Master Mention Informatique
Spécialité AIGLE

Support de cours des modules :

HLIN 603	Objets avancés
GMIN 310	Méta-Programmation et Réflexivité
FMIN 314	Langages à Objets à Typage Statique
GMIN 31D	Spécification et Implémentation des Logiciels à Objets et Composants

Programmation Par Objets : des Concepts Fondamentaux

à leur Application dans les Langages

Millésime 2016

Roland Ducournau

Département d'Informatique
—
Faculté des Sciences – Université de Montpellier

8 septembre 2017

Remerciements. Ce polycopié est le fruit d'une dizaine d'années de cours de Maîtrise et de Master d'Informatique, accompagnés de projets et d'examens, ainsi que de nombreuses discussions avec des étudiants, doctorants, enseignants et chercheurs.
La matière de plusieurs parties provient assez directement des sujets de projets ou d'examens soumis aux étudiants. Plusieurs promotions d'étudiants ont donc servi de cobayes. Qu'ils en soient ici remerciés.

Je caresse parfois l'idée de les consigner pour en faire un livre à léguer à la postérité... mais à ce jour c'est resté un projet. Car, primo, quand le désir de célébrité est éteint, on devient nonchalant ; secundo, [...] font notre bonheur, mais écrire un livre est un pénible labeur ; tertio, une fois mort, je ne pourrais plus lire quiconque ; quarto enfin, ce qu'on écrit cette année, on est sûr de le regretter l'année prochaine !

Shi Nai-an, 施耐庵, préface à « Au bord de l'eau »

1

La programmation par objets

Tout ce que vous avez toujours voulu savoir sur l'objet,
sans avoir jamais osé le demander
d'après wOOdy allen.

Prérequis.

Ce manuel est un cours avancé de programmation par objets. Il est destiné à des étudiants de Master connaissant déjà bien la programmation par objets — au travers d'un ou plusieurs langages, par exemple C++ ou Java— ainsi que la modélisation Uml.

Comme tout cours d'informatique et bien que ce cours ne soit pas très formel, il suppose bien entendu des bases minimales en mathématiques discrètes (algèbre, ensembles, graphes, ordres, logique) et en algorithmique.

Etudiants de L3. Pour les étudiants de l'UE *Objets avancés* de L3, il est conseillé de se focaliser sur les chapitres 2, 5, 6 et 7.

Objectifs.

Ce cours part des concepts pour aller vers les langages. Il se focalise sur les concepts originaux, souvent délicats, de la programmation par objets : les concepts sont d'abord élaborés abstraitement puis sont ensuite confrontés à leur réalisation dans les différents langages. En revanche, ce cours passera sous silence tout ce qui n'est pas propre à la programmation par objets, et que l'on retrouverait aussi bien en programmation fonctionnelle ou impérative.

Outre l'aspect technique, on s'intéressera aussi aux aspects méthodologiques : l'approche objet est très riche et les divers mécanismes qu'elle propose sont pour partie redondants. Le passage d'un même modèle objet assez abstrait à un programme exécutable dans un langage donné présente donc de nombreux degrés de liberté. Le programmeur doit connaître les avantages et les inconvénients de chaque approche, pour pouvoir choisir en connaissance de cause.

Ce manuel ne s'intéresse *pas*, de façon systématique,

— à la *syntaxe*, qui est forcément propre à chaque langage ; il adoptera donc une syntaxe abstraite suffisante pour exprimer sans ambiguïté les concepts analysés ;
— aux *spécifications* détaillées de tel ou tel langage, sauf sur des points très spécifiques et dans un but comparatif avec l'approche abstraite proposée ici ;
— à la *méthodologie* de développement de programmes objet, même si divers éclairages méthodologiques seront apportés sur des points particuliers.

Ce cours n'est pas que théorique et les abstractions étudiées dans ce manuel seront mises en pratique dans divers langages (Java, C++, C♯, Eiffel, Clos, Prm, Python, etc.). Au lecteur de se reporter donc, sur ces divers langages, aux nombreux documents disponibles, qu'il s'agisse de polycopiés des modules de Licence consacrés à Uml, Java et C++, ou des manuels et tutoriaux publiés sur le Web.

Professionnel ou recherche ? La plupart des points abordés dans ce cours intéressent au plus au point le praticien, qui devra développer des applications de grande taille — ou simplement participer à leur développement. Mais comme toujours en informatique, les problématiques de recherche ne sont jamais très éloignées. Aussi, si le contenu de ce cours n'est pas spécifiquement dédié à la recherche, de nombreuses références seront faites aux problématiques de recherche et les références bibliographiques sont pour beaucoup empruntées à la recherche.

Remarque 1.1. Il s'agit ici de la réécriture presque totale et de l'approfondissement d'un premier polycopié rédigé au tournant du III^e^ millénaire. Les principes ont assez peu évolué mais la présentation en est grandement changée. Enfin, cette réécriture n'est pas achevée et certaines parties peuvent se trouver encore en chantier.

1.1 Caractérisation de la programmation par objets

La programmation par objets présente — dans son modèle standard, celui des *langages de classes* [Masini et al., 1989] — quatre caractéristiques originales, qui vont par deux et permettent de distinguer deux familles de langages.

1.1.1 Langages à objets

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 (appelées *attributs*) 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. Voir, à ce propos, la section 8.2.

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 anthropomorphique et « 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++, JAVA et C♯ 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. l'accès aux attributs).

Langages à prototypes 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* : SELF, GARNET, CECIL en ont été des représentants successifs, parmi d'autres, et JAVASCRIPT en est le représentant les plus éminent actuellement.

1.1.2 Langages à classes

Les langages à prototypes 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. Notons que ce découpage entre ce qui est propre à l'objet et ce qui est regroupé dans la classe est arbitraire : c'est le résultat d'un compromis entre *efficacité* et *expressivité*. Certains langages, CLOS par exemple, ne font pas le même compromis et permettent aussi bien d'implémenter des attributs dans la classe que de spécifier des méthodes pour des

objets individuels. 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.

Ces deux notions de classe et de spécialisation vont de pair. On pourrait éventuellement imaginer un « vrai » langage à objets, avec envoi de message et des classes mais sans spécialisation : mais cela n'aurait de sens qu'en typage dynamique car, en typage statique, tous les appels de méthodes se résoudraient statiquement.

Cette dernière assertion est vraie si l'on considère la spécialisation dans son sens le plus large. Cependant, si l'on distingue spécialisation de classes et sous-typage (cf. Chapitre 5), comme en Java, on peut imaginer un langage sans spécialisation de classes mais avec sous-typage et une vraie liaison tardive. D'ailleurs, un tel langage existe : Emerald [Raj et al., 1991, Jul, 2008]. Le langage Go présente, au moins sur ce point, des caractéristiques similaires.

1.1.3 Langage à objets = objet + message + classe + héritage

Dans le langage courant, le terme de « langage à objets » désigne en général les langages à classes et ces 4 caractéristiques sont essentielles pour constituer un « langage à objets ». Certains langages ne satisfont ces critères que partiellement ou marginalement, au sens où la programmation objet n'y constitue que l'une des façons de programmer offertes par le langage : c'est le cas de Clos, C++, C♯, Objective-C ou Ada 95 (et Ada 2005). D'autres sont au contraire, comme Smalltalk, Eiffel ou Prm/Nit, de « purs » langages à objets : toute valeur y est un objet, instance d'une classe, et l'envoi de message est la seule abstraction procédurale.

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 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 : son respect des grands principes du génie logiciel et son adéquation avec notre représentation du monde. D'une certaine manière, le confluent de ces deux approches constitue l'épicentre de notre réflexion.

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 du monde. 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. Il s'agit, au sens propre, d'un outil pour constituer une *ontologie* du monde [Bunge, 1977], et il ne faut pas s'étonner de ce que nous trouverons quelques exemples éclairants en *astrophysique*.

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

1.2.2 Orientation de ce cours

> *In addition to its support for encapsulation and code reuse, an important contribution of object-oriented programming to software engineering is that object-oriented programming is*

a synthesis of programming and knowledge representation.
I.R. Forman et S.H. Danforth [1999].

Cette citation résume parfaitement notre approche. La programmation par objets a un double héritage :

— d'un côté, les langages de programmation dont l'un des jalons, ALGOL, a ouvert la voie au premier langage à objets, SIMULA [Birtwistle et al., 1973], ancêtre de C++ et BETA ;
— d'un autre côté, les langages de représentation des connaissances, à base de réseaux sémantiques ou de *frames*, qui ont été la source de très nombreuses surcouches objet de LISP, le langage de l'intelligence artificielle ;
— entre les deux, le langage SMALLTALK a été le premier langage à objets « pur » en proposant une synthèse des concepts objet introduits dans SIMULA, dans un environnement dont la philosophie venait tout droit de LISP [Kay, 1993].

De la représentation à la modélisation. Un courant important des langages à objets, basé sur la notion de *frame*, s'est intéressé à la représentation des connaissances et à l'utilisation du modèle objet dans une approche de *classification*. Voir par exemple les chapitres correspondants de [Masini et al., 1989, Ducournau et al., 1998].

Nous n'y ferons référence que de façon très marginale car ces langages ne sont plus directement utilisables. En pratique, ces dernières années, c'est la modélisation et UML qui ont repris le flambeau de la représentation.

Contradictions. On verra, en particulier au chapitre 5, que ces deux catégories de motivations et d'héritage conduisent à une contradiction un peu insurmontable : en effet, la fiabilité requiert une sûreté du typage dont la théorie impose malheureusement qu'elle soit en partie incompatible avec la sémantique de la spécialisation.

D'autres contradictions apparaissent dans certains langages. Par exemple, C++ a été conçu comme une synthèse de la programmation par objets et de la programmation C, c'est-à-dire un assembleur de haut niveau. Les deux objectifs sont complètement antinomiques : la programmation par objets repose sur le postulat que l'objet décide de son propre comportement. A l'opposé, la programmation en assembleur ou en C permet au programmeur d'intervenir chirurgicalement sur une donnée au niveau du bit.

Enfin, des contradictions d'un autre ordre peuvent apparaître : ce double point de vue sur les objets n'est pas partagé par tous les spécialistes. Ceux qui viennent du domaine des langages de programmation, en particulier les théoriciens des types, ne reconnaissent pas forcément la légimité de ce double héritage.

1.2.3 Problématiques abordées ici

Les problématiques abordées dans ce cours couvrent donc pour l'essentiel :

— l'approfondissement des quatre caractéristiques clé des langages à objets ;
— l'étude des zones de friction entre les problématique de type génie logiciel et de type représentation des connaissances.

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 aussi être multiple : c'est le cas de JAVA et de C♯ [Microsoft, 2001], le langage de la plate-forme .NET de Microsoft.

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. La *généricité* et les *types paramétrés* posent eux aussi divers problèmes.

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 suivant le type dynamique de tous les arguments. La métaphore de l'envoi de message devient alors inappropriée et l'on parle dans ces langages de *fonctions génériques*.

Les systèmes d'exécution des langages sont devenus essentiels dans leur spécification et leur mise en œuvre. Ainsi, JAVA est indissociable de sa machine virtuelle, de même que C++ est fortement contraint pas son modèle de compilation et la distinction des fichiers `.h` et `.cpp`.

L'implémentation des mécanismes objets constitue une explication concrète des mécanismes fondamentaux. Elle est par ailleurs tributaire de la présence de certaines fonctionnalités comme le typage (statique ou dynamique), l'héritage (simple ou multiple) ou enfin le chargement dynamique.

La modélisation d'objets complexes. Une vision naïve relativement courante des objets consiste à les considérer comme des structures dont les champs ont des valeurs atomiques primitives. A contrario, on appelait *objet complexe* des objets dont les valeurs des champs peuvent elles-mêmes être des objets complexes. La cible principale de la modélisation par objets réside dans ces objets complexes et tous les objets doivent être considérés comme tels. Les objets sont en relation, et l'une des difficultés consiste à modéliser et implémenter ces relations qui sont modélisées en UML par des associations. Le problème se complique encore plus lorsqu'il s'agit de spécialiser ces associations.

La méta-modélisation et la réflexivité. Dans les langages interprétés, à typage dynamique, comme SMALLTALK ou CLOS, les classes doivent exister à l'exécution et il est naturel de procéder à un double mouvement de *réification*, en faisant des classes des objets « de première classe », et d'abstraction, en regroupant les classes en méta-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++ ou OPENJAVA—, la manipulation de *bytecode* — JAVASSIST ou ASM— 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 : c'est la base d'UML et on l'utilisera ici par exemple pour analyser la problématique de l'*héritage multiple*.

Le méta et la réflexivité sont devenus des traits essentiels de l'approche objet, que l'on rapprochera d'une caractérisation de la langue par l'un des grands linguistes du XX° siècle :

> *Je crois que la principale différence entre la langue et les "systèmes sémiotiques" est qu'aucun système sémiotique n'est capable de se prendre lui-même comme objet ni de se décrire dans ses propres termes.*
> E. Benveniste, in *Dernière Leçons. Collège de France (1968-69)* EHESS/Gallimard/Seuil, 2012.

Si l'on considère qu'un langage de programmation par objets est un « système sémiotique » mais n'est pas une langue, on ne peut malheureusement que lui donner tort, à moins de considérer les langages de programmation comme des langues, ce qui n'était probablement pas l'intention de Benveniste. Noter que ce cours date des débuts de Simula, donc de quelques années avant Smalltalk : on pourrait dire que les faits ne lui avaient pas encore donné tort. Cela dit, LISP et ses méta-évaluateurs existaient déjà.

Droits d'accès et visibilité. Parmi les nombreuses problématiques qui ont leur place dans l'approche objet mais qui ne seront abordées ici que superficiellement, citons la question de la *visibilité* des (ou du *droit d'accès* aux) propriétés — ou de façon équivalente celle de la *protection* ou de l'*encapsulation* des objets — qui joue sur le fait que certaines propriétés peuvent être déclarées `public` ou `private`, ou réservées au receveur courant, 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.

1.2.4 Autres problématiques

Modularité, aspects, etc. En étroite relation avec la problématique de la *modularité*, la problématique objet s'est enrichie ces dernières années de la notion de *séparation des préoccupations* qui a conduit à isoler une *programmation par aspects*. Ce sont des questions importantes mais elles ne seront pas abordées par manque de place et de temps : il y faudrait un module supplémentaire. Une autre problématique connexe est celle de la *modularité* : les langages objets font souvent reposer l'unité de code sur la classe, mais certains rajoutent des notions de *package* (Java) ou de *classes internes* (encore Java), alors que d'autres comme C++ ne considèrent que des fichiers non structurés qui contiennent la définition d'une ou plusieurs classes.

Programmation fonctionnelle. La programmation par objets est pour partie contraire à la programmation fonctionnelle au sens où programmer avec des objets relève beaucoup d'une programmation impérative où l'on fait des effets de bord sur l'état des objets. Mais elles sont aussi en partie orthogonales, dans le sens sens où l'on peut faire de la programmation par objets en manipulant des fonctions en tant qu'objets de première classe. De nombreux langages à objets présentent actuellement des constructions syntaxiques proches des λ-expressions en Lisp, avec des syntaxes assez variées pour remplacer le constructeur `lambda`. C'est par exemple le cas de C++ 2011 [Rigault, 2011].

Cependant, rares sont les langages dont la construction `lambda` joue le rôle de véritable *fermeture* comme en Lisp. A part Clos, OCaml et les autres surcouches objet de langages fonctionnels, Scala semble être l'un des rares à proposer des fermetures. Cependant, à partir de λ-expressions il est très facile de faire desz fermtures à la main en passant par des objets créés à cet effet.

Aspects système. D'autres problématiques plus proches du système, comme la concurrence ou la persistance, ne seront pas plus abordées ici, malgré leur intérêt pratique évident.

Un aspect pratique primordial est celui de la gestion mémoire. Tous les langages à objets dignes de ce nom disposent d'une gestion mémoire automatique, avec un *garbage collector* qui gère la récupération de la mémoire non utilisée de façon absolument transparente pour l'utilisateur. Une fois encore, l'unique exception est représentée par C++, pour lequel nous indiquerons, au passage, comment combler cette lacune. Du fait de cet automatisme quasi-universel, la gestion de la mémoire ne sera considérée, dans la suite, que dans les cas où elle a un impact sur d'autres fonctionnalités.

1.3 Des langages de programmation par objets

1.3.1 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 chargement dynamique ou à édition de liens statique ;
— 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 car la frontière est souvent beaucoup moins nette. Ainsi, entre la compilation et l'interprétation, la compilation dans une machine virtuelle (*byte-code*), comme Java, est un intermédiaire. Un langage apparemment interprété comme Smalltalk ou Lisp peut très bien être compilé, en réalité.

Le typage peut aussi être statique sans que les annotations de type soient explicites : c'est le cas des langages de la famille ML, OCaml par exemple, où les types statiques sont inférés. L'héritage simple des classes peut aussi être complété par un héritage (ou plutôt sous-typage) multiple des interfaces, comme en Java et C♯.

Enfin, la surcharge statique de C++ et Java peut déguiser leur sélection simple en une fausse sélection multiple.

1.3.2 Langages examinés

L'objectif du cours est d'examiner a priori tous les langages qui présentent des fonctionnalités dignes d'intérêt ou de critiques autour d'une des problématiques d'héritage multiple, de typage statique, de

langages	interprétés typage dynamique	compilés typage statique
sans méta-niveau		C++ Eiffel
avec introspection	Dylan	Java `java.lang.reflect`
avec méta-niveau compilation chargement		OpenC++ OpenJava Javassist, Asm
réflexif	Smalltalk Clos Python	

Figure 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	*templates*
Java	comp./mv	statique	multiple	simple	`reflect` OpenJava Javassist	non	simple	1.5
Scala	comp./mv	statique	multiple	*mixin*	`reflect`	non	simple	oui
Fortress	?	statique	multiple	*mixin*	?	?	multiple	?
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	—
OCaml	interp/comp	infér. type	multiple	simple	?	?	simple	oui
Python	interp ?	dynamique	—	multiple	oui	oui	simple	—
Ruby	interp.	dynamique		*mixin*	oui	oui ?		—
Swift	comp ?	statique	multiple ?	*mixin*				oui
Objective-C	comp	dynamique	—	simple	oui	oui	simple	—
Ada 2005	comp.	statique	multiple	simple	non	non	simple	oui

Figure 1.2 – Caractéristiques des différents langages

sélection de méthodes ou de méta-niveau. Cela inclut tous les grands langages historiques comme Smalltalk, C++, Eiffel, Clos.

C♯ et les langages de la plate-forme .Net semblent des clones de Java : la plus grande partie de ce qui concerne Java s'applique à eux. Cependant, étant arrivés après Java, les concepteurs de ces langages ont su éviter certaines erreurs, par exemple pour l'intégration des types primitifs (arrivée en Java 1.5 seulement), pour l'héritage multiple ou la généricité.

D'autre langages comme Python sont abordés avec un point de vue très critique et une réticence certaine à cause des nombreux défauts de leurs spécifications.

1.3.3 Langages non (ou peu) considérés

De nombreux langages ne seront pas *a priori* considérés pour des raisons variables :

— certains supposent un *background* important car la programmation par objets n'est pas leur trait dominant et que les étudiants du Master montpelliérain n'en ont pas la culture : c'est le cas en particulier d'OCaml ou Ada ; le problème est différent avec Clos car Lisp (ou Scheme) est déja au programme de la Licence et du cours de compilation de M1 ;

— certains langages comme Dylan ou IlogTalk/Power-Classes semblent avoir disparu corps et biens, malgré leur intérêt certain ; Dylan serait néanmoins repris en logiciel libre.

— Fortress est en langage dédié au calcul scientifique haute performance qui présente de nombreux traits intéressants : son développement serait en cours d'abandon par Oracle.

— certains ne présentent tout simplement aucun des traits étudiés dans ce cours : Objective-C, par exemple, est en héritage simple, typage statique et son unique méta-classe est réservée à l'implémentation ;

— de nombreux autres langages sont vraisembablement dignes d'intérêt, mais il est difficile de se maintenir au courant de tout : les étudiants sont invités à partager leur savoir ou leur curiosité...

— les langages de la famille Beta sont à mettre dans cette rubrique : ils sont issus de Simula et à l'origine de nombreux concepts intéressants mais leur style est très exotique et ils ont très peu pris en France.

Si des étudiants sont intéressés par un de ces langages écartés parce qu'ils ont ce *background* particulier, qu'ils se signalent.

1.4 Méthodologie de programmation par objets

Ce cours n'est pas un cours de méthodologie. Cependant, un point de vue méthodologique implicite nous servira de guide tout au long des thèmes abordés : le point de vue de la *réutilisabilité*. L'un des apports fondamentaux de la programmation par objets est d'avoir mis ce critère en avant.

En pratique, on peut distinguer deux rôles dans le développement des programmes :

— le développeur de bibliothèques va développer des unités de code cohérentes et réutilisables, dont il ne se sert pas forcément lui-même ; il est donc chargé de leur *définition* ;

— le développeur d'applications va réutiliser les classes de bibliothèques existantes pour développer ses propres applications : il n'est concerné que par l'*utilisation* de ces biblothèques.

Si ces deux rôles sont distincts, cela ne signifie pas qu'il faille en faire des fonctions distinctes. Le développeur d'applications va développer ses propres bibliothèques qui pourront être ensuite être réutilisées par le développeur d'une autre application.

S'il y a donc un continuum dans les fonctions, il est bon d'analyser les langages et leurs fonctionnalités suivant ces deux rôles de la *définition* des classes (ou autres entités) et de leur *utilisation*. En particulier, il n'est pas sain que les spécifications d'un langage fasse reposer sur l'utilisateur l'expression de ce qui aurait pu être exprimé à la définition.

La distinction entre ces deux rôles sera particulièrement utile dans l'analyse de la généricité (Section 7) qui représente l'une des composantes les plus difficiles de la programmation par objets. Sous le critère de la difficulté, il est généralement admis que la définition est largement plus difficile que l'utilisation. Là encore, la généricité en est un bon exemple.

La cible principale de ce cours est donc le développeur de bibliothèques, dont les classes sont conçues pour être à la fois efficaces, et simplement réutilisables et extensibles.

1.5 Bibliographie commentée

La littérature sur la programmation par objets est à la fois très vaste et très restreinte. Les livres consacrés à des langages particuliers sont très nombreux mais il y a peu de manuels généraux qui s'intéressent aux concepts indépendamment des langages. La plupart des ouvrages à prétention méthodologique se préoccupent surtout de promouvoir un langage particulier. En ce sens, le présent document est une rareté [1].

Bibliographie générale

[Masini et al., 1989] et [Ducournau et al., 1998] sont des ouvrages de synthèse sur l'approche objet, assez datés, mais qui ont le mérite de faire un panorama complet des différentes sortes d'objets et des problématiques associées. [Gautier et al., 1996] est, comme son nom l'indique, un cours de programmation par objets, prenant comme support les langages C++ et Eiffel pour les comparer.

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. Bien entendu, la bibliographie n'est qu'indicative pour les langages hégémoniques comme C++ et Java. 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/` (accessible en intranet uniquement) propose des pointeurs sur des documents concernant la plupart des langages abordés dans ce cours.

C++ est un langage délicat à utiliser car sa compatibilité avec la philosophie objet est variable : s'il faut vraiment s'en servir, l'étude de son implémentation est indispensable [Ellis and Stroustrup, 1990, Lippman, 1996].

1. L'auteur laisse au lecteur le soin de développer le classique paradoxe réflexif qui va avec cette publicité elle aussi réflexive.

langage	référence	autres	Web
Smalltalk	[Goldberg and Robson, 1983]		http://www.objectshare.com/ http://www.oti.com/links/smaltalk.htm
Common Lisp	[Steele, 1990]		file:/net/doc/cltl/clm/clm.html http://www.lispworks.com/documentation/common-lisp.html http://clisp.cons.org/
Clos	—	[Kiczales et al., 1991] [Keene, 1989, Habert, 1996]	http://ftp.sunet.se/ftp/pub/lang/lisp/oop/clos/
Python	[van Rossum and Drake, 2003]		http://www.python.org/
Dylan	[Shalit, 1997]		http://www.opendylan.org/
Swift			https://swift.org/
Eiffel	[Meyer, 1994] [Meyer, 2001]	[Meyer, 1997]	http://www.eiffel.com/ http://se.ethz.ch/~meyer/#Progress
Smart Eiffel			http://smarteiffel.loria.fr/
C++	[Stroustrup, 1998] [Stroustrup, 2003]	[Ellis and Stroustrup, 1990] [Lippman, 1996]	
C++ 2011			http://www.polytech.unice.fr/~jpr/c++2011
OpenC++			http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/openc++.html
Java	[Arnold and Gosling, 1997]	[Grand, 1997] [Gosling et al., 1996] [Horstmann and Cornell, 1999]	
OpenJava			http://www.csg.ci.i.u-tokyo.ac.jp/openjava/
Javassist			http://www.javassist.org/
Asm			http://asm.ow2.org/
Scala	[Odersky et al., 2004] [Odersky et al., 2008]		http://www.scala-lang.org
Fortress	[Allen et al., 2008]		http://research.sun.com/projects/plrg/Fortress/
D			http://d-programming-language.org/
C♯	[Microsoft, 2001]		
Objective-C	[Apple]	http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/	
OCaml	[Hickey, 2006]		http://caml.inria.fr/ocaml/
Beta	[Madsen et al., 1993]		http://www.daimi.au.dk/~beta/
Ada 2005	[Taft et al., 2006]		
Prm			http://www.lirmm.fr/prm
Nit			http://nitlanguage.org

Figure 1.3 – Bibliographie et sites Web par langage.

Bibliothèque en ligne

Le livre de B. Meyer OOSC_2 [Meyer, 1997] est disponible en intranet au LIRMM (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. Une nouvelle édition de [Meyer, 1994] en cours de rédaction est disponible sur le site http://se.ethz.ch/$\sim$meyer/#Progress.

Le manuel de référence de Common Lisp et Clos est lui aussi disponible en ligne sur le réseau du campus (file:/net/doc/cltl/clm/clm.html) [2]. Il est aussi accessible sur le web et une version alternative se trouve sur le site de Lispworks.

1.6 Plan

Toutes les parties de ce document sont étroitement et récursivement liées et l'ordre de présentation est forcément arbitraire. Comme le lecteur est censé connaître tous les concepts de base, leur présentation approfondie peut être retardée, et la lecture séquentielle reste possible [3]. Néanmoins, des aller-retour entre les différents chapitres seront probablement utiles.

La première partie présente les fondements : les classes, la spécialisation et l'héritage. Le chapitre 2 traite de la spécialisation et de l'héritage de classes de façon abstraite et logique, en remontant à Aristote. Il est éclairé par le chapitre 3 qui formalise les notions de classes et de propriétés sous la forme d'un méta-modèle Uml. Ce méta-modèle est la clé d'une bonne compréhension de l'héritage multiple (Chapitre 4). Cette première partie intéresse tous les étudiants des modules considérés.

La seconde partie s'intéresse ensuite au typage. Le chapitre 5 présente le typage statique et analyse son rapport avec la spécialisation de classes. Le chapitre suivant examine en détail l'appel de méthodes, les différents variantes de sélection de méthodes, en particulier la *surcharge statique* et la *sélection multiple*, ainsi que la question des droits d'accès. La chapitre 7 s'intéresse ensuite à la généricité (les *templates* de C++), sous la forme principale des classes ou types paramétrés. La chapitre 8

2. Ce n'est malheureusement plus vrai, mais le document est facilement trouvable sur le web.
3. On a là un bon exemple d'*amorçage* (Chapitre 11) !

décrit plusieurs fonctionnalités spécifiques qui mettent en jeu les chapitres précédents. Le chapitre 9 décrit schématiquement l'implémentation des langages en typage statique, ce qui peut éclairer les spécifications des langages comme C++ ou JAVA. La partie se termine par une rapide présentation du langage PRM qui présente des réponses à certaines des questions soulevées précédemment (Chapitre 10). Cette seconde partie intéresse en priorité les étudiants qui suivent le module sur le typage statique.

Une dernière partie s'intéresse enfin à la réflexivité, en particulier au travers du langage CLOS et de surcouches de JAVA. Le chapitre 11 approfondit les notions liées à la méta-modélisation, déjà abordée au chapitre 3, à la méta-programmation et à la réflexivité, le tout dans le cadre de langages interprétés, à typage dynamique. Le chapitre 12 est une rapide présentation du langage CLOS. Le chapitre 13 présente ensuite les différentes formes de méta-programmation dans le contexte de langages à objets en typage statique, non interprétés. Le chapitre 14 termine cette partie par un recueil d'exercices de niveau méta. Cette troisième partie concerne surtout les étudiants qui suivent le module de méta-programmation.

Tout au long de ce cours, des exercices proposent divers problèmes au lecteur : certains sont faciles, voire traités quelques pages à côté, d'autres sont plus difficiles, ils peuvent même cacher des pièges (le traitement n'est pas possible dans tous les langages : la dénonciation de l'impossibilité, preuve à l'appui est une bonne réponse) ou constituer des conjectures qui ont paru « évidentes » à l'auteur en cours de rédaction, mais qui ne le sont pas forcément.

Le manuel se termine par un glossaire (Chapitre 16) et un index qui rend ce document aussi proche que possible d'un hypertexte. Il est ainsi conseillé, dans un usage intensif, de combiner le document papier et le document numérique (disponible à l'adresse `http://www.lirmm.fr/~ducour/Publis/objets2.pdf`), pour bénéficier de l'hypertextualité du `pdf` et de la possibilité d'y effectuer des recherches systématiques.

Bibliographie

E. Allen, D. Chase, J. Hallett, V. Luchanco, J.-W. Maessen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, Inc., 2008.

Apple. The Objective-C 2.0 programming language. Technical report, Apple Inc., 2008.

K. Arnold and J. Gosling. *The JAVA programming language, Second edition.* Addison-Wesley, 1997.

G. Birtwistle, O. Dahl, B. Myraug, and K. Nygaard. *Simula begin.* Petrocelli Charter, 1973.

M. Bunge. *Ontology I : The Furniture of the World*, volume 3 of *Treatise on Basic Philosophy.* Reidel, Boston, 1977.

R. Ducournau, J. Euzenat, G. Masini, and A. Napoli, editors. *Langages et modèles à objets : État des recherches et perspectives.* Collection Didactique. INRIA, 1998.

M.A. Ellis and B. Stroustrup. *The annotated C++ reference manual.* Addison-Wesley, Reading, MA, US, 1990.

I. R. Forman and S. H. Danforth. *Putting Metaclasses to Work.* Addison-Wesley, 1999.

M. Gautier, G. Masini, and K. Proch. *Cours de programmation par objets. Applications à Eiffel et comparaison avec C++.* Masson, Paris, 1996.

A. Goldberg and D. Robson. *Smalltalk : the language and its implementation.* Addison-Wesley, 1983.

J. Gosling, B. Joy, and G. Steele. *The JAVA Language Specification.* Addison-Wesley, 1996.

M. Grand. *JAVA Language Reference.* O'Reilly, 1997.

B. Habert. *Objectif : CLOS.* Masson, Paris, 1996. (à la BU de l'UM2).

Jason Hickey. Introduction to the Objective Caml programming language, 2006. URL `http://www.cs.caltech.edu/courses/cs134/cs134b/book.pdf`.

C. S. Horstmann and G. Cornell. *Au cœur de JAVA*, volume 1. Campus Press, France, 1999.

E. Jul. Precomputing method lookup. In *Workshop ICOOOLPS at ECOOP'08*, 2008.

A.C. Kay. The early history of Smalltalk. In J.A.N. Lee and J.E. Sammet, editors, *Proc. 2nd History of Programming Languages Conference, HOPL-II.* ACM Press, 1993.

S.E. Keene. *Object-Oriented Programming in* COMMON LISP. *A programmer's guide to* CLOS. Addison-Wesley, Reading, MA, 1989.

G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, New York, 1996.

O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Les langages à objets*. InterEditions, Paris, 1989.

B. Meyer. *Eiffel : The Language*. Prentice-Hall, 1992.

B. Meyer. *Eiffel, le langage*. InterEditions, Paris, 1994. Traduction française de [Meyer, 1992].

B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

Bertrand Meyer. An Eiffel tutorial. Technical report, ISE, 2001.

Microsoft. C# Language specifications, v0.28. Technical report, Microsoft Corporation, 2001.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nicolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, Lausanne, CH, 2004.

Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, A comprehensive step-by-step guide*. Artima, 2008.

R.K. Raj, E. Tempero, H.M. Levy, A.P. Black, N.C. Hutchinson, and E. Jul. Emerald : a general-purpose programming language. *Software Practice and Experience*, 21(1) :91–118, 1991.

J.-P. Rigault. What's new in C++ 2011 ?, 2011. URL http://www.polytech.unice.fr/~jpr/c++2011.

A. Shalit. *The Dylan Reference Manual : The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, 1997.

G. L. Steele. *Common Lisp, the Language*. Digital Press, second edition, 1990.

B. Stroustrup. *The C++ programming Language, 3^e ed.* Addison-Wesley, 1998.

B. Stroustrup. *Le langage C++, édition spéciale, revue et corrigée*. Pearson Education, Addison Wesley, Paris, 2003.

S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, and P. Leroy, editors. *Ada 2005 Reference Manual : Language and Standard Libraries*. LNCS 4348. Springer, 2006.

Guido van Rossum and Fred L. Drake, Jr. *The Python Language Reference Manual*. Network Theory Ltd, September 2003. ISBN 0-9541617-8-5.

Première partie

Les objets, les classes, la spécialisation et l'héritage

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. Ce sont pourtant des notions bien connues, à la base de la pensée occidentale depuis Aristote.

2.1 Classes, objets et propriétés

Le modèle objet est globalement constitué de trois catégories d'entités : les classes, les objets et les propriétés.

2.1.1 Classes et instances

On en dira pour l'instant le moins possible, les choses se dévoilant progressivement en analysant la spécialisation. Disons simplement qu'une *classe* a des *instances*, qui sont des objets. Un objet est *instance de* une ou plusieurs classes : on parle alors de relation d'*instanciation* entre les instances et les classes. D'un point de vue terminologique, « instance » est un terme relationnel qui fait référence à une classe. La relation « instance de » est analogue à la relation ensembliste « appartient à ». Cette relation d'instanciation entre un objet et sa classe sera analysée plus en détail dans le chapitre 11 sur la réflexivité.

En attendant, on insistera sur le fait qu'une classe n'existe finalement qu'au travers de ses instances. Il est donc primordial de prévoir leur construction et tout modèle (diagramme) de classes doit impérativement s'accompagner d'un *protocole* de construction des instances, qui spécifie la façon dont doivent être construits et mis en relation les différents objets constituant un diagramme d'instances cohérent.

Dans cette optique, la conception doit donc s'appuyer sur 3 étapes principales : un diagramme de classes, un diagramme d'instances et le protocole de construction.

*****(à développer...)*****

Remarque 2.1. Les objets sont créés par instanciation d'*une* (unique) classe, et on parle alors de *mono-instanciation*, par opposition à une hypothétique *multi-instanciation* qui se rencontre plutôt dans des langages assez exotiques et dont nous ne parlerons pas. En pratique, la multi-instanciation passe souvent par la définition d'une classe anonyme, à la volée. C'est le cas, en particulier, en Scala.

2.1.2 Classes et propriétés

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 plusieurs sortes — attributs (données) et méthodes (procédures ou fonctions) sont communs à tous les langages, mais certains y rajoutent des types, dits *virtuels* — 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 qui est commune à toutes les instances. Cette asymétrie n'est motivée que par des considérations d'efficacité.

2.1.3 Classes et modèles de classes

A ce stade, il n'est pas inutile de revenir sur quelques-chose qui est sans doute de l'ordre de l'évidence pour qui programme dans des langages à la JAVA, ou même à la CLOS, ou surtout qui modélise avec UML.

Méta-axiome 2.1 (Classes et modèles) *Une classe est d'abord un modèle*[1] *de ses instances. Elle doit donc décrire, de façon* déclarative*, les propriétés de ses instances. En particulier, toutes ses instances doivent posséder les mêmes propriétés, attributs ou méthodes.*

En corollaire, il n'est pas nécessaire d'exécuter un programme, ni même de disposer du code source du programme, pour savoir ce qu'une instance a dans le ventre. La documentation, par exemple, suffit.

Comme il l'a été dit plus haut, et de façon tautologique, cela exclut du champ de cette étude les langages à objets dits *sans classes*, ou à prototypes, comme SELF ou JAVASCRIPT, dans lesquels on ne définit jamais de classes, sous aucune forme.

Mais cela exclut aussi des langages fort prisés des programmeurs, comme PYTHON ou RUBY. Dans ces langages, il n'y a en effet aucune déclaration des attributs, mais uniquement leur affectation. En RUBY, cette affectation est privée et s'effectue forcément dans le corps de la classe : une analyse statique permettrait donc d'en extraire le modèle. En revanche, en PYTHON, l'affectation peut s'effectuer depuis n'importe où dans le programme, ce qui fait que la structure des instances d'une classe dépend des programmes dans lesquels la classe est utilisée.

Par ailleurs, au-delà de l'absence de modèle et de vision conceptuelle des classes, cette possibilité est une source d'erreurs innombrables : une coquille sur le nom de l'attribut ne sera jamais détectée, même à l'exécution.

2.2 Sémantique de la spécialisation

Les trois entités de base du modèles 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.

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[2]. On généralisera en disant que :

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

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

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

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, celle qui l'a construite.

L'implication (2.1) 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 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. On pourra pas exemple se baser sur ce méta-axiome, qui fixe un objectif méthodologique :

1. Pour la petite histoire, en YAFOOL, langage développé par l'auteur de 1985 à 1994, la construction syntaxique utilisée pour définir des classes s'appelait `defmodel`.
2. 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.

Méta-axiome 2.2 (Réutilisabilité) *La définition d'une classe doit maximiser sa réutilisabilité et son extensibilité future : elle doit donc être conçue pour pouvoir être spécialisée par d'autres programmeurs sans difficultés particulières.*

*En corollaire, la définition d'une classe doit être interprétée suivant l'*hypothèse du monde ouvert.

Il est donc raisonnable de considérer que *Ext* est injective : $Ext(A) = Ext(B) \Rightarrow A = B$. Sous cette condition [3], une conséquence formelle de l'inclusion des extensions est :

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

Remarque 2.2. 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 ... Cependant, l'inclusion de l'Axiome 2.1 est plus simple à comprendre : elle est imposée pour toute définition de l'extension !

2.2.2 Les propriétés et l'intension des classes

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. Dans le syllogisme de Socrate, `mortel` peut être considéré alternativement comme une propriété. 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, le syllogisme se traduit alors ainsi :

Axiome 2.2 (Propriétés des instances) *Toute propriété d'une classe s'applique à ses instances. Plus formellement : soit une classe A, $p \in Int(A)$ une propriété de A et $x \in Ext(A)$ une instance de A, alors $p(x)$ est légal.*

Remarque 2.3. Depuis les débuts de la théorie des ensembles, la dualité des notions de prédicats et d'ensembles est bien connue. Le programmeur objet la pratique aussi, puisqu'il a souvent le choix entre définir explicitement une sous-classe ou se contenter d'étiqueter un sous-ensemble de ses instances par un attribut.

2.2.3 Héritage des propriétés

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* les propriétés de A.

Proposition 2.2 (Héritage) *La sous-classe hérite des propriétés de ses super-classes :*

$$B \prec A \Longrightarrow Int(A) \subseteq Int(B) \tag{2.2}$$

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 qu'*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.

La pratique du programmeur objet poursuit la tradition Aristotélicienne en reprenant la définition de classes par *genre et différence* (en latin *genus et differentia*) : la super-classe et les propriétés héritées constituent le genre et les propriétés propres à la classe, la différence. Cependant, les classes de la programmation par objets ne sont des *définitions* au sens strict car ce ne sont pas des conditions nécessaires et suffisantes. Par exemple, en mathématiques, un `carré` est un `rectangle` dont les `longueur` et `largeur` sont égales. Mais si l'on définit `carré` comme une sous-classe de `rectangle`, une instance de `rectangle` dont les `longueur` et `largeur` sont égales ne sera pas pour autant une instance de `carré`. En programmation par objets, les définitions ne sont que des conditions nécessaires.

