa super-classe, jusqu'à trouver la propriété (réussite) ou à atteindre une classe sans super-classe (échec). Comme l'héritage est simple, l'ensemble des super-classes d'une classe est totalement ordonné et forme un chemin du graphe d'héritage.

##### Extension linéaire

Lorsque l'héritage est multiple, cette dernière propriété n'est plus vraie. La linéarisation est une généralisation de cette technique à l'héritage multiple, consistant à parcourir l'ensemble des super-classes, suivant un ordre total, à la recherche de la propriété.

Le problème est celui de la définition de cet ordre total, qui est appelé *linéarisation*. En héritage simple, la recherche est basé sur le *masquage*, qui fait que l'on recherche la classe unique la plus spécifique qui possède la propriété cherchée. En héritage multiple, on va généraliser en cherchant une des classes les plus spécifiques : on respectera le masquage en prenant une classe non masquée.

La solution est un problème algorithmique bien connu : l'ensemble des super-classes forme un ordre partiel et l'on recherche un ordre total qui soit compatible avec cet ordre partiel, c'est-à-dire qui le contienne. Un tel ordre total s'appelle une *extension linéaire*. La recherche suivant une extension linéaire retournera toujours un élément de l'ensemble de conflit de valeur, donc une classe qui n'est masquée par aucune autre.

##### Ordre local de priorité

Dans l'exemple du losange, on a ainsi 2 extensions linéaires possibles :  $\{D, B, C, A\}$  et  $\{D, C, B, A\}$ . On voit bien que ces ordres induisent un ordre sur les super-classes directes de toute classe : ici  $\{B, C\}$  ou  $\{C, B\}$  sur les super-classes directes de  $D$ .

Le principe va donc être de se servir de cet ordre sur les super-classes directes d'une classe — appelé *ordre local de priorité* — pour spécifier un peu mieux le choix de la linéarisation : on dira par exemple que  $D$  hérite d'abord de  $B$  avant d'hériter de  $C$ .

On cherchera donc un ordre total

- qui respecte le *masquage*, donc qui soit une *extension linéaire* de la relation de spécialisation ;
- qui respecte cet *ordre local de priorité*.

Ces deux contraintes ne sont pas forcément compatibles : l'ordre local n'est pas forcément sans circuits puisqu'il est la réunion de plusieurs ordres totaux indépendants (figure 4.4). De plus, même si cet ordre local est sans circuits, sa réunion avec la relation de spécialisation n'est pas forcément sans circuits.

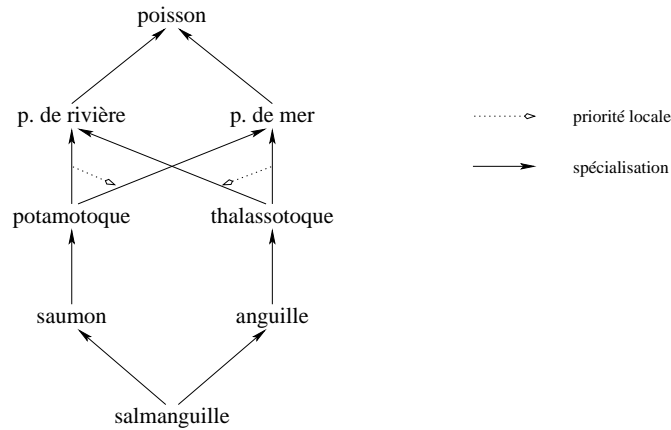


FIG. 4.4 – Circuits dans l'ordre de priorité local : les potamotoques se reproduisent dans les fleuves et vivent dans la mer, alors que les thalassotoques font l'inverse. On peut représenter cela par des ordres locaux inverses, mais avec les progrès de la génétique, le pire est à craindre.

## 4.5.2 Quelques linéarisations

Nous allons voir deux linéarisations : la plus simple qui est utilisée par C++ (et introduite auparavant par COMMON LOOPS), et celle de CLOS qui est un peu moins simple.

### Linéarisation de C++ et COMMON LOOPS

Soit une classe  $A$ , de super-classes directes  $B_1, \dots, B_n$ . On note  $L(C)$  la linéarisation d'une classe  $C$ . La linéarisation de  $A$  est définie récursivement à partir des linéarisations des  $B_i$ , en deux étapes :

1. on concatène les linéarisations des  $B_i$ , en rajoutant  $A$  devant :

$$L = A, L(B_1), \dots, L(B_n)$$

2. dans la liste  $L$ , on enlève les doublons en conservant la dernière occurrence : si  $L = C_1, \dots, C_n$ , si  $C_i = C_j$ , pour tout  $i, j$  avec  $i < j$ , on enlève  $C_i$ .

$L(A)$  est la liste résultant de cette élimination.

Ainsi, dans la figure du losange, en supposant que  $\{B, C\}$  est l'ordre de priorité local des super-classes de  $D$ , on a<sup>2</sup> :

$$\begin{aligned} L(A) &= A \\ L(B) &= B, A \\ L(C) &= C, A \\ L(D) &= D, B, \underline{A}, C, A \end{aligned}$$

Cette linéarisation a deux propriétés :

- c'est une extension linéaire de la relation de spécialisation (exercice : le démontrer) ;
- elle est compatible avec l'ordre local, sauf dans un cas de figure (exercice : lequel ?).

### Linéarisation de CLOS

Elle est basée sur le calcul d'une extension linéaire de la réunion des relations de spécialisation et de l'ordre local de priorité, lorsque c'est possible. Si ce n'est pas le cas, une erreur est déclenchée. Elle respecte donc toujours les deux propriétés recherchées.

Un algorithme général naïf de calcul d'une extension linéaire est le suivant : tant qu'il reste des sommets, prendre un sommet minimal dans la relation. Les extensions linéaires diffèrent suivant la façon dont on choisit un minimal lorsqu'il y en a plusieurs.

La linéarisation de CLOS se définit par un algorithme qui précise ce choix : lorsqu'il y a plusieurs minimaux (suivant la réunion des 2 relations), on prend celui qui est le successeur, dans la relation de spécialisation, de la classe la plus récemment prise.

Supposons que l'on ait déjà pris  $k$  classes, dans l'ordre,  $C_1, \dots, C_k$  et qu'il y ait  $m$  minimaux  $M_1, \dots, M_m$  dans les classes restantes. On va considérer les couples  $(C_j, M_i)$  tels que  $C_j \prec M_i$ , et on prendra le  $M_i$  qui maximise  $j$ , donc tel qu'il n'existe pas  $(C_{j'}, M_{i'})$  avec  $j' > j$ .

<sup>2</sup> En souligné, les doublons enlevés.

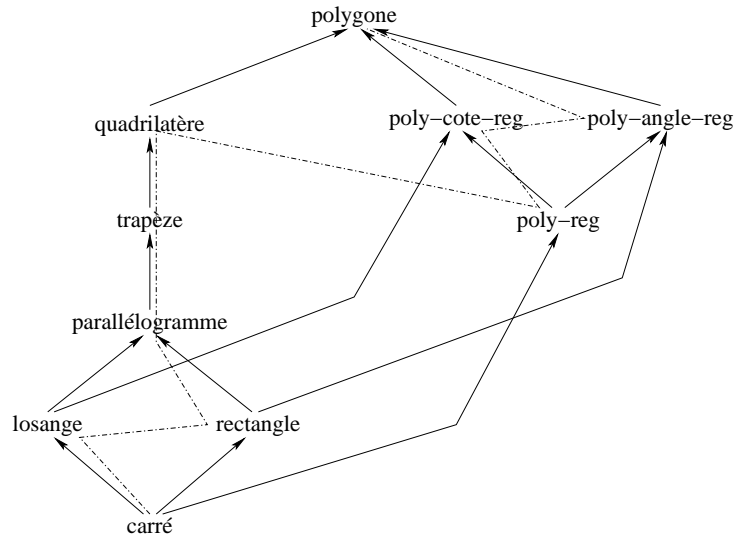


FIG. 4.5 – Les polygones : la classe carré.

### Exemple : les polygones

Soit la hiérarchie de classes de la figure 4.5, modélisant des polygones et en particulier la classe carré. On suppose que l'ordre local est de gauche à droite dans l'origine des arcs.

La première linéarisation donne :

$$\begin{aligned}
 L(P) &= P \\
 L(Pcr) &= Pcr, P \\
 L(Par) &= Par, P \\
 L(Pr) &= Pr, Pcr, \underline{P}, Par, P \\
 L(Q) &= Q, P \\
 L(T) &= T, Q, P \\
 L(Pa) &= Pa, T, Q, P \\
 L(L) &= L, Pa, T, Q, \underline{P}, Pcr, P \\
 L(R) &= R, Pa, T, Q, \underline{P}, Par, P \\
 L(C) &= C, L, \underline{Pa}, \underline{T}, \underline{Q}, \underline{Pcr}, \underline{P}, R, Pa, T, Q, \underline{Par}, \underline{P}, Pr, Pcr, Par, P
 \end{aligned}$$

soit  $L(C) = \{C, L, R, Pa, T, Q, Pr, Pcr, Par, P\}$ .

La linéarisation de CLOS donne le calcul suivant :

- on prend  $C, L, R$  minimaux uniques successifs,
- deux minimaux se présentent alors,  $Pa$  et  $Pr$  : on prend  $Pa$ , super-classe directe du dernier pris, puis, pour la même raison,  $T$  et  $Q$ ,
- on prend ensuite  $Pr, Pcr, Par, P$ , tous minimaux uniques successifs.

soit  $L(C) = \{C, L, R, Pa, T, Q, Pr, Pcr, Par, P\}$ .

Les deux linéarisations donnent le même résultat sur cet exemple.

### 4.5.3 Application à la sélection multiple

Le principe de linéarisation s'étend simplement à la sélection multiple de CLOS si l'on considère qu'il s'agit de sélection simple dans le produit des types (figure 3.2). Comme la hiérarchie d'origine est elle-même en héritage multiple, il va s'agir de linéariser le produit des linéarisations. Opérationnellement, cela va se traduire comme suit : lors de l'appel ( $f \circ e_1 \dots e_n$ ), si chaque  $e_i$  a une linéarisation  $(c_i^j, j = 1..k_i)$ , la méthode sélectionnée sera la première définie sur les tuples de classes dans l'ordre

$$(c_1^1, c_2^1, \dots, c_n^1), (c_1^1, c_2^1, \dots, c_n^2), \dots, (c_1^1, c_2^1, \dots, c_n^{k_n}), \dots, (c_1^{k_1}, c_2^{k_2}, \dots, c_n^{k_n}).$$

## 4.6 De l'héritage multiple à l'héritage simple

Il peut être nécessaire de traduire une hiérarchie d'héritage multiple en héritage simple. Cela peut arriver en portant un logiciel d'un langage disposant de l'héritage multiple dans un langage n'en disposant pas, ou lors d'un passage d'un modèle d'analyse à un modèle d'implémentation. Plusieurs cas sont à considérer.

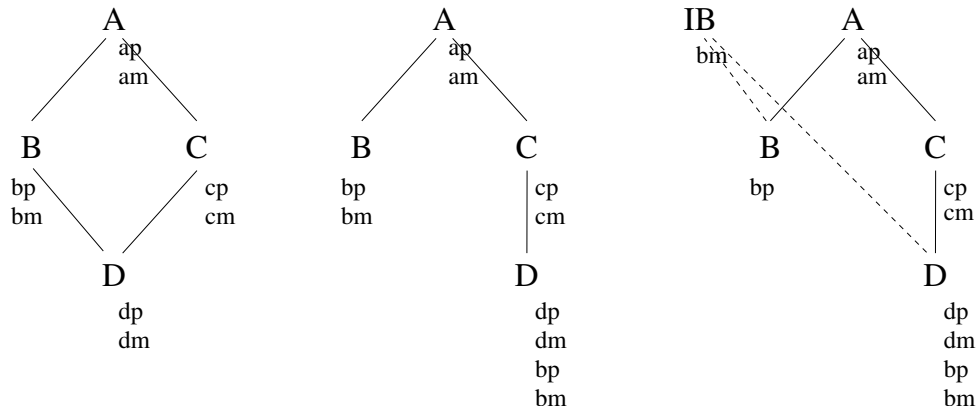


FIG. 4.6 – Transformation d’héritage multiple à simple.

#### 4.6.1 En sous-typage simple

La solution consiste à casser les cycles en supprimant tous les liens à une super-classe, sauf un. La hiérarchie obtenue est donc un arbre recouvrant de la hiérarchie de départ (figure 4.6, centre).

Lorsqu’un lien est coupé, les méthodes et attributs de la super-classe ( $B$ ) doivent être recopiés dans la sous-classe ( $D$ ) : la perte de factorisation est donc importante.

Mais la perte de sémantique est plus préoccupante : les instances de la sous-classe ( $D$ ) ne sont plus instances de la super-classe ( $B$ ). De plus, comme le sous-typage est calqué en général sur la spécialisation,  $D$  n’est plus un sous-type de  $B$ . La perte de réutilisation est donc importante.

#### 4.6.2 En sous-typage multiple

En JAVA par exemple, la présence des interfaces permet de compenser une partie des défauts (figure 4.6, droite). Lorsqu’on coupe un lien d’héritage, on associe à la super-classe ( $B$ ) une interface ( $IB$ ) qui introduit les méthodes introduites par  $B$ . Et la sous-classe  $D$  implémente cette interface. Il ne reste plus qu’à remplacer partout dans le code, le type  $B$  par  $IB$  pour qu’il soit toujours possible d’utiliser des instance de  $D$  là où on attendait des instances de  $B$ .

Bien entendu, il faut toujours copier le code des méthodes de  $B$  dans  $D$ .

#### 4.6.3 En typage dynamique

En typage dynamique, on se trouve dans le même cas qu’en sous-typage multiple, sans l’obligation de définir des interfaces.

### 4.7 Méta-modèle pour les propriétés

Le chapitre 6 peut être d’un grand secours pour comprendre ce qui précède.

# Chapitre 5

## Réflexivité

### 5.1 Introduction

#### 5.1.1 Du méta à la réflexivité

##### Aristote et la métaphysique.

Rappelons que *meta* est une préposition grecque ( $\mu\epsilon\tau\acute{\alpha}$ ) qui signifie « avec » ou « après » selon qu'elle gouverne le génitif ou l'accusatif. Le sens qu'elle prend dans « métaphysique » vient de ce que, dans l'ordre traditionnel des œuvres d'Aristote, les livres qui traitaient de la « philosophie première » venaient après les livres de physique —  $\mu\epsilon\tau\acute{\alpha}\ \tau\acute{\alpha}\ \phi\upsilon\sigma\iota\kappa\acute{\alpha}\ \beta\iota\beta\lambda\iota\acute{\alpha}$ .

J. F. Perrot, [DEMN98]

**Méta-langage.** Le terme de « méta-langage » désigne de façon usuelle, aussi bien en linguistique qu'en logique, un langage dont l'objet est un autre langage<sup>1</sup>.

Ainsi, la logique distingue classiquement (voir [Kle71]) :

- son langage, par exemple celui de la logique des propositions dont les formules «  $A \ \& \ B$  » et «  $A \rightarrow B$  » font partie ;
- les méta-langages qui permettent de parler de ces formules, qu'ils soient informels, par exemple « la formule  $A \ \& \ B$  est vraie », ou formels, par exemple la formule du *modus ponens* dans le formalisme de la déduction naturelle :

$$\frac{A \quad A \rightarrow B}{B} \quad (5.1)$$

qui signifie, dans ce formalisme, que si l'on a  $A$  et  $A \rightarrow B$ , alors on a  $B$ .

**Méta-niveau.** De façon générale, le préfixe « méta » a ainsi pris le sens de « système formel » (au sens le plus informel du terme) dont l'objet est un autre système formel, relativement du même type. On parlera de « niveau » plutôt que de « système formel » et on dira souvent que le premier, le méta-niveau, est « au-dessus » du second, son objet.

**Stratification.** Bien entendu, rien n'empêche que le méta-niveau soit lui-même l'objet d'un méta-méta-niveau — comme le méta-évaluateur LISP du module *Langage, Évaluation, Compilation* de Maîtrise l'a montré — aboutissant ainsi à une pile illimitée de niveaux dont chacun, sauf un, est le niveau méta de celui du dessous. On obtient ainsi une stratification.