3. Les classes paramétrées (Chapitre 7) offrent néanmoins un contre-exemple à cette hypothèse qui n'est en réalité valable que lorsque les classes sont créées par des programmeurs. Lorsqu'elles sont générées par un programme — c'est un peu le cas de la généricité — l'hypothèse n'est plus valide. Cela étant, la conclusion que l'on en tire reste valable mais ce n'est plus une conséquence logique stricte.

2.2.4 Spécialisation des domaines

Les propriétés peuvent être valuées : dans ce cas, la *valeur* de la propriété dans une instance de la classe est contrainte à appartenir à un *domaine*[4].

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

Proposition 2.3 (Inclusion des domaines) *Lorsque l'on spécialise une classe, on spécialise aussi le domaine des propriétés de la classe :*

$$B \prec A \Longrightarrow Dom(B,p) \subseteq Dom(A,p) \tag{2.3}$$

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[5] :

$$Dom(A,p) = \{p(x) \mid x \in Ext(A)\} \tag{2.4}$$

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.

2.2.5 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* car elles varient dans le sens de la spécialisation ;
— les intensions se spécialisent de façon *contravariante* car elles varient en sens inverse ;
— les domaines se spécialisent de façon *covariante*.

Le chapitre 5 présente un usage plus spécifique de ces termes dans le cadre du typage statique.

Remarque 2.4. La signification de covariance/contravariance est essentiellement relative et la mathématique ne reconnaît d'ailleurs que le terme de monotonie, c'est-à-dire de morphisme d'ordres. Intensions et extensions sont effectivement contravariantes, car elles varient en sens inverse les unes des autres, pour la même relation d'ordre, l'inclusion ensembliste. En revanche, le choix de ce qui est covariant par rapport à la spécialisation est arbitraire et, si ce sont les extensions que l'on qualifie de covariantes, c'est parce que c'est leur inclusion qui donne sa sémantique à la spécialisation.

2.2.6 Méta-héritage et transitivité

Quelques lignes plus haut, nous avons essayé de trouver une justification à la transitivité de la relation de spécialisation qui restait malheureusement très informelle.

Curieusement, cette propriété trouve sa justification dans l'héritage, par une sorte de *méta-héritage* (Sur le sens de *méta*, voir le chapitre 11).

En effet, le fait qu'une classe B ait une super-classe A peut être considérée comme une propriété de B, qui est en elle-même héritable. Ainsi si C est une sous-classe de B, C hérite de B le fait que A est une super-classe, ce qui assure la transitivité.

Le cas des interfaces de Java A première vue, et c'est comme cela que nous les considérerons tout au long de ce document, les *interfaces* de Java sont des classes particulièrement abstraites, qui sont « implémentées » par des classes, et transitivement par toutes leurs sous-classes.

Cependant, cette vision logique ne correspond pas complètement à leur spécification en Java : en effet, certaines interfaces comme `serialisable` ne sont pas héritables. Les concepteurs du langage ont manifestement mésestimé cette transitivité implicite, en surchargeant la spécification des interfaces, par celle de mots-clés spécifiques qui ne relèvent pas de la propriété héritable.

Cela explique aussi, sans doute, le rôle spécifique que C♯ attribue aux interfaces *directement* implémentées.

4. 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*.

5. Cela revient à considérer la propriété comme une fonction *surjective* sur son co-domaine. Bien entendu, la définition de ce domaine devient aussi délicate que celle de l'extension (Remarque 2.2, page 21), mais comme pour l'extension l'inclusion tient quelque soit la définition !

2.3 Interprétation ensembliste

Ce court chapitre a introduit un élément clé. Le modèle objet s'interprète essentiellement de façon ensembliste et les classes ne sont guère que des extensions, c'est-à-dire des ensembles d'objets. Tout se résume alors en termes d'appartenance et d'inclusion.

De plus, un élément essentiel de la modélisation objet réside dans les associations UML, qui peuvent s'interpréter comme des relations au sens ensembliste du terme, c'est-à-dire des parties du produit cartésien des extensions des classes reliées par l'association. Dans le cas binaire, une association entre les classes A et B est donc une partie de $Ext(A) \times Ext(B)$.

Avec l'approche objet, concevoir et programmer reviennent à définir « en intension » ces ensembles et ces relations binaires. Mais si, en mathématiques, les ensembles sont constitués d'éléments qui sont d'une certaine manière « déjà là », en programmation par objets, les objets ou les tuples des relations doivent être explicitement construits.

2.4 Classes et modularité

La problématique de la *modularité* est essentielle en génie logiciel et l'histoire des langages de programmation peut se voir comme sa quête.

2.4.1 La classe comme unité de code

Conceptuellement, la modularité s'exprime par deux éléments :

— la définition des unités de code qui forment des *modules* à partir desquelles on va construire des applications,

— et la protection du contenu de ces modules vis-à-vis du monde extérieur, pour réduire le couplage entre modules.

La problématique de la protection, qui est propre au typage statique, sera examinée en Section 6.8, et la présente section va considérer la problématique de l'unité de code.

Dans les langages à objets, l'unité de code est en général la classe. Comme d'habitude, ce principe presque universel donne lieu à de nombreuses variations, avec deux grandes directions : l'unité de code peut être supérieure à la classe, au contraire inférieure, ou panacher les deux.

Classes Java. Ainsi, en Java ou Scala, il serait plus exact de dire que l'unité de code est la classe englobante, les classes internes ne formant pas des unités de code au sens de la modularité (voir ci-dessous, Section 2.4.2).

Fichiers C++. Du côté de C++, le code est organisé en fichiers (`.h` et `.cpp`). Les fichiers `.h` peuvent déclarer le schéma de plusieurs classes et le code d'une classe peut-être disséminé dans plusieurs fichiers `.cpp`.

Packages Java Les langages comme Java, Scala ou C♯ proposent aussi une notion de *package* dont on pourrait croire qu'il s'agit de modules. En fait il n'en est rien, car ce n'est qu'un système de rangement. Ce ne sont pas des unités de code mais des étagères ou ranger les unités de code (c'est-à-dire les classes).

Avec les *packages*, on retrouve ainsi la même problématique de rangement et de protection (Section 6.8) qu'avec les propriétés statiques (Section 6.10).

Modules Prm/Nit Le langage Prm (ou Nit, voir Section 10.4) propose de véritables modules, qui définissent un ensemble de classes. Ces modules sont organisés dans une hiérarchie de dépendance entre modules. Mais ces modules permettent de définir des classes par raffinement successifs [Ducournau et al., 2007].

Fichiers Clos. Dans les surcouches objet de Lisp comme Clos, le code est organisé en fichiers non structurés, dont la seule contrainte est qu'ils doivent pouvoir être chargés et compilés dans un certain ordre : la surclasse avant la sous-classe, la méta-classe avant la classe, la classe et la fonction générique avant les méthodes. On retrouve donc des dépendances sans circuits entre ces fichiers, mais elles restent implicites. Voir Chapitre 12.

2.4.2 Variations sur les classes internes

Jusqu'ici nous avons considéré que les classes étaient indépendantes les unes des autres, la spécialisation exceptée. Plusieurs langages à objets proposent cependant diverses variations sur l'emboîtement de classes. Tous ces langages sont en typage statique, et cela semble bien nécessaire pour que cette problématique ait un sens.

Classe interne. Une première variation consiste à déclarer une classe à l'intérieur de la déclaration d'une autre classe. La justification première relève de la *modularité*, pour regrouper les éléments de programmes qui vont ensemble, tout en cachant éventuellement la classe interne aux autres éléments de programme. Cela peut donc être aussi une affaire de *portée*, et on retrouve alors la problématique de la visibilité et des droits d'accès (Section 6.8).

Classe interne, statique ou pas. Dans un langage comme JAVA, la classe interne peut être déclarée `static`, ou pas. Si elle est statique, ses constructeurs et ses instances se comportent normalement, c'est-à-dire sans lien particulier avec les instances de la classe englobante. En revanche, si la classe interne n'est pas statique, ses instances sont créées avec une syntaxe de type `x.new` en référence à une instance `x` de la classe englobante qui peut ensuite être accédée depuis la classe interne avec une syntaxe de type `Englobante.this`. Dans ce cas, la classe interne implémente en plus une sorte d'agrégation.

Classe locale. En plus des classes internes, JAVA propose des *classes locales* qui sont alors déclarées à l'intérieur d'une méthode et ne peuvent pas être statiques.

Classe anonyme. Finalement, une classe locale peut même être anonyme, sa déclaration se faisant paresseusement à l'occasion d'un `new`. C'est une façon, par exemple, d'instancier (apparemment) des interfaces.

SCALA se sert ainsi des classes anonymes pour faire ce qui ressemble à de la *multi-instanciation* : il est ainsi possible d'instancier une classe avec un ou plusieurs *mixin* (Section 4.6).

Classe virtuelle. Finalement, il existe encore une dernière (?) variété de classe interne, sous la forme des *classes virtuelles* offertes par un langage comme GBETA. Il s'agit alors d'un mélange de classe interne et de type virtuel (Section 7.9), la classe interne pouvant être *redéfinie*, au sens de la redéfinition dans l'héritage, dans une sous-classe.

2.5 Autres approches de la spécialisation et de l'héritage

> Inheritance is commonly regarded as the feature that distinguishes object-oriented programming from other modern programming paradigms, but researchers rarely agree on its meaning and usage.
>
> A. Taivalsaari [1996]

Dans ce chapitre, nous avons fait deux hypothèses majeures, mais de nature très différente : une sémantique extensionnelle (Aristotélicienne) de la spécialisation et de l'héritage, et l'hypothèse du monde ouvert. Ces deux hypothèses ne sont pas forcément universellement admises. [Baker, 1991] est ainsi une critique ironique du besoin d'ouverture de CLOS.

Héritage d'implémentation. 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`.

Un exemple fameux tiré des spécifications de CLOS fait d'un gâteau (`pie`) une sous-classe de fruits (`apple`) et d'épices (`cinnamon`) [Steele, 1990, page 784sq] !

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 ![6]

Il est néanmoins nécessaire de distinguer la réalité représentée de sa représentation. Bien entendu, dans la réalité, une personne n'est pas une chaîne de caractères[7]. Cependant, dans la représentation,

6. Avoir ou être, là est la question ! Lacan l'avait bien dit, à propos d'autre chose...
7. Enfin, pas toutes !

tout se passerait comme si c'était le cas [8]. D'un point de vue méthodologique, il faut garder cette sémantique de spécialisation à l'esprit, quitte à l'enfreindre en connaissance de cause dans des circonstances particulières.

Classes concrètes et feuilles de la hiérarchie. Il est souvent dit dans les méthodes de conception et de développement que seules les feuilles de la hiérarchie peuvent avoir des instances directes. Une classe aurait des sous-classes ou des instances directes, mais pas les deux à la fois. Cela se traduit par le slogan : *"make all non-leaf classes abstract"* [Meyers, 1996, Steimann, 2000].

Ce n'est pas absolument incompatible avec l'hypothèse du monde ouvert, à condition de ne pas en déduire qu'une nouvelle classe ne peut pas avoir de sous-classe parce qu'elle est concrète. La démarche consiste au contraire à définir toute nouvelle classe comme abstraite, en lui associant une unique sous-classe concrète dont la description est vide.

Classes finales. De nombreux langages permettent de spécifier qu'une classe ne pourra plus être spécialisée, par exemple avec le mot-clé `final` en JAVA, `sealed` en C♯ ou `frozen` en EIFFEL. Cela présente souvent des intérêts en matière d'efficacité, mais les limites à la réutilisation sont en général difficiles à mesurer a priori.

Le langage SCALA en présente une variante intéressante : une classe déclarée `sealed` ne peut pas avoir d'autres sous-classes directes que celles qui sont déclarées comme classes internes. Cela permet de spécifier les sous-classes directes de façon exhaustive. Par exemple, on implémentera les listes chaînées, à la LISP, avec une classe `List` déclarée `sealed`, et deux sous-classes internes `EmptyList` et `Cons`, qui devront être publiques et statiques. Cela n'empêche pas de définir, par la suite, différents types de cellules.

2.6 En savoir plus

Sur les rapports entre Aristote et les objets, voir par exemple : [Rayside and Campbell, 2000a,b, Rayside and Kontogiannis, 2001].

Pour une interprétation « logique » et « ensembliste » des hiérarchies de classes, le lecteur intéressé pourra aussi se reporter à [Ducournau, 1996] et à toute la littérature sur les *logiques de description* : [Woods and Schmolze, 1992, Donini et al., 1996]. Ces approches considèrent les classes comme des *conditions nécessaires et suffisantes* — alors qu'elles ne sont que nécessaires dans les langages de programmation usuels — ce qui permet de concevoir des mécanismes d'inférence basés sur la *classification*.

Bibliographie

H.G. Baker. CLOStrophobia : its etiology and treatment. *ACM OOPS Messenger*, 2(4) :4–15, 1991.

F.-M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 191–236. CSLI Publications, Stanford (CA), USA, 1996.

R. Ducournau. Des langages à objets aux logiques terminologiques : les systèmes classificatoires. Technical Report LIRMM-96030, Université Montpellier 2, 1996. (240 pages).

R. Ducournau, F. Morandat, and J. Privat. Modules and class refinement : a metamodeling approach to object-oriented languages. Technical Report LIRMM-07021, Université Montpellier 2, 2007.

S. Meyers. *More Effective C++*. Addison-Wesley, 1996.

D. Rayside and G.T. Campbell. An Aristotelian understanding of object-oriented programming. In *Proc. OOPSLA'00*, SIGPLAN Not. 35(10), pages 337–353. ACM, 2000a.

D. Rayside and G.T. Campbell. An Aristotelian introduction to classification. In M. Huchard, R. Godin, and A. Napoli, editors, *ECOOP'2000 Workshop on Objects and Classification, a natural convergence*. RR LIRMM 2000-095, 2000b.

D. Rayside and K. Kontogiannis. On the syllogistic structure of object-oriented programming. In *Proc. of ICSE'01*, pages 113–122, 2001.

8. Il faudrait ici faire des distinguos subtils : certains langages n'autorisent pas une parfaite *substituabilité* (Chapitre 5). L'héritage `private` de C++ est typique de cette approche.

G. L. Steele. *Common Lisp, the Language*. Digital Press, second edition, 1990.

Friedrich Steimann. Abstract class hierarchies, factories, and stable designs. *Commun. ACM*, 43(4) : 109–111, 2000.

A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3) :438–479, 1996.

W. Woods and J. Schmolze. The KL-ONE family. *Computers and Mathematics with Applications*, 23 (2-5) :133–177, 1992.

3

Méta-modèle des classes et propriétés

Eu égard à la difficulté de transcender notre patron de pensée orienté vers l'objet, il convient plutôt de l'examiner de l'intérieur.

W.V. Quine, *Parler d'objets*, in [Quine, 1977]

L'approche objet va d'abord s'appliquer à la modélisation des « objets » concrets du monde réel, à la Aristote. Mais toute la puissance de l'approche découle de la possibilité de *réifier* des notions abstraites pour en faire des objets.

On réifiera donc les diverses entités, concrètes ou abstraites, d'un domaine d'application, par exemple le secteur bancaire, avec ses guichets et distributeurs de billets (concrets) et ses comptes bancaires (abstraits), mais aussi ses procédures (virements). Mais le domaine d'application peut être lui-même abstrait : ce peut être, par exemple, le domaine des langages informatiques lui-même. On s'intéresse ici à la *réification du modèle objet*.

3.1 Sur le sens de méta

On rentre alors dans le monde étrange du *méta*. Au chapitre précédent, l'analyse de la spécialisation et de l'héritage reposait sur les notions abstraites de classes et de propriétés. Pour concrétiser ces notions et en donner une sémantique assez formelle, rien de tel que de les réifier en en faisant un modèle objet, qui sera un *méta-modèle*.

Aristote et la métaphysique.

> Rappelons que *meta* est une préposition grecque ($\mu\varepsilon\tau\grave{\alpha}$) 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 — $\mu\varepsilon\tau\grave{\alpha}\ \tau\grave{\alpha}\ \phi\upsilon\sigma\iota\kappa\acute{\alpha}\ \beta\iota\beta\lambda\iota\acute{\alpha}$.
>
> J. F. Perrot, in [Ducournau et al., 1998]

Le sens et l'origine du mot méta-physique convergent ainsi vers un au-delà de la physique qui n'est qu'une coïncidence car elle résulte d'un contre-sens.

Langage et 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 [1].

Ainsi, la logique distingue classiquement (voir [Kleene, 1971]) :

— son langage, par exemple celui de la logique des propositions dont les formules « $A \wedge B$ » et « $A \to B$ » font partie ;

— les méta-langages qui permettent de parler de ces formules, qu'ils soient informels, par exemple « la formule $A \wedge B$ est vraie », ou formels, par exemple la formule du *modus ponens* dans le formalisme de la déduction naturelle :

$$\frac{A \qquad A \to B}{B} \tag{3.1}$$

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

1. 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.

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 de la même catégorie. 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. On aboutit ainsi à une « pile méta » dont chaque niveau a son méta-niveau.

Méta-modèle objet. Dans la suite, on ne considérera que le cas où le méta-niveau est lui-même objet : on veut analyser et modéliser le modèle objet à l'aide du modèle objet. Cela n'a rien d'exceptionnel et d'original : il suffit d'imaginer vouloir implémenter un compilateur d'un langage objet dans un langage objet, par exemple le même.

Le chapitre 11 développera cette idée en se penchant sur la *méta-programmation*.

3.2 The Uml model

Remarque 3.1. La suite du chapitre est extraite d'une première version [Ducournau et al., 2007] d'un article en anglais [Ducournau and Privat, 2011].

3.2.1 Introduction

In this section, we present a metamodel for classes and properties in object-oriented languages. Here, we only consider properties which are described in a class but dedicated to its instances, and which depend on their dynamic types. Class properties, i.e. properties which only concern the class itself, not its instances, are excluded from the scope of the metamodel. Hence, we do not consider either *static* methods and variables, the fact that a class may be *abstract* (aka *deferred*), or properties which would be called *non-virtual* in C++ jargon—we only consider properties which are concerned by *late binding*, hence tagged by the `virtual` keyword in C++ when considering methods [2]. In Clos, however, we would also consider slots declared with `:allocation :class`.

This metamodel is intended to be both intuitive and universal. It is likely very close to the intuition of most programmers, when they think of object-oriented concepts. It is universal in the sense that it is not dedicated to a specific language and it is very close to the specifications of most statically typed object-oriented languages, at least when they are used in a simple way. It can be considered as an implicit metamodel of Java and Eiffel—in the latter case, with a limited use of renaming—but it has never been explicitly described in any programming language nor even in Uml [OMG, 2004].

In the following, we successively present an Uml model which provides an informal idea of the metamodel, then a more formal set-theoretical definition. The section ends by considering the resulting run-time behaviour. The analysis of multiple inheritance conflicts, and the comparison with existing languages will be tackled in Section 4.

3.2.2 Object-oriented meta-modeling of object-oriented languages

In an object-oriented setting, metamodeling is a powerful tool which supports intuition and serves as an operational semantics when some meta-object protocol is added. A nice example is the analysis of classes, metaclasses and instantiation made in the ObjVlisp model [3] [Cointe, 1987]. Meta-modeling some part of an object-oriented programming language amounts to defining an object model—i.e. entities like classes, associations, attributes, methods, etc.—for modeling the considered concepts of the language [4]. To this basic specification, we add the following meta-axiome for unambiguity—actually, a metamodel should always be a specification of the meaning of names in a program.

2. 'Virtual' comes from Simula and has several usages—virtual functions (Simula, C++), virtual types (Simula, Beta [Madsen et al., 1993]) and virtual multiple inheritance (C++). It can be understood as 'redefinable', in the sense of 'redefinition' in the present paper—hence, submitted to late binding and depending on the dynamic type of some 'receiver'. However, 'virtual' is also used in the sense of 'abstract', i.e. for a non-implemented method or a non-instantiable class. Recently, OCaml has substituted 'abstract' to 'virtual'— see for instance different versions of [Hickey, 2006]. Though both meanings are related—'abstract' implies 'virtual'—they are different enough to require different terms and we shall only consider the former usage.

3. We take ObjVlisp as an example because it focuses on its object—namely, classes and metaclasses, in their purest form. Actually, the reflective kernel of ObjVlisp is also present in Clos and in other works on reflection, e.g. [Forman and Danforth, 1999]. In contrast, it is not compatible with the Smalltalk reflective kernel [Goldberg and Robson, 1983].

4. Here, we adopt a very restrictive meaning of the term 'metamodel', which is much more general—we only consider *reflective object-oriented* metamodels.

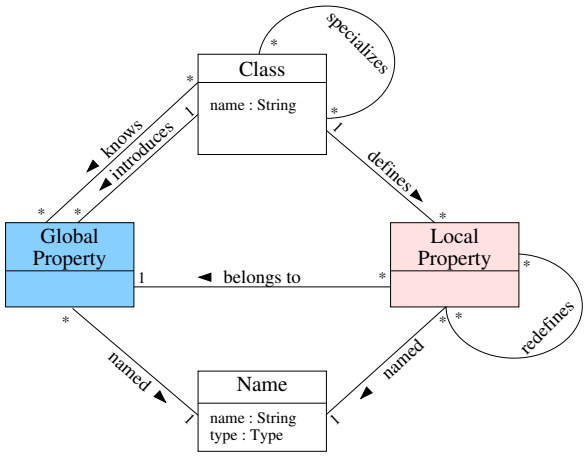


Figure 3.1 – Metamodel of Classes and Properties

Méta-axiome 3.1 (Context-dependent unambiguity) *In the modeled program, any occurrence of an identifier denoting a modeled entity must unambiguously map to a single instance of the meta-model. Of course, this mapping is context-dependent.*

Accordingly, metamodeling allows to get rid of names when considering the modeled entities. This will prove to be of great value, as most difficulties yielded by object-oriented programming lie in the interpretation of names.

Conversely, following Occam's Razor, meta-modeling should aim at minimality. As a counter-example, the Clos reflective kernel includes, but is far larger than, the ObjVlisp model. It may be necessary for fully implementing Clos, but it is useless for conceptually modeling classes, superclasses and instantiation. Models must remain partial.

3.2.3 Classes and properties

The metamodel consists of three main kinds of entities : *classes*, *local properties* and *global properties*—Fig. 3.1. The former is of course natural, but the two latter are key features of the model. When one wants to metamodel properties, it follows from Axiome 3.1 that late binding (aka message sending) implies the definition of exactly two categories of entities. *Local properties* correspond to the attributes and methods as they are defined in a *class*, independently of other possible definitions of the 'same property' in superclasses or subclasses. *Global properties* are intended to model this 'same property' idea in different related classes. They correspond to messages that the instances of a class can answer—in the case of methods, the answer is the invocation of the corresponding local property of the dynamic type of the receiver. Each local property *belongs to* a single global property and is *defined* in a single class.

Global and local properties should in turn be specialized into several specific kinds, according to the values which are associated with the properties—data for attributes, functions for methods or even types for *virtual types*. Attributes and methods are present in all languages but the main complication involves the methods—indeed, attributes are usually quite simpler, though languages such as Clos or Eiffel accept full attribute redefinition. Moreover, a proper distinction between attributes and methods may be not straightforward since it is sound to accept, as in Eiffel, that a method without parameters can be redefined by an attribute. However, there is no need to detail this here. Besides attributes and methods, a third kind of properties must also be considered, namely *virtual types*. Whereas attributes and methods are respectively associated with data and functions, virtual types involve associating a property with a type which depends on the dynamic type of the receiver. Virtual types [Torgersen, 1998, Thorup and Torgersen, 1999] represent a combined genericity and covariance mechanism first introduced in Beta [Madsen et al., 1993] and somewhat similar to Eiffel *anchored types* [Meyer, 1997]. Anyway, in the following, 'property' stands for all three kinds and a partial but more intuitive translation of our terminology is possible—global (resp. local) properties stand for methods (resp. method implementations). Furthermore, in this section, the specific kind of property does not matter.

A class definition is a triplet consisting of the name of the class, the name of its superclasses, presumably already defined, and a set of local property definitions. The specialization relation supports an inheritance mechanism—i.e. classes inherit the properties of their superclasses. When translated in terms of the metamodel, this yields two-level inheritance. First of all, the new class *inherits* from its

```
   class A {
1     public void foo()
      {...}
   }

   class B extends A {
2     public void foo()
      {...}
3     public void bar()
      {...}
   }

   ...

   {...
    A x
    B y
4   x.foo()
5   y.bar()
    ...}
```

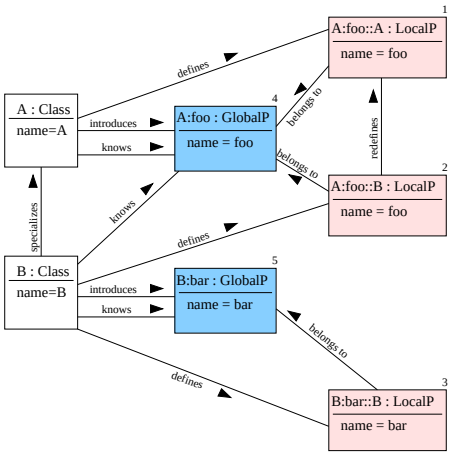


FIGURE 3.2 – A simple JAVA example and the corresponding instance diagram

superclasses all their global properties—this is *global property inheritance*. The subclass *knows* all the global properties *known* by the superclasses. Then each local property definition is processed. If the name[5] of the local property is the same as that of an inherited global property, the new local property is attached to the global property. If there is no such inherited global property, a new global property with the same name is *introduced* in the class. We do not consider here the questionable possibility of introducing a new global property with the same name as an inherited one, as with the `reintroduce` keyword in TURBO PASCAL, or `new` in C♯.

Local property inheritance takes place at run-time—though most implementations statically precompile it. A call site `x.foo(args)` represents the invocation of the *global property* named `foo` of the *static type* (say A) of the receiver `x`. The static typing axiome appears here—in a dynamic typing framework, for instance in SMALLTALK, there is no way to distinguish different global properties with the same name while satisfying Axiome 3.1. Static overloading can also slightly complicate the point. If the name '`foo`' is overloaded in A, i.e. if A knows several global properties named `foo`, which differ by their parameter types or number, a **single** one, i.e. the most specific according to formal and actual parameter types, must be selected at compile-time. A conflict may occur when the most specific property is not unique, e.g. when there is multiple inheritance between parameter types, or with multiple contravariant parameters. Such conflicts can be easily solved by simply making actual parameter types more precise, i.e. paradoxically more general, at the call site. Anyway, at run-time, this call site is interpreted as the invocation of the **single** *local property* corresponding to both the **single** statically selected *global property*, and the *dynamic type* of the value bound to `x`, i.e. the class which has instantiated it. Therefore, when no such local property is defined in the considered class, a local property of the same global property must be inherited from superclasses. Static typing ensures that such a local property exists.

3.2.4 Example

The JAVA example in Figure 3.2 defines seven entities of our metamodel—two classes, `A` and `B` ; three local properties, the method `foo` defined in `A` and the methods `foo` and `bar` defined in `B` ; two global properties, respectively introduced as `foo` in `A` and as `bar` in `B`. The corresponding numbered instance diagram specifies the unambiguous mapping between the code and the instances of the metamodel (Req. 3.1).

The instantiation of the metamodel proceeds as follows, as the code is read :

— `A` class is first created ; it does not inherit any explicit global property—in practice, it would inherit all global properties introduced in the hierarchy root, `Object` ;

— a method named `foo` is defined in `A` : the corresponding local property (1) is created and, since `A` does not know any global property with this name, a global property (4) `foo` is introduced ;

— `B` class is then created as a specialization of `A` ; it then inherits all explicit global properties from `A`, especially the global property `foo` ;

— a method named `foo` is defined in `B` : the corresponding local property (2) is created and attached to the global property (4) `foo` inherited from `A`—this is a redefinition ;

5. The 'name' of properties is a convenient shorthand for more complex identifiers. Static overloading of C++, JAVA, C♯, etc. forces, in these languages, to include the parameter types in the identifier in order to unambiguously denote properties. This implies distinguishing *short names* and *long names*. In the present framework, static overloading is formally defined by the fact that a class knows two global properties with the same short name but different long names.

— a method named `bar` is defined : the corresponding local property (3) is created and, since `B` does not know any global property named `bar`, then a global property (5) `bar` is introduced ;
— in the following code sequence, `foo` (resp. `bar`) is understood as the single global property named `foo` (resp. `bar`) which is known by the static type `A` (resp. `B`) of the receiver `x` (resp. `y`) ; changing the static type of `x` from `A` to `B` would not change the mapping but doing the converse for `y` would yield a static type error—`A` class does not know any global property named `bar`.
— finally, at run-time, the invocation of `foo` will call the local property defined in `A` or in `B`, according to the actual dynamic type of the value of `x`—this is late binding, as usual.

Besides the formalization of basic object-oriented concepts which follows in the next sections, the expected advantage of such a metamodel is that all programming tools could get rid of names and their associated ambiguities and instead just consider reified entities. Actually, these reified entities represent the 'reality'—the *ontology*—of object-oriented programs and names should remain at the human-computer interface.

3.3 Notations and Formal Definitions

This section first defines a model in a static way—i.e. its components and their relationships—then describes the protocols (i) for instantiating it, i.e. the incremental process of class definition, and (ii) for late binding.

Notations

Let E, F and G be sets. 2^E denotes the power set of E, $|E|$ is the size of E, and $E \uplus F$ is the union of the disjoint sets E and F. Given a function $foo : E \to F$, the function $foo^{-1} : F \to 2^E$ maps $x \in F$ to the set $\{y \mid foo(x) = y\}$. Function notations are extended to powersets and cartesian products in the usual way : $\forall G \subseteq E$, $foo(G) = \{foo(x) \mid x \in G\}$ and $\forall R \subseteq E \times E$, $foo(R) = \{(foo(x), foo(y)) \mid (x, y) \in R\}$. Finally, $(E, \leq)$ denotes the graph of a binary relation $\leq$ on a set E. It is a *poset* (partially ordered set) iff $\leq$ is reflexive, transitive and antisymmetric.

Définition 3.1 (Class hierarchy) *A* model of a hierarchy, *i.e. an instance of the metamodel, is a tuple* $\mathcal{H} = \langle X^{\mathcal{H}}, \prec^{\mathcal{H}}, G^{\mathcal{H}}, L^{\mathcal{H}}, N^{\mathcal{H}}, name_{\mathcal{H}}, glob_{\mathcal{H}}, intro_{\mathcal{H}}, def_{\mathcal{H}} \rangle$, *where :*

— $X^{\mathcal{H}}$ *is the set of* classes ;
— $\prec^{\mathcal{H}}$ *is the* class specialization *relationship, which is transitive, antisymmetric and antireflexive ;* $\preceq^{\mathcal{H}}$ *(resp.* $\prec^{\mathcal{H}}_d$*) denotes the reflexive closure (resp. transitive reduction) of* $\prec^{\mathcal{H}}$ *and* $(X^{\mathcal{H}}, \preceq^{\mathcal{H}})$ *is a poset ;*
— $G^{\mathcal{H}}$ *and* $L^{\mathcal{H}}$ *are the disjoint sets of* global *and* local properties ;
— $N^{\mathcal{H}}$ *is the set of* identifiers *(names) of* classes *and* properties ;
— $name_{\mathcal{H}} : X^{\mathcal{H}} \uplus G^{\mathcal{H}} \uplus L^{\mathcal{H}} \to N^{\mathcal{H}}$ *is the naming function of classes and properties ; its restriction to* $X^{\mathcal{H}}$ *is injective ;*
— $glob_{\mathcal{H}} : L^{\mathcal{H}} \to G^{\mathcal{H}}$ *associates with each local property a global property ;*
— $intro_{\mathcal{H}} : G^{\mathcal{H}} \to X^{\mathcal{H}}$ *associates with a global property the class introducing it ;*
— $def_{\mathcal{H}} : L^{\mathcal{H}} \to X^{\mathcal{H}}$ *associates with a local property the class where it is defined.*

The notations are supplemented by the following set of equations and definitions (3.2–3.10).

The sets $X^{\mathcal{H}}$, $G^{\mathcal{H}}$ and $L^{\mathcal{H}}$ correspond to the three classes in the metamodel, whereas the total functions $glob_{\mathcal{H}}$, $intro_{\mathcal{H}}$ and $def_{\mathcal{H}}$ correspond to the three functional associations—all six elements form the metamodel triangle. The 'specializes' association is represented by $\prec^{\mathcal{H}}$ and all other associations, such as 'knows' and 'redefines', are not primitive. On the other hand, $N^{\mathcal{H}}$ and $name_{\mathcal{H}}$ represent relationships between the metamodel and names which are used in the program text. So far, the formalization is a straightforward translation of the Uml diagram in Figure 3.1. The definition of a *legal model* is achieved by a set of constraints which ensure that : (i) the triangular diagram commutes at the instance level, (ii) names in the program text are unambiguous or can be disambiguated.

The metamodel is generic, as all its components are parametrized by $\mathcal{H}$. However, in the rest of this chapter, parameter $\mathcal{H}$ will remain implicit for the sake of readability. All proofs are trivial and left to the reader.

3.3.1 Global Properties

Given a class $c \in X$, G_c denotes the set of global properties of c. Global properties are either *inherited* from a superclass of c, or *introduced* by c. Let $G_{\uparrow c}$ and G_{+c} be the two corresponding subsets. Hence, all G_{+c} are disjoint and

$$G_{+c} \stackrel{\text{def}}{=} intro^{-1}(c) \ , \tag{3.2}$$

$$G_{\uparrow c} \stackrel{\text{def}}{=} \bigcup_{c \prec_{\text{d}} c'} G_{c'} = \biguplus_{c \prec c'} G_{+c'} \ , \tag{3.3}$$

$$G_c \stackrel{\text{def}}{=} G_{\uparrow c} \uplus G_{+c} = \biguplus_{c \preceq c'} G_{+c'} \ , \tag{3.4}$$

$$G = \bigcup_{c \in C} G_c = \biguplus_{c \in C} G_{+c} \ . \tag{3.5}$$

The definitions and equations (3.2-3.4) formally define *global property inheritance.*

Names of newly introduced global properties are constrained :

Contrainte 3.1 *When a global property is introduced, its name must be unambiguous, hence (i) the restriction of name on G_{+c} is injective and (ii) inherited and introduced properties cannot have the same name :*

$$name(G_{+c}) \cap name(G_{\uparrow c}) = \emptyset \ .$$

Note that this constraint is actually implied by the constraints concerning local properties (Section 3.3.2). It follows that the function $gid : G \to X \times N$ that maps a global property $g \in G$ to the pair

$$gid(g) \stackrel{\text{def}}{=} (intro(g), name(g)) \tag{3.6}$$

is injective. Besides this constraint, all $\langle X, \prec, G, name, intro \rangle$ tuples that follow Definition 3.1 are legal.

There is a global property conflict when a class knows two distinct global properties with the same name. Constraint 3.1 implies that such a conflict is always caused by multiple inheritance :

Définition 3.2 (Global property conflict) *Given a class $c \in X$ and two distinct global properties $g_1, g_2 \in G_{\uparrow c}$, a* global property conflict *occurs between g_1 and g_2 when*

$$name(g_1) = name(g_2) \wedge intro(g_1) \neq intro(g_2) \ .$$

Moreover, this implies that there are classes $c', c_1, c_2 \in X$, such that :

$$(c \preceq c' \prec_{\text{d}} c_1, c_2) \wedge (g_1 \in G_{c_1} \backslash G_{c_2}) \wedge (g_2 \in G_{c_2} \backslash G_{c_1}) \ .$$

When there is no global property conflict, for instance in a legal model in single inheritance, the restriction of the function $name : G \to N$ to $G_{\uparrow c}$ is injective. Moreover, in the same condition, Constraint 3.1 implies that the restriction of *name* to G_c is also injective. Therefore, in the context of a class c, the identifier of a global property is unambiguous. Of course, *name* is not constrained to be injective throughout its domain G—hence it must be disambiguated by the context, i.e. the static type of the receiver. Multiple inheritance conflicts will be examined in Chapter 4.

3.3.2 Local Properties

Given a class c, L_c denotes the set of local properties defined in c and, conversely, the function $def : L \to C$ associates with each local property the class where it is defined :

$$L = \biguplus_{c \in C} L_c \qquad \text{with} \qquad L_c \stackrel{\text{def}}{=} def^{-1}(c) \ . \tag{3.7}$$

The correspondence between local and global properties must be constrained in order to close the metamodel triangle, i.e. to make the diagram commute. First, the correspondence is based on property names :

Contrainte 3.2 *The function $glob : L \to G$ associates with each local property a global property, such that both have the same name :*

$$\forall l \in L, name(glob(l)) = name(l) \ .$$

Moreover, it does not make sense to define more than one local property for some global property in the same class, hence :

Contrainte 3.3 *For all $c \in X$, the restriction of glob to L_c is* injective. *Equivalently, for all $g \in G$, the restriction of def to $glob^{-1}(g)$ is injective.*

Therefore, if there is no global property conflict, the restriction of *name* to L_c is also injective. Thus determining the global property associated with a local one is unambiguous in the context of some class c, at least when the name of the global property is unambiguous, i.e. when there is no global property conflict :

$$\forall l \in L_c, \forall g \in G_c, name(l) = name(g) \Longrightarrow glob(l) = g \ . \tag{3.8}$$

Of course, *glob* and *name* are not injective over their whole domain L—hence local property names must be disambiguated by the context, i.e. the enclosing class, at compile-time, and local properties must be selected by the late binding mechanism at run-time (see below).

A last constraint closes the triangle and achieves the definition of a *legal model* :

Contrainte 3.4 *The associated global properties must be known by the considered class and all global properties must have been introduced by a local property definition :*

$$\forall c \in X, G_{+c} \subseteq glob(L_c) \subseteq G_c \ .$$

If a property is considered as *abstract* (aka *deferred*) in its introduction class—i.e. if it has no default implementation—an abstract local property must still be provided.

Définition 3.3 (Legal model) *A* legal model *of a* class hierarchy *is a model which satisfies all Constraints 3.1 to 3.4.*

It follows from Constraint 3.3 that, in a legal model, the function $lid : L \to G \times X$ that maps a local property $l \in L$ to the pair

$$lid(l) \stackrel{\text{def}}{=} (glob(l), def(l)) \tag{3.9}$$

is injective.

Finally, one can formally define *property redefinition*. A local property corresponds to an inherited global property, or to the introduction of a new global property. Let $L_{\uparrow c}$ and L_{+c} be the corresponding sets.

$$L_c = L_{\uparrow c} \uplus L_{+c} \quad \text{with} \begin{cases} L_{\uparrow c} & \stackrel{\text{def}}{=} L_c \cap glob^{-1}(G_{\uparrow c}) \ , \\ L_{+c} & \stackrel{\text{def}}{=} L_c \cap glob^{-1}(G_{+c}) \ . \end{cases} \tag{3.10}$$

Moreover, *glob* is a one-to-one correspondence between L_{+c} and G_{+c}.

Définition 3.4 (Property redefinition) Property redefinition *(aka* overriding*) is defined as the relationship $\ll^{\mathcal{H}}$ (or $\ll$ for short) between a local property in $L_{\uparrow c}$ and the corresponding local properties in the superclasses of c :*

$$l \ll l' \stackrel{\text{def}}{\Longleftrightarrow} glob(l) = glob(l') \wedge def(l) \prec def(l') \ .$$

This is a strict partial order and $\ll_d$ denotes its transitive reduction.

3.3.3 Class Definition and Model Construction

The model is built by successive class definitions.

Définition 3.5 (Class definition) *A* class definition *is a triplet* ⟨`classname`, `supernames`, `localdef`⟩, *where* `classname` *is the name of the newly defined class,* `supernames` *is the set of names of its direct superclasses—they are presumed to be already defined—and* `localdef` *is a set of local property definitions.*

A local property definition involves a property name—in the general meaning, i.e. including parameter types. if static overloading is considered—and other data, e.g. code, which are not needed here.

Let $\mathcal{H}$ be a legal class hierarchy, then a class definition in $\mathcal{H}$ will produce another hierarchy $\mathcal{H}'$. The operational semantics of the metamodel is given by the *meta-object protocol* which determines how this class definition is processed. We informally sketch this 4-step protocol as follows.

(i) **Hierarchy update** A *new* class c with name `classname` is added to X—i.e. $X' = X \uplus \{c\}$. For each name $n \in$ `supernames`, a pair $(c, name^{-1}(n))$ is added to $\prec_d$. Of course, the names of all considered classes are checked for correction—existence and uniqueness. Moreover, `supernames` is a set—this means that is does not make sense to inherit more than once from a given class—and it should also be checked against transitivity edges, that should not be added to $\prec_d$ [6].