**Réflexivité.** La *réflexivité* — on parle aussi de *méta-circularité* — consiste à limiter d'une façon ou d'une autre cette pile de niveaux, en faisant en sorte que l'un des méta-niveaux soit identique à, ou plongé dans, l'un des niveaux inférieurs.

A la limite, on n'a plus qu'un seul niveau, qui est ou contient son propre méta-niveau.

<sup>1</sup> Le lecteur avisé aura remarqué que l'ensemble de ce paragraphe relève d'un méta-méta-langage ! Je rajouterais bien que cette note relève, elle, d'un méta-méta-méta... si le vertige ne me prenait.

### 5.1.2 La réflexivité dans les langages de programmation

**Objets de première classe.** La littérature sur LISP, en particulier COMMON LISP [Ste90], a introduit le terme d'« objet de première classe » pour désigner les entités des langages de programmation qui sont accessibles aux programmeurs autrement qu'à travers la syntaxe du langage : ainsi, en LISP, les symboles et les fonctions sont des objets de première classe. D'autres entités comme les échappements ne le sont pas.

Un objet de première classe est une entité pour lesquelles il existe des valeurs et un type, ainsi qu'une interface fonctionnelle (ou API) permettant de manipuler ces valeurs.

**Réification.** La réification est le procédé qui consiste, dans un processus de modélisation ou de conception, à modéliser, représenter ou implémenter une entité abstraite par un objet.

La réification peut s'appliquer, en particulier, aux objets de première classe d'un langage. En fait, dans un contexte objet, les objets de première classe seront en général réifiés. Le terme d'« objet de première classe » est d'ailleurs révélateur de ce que son contexte d'utilisation n'était pas objet, mais fonctionnel.

**Classes et méta-classes.** Dans le contexte du modèle objet standard, avec des classes et des objets, tel qu'il a été présenté jusqu'ici, la première entité susceptible d'être réifiée est la classe. Il faut donc des classes de classes : sans surprise, on appelle ça des *méta-classes*.

On voit alors bien comment s'opère la boucle de réflexivité : les classes sont des objets comme les autres, juste un peu spécifiques, et les méta-classes des classes comme les autres, juste un peu spécifiques car leurs instances sont des classes.

**Introspection et réflexion.** A ce stade, on peut distinguer deux mises en œuvre possibles d'un méta-niveau dans un contexte objet.

Le premier, qui est qualifié d'*introspection*, permet au langage d'examiner les structures de données qui l'implémentent : c'est le cas du *package reflect* de JAVA.

Dans le second, qui mérite seul le terme de *réflexivité*, l'accès aux structures de données qui implémentent les entités n'est plus seulement passif mais plus ou moins totalement actif : il devient possible de créer de nouvelles entités et de modifier ou de spécialiser les entités existantes, en leur associant des comportements plus spécifiques. A la limite, le langage est implémenté en lui-même.

**Méta-objets.** D'autres entités du langage qui participent à la description des classes, donc des objets, peuvent aussi être réifiées : méthodes, attributs, etc. Ce seront des *méta-objets*.

### 5.1.3 Propriétés de classes, propriétés d'instances

Les classes devenant des objets ont des propriétés — méthodes ou attributs (variables) — comme les autres objets. Il devient donc nécessaire de distinguer soigneusement les propriétés d'un objet des propriétés de sa classe.

#### Attributs d'instances et attributs de classes

**Attribut d'instance alloué dans l'instance.** Les attributs (ou variables) d'instances sont implémentés physiquement dans chaque objet : leur valeur peut donc varier d'un objet à l'autre. On appellera un tel attribut, *attribut d'instance*.

**Attribut d'instance alloué dans la classe.** Lorsque la valeur de l'attribut est constante sur toutes les instances de la classe, il peut être commode et efficace de ne plus implémenter l'attribut dans chaque instance, mais de le factoriser en l'implémentant dans la classe. On appellera un tel attribut, un *attribut d'instance alloué dans la classe*. Par opposition, un attribut d'instance normal est *alloué dans l'instance*.

Dans ce cas, ce sont, comme les méthodes, des propriétés « de classe » destinées aux instances : il est donc indispensable de leur assurer une sélection par liaison tardive.

**Attributs d'instance de la classe.** Les classes étant des objets comme les autres, il est donc possible de définir des attributs d'instance de la classe, qui ne concernent en rien les instances de la classe : par exemple, l'auteur de la classe, sa date de modification ou la liste de ses instances.

Ces attributs ne sont pas directement accessibles depuis une instance de la classe.

**Attributs de classe.** La notion d'attribut de classe est donc problématique : il peut s'agir d'un attribut des instances alloué dans la classe, ou d'un attribut d'instance de la classe.

CLOS permet bien de différencier les deux mais SMALLTALK propose seulement le second<sup>2</sup>.

**Attribut de classe *ad hoc*.** Enfin, on a couramment besoin de données, souvent constantes, pour une classe (et toutes ses méthodes), sans que ces données puissent être assimilées à un attribut d'instance, ni de la classe, ni de l'instance : ainsi, par exemple, la classe `Date` nécessite des données comme la liste des noms des jours de la semaine ou des noms des mois<sup>3</sup>. On appellera ces attributs des *attributs de classes ad hoc* : ils ne relèvent pas directement du modèle objet mais d'un modèle de langage de programmation plus traditionnel, qui en fait des variables locales à une classe (éventuellement à ses sous-classes) et partagées par ses méthodes.

On pourra bien sûr implémenter ces attributs *ad hoc* comme des attributs d'instance de la classe ou comme des attributs d'instance alloués dans la classe, mais il s'agit à l'évidence d'un truc d'implémentation. En CLOS, on pourrait utiliser des fermetures pour obtenir le même genre de fonctionnalité.

**Variables statiques en C++ ou JAVA.** En C++ ou en JAVA, ces diverses versions d'attributs de classes sont implémentées, indifféremment, par des variables *statiques*, qui n'assurent malheureusement pas de liaison tardive.

Notons à ce propos que le vocabulaire commun désigne les attributs comme des *variables d'instances*, au moins dans les langages SMALLTALK, C++ et JAVA. Ce terme s'explique, du moins à l'origine en SMALLTALK, dans la mesure où les attributs ne sont accessibles que sur le receveur courant (`self`), ce qui rend la syntaxe de l'accès identique à celle de l'accès à une variable. La justification ne tient plus dès lors que les attributs peuvent être publics, comme en JAVA et C++. Dans tous les cas, c'est une confusion dangereuse d'assimiler un concept à sa syntaxe.

Pendant, dans le cas des *variables statiques*, le terme est parfaitement justifié puisqu'il n'y a là aucune trace de spécificité objet. Comme dans les autres langages de programmation, le terme désigne des variables dont l'allocation est unique et constante durant toute la durée de l'activation du programme, par opposition aux variables locales qui sont allouées dans la pile, de façon à la fois provisoire et multiple.

### Méthodes de classes, méthodes d'instances

De la même manière, il faut distinguer les méthodes suivant qu'elles concernent (s'appliquent à) les classes ou les instances.

L'exemple le plus significatif est celui de la création des instances : la création d'instance peut être implémentée par une méthode, `new` par exemple. Cette méthode ne pouvant évidemment pas s'appliquer à l'instance que l'on veut créer doit s'appliquer à la classe. C'est donc une méthode de la méta-classe, pour (dont le receveur est) la classe. Que fait cette méthode ? Elle alloue de la mémoire conformément aux besoins de la classe, puis elle initialise cette zone mémoire, conformément aux spécifications de la classe.

Cette initialisation peut être implémentée, elle aussi, par une méthode, mais ce sera cette fois une méthode de la classe, pour (dont le receveur est) l'instance que l'on est en train de créer. En C++ ou JAVA, cette méthode d'initialisation s'appelle *constructeur*, ce qui semble un parfait contre-sens : elle ne construit pas, elle remplit.

**Méthodes statiques** Ce qui s'applique aux variables statiques de C++ ou JAVA s'applique aussi à leurs méthodes statiques, qui ne méritent en aucun cas le nom de méthodes.

### La compatibilité de la méta-classe et des super-classes.

L'existence de ces deux niveaux pose bien vite un problème intéressant : les méthodes des instances peuvent envoyer des messages à la classe, et inversement. Lorsque l'on définit une classe, il faut donc que la méta-classe et les super-classes soient *compatibles*.

Dans un langage typé (s'il existait un langage réflexif typé<sup>4</sup>), il faudrait effectivement que les instances de la classe et la classe des instances soient des attributs typés, ce qui poserait un problème intéressant.

<sup>2</sup> Pour le premier point, cela dépend des implémentations : la plupart n'autorisent pas la redéfinition de variables de classes.

<sup>3</sup> On objectera que ces données constituent en fait les *domaines* des attributs `jour-de-la-semaine` et `mois` : c'est un fait, mais le modèle objet simple présenté ici ne permet pas de traiter ces *domaines*.

<sup>4</sup> Le problème de la théorie des types dans un langage réflexif est un problème ouvert.

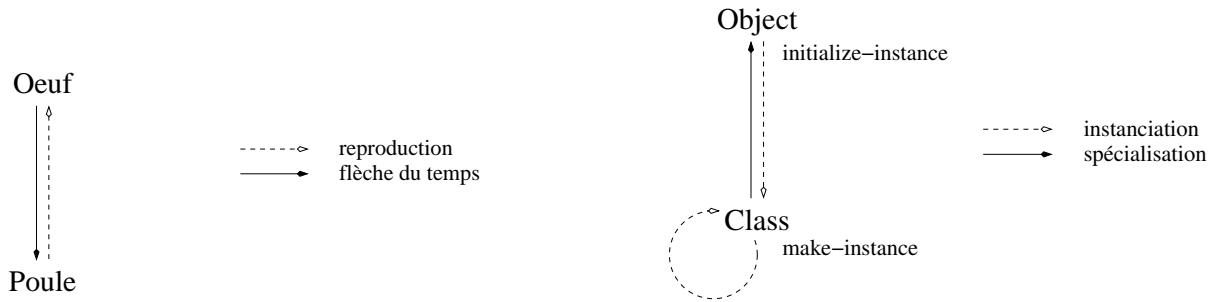


FIG. 5.1 – L’œuf et la poule et le modèle réflexif de OBJVLISP.

### 5.1.4 L’œuf et la poule : un problème d’amorçage

La réflexivité provoque un problème classique d’amorçage (en anglais *bootstrap*<sup>5</sup>), au moins dans les langages interprétés. Dans les langages compilés, le problème existe aussi mais il est moins crucial : il faut un compilateur pour compiler le compilateur. Mais on voit mieux comment faire, en procédant par petits pas<sup>6</sup>.

En revanche, dans les langages interprétés, le problème est aigu. En effet, la classe doit « exister » à l’exécution puisque c’est elle qui a en charge la création des instances. La classe préexiste donc à ses instances. Mais la classe est un objet, donc l’instance d’une classe.

L’œuf et la poule vous disais-je ! L’analogie peut être poussée plus loin (figure 5.1) :

- la relation de reproduction entre la poule et l’œuf est sémantiquement très proche de la relation d’instanciation qui lie la classe à l’instance ;
- l’œuf et la poule sont liés par une relation d’évolution temporelle qui lie deux entités du même niveau — la même, à des moments différents — de même que la relation de spécialisation lie deux classes.

Il n’y a pas de solution à l’amorçage, au sens où l’amorçage ne peut pas être décrit dans le langage objet considéré : il s’effectue au niveau de son implémentation.

**Amorçage et listes circulaires.** Le problème de l’amorçage a une certaine ressemblance avec la création de listes circulaires en LISP. Si l’on ne dispose que de l’interface fonctionnelle constituée par les fonctions `cons`, `car` et `cdr`, il est impossible de faire une liste circulaire : en effet, la cellule doit préexister à son `cdr` et une liste circulaire est une liste qui est son propre `cdr`, direct ou indirect.

Or, faire une liste circulaire est assez facile : il suffit de pouvoir faire une affectation dans le `cdr` d’une cellule, par exemple ainsi :

```
((lambda (x) (setf (cdr x) x)) (cons 1 ()))
```

Ce qui retournera une liste « infinie » de 1. Mais ce n’est plus « fonctionnel ».

## 5.2 La réflexivité dans les langages interprétés

### 5.2.1 Le modèle OBJVLISP

Le modèle OBJVLISP est le modèle objet réflexif le plus simple<sup>7</sup>. Il est caractérisé par 2 catégories d’objets — les objets et les classes — modélisées par 2 classes, `Object` et `Class`.

Bien entendu,

- les classes sont des objets, ce qui se traduit par le fait que `Class` est une sous-classe de `Object` ;
- `Object` et `Class` sont des classes, donc des instances de `Class` ;

<sup>5</sup> Le terme de *bootstrap* désigne classiquement en informatique le procédé par lequel un ordinateur s’initialise en lançant le programme qui va le lancer, c’est-à-dire le système d’exploitation, le programme qui gère les programmes. Le mot *bootstrap* désigne la boucle de chaussure et son utilisation en informatique « fait allusion au baron de Münchhausen qui se soulevait de terre en se tirant par ses sangles de bottes. [...] Cyrano faisait de même en jetant en l’air l’aimant qui l’attirait ensuite vers le ciel. » (citation de P. Cibois, citée par A. Le Diberder dans Le Monde Interatif du 28 mars 2001).

<sup>6</sup> Spinoza mentionnait déjà, au XVII<sup>e</sup> siècle, que c’était un faux problème :

Car, pour forger, il faut un marteau, et pour avoir un marteau, il faut le fabriquer. Ce pourquoi on a besoin d’un autre marteau et d’autres outils, et pour les posséder, il faut encore d’autres instruments, et ainsi infiniment.  
Spinoza, *De la réforme de l’entendement*, 1661.

<sup>7</sup> Soulignons qu’il s’agit d’un modèle et non d’un langage : des implémentations en LISP ont été réalisées, mais elles se sont contentées de valider le modèle.

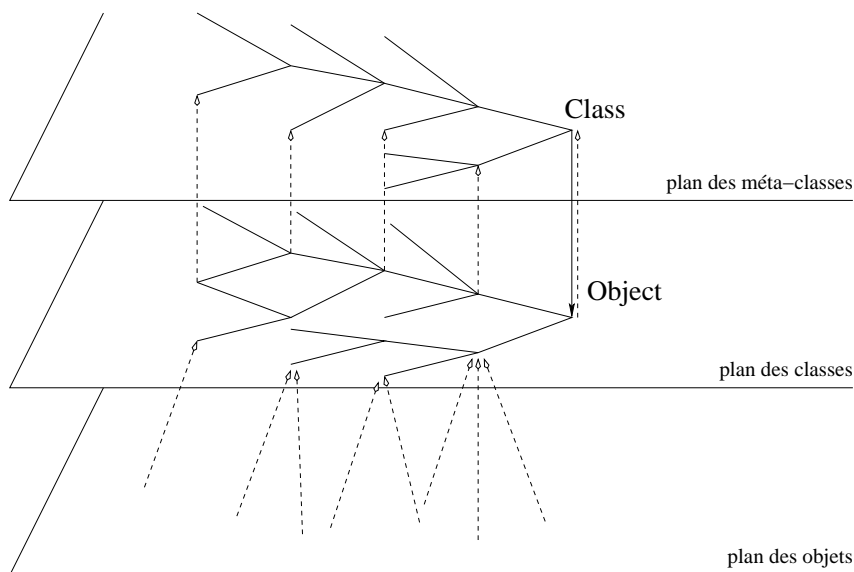


FIG. 5.2 – Extension du modèle réflexif de OBJVLISP

- toute classe est sous-classe de `Object` ;
- toute méta-classe est sous-classe de `Class`.

Le mécanisme d’instanciation est alors réalisé par une méthode `make-instance` définie dans `Class`, couplée à une méthode `initialize-instance` définie dans `Object` (les noms de méthodes sont empruntés à CLOS). La méthode `make-instance` fabrique un objet « vide » qui est initialisé par appel de la méthode `initialize-instance`. La correspondance avec les « constructeurs » (bien mal nommés) de C++ et JAVA est la suivante : la définition d’un constructeur équivaut à une définition de `initialize-instance` mais son appel correspond à un appel de `make-instance`.

L’ensemble est résumé dans la figure 5.1.