6. In both cases, this is contrary to C++ and Eiffel behavior.

(ii) **Global property inheritance** The protocol then proceeds to global property inheritance. $G_{\uparrow c}$ is computed (3.3) and global property conflicts are checked (Def. 3.2).

(iii) **Local definitions** For each definition in `localdef`, a *new* local property is created, with its corresponding name—this yields L_c. $L_{\uparrow c}$ is determined by (3.10). Then, G_{+c} is constituted as the set of *new* global properties corresponding to each local property in $L_{+c} = L_c \backslash L_{\uparrow c}$. L_c and G_{+c} are then respectively added to L and G—i.e. $L' = L \uplus L_c$ and $G' = G \uplus G_{+c}$. Here again, the names of all local properties are checked for correction—existence and uniqueness. Ambiguities resulting from global property conflicts are discussed in Section 4.2.1.

(iv) **Local property inheritance** Finally, the protocol proceeds to local property inheritance and checks conflicts for all inherited and not redefined properties, i.e. $G_{\uparrow c} \backslash glob(L_{\uparrow c})$. Conflicts are discussed in Section 4.2.2.

In the protocol, each occurrence of 'new' denotes the instantiation of a class in the metamodel.

The metamodel is complete, in the sense that all components in Definition 3.1, together with Constraints 3.1 to 3.4 and equations (3.2–3.10), are sufficient to characterize a legal model as long as there is no global property conflict. All such legal models could be generated by Definition 3.5—given a legal class hierarchy $\mathcal{H}$, for all c in X, $\langle name(c), \{name(c') \mid c \prec_d c'\}, name(L_c)\rangle$ form a legal class definition.

3.4 Local property inheritance and method invocation

So far, we have presented the static model, which considers classes and properties at compile-time. These classes behave as usual at run-time, i.e. they have instances which receive and send messages according to the SMALLTALK metaphor. Run-time objects and their construction are not explicit in the model—for instance, we have not merged the previous metamodel within the OBJVLISP kernel. This is actually not required since message sending can be modelled at the class level—it only depends on the *receiver's dynamic type*, which is some class. The precise receiver does not matter and all direct instances of a given class are equivalent—this explains why method invocation can be compiled and efficiently implemented. Constructor methods are no more explicit in the metamodel because there is no room for them—the construction itself is a hidden black box and the so-called constructors are ordinary initialization methods which do not require any specific treatment [7].

Local property inheritance is a matter of selection of a single local property in a given global property, according to the dynamic type of an object called the receiver. Though it applies only to methods in most languages, it can also be used for attributes—and it is indeed used in some languages like CLOS, YAFOOL or EIFFEL. We only detail the case concerning methods. Method invocation usually involves two distinct mechanisms, namely late binding (aka message sending) and call to `super`, and implies some auxiliary functions—$loc, spec : G \times X \to 2^L$, $sel : G \times X \to L$, $cs : G \times X \to 2^X$ and $supl : L \to 2^L$—defined as follows. All functions *loc*, *spec*, *sel* and *cs* are partial and only defined on $g \in G$ and $c \in X$ when $g \in G_c$.

3.4.1 Late binding

Given a class $c \in C$—the dynamic type of the receiver—and a global property $g \in G_c$, *late binding* involves the selection of a local property of g defined in the superclasses of c (including c), hence in the set

$$loc(g, c) \stackrel{\text{def}}{=} \{l \in glob^{-1}(g) \mid c \preceq def(l)\} \ . \tag{3.11}$$

Thus, the selection function *sel* must return a local property such that $sel(g, c) \in loc(g, c)$. If c has a local definition for g—i.e. if $L_c \cap loc(g, c) = \{l\}$—then $sel(g, c) = l$. On the contrary, c must inherit one from its superclasses—this constitutes the second level of inheritance, namely *local property inheritance*. In single inheritance, *sel* returns the *most specific* property, i.e. the single element in

$$spec(g, c) \stackrel{\text{def}}{=} \min_{\ll}(loc(g, c)) \ , \tag{3.12}$$

but its uniqueness is only ensured in single inheritance. In multiple inheritance, a *local property conflict* may occur :

7. 'Constructor' is another misleading term. As a local property, it just denotes an *initializer*, which corresponds to CLOS `initialize-instance`. At a constructor call site, i.e. as a global property, the construction role is ensured by some hidden mechanism—similar to CLOS `make-instance`—which makes the instance before calling the initializer. Anyway, the programmer must usually declare that a given method can be used as a constructor, i.e. as a *primary initializer* directly called at an instantiation site. Normally, any method can be used as a secondary initializer, i.e. when indirectly called by a primary initializer, though an odd specification makes it false in C++ since, in this language, late binding is impossible within a constructor.

Définition 3.6 (Local property conflict) *Given a class $c \in X$ and a global property $g \in G_{\uparrow c}$, a* local property conflict *occurs when $|spec(g,c)| > 1$. The* conflict set *is defined as the set*

$$cs(g,c) \stackrel{\text{def}}{=} def^{-1}(spec(g,c)) = \min_{\preceq}(def^{-1}(loc(g,c)))$$

which contains all superclasses of c which define a local property for g and are minimal according to $\preceq$.

We shall examine, in Section 4.2.2, the case of multiple inheritance, local property conflicts and the reasons for this definition.

3.4.2 Call to `super`

Call to `super` allows a local property l to call another one, say l', which is redefined by l, i.e. $l \ll l'$. This can be ensured by another selection function which selects an element in the set

$$supl(l) \stackrel{\text{def}}{=} \{l'' \mid l \ll l''\} \ . \tag{3.13}$$

Note that we keep the term '`super`' used in Smalltalk and Java as it is the most popular, but we take it with a slightly different meaning. It is here closer to Eiffel `Precursor`. The point is that l and l' belong to the same global property, i.e. $glob(l) = glob(l')$, while Smalltalk and Java accept `super.foo` in method `bar`. Usually, like late binding, call to `super` involves selection of the most specific property, i.e. the single element in $\min_{\ll}(supl(l))$, which is the single local property l' which satisfies $l \ll_d l'$. However, the uniqueness of l' is only ensured in single inheritance, where it is alternatively determined by the uniqueness of c' such that $def(l) \prec_d c'$—then $l' = sel(glob(l), c')$.

The bottom-up call to `super` is not the only way of combining methods. A top-down mechanism has also been proposed in Lisp-based object-oriented languages (Flavors, Clos) under the name of `:around` methods (aka *wrappers*) and in Beta under the name of `inner`. As these mechanisms are exactly symmetric to `super`, they present the same problems in case of multiple inheritance. We shall examine them in Section 4.2.2.

3.5 Étendre et implémenter le méta-modèle.

Le méta-modèle présenté dans ce chapitre reste abstrait car il ne modélise pas de nombreux éléments essentiels des langages de programmation par objets :

- les différentes sortes de propriétés, attributs, méthodes, voire types virtuels, avec cette difficulté qu'il n'y a pas forcément de frontière stricte entre les deux premières : ainsi il est licite de permettre, comme en Eiffel, de redéfinir une méthode sans paramètre comme un attribut ;
- la notion de type et ses rapports aux classes, en particulier dans le cadre de la généricité (Chapitre 7), où les relations entre les classes paramétrées et leurs instances génériques doivent être modélisées.

Dans tous les cas, le méta-modèle implémenté par un véritable compilateur ou ADL doit être autrement plus compliqué.

Exercice 3.1. Étendre le méta-modèle pour tenir compte de la différence entre méthodes et attributs. □

Exercice 3.2. Comment peut-on prendre en compte le fait qu'un attribut peut redéfinir une méthode sans paramètre, comme en Eiffel ? Modéliser en particulier la relation entre un attribut et ses accesseurs. □

Exercice 3.3. Comment intégrer dans le méta-modèle la notion de *surcharge statique* (Section 6.3) ? Peut-on faire apparaître explicitement une association entre méthodes de la même classe en relation de surcharge statique ? □

Exercice 3.4. Comment intégrer dans le méta-modèle les classes paramétrées et leurs instances génériques ? □

Exercice 3.5. Implémenter le méta-modèle (en Java, C++, C♯, Eiffel, Clos, etc.) de façon complète, en implémentant tout le protocole de définition d'une nouvelle classe (Définition 3.5, page 33). □

3.6 En savoir plus

Le principe de la méta-modélisation est à la base d'UML et des langages réflexifs comme SMALLTALK et CLOS (Chapitre 11). Malgré cela, aucun de ces méta-modèles ne permet l'équivalent de ce qui est présenté ici et qui jouera pleinement son rôle dans l'analyse de l'héritage multiple (Chapitre 4).

Dans les chapitres suivants, nous allons aborder différentes notions comme le typage ou la visibilité qui ne font pas partie intégrante du méta-modèle mais qui pourraient y être rajoutées sans problème, c'est-à-dire sans avoir à remettre en cause les spécifications précédentes.

Bibliographie

P. Cointe. Metaclasses are first class : the ObjVlisp model. In *Proc. OOPSLA'87*, SIGPLAN Not. 22(12), pages 156–167. ACM, 1987.

R. Ducournau and J. Privat. Metamodeling semantics of multiple inheritance. *Science of Computer Programming*, 76(7) :555–586, 2011. doi : 10.1016/j.scico.2010.10.006.

R. Ducournau, J. Euzenat, G. Masini, and A. Napoli, editors. *Langages et modèles à objets : État des recherches et perspectives*. Collection Didactique. INRIA, 1998.

R. Ducournau, F. Morandat, and J. Privat. Modules and class refinement : a metamodeling approach to object-oriented languages. Technical Report LIRMM-07021, Université Montpellier 2, 2007.

I. R. Forman and S. H. Danforth. *Putting Metaclasses to Work*. Addison-Wesley, 1999.

A. Goldberg and D. Robson. *Smalltalk : the language and its implementation*. Addison-Wesley, 1983.

Jason Hickey. Introduction to the Objective Caml programming language, 2006. URL http://www.cs.caltech.edu/courses/cs134/cs134b/book.pdf.

S.C. Kleene. *Logique mathématique*. Collection U. Armand Colin, Paris, 1971.

O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

OMG. Unified Modeling Language 2.0 superstructure specification. Technical report, Object Management Group, 2004.

W.V. Quine. *Relativité de l'ontologie et autres essais*. Aubier Montaigne, Paris, 1977.

K.K. Thorup and M. Torgersen. Unifying genericity : Combining the benefits of virtual types and parameterized classes. In R. Guerraoui, editor, *Proc. ECOOP'99*, LNCS 1628, pages 186–204. Springer, 1999.

Mads Torgersen. Virtual types are statically safe. In *Elec. Proc. of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL 5)*, 1998.

4

Héritage multiple

Mais souvent tel objet, qui par une ou plusieurs de ses propriétés
a été placé dans une classe, tient à une autre classe par d'autres propriétés [. ..].
d'Alembert, *Discours préliminaire* à l'Encyclopédie, cité par [Eco, 1994].

Si l'on peut faire remonter la programmation par objets à Aristote, la citation en exergue de ce chapitre montre que la prise de conscience du besoin d'héritage multiple remonte au moins au Siècle des Lumières. De fait, depuis les origines de la programmation par objets, l'héritage multiple a été considéré, simultanément, comme une fonctionnalité souhaitable et comme un problème à la fois conceptuel et technique. Nous en proposons une analyse qui a le mérite de bien distinguer les vrais problèmes des faux.

Ce chapitre présente d'abord le problème de façon intuitive, en expose la modélisation avec l'aide du méta-modèle, présente l'état de l'art des langages existants pour finir par étudier en détail les techniques de linéarisation qui représentent le fin du fin en matière de combinaison de méthodes. On termine en considérant une variante basée sur la notion de *mixin*.

4.1 Le problème

Le degré zéro de l'héritage s'exprime par l'inclusion des intensions de la proposition Aristotélicienne 2.2, page 21 : la sous-classe hérite des propriétés de ses super-classes. Cette inclusion apparemment sans problèmes 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 conflictuel de la même propriété ou de propriétés de mêmes noms.

Le méta-modèle du chapitre 3 est un instrument décisif pour analyser ces problèmes, qui se réduisent aux deux interrogations suivantes :

— une classe peut avoir deux propriétés globales distinctes qui portent le même nom, vraisemblablement parce que cette classe hérite de deux super-classes développées par des programmeurs différents : comment distinguer les deux dans le contexte de la sous-classe ?
— pour une unique propriété globale, une classe peut hériter de plusieurs propriétés locales par des chemins, ou depuis des super-classes, distincts : comment traiter un envoi de message sur une instance de cette classe ?

4.1.1 Le motif du losange

Ces deux interrogations se ramènent à 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 :

$$D \prec B \prec A \qquad \text{et} \qquad D \prec C \prec A \tag{4.1}$$

Sur ces 4 classes, on peut définir des 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.

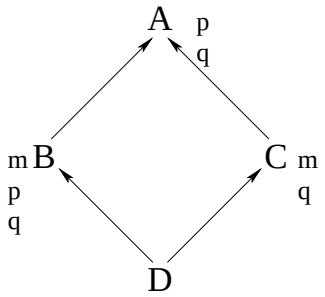


FIGURE 4.1 – Héritage multiple : le schéma du losange exhibe une triple situation, à résoudre différemment suivant qu'il s'agit de m, p ou q.

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`

et pour les propriétés :

$m =$ `Département`
$p =$ `RégimeSécu`

EXERCICE 4.1. L'analyse de l'exemple dans les termes du méta-modèle est laissée au lecteur : quelles propriétés globales et locales se déduisent de ces définitions ? Faire le diagramme d'instances du méta-modèle pour chacune des 3 propriétés m, p et q (Ne pas essayer de faire un shéma complet unique qui serait trop gros). □

4.2 Méta-modèle pour l'héritage multiple

Remarque 4.1. Cette section poursuit la formalisation entreprise au chapitre 3 : elle est extraite du même article en anglais [Ducournau et al., 2007].

In multiple inheritance, conflicts are the main difficulty. They are usually expressed in terms of name ambiguities. The metamodel yields a straightforward analysis in terms of reified entities and distinguishes between two kinds of conflicts which require totally different answers. The following analysis is mostly the same as in [Ducournau et al., 1995], just enhanced with the metamodel.

4.2.1 Global Property Conflict

Un *conflit de propriétés globales* (appelé 'conflit de nom' dans [Ducournau et al., 1995]) se produit lorsqu'une classe connaît deux propriétés globales de même nom (Def. 3.2, page 32 et Figure 4.2). Du fait des contraintes portant sur le nommage non-ambigu des propriétés lors de leur introduction, cette situation résulte du fait que la classe considérée spécialise les deux classes qui ont introduits ces deux propriétés globales.

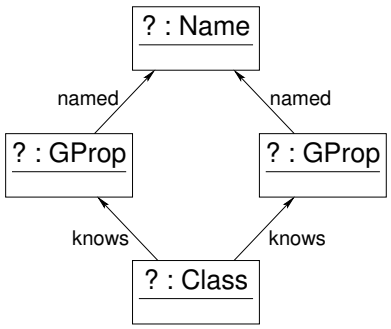
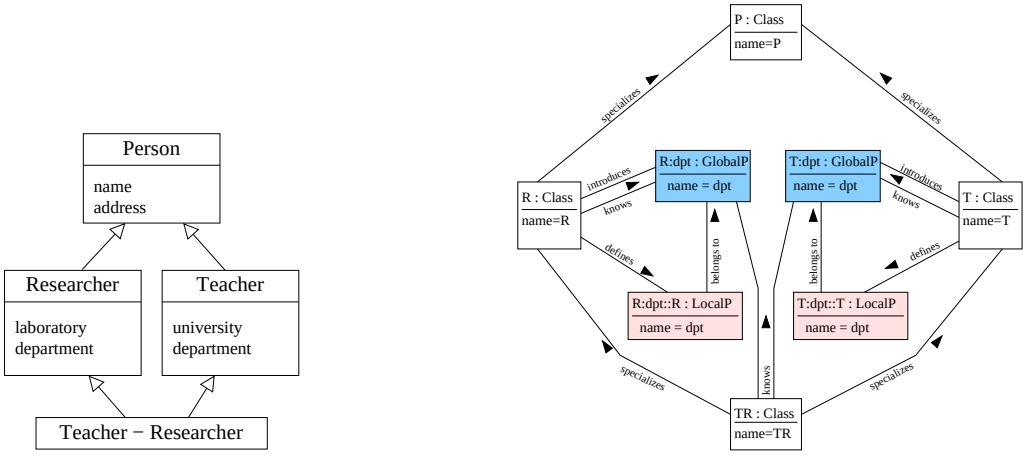


FIGURE 4.2 – Le motif de base du conflit de propriétés globales



(a) Class diagram (b) Instance diagram

FIGURE 4.3 – Global Property Conflict—the class diagram (a) depicts a conflict between the two properties named `department` introduced in two unrelated classes and the instance diagram (b) shows the corresponding metamodel instantiation. All entities are tagged by unambiguous fully qualified names and names are abbreviated.

Example. Figure 4.3 shows two global properties named `department`. The first one specifies a department in a research laboratory. The other one specifies a teaching department in a university. It is then expected that the common subclass `Teacher-Researcher` inherits all the different global properties of its superclasses. However, the name `department` is ambiguous in the subclass context. Anyway, this situation is simply a naming problem. It must be solved and a systematic renaming in the whole program would solve it. Different answers are possible, which do not depend on the specific kind of properties, i.e. attributes, methods or types, since the conflict only involves names :

Nothing, i.e. error The language does not specify any answer to such a conflict but it signals an ambiguity. This forces the programmer to rename at least one of the two properties, but this must be done throughout the program and can be error-prone or even impossible (unavailable source code).

Fully qualified names This simply involves an alternative unambiguous fully qualified syntax, which juxtaposes the property name with the name of a class in which the property name is not ambiguous, for instance the class that introduces the global property. In the example, `Teacher:department` would denote the global property known as `department` in the class `Teacher`. Such a naming would be unambiguous since, in a legal model, *gid* is injective (3.6). A similar solution is used for attributes in C++ with the operator :: [Stroustrup, 1998] [1]. Note that, with fully qualified names, a global property conflict requires a solution only when the programmer wants to use the ambiguous name, in a context where it is ambiguous. Hence, this is a modular and lazy solution.

Local Renaming Local renaming changes the designation of a property, both global and local, in a class and its future subclasses. In the `Teacher-Researcher` class of the example, one can rename `department` inherited from `Teacher` as `dept-teach` and `department` inherited from `Researcher` as `res-dept`. Thus, `department` denotes, in `Researcher`, the same global property as `res-dept` in `Teacher-Researcher`; conversely, in the class `Teacher-Researcher`, `res-dept` and `teach-dept` denote two distinct global properties. This solution is used in EIFFEL [Meyer, 1997]. Renaming here is required even when the programmer does not use the name in a context where it is ambiguous—i.e. when redefining one of the conflicting properties or calling it on a receiver typed by the considered class. Moreover, as class hierarchies are not constrained to form lattices, the same conflicting properties can occur in different subclasses, with possibly different renamings.

Unification, i.e. silence Dynamic languages, like CLOS, and C++ for functions (not for attributes [2]), consider that if two global properties have the same name then they are not distinct. JAVA has the same behaviour when a class implements two interfaces which introduce a global

1. In C++, attributes cannot be redefined—this would be static overloading, hence a new global property. Therefore, a single local attribute corresponds to each global attribute and :: denotes either the local or global one. For methods, :: corresponds to a static call—therefore, it denotes a local property.
2. This dissymmetry between methods and attributes represents one of the most striking flaws of the language specifications.

property with the same name and signature[3]. Hence, the global property conflict is not recognized and the multiple inheritance ambiguities are deferred to local property inheritance. So this solution is very close to the first one—it does not allow the programmer to express his/her intention of distinct global properties, unless there is global renaming. In Figure 4.3, the two departments represent distinct concepts. If the programmer's intention was a single concept, then he/she should have defined a common superclass introducing a single global property for this concept. However, silence adds an extra flaw—i.e. the programmer may remain unaware of the problem or might misunderstand it.

As a provisional conclusion, global property conflicts represent a very superficial problem. They could easily be solved in any programming language at the expense of a cosmetic modification—namely, *qualifying* or *renaming*—and they should not be an obstacle to the use of multiple inheritance. Both solutions need a slight adaptation of the metamodel. This is straightforward in the case of full qualification—all names in the class definition—more generally in the program text—may be simple or fully qualified. An analogous syntax is available in all languages where name-spaces are explicit, e.g. packages in JAVA or COMMON LISP. This is less simple for local renaming—the function $name_{\mathcal{H}}$ is no longer global and must take two parameters, the property and the class. Moreover, renaming clauses must be integrated in Definition 3.5, page 33.

On the contrary, when languages deal with global property conflicts by signaling an error or unifying conflicting properties, the situation may have no solution. A global renaming would be the only way out, but it might be impossible if the source code of the conflicting classes is not available, or if the conflicting classes are contractually used by other people.

En typage dynamique Les conclusions qui précèdent sont valides en typage statique, quant le type statique permet de désambiguïser deux propriétés de même nom introduites par des classes différentes. En revanche, cela ne s'applique pas aux langages à typage dynamique. Dans ces langages, la solution reposera sur

Convention de nom pour les méthodes. Les langages à typage dynamique, CLOS par exemple (voir Chapitre 12), édictent souvent un style de programmation qui impose de nommer les méthodes suivant des conventions particulières : on préfixera par exemple leur nom par tout ou partie du nom d'une classe qui désigne une grande catégorie de classes. Cette convention permet d'éviter la plupart des conflits spontanés.

Encapsulation pour les attributs. Pour les attributs, il n'est pas nécessaire de recourir à une convention de nom : une technique plus simple consiste à les encapsuler, à la SMALLTALK. Seul `self` peut accéder aux attributs, ce qui permet de les traiter comme en typage statique : une syntaxe qualifiée est alors nécessaire.

4.2.2 Local Property Conflict

Conflict definition

Un *conflit de propriétés locales* (appelé 'conflit de valeurs' dans [Ducournau et al., 1995]) se produit lorsqu'une classe hérite deux propriétés locales plus spécifiques de la même propriété globale (Def. 3.6, page 35 et Figure 4.4).

Example. A *local property conflict* (in [Ducournau et al., 1995], it was called 'value conflict') occurs when a class inherits two local properties from the same global property, with none of them more specific than the other according to $\ll$ (Def. 3.6, page 35). Figure 4.5a illustrates this situation with two classes, `Rectangle` and `Rhombus`, redefining the method `area` whose global property was introduced into a common superclass, `Quadrilateral`. In the common subclass `Square`, none is most specific—which one must be selected ? Figure 4.5c depicts the model of the example, restricted to the concerned property. Let g be the `area` global property introduced in `Quadrilateral`—`Q:area` for short—and c be the `Square` class, abbreviated in `S` in the diagram. Then,

$$\begin{aligned} loc(g,c) &= \{\texttt{Q:area::Q}, \texttt{Q:area::Rh}, \texttt{Q:area::Re}\},\\ spec(g,c) &= \{\texttt{Q:area::Rh}, \texttt{Q:area::Re}\},\\ cs(g,c) &= \{\texttt{Rh}, \texttt{Re}\}. \end{aligned}$$

The conflict vanishes if one removes the definition of one of the two conflicting local properties, say `Q:area::Rh`, though `Rhombus` still inherits a local property from `Quadrilateral` (Fig. 4.5b). However, some

3. JAVA provides a fully qualified syntax for classes, not for properties.

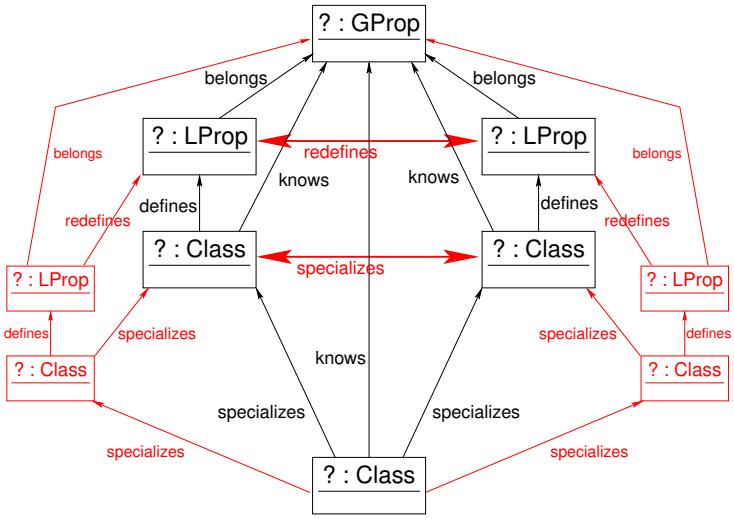


FIGURE 4.4 – Le motif de base du conflit de propriétés locales. Le motif en noir doit exister, de telle sorte que le motif en rouge (grisé en version noir et blanc) n'existe pas : dans le motif noir, il n'y a pas de spécialisation entre les classes, ou de redéfinition entre les propriétés locales, et ces éléments, classes ou propriétés locales, sont minimaux par rapport à la relation considérée, relativement aux classe et propriété globale considérées.

languages, e.g. EIFFEL, consider that there is still a conflict in `Square`, between the property defined in `Rectangle` and the property inherited by `Rhombus`, unless the latter is abstract. Therefore, it is important to understand why we choose this conflict definition. Apart from redefinition, specialization is inherently *monotonic*—i.e. in the definition of A, what is true for an instance of A is also true for an instance of any subclass of A. This goes back to Aristotelian syllogistic—see Chapter 2. On the contrary, property redefinition entails *non-monotonicity*, in the sense of *nonmonotonic* (aka *defeasible*) inheritance theories, e.g. [Touretzky, 1986, Horty, 1994]. A local property is a *default value* for the instances of the class which defines it. For all instances of a subclass A', the redefining property *overrides* (or *masks*) the redefined one. So property redefinition must be understood following the *masking rule* :

Axiome 4.1 (MASKING RULE) *Let g be a global property, l a local property of g defined in class A. Then, l implements g for all instances of A,* unless otherwise stated, *i.e. unless the considered object is an instance of a subclass of A which redefines g. So, if $A' \prec A$ and $l' \ll l$ redefines l in A', l' will* mask *l for all instances of A', direct and indirect alike.*

So in the considered example of Figure 4.5b, `area::Q` implements `area` for all instances of `Q`, except instances of `Re`, and `area::Re` implements it for all instances of `Re`, including those of `Square`. This means that *defining* a property is stronger than *inheriting* it. In this perspective, method combination is a way to recover some monotonicity—if l' calls `super`, all instances of A' will behave like those of A, plus some extra behaviour.

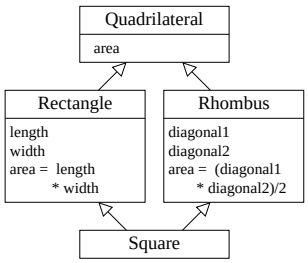
Conflict solutions

Unlike the global property conflict, there is no intrinsic solution to this problem. Consequently, either the programmer or the language must bring additional semantics to solve the local property conflict and this additional semantics may depend on the kind of properties, i.e. attribute, method or type. There are roughly three ways to do it :

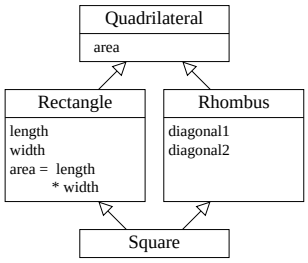
Nothing, i.e. error The considered language does not specify any answer to local property conflicts but it signals an error at compile-time. This forces the programmer to define a local property in the class where the conflict appears. In this redefinition, a call to `super` is often desired but it will be ambiguous (see below).

A variant of this approach makes the class which introduces the conflict (`Square`) *abstract*, by implicitly defining an *abstract* local property[4] (`area`) in this class. This forces the programmer to define the property in all direct non-abstract subclasses.

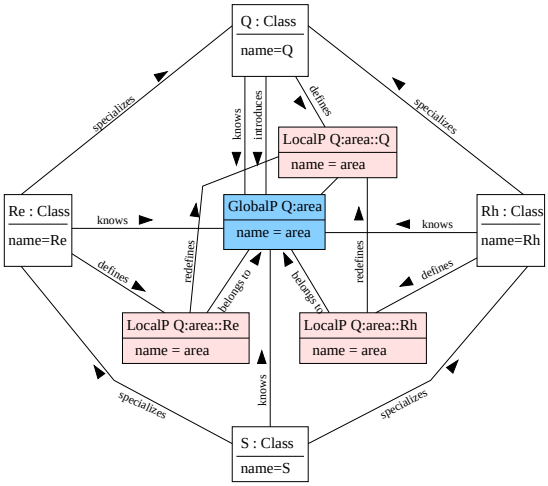
4. When considering *abstract local properties*, conflict definition must be slightly adapted. An actual conflict occurs when $spec(g, c)$ contains several non-abstract properties. If all members of $spec(g, c)$ are abstract, the inherited property in c is also abstract.



(a) Conflict class diagram



(b) No conflict



(c) Conflict instance diagram

FIGURE 4.5 – Local Property Conflict—the class diagram (a) depicts a conflict between the two local properties `area` redefining the same property in two unrelated classes and the instance diagram (c) shows the corresponding metamodel instantiation. In contrast, there is no conflict in class diagram (b).

Selection The programmer or the language arbitrarily select the local property to inherit. In many dynamic languages, the choice is made by a *linearization* (Section 4.2.3) ; in EIFFEL, the programmer can select the desired property with the `undefine` inheritance clause.

Combining For some values or particular properties—especially for some *meta-properties*[5]—the conflict resolution must be done by combining the conflicting values. For instance, in JAVA, when the conflict concerns the declared exceptions of methods, it should be solved by taking the intersection of all declared exception sets[6]. Another example is the return type of methods which can be covariantly redefined—the lower bound of conflicting types, if it exists, is the solution. Virtual types require a similar solution. It follows that the type system must include *intersection types*. Static typing will be examined in Section 5.7, page 84. Combining is also needed by EIFFEL contracts with disjunction of preconditions and conjunction of postconditions. Generally speaking, *method combination* is often the solution when several methods conflict—this is examined hereafter.

So the solution is *redefining*, *selecting* or *combining*, or a mix since the redefinition can be combined with the selection. For instance, in C++, selection can be done by redefining the local property, with a static call to the selected one. This static call is done with the :: operator. It is always possible in a legal model since the *lid* function (3.9) is injective. To be compatible with the syntax for global properties, we shall use a slightly different syntax here—e.g. `area::Rhombus`[7] denotes the local property `area` defined in the class `Rhombus` (Fig. 4.5). If the global property name is ambiguous, full qualification can lead to `Quadrilateral:area::Rhombus`.

Call to `super` and method combination

Call to `super` presents a similar, but more general, kind of conflict. Suppose first that the current local property l has been determined without local property conflict. Let $g \in G$ be the considered global property and $c \in X$ be the receiver's dynamic type. This means that $spec(g, c) = \{l\}$ and $loc(g, c) \backslash \{l\} = supl(l)$—see definitions in equations (3.11-3.13). In other words, all other local properties that might be combined are in $supl(l)$. Therefore, the situation is exactly the same as late binding, except that the selection or combination process must now consider $supl(l)$ instead of $spec(g, c)$. Then the set $\min_{\ll}(supl(l)) = \{l' \mid l \ll_d l'\}$ may not be a singleton, thus making `super` as ambiguous as a local property conflict. This is, for instance, the case if l has been defined to solve a local property

5. Local properties are objects, hence 'composed' of some meta-properties. So solving the conflict always amounts to combining the conflicting objects and, for each meta-property, selecting or combining. This is for instance quite explicit in the CLOS slot-description specifications.

6. En fait, il ne s'agit pas d'ensembles mais d'anti-chaînes, c'est-à-dire d'ensembles de types d'exceptions 2 à 2 non spécialisés. L'intersection est une anti-chaîne minimale qui est couverte par chacune des anti-chaînes d'origine.

7. Instead of `Rhombus::area` in C++.

conflict. The solution is to consider that `super` is legal only when $\min_{\ll}(supl(l))$ is a singleton. This is the approach of EIFFEL, with `Precursor`. If this is not a singleton, an explicit selection among the conflicting local properties is required. An alternative would be a static call, as with :: in C++. Static calls have, however, a major drawback, as they may yield multiple evaluations of the local property defined in the root `Quadrilateral` of the diamond example in Figure 4.5—consider a method `foo` with a local definition in each four classes, with each definition, except the diamond root, statically calling all local properties defined in its direct superclasses.

In a conflict case, when l has been arbitrarily selected in $spec(g, c)$, the point is that $supl(l)$ is far from including all other local properties in $loc(g, c)$—actually $spec(g, c)\backslash\{l\}$ and $supl(l)$ are disjoint non-empty sets. In a linearization framework, as in CLOS, the analogue of `super`, called `call-next-method`, involves calling the local property following l in the linearization of $loc(g, c)$. So, `call-next-method` avoids multiple evaluation risks, but linearizations also have their own drawbacks (Section 4.2.3). Of course, call to `super` can also occur in a local property which has been invoked by a call to `super`, not by a 'primary' late binding, but this does not yield any complication.

Finally, there are four possibilities :

1. `call-next-method` and linearizations, discussed hereafter ;
2. a constrained keyword `super`, which behaves like `Precursor` in EIFFEL, i.e. only sound when there is no conflict ;
3. static call, with a fully qualified syntax such as `foo::C` ;
4. a qualified use of `super`, which allows the programmer to explicitly reference the class when there is a conflict—e.g. `super⟨C⟩`.

Among these options, we exclude static calls because they explicitly mention the property name, like `super` in SMALLTALK and JAVA. The three others are all acceptable and `area::Rhombus` would be replaced by `super⟨Rhombus⟩`, but only within the code of an `area` method. Moreover, all three mechanisms are compatible with each other. A language can provide all of them—linearizations are more flexible, whereas unqualified `super` has a restricted use and its qualified version can lead to multiple evaluation.

Finally, `call-next-method` should also be considered in a top-down fashion. In BETA, `inner` is restricted to single inheritance but CLOS wrappers are integrated with `call-next-method`—actually, the same `call-next-method` function works top-down in wrappers and bottom-up in ordinary methods. In contrast, constrained or qualified `super` cannot work top-down, since there is no way to deal with conflicting subclasses. So, we shall not develop a top-down version for `call-next-method`, since it is exactly like the bottom-up mechanism, in reverse order.

4.2.3 Linearizations

Linearizations have been widely used in dynamically typed object-oriented languages such as FLAVORS, LOOPS, CLOS [DeMichiel and Gabriel, 1987], YAFOOL [Ducournau, 1991], POWER-CLASSES [ILOG], DYLAN [Shalit, 1997], PYTHON [van Rossum and Drake, 2003], etc. To our knowledge, their only use in statically typed object-oriented languages with full multiple inheritance concerns constructors and destructors in C++ [Huchard, 2000]. They are, however, also used in statically typed mixin-based languages, such as GBETA and SCALA (see Section 4.6).

Principle

The linearization principle involves computing, for each class $c \in X$, a total ordering on the set of superclasses of c, i.e.

$$supc(c) \stackrel{\text{def}}{=} \{c' \mid c \preceq c'\}. \tag{4.2}$$

This ordering is called *class precedence list* in CLOS.

Définition 4.1 (CLASS LINEARIZATION) *Given a class hierarchy $\mathcal{H}$, a* class linearization *is defined as a function $clin^{\mathcal{H}} : X \to (X \to \mathbf{N})$ (clin for short) such that $clin(c)$ (noted hereafter $clin_c$ for the sake of readability) is a bijective function $clin_c : supc(c) \to 0..|supc(c)| - 1$ (aka a* permutation*). It yields a total order $(supc(c), \leq_{clin(c)})$, whereby $x \leq_{clin(c)} y \stackrel{\text{def}}{\iff} clin_c(x) \leq clin_c(y)$. Moreover, $clin_c(c) = 0$ for all c. An alternative notation is the following : $clin(c) = (c_0, c_1, ..., c_k)$, with $c = c_0$, $supc(c) = \{c_i \mid i \in 0..k\}$ and $clin_c(c_i) = i$ for all $i \in 0..k$.*

Class linearizations only involve the poset $(X, \preceq)$ and they are just dedicated to the solution of local property conflicts. Therefore class linearizations must then be mapped from classes to local properties, i.e. from the poset $(supc(c), \preceq)$ to the poset $(loc(g, c), \ll)$, for each global property $g \in G_c$:

Définition 4.2 (Local property linearization) *Given a class hierarchy $\mathcal{H}$, equipped with a class linearization clin, a* local property linearization *is defined by the function llin : $G \times X \to (L \to \mathbf{N})$ such that $llin(g,c) = (l_0, l_1, ..l_m)$, with $loc(g,c) = \{l_i \mid i \in 0..m\}$ and $0 \leq i < j \leq m \Rightarrow clin_c(def(l_i)) < clin_c(def(l_j))$. It yields an analogous total order $(loc(g,c), \leq_{llin(g,c)})$.*

In this framework, the selection function *sel* selects the first property in this ordering—i.e. $sel(g,c) = l_0$—and the call to `super` is carried out by the `call-next-method` mechanism :

Définition 4.3 (Call next method) *The* call next method *mechanism relies on the partial function $cnm : X \times L \to L$ such that, with the previous definition notations, $cnm(c, l_i) = l_{i+1}$ when $i < m$, and $cnm(c, l_m)$ is undefined.*

So when used for combination, linearizations avoid possible multiple evaluations which may occur with C++ static calls or with qualified `super`. It is essential to note that, in the expression $cnm(c, l_i)$, c is not the class which defines l_i, i.e. $def(l_i)$, but the receiver's dynamic type [8], hence $c \preceq def(l_i)$. Contrary to single inheritance, $l \ll cnm(c,l)$ is not always verified. Here, some authors diagnose a drawback—this would break class modularity. This seems, however, unavoidable in method combination when inheritance is multiple.

Requirements

Some theoretical studies have determined what should be a 'good' linearization. We review here their main conclusions.

Linear extensions In order to ensure that the selection respects the *masking rule* (Req. 4.1), i.e. that no other property would be more specific—the total order must be a *linear extension* [9]. This means that

$$c \preceq c' \preceq c'' \Longrightarrow clin_c(c') \leq clin_c(c'') \tag{4.3}$$

or, equivalently, that the restriction $\preceq / supc(c)$ is a subset of $\leq_{clin(c)}$, for all $c \in X$. This implies that the selected property is taken from the *conflict set* [Ducournau et al., 1995]—i.e. $def(sel(g,c)) \in cs(g,c)$. This requirement is easy to meet and is satisfied in most recent languages—it was actually satisfied in mid-80s languages—with the notable exception of Python [10]. When used for selection, linear extensions have the desired behaviour when there is no local property conflict—they select the single most specific local property. Hence, when there is a conflict, linearizations represent only a default selection mechanism and the programmer can switch it off anyway by simply redefining the property to solve the conflict. From now on and unless otherwise stated, we shall consider that all linearizations are linear extensions.

Monotonicity, i.e. linearization inheritance Another important requirement is that the linearization should be *monotonic*—i.e. the total ordering of a class extends that of its superclasses. This amounts to inheriting linearizations—i.e. $\leq_{clin(c')}$ is a subset of $\leq_{clin(c)}$, for all $c \prec c'$ in X—or, equivalently, it means that :

$$c \preceq c' \preceq c'', c''' \Longrightarrow \big(clin_c(c'') \leq clin_c(c''') \iff clin_{c'}(c'') \leq clin_{c'}(c''')\big) \tag{4.4}$$

Since $clin_c(c) = 0$, a monotonic linearization is always a linear extension.

Obviously, when the linearization is used for combination, monotonicity makes the order of method invocations preserved by inheritance—of course, *llin* is also monotonic. Furthermore, monotonicity implies a nice property when the linearization is used for selection—namely, a class always behaves like at least one of its direct superclasses or, equivalently, *inheritance cannot skip a generation.*

However, this second requirement is not as easy to meet as the first one. Actually, given a class hierarchy $(X, \preceq)$ equipped with a monotonic linearization *clin*, and two classes $c_1, c_2 \in X$, it may be impossible to extend the hierarchy to a common subclass of c_1 and c_2 without losing monotonicity, because $clin_{c_1}$ and $clin_{c_2}$ conflict on some pair x, y—i.e. $clin_{c_1}(x) < clin_{c_1}(y)$ and $clin_{c_2}(x) > clin_{c_2}(y)$. This *linearization conflict* involves a cycle in the union of $\leq_{clin(c_1)}$ and $\leq_{clin(c_2)}$.