**L’amorçage en OBJVLISP.** Dans ce modèle très simple, on voit bien le problème d’amorçage, qui se situe à deux niveaux :

- la classe `Class` est instance d’elle-même : la circularité est manifeste ;
- la classe `Class` est sous-classe de la classe `Object` qui est elle-même instance de `Class` : la super-classe devant préexister à la sous-classe et la classe à ses instances, là encore, la circularité est claire.

De façon similaire à la construction d’une liste circulaire, bien que plus complexe, l’amorçage de OBJVLISP va consister à construire « à la main », c’est-à-dire au niveau du langage sous-jacent, LISP en l’occurrence, les structures de données qui implémentent les objets `Object` et `Class` en établissant les relations circulaires qui les lient, comme si ils avaient été construits par les procédés normaux d’instanciation.

**Extension du modèle.** Dans le modèle OBJVLISP, il est possible de définir de nouvelles méta-classes — il suffit de définir des sous-classes de `Class` — et il est possible d’associer à n’importe quelle classe n’importe quelle méta-classe lors de sa définition : la classe sera alors instance de la méta-classe considérée.

En première approximation, cela se ramène à définir des classes et des méta-classes dans deux plans parallèles, dont les seuls rapports sont constitués par les relations d’instanciation qui lient les premières aux secondes (figure 5.2). Les relations de spécialisation sont cantonnées à chacun des deux plans, à l’exception notable de la relation entre `Class` et `Object` : c’est là que réside la réflexivité.

## 5.2.2 Le modèle SMALLTALK

Le modèle réflexif de SMALLTALK est historiquement le premier modèle de méta-classes, mais il est à la fois plus complexe, moins élégant et moins flexible que celui d’OBJVLISP.

En effet, il n’y est pas possible de définir des méta-classes explicites que l’ont utiliserait pour telle ou telle classe. Chaque classe se voit associer, à la définition, une nouvelle méta-classe (le nom de la class suffixé par `class`), comme le montre la figure 5.2.2, chaque méta-classe ayant, pour super-classe, la méta-classe de la super-classe. C’est une façon simple, mais très rigide, d’assurer la compatibilité de la méta-classe et de la super-classe.

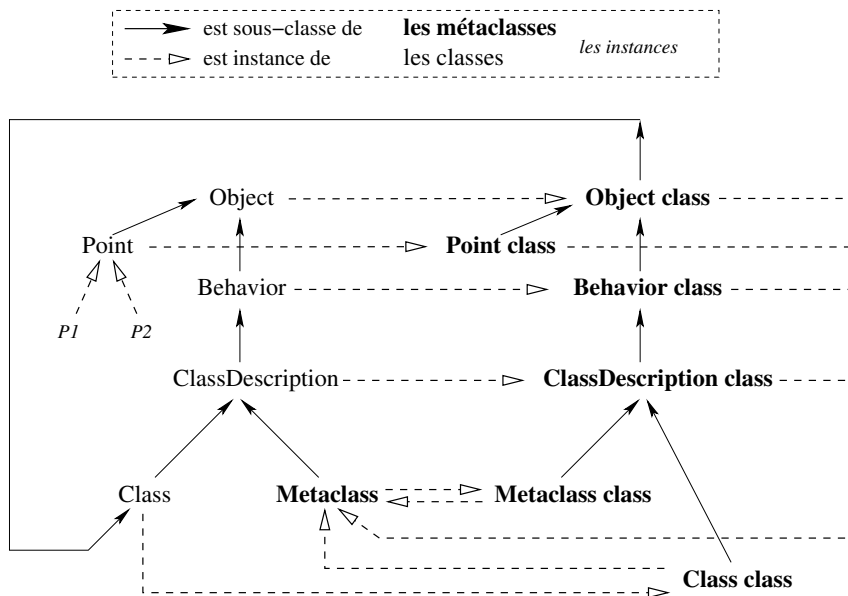


FIG. 5.3 – Le modèle réflexif de SMALLTALK (d'après G. Pavillet)

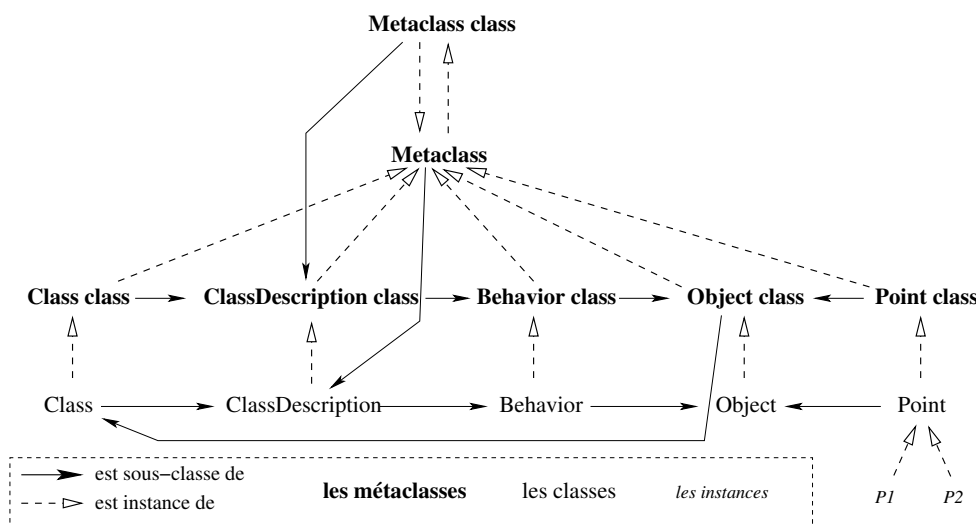


FIG. 5.4 – Vue par plan du modèle réflexif de SMALLTALK (d'après G. Pavillet)

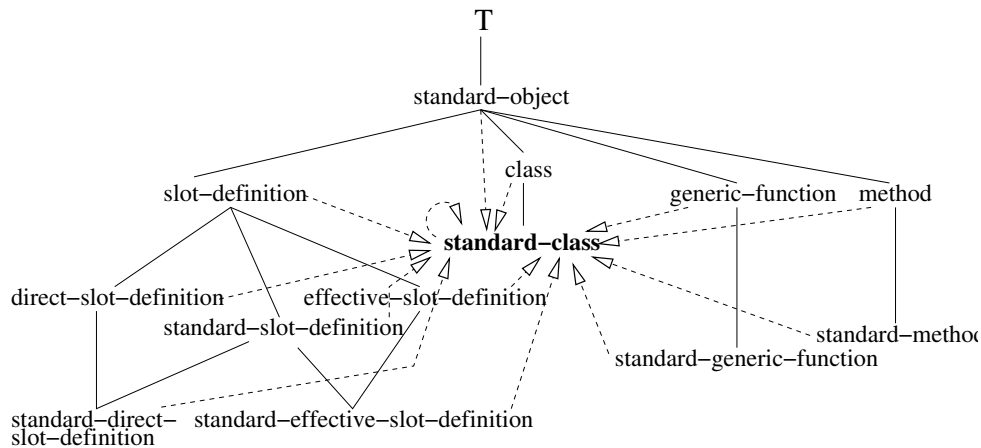


FIG. 5.5 – Le modèle réflexif de CLOS (d’après G. Pavillet)

Enfin, on remarque que la boucle de réflexivité ne se situe pas au niveau de la racine des classes, mais entre la classe des méta-classes et « sa méta-classe ».

Le modèle réflexif de SMALLTALK reproduit ainsi parfaitement le modèle des plans parallèles de la figure 5.2 si l’on met de côté la boucle de réflexivité : qui plus est, les deux plans sont parfaitement isomorphes (figure 5.2.2).

### 5.2.3 Le modèle CLOS

Le modèle CLOS peut être considéré comme une complexification du modèle OBJVLISP où les deux classes `standard-object` et `standard-class` prennent respectivement la place de `Object` et de `Class` (figure 5.5).

Une partie du modèle de CLOS, non représentée sur la figure, prend en charge les types de base, qui sont aussi des classes, et leurs valeurs, qui sont aussi des objets, ce qui explique les classes `T` et `class`.

L’originalité du modèle réflexif de CLOS réside dans le fait qu’il réifie aussi les attributs, sous le nom de *slot*, et tout ce qui est fonction ou méthode.

**Fonctions génériques et méthodes.** L’autre originalité, développée par ailleurs, est que la sélection de la méthode à appliquer lors de « l’envoi de message » se fait suivant le type de tous les arguments. Du coup, les méthodes ne sont plus définies dans la classe, comme en C++, JAVA ou EIFFEL, mais en-dehors. Cela entraîne, en particulier, la possibilité de rajouter de nouvelles méthodes à des classes préexistantes.

Enfin, on distingue, en CLOS, la *méthode*, définie par la macro `defmethod`, de la *fonction générique* qui regroupe toutes les méthodes de même nom.

**Compatibilité des méta-classes en CLOS.** La compatibilité de la méta-classe avec les super-classes est vérifiée en CLOS par la fonction générique `validate-superclass` qui prend 2 paramètres, la classe et une super-classe (elle est itérée sur chacune de ces dernières). La fonction `validate-superclass` exprime donc une compatibilité entre méta-classes : celle de la classe et celle de la super-classe.

## 5.3 La méta-programmation dans les langages compilés

Les exemples et modèles précédents de réflexivité concernaient tous des langages interprétés, où les classes existent lors de l’exécution du programme, ce qui en fait des objets de première classe. C’est le seul cas où le terme de réflexivité s’applique, lorsque le niveau méta est plongé dans le niveau de base et sert à l’implémenter.

Mais la méta-modélisation ou méta-programmation des classes peut aussi s’effectuer avant l’exécution, en intervenant dans l’une des étapes qui la précèdent : la compilation ou le chargement. Au total, les classes et les instances peuvent coexister, ou non, dans le même espace-temps (figure 5.6).

réflexivité	fonctionnalité	espace	temps	langages
exécution	make-instance	identique	identique	SMALLTALK, CLOS
chargement	load-class	identique	différent	JAVASSIST
compilation	compile-class	différent	différent	OPENC++, OPENJAVA

FIG. 5.6 – Les 3 moments de la méta-programmation : suivant le cas, l'espace-temps des classes et des instances est le même ou pas.

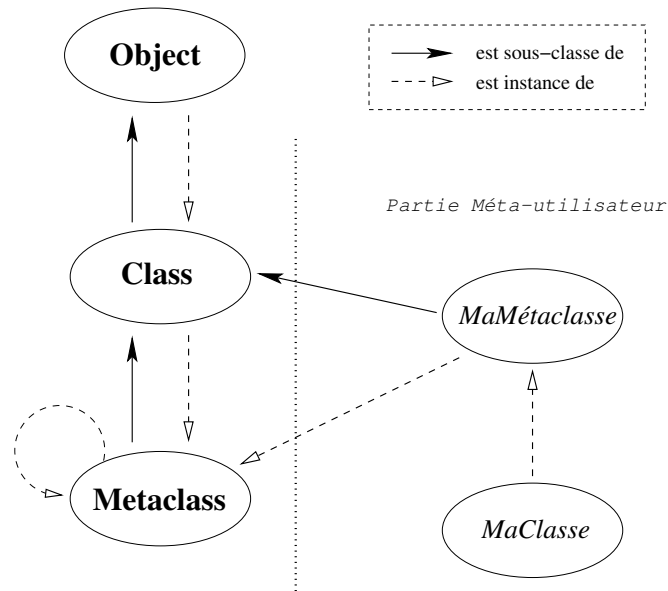


FIG. 5.7 – Le modèle réflexif de OPENC++ (d'après G. Pavillet)

### 5.3.1 L'introspection en JAVA

#### Le package reflect

Le langage JAVA repose sur une machine virtuelle (la JVM [MD97]) où les classes sont chargées de façon dynamique. Il a donc été naturel de faire exister les classes lors de l'exécution, au moins dans un sens introspectif. Le langage de base et le package `java.lang.reflect` contiennent tous les éléments pour accéder, en lecture seule, à partir d'un objet, à la structure de sa classes et à ses propriétés. Il est donc possible d'écrire des applications qui peuvent traiter des objets de type encore inconnu.

#### La méta-programmation au chargement : JAVASSIST

De la même manière, comme la première méta-fonctionnalité des classes est le chargement du bytecode qui leur est associé, il était naturel de chercher à intervenir sur cette étape. JAVASSIST permet de générer du code JVM ou JAVA lors du chargement d'une classe.

### 5.3.2 La méta-programmation à la compilation : OPENC++ et OPENJAVA

Depuis fort longtemps les langages de programmation sont munis de préprocesseurs (PL/1, C) ou de langages de macros (LISP) qui permettent de générer plus ou moins facilement du code.

La réflexivité dans les compilateurs revient à proposer un préprocesseur objet, basé à la fois sur l'arbre syntaxique du programme (comme les macros LISP et au contraire des préprocesseurs purement textuels comme celui de PL/1) et sur un méta-modèle plus ou moins développé. OPENC++ repose ainsi essentiellement sur l'arbre syntaxique et n'offre qu'un protocole de très bas niveau, alors que OPENJAVA, comme JAVASSIST, repose sur un méta-modèle un peu moins primitif qui permet d'élever un peu le niveau de la méta-programmation.

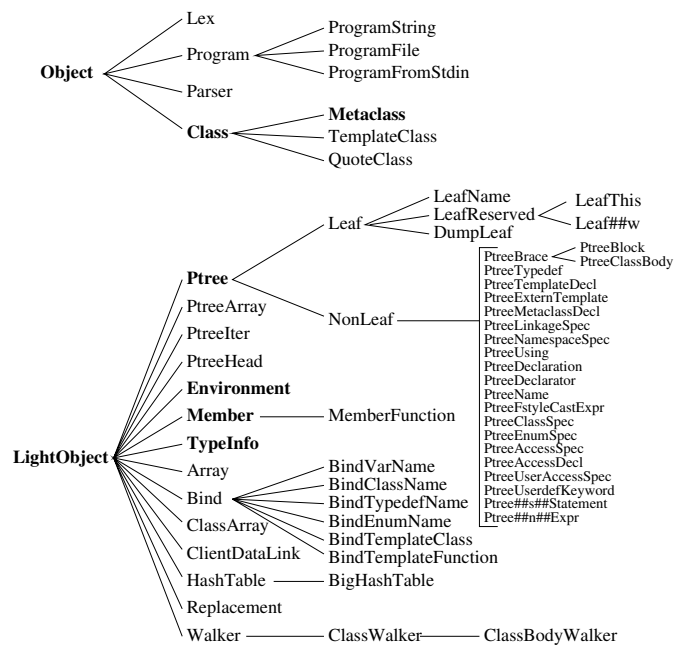


FIG. 5.8 – Les classes de OPENC++ (d’après G. Pavillet)

## Chapitre 6

# Méta-modèle pour les propriétés

Ce chapitre peut être d'un grand secours pour comprendre les chapitres précédents, en particulier tout ce qui concerne les « entités génériques » qui ont été à la base de notre raisonnement pour la redéfinition, la surcharge statique (section 3.8.4) et l'héritage de nom (section 4.2).

La méta-modélisation est une approche féconde pour expliquer, spécifier, voire implémenter les langages et systèmes à objets. Le noyau réflexif d'OBJVLISP présente ainsi une synthèse lumineuse du rôle des classes et des méta-classes dans le mécanisme d'instanciation : c'est sans doute le méta-modèle le plus petit qui apporte le plus d'enseignement. A l'opposé, le *meta-object protocol* (MOP) de CLOS [KdB91] et tous ceux qui s'en sont inspirés, modélisent la totalité des éléments nécessaires à la méta-programmation, au détriment d'une vision synthétique.

Au sens strict, une méta-modélisation consiste en la réification de certaines entités à la base d'un modèle. Or le modèle objet des langages à classes est constitué, dans toutes ses nombreuses variantes, de trois catégories d'entités : les objets eux-mêmes, les classes et les propriétés (méthodes ou attributs). Si OBJVLISP règle idéalement le cas des objets, des classes et de l'instanciation, aucun méta-modèle simple ne prend en compte de façon satisfaisante le cas des propriétés. A tel point que le vocabulaire même de la programmation et de la modélisation par objets est incapable d'appréhender correctement les méthodes dans leur dualité : la méthode, bien déterminée, telle que définie dans une classe ou celle qui est invoquée par un envoi de message de façon plus indéterminée. Seul CLOS utilise un vocabulaire et un méta-modèle qui permet de distinguer les deux, avec les notions de méthodes et de fonctions génériques [Ste90], mais l'approche fonctionnelle du langage et la sélection multiple en sont la cause et ce méta-modèle correct n'a pas été étendu aux attributs.

Dans ce chapitre, nous proposons donc un méta-modèle pour les propriétés, avec un objectif de simplicité, qui nous fera préférer l'esprit de la programmation par objets à la lettre des différents langages qui l'incarnent. A l'exemple de CLOS, ce méta-modèle distingue les *propriétés génériques* des *propriétés locales*, propres à une classe.

### 6.1 Le méta-modèle, version minimale

Le méta-modèle des propriétés doit prendre en compte deux éléments. D'une part, une classe *possède* des propriétés, méthodes ou attributs : le vocabulaire est beaucoup plus riche, car la classe peut les *introduire*, les *définir*, les *redéfinir*, ou les *hériter*, chacun de ces verbes ayant un sens bien précis. D'autre part, un envoi de message consiste à invoquer une méthode qui n'est déterminée qu'à l'exécution, par le type dynamique du receveur : c'est la liaison tardive. Ce qui vaut pour les méthodes s'applique aussi aux attributs dans la mesure où du code (assertions en EIFFEL, réflexes dans les *frames*) peuvent leur être attaché, ou que leur type ou leur protection (EIFFEL), voire la totalité de leur description (CLOS), peuvent être redéfinis.

#### 6.1.1 Les éléments du méta-modèle

La prise en compte de cette dualité des propriétés provoquée par la liaison tardive nécessite manifestement de distinguer deux catégories d'entités. En effet, un méta-modèle doit permettre de remplacer statiquement, dans toute expression d'un programme, chaque identificateur par l'entité uniquement déterminée que cet identificateur désigne. Les *propriétés génériques* constituent ce qui est invariant par spécialisation et redéfinition, ainsi que la cible uniquement déterminée de l'envoi de message. Les *propriétés locales* sont les propriétés telles que définies dans une classe.

Une propriété générique est constituée de propriétés locales, chacune de ces dernière appartenant à une unique propriété générique (figure 6.1). Une propriété générique a un nom, qui est aussi celui de ses propriétés locales.

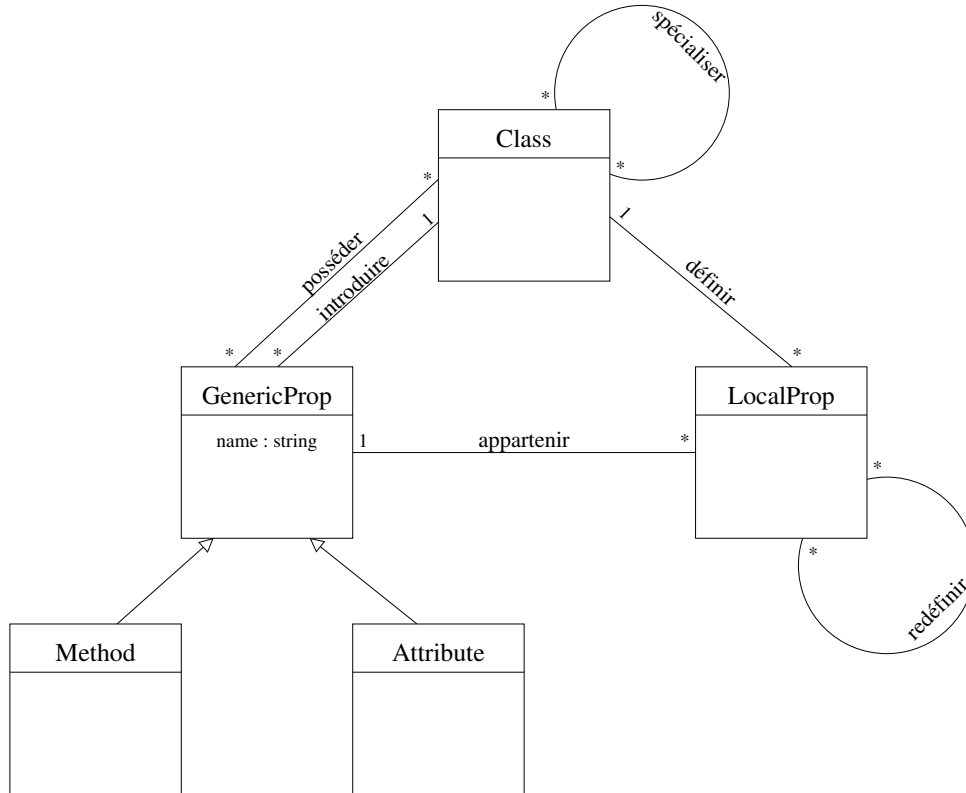


FIG. 6.1 – Méta-modèle des propriétés

Une classe *possède* des propriétés génériques, chacune d’elles étant *introduite* par une unique classe. Une propriété locale est *définie* par une unique classe. Enfin, la *spécialisation* est une relation entre classes alors que la *redéfinition* est une relation entre propriétés locales d’une même propriété générique. Les mécanismes `super` de SMALLTALK ou JAVA et `precursor` de EIFFEL correspondent très exactement à la relation de redéfinition.

Pour finir, on peut constater que seules les propriétés génériques ont besoin d’être spécialisées en méthodes et attributs, les propriétés locales « héritant » cette caractéristique de leur propriété générique, dont c’est un invariant. EIFFEL autorise bien la redéfinition d’une méthode sans paramètres en attribut. La prise en compte la plus simple consiste sans doute à avoir 2 propriétés génériques distinctes, la méthode étant redéfinie comme un accesseur de l’attribut dans la classe qui introduit ce dernier. Il est ainsi possible de conserver la seule spécialisation des propriétés génériques en méthode ou attribut. De façon similaire, on pourra modéliser explicitement les accesseurs d’un attribut par une catégorie spécialisée de propriétés génériques. Bien entendu, le modèle peut être compliqué en spécialisant aussi les propriétés locales, mais ce n’est pas une nécessité.

### 6.1.2 Contraintes

Plusieurs contraintes sont nécessaires pour assurer la correction du méta-modèle. Dans la suite,  $c$ ,  $pg$  et  $pl$  désignent respectivement des classes, propriétés génériques et propriétés locales,  $Def$ ,  $Pos$  et  $Intro$  les associations *définir*, *posséder* et *introduire*,  $Gen(pl)$  désigne la propriété générique d’une propriété locale,  $Class(pl)$  (resp.  $Class(pg)$ ) la classe de définition (resp. introduction) d’une propriété locale (resp. générique) et  $\prec$  est la fermeture transitive de la relation de spécialisation.

$$pl \in Def(c) \Rightarrow Gen(pl) \in Pos(c) \quad \text{triangle} \quad (6.1)$$

$$Class(pl_1) = Class(pl_2) \ \& \ Gen(pl_1) = Gen(pl_2) \Rightarrow pl_1 = pl_2 \quad \text{unicité} \quad (6.2)$$

$$pg \in Pos(c_1), c_2 \prec c_1 \Rightarrow pg \in Pos(c_2) \quad \text{héritage} \quad (6.3)$$

$$Class(pl) \preceq Class(Gen(pl)) \quad \text{intro} \quad (6.4)$$

La contrainte entre redéfinition et spécialisation exprime qu’une redéfinition implique une spécialisation :

$$pl_1 \text{ Redef } pl_2 \Rightarrow Gen(pl_1) = Gen(pl_2) \ \& \ Class(pl_1) \prec Class(pl_2) \quad (6.5)$$

La réciproque n'est vraie qu'en l'absence de redéfinitions intermédiaires : contrairement à la spécialisation ( $\prec$ ), la redéfinition n'est pas transitive.

### 6.1.3 Extraction du méta-modèle

La définition d'une classe va conduire à mettre en œuvre les opérations suivantes pour en extraire les instances du méta-modèle :

1. extraction récursive du méta-modèle des super-classes directes ;
2. héritage de toutes les propriétés génériques des super-classes directes (avec traitement des conflits de nom, s'il y a lieu, voir section 6.3) ;
3. pour chaque propriété définie ou redéfinie dans la classe,
  - a) recherche de la propriété générique correspondante, si elle existe, ou création d'une nouvelle propriété générique sinon ;
  - b) création de la propriété locale et rattachement à la propriété générique.

Dans chaque expression d'envoi de message, de la forme  $x.f \circ \circ(\text{arg})$ , le nom du message ( $f \circ \circ$ ) permet, dans le contexte de l'expression (par exemple, le type statique de  $x$ ) de déterminer uniquement la propriété générique correspondant. L'envoi de message lui-même consistera alors à rechercher, dans cette propriété générique, la propriété locale « la plus spécifique », c'est-à-dire non redéfinie, pour la classe de la valeur de  $x$ .

## 6.2 Influence du typage

Le méta-modèle doit être légèrement amendé pour tenir compte du typage et des spécificités de certains langages.

### 6.2.1 Typage statique

En typage statique et en première approximation, une propriété générique est uniquement déterminée par son nom et sa classe d'introduction. Lors de la définition d'une propriété locale dont le nom n'appartient à aucune propriété générique héritée, une nouvelle propriété générique est créée.

Cette détermination unique par le nom et la classe d'introduction implique que, pour une classe donnée, un nom de propriété soit non ambigu. Deux exceptions se présentent cependant : la surcharge statique et l'héritage multiple (section suivante).

#### Surcharge statique

La surcharge statique pratiquée par C++ et JAVA oblige à rajouter au nom les types des paramètres, qui sont invariants par redéfinition dans le cas de ces langages. Il y a surcharge statique, dans une classe, lorsque cette classe possède deux propriétés génériques de même nom : le couple nom et signature apporte alors une désignation non ambiguë de la propriété générique. En revanche, un envoi de message peut être ambigu si le type statique des arguments est conforme aux signatures incomparables de plus d'une propriété générique.

Cependant, la modélisation de la surcharge comme une association entre entités du méta-modèle nécessite de compliquer ce dernier. En effet, pour que la modélisation soit correcte, la surcharge doit concerner des propriétés génériques (et non des propriétés locales) mais, pour qu'elle soit précise, il faudrait que ce soit localisé dans une classe : il est inutile d'avoir une relation de surcharge entre des méthodes de même nom qui n'appartiennent pas à des classes communes. On peut arriver au résultat recherché, avec une assez grande redondance néanmoins, en rajoutant une catégorie d'entités, les *propriétés surchargées*, intermédiaires entre les propriétés génériques et locales : ces propriétés surchargées sont rattachées à une classe et regroupent l'ensemble des propriétés locales de la propriété générique qui sont définies dans la classe et ses sous-classes (figure 6.2). La contrainte de la surcharge est alors :

$$ps_2 \text{ Overload } ps_1 \Leftrightarrow \begin{cases} \text{Class}(ps_1) = \text{Class}(ps_2) \\ \text{Name}(ps_1) = \text{Name}(ps_2) \\ \text{Sign}(ps_1) \neq \text{Sign}(ps_2) \end{cases} \quad \text{surcharge} \quad (6.6)$$

Si la modélisation de la surcharge n'est pas de première nécessité — dans sa spécification la plus acceptable, celle de JAVA, un simple renommage de propriété permet de la faire disparaître sans changer le comportement des programmes —, la modélisation du typage statique peut nécessiter une modélisation analogue, par une explicitation de la restriction d'une propriété générique à une classe et à ses sous-classes. On en verra deux autres exemples par la suite.

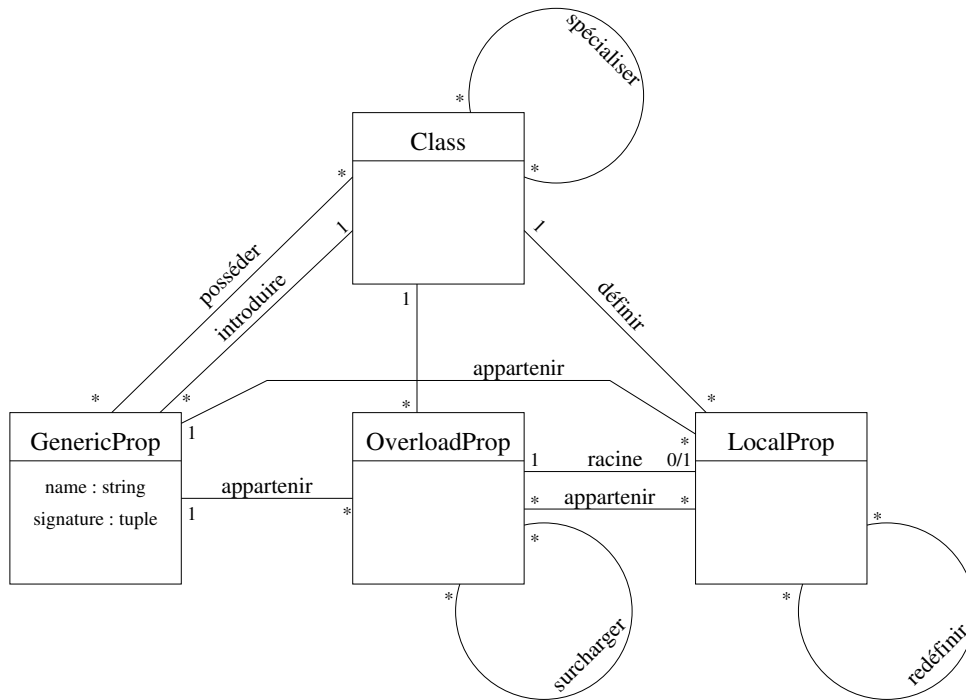


FIG. 6.2 – Méta-modèle de la surcharge statique

### Masquage et visibilité

Il faudrait compléter le modèle pour tenir compte des mécanismes de protection (`export` en EIFFEL, `private`, `protected` et `public` en C++ et JAVA). Sans aller jusque là, il faut d'abord tenir compte des règles de *masquage*, qui sont à la base de l'héritage de valeur. Par défaut, le masquage s'identifie à la redéfinition. C'est le cas dans tous les langages sans surcharge statique, ainsi qu'en JAVA. Ce n'est pas le cas en C++, où le masquage porte sur les noms. En C++, la correspondance entre propriétés surchargées et propriétés locales est biunivoque et seules les propriétés surchargées du type statique du receveur sont *visibles*<sup>1</sup>. On peut néanmoins unifier le comportement de JAVA et de C++ en ce qui concerne la visibilité (seules les propriétés surchargées du type statique du receveur sont *visibles*) en différenciant la formation des propriétés surchargées (correspondance biunivoque avec l'association définir en C++, avec l'association posséder en JAVA) :

$$\forall c, \forall pg \in Pos(c), \exists ps : Class(ps) = c \ \& \ Gen(ps) = pg \quad \text{JAVA} \quad (6.7)$$

$$\forall c, \forall pl \in Def(c), \exists ps : Class(ps) = c \ \& \ Rac(pl) = ps \quad \text{C++} \quad (6.8)$$

Le masquage des noms de C++ se traduit alors par le fait que, lorsqu'aucune propriété d'un certain nom n'est définie dans une classe, toutes les propriétés surchargées de ce nom sont héritées de ses super-classes directes. Dans chaque classe, à un nom de propriété correspond un ensemble de propriétés surchargées, qui s'héritent et se masquent.

### 6.2.2 Typage dynamique

Le typage dynamique conduit à un phénomène que l'on a appelé *surcharge d'introduction*, lorsque deux propriétés génériques de même nom sont introduites dans deux classes incomparables. En l'absence d'annotations de type, il n'y a pas moyen de les distinguer car le même envoi de message  $x.foo(arg)$  peut s'adresser alternativement à des receveurs de l'une ou l'autre classe. Il s'ensuit qu'une propriété générique peut être introduite par plus d'une classe, pourvu que ces classes soient incomparables. C'est le cas de SMALLTALK, dont le méta-modèle ne comporte pas de propriétés génériques, qui n'apparaissent que comme « sélecteurs de méthodes ».

<sup>1</sup> On voit bien ici à quel point les notions de visibilité et de protection sont distinctes, malgré leur confusion fréquente, en particulier en UML.