8. So, the existence of this 'next' method cannot always be statically—i.e. when compiling c—ensured and an auxiliary function `next-method-p` allows the programmer to check it at run-time. However, in the present framework, when the linearization is a linear extension (see hereafter), this run-time check is only required when the method has been declared *abstract* in superclasses.

9. Linear extensions are also called *topological sorting* [Knuth, 1973].

10. En fait, Python 2 a 2 linéarisations différentes : un parcours en profondeur pour les '*classic classes*' et la linéarisation monotone C3 pour les '*new-style classes*'.

So, in practice, monotonicity must likely remain a desired but not required property. For instance, the strategy proposed by [Forman and Danforth, 1999] involves considering whether the disagreement provoked by such a conflict is 'serious'. by merging their definitions.

Local and extended precedence order The actual linearization principle is to totally order unrelated superclasses, especially direct superclasses. As such orderings are rather arbitrary, they are usually explicitly given by the programmer, for instance as the order of superclass declaration—hence, in Definition 3.5, page 33, `supernames` is an *ordered* set. These orders are called *local precedence orders* in CLOS, and the linearization is required to respect them. This is, however, not always possible, for the same reasons as monotonicity.

Prospects

Linearizations can be further improved in several ways.

Partial linearizations It follows from Definitions 4.1 and 4.2, page 44 that linearizations are only intended to order local properties, for selecting and combining them. Therefore, only $llin(g, c)$ should be required to be monotonic linear extensions. For instance, if there is no local property conflict, local precedence orders do not matter and any linear extension makes a good linearization. In the case of a linearization conflict, if there is no conflict between local properties in the cycle, this cycle in $clin(c)$ does not appear in any $llin(g, c)$ and the cycle ordering does not matter at all. So, instead of computing $clin(c)$ as a total order, one could restrict it to a partial order that would only totally order all local properties, for each global property $g \in G_c$. More precisely, $\leq_{clin(c)}$ could be defined as $\bigcup_{g \in G_c} def(\leq_{llin(g,c)})$. Thus, monotonicity should be more often—but still not always—preserved.

Specific linearizations However, one might also allow the programmer to specify a partial linearization for each global property—there is no necessity for two global properties to combine their local properties in the same order. So each class could provide a default linearization which could be overridden by the programmer for individual global properties. This is, however, just a research issue.

Finally, linearizations can coexist with other combination techniques, e.g. the qualified call to `super` that we propose.

4.3 Les solutions proposées par les différents langages

Aucun langage « grand public » ne dispose du nommage non ambigu défini plus haut permettant de traiter correctement l'héritage multiple. Nous considérerons juste ici C++, EIFFEL et CLOS.

4.3.1 Héritage multiple en C++

Le comportement de C++ diffère d'abord suivant qu'il s'agit des attributs et des méthodes. Il est de plus paramétrable par la présence du mot-clé `virtual` pour annoter les super-classes.

Dans tous les cas, il est utile de se reporter à l'implémentation du langage (Chapitre 9) pour comprendre la façon dont il fonctionne.

Le cas des attributs

Par défaut, c'est-à-dire sans le mot-clé `virtual`, C++ traite mal l'héritage de propriétés globales (de nom) pour les attributs. Cela se traduit par un *héritage répété* de A par D, c'est-à-dire la duplication des attributs de A dans une instance de D (Figure 4.6). On parlera dans de cas d'*héritage non-virtuel*. Dans ce cas, la notation :: permet de désigner l'un des 2 attributs faussement en conflit. Cet héritage répété introduit des redondances qui sont inacceptables : si A est la classe `Personne`, ses attributs `nom`, `âge` et `adresse` seraient dupliqués et auraient des valeurs différentes suivant qu'on les accède par des méthodes de B ou de C.

La présence du mot-clé `virtual` dans la définition des classes B et C est donc nécessaire pour éviter cet héritage répété. On parle alors d'*héritage virtuel*. C'est la seule façon correcte de faire de l'héritage multiple en C++ de façon complètement réutilisable, c'est-à-dire de telle sorte que les classes puissent être spécialisées en héritage multiple sans limitation, au moins en ce qui concerne les attributs.

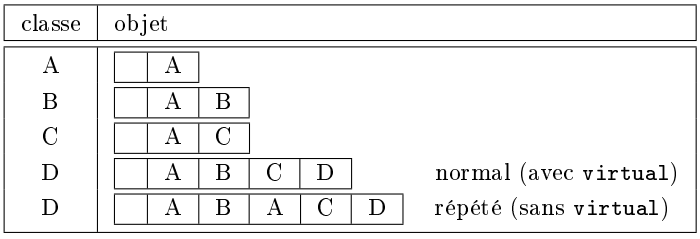


FIGURE 4.6 – L'implémentation des objets avec un héritage normal ou répété : dans chaque table le nom de la classe représente l'ensemble des attributs qu'elle introduit.

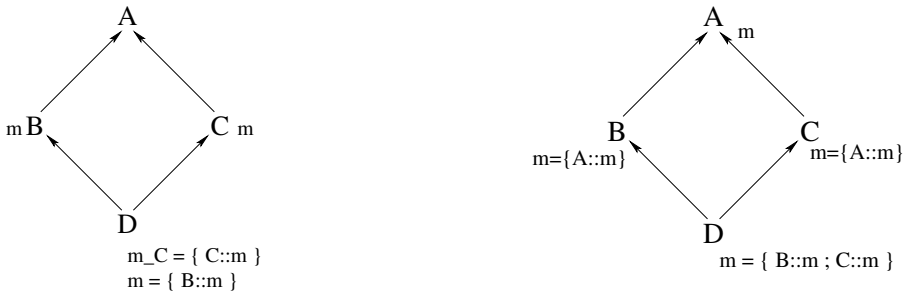


FIGURE 4.7 – Héritage multiple en C++ : bricolage pour le conflit de propriétés globales (gauche) et héritage répété pour le conflit de propriétés locales (droite).

Le cas des méthodes

Dans le cas des méthodes, il faut distinguer les conflits de propriétés locales et globales. Rappelons d'abord qu'en C++ une propriété globale s'identifie par son nom et les types de ses paramètres. Les conflits doivent donc s'analyser à types de paramètres identiques. Si les types de paramètres sont différents, il s'agit d'une *surcharge statique* (Section 6.3, page 92).

Dans les 2 cas de conflits, le langage ne fait rien par défaut et il ne réagit que si l'on cherche à utiliser la propriété conflictuelle dans un contexte où elle est conflictuelle. Si m est la méthode en conflit, la séquence de code `{D* x; x->m();}` est impossible à moins de redéfinir m dans D.

Dans le cas d'un conflit de propriétés globales, cela signifie que les 2 propriétés m de B et C restent distinctes dans D tant qu'on ne cherche pas à faire `{D* x; x->m;}`. En revanche, dès que m est redéfini dans D, les 2 propriétés sont unifiées.

Par ailleurs, la notation :: permet aussi bien de résoudre un conflit de propriétés globales qu'un conflit de propriétés locales, 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, page 38, si une méthode m est définie dans chacune des 4 classes de telle sorte que m_B et m_C fassent appel à A::m et que m_D fasse appel à B::m et C::m, alors le code de m_A sera exécuté deux fois sur les instances de D (Figure 4.7 droite).

En pratique, face à un conflit de propriétés globales, le programmeur C++ est relativement désarmé. S'il définit la classe D, la redéfinition de m vaudra autant pour m_B que pour m_C : 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 m_C en m_C, qui appellerait C::m. La redéfinition de m en D serait une redéfinition de m_B, dont le comportement serait normal (Figure 4.7 gauche). Pour m_C, sur un receveur de type statique D, on utilise m_C. Mais le comportement est incorrect lorsque l'on appelle m_C sur un receveur de type statique C mais de type dynamique D : c'est m_B, 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.

EXERCICE 4.2. Définir les 4 classes du losange (ne pas oublier de respecter les règles de la programmation objet en C++, Section 5.3.1, page 76). Examiner le comportement de ces classes suivant que l'héritage se fait avec le mot-clé `virtual` ou pas. □

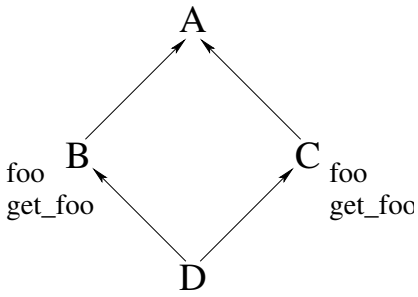


FIGURE 4.8 – Héritage multiple en C++ : incohérence attribut / méthode. La classe D possède deux attributs, mais doit les accéder par un unique accesseur.

Cohérence attribut / méthode

Le plus fascinant dans la spécification de l'héritage multiple en C++ est qu'il se comporte de façon différente suivant que sont concernés des attributs et des méthodes. En cas de conflit sur deux attributs de même nom, munis d'accesseurs de même nom, la classe considérée sera bien munie de deux attributs, mais n'aura qu'un unique accesseur pour y accéder (Figure 4.8).

4.3.2 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 propriétés globales du moins. En cas de conflit de propriétés globales, un renommage est imposé dans la déclaration de la classe (mot-clé `rename`).

En EIFFEL, pour l'héritage de propriétés locales, le masquage n'est pas implicite. À partir du moment où plusieurs propriétés locales sont accessibles par des chemins différents, même dans le cas de la propriété p qui ne présente pas formellement de conflit de propriétés locales, EIFFEL impose au programmeur un masquage manuel avec le mot-clé `undefine`.

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

EXERCICE 4.3. Définir les 4 classes du losange en EIFFEL, en utilisant les mots-clés `redefine`, `undefine` et `rename` pour obtenir le comportement souhaité, c'est-à-dire celui du méta-modèle. □

4.3.3 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 à son tour spécialiser (mot-clé `extends`) plusieurs interfaces. Les interfaces ne déclarant que des signatures de méthodes, seul l'héritage de propriétés globales est multiple en JAVA.

Curieusement, l'héritage de propriétés globales de JAVA ressemble plus à celui des langages à typage dynamique qu'à celui des langages à typage statique : en cas de conflit de propriétés globales, 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. 6.3.9). 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.

Ce défaut de JAVA a été identifié et une proposition des *hygienic methods*, basée sur des noms qualifiés, a été faite mais sans lendemains [Kusmierek and Bono, 2007].

Le système de types de JAVA a fait de nombreux émules : tous les langages de la plateforme .NET de Microsoft, dont C♯ sont dans ce cas. EIFFEL# représente ainsi une version d'EIFFEL compatible avec ce système de types. La dernière révision d'ADA 2005 est dans le même cas.

EXERCICE 4.4. Définir les 4 classes et interfaces du losange en JAVA, et montrer le comportement du langage en ce qui concerne le conflit de propriétés globales. □

4.3.4 Héritage multiple en .NET et C♯

Le principe du sous-typage multiple s'applique à .NET de façon similaire à ce qu'il est en JAVA. Cependant, le langage C♯ autorise (mais n'impose pas) une qualification des noms dans le cas d'un

conflit de propriétés globales, lorsque la même signature de méthode est introduite par plusieurs interfaces. La redéfinition de la méthode dispose alors d'une syntaxe particulière pour désigner la bonne propriété. En revanche, aucune syntaxe n'est disponible pour l'appel de méthode et le programmeur doit passer par un *cast*, comme pour la désambiguïsation de la surcharge statique (Section 6.3.2, page 93).

*****(à développer...)*****

4.3.5 Héritage multiple en CLOS

En CLOS le problème de l'héritage de propriétés globales 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 des attributs, on verra le problème des méthodes aux chapitres 6 et 12).

Pourtant, les attributs et les méthodes sont réifiés, mais les premiers ne constituent pas *propriétés globales* : 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 propriétés locales, CLOS est basé sur un mécanisme original à base de *linéarisation* (section suivante) et la combinaison des méthodes est réalisée par la fonction `call-next-method` qui fait appel à la méthode suivante dans la linéarisation (Section 6.7, page 105). Cette approche évite l'héritage répété causé par la notation :: de C++.

4.4 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*.

4.4.1 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.

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. En pratique, la linéarisation est calculée une bonne fois pour toutes pour chaque classe, puis utilisée en général statiquement, par exemple pour remplir la table des méthodes.

Le problème est donc celui de la spécification de cet ordre total, qui est appelé une *linéarisation*.

4.4.2 Propriétés des linéarisations

Un certain nombre de critères permettent de choisir parmi les nombreux ordres possibles.

(1) Extension linéaire

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 contionuera à respecter 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*.

Définition 4.4 (Extension linéaire) *Soit $(E, \leq)$ un ensemble partiellement ordonné. Une* extension linéaire *de cet ordre partiel est un ordre total $(E, \leq_t)$ tel que $\leq \subseteq \leq_t \subset E \times E$ ou, de façon équivalente, tel que $x \leq y$ implique $x \leq_t y$.*

La recherche d'une propriété en suivant une extension linéaire respecte l'Axiome 4.1 de masquage. Elle retournera toujours un élément de l'ensemble de conflit de propriétés locales, 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) pourr 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 toujours compatibles : l'ordre local n'est pas forcément sans circuits puisqu'il est la réunion de plusieurs ordres totaux indépendants (Figure 4.9a). De plus, même si cet ordre local est sans circuits, sa réunion avec la relation de spécialisation n'est pas forcément sans circuits.

Malgré ces problèmes potentiels, on peut définir la structure de graphe sur laquelle peuvent s'appliquer les linéarisations :

Définition 4.5 (Graphe de spécialisation d'une classe) *Un* graphe de spécialisation d'une classe *est un* graphe enraciné sans circuit *formé d'un triplet* $\langle X_c, super, c\rangle$ *où* c *est une classe,* X_c *est un ensemble de classes incluant* c*, et* $super : X_c \to 2_c^X$ *une fonction qui associe à chaque classe* $x \in X_c$ *l'ensemble totalement ordonné* $super(x) = (y_1, .., y_k) \subset X_c$ *de ses super-classes directes. De plus,* X_c *est la fermeture de* c *par* $super$*.*

Noter que $super$ ne dépend pas de c, ce qui signifie que l'ordre total $super(x)$ sera le même pour toutes les sous-classes de x. En revanche, $super(x)$ est une propriété de x, pas des y_i qui peuvent être ordonnées autrement par une autre sous-classe directe commune.

Topologie du graphe et arcs de transitivité

A ces premières propriétés, il faut rajouter une exigence : la linéarisation ne doit reposer que sur la topologie du graphe de spécialisation, y compris les ordres locaux de priorité. Cela peut paraître naturel, mais on pourrait imaginer utiliser de nombreuses autres informations comme le nom de la classe, sa date de création, voire l'âge du concepteur !

Par ailleurs, la relation de spécialisation forme une relation d'ordre partiel dont l'interprétation (Chapitre 2) ne tient pas compte des arcs de transitivité. La question se pose donc de savoir s'il faut tenir compte d'éventuels arcs de transitivité dans la définition de la linéarisation : si la définition d'une classe liste une super-classe indirecte parmi ses super-classes directes, cela peut avoir un effet sur l'ordre local de priorité, donc sur le résultat final.

Cela ne nous paraît pas souhaitable et nous décrirons des linéarisations et des algorithmes qui sont tous basés sur le graphe de couverture de la relation d'ordre (c'est-à-dire sur sa réduction transitive [11]). Cependant, les linéarisations de plusieurs langages (Clos, Dylan, Python) tiennent compte des transitivités.

Formellement, ces deux contraintes nécessitent de passer par la notion d'isomorphisme entre 2 graphes de spécialisation de classes :

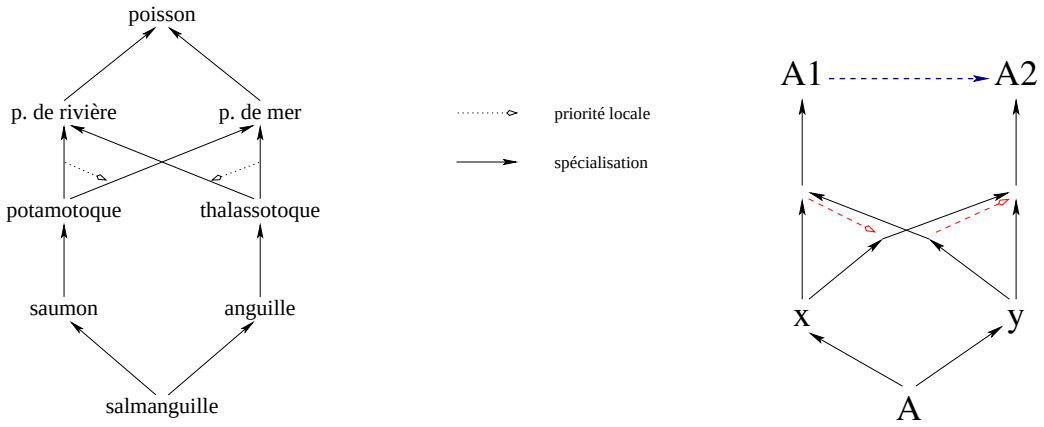
Définition 4.6 (Isomorphisme de graphes de spécialisation de classe) *Soit* $\langle X, super, c\rangle$ *et* $\langle X', super, c'\rangle$ *deux graphes de spécialisations des classes* c *et* c'*. L'isomorphisme entre ces deux structures est une bijection* $f : X \to X'$ *telle que* $f(c) = c'$ *et, pour toute classe* $x \in X$*,* $super(x) = (y_1, .., y_k)$ *si et seulement si* $super(f(x)) = (f(y_1), .., f(y_k))$*.*

On souhaite donc que deux graphes de spécialisation isomorphes aient des linéarisations isomorphes. Il faut comprendre cette contrainte à la lumière de l'*hypothèse du monde ouvert* (Méta-axiome 2.2) : la linéarisation d'une classe ne doit pas dépendre de ses sous-classes.

(2) Monotonie

Une fois ces contraintes fondamentales satisfaites, on peut envisager une autre propriété très intéressante mais qui se révèle plus dure à satisfaire : la linéarisation doit être *monotone*. La *monotonie*

11. Graphe de couverture, réduction transitive et réflexive, Diagramme de Hasse : quelques quasi synonymes dont vous devez rechercher la définition de toute urgence s'ils ne vous sont pas familiers.



(a) Conflit de priorité locale (b) Priorité étendue

FIGURE 4.9 – (a) 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. D'après Ducournau et Habib [1989].
(b) Ordre de priorité étendu : pour A, l'ordre entre A_1 et A_2 est déduit de l'ordre de priorité locale déterminé par leurs sous-classes communes maximales x et y.

repose sur le fait que l'ordre total de la linéarisation d'une classe est préservé par les linéarisations de ses sous-classes [Ducournau et al., 1992, 1994] :

$$B \prec A \Rightarrow L(A) \subset L(B)$$

On voit bien l'interêt de cette propriété : toutes les classes trancheront les mêmes conflits de la même façon, ou combineront les mêmes méthodes dans le même ordre.

Une linéarisation monotone respecte forcément l'ordre local de priorité — en fait, elle en définit un.

(3) Ordre étendu de priorité

L'ordre local de priorité peut aussi être généralisé : si C a pour super-classes directes $B_1, .. , B_k$, alors pour tout i, j tels que $0 < i < j \leq k$, et pour toutes classes A_i et A_j incomparables par $\prec$ et super-classes respectives de B_i et B_j (i.e. $B_i \prec A_i$ et $B_j \prec A_j$), alors A_i est avant A_j dans la linéarisation de C [Huchard et al., 1991].

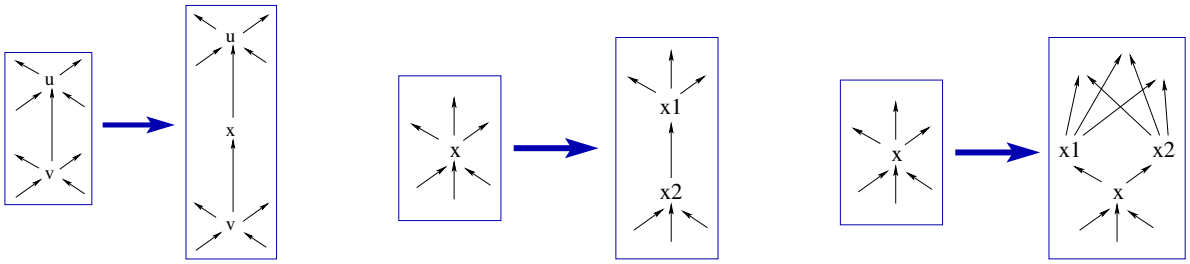
La définition exacte est un peu plus compliquée : pour tout couple de classes incomparables A_i et A_j, on ne considère pas la priorité locale induite par A mais celle qui est induite par les sous-classes communes maximales de A_i et A_j, qui sont aussi super-classes de A.

Stabilité.

Une autre propriété importante est la *stabilité*, c'est-à-dire le fait que la linéarisation ne change pas fondamentalement en cas de petite modification incrémentale de la hiérarchie de classes. Plusieurs modifications élémentaires sont à considérer, qui représentent toutes des opérations naturelles de reconception d'une hiérarchie. Dans chaque cas, les propriétés définies dans une classes sont réparties dans deux autres classes et la classe rajoutée doit simplement s'insérer dans les linéarisations, sans les bouleverser :

- *subdivision d'un arc* : en ajoutant un sommet x sur un arc (u, v), la linéarisation d'une sous-classe a, $(a, .., u, .., v, .., z)$, doit être simplement transformée par l'insertion de x quelque part entre u et v : $(a, .., u, .., x, .., v, .., z)$;
- *éclatement série d'un sommet* : la fission d'un sommet x en deux sommets x_1 et x_2 tels que $x_2 \prec x_1$, où les super-classes directes de x deviennent celles de x_1 et ses sous-classes directes deviennent celles de x_2 : $(a, .., x, .., z)$ doit alors se transformer en $(a, .., x_2, x_1, .., z)$;
- *éclatement parallèle d'un sommet* : le contenu de la classe x est éclaté dans deux classes x_1 et x_2 incomparables, dont les super-classes sont celles de x et dont x est la sous-classe commune : $(a, .., x, .., z)$ doit alors se transformer en $(a, .., x, x_1, x_2, .., z)$.

Ces transformations ne prétendent pas à l'exhaustivité : par exemple, l'éclatement parallèle pourrait se généraliser au cas n-aire, et chacune des classes résultantes n'a pas forcément l'obligation de conserver toutes les super-classes de la classe d'origine.



(a) Subdivision d'arc (b) Eclatement série (c) Eclatement parallèle

FIGURE 4.10 – Modifications incrémentales pour lesquelles les linéarisations devraient être stables.

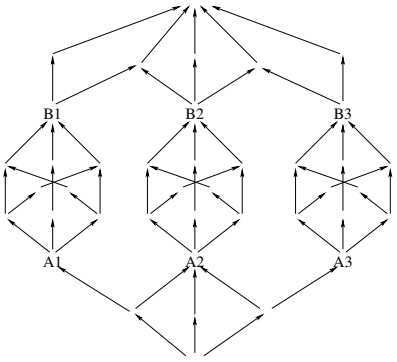


FIGURE 4.11 – Préservation des modules : ce graphe de spécialisation de classe contient 3 modules entre les classes A_i et B_i, pour $i = 1, 2, 3$. Tous les chemins de l'intérieur vers l'extérieur du module passent par A_i ou B_i. Dans une linéarisation qui préserve les modules, toute linéarisation d'une sous-classe de A_i ordonnera de façon consécutive l'intérieur de chaque module A_i-B_i.

EXERCICE 4.5. De façon générale, la monotonie favorise la stabilité. Montrer que toute linéarisation monotone est stable pour ces 3 transformations. □

Préservation des modules.

La modularité est au cœur des préoccupations du génie logiciel. Il se trouve que la notion de module et de décomposition modulaire est aussi centrale en théorie des graphes. Les deux domaines se rencontrent dans le fait les linéarisations peuvent préserver les modules, c'est-à-dire les linéariser de façon consécutive.

C'est le cas pour la plupart des linéarisations connues lorsqu'un module est défini de la façon suivante.

Définition 4.7 (Module du graphe d'héritage) *Soit deux classes x et y d'une hiérarchie de classes, avec $y \prec x$. On dit que x et y forment un module si, pour toute classe z comprise entre x et y ($y \preceq z \preceq x$), toute super-classe de z est comparable avec x et toute sous-classe de z est comparable avec y.*

En d'autres termes, x et y forment un « losange » dont l'intérieur ne communique avec l'extérieur qu'en passant par x et y.

La préservation du module se traduit par le fait que a linéarisation de toute sous-classe de y contient toutes les classes du module de façon consécutive.

4.5 Quelques linéarisations

Nous allons voir plusieurs linéarisations :

— la plus simple est utilisée par C++ et par SCALA (voir Section 4.6.2, page 60), après avoir été introduite par COMMON LOOPS : elle ne vérifie que la première propriété ;

— celle de CLOS est un peu moins simple mais vérifie les deux premières ;

— la linéarisation C3, utilisée partiellement en PYTHON, les vérifie toutes et a aussi l'avantage d'être relativement simple ; des variantes sont aussi utilisées dans le langage GBETA ainsi que dans le système SOMOBJECTS TOOLKIT d'IBM ;

— la linéarisation de DYLAN est elle-aussi monotone, mais elle présente la même complication que celle de CLOS.

Suivant les langages, la linéarisation peut être dénommée différemment : *class precedence list* en CLOS et DYLAN, *method resolution order* en PYTHON et en SOMOBJECTS TOOLKIT.

4.5.1 Linéarisation de C++ et COMMON LOOPS

Cette linéarisation est classique et simple (à calculer ou à comprendre) mais elle possède un gros défaut : elle ne respecte pas bien la priorité locale. On peut la décrire par différents algorithmes. Introduite par COMMON LOOPS, elle est utilisée par C++ pour les constructeurs et destructeurs, ainsi que par SCALA dans un contexte de *mixins* (cf. Section 4.6.2).

Remarque 4.2. Ne pas confondre la linéarisation, qui est un objet mathématique particulier, avec l'algorithme qui permet de la calculer : cette section présente une unique linéarisation, mais plusieurs algorithmes.

Algorithme 1. Soit une classe A, de super-classes directes $B_1, \ldots, B_n$. 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, dans l'ordre, en ajoutant A en premier :

$$L = A \,.\, L(B_1) + \cdots + L(B_n)$$

   « . » et « + » sont ici des équivalents des fonctions `cons` et `append` en LISP : la notation non parenthésée suppose que « . » a une priorité moins élevée. La liste vide est bien sûr l'élément neutre de toutes les opérations de concaténation considérées dans la suite : $x + () = () + x = x$;

2. dans la liste L ainsi obtenue, on enlève les doublons en conservant la *dernière occurrence* : si $L = C_1, \ldots, C_n$, pour tout i, j avec $i < j$, si $C_i = C_j$, on enlève C_i.
   $L(A)$ est la liste résultant de cette élimination.

Exemple. Ainsi, dans la figure du losange, en supposant que (B, C) est l'ordre de priorité local des super-classes de D, on a :

$L(A)$	=	(A)	
$L(B)$	=	(B, A)	
$L(C)$	=	(C, A)	
$L(D)$	=	$(D, B, \underline{A}, C, A)$	on supprime le doublon souligné

Algorithme 1b. Alternativement, on fusionne les 2 étapes en définissant la linéarisation par une concaténation sans doublon :

$$L(A) = A \,.\, L(B_1) \oplus \cdots \oplus L(B_n)$$

où la concaténation sans doublon $\oplus$ est définie sur des listes non vides par

$$(a_1, .., a_k) \oplus (b_1, .., b_m) = \begin{cases} (a_2, .., a_k) \oplus (b_1, .., b_m) \text{ si } a_1 \in \{b_1, .., b_m\} \\ a_1 \,.\, (a_2, .., a_k) \oplus (b_1, .., b_m) \text{ sinon} \end{cases}$$

Parcours en profondeur. Les deux autres algorithmes reposent sur un *parcours en profondeur*. Dans un graphe orienté sans circuit comme une hiérarchie de spécialisation, un tel parcours peut se déterminer par 3 éléments :

1. l'ordre de parcours des successeurs d'un sommet ;
2. le fait de marquer ou pas un sommet quand on y arrive pour la première fois (s'il est déjà marqué, on revient en arrière) ;
3. le moment où l'on ramasse (ou numérote) un sommet : en arrivant (descendant, empilant) ou en repartant (remontant, dépilant).

Par exemple, PYTHON utilise pour les '*classic classes*' un parcours en profondeur *avec marquage* et ramassage *en empilant*, en prenant les successeurs dans l'*ordre* local de priorité. Une telle linéarisation n'est pas une extension linéaire : sur l'exemple du losange (Figure 4.1), elle retourne (D, B, A, C).

Algorithme 2. Il consiste en un parcours en profondeur, *sans marquage*, avec ramassage *en empilant*, en prenant les successeurs dans l'*ordre* de la priorité locale. On enlève ensuite les doublons en gardant les *dernières occurrences*.

Algorithme 3. Il consiste en un parcours en profondeur, *avec marquage* et ramassage *en dépilant*, en prenant les successeurs dans l'*ordre inverse* de la priorité locale. On *inverse* ensuite le résultat.

Propriétés. Cette linéarisation a deux propriétés :
— c'est une extension linéaire de la relation de spécialisation ;
— elle est compatible avec l'ordre local, sauf dans un cas (Figure 4.12a) ;
— elle n'est donc pas monotone ;
— mais elle est stable pour la subdivision d'arc et l'éclatement série.

Exercices.

Exercice 4.6. Montrer que $\oplus$ est bien associatif (c'est nécessaire si l'on veut que l'algorithme 1b soit bien spécifié). □

Exercice 4.7. Ecrire la fonction Lisp qui implémente $\oplus$, dans le cas binaire puis dans le cas n-aire. S'inspirer de la fonction Lisp `append` (voir polycopié de Lisp [Ducournau, 2013]). □

Exercice 4.8. Montrer que l'algorithme 1 calcule bien une extension linéaire de la relation de spécialisation. □

Exercice 4.9. Montrer que le résultat de l'algorithme 1 respecte bien la priorité locale, sauf dans le cas de la figure 4.12a. En considérant l'exemple de la figure 4.12b, préciser comment il faut généraliser ce contre-exemple pour le rendre générique. □

Exercice 4.10. Montrer que les algorithmes 1 et 2 calculent bien la même linéarisation. □

Exercice 4.11. Montrer que les algorithmes 1 et 3 calculent bien la même linéarisation. □

Exercice 4.12. Ecrire la fonction Lisp, prenant en entrée des listes et retournant une liste, qui implémente l'algorithme 1. □

Exercice 4.13. Quel algorithme a la meilleure complexité ? □

Exercice 4.14. Montrer que cette linéarisation est stable pour la subdivision d'arc et l'éclatement série. □

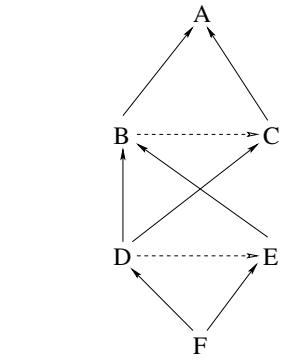
Exercice 4.15. Montrer que cette linéarisation préserve les modules. □

Exercice 4.16. Définir en C++ les classes des exemples des figures 4.9, 4.12 et 4.13 et tracer les exécutions pour en déduire la linéarisation utilisée par les constructeurs et les destructeurs. □

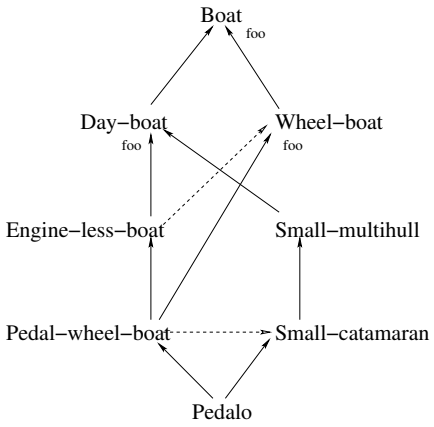
Remarque 4.3. La linéarisation de C++ utilise bien la linéarisation précitée, mais l'ordre local de priorité est donnée par l'ordre inverse de l'ordre de déclaration des superclasses. Par ailleurs, pour faire ces exemples dans ce langage, il ne faut pas oublier les consignes générales pour faire de la programmationj objet en C++ (voir Section 5.3.1, page 76).

4.5.2 Linéarisation de Clos

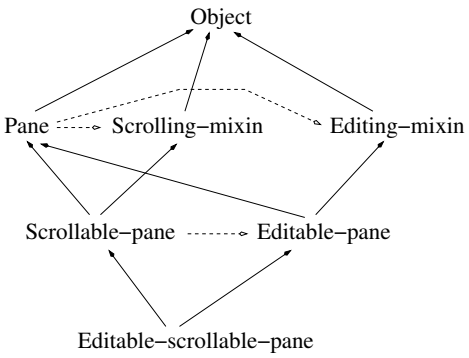
Cette deuxième linéarisation a été conçue pour corriger le principal défaut de la précédente, le non respect de la priorité locale. Elle est basée sur le calcul d'une extension linéaire de la *réunion des relations de spécialisation et des ordres locaux de priorité*, lorsque c'est possible, c'est-à-dire qu'il n'y a pas de circuit. En cas de circuit, une exception est signalée. La linéarisation respecte donc toujours les deux premières propriétés recherchées.



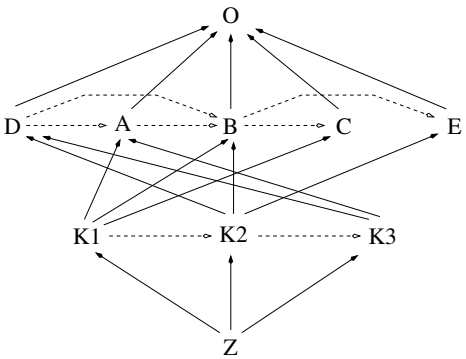
(a) Priorité locale



(b) Monotonie 1



(c) Monotonie 2



(d) Monotonie 3

FIGURE 4.12 – Contre-exemples : le contre-exemple générique du respect de la priorité locale par la linéarisation de C++ (a), d'après Ducournau et Habib [1989], et celui de la monotonie par la linéarisation de CLOS (b), d'après Ducournau *et al.* [1994]. Les deux derniers exemples (c-d) sont trouvés sur le Web. Dans chaque exemple de monotonie, 2 classes x et y incomparables (par la spécialisation et la priorité locale) peuvent être inversées dans les linéarisations finales de deux sous-classes w et z, avec $w \prec z \prec x, y$.

Algorithme 4 (général naïf). Un algorithme général naïf de calcul d'une extension linéaire est le suivant : tant qu'il reste des sommets, prendre un sommet minimal suivant la relation considérée. Les extensions linéaires diffèrent suivant le *critère de choix du minimal*, lorsqu'il y en a plusieurs, et toute extension linéaire peut être calculée par ce schéma général d'algorithme.

On rappelle la définition d'un minimal dans une relation d'ordre :

Définition 4.8 *Soit $(E, \leq)$ une relation d'ordre partiel, F une partie de E et $x \in F$. On dit que x est* minimal *dans F ssi il n'existe pas $y \in F$ tel que $y \leq x$ et $y \neq x$.*

On utilise le terme de minimum *pour un minimal unique. Les termes de* maximal *et* maximum *ont une définition duale.*

Algorithme 5 : critère de choix du minimal en Clos. La linéarisation de Clos se définit par le critère de choix[12] d'un minimal : 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, $C_1, \ldots, C_k$, et qu'il y ait m minimaux $M_1, \ldots, M_m$ dans les classes restantes. On va considérer les couples (j, i) tels que $C_j \prec M_i$, et on prendra le M_i dont le couple (j, i) maximise j, donc tel qu'il n'existe pas de couple (j', i') tel que $j' > j$ et $C_{j'} \prec M_{i'}$.

Non monotonie et instabilité. La figure 4.12b montre un exemple où la linéarisation de Clos respecte bien l'ordre de priorité locale mais n'est pas monotone.

$L(Pwb)$	$=$	$(Pwb, Elb, \mathbf{Db}, \mathbf{Wb}, B)$	Db avant Wb
$L(P)$	$=$	$(P, Pwb, Elb, \mathbf{Wb}, Sc, Sm, \mathbf{Db}, B)$	Wb avant Db

Cela peut avoir des conséquences importantes dans la combinaison des méthodes, les deux méthodes `foo` de l'exemple n'étant plus appelées dans le même ordre suivant le type dynamique du receveur. Cela peut aussi avoir des conséquences très paradoxales sur la *monotonie* du comportement : les instances directes de `Pedal-wheel-boat` et de `Small-catamaran` sélectionnent la méthode `foo` de `Day-boat`. Et pourtant les instances de leur sous-classe commune `Pedalo` sélectionnent la méthode `foo` de `Wheel-boat`. Il serait pourtant normal d'exiger que pour chaque méthode héritée, une classe se comporte comme au moins une de ses super-classes directes. Contrairement à la génétique, l'héritage ne devrait pas sauter de génération.

La linéarisation est aussi instable : la hiérarchie de la figure 4.12b provient de la transformation de celle de la figure 4.12a par l'ajout de `Engine-less-boat` et `Small-multihull` (subdivision d'arc). Ces deux ajouts provoquent l'inversion de B (`Day-boat`) et C (`Wheel-boat`).

Exemple : les polygones

Soit la hiérarchie de classes de la figure 4.13, 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 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 $L(C) = (C, L, R, Pa, T, Q, Pr, Pcr, Par, P)$.

On constate sur la figure 4.13 que les deux linéarisations donnent le même résultat sur cet exemple.

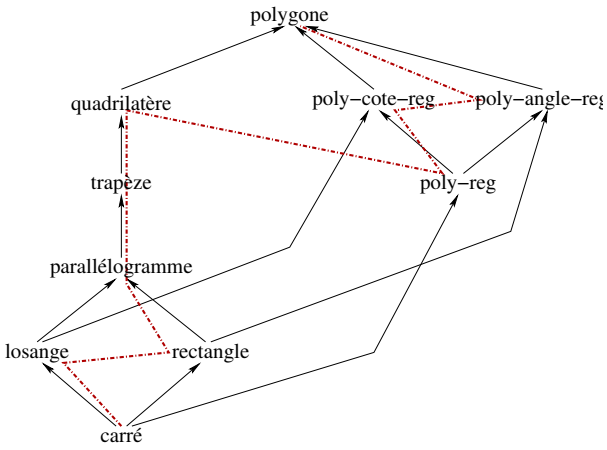
Exercices.

Exercice 4.17. Montrer que la linéarisation de Clos respecte bien la priorité locale, même dans le cas générique de la figure 4.12a. □

Exercice 4.18. Définir en Clos les classes des exemples des figures 4.9, 4.12 et 4.13 et vérifier les propriétés des linéarisations obtenues, en particulier leur monotonie.

Identifier, dans chaque exemple de monotonie, le quadruplet w, x, y, z, avec $w \prec z \prec x, y$, où les 2 classes x et y sont incomparables (par la spécialisation et la priorité locale) et inversées dans les linéarisations finales de leurs deux sous-classes w et z. □

12. Dans la littérature anglo-saxonne, le terme de *tie-break* est souvent utilisée.



$L(P)$	=	(P)	
$L(Pcr)$	=	(Pcr, P)	
$L(Par)$	=	(Par, P)	
$L(Pr)$	=	$(Pr, Pcr, \underline{P}, Par, P)$	on supprime le doublon souligné
$L(Q)$	=	(Q, P)	
$L(T)$	=	(T, Q, P)	
$L(Pa)$	=	(Pa, T, Q, P)	
$L(L)$	=	$(L, Pa, T, Q, \underline{P}, Pcr, P)$	idem
$L(R)$	=	$(R, Pa, T, Q, \underline{P}, Par, P)$	idem
$L(C)$	=	$(C, L, \underline{Pa}, \underline{T}, \underline{Q}, \underline{Pcr}, \underline{P}, R, Pa, T, Q, \underline{Par}, \underline{P}, Pr, Pcr, Par, P)$	idem

FIGURE 4.13 – La hiérarchie de polygones et le calcul de la première linéarisation. Les priorités locales sont toutes de gauche à droite.