## 6.3 Héritage multiple

Le méta-modèle que nous proposons explicite simplement les différences entre les héritages (et conflits) de noms et de valeurs distingués dans pour prendre en compte les problèmes posés par l'héritage multiple. L'*héritage de valeur* est un héritage de propriétés locales, pour une même propriété générique : l'ensemble de conflit est le sous-ensemble de propriétés locales de la propriété générique, héritées par la classe considérée et les plus spécifiques, c'est-à-dire qu'aucune n'est redéfinie par une autre. L'*héritage de nom* est un héritage de propriétés génériques. Une classe hérite de toutes les propriétés génériques de ses super-classes : il y a conflit si 2 propriétés génériques héritées ont le même nom (et signature en cas de surcharge statique) et non pas si la même propriété générique est héritée « deux fois ». L'existence du conflit de nom dépend donc directement de la définition des propriétés génériques. En typage statique, elles sont identifiées par leur nom et la classe d'introduction : cela conduit à la bonne définition du conflit de nom. En typage dynamique, comme il n'est pas possible de distinguer des propriétés génériques de même nom, il n'y a plus de conflit de nom (ou il est résolu systématiquement par l'identité), ce qui pose un problème sémantique certain.

### 6.3.1 Cas particulier de C++

C++ dispose du mot-clé `virtual`<sup>2</sup> pour qualifier les super-classes en cas d'héritage multiple. L'usage de ce mot-clé est nécessaire pour que le comportement de l'héritage multiple soit sémantiquement correct, mais son absence réduit le surcoût entraîné par l'implémentation standard de l'héritage multiple. Le méta-modèle proposé ici ne s'applique que lorsque l'héritage est « virtuel » : dans le cas contraire, la présence d'héritage répété conduit à la duplication de propriétés génériques, un peu comme avec le renommage d'EIFFEL.

### 6.3.2 Cas particulier d'EIFFEL

EIFFEL présente plusieurs particularités qui peuvent nécessiter une adaptation du méta-modèle. Le traitement de l'héritage de valeur ne considère pas le masquage en cas d'héritage multiple, ce qui induit des conflits de valeurs inutiles : tout se passe comme si les propriétés locales héritées étaient effectivement recopiées dans les sous-classes. L'effet sur le méta-modèle est assez faible.

La prise en compte du renommage (clause `rename`) est plus compliquée. Pour un renommage simple en cas de conflit de nom, lorsque 2 propriétés génériques de même nom sont héritées, il suffit que l'association posséder soit implémentée dans les classes par un dictionnaire nom-propriété générique, au lieu d'une simple liste de propriétés génériques. Et l'on complète les propriétés génériques par un attribut listant les éventuels alias. Une solution alternative consiste en une modélisation analogue à celle de la surcharge statique. Mais le renommage peut aussi être utilisé, de façon contestable, en l'absence de conflit de nom, par exemple, mais pas forcément, en cas de conflit de valeurs. Dans ce cas, le renommage peut conduire à dupliquer la propriété générique, en un héritage répété assez similaire à celui de C++ en cas d'héritage multiple non virtuel. Lorsque la même propriété générique est héritée sous deux noms différents, il faut alors en créer une nouvelle. Il est enfin possible en EIFFEL d'associer la même définition à deux *features* différents : c'est d'ailleurs aussi une conséquence du renommage avec duplication. On peut traduire cela, dans le méta-modèle, soit en associant la même propriété locale à deux propriétés génériques distinctes (ce qui nécessite une adaptation du méta-modèle), soit en distinguant la propriété locale et le code, les deux propriétés locales pointant alors sur le même code.

---

<sup>2</sup> A propos de mot-clé `virtual`, il va de soi, mais cela va mieux en le disant, que seules les méthodes « virtuelles » relèvent du méta-modèle dont il est question ici.

# Chapitre 7

## Implémentation

Si l'implémentation ne peut pas et ne doit pas servir à spécifier un langage, elle peut néanmoins aider à le comprendre.

### 7.1 En héritage et sous-typage simples

En héritage et sous-typage simples, un objet est implémenté comme une table de ses attributs, auxquels s'ajoute un pointeur sur une table de méthodes commune à toutes les instances d'une même classe (figure 7.1). La première table contient, pour chaque objet, la valeur de ses attributs. La seconde table contient les adresses des méthodes. L'implémentation des objets et de l'envoi de message se caractérise par les invariants suivants.

**Invariant 7.1** *La valeur d'une référence — paramètre, variable, attribut ou valeur de retour d'un appel fonctionnel — sur un objet est invariante relativement à son type statique.*

**Invariant 7.2** *Chaque attribut ou méthode — c'est-à-dire chaque propriété générique — a un indice non ambigu et invariant par héritage, donc indépendant du type statique du receveur.*

Le sous-typage simple se caractérise ainsi par une invariance absolue vis-à-vis des types statiques, qui n'ont aucun rôle à l'exécution. L'implémentation se conforme donc à la sémantique fondamentale de l'approche objet, pour laquelle le type dynamique exprime l'essence de l'objet, le type statique étant purement contingent. On notera par la suite  $\tau_s$  et  $\tau_d$  les types statique et dynamique d'une entité, avec  $\tau_d \leq \tau_s$  : le type statique est celui qui annote explicitement une entité d'un programme<sup>1</sup>, alors que le type dynamique est la classe dont l'instanciation a créé l'objet qui value l'entité considérée.

L'envoi de message se compile alors par une séquence de trois instructions :

```
load [object + #tableOffset], table
load [table + #selectorOffset], method
call method
```

et l'accès à un attribut est immédiat :

```
load [object + #attributeOffset], attribute
```

Le calcul des tables de méthodes (pour la classe) et d'attributs (pour les instances) est le suivant : pour chaque classe dont la super-classe a déjà été compilée, on numérote les attributs (resp. méthodes) introduits dans la classe, en partant de l'indice maximum des attributs (resp. méthodes) de la super-classe. L'absence d'héritage multiple et de surcharge d'introduction garantit la correction du résultat (c'est-à-dire qu'il n'y a jamais aucun problème avec les indices des méthodes héritées et qu'il n'est jamais nécessaire de vérifier que l'indice que l'on veut affecter n'est pas déjà occupé).

C'est l'implémentation de base de la plupart des langages à objets, aussi bien celle de C++, tant qu'on reste en héritage simple et « non virtuel » (sans usage du mot-clé `virtual` pour l'héritage), que celle de JAVA en ne se servant pas des interfaces. La figure 7.1 résume cette implémentation.

<sup>1</sup> Ou qui peut s'en déduire, assez directement, par exemple, le type de retour d'une méthode, en cas de redéfinition covariante.

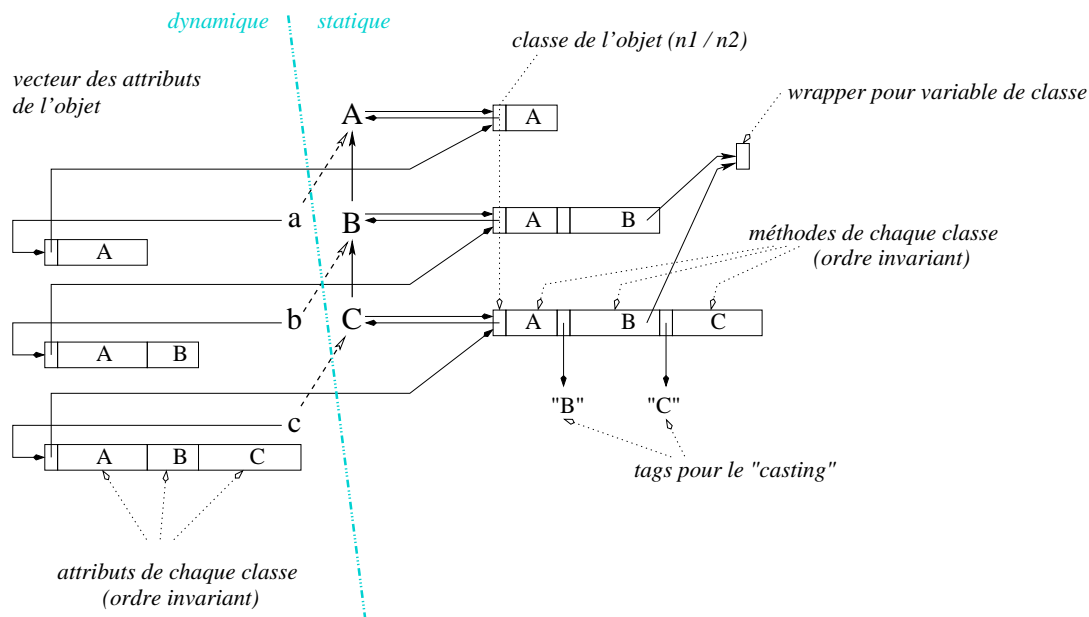


FIG. 7.1 – Structure des objets et des tables de méthodes en héritage simple : 3 classes *A*, *B* et *C* et leurs instances respectives, *a*, *b* et *c*.

## 7.2 En héritage multiple

### 7.2.1 Le problème de l'héritage multiple

En héritage multiple, le problème se complique considérablement comme le montre [ES90, chapitre 10] dans le cas de C++. La complication est d'autant plus importante que C++ offre des traits de langage — en l'occurrence le double rôle du mot-clé `virtual` — qui ne relèvent pas d'un bon usage de l'approche objet. Nous nous placerons, dans cette section, dans un cadre plus orthodoxe qui reviendrait, en C++, à utiliser systématiquement `virtual :`

- toutes les fonctions sont *virtuelles*, au sens où elles sont toutes sélectionnées par liaison tardive,
- tous les héritages sont *virtuels*, au sens où chaque super-classe n'est utilisée qu'une seule fois.

Ces précautions liminaires sont inutiles pour un langage objet normalement constitué comme Eiffel [Mey92, Mey97].

Le problème causé par l'héritage multiple s'énonce simplement : l'indice d'une méthode ou d'un attribut ne peut plus être invariant par héritage (invariant 7.2), en tout cas tant que ces indices sont calculés de façon séparée, en cherchant à les minimiser. La raison en est la suivante : étant donné deux classes incomparables *B* et *C*, qui occupent les mêmes indices, il est toujours possible d'en définir une sous-classe commune *D*. On se trouve donc en situation de conflit puisque deux attributs ou deux méthodes sont en compétition pour le même indice.

Ce premier constat a une conséquence décisive : si l'on veut que l'envoi de message s'effectue par une indication dans une table, un pointeur sur un objet n'est plus invariant suivant son type statique (invariant 7.1).

### 7.2.2 Principe d'implémentation

On est donc conduit à relâcher l'invariance de l'indice des attributs et méthodes comme suit :

**Invariant 7.3** *Chaque attribut a un indice non ambigu et invariant dans le contexte du type statique qui l'introduit, donc indépendamment du type dynamique.*

**Invariant 7.4** *Chaque méthode a un indice non ambigu et invariant dans le contexte d'un type statique qui connaît la méthode, donc indépendamment du type dynamique.*

Mais ces indices ne sont plus invariants par héritage, c'est-à-dire entre deux types statiques liés par une relation de spécialisation, pas plus que l'objet lui-même : la valeur de `self` et de tout pointeur sur un objet dépend de son type statique. Tout se passe comme si les tables des attributs (l'objet) et des méthodes étaient constituées de sous-tables (ou sous-objets), une par super-classe. Et l'invariant principal est alors le suivant (figure 7.2) :

**Invariant 7.5** *Toute entité de type statique *T* est liée au sous-objet correspondant à *T*, muni de sa propre table de méthodes.*

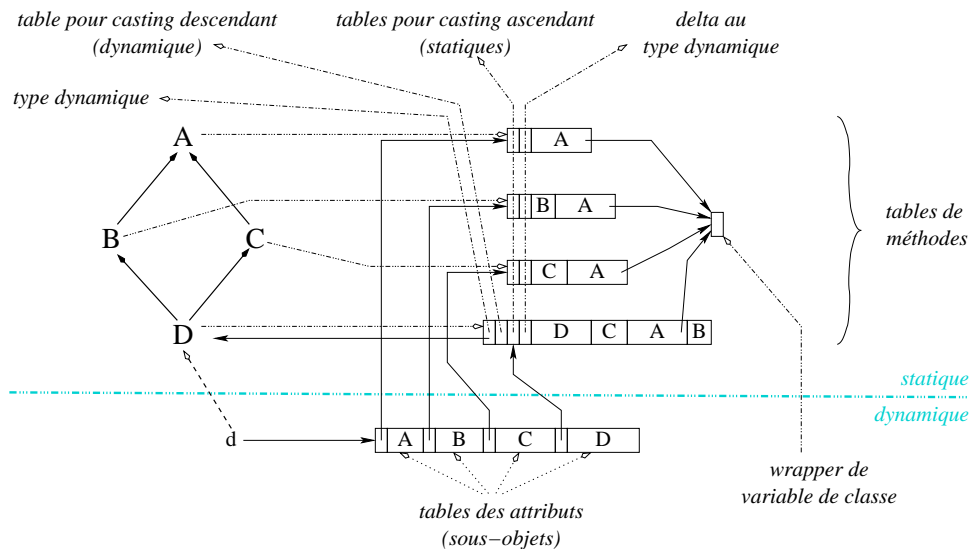


FIG. 7.2 – Tables des attributs et des méthodes en héritage multiple : contrairement à la figure 7.1, une seule instance est décrite.

Cet invariant est trivialement vérifié par l’implémentation de l’héritage simple (invariant 7.1). En héritage multiple, il est nécessaire de rajouter une propriété de non trivialité, qui est la source du surcoût de cette implémentation :

**Invariant 7.6** *Deux sous-objets de types statiques différents sont distincts.*

L’unique exception a lieu lorsqu’une classe  $F$  spécialise une classe  $E$ , en héritage simple, sans rajouter de nouveaux attributs : le sous-objet de type  $F$  peut disparaître dans le sous-objet de type  $E$  muni de la table de méthodes de  $F$ .

Contrairement au sous-typage simple, l’implémentation de l’héritage multiple se caractérise donc par une dépendance absolue vis-à-vis des types statiques, à tel point qu’il n’est jamais évident que le comportement des programmes respecte bien la sémantique invariante de rigueur. Pour la théorie des types, un type n’a qu’un but dans la vie : se faire éliminer (*erasure*) par le compilateur, qui prouve qu’il peut le faire sans risque d’erreur de type à l’exécution. Avec cette implémentation, c’est manifestement raté.

Chaque sous-objet ne contient que les attributs *introduits* par son type statique. Chaque table de méthodes d’un type statique contient toutes les méthodes connues par ce type, mais avec des valeurs (adresses) correspondant aux méthodes effectivement héritées par le type dynamique. Ainsi, deux instances directes de classes différentes ne partagent pas les tables de méthodes de leurs super-classes communes : ces tables ont la même structure, mais pas le même contenu (figure 7.3).

Pour un type statique donné, l’ordre de ces méthodes est *a priori* quelconque, mais il est raisonnable de les regrouper par classe (ce qui est fait dans la figure) et d’assurer une certaine invariance par spécialisation, lorsque c’est possible (cas d’héritage simple). Mais cette organisation n’a aucun effet sur l’efficacité de l’implémentation.

L’invariant 7.5 impose de recalculer la valeur de `self` à chaque envoi de message : lorsque le receveur est une entité de type statique  $\tau_s$ , et que la méthode sélectionnée a été définie dans la classe  $v$ , il faut savoir de combien il faut incrémenter ou décrémenter `self` pour obtenir, à partir du sous-objet de type  $\tau_s$  un sous-objet de type  $v$ , ce que l’on notera  $\Delta_{\tau_s, v}^2$ . La table des méthodes est donc double : elle contient, pour chaque méthode, l’adresse de la méthode ainsi que la valeur de cet incrément.

Au total, l’envoi de message se compile donc par une séquence de cinq instructions<sup>3</sup> :

```
load [object + #tableOffset], table
load [table + #deltaOffset], delta
load [table + #selectorOffset], method
add object, delta, object
call method
```

<sup>2</sup> Précisons que la notation  $\Delta_{t, u}$ , ainsi que toutes les notations  $\Delta$  qui vont suivre, sous-entend un type dynamique  $\tau_d$  donné, dont l’explicitation alourdirait trop la notation.