EXERCICE 4.19. Ecrire en COMMON LISP la fonction `check-linerisation`, qui prend en paramètre une classe (ou un nom de classe) et recherche tous les quadruplets w, x, y, z tels que spécifiés dans l'exercice précédent. □

4.5.3 La linéarisation C3

Cette linéarisation a été proposée par [Barrett et al., 1996] pour respecter les 3 propriétés énoncées ci-dessus — extension linéaire, monotonie et ordre étendu de priorité —, d'où son nom. Les propriétés de respect de l'ordre local et de stabilité viennent en corollaire. Elle est aussi stable pour la subdivision d'arc et l'éclatement série. C'est aussi la linéarisation de PYTHON 3[13]. RUBY semble linéariser les *mixins* avec C3, car sa linéarisation donne les mêmes résultats sur les exemples critiques ci-dessus (Figure 4.12). Cependant, les spécifications ne sont pas explicites.

Fusion binaire. La définition qu'en donne [Ernst, 1999] ressemble à celle de l'algorithme 1b :

$$L(A) = A \centerdot L(B_1) \boxplus \cdots \boxplus L(B_n)$$

mais l'opération $\oplus$ de concaténation sans doublons est remplacée par l'opération de fusion $\boxplus$ qui est définie sur des listes non vides par :

$$(a_1, .., a_k) \boxplus (b_1, .., b_m) = \begin{cases} a_1 \centerdot (a_2, .., a_k) \boxplus (b_2, .., b_m) \text{ si } a_1 = b_1 \\ a_1 \centerdot (a_2, .., a_k) \boxplus (b_1, .., b_m) \text{ si } a_1 \notin \{b_1, .., b_m\} \\ b_1 \centerdot (a_1, .., a_k) \boxplus (b_2, .., b_m) \text{ si } a_1 \in \{b_1, .., b_m\} \wedge b_1 \notin \{a_1, .., a_k\} \\ \text{impossible si } a_1 \in \{b_2, .., b_m\} \wedge b_1 \in \{a_2, .., a_k\} \end{cases}$$

Mais cette fusion binaire n'est pas associative, et l'ordre de fusion influe sur le résultat dès qu'une classe a 3 super-classes directes, ou plus.

EXERCICE 4.20. Démontrer que l'opération $\boxplus$ retourne bien une extension linéaire si l'union de ses deux opérandes ne contient pas de circuit. Montrer qu'en présence de circuit, l'algorithme tombe sur le cas d'impossibilité. □

13. En version 2, pour les '*new-style classes*' seulement [Simionato] : pour les '*classic classes*', la linéarisation est un bête parcours en profondeur qui n'est même pas une extension linéaire.

EXERCICE 4.21. Montrer que l'opération $\boxplus$ n'est pas associative et qu'il est possible que $a \boxplus (b \boxplus c)$ soit calculable alors que $(a \boxplus b) \boxplus c$ ne l'est pas. Et réciproquement. □

EXERCICE 4.22. Définir la fonction LISP qui implémente $\boxplus$. Quelle est sa complexité ? □

EXERCICE 4.23. Améliorer cette définition en tenant compte du fait que, si le 3ième cas se présente, il n'est plus nécessaire de considérer le 2ième cas avant de rencontrer le 1ier cas. La complexité change-t-elle ? □

EXERCICE 4.24. Appliquer cette linéarisation aux exemples des figures 4.9, 4.12 et 4.13 et vérifier les propriétés des linéarisations obtenues. Le faire avec les deux ordres possibles de fusion. Comparer avec les linéarisations précédentes. □

EXERCICE 4.25. Dans l'exemple de la figure 4.12d, calculer les linéarisations de $K1$, $K2$ et $K3$ par cet algorithme. Qu'en déduisez-vous pour Z. □

Fusion n-aire. Pour résoudre le problème de l'associativité, il ne suffit pas de choisir un ordre pour appliquer la fusion binaire : en effet, le fait de faire une fusion rajoute des arcs de comparabilité qui peuvent entraîner des circuits lors de la fusion suivante. Il faut donc définir une opération de fusion n-aire $\biguplus$, qui donne un résultat tant que l'union des opérandes est sans circuit. La linéarisation se définit alors par :

$$L(A) = A \centerdot \biguplus_i L(B_i)$$

Soit p listes $L_i = (a_1^i, .., a_{k_i}^i)$ à fusionner, pour $i \in 1..p$. On définit $\biguplus_i L_i$ de la façon suivante : soit j le plus petit entier tel que a_1^j n'appartient à aucun des $(a_2^i, .., a_{k_i}^i)$ pour tout $i \neq j$. Alors,

$$\biguplus_i L_i = \begin{cases} a_1^j \centerdot \biguplus_i L_i' \text{ si } j \text{ existe, avec } L_i' = \begin{cases} L_i \text{ si } a_1^j \neq a_1^i \\ (a_2^i, .., a_{k_i}^i) \text{ si } a_1^j = a_1^i \end{cases} \\ \text{impossible sinon} \end{cases}$$

Noter que si j n'existe pas, il y a un ou plusieurs circuits, c'est-à-dire une chaîne $n_0, n_1, .., n_q = n_0$ avec $1 < q \leq k$, telle que, pour tout $0 \leq i < q$, $a_1^{n_i} \in \{a_2^{n_{i+1}}, .., a_{k_{n_{i+1}}}^{n_{i+1}}\}$. Inversement, s'il y a un tel circuit, on finira par arriver à une étape où j n'existe pas.

EXERCICE 4.26. Définir la fonction LISP qui implémente $\biguplus$. Quelle est sa complexité ? □

EXERCICE 4.27. Définir un algorithme efficace pour implémenter $\biguplus$. Tenir compte du fait que les $j-1$ premières linéarisations sont gelées tant que leur premier élément n'est pas enlevé. □

EXERCICE 4.28. Appliquer cette linéarisation aux exemples des figures 4.9a à 4.13 et vérifier la linéarisation obtenue. Le faire avec les deux ordres possibles de fusion. Comparer avec les linéarisations précédentes. □

EXERCICE 4.29. Montrer que cette linéarisation est stable. □

Fusion classique. La fusion $\boxplus$ (ou $\biguplus$) est une simple réécriture de l'algorithme de fusion, classique dans le cadre d'un ordre total. La fusion classique se définit ainsi :

$$(a_1, .., a_k) \boxplus (b_1, .., b_m) = \begin{cases} a_1 \centerdot (a_2, .., a_k) \boxplus (b_2, .., b_m) \text{ si } a_1 = b_1 \\ a_1 \centerdot (a_2, .., a_k) \boxplus (b_1, .., b_m) \text{ si } a_1 < b_1 \\ b_1 \centerdot (a_1, .., a_k) \boxplus (b_2, .., b_m) \text{ si } a_1 > b_1 \end{cases}$$

Dans notre cas, l'ordre n'est ni total, ni explicite, mais il est donné en intension sous la forme $a_1 < a_2 < .. < a_k$ et $b_1 < b_2 < .. < b_m$. On remplace donc la comparaison directe entre a_1 et b_1 par un test d'appartenance dans l'autre ensemble : $a_1 > b_1$ devient $a_1 \in \{b_2, .., b_m\}$. Si l'ordre est total, le résultat sera exactement celui d'une fusion classique, mais la complexité du calcul est plus élevée, puisque le test est en temps linéaire.

EXERCICE 4.30. Définir en LISP la fonction `merge` qui implémente la fusion classique de listes ordonnées d'entiers. Montrer qu'elle donne le même résultat que la fusion de C3 lorsque cette dernière est appliquée à des listes ordonnées d'entiers. □

Commentaires. La définition originale de [Barrett et al., 1996] repose sur la fusion n-aire mais la transcription binaire qu'en fait [Ernst, 1999] est incorrecte. Son schéma d'algorithme est d'ailleurs peu explicite sur l'origine des opérandes de $\boxplus$ (qui s'appelle & dans le papier). [Forman and Danforth, 1999] semble faire la même erreur. Traiter la fusion de façon binaire revient à s'imposer inutilement une hypothèse du monde ouvert, en ne voulant pas considérer les super-classes suivantes.

On trouve sur le Web des implémentations inutilement compliquées de C3 : la transcription directe de la définition de $\boxplus$ est à la fois plus simple et plus efficace.

4.5.4 La linéarisation de Dylan

La linéarisation de Dylan [Barrett et al., 1996] est un compromis entre celle de Clos et C3. Comme C3, elle calcule une extension linéaire de l'union des linéarisations des superclasses, à laquelle il faut ajouter l'ordre de priorité locale de la classe considérée. Mais elle calcule cette extension linéaire avec le même critère de choix du minimal que la linéarisation de Clos.

Les linéarisations de Clos et Dylan ne peuvent pas se décrire par un opérateur de fusion, même n-aire. En effet, comme pour C3, on pourrait considérer l'ensemble $J = \{j \mid a_1^j \notin (a_2^i, .., a_{k_i}^i), \forall i \neq j\}$. Si $J = \emptyset$, la linéarisation est impossible. Sinon, C3 choisit $\min(J)$. En revanche, la linéarisation de Dylan fait le même choix contextuel que la linéarisation de Clos : elle choisit $j \in J$ de telle sorte que a_1^j ait une sous-classe directe la plus récemment prise. Le choix ne dépend pas seulement des opérandes.

Remarque 4.4. Les linéarisations de Dylan et C3 diffèrent sur encore un point de la présentation qui en est faite ici : les auteurs considèrent que les arcs de transitivité peuvent être utilisés pour régler finement le résultat d'une linéarisation. Un argument supplémentaire doit donc être passé à l'opérateur de fusion : l'ordre de priorité locale de la classe considérée, c'est-à-dire $(B_1, .., B_n)$. C'est inutile s'il n'y a pas d'arcs de transitivité.

Par ailleurs, la première linéarisation monotone proposée dans [Ducournau et al., 1994] était basée sur le calcul d'une extension linéaire de l'union des linéarisations des super-classes, en lui appliquant la linéarisation de Loops et non celle de Clos. Elle est très proche de C3 mais C3 semble préférable.

4.5.5 Quelques suggestions pour la recherche

La monotonie est une contrainte relativement forte, et l'on pourrait chercher à assouplir les linéarisations monotones.

Linéarisation quasi-monotone

Une position envisageable consisterait à rechercher la monotonie mais sans l'imposer.

On reprend donc le principe de la fusion de C3, mais en modifiant la définition de $\boxplus$ de la façon suivante. Si, pour tout j, il existe i tel que a_1^j appartient à $(a_2^i, .., a_{k_i}^i)$, on définit alors j comme le plus petit entier tel qu'il existe k tel que a_k^j soit minimal au sens de $\prec$ parmi les éléments de $\cup_i L_i$. Et on prend alors le plus petit k dans L_j.

$$\boxplus_i L_i = a_k^j \,.\, \boxplus_i L_i' \text{ avec } L_i' = L_i \backslash \{a_k^j\}$$

et on continue l'algorithme C3.

Cette linéarisation a l'avantage de respecter la propriété principale d'extension linéaire, d'être stable pour la plupart des transformations, et d'être toujours calculable. En contrepartie, elle ne respecte les autres propriétés (monotonie, priorité locale et étendue) que si c'est possible, c'est-à-dire tant qu'une linéarisation monotone est possible.

Linéarisation partielle

Une alternative consisterait à n'ordonner que ce qui est nécessaire. Les linéarisations servent à ordonner les propriétés locales d'une même propriété globale pour les combiner ou résoudre les conflits. Cet objectif peut être sans objet s'il y a pas ou peu de conflits : seules quelques classes incomparables par $\prec$ ont besoin d'être ordonnées.

Aussi, une solution pour préserver autant que possible la monotonie serait de ne calculer qu'une linéarisation partielle dans laquelle les propriétés locales d'une même propriété globale seraient toutes totalement ordonnées, mais où les classes ne seraient pas forcément totalement ordonnées.

Une linéarisation partielle serait donc constituée d'un ensemble d'arcs d'incomparables en plus de $\prec$ et l'algorithme de fusion généraliserait celui de C3 en acceptant en entrée des linéarisations partielles, pour produire une autre linéarisation partielle. En fait, dans ce cas, le terme de linéarisation est impropre puisqu'il s'agit de faire des *extensions non-linéaires*.

Algorithme abstrait. En entrée, une liste ordonnée des linéarisations partielles des super-classes directes, $L(B_1), .., L(B_k)$, ainsi qu'un ensemble d'antichaînes de $\cup_i L(B_i)$, chacune totalement ordonnée, représentant les propriétés locales à ordonner pour la classe A considérée et dont l'union forme une relation Q.

En sortie, une linéarisation partielle de la classe A.

Chaque linéarisation partielle $L(B_i)$ est l'union $\prec_{B_i} \uplus R_{B_i}$, où le premier terme représente la restriction de la spécialisation sur les super-classes de B_i et le second représente les arcs entre incomparables caractérisant chaque linérisation partielle. Alors, la linéarisation $L(A) = \prec_A \uplus R_A$ est la réduction transitive de $(\cup_i R_{B_i}) \cup Q \cup \prec_A$ et $R_A = L(A) \setminus \prec_A$. On peut voir R_A comme une espèce de réduction transitive de $(\cup_i R_{B_i}) \cup Q$ où la transitivité implique aussi $\prec$ (si $x \prec y$ et $(y, z), (x, z) \in R_A$, alors on enlève (x, z) de R_A).

La mise en œuvre algorithmique est plus difficile.

*****(à développer...)*****

4.6 Alternative à l'héritage multiple : les *mixins*

Les langages en typage statique et pur héritage simple sont quasiment inexistants. Tous les langages sont soit en typage dynamique comme SMALLTALK, soit en sous-typage multiple comme JAVA : même ADA 2005 a adopté ce système de types.

Une autre alternative à l'héritage multiple est basée sur la notion de *mixin*. C'est une approche qui ressemble beaucoup à la célèbre publicité pour un apéritif non alcoolisé : ce n'est pas de l'héritage multiple mais c'est comme de l'héritage multiple.

Remarque 4.5. La suite de cette section est extraite du même article en anglais [Ducournau et al., 2007].

4.6.1 Mixins in SCALA

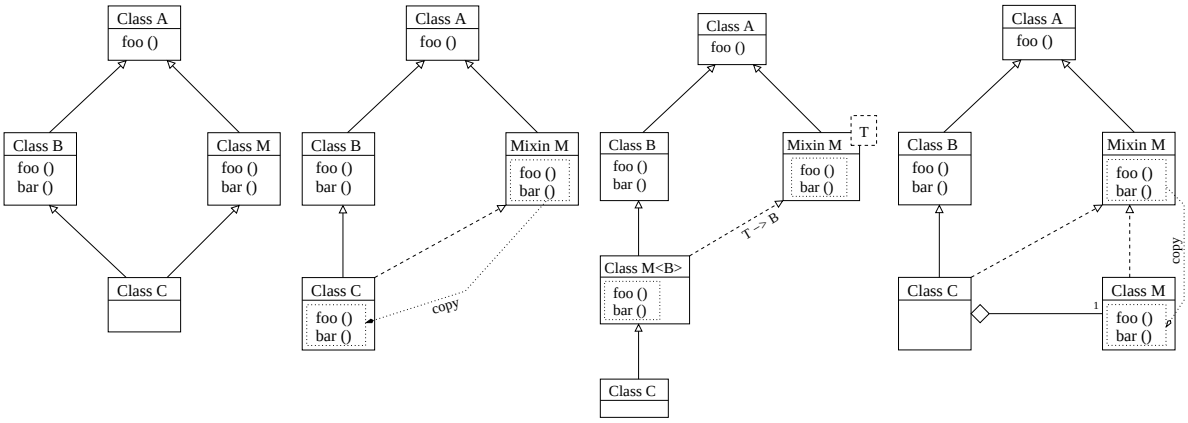
Several alternatives have been proposed, which rely on a degraded form of multiple inheritance, e.g. JAVA or C♯ interface multiple subtyping, *mixins* [Bracha and Cook, 1990] or *traits* [Ducasse et al., 2005]. *Interfaces* have been discussed about JAVA—they only imply slightly constraining the metamodel.

Mixins (aka *mixin classes*) are commonly presented as an alternative to full multiple inheritance. They first appear as a programming style in LISP-based languages, before becoming explicit patterns or even first class entities in theories or actual programming languages. Mixin proposals are numerous and variable—so this section must not be considered as a complete survey. Generally speaking, mixins are *abstract classes* with some restrictions in the way they are defined and related to other classes or mixins. Above all, a mixin is not self-sufficient—it must be used to qualify a class. To take up a distinction from linguistics, a class is *categorematic*, like a noun, whereas a *mixin* is *syncategorematic*, like an adjective [Lalande, 1926].

For instance, in the SCALA language [Odersky et al., 2004, 2008], one can define a class C such that C `extends` B `with` M, where B is the direct superclass of C and M is a *mixin*. An additional constraint is that the superclass of M must be a superclass of B. Actually, the constraint is a little bit more general—the single mixin M can be replaced by a set of $\prec$-related mixins such that all their superclasses[14] must be superclasses of B. Intuitively, the effect of this definition is to copy the definition of M into B (Fig. 4.14b). An alternative and more formal view involves *parametrized heir classes*, whereby C `extends` $M\langle B\rangle$ (Fig. 4.14c). This is the common way of using mixins with C++ *templates* [VanHilst and Notkin, 1996, Smaragdakis and Batory, 1998]. The *heterogeneous* implementation of templates makes it possible, but mixins cannot be separately compiled[15]. On the contrary, the *homogeneous* implementation of generics makes it impossible in JAVA. Finally, in a last view compatible with a *homogeneous* implementation, M is transformed into a *proxy*, which involves both an interface and a class : C implements the interface and is associated with the class by an aggregation, in such a way that each instance of C contains exactly one instance of M (Fig. 4.14d). The latter approach has also been called *automated delegation* [Viega et al., 1998]. Actually, the SCALA approach is a midterm between copy and proxy. M is compiled as an abstract class whose all methods are *static* and the corresponding redefined method in C calls this static method with an extra parameter, `this`.

Actually, mixins are not exempt from multiple inheritance conflicts—for instance, B and M may each introduce a property with the same name `bar`, or redefine the same property `foo` introduced in

14. Of course, here 'superclass' denotes a class, not a mixin.
15. See Chapter 7.



(a) Multiple inheritance (b) Copy view of mixins (c) Parametrized heir (d) Proxy

FIGURE 4.14 – Multiple Inheritance and Mixins—(a) in multiple inheritance, the example reproduces the two conflicting situations in Figures 4.3 and 4.5—`foo` stands for `area` and `bar` for `department`; (b) the same example involving a copy view of mixins; (c) mixins as parametrized heir classes; (d) mixins as proxies. Solid arrows denote class specialization (JAVA`extends`) and dashed arrows represent interface implementation (JAVA`implements`).

their common superclass A (Fig. 4.14). Hence, mixins are not incompatible with the present meta-model, which could be extended to include them, in the same manner as for JAVA interfaces. Global property conflicts are exactly the same and require the same solutions. Actually, most mixin-based languages, e.g. SCALA, do not recognize these conflicts and 'solve' them by *unification* (Section 4.2.1). MIXJAVA presents a notable exception, with a `view` keyword which behaves like full qualification [Flatt et al., 1998]. Regarding local property conflicts, there is no uniform policy among various mixin propositions, but all involve some explicit or implicit linearization. The previous definitions—by copy or parametrization—yield the same result and the mixin M overrides the direct superclass B. SCALA uses the COMMON LOOPS linearization in such a way that M—or all the mixin set—precedes the totally ordered superclasses.

However, mixins are also—sometimes above all—a specification of how things are implemented. The code of M is at least indirectly copied into C. Hence, mixins are compatible with a single-inheritance implementation, which is presumed to be more efficient than full multiple-inheritance implementations, or which is imposed by the target runtime system, e.g. in SCALA. However, in SCALA, the code is compiled into the JVM bytecode (or the .NET CRL) at the expense of translating each SCALA type into a JAVA interface. Therefore, the resulting implementation makes extensive use of method invocation on an interface-typed receiver—this is the so-called `invokeinterface` primitive, which is not renowned for its efficiency [Alpern et al., 2001, Ducournau, 2008]. See also [Ducournau, 2011] for a survey of object-oriented implementations.

Traits[16] [Ducasse et al., 2005] are a variant of mixins which provides a more formal way of combining them, with a finer grain. Traits are intermediate between SCALA mixins and JAVA interfaces—they can only define methods, contrary to mixins which can also define attributes[17], but these methods can have implementations, contrary to JAVA interfaces. Moreover, traits cannot have superclasses or even 'super-traits', but they can be explicitly combined to make composite traits. Different combination operators allow the programmer to precisely manage name conflicts, but at the local property level only—like their underlying languages, SMALLTALK and JAVA, traits do not recognize global property conflicts. In a static typing framework, traits are types, like usual mixins [Nierstrasz et al., 2006].

4.6.2 La linéarisation de SCALA

La linéarisation de SCALA est basée sur l'héritage simple des classes. L'idée générale de construction est de placer les mixins en tête de la linéarisation et les classes ensuite. Cependant, comme les superclasses de la classe considérée peuvent elles aussi utiliser des mixins, le principe exact est que la linéarisation de la super-classe soit un suffixe de la linéarisation de la sous-classe. L'algorithme est présenté comme suit dans [Odersky et al., 2008].

16. The term 'trait' is overloaded. Actually, SCALA uses 'trait' instead of 'mixin' but it merges different approaches, so we prefer to avoid confusions and call them mixins. 'Trait' was previously used in SELF [Ungar et al., 1991] with yet another, though similar, meaning. In C++, 'trait' denotes a programming pattern which allows the programmer to somewhat 'refine' primitive types, especially characters.

17. This limitation of traits may be caused by the target language, SMALLTALK.

Algorithme 7. Soit une classe C, sous-classe de B avec quelques mixins M_1, .., M_k.

Alors $L(C) = C.l(M_k) \oplus .. \oplus l(M_1) \oplus L(B)$, où $\oplus$ a le même sens que dans l'Algorithme 1b (page 52), et l calcule une certaine linéarisation (qui n'est pas précisée) de ses arguments. Dans $L(C)$, on retrouve donc entre C et $L(B)$ les M_i et leurs super-mixins qui ne sont pas utilisés par B.

Présenté ainsi, cet algorithme a l'air bizarrement familier. En effet, si $l = L$, il s'agit juste de l'Algorithme 1b avec l'ordre local de priorité $(M_k, .., M_1, B)$ pour C. Noter que le livre de référence sur Scala ne spécifie pas précisément l, mais il est probable qu'il s'agit bien de la même linéarisation.

Cette linéarisation est, par construction, monotone pour les classes, puisque $L(C)$ contient $L(B)$. Cela peut sembler contradictoire avec le contre-exemple (Figure 4.12a, page 54). Heureusement, il n'en est rien. D'une part, la linéarisation n'est pas monotone pour les traits, qui n'ont pas de linéarisations propres, et dont les super-mixins peuvent être ordonnés différemment suivant les classes qui les utilisent. D'autre part, le contre-exemple repose sur un ordre local de priorité implicite de gauche à droite. Appliqué à Scala, la classe est en dernier dans l'ordre local de priorité donc, sur l'exemple, F et E seraient des classes, D un mixin. Du coup, la non-monotonie (entre D et F) concerne un mixin.

Il reste la question de savoir si cette non-monotonie partielle est un problème ou pas. Lorsque le programmeur définit une classe, il peut compter sur la monotonie de la linéarisation pour les classes : c'est un élément primordial très positif. Cependant, lorsqu'il définit un trait comme une spécialisation de plusieurs autres traits, le programmeur ne maitrise plus rien. Les concepteurs du langage semblent considérer qu'il s'agit d'un mauvais style de programmation, les traits devant être utilisés de préférence à plat (comme avec ceux de [Ducasse et al., 2005]), et non dans des spécialisations complexes. L'objection peut bien entendu être considérée lorsque le trait se contente d'introduire de nouvelles propriétés. Mais lorsqu'une redéfinition avec combinaison est nécessaire, l'objection ne tient plus.

4.6.3 Exercices en Scala

Scala est un langage qui se compile dans la machine virtuelle Java (Jvm) et qui est donc complètement interopérable avec du code Java normal. On peut en particulier écrire en Scala du pur Java, modulo la syntaxe qui est assez différente.

Exercice 4.31. Implémenter l'exemple du losange (Figure 4.1, page 38) en Scala. Discuter des différentes possibilités. Examiner par introspection (Chapitre 11) les instances de ces classes pour en déduire la façon dont le langage est projeté sur Java. Vous pouvez aussi utiliser un décompilateur. □

Exercice 4.32. Faire de même pour l'exemple des polygones (Figure 4.13, page 56). □

Exercice 4.33. Examiner les linéarisations obtenues par Scala sur les exemples des figures 4.9a à 4.13. Attention, Scala inverse l'ordre local de priorité : les mixins sont avant les classes, et le (la ?) dernièr(e) mixin en premier. □

4.6.4 Autres langages avec *mixins*

On retrouve aussi des *mixins* dans beaucoup d'autres langages :

— Ruby (sous le nom de *module*) : dans ce langage, les mixins ne peuvent pas spécialiser des classes (contrairement à Scala), mais ils sont linéarisés par une linéarisation qui semble être C3 ;

— Fortress, un avatar moderne de Fortran, avec un système de types sophistiqué et un système d'exécution à la Java, conçu puis abandonné par Sun/Oracle ;

— Swift (sous le nom de *protocole*) : un langage très récent développé par Apple.

4.6.5 Conclusion sur *mixins* et traits

La critique des *mixins* et des traits peut ainsi se faire sur deux plans. Tels qu'ils sont réalisés en Scala, ils réduisent les problèmes de l'héritage multiple de façon quantitative (taille et fréqence), sans en traiter réellement aucun. Tels qu'ils sont spécifiés dans [Ducasse et al., 2005], ils représentent une spécification acceptable en typage dynamique, où les propriétés ne sont pas distingables par leur classe d'introduction. Il est donc possible de mettre 2 traits côte-à-côte, qui définissent chacun une méthode de même nom, sans qu'il y ait de doute sur l'identité de ces 2 méthodes, c'est-à-dire leur appartenance à la même propriété globale implicite. En revanche, en typage statique, cela n'est plus possible, et en tout cas pas souhaitable. Si une méthode est introduite par un trait, elle doit garder la trace de cette introduction et ne peut pas être confondue avec la méthode de même nom introduite par un autre trait non relié au premier.

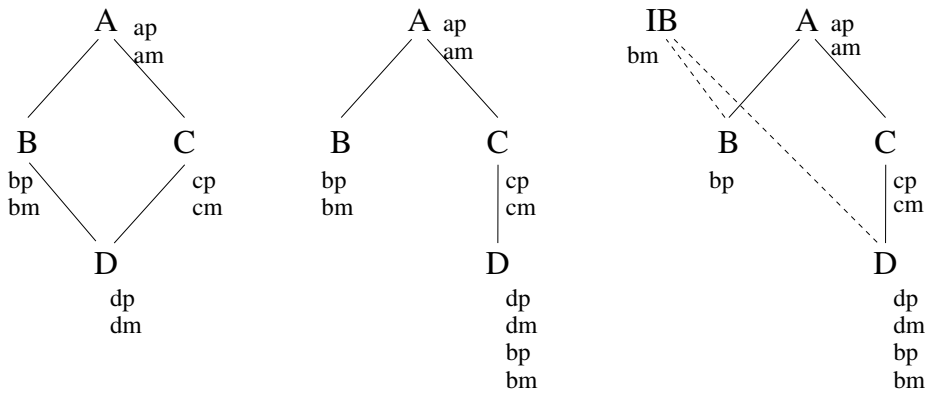


FIGURE 4.15 – Transformation d'héritage multiple à simple. Les `xp` représentent des attributs, et les `xm` des méthodes.

Les traits de SCALA ont pour principale justification de permettre de l'héritage multiple dans le contexte du système d'exécution de JAVA, basé sur un héritage simple de classes et le sous-typage multiple des interfaces. De ce fait, SCALA n'élude pas la "difficulté" des linéarisations, même s'il choisit un algorithme simple mais imparfait. En revanche, la justification des mixins dans un langage comme RUBY, en typage dynamique, est moins claire, dans la mesure où l'implémentation d'un tel langage est aussi difficile qu'avec de l'héritage multiple.

En RUBY et SCALA, il est très facile de faire de l'héritage multiple : il suffit de ne définir que des mixins, et pour chaque mixin de définir la classe correspondante, qui inclut le mixin correspondant, et rien d'autre. Dans le cas de SCALA, on utilise uniquement les mixins comme annotations de type.

C'est exactement le même principe que pour les interfaces, où on peut associer à chaque classe une interface qui déclare toutes les méthodes de la classe, en n'utilisant que les interfaces pour les annotations de type.

C'est encore la même chose que pour les classes abstraites avec le slogan : *"make all non-leaf classes abstract"* [Meyers, 1996, Steimann, 2000]. Il suffit, pour toute classe concrète et non finale, de la déclarer abstraite tout en définissant une unique sous-classe concrète et finale, qui ne déclare rien. Encore une fois, pour chaque paire abstraite/concrète, on utilise la classe abstraite pour les annotations de type, et la concrète/finale pour les instanciations.

Pour les classes abstraites, il n'y a pas de différence en terme d'efficacité. En revanche, pour les interfaces ou les mixins, en SCALA par exemple, le fait d'utiliser des interfaces comme annotations de type impose à la machine virtuelle JAVA de remplacer les habituelles opérations `invokevirtual`, qui sont assez efficaces, par des `invokeinterface` qui le sont beaucoup moins (cf. Chapitre 9).

En théorie, on pourrait donc faire des "surcouches" de RUBY ou SCALA en pur héritage multiple. En ce qui concerne SCALA, ce serait néanmoins avec une implémentation et une spécification dégradées par rapport à ce que l'on est capable de faire.

4.6.6 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'une passage d'un modèle d'analyse à un modèle d'implémentation. Plusieurs cas sont à considérer.

En sous-typage simple. La solution consiste à casser les cycles (non orientés !) 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 4.15, centre).

Lorsqu'un lien est coupé, les méthodes (`bm`) et attributs (`bp`) de la super-classe (B) doivent être recopiées 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.

En sous-typage multiple. En JAVA par exemple, la présence des interfaces permet de compenser une partie des défauts (Figure 4.15, 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 (ici `bm`). 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. Cela suppose bien sûr de bien encapsuler les attributs et d'utiliser des accesseurs pour `bp`.

Bien entendu, il faut toujours copier le code des méthodes de B dans D, à moins que l'on préfère adopter l'approche suivie pour l'implémentation de Scala en passant par des méthodes statiques.

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

4.7 En savoir plus

Malgré ses difficultés — qui ne sont pas insurmontables, loin de là ! — l'héritage multiple est une nécessité, au moins en typage statique. En témoigne le fait qu'en typage statique — si l'on exclut les langages à la Go qui ne veulent même pas entendre parler d'héritage —, il n'y a quasiment pas de langages qui ne proposent au minimum le *sous-typage multiple*, c'est-à-dire l'héritage multiple des interfaces, comme en Java. Tous les langages de la plate-forme .Net, ainsi qu'Ada 2005, se sont alignés. Seuls les langages en typage dynamique comme Smalltalk ou Objective-C résistent encore. Mais cela s'explique par le fait que leur système de types implicite est celui de Java (voir Chapitre 5).

L'essentiel de cette analyse de l'héritage multiple résulte de travaux menés au LIRMM dans les années 90, en particulier sur les linéarisations [Ducournau and Habib, 1987, 1991, Huchard et al., 1991, Ducournau et al., 1992, 1994, Huchard, 2000]. Ces travaux sur les linéarisations ont été repris et appliqués à d'autres langages, principalement Dylan, Python et Gbeta [Barrett et al., 1996, Ernst, 1999]. [Forman and Danforth, 1999, Annexe A] fait une présentation très pédagogique des linéarisations et de leurs propriétés. Cependant, l'essentiel de l'analyse et la distinction des deux catégories de conflits, faite dans [Ducournau et al., 1995], puis approfondie et formalisée dans [Ducournau et al., 2007, Ducournau and Privat, 2011], reste originale et inappliquée (à l'exception du langage Prm, Chapitre 10) : à l'exception récente de C♯ (et encore, avec une syntaxe perfectible), aucun langage grand public ne traite bien l'héritage multiple, en particulier les conflits de propriétés globales qui ne devraient pas poser de problèmes. [Ducournau and Privat, 2011] fait le point sur ces questions.

La notion de *mixin* vient des Flavors, un ancêtre de Clos, dont les concepteurs au MIT se réunissaient au glacier d'en-face et considéraient que les classes étaient des parfums (*flavors*) que l'héritage consistait à mélanger (*mix-in*) ! On ne se méfie jamais assez des métaphores. Les *mixins* sont donc une façon asymétrique de faire de l'héritage multiple qui se justifie mal. Bien entendu, une implémentation dans la Jvm impose une asymétrie. Mais cette implémentation est relativement inefficace — en Scala tous les types deviennent des interfaces dont l'implémentation est coûteuse (voir Chapitre 9) — et une implémentation directe en héritage multiple ne serait probablement pas moins efficace. Scala est un nouveau langage qui présente néanmoins l'intérêt d'introduire dans la programmation par objets des traits spécifiques de la programmation fonctionnelle comme les fonctions de première classe ou le filtrage.

Bibliographie

B. Alpern, A. Cocchi, S. Fink, and D. Grove. Efficient implementation of Java interfaces : Invokeinterface considered harmless. In *Proc. OOPSLA'01*, SIGPLAN Not. 36(10), pages 108–124. ACM, 2001. doi : 10.1145/504311.504291.

Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, Andrew L. M. Shalit, and P. Tucker Withington. A monotonic superclass linearization for Dylan. In *Proc. OOPSLA'96*, SIGPLAN Not. 31(10), pages 69–82. ACM, 1996.

G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA/ECOOP'90*, SIGPLAN Not. 25(10), pages 303–311. ACM, 1990.

L.G. DeMichiel and R.P. Gabriel. The Common Lisp Object System : An overview. In J. Bezivin, P. Cointe, J.-M. Hullot, and H. Liebermann, editors, *Proc. ECOOP'87*, LNCS 276, pages 201–220. Springer, 1987.

Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits : A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2) :331–388, 2005.

R. Ducournau. *Yet Another Frame-based Object-Oriented Language : YAFOOL Reference Manual.* Sema Group, Montrouge, France, 1991.

R. Ducournau. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 30(6) :1–56, 2008. doi : 10.1145/1391956.1391960.

R. Ducournau. Implementing statically typed object-oriented programming languages. *ACM Comp. Surv.*, 43(4), 2011. doi : 10.1145/1922649.1922655.

R. Ducournau. Petit Imprécis de LISP. Université Montpellier 2, polycopié de Master Informatique, 85 pages, 2013.

R. Ducournau and M. Habib. On some algorithms for multiple inheritance. In J. Bezivin, P. Cointe, J.-M. Hullot, and H. Liebermann, editors, *Proc. ECOOP'87*, LNCS 276, pages 243–252. Springer, 1987.

R. Ducournau and M. Habib. La multiplicité de l'héritage multiple. *Technique et Science Informatiques*, 8(1) :41–62, 1989.

R. Ducournau and M. Habib. Masking and conflicts, or to inherit is not to own. In M. Lenzerini, D. Nardi, and M. Simi, editors, *Inheritance Hierarchies in Knowledge Representation and Programming Langages*, chapter 14, pages 223–244. John Wiley & Sons, 1991.

R. Ducournau and J. Privat. Metamodeling semantics of multiple inheritance. *Science of Computer Programming*, 76(7) :555–586, 2011. doi : 10.1016/j.scico.2010.10.006.

R. Ducournau, M. Habib, M. Huchard, and M.-L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proc. OOPSLA'92*, SIGPLAN Not. 27(10), pages 16–24. ACM, 1992.

R. Ducournau, M. Habib, M. Huchard, and M.-L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proc. OOPSLA'94*, SIGPLAN Not. 29(10), pages 164–175. ACM, 1994.

R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, and A. Napoli. Le point sur l'héritage multiple. *Technique et Science Informatiques*, 14(3) :309–345, 1995.

R. Ducournau, F. Morandat, and J. Privat. Modules and class refinement : a metamodeling approach to object-oriented languages. Technical Report LIRMM-07021, Université Montpellier 2, 2007.

U. Eco. *La recherche de la langue parfaite*. Seuil, Paris, 1994.

Erik Ernst. Propagating class and method combination. In R. Guerraoui, editor, *Proc. ECOOP'99*, LNCS 1628, pages 67–91. Springer, 1999.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. POPL'98*, pages 171–183. ACM Press, 1998.

I. R. Forman and S. H. Danforth. *Putting Metaclasses to Work*. Addison-Wesley, 1999.

J.F. Horty. Some direct theories of nonmonotonic inheritance. In D. Gabbay and C. Hogger, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 2 : Nonmonotonic Reasoning*. Oxford University Press, 1994.

M. Huchard. Another problematic multiple inheritance mechanism : Construction and destruction in C++ in the virtual inheritance case. *J. Obj. Orient. Program.*, 13(4) :6–12, 2000.

M. Huchard, M.-L. Mugnier, M. Habib, and R. Ducournau. Towards a unique multiple inheritance linearization. In Augustin Mrazik, editor, *Proc. EurOOP'91*, sep. 1991.

ILOG. *Power Classes reference manual, Version 1.4*. ILOG, Gentilly, 1996.

D. E. Knuth. *The art of computer programming, Sorting and Searching*, volume 3. Addison-Wesley, 1973.

J. Kusmierek and V. Bono. Hygienic methods - introducing HygJava. In *TOOLS 2007*, volume 10 of *JOT*. ETH Zurich, 2007.

A. Lalande. *Vocabulaire technique et critique de la philosophie*. Presses Universitaires de France, 1926.

B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

S. Meyers. *More Effective C++*. Addison-Wesley, 1996.

Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. Flattening traits. *J. Object Technology*, 5 (4) :129–146, 2006.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nicolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, Lausanne, CH, 2004.

Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, A comprehensive step-by-step guide*. Artima, 2008.

A. Shalit. *The Dylan Reference Manual : The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, 1997.

M. Simionato. The Python 2.3 method resolution order. URL http://www.python.org/download/releases/2.3/mro/.

Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In E. Jul, editor, *Proc. ECOOP'98*, LNCS 1445, pages 550–570. Springer, 1998.

Friedrich Steimann. Abstract class hierarchies, factories, and stable designs. *Commun. ACM*, 43(4) : 109–111, 2000.

B. Stroustrup. *The C++ programming Language, 3^e ed.* Addison-Wesley, 1998.

D.S. Touretzky. *The Mathematics of Inheritance*. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1986.

D. Ungar, C. Chambers, B.W. Chang, and U. Hølzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3) :223–242, 1991.

Guido van Rossum and Fred L. Drake, Jr. *The Python Language Reference Manual*. Network Theory Ltd, September 2003. ISBN 0-9541617-8-5.

Michael VanHilst and David Notkin. Using role components to implement collaboration-based designs. In *Proc. OOPSLA'96*, SIGPLAN Not. 31(10), pages 359–369. ACM, 1996.

John Viega, Bill Tutt, and Reimer Behrends. Automated delegation is a viable alternative to multiple inheritance in class based languages. Technical Report CS-98-03, University of Virginia, Charlottesville, VA, USA, 1998.

Deuxième partie

A propos du typage et de la liaison tardive : du statique au dynamique, de la compilation à l'exécution

5

Typage statique et sous-typage

Le typage statique est une notion fondamentale des langages de programmation, dont l'approche objet a, si l'on peut dire, hérité. Cependant, la prise en compte du typage statique 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, les spécifications de nombreux langages sont telles que le typage peut entraîner des ambiguïtés (pour le compilateur) et des confusions (pour le programmeur) dans la sélection de méthodes.

5.1 Typage et sous-typage

Dans tous les langages, les valeurs ont un type, soit que ce type soit codé dans la chaîne de bits représentant la valeur, soit qu'il se déduise simplement de l'usage de cette valeur dans le programme. La question est de savoir comment un langage permet d'exprimer ces types et dans quelle mesure la machine d'exécution du langage garantit contre les erreurs de types à l'exécution.

5.1.1 Typage statique

Statique et dynamique Rappelons d'abord le sens des termes 'statique' et 'dynamique' dans le contexte des langages de programmation. Le premier qualifie tout ce qui peut s'effectuer à froid, à la compilation, en l'absence de valeur. Le second s'applique au contraire à tout ce qui se passe à l'exécution, à chaud, et qui repose sur l'existence des valeurs.