<sup>3</sup> En italiques, les instructions supplémentaires par rapport à l’héritage simple.

type statique → ↓ dynamique	A	B	C	D													
A	<table border="1"><tr><td></td><td>A</td></tr></table>		A	—	—	—											
	A																
B	<table border="1"><tr><td></td><td>A</td></tr></table>		A	<table border="1"><tr><td></td><td>A</td><td>B</td></tr></table>		A	B	—	—								
	A																
	A	B															
C	<table border="1"><tr><td></td><td>A</td></tr></table>		A	—	<table border="1"><tr><td></td><td>A</td><td>C</td></tr></table>		A	C	—								
	A																
	A	C															
D	<table border="1"><tr><td></td><td>A</td></tr></table>		A	<table border="1"><tr><td></td><td>A</td><td>B</td></tr></table>		A	B	<table border="1"><tr><td></td><td>A</td><td>C</td></tr></table>		A	C	<table border="1"><tr><td></td><td>A</td><td>B</td><td>C</td><td>D</td></tr></table>		A	B	C	D
	A																
	A	B															
	A	C															
	A	B	C	D													

FIG. 7.3 – Tables des méthodes pour l'exemple de la figure 7.2, suivant les types statiques et dynamiques. Pour un même type statique (verticalement), les tables sont isomorphes mais diffèrent par leurs contenus (adresses des méthodes) ; en revanche, pour un même type dynamique (horizontalement), les morceaux isomorphes contiennent les mêmes adresses mais pas les mêmes décalages.

Une technique alternative consiste à définir une petite fonction intermédiaire qui fait le décalage (technique dénommée *thunks* provenant d'ALGOL, mais pas utilisable sur tous les processeurs, d'après [ES90]). En cas d'implémentation par *thunk*, la séquence d'instructions est la même qu'en héritage simple, mais elle provoque un branchement au code suivant :

```
add object, #delta, object
jump #method
```

On économise ainsi une indirection dans une table, au prix d'un saut constant<sup>4</sup>.

### 7.2.3 L'héritage multiple non virtuel de C++

L'implémentation de C++ telle qu'elle est décrite dans [ES90, Lip96] diffère de celle que nous avons présentée ici sur deux points :

- elle est plus complexe à exposer et à implémenter car elle traite aussi bien l'héritage « virtuel » que « non virtuel », suivant l'usage qui est fait du mot-clé `virtual` ;
- son surcoût est moindre dès qu'il n'est pas fait usage de ce mot-clé `virtual` ;
- le mélange d'héritage virtuel et non virtuel, ajouté aux protections d'héritage, donne une complexité telle à l'héritage qu'il est fortement déconseillé d'en exploiter toute la combinatoire.

Le principe d'implémentation de l'héritage non virtuel est le suivant<sup>5</sup> :

- une classe sans super-classe s'implémente exactement comme en héritage simple ;
- en cas d'héritage, qu'il soit simple ou multiple, l'implémentation des instances de la sous-classe est obtenue par concaténation des implémentations des instances des super-classes directes<sup>6</sup>, prolongée par les attributs introduits par la sous-classe : la nouvelle classe a autant de tables de méthodes que ses super-classes directes réunies, les méthodes qu'elle introduit prolongeant la table de la première de ses super-classes.

**Invariant 7.7** *Le sous-objet associé à  $\tau_d$  est concaténé au sous-objet de l'une de ses super-classes directes non virtuelles et leurs tables de méthodes sont communes.*

En cas d'héritage simple, l'implémentation est donc exactement celle du sous-typage simple, modulo la présence des décalages, bien qu'ils soient tous nuls : la technique des *thunks* semble donc plus adaptée, puisqu'elle fait alors disparaître, en pratique, les décalages. En effet, alors qu'en héritage multiple standard, l'unique *thunk* de décalage zéro correspond au cas où la méthode est définie dans  $\tau_s$ , dans l'héritage multiple non virtuel, le cas est beaucoup plus fréquent et c'est toujours vrai pour les cas d'héritage simple.

En cas d'héritage multiple effectif, la compilation des envois de messages ou des accès aux attributs est exactement la même qu'en héritage multiple : la seule différence est que le recours effectif aux décalages est moins fréquent pour les attributs.

### Héritage répété

L'absence du mot-clé `virtual` a un effet très positif sur le coût, mais très négatif sur la sémantique de l'héritage multiple et sur la réutilisabilité. Dans l'exemple de la figure 7.2, lorsque l'héritage de la classe *A* par *B*

<sup>4</sup> Saut qui peut être parfaitement anticipé par le processeur, sauf que la séquence précédent le saut est ici un peu courte.

<sup>5</sup> L'auteur s'excuse de décrire une fonctionnalité par son implémentation, mais il s'agit ici d'un cas typique où l'organe a créé la fonction.

<sup>6</sup> Dans le cas d'un mélange d'héritage virtuel et non virtuel, les super-classes indirectes virtuelles sont exclues de cette concaténation et rajoutées à la fin.

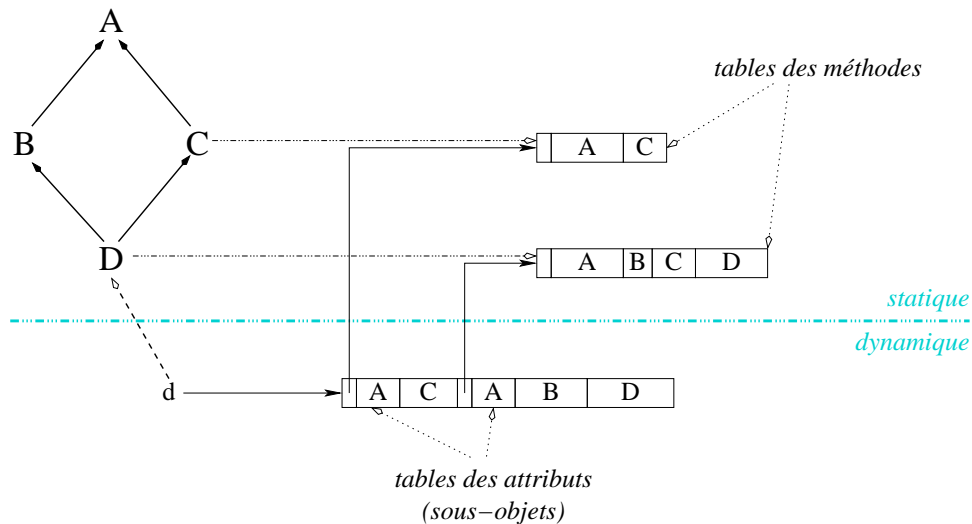


FIG. 7.4 – Tables des attributs et des méthodes en héritage multiple « non virtuel »

type statique → ↓ dynamique	A	B	C	D						
A	<table border="1"><tr><td>A</td></tr></table>	A	—	—	—					
A										
B	B	<table border="1"><tr><td>A</td><td>B</td></tr></table>	A	B	—	—				
A	B									
C	C	—	<table border="1"><tr><td>A</td><td>C</td></tr></table>	A	C	—				
A	C									
D	C/D	D	<table border="1"><tr><td>A</td><td>C</td></tr></table>	A	C	<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>D</td></tr></table>	A	B	C	D
A	C									
A	B	C	D							

FIG. 7.5 – Tables des méthodes, avec partage de tables ou en héritage non virtuel : une classe fait référence à la table qui lui est associée en tant que type statique, sur la même ligne.

et  $C$  n'est pas virtuel, la classe  $A$  est dupliquée dans l'objet, c'est-à-dire que deux sous-objets  $A$  sont incorporés physiquement dans les sous-objets  $B$  et  $C$  : on parle alors d'*héritage répété*<sup>7</sup>. Dans le cas contraire, lorsque le graphe d'héritage d'une classe est une arborescence, on parlera d'*héritage arborescent*.

Dans tous les cas, il n'est alors nécessaire de créer des tables de méthodes supplémentaires qu'en cas d'héritage multiple : lorsqu'une classe à  $k$  super-classes directes, il faut  $k - 1$  tables de méthodes supplémentaires (figure 7.4). Si l'héritage est arborescent, le nombre total de tables est égal au nombre de super-classes sans super-classes.

### 7.3 En héritage simple et sous-typage multiple

Entre les deux extrêmes du sous-typage simple et de l'héritage multiple, se situe le cas intermédiaire de langages qui différencient classes et types, tout en assimilant la spécialisation de classes au sous-typage, et qui se restreignent à un héritage simple. C'est typiquement le cas de JAVA, avec des classes en héritage simple et des interfaces en sous-typage multiple. Tous les langages de la plate-forme .NET de Microsoft, dont C# et Eiffel#, ont la même stratégie de typage, de même que le langage CLAIRE.

La spécification du problème est très exactement celle de JAVA [AG97, Gra97] : les classes sont en héritage simple. Elles seules peuvent définir des attributs et le corps de méthodes, et avoir des instances. Les interfaces servent essentiellement à factoriser des signatures de méthodes.

L'implémentation des attributs est celle du sous-typage simple et `self` reste invariant puisque son type statique est toujours une classe. En revanche, la question se pose pour les entités typées par une interface. En effet, les

<sup>7</sup> [Mey97] a un usage un peu différent du terme d'héritage répété, qui désigne la figure d'héritage en losange (figures 7.2 et 7.4) : Eiffel permet alors de dupliquer certaines des propriétés de la racine du losange, après un renommage, de même qu'il permet, inversement, de fusionner des propriétés différentes héritées de diverses super-classes. L'implémentation en reste mystérieuse. Eiffel permet aussi un héritage répété d'une même super-classe directe, modulo un renommage adéquat pour éviter les conflits. Au-delà de la curiosité sémantique, ce trait semble parfaitement redondant avec la possibilité d'expansion des attributs (mot-clé `expanded`). Un constat analogue est possible avec l'héritage `private` de C++ qui a tendance à tourner à l'agrégation. Le terme de *composant* (*component*) ou de *membre* (*member*) est courant, dans les langages et les méthodes, pour désigner les propriétés (attributs ou méthodes) des classes ou des objets. Encore faut-il distinguer les deux, le logique et le physique.

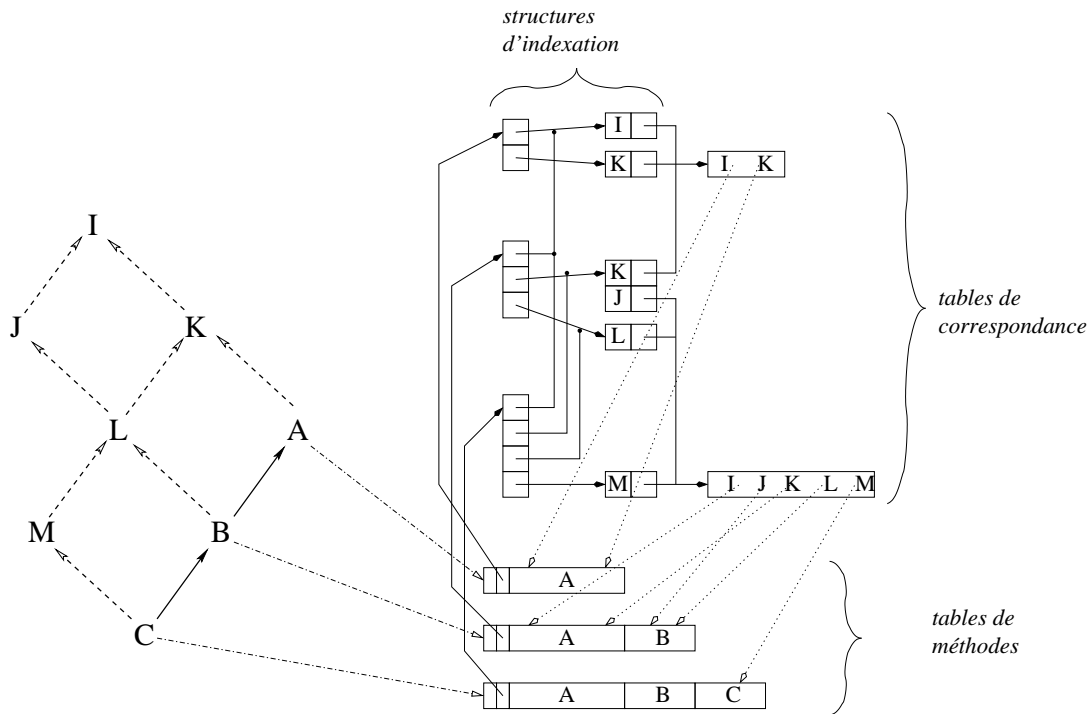


FIG. 7.6 – Sous-typage multiple et héritage simple : variante du sous-typage simple, avec tables de conversion (variante 1).

indices des méthodes sont toujours invariants par héritage mais ne le sont plus par sous-typage : les indices des méthodes d'une même interface varient donc suivant les classes qui l'implémentent. L'invariant 7.2 du sous-typage simple est donc toujours vérifié, mais pour les classes seulement. De même, la vérification de types et le *casting* descendant peuvent s'implémenter comme en sous-typage simple, par la double numérotation  $n_1/n_2$ , quand la cible est une classe.

Dans un tel contexte, l'implémentation standard de l'héritage multiple serait une complication inutile : il faut manifestement conserver les invariants du sous-typage simple, tant qu'il s'agit de classes.

On a néanmoins deux catégories de solutions, suivant que l'on se ramène au sous-typage simple, par complication, ou à l'héritage multiple, par simplification.

### 7.3.1 Variante du sous-typage simple

Au lieu de se ramener aux techniques d'implémentation de l'héritage multiple, en les simplifiant, on peut se ramener au sous-typage simple, en le complexifiant.

On conserve alors de l'héritage simple les deux invariants 7.1 et 7.2, seul le second étant restreint aux entités typées par une classe. Les tables de méthodes sont donc calculées comme en sous-typage simple, mais il faut rajouter des structures d'indirections pour les cas où une entité est typée par une interface : l'invariance des indices de méthodes ne concerne que les classes.

Pour les tables de méthodes des interfaces, plusieurs possibilités sont envisageables :

1. des tables de conversion entre les indices des méthodes dans l'interface et les indices des méthodes dans la classe (figure 7.6) ;
2. des tables de méthodes de plein droit ;
3. si les méthodes sont regroupées par classe ou interface d'introduction, un décalage dans la table de méthode suffit ;
4. ou un pointeur sur le bloc des méthodes de l'interface.

Chaque classe devra permettre d'accéder ainsi à toutes les interfaces qu'elle implémente.

Mais il n'est pas possible d'attacher cette table de correspondances à une entité particulière de façon à ce qu'elle soit accessible en temps constant : il faut rechercher la table de correspondance associée à l'interface considérée dans une structure d'indexation attachée à la table de méthodes de l'objet. La recherche de cette table de conversion est donc une sorte de *lookup* dynamique.

# Chapitre 8

## Bref manuel CLOS

### 8.1 Définition de classes et méthodes

#### 8.1.1 La macro DEFCLASS

La macro `defclass` permet de définir les classes. Sa syntaxe est la suivante :

```
(defclass <nom-de-classe>
  (<super-classe>*)
  (<description-attribut>*)
  <options-de-classe>*)
```

Chaque description d'attribut est une liste

```
(<nom-d-attribut> {<mot-cle> <expr>}*)
```

#### Exemple :

```
(defclass polygone (standard-object)
  ((list-angles :initarg :angles
                :accessor poly-angles)
   (list-sides :initarg :sides
               :accessor poly-sides)
   (side-nb :accessor poly-nb)
   (angles-sum :accessor poly-angles-sum))
  (:metaclass standard-class)
)

(defclass quadrilatere (polygone)
  ((side-nb :allocation :class
            :initform 4))
)
```

**Super-classes et méta-classe.** La super-classe par défaut est `standard-object` et la méta-classe par défaut est `standard-class`. Si l'on utilise une autre méta-classe, il faut qu'elle soit compatible avec la méta-classe de chacune des super-classes, c'est-à-dire que l'application de la fonction générique `validate-superclass` à la classe et à une super-classe retourne vrai pour chaque super-classe.

**Description d'attribut.** Chaque attribut peut être décrit par différents mots-clés, dont les principaux sont :

- des mots clés d'initialisation lors de la création d'une instance : `:initform` est suivi d'une expression dont la valeur servira à l'initialisation de l'attribut si aucune autre initialisation n'est faite ; `:initarg` est suivi par un mot-clé qui servira à introduire la valeur à affecter à l'attribut, dans la liste d'arguments des fonction génériques `make-instance` et `initialize-instance`.

Exemple : `(make-instance 'quadrilatere :sides '(1 2 3 4))`.

- des mots clés pour la définition de fonction accesseur : `:accessor` introduit le nom de la fonction qui servira aux accès en lecture et en écriture (par `setf`).

- le mot clé `:allocation` qui gouverne l'implémentation de l'attribut, dans la classe (s'il est suivi de `:class`) ou dans l'instance (s'il est suivi de `:instance`, c'est le défaut). Dans le premier cas, l'attribut est partagé par toutes les instances : modulo le fait qu'il est redéfinissable dans les sous-classes, c'est à peu près la même chose qu'un attribut `static` en C++ ou JAVA.

**NB.** Les mots-clés `:accessor` et `:initarg` ont un rôle syntaxique typiquement *méta*. Mais, leur comportement se différencie en cas d'usage multiple du même symbole :

- si deux attributs différents se partagent le même mot-clé d'initialisation, ils seront initialisés à la même valeur par l'usage de ce mot-clé ;
- si deux attributs différents se partagent le même accesseur, seul l'un des deux sera accédé ;
- inversement, un même attribut peut avoir plusieurs mots-clés d'initialisation et plusieurs accesseurs.

Le résultat est indéterminé lorsque l'on initialise une instance en utilisant plus d'un mot-clé s'appliquant à un attribut donné.

**Options de classe.** La seule qui puisse nous concerner est l'option qui permet de choisir une méta-classe particulière :

```
(:metaclass <une-metaclasses>)
```

La classe en cours de définition sera une instance directe de `<une-metaclasses>`.

En cas d'absence de cette option, c'est la méta-classe `standard-class` qui est utilisée.

**Héritage et redéfinition d'attributs.** La définition d'une sous-classe permet de redéfinir les attributs d'une super-classe. Tous les mots-clés sont redéfinissables, avec les sémantiques respectives suivantes :

- pour le mot-clé d'initialisation par défaut `:initform`, sa redéfinition masque les définitions antérieures ;
- pour le mot-clé d'initialisation `:initarg` et les accesseurs (`:accessor`, `:reader` et `:writer`), la redéfinition se cumule aux définitions antérieures : la seule contrainte est l'absence de conflit dans ces définitions, le même accesseur ou mot-clé ne pouvant pas être utilisé pour 2 attributs différents de la même classe ;
- pour le mot-clé `:allocation`, la redéfinition est un peu plus compliquée :
  - la redéfinition masque les définitions antérieures (on peut donc passer de `:class` à `:instance` et inversement),
  - mais une absence de redéfinition du mot-clé dans la redéfinition de l'attribut revient à redéfinir le mot-clé avec la valeur par défaut `:instance` ;
  - enfin, `:allocation :class` a pour portée l'ensemble des instances, directes et indirectes, de la classe, à l'exception des instances des sous-classes pour lesquelles l'attribut est redéfini : si l'on veut que l'attribut de la sous-classe soit physiquement différent, il faut redéfinir l'attribut.

Les éventuels conflits résultant de ces redéfinitions (pour `:initform` ou `:allocation`) sont résolus par la linéarisation.

### 8.1.2 Les macros `DEFMETHOD` et `DEFGENERIC`

La définition de méthode se fait par la macro `defmethod`, qui rajoute une méthode à la fonction générique de même nom. La macro `defgeneric` sert à définir cette fonction générique : son emploi est facultatif car elle est appelée automatiquement par `defmethod` en cas de besoin.

**Exemple :**

```
(defmethod initialize-instance ((p polygone) &rest initargs)
  (call-next-method)
  (when (and (slot-boundp p 'list-sides)
            (slot-boundp p 'list-angles)
            (not (= (length (poly-sides p)) (length (poly-angles p)))))
    (error "mauvais nombres de côtés et d'angles : ~s" p)))
```

`Defmethod` a la même syntaxe que `defun`. Les seules différences sont :

- il est possible de typer tous les paramètres obligatoires, en le faisant figurer dans une liste (`<paramètre>` `<classe-du-paramètre>`). L'absence de typage d'un paramètre revient à le typer par le type universel `T`. Les différentes méthodes de la même fonction générique (c'est-à-dire de même nom) se différencient par le type de leurs paramètres (au moins un).

- différents moyens permettent de « combiner » les méthodes d’une même fonction générique : parmi ces moyens, l’appel de la fonction `call-next-method` (sans argument) permet de faire appel à la méthode suivante dans la *linéarisation* (cf. chapitre 4).

**Conformité entre méthodes et fonctions génériques.** En première approximation, l’usage de `defgeneric` se réduit à indiquer la structure de la liste de paramètres : la définition de la fonction générique `initialize-instance` doit donc ressembler à

```
(defgeneric initialize-instance (o &rest initargs))
```

toute nouvelle méthode devant se conformer à cette structure. La conformité en question se résume par :

- chaque méthode a le même nombre de paramètres obligatoires que la fonction générique ;
- en première approximation, chaque méthode a les mêmes paramètres optionnels (`&optional`, `&rest`), au nom près, et les mêmes paramètres par mots-clés (`&key`)<sup>1</sup> que la fonction générique.

Lorsque `defgeneric` n’est pas appelé explicitement, le premier `defmethod` — c’est-à-dire un `defmethod` portant sur une fonction générique qui n’a pas été encore définie — provoque l’appel de `defgeneric` avec, comme arguments, la liste de paramètres de la méthode.

## 8.2 Accès aux attributs

L’accès aux attributs (ou *slots* dans la terminologie CLOS) peut se faire — de façon approximativement équivalente — au moyen d’accesseurs ou de la primitive `slot-value`.

Il est possible de définir autant d’accesseurs que l’on veut pour un attribut donnée. Soit, par exemple, la définition suivante :

```
(defclass <classname> (<superclass>*)
  ((<slot-name> :accessor <slot-accessor>
               :reader <slot-reader>
               :writer <slot-writer>)))
```

Les accesseurs `<slot-accessor>`, `<slot-reader>` et `<slot-writer>` sont définis comme des fonctions génériques, avec des méthodes pour toutes les classes concernées.

Alors, pour une instance `<object>` de `<classname>` :

```
(<slot-accessor> <object>)
(<slot-reader> <object>)
(slot-value <object> '<slot-name>)
```

sont équivalents, de même que :

```
(setf (<slot-accessor> <object>) <value>)
(<slot-writer> <object> <value>)
(setf (slot-value <object> '<slot-name>) <value>)
```

L’accès aux attributs ne dépend pas de leur allocation (mot-clé `:allocation`) et il n’est pas possible de rendre un attribut *constant* ou *en lecture seule*.

## 8.3 Fonctions utiles

**(call-next-method)** appelle la méthode suivante dans la linéarisation, en lui passant les mêmes arguments que ceux de la méthode courante : déclenche une erreur en cas d’absence de méthode suivante ;

**(class-name class)** retourne le nom (symbol) de la classe `class`.

**(class-of objet)** retourne la classe de l’objet, c’est-à-dire l’instance d’une méta-classe, pas son nom ;

**(find-class nom-de-classe)** retourne la classe, c’est-à-dire l’instance d’une méta-classe, à partir de son nom ;

**(initialize-instance (o standard-object) &rest initargs)** est la fonction d’initialisation des instances, appelée par `make-instance`.

<sup>1</sup> Avec la possibilité d’en accepter d’autres, par `&allow-other-keys`, qui est indispensable pour la redéfinition de `initialize-instance`.

- (**make-instance** (*c standard-class*) &rest *initargs*) est la fonction de création des instances, qui appelle `initialize-instance` pour les initialiser.
- (**next-method-p**) retourne vrai (`t`) s'il existe une méthode suivante dans la linéarisation, donc si l'on peut appeler `call-next-method` sans risque ;
- (**slot-boundp objet slot**) retourne vrai si l'attribut de nom `slot` de l'objet de nom `objet` a une valeur, et faux sinon ;
- (**slot-value objet slot**) retourne la valeur de l'attribut de nom `slot` de l'objet de nom `objet` (s'il existe, déclenche une erreur sinon) : cette fonction est utile si l'on n'a pas défini d'accessor (erreur de style) et surtout si `slot` est calculé ;  
on peut faire des `setf` sur `slot-value` ;
- (**type-of objet**) retourne le type de l'objet ;
- (**validate-superclass (class standard-class) (super standard-class)**) `class` est une classe et `super` l'une de ses super-classes directes : retourne vrai si les classes de `class` et de `super` — c'est-à-dire des méta-classes — sont compatibles : c'est à l'utilisateur de définir des méthodes pour chacune des méta-classes qu'il définit. Par défaut, toute méta-classe est compatible avec elle-même et incompatible avec les autres.

## 8.4 Lancement

On lance CLOS avec la commande `clos` qui lance un interprète CLISP avec le package PCL (*Portable Common Loops*), qui est l'une des implémentations portables de CLOS.

**Attention !** ne travaillez pas directement sous CLISP ni sous EMACS-LISP, qui semblent connaître CLOS : ce n'est qu'une grossière approximation pour le second. Quant au premier, s'il correspond grossièrement aux spécifications de CLOS au premier niveau, il n'implémente pas la réflexivité et le MOP de façon correcte.

Voir le polycopié de LISP pour l'utilisation de CLISP.

# Chapitre 9

## Exercices et problèmes

Les divers problèmes de modélisation et d'implémentation décrits ci-dessous forment un tout cohérent, au moins en CLOS. Les polygones offrent un terrain d'application aux autres fonctionnalités : l'inspection d'objet est utile pour leur mise au point et il est possible de mémoriser leurs instances pour constituer une base de données de polygones.

### 9.1 Les polygones

**Objectif.** Développer l'exemple esquissé au chapitre 8, en modélisant

- des polygones, avec différentes spécialisations comme les polygones avec un nombre fixe de côtés (triangle, quadrilatère, dodécagone, etc.), les polygones ayant des côtés ou des angles égaux, les polygones réguliers, et leurs diverses spécialisations communes (rectangle, carré, losange, triangle-équilatéral, etc.);
- leurs attributs principaux : liste des angles et des côtés, nombre de côtés, somme des angles<sup>1</sup>, pour tous les polygones, plus des attributs spécifiques à certaines classes, comme largeur, longueur, côté, etc.
- leurs méthodes principales, essentiellement pour le calcul de la surface et du périmètre.

On se cantonnera bien entendu aux cas particuliers géométriquement simples.

#### 9.1.1 En CLOS.

**Spécifications.** On veillera à préciser judicieusement, dans la définition des classes,

- les attributs qui doivent être initialisés par une constante (`:initform`) et qui peuvent être alloués dans la classe;
- les attributs qui doivent être initialisés lors de la création d'instance (`:initarg`);
- les attributs qui peuvent être calculés à partir d'autres.

En particulier, on définira les méthodes `initialize-instance` nécessaires à la vérification de la cohérence entre les différents attributs (par exemple, liste des angles et des côtés de même longueur, somme des angles du bon multiple de  $\pi$ , etc.), ou au calcul de certains attributs.

**Initiation à CLOS.** On commencera par définir une ou deux classes simples, créer des instances, vérifier les initialisations, et tester l'usage des principales fonctions décrites (paragraphe 8.3).

Tester par exemple les rapports entre `make-instance`, `find-class`, `type-of`, `class-name` et `class-of` : quels en sont les invariants ?

#### 9.1.2 Dans un autre langage.

On cherchera à transposer cette implémentation de CLOS vers les autres langages.

**En C++.** L'un des problèmes sera celui des attributs de classe : leur usage n'est pas aussi flexible que le « `:allocation :class` » de CLOS.

**En JAVA.** Le problème est bien entendu celui de l'héritage simple.

<sup>1</sup> Dont on se rappellera qu'elle vaut  $(n - 2)\pi$ , où  $n$  est le nombre de côtés.

**En Eiffel.**

**En Smalltalk.** Le problème est bien entendu celui de l'héritage simple.

## 9.2 Classes qui mémorisent leurs instances

### 9.2.1 En CLOS.

**Objectif.** Définir la classe `memo-class` des classes qui mémorisent leurs instances. Pour simplifier, on pourra lui associer la classe `memo-object` des objets qui sont mémorisés par leur classe.

**Spécification.** La méta-classe `memo-class` possède un attribut dont la valeur est la liste de ses instances directes.

Définir la ou les méthodes `validate-superclass` qui assure la compatibilité de `memo-class` avec `memo-object` et ses sous-classes. Attention ! il faut pouvoir définir `memo-object` avant de ne pouvoir définir comme `memo-class` que des sous-classes de `memo-object`.

Définir les méthodes `make-instance` ou `initialize-instance` des classes concernées.

Les objets mémorisés par leur classe ne peuvent plus être ramassés par le *garbage collector*, à moins d'avoir été enlevés de la liste d'instances de leur classe. Il faut donc définir une méthode `delete-object` de `memo-object` et/ou une méthode `delete-instance` de `memo-class`.

**Requêtes.** Dès que les classes mémorisent leurs instances, il est possible de s'en servir comme d'une base de données. On peut, en particulier, faire des requêtes, c'est-à-dire rechercher les objets — les instances d'une classe — qui vérifient telle ou telle propriété, par exemple pour lesquels certains attributs ont certaines valeurs.

Définir la méthode `select-instance`, qui prend en argument une classe et une liste alternée de nom d'attributs et de valeurs et qui retourne la liste des instances (directes ou indirectes) de la classe dont les attributs ont les bonnes valeurs. On pourra compliquer en acceptant des valeurs fonctionnelles, qui seront alors des prédicats à appliquer à la valeur de l'attribut.

Ainsi

```
(select-instance 'polygone
                 'side-nb 4
                 'list-angles #'(lambda (x) (member 90 x)))
```

retournera la listes de tous les quadrilatères dont l'un des angles est droit.

On pourra optimiser en tenant compte des attributs qui sont constants sur une classe : dans l'exemple précédent, tous les rectangles doivent être retournés sans avoir besoin de faire aucun test.

Enfin, pour accéder aux sous-classes de la classe argument, on aura besoin de la fonction `class-direct-subclasses` : en héritage multiple, il faudra veiller à ne pas examiner plusieurs fois certaines classes.

### 9.2.2 Dans un autre langage

L'objectif sera alors d'essayer de transposer la modélisation et l'implémentation en CLOS, si c'est possible.

**En Smalltalk.**

**En Openc++.** Il faudrait générer, dans chaque classe, un constructeur ou une méthode appelée par un constructeur, qui mémorise la nouvelle instance dans une variable `static` de sa classe (et uniquement dans celle-ci). Le problème réside dans les limitations des constructeurs en C++.

**En C++ ou Java.**

## 9.3 Inspecteur d'objets

**Objectif.** Il s'agit de définir la méthode `inspect-object` qui affiche le type de l'objet, ainsi que le nom et la valeur de chacun des attributs d'un objet.

Attention, il ne faut pas confondre l'inspecteur d'objet et l'inspecteur de classe : l'inspecteur d'objet ne s'intéresse qu'aux attributs de l'objet et à leur valeurs (ce qui distingue deux instances de la même classe), alors que l'inspecteur de classe, même appelé sur une instance, ne s'intéresse qu'à ce qui est commun : le noms des attributs (sans leurs valeurs) et des méthodes.

**Spécification en CLOS.** On pourra décomposer en trois fonction génériques :

- `inspect-object` qui inspecte son argument, n'importe quelle instance de `standard-object`, par appel de la fonction suivante ;
- `inspect-instance` qui prend en argument une classe et une instance de celle-ci, affiche le nom de la classe et inspecte chacun de ses attributs dans l'objet par la fonction suivante ;
- `inspect-slot` qui prend en argument une description d'attribut (`effective-slot-definition`) et un objet, et affiche le nom de l'attribut et sa valeur dans l'objet (en faisant attention au cas où l'attribut n'est pas initialisé).

La définition des méthodes associées va nécessiter de connaître le nom et les accesseurs de plusieurs attributs des méta-objets concernés, classes ou descriptions d'attribut. Cela crée un intéressant problème d'amorçage : si l'on disposait de la fonction `inspect-object`, il serait simple d'inspecter les classes, de découvrir les noms de leurs attributs, d'inspecter la description correspondante dans la méta-classe pour avoir l'accesseur, etc. Comme ce n'est pas le cas, il faut rechercher ces différents noms avec les moyens du bord : en l'occurrence le fait que les noms recherchés commencent respectivement par `class-` ou `slot-`. L'usage de la tabulation comme complétion de symbole en CLISP permet alors de lister toutes les fonctions concernées.