Le dipôle statique-dynamique s'applique, entre autres, aux types et au typage (c'est l'objet de ce chapitre), aux erreurs (Section 5.1.3), à des données ou à leur allocation (voir le mot-clé `static`), au chargement de classes ou à l'édition de liens, et enfin à la sélection de méthodes (Section 6.3).

On pourrait dire que 'statique' s'applique à la syntaxe et 'dynamique' à la sémantique si, dans le domaine de la compilation, les types n'étaient pas classés, peut-être improprement, dans la sémantique.

Typage statique et typage dynamique. Les langages de programmation se distinguent suivant que leur typage est statique ou dynamique.

Le typage statique se caractérise par le fait que les diverses entités du programme (variables, fonctions, etc.) sont explicitement annotées par des types, ces annotations signifiant que la valeur de l'expression doit appartenir au type en question. C'est le cas de C, C++, Java, Eiffel, Pascal, C♯, etc. Dans la suite nous adopterons la notation `x: T` pour ces annotations de type. Dans ce contexte, le type des valeurs n'a pas besoin d'être codé dans la valeur car chaque chaîne de bits peut être interprétée suivant le type statique de l'expression qui la désigne. Dans ce contexte, la notion de *typage fort* s'applique à des langages dans lesquels la correction des types est imposée : en gros, toute expression est annotée par un type et il n'est pas possible d'appliquer à une expression une opération qui serait inconnue de son type.

En typage dynamique, il n'y a en revanche aucune annotation de type dans les programmes : c'est le cas de Lisp, Scheme, Clos, Smalltalk, Python, Ruby et bien d'autres langages. L'absence d'annotation de type rend indispensable un codage du type dans la valeur et des vérifications dynamiques de type à l'exécution. 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 deux catégories de langages :

— les langages dont le prototype est ML— et ses variantes et extensions Caml et OCaml— qui ne disposent pas d'annotations de type mais dont les types statiques sont déduits par une inférence de types ;

— des langages comme Objective-C, dans lesquels les annotations de type sont possibles, mais dont le système (eg compilateur) n'impose aucune vérification des types[1].

Types abstraits et polymorphisme. Le *type abstrait* est une notion traditionnelle des langages de programmation, qui se définit par une *interface*, c'est-à-dire un ensemble de signatures de fonctions, donc quelque-chose de très proche d'une *intension*. La notion de *polymorphisme* en découle : un type abstrait peut avoir différentes implémentations, par exemple le type `Point` peut s'implémenter par un `PointCartésien` ou par un `PointPolaire`. Ce polymorphisme particulier est appelé *polymorphisme d'inclusion* et correspond au sous-typage : `PointCartésien` est un sous-type de `Point`.

Une autre forme de polymorphisme, le *polymorphisme paramétrique* sera examinée au chapitre 7. Les autres emplois du terme sont en général abusifs.

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. Un type abstrait se réduit ainsi à une pure 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.

Dans les théories des types, on distingue en général les *types nominaux*, qui sont des symboles, et les *types structuraux*, qui sont des ensembles (ou plutôt *record*) d'opérations nommées, c'est-à-dire des intensions. On peut donc voir une classe comme un type nominal, qui a un type structurel.

Si l'on fait coïncider les types et les classes, il est alors tout aussi naturel de chercher à 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, de la spécialisation et du sous-typage, 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.

Tous les langages que nous allons étudier pratiquent le *typage nominal*, en *identifiant* (au moins en première approximation) la spécialisation et le sous-typage. La seule exception, qui restera assez marginale dans ce cours, concerne le langage OCaml, qui pratique un *typage structurel* et *distingue* la spécialisation et le sous-typage. Le langage Go va encore plus loin, puisque ses types sont structurels, mais il n'a ni spécialisation, ni héritage, ce qui l'exclut du champ de ce cours.

5.1.2 Polymorphisme et objets

Dans les langages à typage statique, il faut distinguer
— le type *statique* des *expressions* du programme, qui concernent le compilateur et le programmeur, et qui résulte d'une annotation explicite (ou d'une inférence en ML) ;
— le type *dynamique* des *valeurs* que ces expressions prennent à l'exécution : en programmation par objets, ce type dynamique s'identifie avec la classe qui a instancié la valeur.

Avec le polymorphisme d'inclusion lié au sous-typage, le type dynamique des valeurs peut être un sous-type strict du type statique des expressions et le principe fondamental du typage impose que ce soit toujours un sous-type.

Axiome 5.1 (Polymorphisme d'inclusion) *Le type dynamique d'une valeur doit toujours être conforme à, ou sous-type de, son type statique.*

En pratique, cela signifie juste que l'intension du type dynamique doit contenir celle du type statique, comme pour la spécialisation de classes (Propriété 2.2.

Le cas du receveur

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), dont le type statique est toujours implicitement (celui de) la classe qui définit la méthode, que le typage soit statique ou dynamique. En revanche, à cause de la liaison tardive, le type dynamique du receveur, lui, peut être n'importe quelle sous-classe : `self` ne peut jamais être présumé *monomorphe*, à moins que la classe soit déclarée *finale* (par exemple avec le mot-clé `final` en Java, `sealed` en C♯ ou `frozen` en Eiffel) ce qui n'est pas recommandé car cela contrevient au Méta-Axiome 2.2, page 21.

1. Objective-C est un langage qui mélange de la programmation objet à la Smalltalk avec de la programmation C. Le code C est typé comme d'habitude, mais les inserts de code objet ne sont pas typés.

Comme `self` est statiquement typé, on pourrait supposer que le langage en profite pour vérifier sur tous les appels de méthodes sur `self` que la méthode existe. Curieusement, il n'en est rien en SMALLTALK, sans doute pour une raison de culture, mais surtout parce que le langage est complètement dynamique : le programmeur définit une méthode en cliquant dans un éditeur, il ne définit jamais une classe globalement. Si une méthode est inconnue, c'est peut-être la prochaine qui va être définie. On a un problème similaire en LISP lorsque l'on définit une fonction `foo` qui appelle une fonction `bar` : l'interprète doit bien patienter puisque la fonction `bar` va peut-être suivre, et il est même possible qu'elle appelle récursivement `foo`.

Le cas de `super` sera analysé en Section 6.7.1, page 105.

Finalement, dans le cas des *multi-méthodes* et de la sélection multiple (Section 6.2), par exemple en CLOS, il n'y a pas de receveur à proprement parler, et tous les paramètres ont un type statique explicite ou implicite.

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). Le passage de paramètre inclut bien entendu le cas du receveur.

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

Méta-axiome 5.1 (Principe objet) *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.

Il s'agit bien sûr d'un truisme dans la cas du typage dynamique.

5.1.3 Typage sûr

La notion de type est inséparable de la notion d'erreur de type qui consiste, en typage dynamique, à essayer d'appliquer à une valeur d'un certain type un opérateur que le type ne connaît pas. En typage statique, cela se réduit à enfreindre l'Axiome 5.1, page 70.

Effacement de type

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 ce commentaire est devenu obsolète (fatal destin de tout commentaire), le compilateur devrait s'en apercevoir. C'est aussi une contrainte, le typage dynamique étant plus approprié à la programmation rapide.

Pour le compilateur, le typage statique sert à vérifier la correction du programme et à générer le code approprié. Dans le code généré, les types statiques peuvent avoir totalement disparu. C'est ce que l'on appelle l'*effacement de type*.

Supposaons que l'on dispose de deux versions d'un même langage, l'une en typage statique, l'autre en typage dynamique, avec des systèmes d'exécution rigoureusement équivalents. De tout programme statiquement correct, on pourrait tirer, en effaçant les annotations de type, un programme en typage dynamique correct.

Typage statique sûr

Le rôle essentiel du typage statique est d'assurer, ou de permettre au compilateur d'assurer, l'absence d'*erreur de type* à l'exécution. C'est ce que l'on appelle le *typage sûr*. Il y a cependant deux interprétations de la notion de typage sûr. La plus courante interdit toute erreur de type à l'exécution mais une acception plus large [Cardelli, 2004] se contente d'interdire les erreurs de type qui ne sont pas

rattrapées. La différence est d'importance puisque les systèmes LISP par exemple, qui sont complètement dynamiques, ne sont absolument pas sûrs dans le premier sens mais complètement sûrs dans le second.

Nous adopterons la première définition mais considérerons aussi des systèmes de types qui ne sont sûrs qu'au sens de Cardelli, car leurs (rares) erreurs de type sont détectées à la compilation, ce qui permet de les protéger par un test dynamique.

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 [2] qu'une méthode.

Erreur 5.1 (Message inconnu à l'exécution) *L'erreur `message inconnu` survient à l'exécution quand le type dynamique du receveur (`x`) ne connaît pas la méthode `foo`.*

Erreur 5.2 (Mauvais type d'argument à l'exécution) *L'erreur `mauvais type d'argument` survient à l'exécution lorsque le type dynamique de l'argument `arg` n'est pas conforme au type du paramètre de `foo` dans le type dynamique de `x`.*

Pour garantir des exécutions sans erreurs de type dès la compilation, le typage sûr va se baser sur une interprétation *pessimiste* de ces erreurs de type, en remplaçant les types dynamiques par les types statiques, mais en imposant une *quantification universelle*. Pour typer un fragment de programme dans le cadre de l'hypothèse du monde ouvert (Méta-Axiome 2.2, page 21), il faut considérer toutes les exécutions possibles de tous les programmes corrects incluant ce fragment. En pratique, la quantification va se faire sur tous les types dynamiques potentiels, qui doivent toujours être un sous-type du type statique selon l'Axiome 5.1, page 70 :

Erreur 5.3 (Message inconnu à la compilation) *L'erreur `message inconnu` survient à la compilation quand le type* statique *du receveur (`x`) ne connaît pas la méthode `foo`.*

Noter que cette erreur empêche de considérer que le typage dynamique consiste simplement en un typage statique implicite de toutes les expressions par le type le plus général (`Any`, `Object`, etc.)

Erreur 5.4 (Mauvais type d'argument à la compilation) *L'erreur `mauvais type d'argument` survient à la compilation lorsque le type* statique *de l'argument `arg` n'est pas* conforme *au (c'est-à-dire sous-type du) type du paramètre de `foo` dans le type* statique *de `x`.*

Remarque 5.1. On peut assimiler à l'erreur « Mauvais type d'argument » la non concordance entre les nombres de paramètres et d'arguments. Sauf exception, on ne considère ici que des méthodes avec un unique paramètre, mais on se ramène au cas général dans le produit des types (Section 6.2.1, page 88).

Remarque 5.2. On peut aussi assimiler à l'erreur « Mauvais type d'argument » la non concordance entre le type d'un attribut et sa valeur — on appellera ça une *chimère*, comme dans la mythologie, puisque cela peut se traduire par des lions à tête d'homme ou des hommes à tête de taureau.

Sous-typage, substituabilité et spécialisation

Le sous-typage se définit alors à partir du typage sûr, en passant par la notion de *substituabilité*. C'est ce que l'on appelle souvent le principe de substitution de Liskov.

Définition 5.1 (Substituabilité) *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 à l'exécution.*

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

Pour qu'une sous-classe puisse être considérée comme un sous-type (on peut alors identifier $\prec$ 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.

2. Voir plus loin, Section 5.2.2, page 74, comment un attribut peut s'interpréter comme 2 méthodes.

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[3]. 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.

5.2 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[4] 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.

5.2.1 Redéfinition vs. 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).

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.

Dans le cas d'une redéfinition, cette description n'est que partielle et constitue une modification de la description dans la super-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.

En revanche, 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.

Le méta-modèle du chapitre 3 caractérise la différence entre redéfinition et surcharge statique par le fait que la première concerne une propriété globale unique, alors que la seconde concerne des propriétés globales distinctes.

5.2.2 Redéfinition sans surcharge statique

Nous aborderons plus en détail la surcharge statique en section 6.3, page 92 et nous intéressons ici à la redéfinition, dans un contexte où il n'y a pas de surcharge statique. Attention ! Nous nous plaçons ici dans un cadre abstrait dont nous verrons plus loin qu'il diffère des langages usuels comme Java et C++. Ne pas chercher à transposer directement.

Notations. — Dans la suite, on notera $m_A(u) : v$ une méthode de nom m, définie dans la classe A 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 6.2). 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.

Rapportée au méta-modèle, cette notation signifie que m est une propriété globale et que m_A est la propriété locale définie dans la classe A qui lui est associée.

Redéfinition covariante du type de retour. Soit une méthode $m_A() : t$ définie dans la classe A avec un type de retour t et redéfinie par $m_B() : u$ dans la classe B, sous-classe de A avec un type de retour u. Dans tout contexte contenant le fragment :

3. Rappel : un préordre est une relation *réflexive* et *transitive*. Un ordre partiel est un préordre *antisymétrique*.
4. Au sens du méta-modèle (Chapitre 3), c'est-à-dire des propriétés dont le nom ne correspond à aucune propriété des super-classes.

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	$m_A : t$	$m_B : u$	$t = u$

FIGURE 5.1 – Résumé des règles de redéfinition en typage sûr, pour 2 classes A et B, avec $B <: A$

```
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 pour que la dernière affectation soit légale (Erreur 5.4, page 72).

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

Remarque 5.3. En pratique, l'affectation polymorphe de l'exemple serait 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 `foo(new B)` où le corps de `foo (x : A)` contient l'exemple précédent (à l'exception du `new`).

Redéfinition contravariante du type des paramètres. Soit une méthode $m_A(t)$ définie dans la classe A avec un type de paramètre t et redéfinie par $m_B(u)$ dans la classe B, sous-classe de A avec un type de paramètre u. Dans tout contexte contenant le fragment :

```
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 B doit avoir le même nombre de paramètres que celle de A et accepter, comme la méthode de A, 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 $foo : t$ se ramène à celui d'une méthode :

- lorsqu'on accède en lecture à l'attribut, on peut le considérer comme une méthode sans paramètre $foo() : t$, dont le type de l'attribut est le type de retour : la redéfinition doit donc être covariante ;
- lorsqu'on accède en écriture (affectation) à l'attribut, on peut le considérer comme une méthode $foo(t)$ avec un paramètre, la valeur à affecter, dont le type est le type de l'attribut : la redéfinition doit donc 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*.

Cette analyse des attributs revient à identifier un attributs à ses *accesseurs* implicites. En corollaire, elle montre qu'il est possible et sûr de redéfinir une méthode sans paramètres par un attribut de même type : du point de vue des types, cela revient juste à introduire une méthode d'écriture. Le langage EIFFEL le permet.

Règle de contravariance

On résume cette analyse par la règle suivante, qui tire son nom du caractère contre-intuitif du comportement des paramètres :

Définition 5.2 (RÈGLE DE CONTRAVARIANCE) *Elle se définit par la redéfinition covariante des types de retour, contravariante des types de paramètres et invariante des types d'attributs.*

5.2.3 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*, une redéfinition *covariante* reste sûre (voir aussi Section 6.8.2).

Méthodes et types fonctionnels. Une méthode peut être considérée comme un attribut immutable dont le type est fonctionnel (on parle aussi de type *flèche*). Si l'on note $t_1 \to t_2$ 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 :

$$t_1 \to t_2 <: u_1 \to u_2 \text{ ssi } t_2 <: u_2 \ \ \& \ \ u_1 <: t_1 \tag{5.1}$$

Pour garantir la substituabilité, le sous-type doit accepter plus de valeurs en paramètre et retourner moins de valeurs. C'est sous cette forme que la règle de contravariance a été introduite par [Cardelli, 1988].

La méthode étant immutable, sa redéfinition est covariante, ce qui ramène bien aux conclusions du paragraphe 5.2.2.

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 dotée d'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 à peu près aussi difficile que celle d'un appel de méthode. Voir, en Section 6.11, quelques développements sur les aspects fonctionnels des objets.

Type d'un objet. Le type ou la classe d'un objet peut lui-même être considéré comme un attribut immutable, 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 ! Considérer le type d'un objet comme un attribut revient à introduire du *méta* (Chapitre 11).

5.2.4 Contradiction entre typage sûr et sémantique de spécialisation

On a atteint ici la principale zone de friction entre les deux origines de la programmation par objets.

Les conclusions qui précèdent sur les règles de redéfinition sont imposées par le sous-typage dans un cadre de typage sûr. Elles 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 pour les paramètres car elle est contraire à la règle d'inclusion des domaines (Propriété 2.3, page 22). 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 [5].

L'opposition réside dans le double 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 dans toutes les exécutions possibles de tous les programmes possibles.

— 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 !

Le problème est que les types et les domaines n'ont pas le même rôle mais qu'il n'est pas possible d'exprimer les domaines de façon statique autrement que par un type.

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, elles rentrent par la fenêtre.

5. Et la crise de la vache folle ne résulte ainsi que d'une banale erreur de type !

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.

D'autres langages privilégient l'expressivité de la représentation, quitte à imposer des contraintes pour réduire les risques : c'est le cas de EIFFEL, voir Section 5.3.4, et du SGBD O_2.

Impossible synthèse. Une synthèse parfaite est impossible mais un compromis est souhaitable. La question n'est pas symétrique. Par exemple, les puristes du typage réclameront sans doute un langage contravariant, sans trop se préoccuper de son expressivité, alors que ceux de la représentation voudraient bien un langage covariant, mais ils sont quand même conscients des problèmes de typage.

Les objectifs ne sont pas non plus symétriques : on ne développe pas une application pour le plaisir de la voir certifier sans erreur de types par le compilateur mais pour opérationnaliser une certaine modélisation du monde. Si cette modélisation comporte des erreurs de type — par exemple veiller à ne pas donner de farines animales à des vaches — l'introduction de ces erreurs de type dans les programmes n'est pas anormale.

Notre compromis consistera donc à privilégier l'expressivité — la covariance doit pouvoir être exprimée — en essayant d'obtenir le typage le plus sûr possible — les erreurs de type doivent rester localisées et rares. Dans tous les cas, le typage doit rester sûr au sens de Cardelli. Une réponse satisfaisante devra attendre le Chapitre 7.

On pourrait exprimer cette position par un nouvel axiome

Méta-axiome 5.2 (Spécialisation et sous-typage) *La spécialisation doit être du sous-typage, et le typage doit rester sûr (au sens strict, et sauf dérogation*[6]*).*

Nous examinerons comment les langages se comportent par rapport à cet axiome, les quelques infractions existantes à cet axiome, ainsi que des dérogations à la sûreté qui paraissent souhaitables.

5.3 Variations sur quelques langages

Comme sur la plupart des points qui font l'objet de ce cours, il n'y a pas deux langages qui se comportent exactement de la même manière.

5.3.1 En C++

Valeurs, références, passage de paramètres

C++ est avant tout une extension du langage C. A ce titre, ses spécifications très complexes incluent de nombreux traits qui sont éloignés, voire incompatibles, avec la programmation par objets.

La première difficulté réside dans la désignation des valeurs ou des objets. Dans les langages de programmation, il existe deux grands modes de désignation des valeurs ou objets : la valeur immédiate et l'adresse de la zone mémoire allouée pour l'objet.

Dans les langages "modernes" (LISP, SMALLTALK, JAVA, etc.),

— les valeurs immédiates sont utilisées pour les types primitifs (entier, flottant, booléen, etc.),

— tous les types construits sont désignés par leur *adresse* : on parle, en général, de *référence*.

Dans ce contexte, les valeurs immédiates ne sont jamais créées explicitement (par un équivalent de `new`), mais sont soit des *litéraux*, soit obtenues par calcul. En revanche, toutes les instances des types construits sont obtenues par allocation explicite (`new`), à part certains types, comme `string`, qui ont aussi des litéraux et peuvent s'obtenir par calcul.

Cependant, dans le langage C (et donc en C++), il est possible de traiter les objets construits comme des valeurs immédiates, avec une allocation implicite[7].

Il en résulte une complication extrême aussi bien dans les possibilités offertes aux programmeurs que dans la façon d'en parler, et le vocabulaire devient très délicat.

Exemple. Etant donné une classe `C`, `C x;` déclare en JAVA une variable de type `x` dont la valeur sera l'adresse d'une instance de `C` explicitement allouée par `new C();`. Dans le vocabulaire courant, on dira en général que `x` est une référence, ou pointe, vers (ou sur) un tel objet. Ce n'est pas une réelle structure de donnée puisque cette référence cache ce que l'on appelait *liaison* en LISP, dans le cours

6. Il n'y a pas de dérogation possible au sens faible du typage sûr qui veut que toute erreur de type à l'exécution soit rattrapée par une exception.

7. On peut aussi voir le problème en sens inverse : au lieu de considérer des valeurs, par exemple de type `int`, le langage considère la boîte implicite qui contient la valeur. Du coup, même pour les entiers et bien que ce soit des valeurs, il faut passer mentalement par des constructions comme le passage des paramètres par copie...

de compilation. La variable `x` n'a pas forcément d'existence en mémoire, et elle est plus dans l'espace du programme que dans l'espace des données. En revanche, l'objet lui-même est bien dans l'espace des données et il est, en général, alloué dans le *tas* (*heap*).

En C ou C++, le même code `C x;` déclare une valeur immédiate, qui n'est pas allouée explicitement dans le tas, mais implicitement à l'entrée de la procédure, et en général dans la pile. La mémoire sera libérée à la sortie de la procédure. Pour avoir le même comportement qu'en JAVA, il faut faire `C* x;`, et les opérateurs `&` et `*` permettent de passer d'un mode à l'autre [8].

La sémantique des deux mode de désignation est très différente. Avec une valeur immédiate, le polymorphisme est très réduit. Par ailleurs l'affectation se traduit par une copie. Dans le fragment `C x,y; x=y;`, le contenu de l'objet `y` est vidé dans l'objet `x`, alors que dans `C* x,y; x=y;`, la liaison de la variable `x` est modifiée pour pointer sur l'objet pointé par `y`. Dans le cas d'une affectation polymorphe, avec `C x; D y; x=y;`, où `D` <: `C`, le contenu de l'objet désigné par `y` sera tronqué pour rentrer dans l'objet désigné par `x`.

La dualité des désignations se traduit bien sûr dans le vocabulaire. Alors qu'en JAVA référence et pointeur sont en gros équivalents, en C et C++, référence s'emploie pour une valeur immédiate alors que pointeur s'emploie pour une adresse. En pratique, tout se passe comme si le pointeur était une structure de donnée dont la valeur était l'adresse de l'objet pointé, la variable désignant cette structure de donnée et non directement l'adresse de l'objet.

Le vocabulaire C s'accompagne des deux termes *référencement* et *déréférencement*, qui désignent les opérations `&` et `*` qui permettent d'obtenir un objet immédiat à partir de son adresse, ou l'inverse. Ces termes n'ont aucun sens en JAVA.

En ce qui concerne le passage de paramètres lui-même, voir ce qui en est dit Section 6.1.

Remarque 5.4. Les conventions ci-dessus permettent de programmer en C++ de la même façon qu'en JAVA. Cependant, programmer dans un langage à objets se fait rarement à partir de rien. On réutilise des classes de librairies existantes, soit en se servant de leur API, soit en les spécialisant. Dans les deux cas, on sera amené à composer avec d'autres styles de programmation en C++. Adopter ces conventions ne signifie donc pas qu'il n'est pas nécessaire de connaître autre chose du langage.

Programmation objet en C++

L'essence de la programmation objet est que l'objet (c'est-à-dire son type dynamique) détermine son propre comportement (Méta-Axiome 5.1, page 71). 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, une grande partie de ce qui précède et de ce qui suit ne s'applique qu'à un sous-ensemble de C++ car, par défaut, le langage ne fait pas ou mal de la programmation par objets :

- le mot-clef `virtual` doit être utilisé partout, aussi bien pour les méthodes que pour l'héritage (Chapitre 4) ;
- de plus l'héritage ne doit pas être `private` : comme c'est le défaut, il faut annoter explicitement toutes les superclasses par `public` ;
- le comportement des classes dépend de leur implémentation qui elle-même dépend de ce qu'elles contiennent : pour avoir un comportement objet « normal », il faut que toute classe comporte au moins une méthode `virtual` — à défaut, la classe est considérée comme une `struct` ;
- 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, avec un « `*` » systématique sur tous les types non primitifs ;
- l'appel de méthode ne peut plus se faire avec le « . » de JAVA, mais doit se faire avec « `->` » ;
- les conversions implicites sont un danger permanent (Section 5.4.2).

Respect de la règle de contravariance. Moyennant ces mises en garde, lorsque le langage est utilisé dans un « bon style objet », il se comporte de la façon suivante sur les différents points abordés jusqu'ici dans ce chapitre :

- les types des attributs et des paramètres de méthodes sont invariants par redéfinition ;
- le type de retour est covariant mais les concepteurs ont été très réticents et il peut exister encore des compilateurs pour lesquels il est invariant [9].

8. Notez que l'on peut écrire `C *x;` tout aussi bien, les deux étant équivalents mais ne disant pas la même chose.

9. Avec le temps, je pensais que cette dernière réserve était probablement devenue obsolète. Il n'en est rien, car il existe encore en 2016 un compilateur, sous Windows, qui refuse la covariance.

Sous-typage en C++

Tout bien considéré, il est possible de programmer en C++ en respectant l'axiome 5.2 de façon absolue. Cependant, au moins deux caractéristiques du langage font que lui-même ne respecte pas l'axiome et que les infractions sont trop grosses pour qu'on puisse lui accorder des dérogations :

— l'héritage n'est pas forcément `public` : si `B` hérite de `A` de façon privée, `B` est bien vu comme un sous-type de `A` dans le code de `B`, mais pas en dehors : on ne peut plus faire `A a = new B();`

— la spécification de la surcharge statique est telle que si la classe `A` déclare une méthode `foo(T)` et que sa sous-classe `B` déclare `foo(U)` sans redéfinir `foo(T)`, alors `foo(T)` n'est pas visible pour `B`. La substituabilité est toujours assurée sur les valeurs (`A a = new B()` est toujours possible) mais elle n'est plus assurée sur les types : on ne peut pas remplacer `A` par `B` impunément.

Dans les deux cas, cela pourra avoir un effet sur la généricité, car une instance générique `Bar`⟨`A`⟩ pourrait être valide sans que `Bar`⟨`B`⟩ le soit (voir Chapitre 7).

Il n'est pas exclu qu'il existe d'autres caractéristiques du langage qui soient problématiques vis-à-vis du sous-typage.

Gestion mémoire en C++

Comme annoncé dans l'introduction, il n'y a, par défaut, aucune gestion automatique de la mémoire (*garbage collector*) en C++. Lorsque le programmeur fait des allocations dans le *tas*, par `new`, il doit aussi prévoir leur récupération par des `delete`. Dans un cadre de véritable programmation par objets, il s'agit, pour le programmeur, d'une mission impossible. Deux types d'erreur sont en effet possibles :

— La non-récupération d'un objet inaccessible peut produire des fuites de mémoire qui garantissent le crash du programme au bout d'un certain temps.

— En sens inverse, récupérer un objet encore accessible se termine à brève échéance par une *segmentation fault*.

Les versions les plus récentes de C++ ont introduit des pointeurs avec compteurs de référence qui permettent de récupérer les objets qui ne sont plus référencés. Ces pointeurs résolvent la question de la gestion mémoire en l'absence de références cycliques. Pour les cycles, une deuxième catégorie de pointeurs a été spécifiée dans le langage. Ces extensions peuvent dispenser de l'usage explicite de `delete`, mais leur usage est obscur et entièrement à la charge du programmeur : si l'erreur du deuxième type est bien évitée, celle du premier type perdure.

Pour combler cette lacune de C++, et éviter la charge de la gestion humaine des pointeurs à compteur de référence, nous conseillons d'utiliser la bibliothèque de gestion mémoire de Boehm qui fournit un *garbage collector* robuste et relativement performant (http://www.hboehm.info/gc/). Le programme résultant sera sans doute moins performant qu'avec une gestion mémoire finement réglée à la main, mais il sera plus robuste et son développement et sa mise au point seront beaucoup plus simples et rapides.

5.3.2 En Java

Contrairement à C++, Java est un langage qui est presque purement objet. Les sections précédentes s'appliquent donc à son mode de désignation des valeurs, identique à celui de Lisp), par leurs adresses, sans avoir besoin d'en faire explicitement des pointeurs.

Type de paramètres et de retour. Comme pour C++, le type des paramètres est invariant : les redéfinir revient à faire de la *surcharge statique* (Section 6.3).

Jusqu'en Java 1.4 inclus, le type de retour était invariant : cette limitation sans aucune justification a été levée avec la 1.5.

Interfaces de Java. La notion d'*interface* de Java est un très bon exemple d'entité intermédiaire entre classe et type : une interface ne définit que des signatures de méthodes. 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 6.3.9 et 4.6.6.

5.3.3 En C♯

Ce langage est un compromis entre C++ et Java. Il reprend au premier le mot-clé `virtual` pour les méthodes, et la plupart des grands principes du second, dont le respect de la règle de contravariance avec invariance des paramètres, mais aussi celle du type de retour, comme en Java 1.4.

5.3.4 Redéfinition covariante en Eiffel

Eiffel autorise une pleine redéfinition covariante des types de retour, de paramètres et d'attributs.

Règle des catcalls. Cependant, le langage tient un discours ambigu sur le typage sûr. Pour pouvoir prétendre à la sûreté tout en adoptant une politique covariante, le langage s'appuie sur la règle des *catcalls*[10] *polymorphes* :

Définition 5.3 (Catcall) *Un* catcall *est un appel de méthode de la forme* `x:A; x.foo(arg);` *où* `foo` *a été redéfinie de façon covariante dans une sous-classe de A.*

Définition 5.4 (Appel polymorphe) *Un appel est* polymorphe *s'il ne peut pas être prouvé par le compilateur que le type dynamique du receveur est toujours identique à son type statique.*

Erreur 5.5 (Catcall polymorphe) *Un* catcall *polymorphe est interdit à la compilation.*

Si l'on rajoute `x=new A` dans le code de la Définition 5.3, l'appel n'est vraisemblablement plus polymorphe. Mais cette définition a l'inconvénient de reposer sur les capacités d'inférence du compilateur : l'information que `x=new A` peut être déductible d'un code d'une plus grande complexité, aussi bien d'un point de vue intra-procédural (plusieurs fils d'exécution impliquant tous `x=new A`) qu'extra-procédural, `new A` étant effectué par des méthodes appelées. L'erreur peut donc être *injuste*, au sens où le programmeur peut prouver que `x=new A`, alors que le compilateur en est incapable. A contrario, l'erreur peut être incompréhensible par le programmeur qui croit que `x=new A`, alors que ce n'est pas toujours le cas.

Remarque 5.5. Si un appel n'est pas polymorphe, il est *monomorphe* et peut être compilé par un appel statique (Chapitre 9). Cette notion de polymorphisme est locale et peut s'examiner sous l'hypothèse du monde ouvert, en compilation séparée. En revanche, la définition même des *catcalls* nécessite l'hypothèse du monde clos, donc une compilation globale : elle est parfaitement contraire au méta-axiome 2.2, page 21.

De plus, il apparaît que les implémentations d'Eiffel que nous avons pu tester, Smart Eiffel en 2006 et Eiffel Studio en 2007, n'implémentent pas cette règle. Les *catcalls* polymorphes sont acceptés, sans même qu'une vérification dynamique de type soit insérée : le système de types n'est plus sûr au sens de Cardelli et la création de chimères devient possible. Dans des versions plus récentes d'Eiffel Studio, la vérification de type est effectuée à l'exécution.

Types ancrés. Eiffel propose aussi une notion de *type ancré* qui permet de lier avec le mot-clé `like` les types de plusieurs propriétés de façon à les faire varier de façon covariante mais en parallèle. Cette notion sera examinée dans un cadre un peu différent avec les *types virtuels* (Section 7.9).

5.3.5 En OCaml

En OCaml, le typage est structurel et pas nécessairement déduit de la spécialisation. De plus, le sous-typage repose sur la contravariance du type de paramètre, et non pas sur l'invariance comme dans les langages considérés jusqu'ici.

Il est donc possible que

— `B` $<:$ `A`, alors que `A` et `B` n'ont aucun rapport de spécialisation ;

— `B` $\prec$ `A`, mais que `A` et `B` n'ont aucun rapport de sous-typage : c'est le cas en particulier si `B` redéfinit une méthode de `A` en changeant son type de paramètre, par exemple avec l'intention de faire de la surcharge statique ou de la redéfinition covariante, mais il ne s'agit en fait ni de l'un ni de l'autre.

5.4 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).

On peut distinguer quatre sortes de coercition, suivant la relation de sous-typage existant entre les types source et cible, et suivant que la zone mémoire est copiée ou réinterprétée.

10. Voir aussi la note 12, page 83.

5.4.1 Réinterprétation d'une zone mémoire

Ce premier cas est de trop bas niveau pour figurer honorablement dans la panoplie de la programmation objet, mais c'est une possibilité de C, qui se retrouve donc en C++.

Il s'agit d'une *réinterprétation* de la zone mémoire source, présumée d'un certain type source, dans un type cible qui n'a aucun rapport sinon de pouvoir recouvrir la même zone mémoire. Il n'y a aucune transformation, et il peut, éventuellement, y avoir une copie pour éviter de modifier l'original. On prendra par exemple les 64 bits d'un flottant pour les réinterpréter comme un entier. Les voies du programmeur sont impénétrables et on espère juste qu'il sait ce qu'il fait.

L'exemple le plus typique concerne l'arithétique de pointeurs : une adresse est réinterprétée comme un entier, pour pouvoir faire des additions. Un objet sera à son tour considéré comme un tableau d'entiers.

Le système d'exécution des langages est forcément basé sur des fonctionnalités de bas niveau qui reposent sur ce genre d'opérations, le *garbage collecter* par exemple.

Nous ne considérerons pas plus longtemps cette possibilité dans le cadre de la programmation objet.

5.4.2 Conversion entre types incomparables

C'est la coercition classique des langages de programmation. On veut par exemple convertir un entier en flottant (`2` donnera `2.0`) 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, un seul principe général s'applique : il s'agit d'une *copie* durant laquelle la donnée est transformée, mais l'originale n'est pas modifiée. A priori et si l'on exclut la coincidence d'un point fixe miraculeux, les représentations binaires de l'original et de la copie diffèrent.

Pour ce qui est de la conversion, il n'y a que des cas d'espèces, à traiter pour chaque paire de types source-cible.

Conversions implicites. Certains langages comme C++ pratiquent des conversions implicites qui sont très dangereuses en elles-mêmes. Couplées avec la surcharge statique elles peuvent rendre fous les meilleurs. En effet, en cas d'erreur de type, le compilateur essaie toutes les conversions possibles. De plus, en C++, un constructeur avec un unique paramètre sert de fonction de conversion implicite du type de paramètre à la classe considérée. Il est possible d'éviter cela en annotant le constructeur avec le mot-clé `explicit` [11].

Dans les tests, il est impératif d'éviter les types primitifs usuels (`int`, `float`, `string`) car les conversions implicites 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.

Traitement polymorphe des types primitifs et *autoboxing*. Un cas particulier de conversions résulte de l'intégration des types primitifs dans les classes du langage. Il est traité à la Section 8.1.

5.4.3 Coercition ascendante

La coercition « ascendante » — le type source est un sous-type du type cible — permet de prendre une instance de la sous-classe pour une instance de la super-classe. Mais comme c'est l'essence même du polymorphisme et de la spécialisation (Axiome 2.1, page 20), le procédé apparaît comme *ad hoc* et son sens peut paraître incertain. S'agit-il de considérer que l'objet devient une instance directe de la super-classe, ce qui permettrait d'empêcher d'utiliser une méthode m spécifique à la sous-classe B, pour la remplacer par la méthode m d'une super-classe A ? Le procédé peut-il ne s'appliquer qu'à la méthode m, l'objet receveur conservant son type B pour les méthodes appelées par m ? Ou peut-il aussi, mais pas forcément, s'appliquer récursivement à toutes les méthodes appelées par m, ce qui revient à modifier le type dynamique de l'objet. Ces doutes sont à eux seuls un problème.

En réalité, la coercition ascendante ne fait que changer le type statique par lequel on manipule un objet, mais ses type dynamique et comportement (Axiome 5.1, page 71) restent inchangés. C'est essentiellement ce que l'on fait lors d'une affectation, ou d'un passage de paramètre, polymorphe, c'est-à-dire lorsque le type statique de la source est un sous-type strict du type statique de la destination — c'est alors appelé une « coercition implicite » en C++. Le type statique apparaît ainsi comme un

11. Encore une fois, le langage brille par sa capacité à imposer un mot-clé explicite pour les usages "normaux", l'absence de mot-clé ouvrant la voie à des usages aberrants.

point de vue sur l'objet référencé, et la coercition ascendante n'est qu'un changement de point de vue sur cet objet.

Remarque 5.6. Les considérations qui précèdent s'appliquent à C++ lorsque les types statiques considérés sont des références (types avec '*'). Dans ce langage, on obtiendrait cependant une véritable conversion, lors d'une affectation polymorphe dont les types ne sont pas des références (types sans '*') : il s'agit alors d'une copie où tout ce qui « dépasse » est tronqué. Voir aussi la Section 5.3.1, page 76.

Coercition ascendante explicite en C++ et Java. La coercition ascendante ne constitue donc qu'une information donnée au compilateur et l'existence d'une construction syntaxique suppose qu'il y a un besoin. Ce besoin réside pour l'essentiel dans des mécanismes du langage qui ne sont pas strictement objets : la surcharge statique et ses ambiguïtés ainsi que, en C++ ou C♯, les « méthodes » non `virtual`, à liaison statique, qui font que le type dynamique n'est plus considéré. Dans ces langages, la résolution de conflits de propriétés globales relève aussi de la désambiguïsation de la surcharge statique.

Dans ces cas, en C++ il faut utiliser l'opérateur `static_cast` (la syntaxe parenthésée de C ne doit pas être utilisée). En Java, on utilise en revanche cette même syntaxe parenthésée, qui n'a pas exactement le même comportement qu'en C++. En Eiffel ou Prm, il n'y a pas de coercition ascendante.

Une justification supplémentaire d'un tel mécanisme est implémentatoire : en C++, avec l'héritage multiple (Chapitre 4), 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 [Ellis and Stroustrup, 1990, Lippman, 1996] (Chapitre 9). Sur le plan physique, la coercition ascendante correspond donc à un véritable mécanisme, dont le coût est non nul, même si faible. Mais les détails d'implémentation ne devraient jamais remonter au niveau conceptuel.

5.4.4 Coercition descendante

La coercition « descendante » — le type source est un super-type du type cible — 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, en particulier là où le sous-typage est interdit. *A priori*, la coercition descendante garantit l'absence de sûreté du typage et la présence d'un mécanisme de coercition est contradictoire avec la revendication d'un typage sûr.

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` (Chapitre 7). 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*).

L'implémentation de la coercition descendante consiste donc en un test de sous-typage, c'est-à-dire une vérification dynamique de type pour s'assurer qu'il n'y a pas d'erreur et 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 latérale. En cas d'héritage multiple, les types source et cible peuvent être incomparables : pour que le test réussisse, il faut juste que le type dynamique soit un sous-type des deux.

Coercition descendante en C++, Java et Eiffel. En C++, il est impératif d'utiliser l'opérateur `dynamic_cast` : `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 est implicitement surchargée pour servir d'équivalent aussi bien au `dynamic_cast` dans le cas descendant, qu'au `static_cast` dans le cas ascendant. Par ailleurs, l'absence de généricité en Java 1.4 (jusqu'à la version 1.4) a conduit à un usage assez immodéré de la coercition descendante : cette malheureuse habitude ne doit en aucun cas être conservée maintenant.

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.

Gestion des échecs. Le bon usage de la coercition descendante consistera à l'accompagner d'un traitement pour rattraper les cas où la prévision ne s'est pas vérifiée.

Suivant les langages, l'échec de la coercition se traduit de différentes façons :

— en Java, le *cast* (syntaxe parenthésée) signale une exception, mais l'opérateur `instanceof` est booléen ;

— en C++, `dynamic_cast` retourne `null` en cas d'échec si l'argument est un pointeur ; si c'est un objet immédiat, une exception est signalée ;
— en Eiffel, en cas d'échec de la tentative d'affectation, la valeur `null` est affectée ;
— en C♯, un cast peut se réaliser indifféremment avec la syntaxe parenthésée de Java ou avec le mot-clé `as` : l'échec signale une exception dans le premier cas, et on pourrait supposer, pour justifier la double syntaxe, que l'échec de `as` retourne `null` ;
enfin, le mot-clé `is` correspond à `instanceof` en Java.

Coercition descendante implicite

Clauses `catch` Dans les langages objets, les systèmes de gestion d'exception sont en général organisés autour d'une hiérarchie de classes d'exception. Le signalement d'une exception donne alors lieu à la création d'une instance d'une de ces classes, conjuguée avec un échappement (`throw`). Le rattrapage de l'échappement s'effectue en général par un `catch` dont chaque clause est étiquetée par un type d'exception. Le principe est d'activer la première clause dont l'étiquette est un super-type de la classe de l'exception. C'est exactement le même principe que `typecase`.

Covariance Lorsqu'un trait d'un langage n'est pas sûr (par exemple s'il est covariant), le compilateur est censé insérer une coercition descendante implicite pour vérifier le type de la valeur considérée. C'est ce qui permet de rendre le système sûr au sens de Cardelli [2004]. Java le fait bien pour les tableaux (Chapitre 7), mais les implémentations d'Eiffel que nous connaissons ne le font pas (voir Remarque 5.5).

Coercition et conversion implicite. En C++, une conversion implicite peut se confondre dangereusement avec une coercition descendante, lorsqu'une classe a un constructeur dont le type de l'unique paramètre est une super-classe. Noter qu'il serait relativement anormal, mais pas impossible, que le type du paramètre soit une sous-classe.

5.4.5 Test de sous-typage

La coercition permet de considérer une valeur sous un autre type statique. Un test de sous-typage, lui, ne peut que tester si la valeur considérée est d'un certain type. Si le test est positif, ce n'est pas suffisant, au moins en typage statique, pour pouvoir appliquer à la valeur des méthodes connues seulement par le type cible (par exemple `bar()`), à moins que le langage soit équipé de capacité minimale d'inférence de type.

```
x : A
if (x instanceof B) {
   x.bar() // OK ici x : B
   }
```

Dans la portée du succès de la condition, le compilateur peut considérer que la variable `x` est typée par `B`. Mais cela ne marcherait pas si l'on remplace `x` par une expression quelconque :

```
x : Stack<A>
if (x.pop() instanceof B) {
   x.pop.bar() // ERREUR ici x.pop() : A
   }
```

En Java. On peut compléter le cast avec syntaxe parenthésée par le mot-clé `instanceof` ou la méthode `isInstance` : la première prend un nom de classe en paramètre alors que la seconde prend une instance de la classe `Class` (Section 13.1.1). La seule différence par rapport au cast réside dans la gestion des échecs : si l'on veut appliquer une méthode propre au type cible, il faut faire aussi un *cast*, sans avoir besoin de rattraper l'exception, mais en espérant que le compilateur n'effectue pas le test deux fois.

En typage dynamique. La coercition de type n'a pas de sens en typage dynamique, mais le test de sous-typage peut rester nécessaire. Par exemple, en Clos, on peut utiliser les deux fonctions `typep` et `subtypep`.

5.5 Variance des exceptions, des contrats ou des 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.

5.5.1 Covariance des 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 les instances 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 !

Si l'on considère les exceptions signalées par une méthode, le raisonnement est similaire. Une méthode retourne une valeur ou sort anormalement en signalant une exception. L'ensemble des exceptions signalables doit donc varier de façon covariante — la spécialisation et le typage sûr sont ici d'accord. Si la méthode redéfinie signale des exceptions qui n'étaient pas prévues dans la super-classe, il faut les rattraper localement.

Enfin, les exceptions sont en général organisées en hiérarchies de types d'exception. La covariance s'applique à nouveau sans problème : le sous-type signalé dans la sous-classe est bien rattrapé par le super-type dans la super-classe.

5.5.2 Contravariance des 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' (Section 6.8). 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'.

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

Bien entendu, par la sémantique de la spécialisation, l'ensemble des classes qui accèderont effectivement à la propriété évoluera plutôt de façon covariante. La redéfinition covariante des types de paramètres 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 *catcall*s [12].

5.5.3 Contravariance des préconditions et covariance des postconditions

Certains langages comme Eiffel permettent de spécifier des *contrats* qui garantissent le bon usage d'une classe ou d'une méthode. Pour les méthodes, ces contrats sont exprimés, avec les mots-clés `require` et `ensure`, par des pré- et post-conditions qui sont des expressions booléennes.

On remarque que le type des paramètres et les droits d'accès statiques constituent des préconditions particulières, alors que le type de retour et les exceptions signalables sont des postconditions. Le raisonnement habituel concluerait donc que :

— le typage sûr impose la contravariance des préconditions et la covariance des postconditions ;
— une approche moins sûre fondée sur la spécialisation devrait demander la covariance des préconditions.

Curieusement, le langage Eiffel est ici incohérent car il demande la contravariance des préconditions.

Invariants de classes. Les contrats incluent en général une troisième catégorie d'assertions, les invariants de classe. En Eiffel, on utilise le mot-clé `invariant`. Ce sont des expressions booléennes qui s'imposent à toute instance de la classe, à tout moment.

En pratique, chaque méthode est censée être transactionnelle : l'invariant est présumé vérifié à l'entrée de la méthode et doit le rester à la sortie, où il peut-être vérifié. Un invariant de classe est

12. Où *cat* signifie *Changing Availability* (c'est-à-dire le droit d'accès) *or Type*.

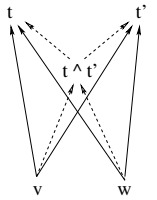


FIGURE 5.2 – Type intersection

donc une pré- et post-condition pour chacune de ses méthodes. Les invariants de classe s'héritent en se cumulant par conjonction booléenne. On peut donc les voir comme une méthode particulière de toute classe, dont la combinaison est aussi particulière.

5.6 Autres problèmes de typage

Les problèmes de typage ne se restreignent pas à la covariance. Certains tournent autour de l'expression du type du receveur, quand il est en paramètre (méthodes binaires) ou quand il est en type de retour (perte d'information). Ils seront examinés à la Section 7.9.3, page 141.

5.7 Typage statique et héritage multiple

Dans la configuration du losange (Figure 4.1, page 38) le conflit de propriétés locales peut aussi faire intervenir les types des paramètres ou de retour. Soit la propriété p de l'exemple, dont les deux méthodes en conflit sont $p_B : t \to u$ et $p_C : t' \to u'$.

Si la redéfinition dans D est requise, comme en C++ ou EIFFEL, le type de retour doit être à la fois un sous-type de u et u' et, en EIFFEL, le type du paramètre doit être un sous-type de t et de t'.

Types intersection En revanche, si la redéfinition n'est pas requise, par exemple dans un langage hypothétique avec linéarisation, dans l'envoi de message `x:D ; x.p(arg)`, le type « apparent » de p devrait être $t \cap t' \to u \cap u'$ en cas de covariance ou $t \cup t' \to u \cap u'$ en cas de contravariance.

Ces types union et intersection se définissent d'ailleurs de façon à retrouver la contrainte précédente :

$$x <: t \cap t' \iff (x <: t \wedge x <: t')$$
$$t \cup t' <: x \iff (t <: x \wedge t' <: x)$$

Le type intersection $t \cap t'$ peut donc être vu comme le super-type implicite de toute sous-classe commune à t et t' (Figure 5.2).

Existence d'une sous-classe commune. L'existence d'une sous-classe commune est cependant nécessaire. On pourrait considérer qu'il n'est pas possible d'imposer l'existence de cette sous-classe commune, lors de la compilation (Méta-axiome 2.2). La vérification pourrait avoir lieu à l'édition de liens — cependant, dans un cadre de chargement dynamique, comme en JAVA, l'éventuelle erreur deviendrait une erreur à l'exécution.

Cependant, en cas de covariance stricte du type de retour, on ne voit pas bien comment on pourrait maintenir le typage sûr sans redéfinir la méthode : en effet, la méthode sélectionnée par linéarisation retournerait sûrement un u (ou un u') mais il n'y a pas de raison qu'elle retourne sûrement un $u \cap u'$. L'existence d'une sous-classe commune à u et u' se poserait donc dès la compilation.

En cas d'invariance du type de retour, la sélection par linéarisation ne pose pas de problème de paramètres [13] si la redéfinition est covariante — de toute façon il faut prévoir une vérification dynamique de type.

Le problème de l'existence d'une sous-classe commune se reposera en Sections 6.2 et 7.10, dans des cadres légèrement différents.

13. Il y a une logique à cette asymétrie. Une méthode est chargée de retourner une valeur : la charge de la preuve d'existence lui revient donc. En revanche, elle reçoit des paramètres dont elle n'a pas à prouver l'existence.

5.8 Du typage statique au typage dynamique, et réciproquement

Il est très simple de passer d'un langage en typage statique à un langage à typage dynamique, à fonctionnalités équivalentes bien entendu. Si l'on prend un programme correct en typage statique, l'effacement de tous les types statiques donnera un programme approximativement correct en typage dynamique. Cela suppose juste d'écarter les ambiguïtés dont la désambiguïsation nécessite des types : surcharge statique ou conflits de propriétés globales.

En sens inverse, il n'est pas possible de faire aussi simple. Si l'on prend le cas de SMALLTALK ou de tous les langages en héritage simple et typage dynamique, on peut les typer dans une hiérarchie de classes en sous-typage multiple, à la JAVA, en rajoutant des interfaces pour introduire toutes les méthodes qui sont introduites par plusieurs classes en typage dynamique. Cependant, cela ne rend pas le programme correct pour autant car les types de paramètres ne sont pas connus et tout se passe comme si l'appel d'une méthode était toujours précédé d'une vérification dynamique de type pour s'assurer que le type dynamique connait bien la méthode appelée.

*****(à développer...)*****

5.9 En savoir plus

Le typage statique est intimement lié à la question de la sélection de méthodes et à la notion de généricité, qui font l'objet des deux chapitres suivants. L'analyse du typage et de la covariance qui a été commencée ici s'y poursuivra.

La question de la covariance a fait couler beaucoup d'encre. La covariance du type des paramètres a été activement défendue par les promoteurs d'EIFFEL, en particulier dans [Meyer, 1997] qui représente une réponse aux critiques faites par les apôtres du typage sûr.

L'analyse faite ici recoupe pour l'essentiel celle de [Shang, 1996]. Dans deux articles du début du millénaire [Ducournau, 2002b,a], je devais reprendre pour partie la même analyse, moins achevée, et le même exemple de la vache folle dont je dois rendre à son auteur les droits d'antériorité qui lui reviennent.

La covariance est un besoin de modélisation mais sa forme généralisée présentée dans ce chapitre la rend très peu sûre — elle trouvera une forme plus aboutie avec les *types virtuels*, au chapitre 7.

Bibliographie

L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76 :138–164, 1988.

L. Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, 2nd edition, 2004.

R. Ducournau. "Real World" as an argument for covariant specialization in programming and modeling. In J.-M. Bruel and Z. Bellahsène, editors, *Advances in Object-Oriented Information Systems, OOIS'02 Workshops Proc.*, LNCS 2426, pages 3–12. Springer, 2002a.

R. Ducournau. Spécialisation et sous-typage : thème et variations. *Revue des Sciences et Technologies de l'Information, TSI*, 21(10) :1305–1342, 2002b.

M.A. Ellis and B. Stroustrup. *The annotated C++ reference manual.* Addison-Wesley, Reading, MA, US, 1990.

S. B. Lippman. *Inside the C++ Object Model.* Addison-Wesley, New York, 1996.

B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, second edition, 1997.

David L. Shang. Are cows animals? http://www.visviva.3d-album.com/transframe/papers/covar.htm, 1996. URL http://www.visviva.3d-album.com/transframe/papers/covar.htm.

6

Liaison tardive ou envoi de message

Ce chapitre s'intéresse à tout ce qui concerne les appels de méthodes et analyse la *liaison tardive*, dans son principe et dans ses différentes variantes comme la *sélection multiple*, ainsi que des mécanismes supplémentaires comme la *combinaison de méthodes* ou parasites comme la *surcharge statique*.

Ce chapitre s'intéresse aussi bien aux langages à typage statique qu'à typage dynamique, même si certains points ne s'adressent qu'aux premiers.

6.1 Principe de la 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. 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 : ils ne sont pas censés influencer la sélection mais leur type est vérifié pour détecter l'erreur 5.4, page 72.

Le principe de l'envoi de message ou *liaison tardive* a été formalisé dans le chapitre 3 sur le méta-modèle.

L'appel de méthode a alors en général une syntaxe particulière qui isole le *receveur*, par exemple `x.foo(arg)` [1]. Dans ce cas, dans le corps de la méthode appelée, l'objet receveur est en général la valeur d'un paramètre implicite : `self` en SMALLTALK, `this` en C++, JAVA ou C♯ et `Current` en EIFFEL. Le type statique implicite de ce paramètre est toujours la classe qui définit la méthode.

Variantes. Le principe général de l'envoi de message est maintenant bien connu et il n'y aurait pas lieu d'y revenir de façon approfondie si les langages réels ne différaient pas de ce modèle abstrait de différentes façons :

- la *surcharge statique* permet d'avoir plusieurs méthodes (voire attributs) du même nom dans le contexte d'une seule classe : bien que cela ne change pas fondamentalement le modèle, la confusion entre la redéfinition et la surcharge complique la compréhension de la sélection de méthodes ;
- certains langages comme CLOS ne font pas la sélection de méthode sur la base du type dynamique d'un seul objet, le receveur, mais suivant le type dynamique de tous les arguments : c'est la *sélection multiple* ;
- la combinaison de méthodes est spécifiée de façon très variable et de nombreux langages ne correspondent que partiellement au méta-modèle ; en particulier l'appel à `super` peut ne pas s'effectuer à l'intérieur de la même *propriété globale* ; un langage comme C++ n'a d'ailleurs qu'un équivalent lointain de l'appel à `super`.

De plus, ces diverses variantes ne sont pas complètement indépendantes, car elles peuvent se recouvrir partiellement ou se combiner, augmentant ainsi les risques de confusion.

Passage de paramètres. Dans les langages "modernes" au sens de la Section 5.3.1, les appels de procédures se font comme dans les langages fonctionnels : les paramètres sont les entrées de la fonction et la valeur retournée est son unique sortie. Le passage des paramètres est dit "par valeur" et une affectation d'un paramètre dans le corps de la fonction ne modifie pas l'environnement d'appel [2]. C'est ce mode de passage de paramètres qui est supposé dans l'ensemble de ce cours.

1. Mais C++ remplacera le « `.` » par une « `->` ».
2. La fonction appelée peut néanmoins modifier les objets passés en paramètre, s'ils ne sont pas immutables.

Ce mode de passage de paramètres n'est pas universel. Il existe aussi un mode de passage de paramètre "par nom" (ou "par variable") qui permet de modifier la liaison d'une variable dans l'environnement de l'appel. Dans les langages de programmation comme ADA ou PASCAL, le programmeur peut indiquer pour chaque paramètre s'il est en entrée, en sortie, ou les deux. En C ou C++ cela se fait en utilisant l'un ou l'autre mode de désignation des valeurs (cf. Section 5.3.1). Par principe, le passage de paramètres par nom a une sémantique de copie : ce n'est pas l'objet immédiat lui-même qui est passé, mais une copie. Mais le passage de paramètres en C ou C++ est encore plus compliqué, puisque l'on peut passer l'adresse d'un objet immédiat (une référence) pour partager cet objet entre appelant et appelé, au lieu d'en faire une copie.

Le mode de passage de paramètres par nom est revendiqué pour des questions d'efficacité : il est ainsi possible de modifier une entrée d'un tableau depuis la fonction appelée, sans avoir besoin de passer le tableau en paramètre puis de faire une indirection. Cela parait plus efficace. En réalité, cela oblige le langage à permettre au programmeur de pointer au milieu des objets et cela rend l'éventuelle gestion automatique de la mémoire (*garbage collector*) beaucoup moins efficace car elle doit être *conservative* et supposer que chaque valeur entière peut être un pointeur à l'intérieur d'un objet. Et c'est bien entendu absolument contraire à la philosophie objet qui veut qu'un objet décide de son propre comportement.

Retour de fonction. Le principe des fonctions comme unique procédure a un seul défaut, c'est qu'il rend compliqué le retour par une fonction de plusieurs valeurs. Il existerait cependant une solution assez simple à cela, qui est représentée, dans un contexte de typage dynamique, par les *valeurs multiples* de COMMON LISP. Ces valeurs multiples sont en fait des tuples, que l'on peut typer statiquement avec les types produits (voir plus loin). Une fonction `foo` pourra avoir le type $\texttt{T} \to \texttt{U} \times \texttt{V}$. Elle retournera un `U` et un `V`. On pourra alors écrire :

```
1     x:T; y:U; z:V; (y,z)=foo(x);
2     x:T; (y:U,z:V)=foo(x);
3     x:T; y:U; y=foo(x);
3     x:T; z:V; (_,z)=foo(x);
5     y:U×V; y=foo(x);
6     y:U; z:V; bar(y,z);
7     x:T; bar(foo(x));
```

Le deuxième cas est une variante syntaxique du premier. Dans les troisième et quatrième cas, cela signifie que la fonction appelante n'utilise pas les deux valeurs. Noter que le détail important de ces valeurs multiples est que le tuple n'est pas un objet de première classe, sauf dans le cas 5. Le compilateur pourra décider de les allouer explicitement, ou plutôt de les passer dans la pile ou dans des registres.

On peut aussi imaginer passer un tuple en paramètre d'une fonction à la place de chacun des arguments de la fonction, comme dans le cas 6-7 où `bar` prend deux paramètres de type `U` et `V` mais peut les récupérer dans le tuple de type `U×V` retourné par `foo`.

6.2 Sélection multiple : une vraie liaison tardive sur plusieurs paramètres

6.2.1 Sélection multiple dans le produit des types

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 et types produit. Techniquement, la sélection multiple de CLOS se ramène à la sélection simple : il suffit de considérer le type produit qui peut être considéré comme un élément du produit cartésien des ensemble de types. 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 [3].

3. Cet artifice nous a permis au Chapitre 5 de ne considérer qu'un unique paramètre de méthodes.

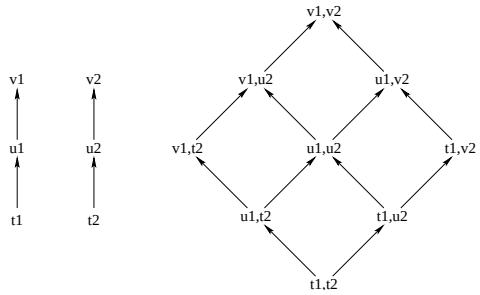


FIGURE 6.1 – Produit de deux hiérarchies de types.

On notera donc $m_{t_1 \times \cdots \times t_n}() : u$, 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 :

$$t_2 \times u_2 <: t_1 \times u_1 \iff t_2 <: t_1 \wedge u_2 <: u_1 \tag{6.1}$$

Ce passage aux types produit a un effet important : il entraîne 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 6.1.

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 :

$$t <: u \text{ et } \nexists v \text{ différent de } t \text{ et } u \text{ tel que } t <: v <: u, \text{ avec une méthode attachée à } v \tag{6.2}$$

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* (Section 4.2.2, page 40).

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 $t_1, t_2, \ldots, t_n$, le mécanisme va sélectionner une méthode attachée aux types $u_1, u_2, \ldots, u_n$, tels que

$$\forall i, t_i <: u_i \text{ et } \nexists v_1, \ldots, v_n \text{ tq } \forall i, t_i <: v_i <: u_i, \text{ et } \exists i : v_i \neq u_i, \text{ avec une méthode attachée à } v_1, \ldots, v_n \tag{6.3}$$

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 :

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

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'hérite pas forcément d'une unique méthode plus spécifique : dans la figure 6.1, on peut par exemple définir deux méthodes, respectivement sur les types $t_1 \times u_2$ et $u_1 \times t_2$.

Typage statique des paramètres. On a déjà constaté que le receveur courant était toujours typé statiquement, même dans un langage à typage dynamique. Ce constat s'étend à tous les paramètres dans le cas de la sélection multiple. Cependant, dans un langage comme CLOS, les règles du typage dynamique continuent à s'appliquer : il reste possible d'appeler une méthode sur un argument dont le type statique ne connaît pas la méthode. La raison en est double. Premièrement, certains paramètres ne participent pas à la sélection : ils ne sont donc pas typés explicitement autrement que par le type universel (la racine de la hiérarchie de type), ce qui n'empêche pas de s'en servir. Un autre raison est que CLOS est comme SMALLTALK (voir Section 5.1.2, page 70) un langage dynamique : la méthode inconnue va peut-être être définie juste après.

Linéarisation et sélection multiple

Le principe de linéarisation (Section 4.4, page 48) 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 6.1). 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 (`foo` $e_1 ... e_n$), 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
$(c_1^1, c_2^1, .., c_n^1)$, $(c_1^1, c_2^1, .., c_n^2)$, ..., $(c_1^1, c_2^1, .., c_n^{k_n})$, ..., $(c_1^{k_1}, c_2^{k_2}, .., c_n^{k_n})$.

6.2.2 Multi-méthodes encapsulées

La sélection multiple à la CLOS s'éloigne beaucoup du modèle objet standard dans la mesure où les fonctions génériques sont orthogonales aux classes (voir Chapitre 12). Un autre modèle est possible, dans lequel les méthodes restent encapsulées dans les classes mais présentent plusieurs « branches » qui sont sélectionnées dynamiquement suivant le type des autres paramètres. On appellera cela des *multi-méthodes encapsulées* [Castagna, 1997].

Spécifications. On se place dans un langage en typage statique sûr à la JAVA, dont on modifie les spécifications de la façon suivante :

- la surcharge statique n'appartient plus au langage ;
- dans une classe, on peut définir plusieurs méthodes de même nom, avec des paramètres de même nombre, mais de types différents et des types de retour différents ; on appellera ces différentes méthodes des *branches* et l'ensemble des méthodes de même nom dans une classe, une *multi-méthode* ;
- la syntaxe est la syntaxe implicite de la surcharge statique : les branches diffèrent par le type de leurs paramètres ;
- la sémantique se définit par la façon dont se fait l'appel de méthode `x.foo(arg)` — on se restreint pour commencer au cas où il n'y a qu'un seul paramètre :
  1. sélection de la multi-méthode `foo` suit le principe habituel (la plus spécifique dans les super-classes du type dynamique du receveur ;
  2. dans cette multi-méthode, sélection de la branche la plus spécifique suivant le type dynamique de `arg`.
  3. **si aucune branche ne convient dans la multi-méthode, il n'y a pas de recherche dans les super-classes mais une exception est signalée.** C'est la multi-méthode de la sous-classe qui masque globalement celle de la super-classe et non pas chaque branche de la sous-classe qui masquerait la branche correspondante dans la multi-méthode de la super-classe [4].

EXERCICE 6.1. Etendre le méta-modèle des classes et des propriétés du Chapitre 3 pour y inclure les multi-méthodes et les branches. En faire le diagramme de classes UML. □

Règle de sous-typage. On reste d'abord dans le cas où les multi-méthodes n'ont qu'un paramètre en plus du receveur : on généralisera simplement la notation en prenant le type du paramètre dans le produit des types. On propose la notation suivante : $m_{A,\{t_1 \to u_1, .., t_n \to u_n\}}$ est une *multi-méthode* de nom m définie dans la classe A avec des *branches* $m_{A,t_i \to u_i}$ de type de paramètre t_i et de type de retour u_i. Le type de la multi-méthode est $\{t_1 \to u_1, .., t_n \to u_n\}$.

La première question qui se pose est celle du sous-typage : comment généraliser la *règle de contravariance* (page 74) aux multi-méthodes ? Si m est redéfinie dans $B \prec A$, quelles sont les contraintes à imposer sur m_B de type $\{t'_1 \to u'_1, .., t'_p \to u'_p\}$ pour que la substituabilité soit assurée, c'est-à-dire pour que $\{t'_1 \to u'_1, .., t'_p \to u'_p\} <: \{t_1 \to u_1, .., t_n \to u_n\}$?

La règle s'exprime comme suit : pour toute branche de m_A, il doit exister une branche de m_B qui puisse accepter toutes les valeurs de paramètre qu'accepte la branche de m_A. Donc

$$\forall i \in 1..n, \exists j \in 1..p \text{ t.q. } t_i <: t'_j \text{ et alors, pout tout tel } j, u'_j <: u_i \tag{6.4}$$

4. Comme il n'y a pas forcément une branche dont le type de paramètre est un super-type de celui de toutes les autres branches, on peut envisager de définir, dans les redéfinitions uniquement, une branche `otherwise` qui rattraperait tous les types de paramètres non capturés par les branches explicites mais déjà introduits dans la super-classe.

Ambiguïtés.

Dès qu'il y a plusieurs paramètres ou de l'héritage multiple, les mêmes ambiguïtés qu'en surcharge statique peuvent apparaître (Section 6.3.2). Cependant, ces ambiguïtés reposent ici sur les types dynamiques des arguments et non pas sur leurs types statiques. Il s'agit donc de *conflits de propriétés locales*.

Cas 1. Lors d'un appel `x.m(a`^{1}`,..,a`^{k}`)`, d'une multi-méthode m_A ayant n branches avec des types variés pour ses k paramètres, il doit exister dans m_A une *unique* branche *plus spécifique* de type $\{t_i^1 \times .. \times t_i^k \to u_i\}$, la spécificité étant définie par $<:$ sur le produit des types. Mais le produit des types conduit à des risques de conflits d'héritage multiple. Par exemple, le type de m_A est $\{t \times t' \to u, t' \times t \to u\}$, avec $t' <: t$. Les deux branches sont tout aussi spécifiques lorsque les 2 arguments sont de type t'.

Cas 2. De façon similaire, si $n = 2$ et $k = 1$ et que la multi-méthode a le type $t_1 \to u_1, t_2 \to u_2$, où t_1 et t_2 ne sont pas comparables par $<:$ mais ont un sous-type commun, t'', qui est alors sous-type de $t_1 \cap t_2$. L'appel est ambigu si, à l'exécution, le type dynamique de l'argument est un sous-type de t'' : les deux branches sont aussi spécifiques l'une que l'autre.

Cas 1 : erreur à la compilation. Ces deux catégories d'ambiguïtés n'ont pas la même portée. La première provoque une erreur *modulaire*, qui peut être diagnostiquée sous l'*hypothèse du monde ouvert*. On (le compilateur) constate, statiquement, que les branches de la multi-méthode peuvent induire une ambiguïté : le programmeur est forcé de définir une branche supplémentaire pour lever l'ambiguïté. On impose alors aussi que si la multi-méthode a deux branches $t_1 \to u_1$ et $t_2 \to u_2$, telles que $t_2 <: t_1$, alors $u_2 <: u_1$. Une multi-méthode est *bien formée* si elle assure ces deux propriétés.

Cas 2 : erreur à l'édition de liens. En revanche, dans la seconde ambiguïté, l'existence de t'' ne peut pas être connue sans l'*hypothèse du monde clos*, donc une compilation globale. En cas d'héritage (ou de sous-typage) multiple, les multi-méthodes encapsulées avec plusieurs paramètres ne sont donc pas sûres. Comme dans le cas de l'héritage multiple (Section 5.7), cela peut donner lieu à une erreur de type à l'édition de liens. Mais l'erreur est inverse de celle causée par les types intersection : ici, c'est la présence d'un sous-type commun qui constitue une erreur.

Alternative. Il y a au moins deux façons de permettre d'éviter ces erreurs non modulaires. D'une part, la sélection peut abandonner la règle de l'unique plus spécifique et se baser sur des *linéarisations*, comme pour l'héritage multiple. D'autre part, le système de types peut inclure explicitement des *types intersection*. La programmeur peut alors prévoir une branche pour le type $t_1 \cap t_2$, même s'il ne connaît pas encore de classe qui implémente ce type. Si la branche n'est pas spécifiée, le cas est implicitement réputé impossible.

6.2.3 Le compromis de Fortress

Le langage Fortress est un langage récent qui propose des *multi-méthodes* dans un contexte de typage statique, en combinant élégamment les multi-méthodes encapsulées de Castagna, avec des multi-méthodes non-encapsulées, à la Clos, pour résoudre le problème de sélection unique en monde ouvert que pose la règle de Castagna.

*****(à développer...)*****

6.2.4 Redéfinition covariante et sélection multiple.

La sélection multiple peut être vue comme un moyen de contourner le problème de contravariance du type des paramètres en faisant disparaître ces derniers. Elle ne résout pas pour autant la contradiction entre la spécialisation et le sous-typage. D'une part, elle est sans effet sur les attributs. 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`. Il faut donc définir deux branches pour la classe `phoque` : l'une typée par `phoque` et `nourriture` signale une exception, et l'autre, typée par `phoque` et `poisson`, implémente le comportement souhaité. Cependant, si l'on souhaite réutiliser dans `phoque` le comportement général des `animaux`, par exemple par un appel à `super` ou son équivalent (voir Section 6.7), on risque d'avoir des problèmes, comme le montre l'exercice ci-dessous.

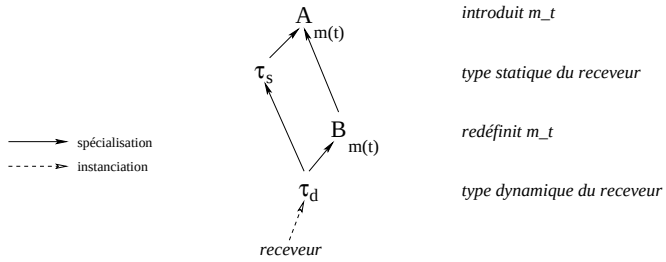


FIGURE 6.2 – Surcharge statique et dynamique : les 4 classes impliquées dans la sélection.

EXERCICE 6.2. Implémenter cet exemple en CLOS (Chapitre 12) et montrer que l'usage de `call-next-method` ne permet pas d'obtenir le comportement souhaité. □

Méthodes binaires. Ce sont des méthodes dont le paramètre a pour type la classe courante. L'exemple typique consiste en méthodes de comparaison (égalité ou inégalité). La sélection multiple est la seule façon de mettre en œuvre ce genre de méthode en évitant toute erreur de type. Le cas le plus classique est celui de la méthode d'égalité, qui met en jeu d'autres problèmes et est traité à la Section 8.2 dans le cadre du typage statique, et Section 12.2.8 dans le cadre de CLOS.

6.2.5 Exemple : les suites de nombres

*****(à développer...)*****

6.3 Surcharge statique : une fausse sélection multiple.

La surcharge statique est caractérisée par
— la redéfinition invariante du type des paramètres ;
— la possibilité de définir ou d'hériter, dans une classe, plusieurs méthodes de même nom mais de types de paramètres différents.

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.

6.3.1 Surcharge statique correcte (JAVA 1.5).

Au chapitre 3, une analyse « conceptuelle » de la spécialisation et de l'héritage nous a conduit à considérer que le nom des propriétés, telles qu'elles sont définies, héritées et utilisées, désigne une « propriété globale » sous-jacente, invariante par héritage et redéfinition. C'est le point de vue illustré par CLOS avec la distinction entre *fonction générique* et *méthode*.

Dans cette vision conceptuelle des objets, la surcharge statique est un phénomène d'ambiguïté du nommage des propriétés globales, alors que l'envoi de message et l'héritage sont des phénomènes relatifs à la même propriété globale, 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[5]. Ce nouveau nommage étant non ambigu (injectif), il peut alors servir à identifier les propriétés globales. 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 propriétés de nom m du type *statique* τ_s du receveur ;
2. sélection, dans cet ensemble, de la propriété globale m_t *la plus spécifique* adaptée au type *statique* des arguments et introduite dans la classe A ;

5. Les compilateurs utilisent effectivement ce renommage systématique : ce n'est qu'un cas particulier de ce que le jargon appelle *name mangling*, pour désigner la construction automatique de noms non ambigus.

3. sélection, à l'exécution et dans la propriété globale m_t, de la méthode m_t_B *la plus spécifique* correspondant au type *dynamique* τ_d du receveur.

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

$$\tau_d <: B <: \tau_s <: A \tag{6.5}$$

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 la propriété globale (Figure 6.2).

Surcharge statique et `super` Voir Section 6.7.1

6.3.2 Ambiguïtés 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 globale *la plus spécifique* compatible avec le type des paramètres. Mais il n'y a pas forcément d'unique plus spécifique et il peut se présenter les mêmes cas d'ambiguïtés que pour la sélection multiple (Section 6.2.2). La différence est que ces ambiguïtés reposent ici uniquement sur les types statiques — il s'agit donc de *conflits de propriétés locales* — ce qui offre toute lattitude au programmeur pour les lever.

Ambiguïtés liées à l'héritage multiple. En cas d'héritage multiple, ce choix de la *plus spécifique* peut conduire à des conflits similaires à ceux de l'héritage multiple (Chapitre 4). Ainsi par exemple, pour un appel de méthode $x : A; y : v; x.m(y);$, si les méthodes $m_A(t)$ et $m_A(u)$ sont définies et que v est un sous-type de t et de u.

Ambiguïtés avec plusieurs paramètres. Mais la surcharge statique conduit à des ambiguïtés spécifiques dans le cas où il y a plusieurs arguments : si $u <: t$, $m_A(t, u)$ 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 (Section 6.2). Devant de telles ambiguïtés statiques, le compilateur ne peut que générer une erreur.

Surcharge statique en Java 1.4, C♯ et Scala Curieusement, Java 1.4 signalait une ambiguïté dans un cas similaire mais pourtant assez différent, puisqu'il concerne le receveur et le paramètre secondaire : il considérait 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 $u <: t$. 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 6.4.

Le comportement de Java 1.4 était précisé dans les spécifications du langage [Gosling et al., 1996]. 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`) [6],
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).

Scala reproduit la même anomalie. En revanche, en C♯, cette situation ne provoque pas d'erreur mais la définition du plus spécifique est différente : c'est la méthode $m_B(t)$ qui est maintenant sélectionnée, comme en C++. Cependant, la sélection de C♯ n'est pas basée sur la même logique de masquage de nom.

On peut expliquer ces différents comportements en considérant différents ordres induits sur les signatures :

— en Java 1.5 (et versions suivantes), c'est l'ordre sur le produit des types de paramètres, receveur non compris ;
— en Java 1.4 et Scala, c'est l'ordre sur le produit des types de des paramètres, en incluant le receveur (dont le type statique est la classe de définition) ;
— en C♯, c'est l'ordre lexicographique sur les types de définition et le produit des types de paramètres ;

6. Le fait que la protection intervienne dans la sélection est en soi aberrant.

	t	$t[u]$	u		
A	$m_A(t)$	$m_A(t)$	$m_A(t)$	surcharge	
$A[B]$	$m_A(t)$	$m_A(t)$	$m_A(t)$	statique	C++
B	$+-$	$+-$	$m_B(u)$	incorrecte	

A	$m_A(t)$	$m_A(t)$	$m_A(t)$	surcharge	
$A[B]$	$m_A(t)$	$m_A(t)$	$m_A(t)$	statique	Java, C♯, Scala, ...
B	$m_A(t)$	$m_A(t)$	$m_B(u)$	canonique	

A	$m_A(t)$	$m_A(t)$	$m_A(t)$	covariance	
$A[B]$	$+*$	$m_B(u)$	$m_B(u)$	avec typage	Eiffel
B	$+-$	$-*$	$m_B(u)$	statique	

A	$m_A(t)$	$m_A(t)$	$m_A(t)$	covariance
$A[B]$	$+$	$m_B(u)$	$m_B(u)$	avec typage
B	$+$	$m_B(u)$	$m_B(u)$	dynamique

A	$m_{A\times t}$	$m_{A\times t}$	$m_{A\times t}$	sélection	
$A[B]$	$m_{A\times t}$	$m_{B\times u}$	$m_{B\times u}$	multiple	Clos
B	$m_{A\times t}$	$m_{B\times u}$	$m_{B\times u}$		

Figure 6.3 – 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.

— enfin, en C++, il s'agit du même ordre qu'en Java 1.5, mais uniquement sur les signatures qui ne sont pas masquées par une rédéfinition ou surcharge du nom de la méthode.

implicite S'ajoutent à cette complexité le fait que C♯ (et peut-être aussi C++ ?) considère les conversions implicites comme du sous-typage pour la surcharge statique.

On voit que les propriétés globales n'interviennent pas, toutes les méthodes étant mises sur le même plan. La sélection que nous avons proposée, qui est aussi celle de Java 1.5, repose quant à elle sur la définition suivante de la spécificité : $m(u)$ est plus spécifique que $m(t)$ ssi $u <: t$. La spécificité porte sur les propriétés globales et non plus sur les propriétés locales et la classe de définition n'intervient plus.

Surcharge statique et coercition ascendante Il a été assuré précédemment que la coercition (ou *cast*) 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 sur le compilateur. 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.

On peut ainsi lever tous les cas d'ambiguïté, en précisant la signature de la méthode que l'on veut activer. Curieusement, cette « précision » passe par la « généralisation » d'un paramètre. On pourrait néanmoins se passer de la surcharge puisque une affectation polymorphe la remplace parfaitement :

```
{ x : B
  y : u
  a : A
  a=x
  a.m(y)}     // m_t
```

Avec le traitement des conflits de nom en héritage multiple (C++, C♯) c'est le seul usage pratique de la coercition ascendante.

	t	$t[u]$	u		
A	$+-$	$-$	$m_A(u)$	surcharge statique incorrecte	C++, C♯
$A[B]$	$+-$	$-$	$m_A(u)$		
B	$m_B(t)$	$m_B(t)$	$m_B(t)$		
A	$+-$	$-$	$m_A(u)$	surcharge statique incorrecte	Java 1.4, Scala
$A[B]$	$+-$	$-$	$m_A(u)$		
B	$m_B(t)$	$m_B(t)$	$-$		
A	$+-$	$-$	$m_A(u)$	surcharge statique canonique	Java 1.5
$A[B]$	$+-$	$-$	$m_A(u)$		
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	$m_{B\times t}$	$m_{B\times t}*$	$m_{B\times t}*$		

Figure 6.4 – 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 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++, C♯, Java, Scala, ...
$A[B]$	$m_B(t)$	$m_B(t)$	$m_B(t)$		
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	$m_{B\times t}$	$m_{B\times u}$	$m_{B\times u}$		

Figure 6.5 – Surcharge statique et sélection multiple, avec 3 méthodes $m_A(t)$, $m_B(t)$ et $m_B(u)$.

Ambiguïté due à la généricité. Voir Section 7.11, page 146.

6.3.3 Surcharge statique en C++ (incorrecte).

Ce ne sont plus les propriétés globales qui se masquent, mais les ensembles de propriétés 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* [Ellis and Stroustrup, 1990, p. 205]. Ce comportement entraîne une très mauvaise qualité 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 (du moins relativement au protocole canonique de Java 1.5).

En complément, cette spécification de la surcharge ne garantit pas que la sous-classe soit un sous-type (Axiome 5.2). L'effet peut être particulièrement pénalisant avec les classes paramétrées, qui pourraient être instanciables par la super-classe nmais pas par la sous-classe (Chapitre 7).

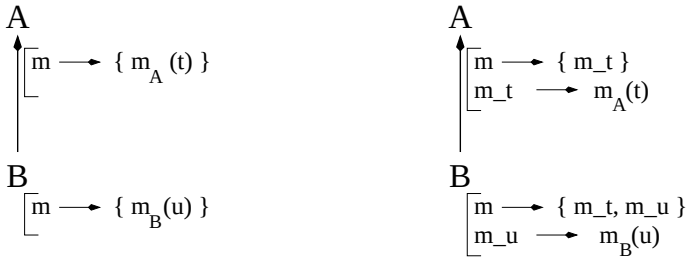


Figure 6.6 – Surcharge statique. En C++ (à gauche), chaque nom est lié à un ensemble de méthodes surchargées et la spécialisation provoque un masquage des noms. En Java (à droite), chaque nom est lié à un ensemble de propriétés globales que la spécialisation cumule, le masquage ne s'effectuant qu'au niveau de chaque propriété globale.

La protocole de C++ ressemble donc à celui de C♯, avec cette différence qu'en cas d'échec dans la première classe qui définit une méthode du nom considéré, le compilateur ne cherche pas dans les super-classes. Noter que l'héritage multiple interdirait de toute façon à C++ d'appliquer le protocole de C♯.