**En OPENC++.** La classe n'existant plus réellement à l'exécution, il faut générer, dans chaque classe, une méthode `inspect-object` qui fasse le travail. C'est typiquement de la compilation.

**En JAVA.** Avec le *package* `reflect`, l'inspection d'objet ne pose aucun problème.

## 9.4 Types paramétrés en CLOS

**Objectif.** Définir des types paramétrés dans un langage réflexif avec typage dynamique : par exemple, les types `liste-chainée[T]` ou `pile[T]`. Le typage reste bien entendu dynamique et les méthodes du type vérifient dynamiquement les types des arguments.

La discussion sur les types paramétrés (paragraphe 3.3) a montré leur proximité conceptuelle avec les méta-classes : la solution passera donc par la définition de classes et de méta-classes.

**Spécification.** Une façon de faire est de définir une méta-classe, par exemple `pile-class`, qui représentera le type avec son paramètre formel (`pile[T]`) et une classe `pile-object` qui représentera la classe de toutes les piles.

La création d'une instance de `pile[A]` pour une classe `A` reviendra alors à :

- regarder si une classe `pile-A`, instance de `pile-class` avec un paramètre `A` a déjà été définie<sup>2</sup> : si ce n'est pas le cas, la créer ;
- instancier cette classe `pile-A`.

Il faudra donc définir une nouvelle fonction générique de création d'instance

```
(make-parametric-instance 'pile 'A)
```

La vérification de type étant dynamique, on peut envisager d'enfreindre la règle de sous-typage des types paramétrés et d'accepter que `pile[A]` soit une sous-classe de `pile[B]` lorsque `A` est une sous-classe de `B`. Dans tous les cas, il faudra faire de `pile[A]` une sous-classe de `pile-object`.

Il faut enfin définir une fonction générale `parametric-type-p`, similaire à la fonction LISP `typep`, mais prenant un argument supplémentaire, le type du paramètre. On écrirait par exemple :

```
(parametric-type-p val 'pile 'A)
```

<sup>2</sup> Il pourrait donc être utile, si les classes ne le font pas déjà, de faire de ces méta-classes des classes qui mémorisent leurs instances.

pour vérifier que `val` est du type `pile[A]`. Une telle vérification doit se faire en passant par la classe et la méta-classe. On peut la faire de façon équivalente, avec une méthode `stackp` de `pile-object` :

```
(stackp val 'A)
```

On peut ensuite généraliser pour passer à des types paramétrés sur plusieurs paramètres, par exemple le type `dico[T,U]` des dictionnaires — ou structures d'indexation, table de hachage par exemple — dont la clé est de type `T` et la valeur de type `U`.

**En SMALLTALK.**

## 9.5 Variables de classes, en C++ ou JAVA

La notion d'attribut (ou variable) « de classe » est délicate car elle recouvre au moins 3 notions différentes suivant les langages :

- les variables `static` de C++ et JAVA dont la sélection est statique (appelées « attribut statique » dans la suite) ;
- les attributs d'instances dont la valeur est partagée par toutes les instances d'une classe (et de ses sous-classes) et qui sont alloués dans la classe (mot-clé `:allocation :class` de CLOS) (appelées « attribut de classe » dans la suite) ;
- les attributs de la classe considérée comme une instance de méta-classe, dans un modèle réflexif à la CLOS (appelées « attribut de méta-classe » dans la suite).

Aucune de ces 3 notions ne se réduit directement à une autre, mais il est possible de simuler, à la main, le fonctionnement de l'une par l'autre.

### 9.5.1 Exemple

Soit 3 classes  $A$ ,  $B$  et  $C$  telles que  $C \prec B \prec A$ , et toutes 3 instances de la même méta-classe  $M$ . Soit 3 attributs  $p$ ,  $q$  et  $r$  tels que :  $p$  est un attribut de méta-classe de  $M$ ,  $q$  est un attribut de classe de  $A$ , et  $r$  est un attribut statique de  $A$ .  $q$  et  $r$  sont en plus « redéfinis » dans  $C$ . Soit enfin 3 objets  $a$ ,  $b$  et  $c$ , instances respectives de ces 3 classes.

### 9.5.2 Simulation en C++ ou JAVA

1. Valuer ces attributs (par une valeur entière par exemple) de telle sorte que deux valeurs identiques traduisent un partage d'attribut (alternativement, faites un schéma explicitant les zones mémoire affectées à ces différentes entités) ;
2. Avec une syntaxe à la JAVA, montrer sur cet exemple la différence de comportement entre ces 3 catégories d'attributs, au niveau syntaxique, en montrant comment on y accède depuis une instance de  $A$  (ou de ses sous-classes) ;
3. au niveau sémantique, en montrant quel attribut est accédé suivant les types statiques et dynamiques de l'objet (faites par ex. un tableau).

Montrer comment il est possible, avec les attributs statiques, de simuler « à la main » le comportement :

1. de l'attribut de méta-classe  $p$  ;
2. de l'attribut de classe  $q$ .

Dans les 2 cas, donner le code C++ ou JAVA nécessaire (déclaration d'attributs statiques et code éventuel de méthodes), dans les 3 classes  $A$ ,  $B$  et  $C$ , pour simuler ces 2 attributs.

## 9.6 Modélisation d'animaux et d'espèces animales

La réflexivité des objets est souvent présentée avec un objectif d'implémentation des langages : dans ce cas, les méta-objets ne servent qu'à décrire des entités du langage.

Pourtant, la notion de méta-classe peut trouver une application dans le « monde réel ». La modélisation des espèces animales et des animaux en offre un bon exemple. Les trois plans de la figure 5.2 contiennent alors respectivement, des animaux individuels (mon chat, milou, etc.), des espèces animales (le chat, le tigre, etc.) ainsi que des super-classes (vertébrés, mammifères, etc.), et des classes d'espèces animales (les espèces en voie de disparition, disparues, menacées, etc.).

## 9.7 Simulation de la redéfinition covariante avec la surcharge statique

En C++ et JAVA la surcharge statique est souvent confondue avec la redéfinition. Il est néanmoins possible de simuler la redéfinition covariante avec la surcharge.

Spécifier une méthodologie de programmation permettant de remplacer la redéfinition covariante par une redéfinition invariante accompagnée d'une surcharge, de telle sorte que le comportement des appels de méthodes soit en tout point identique à celui qui résulterait d'une surcharge statique.

Spécifier la méta-programmation en OPENC++ ou OPENJAVA permettant de générer automatiquement ce couple redéfinition-surcharge.

# Chapitre 10

## Glossaire

Le vocabulaire des objets est très (trop) riche, de nombreuses notions se voyant attribuer des noms différents suivant les auteurs et les langages, pour ne pas parler des langues. Ce glossaire ne vise pas à recenser tout le vocabulaire du domaine, mais seulement à définir précisément les principales notions.

**accesseur** nom donné aux méthodes d'accès aux attributs.

**attribut** propriété d'un objet de type donnée. Un attribut peut toujours se ramener à deux méthodes (ses *accesseurs*), de lecture et d'écriture.

**classe** entité de base du modèle objet, qui regroupe les objets similaires (extension) et factorise leurs propriétés communes (intension).

**coercition** l'hypocrisie du typage sûr : son seul usage objet est un test de sous-typage dynamique (*downcast*) dont l'usage devrait être strictement encadré.

**constructeur** fausse méthode et faux-ami : le constructeur est en fait une méthode d'initialisation des instances, lors de l'instanciation.

**contravariance** indique simplement une évolution monotone, en sens contraire (contravariance des intensions). La *règle de contravariance* impose une contravariance des types de paramètres.

**covariance** indique simplement une évolution monotone, dans le même sens (covariance des extensions). La *règle de contravariance* impose une covariance des types de retour.

**domaine** ensemble des valeurs prises par une propriété (attribut, paramètre ou retour de méthode).

**dynamique** désigne de façon générale ce qui se fait à chaud, en présence de valeurs, à l'exécution.

**encapsulation** le fait que les données d'un objet ne puisse être accédées que par l'interface de ses méthodes. Stricte en SMALLTALK, elle se décline en Eiffel par des clauses d'exportation qui en conservent le sens. En Java et en C++ elle est remplacée par des droits d'accès (*private*, *public*, etc.).

**extension** ensemble des instances d'une classe, son interprétation. Le terme est aussi utilisé dans la littérature (mais pas ici) pour désigner la définition d'une sous-classe relativement à sa super-classe.

**généricité** désigne ici le fait de pouvoir définir des types ou classes paramétrés par un type. La généricité peut être *bornée*, voire *F-bornée*.

**héritage** notion duale de la spécialisation, caractérisée par l'inclusion des intensions. Se décline en héritage *simple* ou *multiple* et, dans ce dernier cas, s'analyse par un héritage de *nom* et de *valeur*.

**instance** un objet, vu comme « instance de » sa/ses classe(s). L'instance peut être *directe* (ou *propre*) ou *indirecte*.

**instanciation** procédé par lequel les instances sont créées à partir d'une classe. Le terme est aussi utilisé pour désigner le fait de substituer une valeur à une variable, par exemple pour l'instanciation des classes paramétrées. Les deux sens s'y rejoignent, puisque cette dernière peut être vue comme l'instanciation d'une méta-classe.

**intension** ensemble des propriétés d'une classe.

**invariance** La *règle de contravariance* impose une invariance des types des attributs.

**liaison tardive** le fait qu'un appel de méthode (envoi de message) ne puisse être résolu qu'à l'exécution, d'après le type dynamique du receveur.

**masquage** phénomène associé à la redéfinition, qui fait que la propriété redéfinie masque ses définitions dans les super-classes.

**méta** le fait d'en parler, on est déjà au niveau méta supérieur. Se décline en *méta-langage* (langage sur un langage), *méta-modèle* (modèle d'un modèle), *méta-programmation* (programmation au niveau méta) et bien sûr *méta-classe* (classe dont les instances sont des classes). Les niveaux méta forment une pile potentiellement infinie, ou une boucle vertigineuse, la **réflexivité**. Les *méta-objets* constituent l'ensemble des objets du méta-modèle permettant une méta-programmation d'un langage, au travers d'un *meta-object protocol*.

**méthode** propriété d'un objet de type procédure.

**objet** c'est l'objet de tout ça.

**overloading** voir **surcharge**.

**overriding** voir **redéfinition**.

**polymorphisme** caractérise le fait qu'une variable puisse prendre des valeurs de type différents. Se décline en polymorphisme *d'inclusion*, associé au sous-typage, et polymorphisme *paramétrique* associé à la généralité.

**propriété** la troisième notion primitive, avec les classes et les objets. Les propriétés appartiennent aux objets mais sont factorisées dans les classes. Il y en a deux catégories, les attributs (données) et les méthodes (procédures). Leur bonne compréhension nécessite de les modéliser avec des propriétés *génériques* et des propriétés *locales*.

**receveur** le paramètre singulier qui « reçoit le message » et qui doit l'interpréter suivant son type dynamique. C'est en général une pseudo-variable, *self*, *this* ou *current* suivant les langages.

**réflexivité** quand le **méta** devient vertigineux : les méta-objets sont des objets comme les autres, qui existent à l'exécution et la font.

**réification** (du latin *res*, la chose) faire des objets de n'importe quoi. Le méta-modèle objet est la réification du modèle objet.

**sélection de méthode** c'est une problématique en soi, liée à la redéfinition et à la surcharge statique. La sélection peut aussi être *multiple*.

**slot** (en anglais, la fente de la machine à sous) nom donné aux attributs dans les langages où leur accès s'effectue au moyen de primitives du langage, comme *slot-value* en CLOS.

**sous-typage** le pendant pour les types de la spécialisation pour les classes : se caractérise par la substituabilité.

**spécialisation** se définit comme l'inclusion des extensions.

**static** comme *virtual*, caractérise ce qui n'est pas objet, mais en JAVA aussi.

**statique** désigne de façon générale ce qui se fait à froid, en l'absence de valeur, à la compilation.

**substituabilité** le principe à la base du sous-typage : le sous-type peut remplacer partout le super-type sans erreur de type à l'exécution.

**super** L'appel à *super* désigne un mécanisme qui permet à la redéfinition d'appeler la méthode redéfinie. En SMALLTALK et JAVA, c'est une pseudo-variable dont la sémantique dépasse l'objectif. En EIFFEL ou CLOS c'est un mécanisme proprement défini.

**surcharge** terme passablement « surchargé » désignant le fait qu'un nom puisse avoir plusieurs interprétations suivant le contexte. A réserver à la **surcharge statique**.

**surcharge statique** le fait que le même nom de propriété puisse être utilisé dans le contexte de la même classe, la différence se faisant par les types et le nombre des paramètres (pour les méthodes).

**type** entité proche des classes caractérisée par un ensemble d'opérateurs et, de façon duale, un ensemble de valeurs. On doit distinguer le type *statique*, annotation dans les programmes, du type *dynamique*, type des valeurs à l'exécution.

**typage** désigne une approche ou un système de type. Il peut être *statique*, *dynamique* ou *sûr*.

**variable de classe** désignation ambiguë, soit des attributs des classes (c'est-à-dire des instances de méta-classes), soit des attributs des objets qui sont factorisés dans les classes ( *:allocation :class* en CLOS), soit enfin, très improprement, des variables statiques en C++ ou JAVA.

**variable d'instance** nom donné aux attributs dans les langages comme SMALLTALK, JAVA ou C++ où les attributs sont accédés sous la même syntaxe que les variables ou paramètres.

**virtual** et son adjectivation (*virtuel*) : c'est toute la différence entre C++ et la programmation par objets.

# Bibliographie

- [AG97] K. Arnold and J. Gosling. *The JAVA programming language, Second edition*. Addison-Wesley, 1997.
- [DEMN98] R. Ducournau, J. Euzenat, G. Masini, and A. Napoli, editors. *Langages et modèles à objets : État des recherches et perspectives*. Collection Didactique. INRIA, 1998.
- [ES90] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading (MA), USA, 1990.
- [GJS96] J. Gosling, B. Joy, and G. Steele. *The JAVA Language Specification*. Addison-Wesley, 1996.
- [GMP96] M. Gautier, G. Masini, and K. Proch. *Cours de programmation par objets. Applications à Eiffel et comparaison avec C++*. Masson, Paris, 1996.
- [GR83] A. Goldberg and D. Robson. *Smalltalk-80, the Language and its Implementation*. Addison-Wesley, Reading (MA), USA, 1983.
- [Gra97] M. Grand. *JAVA Language Reference*. O'Reilly, 1997.
- [Hab96] B. Habert. *Objectif : CLOS*. Masson, Paris, 1996. (à la BU de l'UM2).
- [HC99] C. S. Horstmann and G. Cornell. *Au cœur de JAVA*, volume 1. Campus Press, France, 1999.
- [Hof85] D. Hofstadter. *Gödel, Escher, Bach, les brins d'une guirlande éternelle*. InterEditions, Paris, 1985.
- [KdB91] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge (MA), USA, 1991.
- [Kee89] S.E. Keene. *Object-Oriented Programming in Common Lisp. A Programmer's Guide to CLOS*. Addison-Wesley, Reading (MA), USA, 1989.
- [Kle71] S.C. Kleene. *Logique mathématique*. Collection U. Armand Colin, Paris, 1971.
- [Lip96] S.B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, New York (NY), USA, 1996.
- [MD97] J. Meyer and T. Downing. *JAVA Virtual Machine*. O'Reilly, 1997.
- [Mey92] B. Meyer. *Eiffel : The Language*. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, UK, 1992.
- [Mey94] B. Meyer. *Eiffel, le langage*. InterEditions, Paris, 1994. Traduction française de [Mey92].
- [Mey97] B. Meyer. *Object-Oriented Software Construction*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, second edition, 1997.
- [Mic01] Microsoft. *C# Language specifications, v0.28*. Technical report, Microsoft Corporation, 2001.
- [MNC<sup>+</sup>89] G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Les langages à objets*. InterEditions, Paris, 1989.
- [Ste90] G.L. Steele. *Common Lisp : The Language, Second Edition*. Digital Press, Bedford (MA), USA, 1990.
- [Str98] B. Stroustrup. *The C++ programming Language, 3<sup>e</sup> ed.* Addison-Wesley, 1998.
- [Str03] B. Stroustrup. *Le langage C++, édition spéciale, revue et corrigée*. Pearson Education, Addison Wesley, Paris, 2003.