6.3.4 Surcharge statique et redéfinition en C♯

Le cas de C♯ est particulièrement intéressant car il met en jeu des traits de langages, la spécification de la surcharge et des considérations de robustesse face à l'évolution des librairies.

Comme il a été dit plus haut, dans le cas d'ambiguïté soulevé par JAVA 1.4 (Section 6.3.2), la sélection de C♯ diffère de celle de JAVA. En réalité, le protocole de sélection statique de C♯ ressemble à celui de JAVA mais est subtilement différent. Au lieu d'appliquer les étapes 1-2 de sélection statique du protocole de la Section 6.3.1 à toutes les méthodes de nom m connues par le type du receveur, ces étapes sont appliquées itérativement au type statique et à ses sous-classes, jusqu'à ce que l'étape 2 retourne un ensemble non vide. Les signatures introduites par les super-classes ne sont donc pas considérées si la classe courante dispose d'au moins une signature compatible.

Avantages et inconvénients

La justification de cette spécification est la suivante. La classe A peut être fournie par une bibliothèque et la classe B développée par le programmeur considéré. Le programmeur peut introduire dans B une méthode $m(t)$ dans un contexte où le nom m ne serait pas connu de A. Cependant, dans une version ultérieure de la bibliothèque, une méthode $m(u)$ pourrait être introduite dans A, et si $m(u)$ est plus spécifique que $m(t)$ ($u <: t$), il se pourrait que la première capture les appels initialement destinés à $m(t)$.

La spécification de la sélection statique est donc un moyen d'empêcher cette capture, le code utilisant B n'étant pas perturbé par la modification de A.

Cette spécification alternative aura cependant un inconvénient notable. Si l'on découpe une classe comme décrit dans la discussion sur la *stabilité* des évolutions, Section 4.4.2, page 50, deux méthodes de même nom peuvent être réparties dans les deux classes résultantes de telle sorte que la sélection statique ne se fasse plus de la même manière.

Au total, l'approche de C♯ ressemble à celle de C++, en réduisant l'échec à la compilation de C++ tout en assurant le respect de l'Axiome 5.2. Mais l'approche de JAVA 1.5 reste à nos yeux préférable, car elle met sur le même plan toutes les signatures d'une classe, sans les stratifier par héritage. Le problème de stabilité posé par C♯ est néanmoins réel, mais peut se résoudre à la main, en imposant dans la sous-classe la redéfinition de toutes les méthodes surchargées.

Déclaration des méthodes en C♯

La définition de méthode en C♯ repose sur 3 mots-clés :
- `virtual`, qui spécifie que la méthode est *virtuelle*, donc soumise à liaison tardive ;
- `new`, qui indique une introduction, ou réintroduction si la signature est déjà connue des super-classes ;
- `override`, qui indique une redéfinition.

Si `virtual` est seul présent, `new` est sous-entendu. Inversement, `override` est obligatoire en cas de redéfinition. La justification de `override` est la même que celle du traitement de la surcharge statique : l'introduction ultérieure d'une méthode $m(t)$ dans A ne pourra pas transformer l'introduction de $m(t)$ dans B en une redéfinition.

Dans l'exemple ci-dessus, le programmeur définit $m(t)$ dans B avec un `new` implicite ou explicite, et cette définition ne peut en aucun cas être prise pour une redéfinition d'une méthode de nom m dans A. Cependant, utiliser `new` pour introduire une méthode d'une signature héritée paraît particulièrement hérétique : le programmeur n'a qu'à choisir un autre nom. De plus, l'utilisation effective de `new` a un effet sémantique sur le sous-typage : on peut toujours substituer textuellement le sous-type au super-type, car ça compilera toujours. Mais le code compilé ne sera pas le même et la sémantique ne sera pas conservée.

```
class A { virtual foo(T) ... }
class B inherit A { virtual new foo(T) ... }

main() {
     A b = new B();
     T t
     b.foo(t) }
```

Dans cet exemple, la méthode `A.foo` est invoquée. Si l'on remplace `A` par `B` dans le `main`, c'est `B.foo` qui sera invoquée. La substituabilité est vérifiée du point de vue de l'absence d'erreur de compilation, mais pas du point de vue de la sémantique.

Le mot-clé `new` pourrait donc être réservé à la correction d'une incompatiblité due à l'évolution d'une super-classe : c'est un patch, pas forcément glorieux, mais qui a l'avantage de corriger le problème.

Mais cela n'empêcherait pas la capture par surcharge. Une solution a posteriori serait de rajouter au langage un mot-clé `overload` remplaçant `new` lorsqu'on souhaite faire une surcharge statique explicite. La sélection statique pourrait alors redevenir celle de JAVA 1.5, qui est certainement la plus satisfaisante pour l'esprit.

6.3.5 Surcharge statique et substituabilité.

La substituabilité peut se définir avec plusieurs degrés de liberté. Le degré zéro consiste à considérer la substitution d'une *instance* de type t_2 à une *expression* de type t_1. Avec cette interprétation, la surcharge n'a jamais d'effet sur la substituabilité.

On peut aussi la comprendre comme la possibilité de substituer un sous-type t_2 à un type t_1 dans une annotation de type d'un (fragment de) programme, sans en changer le comportement ou au moins la correction. Ce n'est pas toujours possible, 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, si `x.foo(arg)` est correct quand le type de `x` est t_1, on s'attend à ce que l'expression reste correcte lorsque le type de `x` est un sous-type t_2.

En réalité, ce n'est pas toujours le cas, même en JAVA, où la substitution peut provoquer une erreur de compilation. En effet, il peut y avoir une ambiguïté de surcharge avec t_2 qui serait levée avec t_1. De même, en C++, le `foo` de t_1 a pu être masqué par un autre `foo` dans t_2.

Enfin, même si l'expression reste correcte (i.e. compilable), son interprétation peut changer car la résolution de la surcharge sur t_2 peut être différente de celle sur t_1.

Ainsi, suivant l'interprétation fine de ce que l'on entend pas substituabilité, la surcharge statique apparaît comme une limitation au principe que la spécialisation de classes soit du sous-typage (Axiome 5.2). Si l'on veut maintenir cet axiome, il est sans doute nécessaire d'appliquer le mode d'emploi pratique que nous proposions pour C++ : lorsqu'on introduit une surcharge dans une classe, il faut redéfinir toutes les méthodes de même nom (et de même nombre de paramètres) héritées des super-classes. Alternativement, on pourrait interdire d'introduire une surcharge dans une classe lorsqu'une méthode de même nom et nombre de paramètres a été introduite dans une super-classe.

6.3.6 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, dans tous les langages, si l'on effectue un renommage global pour éliminer d'un coup toute surcharge. Les différences subtiles de spécification entre les langages font que la réponse n'est pas évidente si l'on considère un renommage progressif. En C++ par exemple, à cause du masquage de noms, il est moins évident que le renommage d'une méthode ne perturbe pas la sélection des autres.

6.3.7 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 : bien entendu, les différents attributs surchargés ne peuvent pas être introduits dans la même classe. Le type de l'attribut n'a aucun effet dans la sélection : il est donc possible de surcharger un attribut sans changer son type.

La surcharge d'attribut est fortement déconseillée, mais le programmeur n'est pas à l'abri d'un mauvais copier-coller et il doit savoir que ça existe.

Dans le cas où il s'agit d'une *variable statique*, la surcharge peut être un moyen de simuler une *variable de classe* ou un *attribut partagé* en encapsulant les accès à ces variables statiques par des *accesseurs* qui eux ne sont pas statiques. Voir Sections 11.2.3 et 14.6.

6.3.8 Surcharge statique et redéfinition

Dans les langages avec surcharge statique, la différence entre la surcharge et la redéfinition résulte en général simplement de la comparaison entre la signature de la méthode telle qu'elle est définie avec

les signatures des méthodes héritées. C'est comme cela que c'est spécifié en JAVA et C++, ainsi que dans le méta-modèle (Chapitre 3).

En revanche, certains langages permettent aussi de surcharger une méthode héritée avec une méthode de même signature : ce sont les mots-clés `reintroduce` en TURBO PASCAL, et `new` en C♯. Comme EIFFEL avec `redefine`, ce dernier langage oblige aussi à spécifier explicitement qu'une définition est une redéfinition, avec le mot-clé `override`. A défaut, le mot-clé `new` est sous-entendu.

6.3.9 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.

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.

6.3.10 Surcharge statique en typage dynamique

La surcharge statique n'est pas complètement propre aux langages à typage statique dans la mesure où elle peut aussi reposer sur le nombre de paramètres. Il est donc possible de la rencontrer dans des langages à typage dynamique, où elle ne pose néanmoins aucun problème particulier.

Ainsi, en SMALLTALK et OBJECTIVE-C où les paramètres sont passés par mots-clés, un peu comme avec `&key` en COMMON LISP, la surcharge statique s'appuie sur le nombre et le nom des paramètres.

6.3.11 Pour ou contre la surcharge statique

Inconvénients. Pour résumer, les arguments contre la surcharge statique sont les suivants :

- elle est conceptuellement dangereuse puisqu'elle habitue à utiliser un même nom pour des choses différentes,
- elle est inutile puisqu'il est possible de renommer les méthodes en question,
- elle est dangereuse car elle provoque des instabilités en cas de transformation,
- et limite la substituabilité,
- elle provoque des ambiguïtés,
- est la cause de la paresse des concepteurs de langages en matière de spécification des constructeurs,
- et c'est, avec le traitement des conflits d'attributs en héritage multiple de C++, la seule justification de l'existence du *cast* ascendant ;
- pour enfoncer le clou du cercueil, le fait que JAVA 1.4, JAVA 1.5, C♯, C++ et SCALA en aient cinq spécifications différentes n'est clairement pas bon signe, surtout si l'on rajoute à cela le comportement, encore différent, d'OCAML d'un côté et de la sélection multiple de l'autre.

Avantages. Après ce déluge de critiques, on peut néanmoins noter quelques arguments en faveur de la surcharge statique :

- lorsque les types primitifs ne sont pas bien intégrés aux classes, par exemple en JAVA ou C++, en ce sens qu'une valeur d'un type primitif ne peut pas être affectée à une variable typée par `Object` (Section 8.1), il n'est pas possible d'utiliser de méthodes polymorphes sur les types primitifs ; pour faire des entrées-sorties, par exemple la méthode `write(Object)`, il faut une méthode spécialisée par type primitif, et la surcharge évite d'avoir à définir des méthodes `writeInt(int)` ;
- en particulier, lorsque les langages ont une syntaxe infixe pour les opérateurs courants, il est bien pratique de les surcharger pour tous les types primitifs (mais là encore, c'est parce que les types primitifs ne sont pas bien intégrés aux classes) ;
- le schéma de simulation de la covariance décrit ci-dessous repose sur la surcharge statique, ainsi que sur une autre anomalie, le fait que `super` explicite le nom de la méthode et soit implicitement typé par la super-classe.

Finalement, ces seuls arguments positifs sont des cautères sur des jambes de bois : un langage à objets bien conçu, avec un système de types bien intégré, comme Eiffel ou Nit, n'a pas besoin de surcharge statique.

Conseils. Nos conseils ultimes seront :
— pour les concepteurs de langages, d'exclure la surcharge statique de leurs spécifications,
— pour les concepteurs de librairies ou d'applications dans des langages avec surcharge statique, de ne pas introduire de méthodes (et encore moins d'attributs) surchargés,
— pour les utilisateurs de librairies dans des langages avec surcharge statique, de faire attention !

Dans un langage comme Java, C♯ ou C++, où la mauvaise intégration des types primitifs, le poids de l'histoire et du code accumulé interdisent de supprimer la surcharge, on suggère d'introduire un mot-clé ou une annotation permettant d'expliciter une surcharge ou son absence (Section 6.3.4).

6.3.12 Programmer "Schtroumpf"

Mal *schtroumpfer* les *schtroumpfs*,
c'est *schtroumpfer* au *schtroumpf* du *schtroumpf*

Albert Camus, traduit en schtroumpf par l'auteur

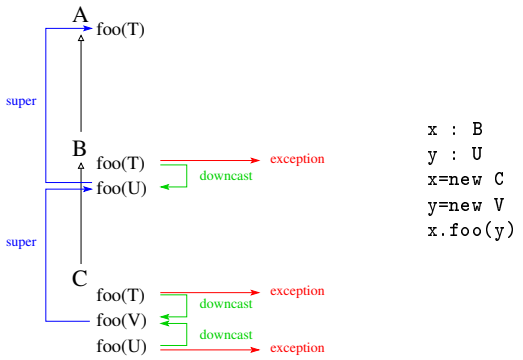
L'exercice suivant a pour objectif de convaincre le lecteur du caractère inacceptable de la surcharge statique :

Exercice 6.3. Prendre une application objet existante, développée en Java, Scala, C++ ou C♯ ; renommer toutes les méthodes *introduites* dans les classes de ce projet, en utilisant exclusivement deux noms de méthodes, `foo` et `bar` ; on procédera de super-classe en sous-classe, méthode introduite après méthode introduite ; lorsque, pour une signature donnée, les deux méthodes `foo` et `bar` de cette signature sont déjà connues par la classe, on ajoutera un paramètre supplémentaire, de n'importe quel type qui résolve le conflit, en rajoutant un argument supplémentaire, `null` ou litéral, dans tous les sites d'appel correspondant. □

Exercice 6.4. En Java (ou Scala) développer le plug-in Eclipse qui fait cette transformation de façon automatique. □

6.4 Simulation de la redéfinition covariante

Si l'on ne dispose pas d'un langage qui admette la redéfinition covariante, on peut la simuler. Il faut pour cela disposer de 3 mécanismes : redéfinition invariante, surcharge statique et coercition de types. Java et C++ s'y prêtent donc très bien.



Le schéma montre les signatures nécessaires, ainsi que le flux de contrôle entre les différentes méthodes, par *upcast* et appels à `super`. La séquence de code montre la nécessité de la présence de la signature `foo(U)` dans C, aussi bien en Java qu'en C++.

Figure 6.7 – Redéfinition covariante sur 3 classes successives, avec $C \prec B \prec A$ et $V \prec U \prec T$.

6.4.1 Principe

On peut voir cela comme un *pattern* (ou patron) « simulation de la redéfinition covariante ».

Pattern 6.1 (Simulation de la covariance par surcharge) *Le principe du* pattern *est le suivant :*
- *la méthode $m_A(t)$ est définie dans la classe A ;*
- *une méthode $m_B(u)$ est définie dans la sous-classe $B \prec A$, avec $u <: t$: elle effectue le comportement souhaité pour les instances de B ;*
- *une autre méthode $m_B(t)$ est définie dans B pour rattraper les mauvais paramètres : cette méthode applique une coercition de t vers u sur son paramètre et appelle m, c'est-à-dire $m_B(u)$ en cas de réussite ; elle gère l'échec de façon ad hoc.*

Il faut bien sûr appliquer le *pattern* à chaque étage de la spécialisation : si l'on veut définir $m_C(v)$, avec $C \prec B$ et $v <: u$, il faut aussi définir $m_C(t)$ et $m_C(u)$ (Figure 6.7). On vérifiera, par des programmes test simples, que les variations sur la surcharge statique qui existent entre C++ et Java n'ont pas d'effet ici.

Avantages. Ce patron présente des avantages ...
- le principal avantage est que le comportement à l'exécution est celui qu'on attend (c'est bien le moins !), à un détail près cependant (voir les inconvénients) ;
- un avantage secondaire concerne l'efficacité : le test de sous-typage est souvent inefficace, et la branche $m_B(u)$ en fait l'économie ; c'est aussi pour l'efficacité qu'il faut définir $m_C(t)$;

Inconvénients ... et des inconvénients :
- un inconvénient important (le détail précité) est que le mauvais comportement à l'exécution ne peut pas être écarté dès la compilation : la vérification statique ne va pas permettre d'éliminer des fragments comme `{x:B; y:t; x.m(y);}` que le pessimisme du compilateur ne devrait pas pouvoir accepter ;
  En réalité, cette impossibilité peut être partiellement levée en C++, en jouant sur les droits d'accès que l'on peut redéfinir : on mettra $m(t)$ `private` dans B, ce qui fera rejeter le fragment précédent en dehors de B.
- l'inconvénient majeur réside dans le caractère subjectif de la solution : tout repose sur la discipline du programmeur ; en particulier, le concepteur des classes A et B de la Figure 6.7 doit convaincre le programmeur de l'extension C de la nécessité d'appliquer le même schéma...
- un dernier inconvénient, sans doute relativement mineur, réside dans le nombre de signatures de méthode qui entraîne une augmentation du degré du polynome qui caractérise la taille des implémentations dans le pire des cas : au lieu d'être quadratique elle deviendra cubique et, dans le cas de C++, elle passera de cubique à la puissance 4 !

6.4.2 Méthodes binaires

Les méthodes binaires, dont le paramètre a pour type la classe courante (voir Section 6.2.4 et 5.6) peuvent se définir en recourant à la simulation de la covariance, mais l'échec de la coercition ne se traduit pas forcément par une exception.

Exercice 6.5. Donner le schéma de la méthode `equal` dans un cadre de simulation de la covariance qui ne réponde `vrai` que pour deux objets du même type dynamique. Dans un deuxième temps, s'assurer que le test est bien symétrique (`equal` doit implémenter une relation d'équivalence).

Pour une spécification complète, voir Section 8.2. □

6.5 Implémenter une association Uml et ses redéfinitions covariantes

6.5.1 Implémenter une association binaire

Nous avons vu (Section 2.3, page 23) qu'une association Uml pouvait s'interpréter comme une relation, au sens mathématique du terme, donc comme une partie du produit cartésien des extensions des classes considérées. Lorsque la relation est binaire, l'implémentation la plus simple consiste à définir, dans les 2 classes A et B reliées par l'association, un attribut typé par l'autre classe.

Pattern 6.2 (Implémentation d'une association Uml) *L'association de base entre les classes A et B peut s'implémenter dans la classe A par :*

— *un attribut typé par B ou une collection de B ;*

— *des méthodes typées par B pour accéder à cet attribut (lecture, affectation, ajout, etc.).*

On fera de même dans B en inversant A et B.

Protocole d'association

Ce schéma statique doit s'accompagner d'un protocole de construction de l'association qui revient à rajouter un nouveau tuple dans le produit cartésien. Associer deux instances de A et B, respectivement a et b, est un processus délicat :

— il faut d'abord que a et b ne soient pas déjà associés ;

— il faut maintenir la symétrie et la cohérence entre les 2 attributs : si a référence b, il faut que b référence a ;

— il faut pouvoir déassocier a et b ;

— si l'un des rôles de l'association est d'arité 1, l'affectation de l'attribut correspondant doit provoquer la désassociation de la valeur précédente qui va être écrasée ;

— l'arité (ou cardinalité) des deux rôles de l'association doit aussi être prise en compte ;

— enfin, l'association de a et b n'est pas censée réussir dans tous les cas, puisqu'elle est soumise à des conditions de cardinalité et à des conditions de type en cas de redéfinition covariante (voir ci-dessous) : comme l'association de a et b repose sur la mise à jour de deux attributs, il faut en faire une transaction, de telle sorte que les deux soient modifiés ou aucun ; si la mise à jour du second est impossible, il faut annuler la mise à jour du premier, ce qui est difficile : le plus simple est donc de séparer les opérations de vérification et celles de mise à jour.

Remarque 6.1. La circularité des définitions pose un problème constant en C++. Pour pouvoir définir ces 2 classes A et B qui se font mutuellement référence l'une à l'autre, il est nécessaire de faire des définitions vides `class A;` avant la définition de B, et réciproquement.

6.5.2 Protection des collections

Lorsque l'association, ou plutôt l'un des ses rôles, est multi-valuée il faut passer par une collection, par exemple dans A un attribut `mesB` typé par `Set⟨B⟩`. Mais cette collection n'est pas un vrai objet du modèle. En particulier,

— elle ne doit pas être modifiée en dehors de la classe, et l'attribut `mesB` doit avoir le plus haut niveau de protection (au moins `private`),

— elle doit donc être créée dans la classe, dans le constructeur, et non pas être passée en argument du constructeur,

— elle ne doit pas être passée au monde extérieur comme paramètre de méthode, mais uniquement comme receveur dans le protocole de mise à jour de l'association,

— elle ne devrait pas être retournée par les méthodes de la classes, par exemple l'accesseur `get-mesB`, autrement que comme une collection immutable.

Pour garantir le dernier point, le langage peut proposer des collections immutables, super-types des collections mutables. Par exemple `ImmutableSet` serait un super-type de `Set`, et le type de retour de `get-mesB` serait `ImmutableSet⟨B⟩`.

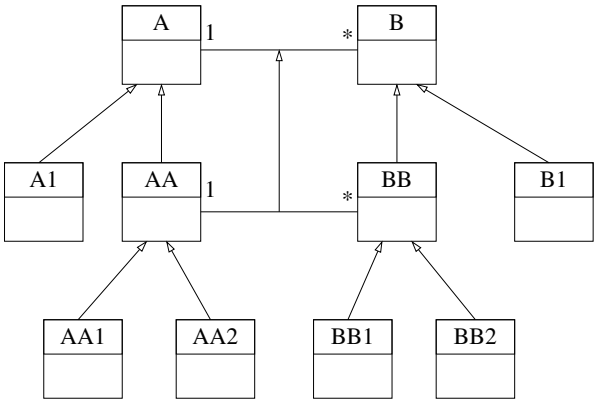


FIGURE 6.8 – Une association entre les classes A et B est redéfinie entre les classes AA et BB.

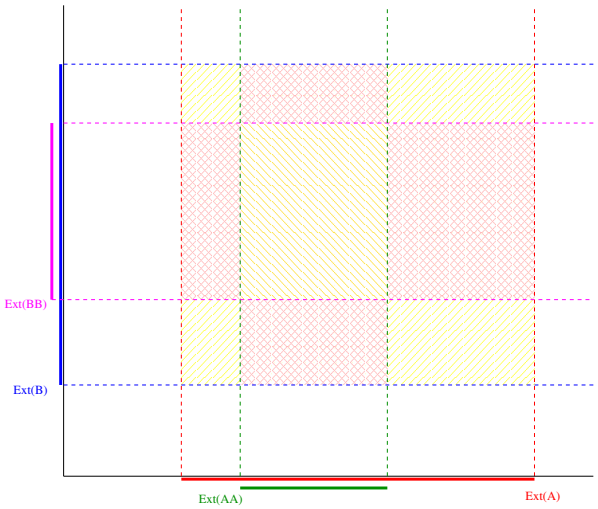


FIGURE 6.9 – La spécialisation d'association dans le produit cartésien des extensions : les parties jaunes sont les seules parties possibles du produit cartésien.

Le *joker* de JAVA est une alternative (voir Section 7.4.3). En effet, `Set⟨? <: B⟩` se comporte comme une collection immutable. Voir la Section 8.3.

Par ailleurs, l'attribut qui référence la collection devrait a priori être immutable. Voir la Section 5.2.3.

6.5.3 Spécialisation d'associations

La figure 6.8 décrit un modèle UML basé sur deux classes A et B reliées par une association et spécialisées par diverses sous-classes. Parmi ces sous-classes, AA et BB redéfinissent l'association, au sens où l'association met en relation une instance de AA avec exclusivement des instances de BB, et réciproquement.

Une association UML doit se comprendre comme une relation binaire entre les extensions des deux classes, donc un élément du produit cartésien de ces extensions, $Ext(A) \times Ext(B)$. La spécialisation de l'association entre les classes AA et BB revient à dire les restrictions de la relation à $Ext(AA) \times Ext(B)$ et à $Ext(A) \times Ext(BB)$ sont incluses dans $Ext(AA) \times Ext(BB)$. La Figure 6.9 symbolise cela : les rectangles roses hachurés dans les deux sens ont une intersection vide avec la relation binaire.

Il y a de nombreuses façons d'implémenter une telle association. Parmi elles, la covariance s'impose par sa simplicité.

On peut ensuite considérer le cas de la spécialisation.

Pattern 6.3 (Spécialisation d'une association UML par simulation de la covariance) *La spécialisation de l'association entre les classes AA et BB consiste à :*

- *ne pas redéfinir les attributs qui implémentent les 2 rôles de l'association : avec la surcharge statique, ce serait définir un nouvel attribut ;*
- *redéfinir les méthodes d'accès en écriture en simulant la covariance : les valeurs des attributs seront donc toujours du bon type ;*

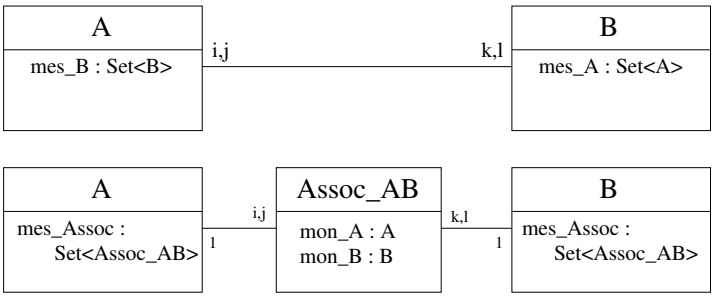


FIGURE 6.10 – Une association entre les classes A et B, implémentée de façon directe ou par réification.

— *redéfinir les méthodes d'accès en lecture pour redéfinir de façon covariante le type de retour et appliquer une coercition sur la valeur retournée (qui est du bon type dynamique mais pas statique).*

Le dernier point est essentiel, mais se traite différemment suivant l'arité du rôle. En effet, dans l'exemple, dans BB, l'accesseur à l'attribut est typé par AA : il se contente donc juste d'encapsuler l'appel à `super` par un *downcast* vers A, lequel réussira toujours si la construction des objets est correcte. En revanche, dans AA, l'accesseur est typé par une collection de BB, et le *downcast* n'est pas possible, parce qu'il n'y a pas de sous-typage entre une collection de BB et une collection de B (voir Chapitre 7). Il faudra donc faire le *downcast* chaque fois que l'on extrait un élément de la collection, ce qui est fastidieux. En particulier, il faudra le faire chaque fois que l'on itère sur la collection, ce qui ne peut pas toujours être encapsulé dans le code des classes AA et BB et va donc impacter le code utilisateur.

6.5.4 Réification des associations

Les associations sont souvent implémentées par *réification*. Une classe `assoc-AB` représente alors l'association entre les classes `A` et `B`, et chacune de ses instances représente un couple d'instances de `A` et `B`. Cela permet d'associer à chaque paire des propriétés représentant le contexte de l'association, par exemple la date et l'auteur de l'association des 2 instances. Les contraintes de cohérence ne sont pas plus simples à établir puisqu'il faudra vérifier qu'il n'y a pas deux instances d'`assoc-AB` pour le même couple.

La réification est donc plus puissante que ce nous avons présenté, mais elle pose un problème récursif majeur : nous avons maintenant 2 associations binaires, entre `A` et `assoc-AB` d'une part, entre `B` et `assoc-AB` d'autre part. Chacune de ces associations représente un *rôle* de l'association d'origine. Ces deux associations-là devront bien être implémentées sans réification (Figure 6.10).

Généralisation : associations n-aires La généralisation aux associations n-aires, pour $n > 2$ ne peut pas se faire autrement que par *réification*. L'association doit être modélisée par une classe et les n rôles de l'association n-aire sont modélisés par n associations binaires.

*****(à développer...)*****

Remarque 6.2. Dans ce contexte d'associations, le terme d'*arité* prend deux sens sans rapport, suivant qu'il s'applique à une association d'un point de vue global (elle peut être binaire, c'est le nombre de rôles), ou à chacun de ses rôles, dont l'arité est généralement caractérisée par un intervalle qui représente le nombre de valeurs possibles du rôle.

6.5.5 Génération automatique par méta-programmation

La simulation de la covariance est une solution *ad hoc* qui a l'avantage de résoudre le problème mais a l'inconvénient de reposer sur la discipline *des* programmeurs. Il ne suffit pas pour *un* programmeur de s'y plier, il faut encore qu'il avertisse *tous* les programmeurs qui vont réutiliser son code.

C'est une contrainte énorme, puisque les programmeurs doivent à la fois connaître le *pattern* et savoir à quelles méthodes il faut l'appliquer. Par ailleurs, le passage à l'échelle n'est pas assuré : une chaîne de spécialisation de n classes nécessite n surcharges !

La méta-programmation (Chapitre 11) serait une solution beaucoup plus élégante : le programmeur peut alors spécifier des mots-clés supplémentaires et les programmer pour générer le code *ad hoc*.

6.5.6 Alternative avec la sélection multiple

La sélection multiple offre aussi un moyen efficace pour simuler la redéfinition covariante, sur le même principe mais en plus simple car la sélection repose sur le type dynamique du paramètre. On décrit ici le cas des fonctions génériques de CLOS mais l'adaptation aux multi-méthodes encapsulées est immédiate — il faudrait juste qu'un langage les propose.

Pattern 6.4 (Simulation de la covariance par sélection multiple) *Le principe du* pattern *est le suivant :*

- *la méthode $m(A,t)$ est d'abord définie ;*
- *on définit ensuite une méthode $m(B,u)$ qui effectue le comportement souhaité pour les instances de B ;*
- *une autre méthode $m(B,t)$ est définie pour rattraper les erreurs qui sont gérées de façon* ad hoc.

Il faut donc toujours définir les 3 mêmes méthodes et gérer le cas d'erreur, mais la coercition explicite n'est plus nécessaire. Par ailleurs, il n'est pas nécessaire de définir $m(C,u)$ pour une sous-classe C de B : quelle que soit la profondeur de la spécialisation, 2 méthodes suffisent dans chaque classe.

6.6 Simulation de la sélection multiple

Il existe différentes façons — encore des patrons — de simuler la sélection multiple dans un langage à sélection simple. Plus les paramètres sur lesquels porte la sélection sont nombreux, plus la simulation est lourde : nous nous contenterons d'examiner le cas d'une sélection sur 2 paramètres, le receveur plus un paramètre de multi-méthode.

6.6.1 Version naïve par tests de sous-typage

La façon la plus naïve consiste simplement à inclure, dans le corps d'une méthode en sélection simple sur le premier paramètre, des tests de sous-typage sur les autres paramètres. Les inconvénients sont cependant nombreux :

- lourdeur syntaxique, surtout dans les langages cités ici qui ne disposent pas d'une construction comme `typecase` ;
- absence de modularité : si l'on veut rajouter une sous-classe du type deuxième paramètre, il faut modifier les classes du premier paramètre (mais c'est aussi un défaut des multi-méthodes encapsulées).

6.6.2 Par surcharge statique et cascade d'appels

De façon générale, la programmation objet permet de se passer de test de sous-typage en les remplaçant par des appels de méthodes. La solution passe donc par une cascade de 2 appels de méthodes et si l'on voulait traiter une sélection sur n paramètres, il faudrait une cascade de n appels de méthodes.

Soit une méthode m à sélection sur ses deux paramètres, et définie dans les couples $\{(t_i, u_i)\}$ pour $i = 1..k$. Les couples (t_i, u_i) sont tous 2 à 2 distincts mais les t_i (resp. u_i) ne le sont pas.

Pour chaque $i \in 1..k$,

- on définira dans t_i la méthode $mm_{t_i}(x : u_i)$ dont le corps consiste dans l'appel `x.mmm(this)`.
- dans u_i, on définit la méthode $mmm_{u_i}(t_i)$ dont le corps est le corps d'origine de m_{t_i,u_i}. Du fait de la surcharge statique la méthode mmm sélectionnée sera celle de paramètre t_i.

Sans surcharge statique, il faudrait renommer $mmm(t_i)$ en mmm_i, par exemple, mais le résultat serait le même (en JAVA en tout cas). A contrario, en C++, il est sans doute préférable de renommer si l'on ne veut pas être obligé de redéfinir de nombreuses méthodes mmm pour éviter le masquage.

EXERCICE 6.6. Montrer qu'en JAVA, cette transformation correspond exactement à la sélection multiple dans le produit des types lorsqu'il n'y a pas de conflit d'héritage multiple.

Comparer avec CLOS dans le cas où il y a un conflit. □

EXERCICE 6.7. Même question avec C++. Comment faut-il modifier le schéma pour obtenir le bon comportement. □

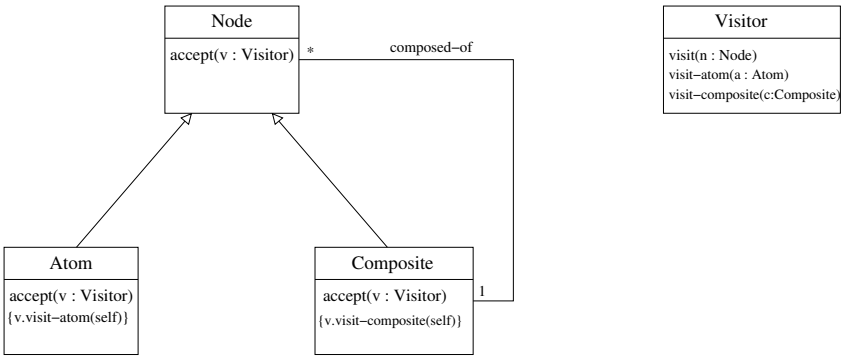


FIGURE 6.11 – Le patron *Visiteur*.

6.6.3 Le *pattern* visiteur et ses variantes

Le patron *Visiteur* représente à la fois l'un des patrons les plus fréquents, un succédané de sélection multiple et un exemple typique de surcharge statique.

On peut voir de deux façons le patron Visiteur.

— Lors de la conception d'une instance d'un patron *Composite*, par exemple un ensemble de classes représentant une structure de listes ou d'arbres, le patron Visiteur permet d'anticiper l'extension future de ces classes (Figure 6.11). Au lieu d'introduire une nouvelle méthode dans le composite, on va créer une classe qui va implémenter la méthode de visite.

— Alternativement, on peut voir ce patron comme une façon de simuler la sélection multiple entre deux catégories de classes, celles du composite d'un côté, et celle du visiteur de l'autre.

Le *protocole* est le suivant. La classe `Node` du composite introduit une méthode `accept` prenant en paramètre le visiteur, et chaque implémentation de cette méthode `accept` dans les sous-classes concrètes `xxx` de `Node` appelle une méthode `visit-xxx` sur le visiteur en lui passant l'instance de `Node` courant en paramètre. L'implémentation (en jargon moderne, le code métier) de la visite se fait dans ces méthodes `visit-xxx` du `Visitor`. Si l'on veut effectuer des visites différentes, il suffit de faire des sous-classes de `Visitor`.

La surcharge statique intervient alors par le fait que toutes ces méthodes `visit-xxx` peuvent s'appeler `visit`, puisque chacune est sélectionnée statiquement par le type `xxx` de son paramètre. Cet exemple montre bien à quel point la surcharge équivaut à du renommage[7].

Sur ce patron très simple se greffent de nombreuses variantes que nous ne décrirons pas ici. Le lecteur intéressé trouvera une littérature très abondante sur le web.

EXERCICE 6.8. Définir une structure de liste chaînée, à la LISP, et le visiteur qui permet d'afficher une liste. □

6.7 Combinaison de méthodes

6.7.1 Appel à `super`

Le principe de base de la combinaison des méthodes consiste à définir une méthode dans la sous-classe qui appelle la méthode redéfinie dans la super-classe. On appelle cela l'appel à `super` pour reprendre le terme utilisé en SMALLTALK et JAVA.

Dans sa spécification originale en SMALLTALK, l'appel à `super` est entaché de deux défauts qui concernent aussi bien sa syntaxe que son explication et qui perdurent jusqu'à maintenant dans la plupart des langages.

Pseudo-variable `super`. `super` est spécifié comme une pseudo-variable similaire à `self` (ou `this`) et il est souvent décrit comme « `self` dans la super-classe ». C'est vrai aussi en JAVA et C♯ (mais pour ce dernier avec le mot-clé `base` remplaçant `super`), et même en EIFFEL où le mot-clé `Precursor` a la même syntaxe que `Current`. Or cette notion de « `self` dans la super-classe » n'a pas grand sens. En tant que variables, les valeurs de `super` et de `self` sont identiques, et seul leur type statique diffère : le type de `self` est la classe courante alors que celui de `super` est la super-classe. Cependant, la liaison tardive «

7. Modulo le cas particulier de C++ qui nécessite quelques précautions à cause de son interprétation très personnelle du masquage.

normale » ne dépend pas du type statique, et le rôle joué par `super` repose en réalité sur un mécanisme spécifique caché.

Soit une class `B`, sous-classe directe de `A`. Dans `B`, tout se passe comme si l'on avait les deux déclarations de type

```
this : B
super : A
```

On pourrait penser que dans le contexte de `B`, les deux instructions suivantes sont équivalentes :

```
((A)this).foo()
super.foo()
```

Mais en fait la première fait une liaison tardive alors que la seconde fait un *appel statique*.

En réalité, la bonne spécification de `super` consiste à en faire un appel de fonction, comme en Clos avec `call-next-method`.

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 propriété globale. 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.

Spécification de la méthode « super-appelée ». En Java, Smalltalk et C♯, la syntaxe de l'appel à `super` autorise (voire impose) la mention d'un nom de méthode qui peut être différent du nom de méthode courant. Cela revient à spécifier un mécanisme de liaison tardive particulier, dans lequel le *lookup* commencerait à la super-classe (ou au type statique). Comme souvent, on a l'impression que la spécification découle de la possibilité de l'implémentation.

Cependant à la lumière du méta-modèle (Chapitre 3), ce mécanisme devrait être spécifié de façon plus restrictive : la méthode appelée et la méthode appelante doivent appartenir toutes deux à la même propriété globale.

C'est ce qu'impose `Precursor`, l'équivalent Eiffel de `super`, qui ne permet pas de mentionner le nom de la méthode. Le `super` de Prm/Nit a le même comportement.

Noter que c'est la conjonction de ces deux défauts qui permet l'usage de `super` dans un contexte de surcharge statique, par exemple pour la simulation de la covariance (Section 6.4) : la surcharge est désambiguïsée par le type statique de `super` et l'appel se fait sur une méthode de même nom mais de types de paramètres différents. Si `super` était bien spécifié, on ne pourrait pas faire l'appel à `super` dans $m_B(u)$ et l'introduction de cette signature serait impossible. Le patron se ramènerait donc à une simple définition, sans surcharge, dans laquelle le type dynamique du paramètre est vérifié avant de faire appel à `super`.

6.7.2 En héritage multiple

En héritage multiple, le principe de `super` devient lui-même ambigu si la méthode appelante a été définie pour résoudre un conflit de propriétés locales. Le cas de C++ avec des appels statiques a été examiné en Section 4.3.1, page 45.

En Eiffel. Le mécanisme équivalent à `super`, nommé `Precursor`, n'est applicable que s'il n'y a pas de conflit, ou si ce conflit a été résolu par `undefine`. Par ailleurs, le nom de la méthode ne peut pas être mentionné, ce qui corrige le défaut de `super` en Java ou Smalltalk. Si l'on veut combiner les méthodes de plusieurs super-classes, il faut en renommer (par `rename`) pour éviter le `undefine`. Mais le risque d'héritage répété est alors le même qu'en C++.

Avec les linéarisations Dans un langage avec linéarisation comme Clos, `super` est remplacé avantageusement par un mécanisme nommé `call-next-method` qui consiste à appeler la méthode suivante dans l'ordre de la linéarisation [8]. Le gros avantage est d'éviter tout risque d'héritage répété et de double évaluation.

8. Noter qu'en Scala, `super` désigne en fait le mécanisme de `call-next-method`.

Si `super` ne pose pas de problème de typage parce que l'appelée est vue par l'appelante, ce n'est plus le cas avec `call-next-method`. Comme `call-next-method` peut appeler n'importe quelle méthode de la même propriété globale, le type statique apparent de ses paramètres et de sa valeur de retour ne peuvent pas être plus précis que ceux de la classe d'introduction. Il est donc impossible d'une part de passer des paramètres explicites en cas de redéfinition covariante, et d'autre part de retourner la valeur retournée par `call-next-method` — des vérifications dynamiques (coercition) sont nécessaires.

Mais `call-next-method` peut être appelé sans arguments explicites : dans ce cas, ce sont les arguments passés à la méthode appelante qui sont transmis tels que à la suivante. Paradoxalement, dans cet usage, la redéfinition covariante est sûre : une vérification est nécessaire en entrée, mais les méthodes peuvent ensuite s'enchaîner sans risque d'erreur.

Remarque 6.3. Noter que l'appel statique de C++ (::), le `super` de SMALLTALK et JAVA, ou le `Precursor` d'EIFFEL correspondent au même mécanisme fondamental qui consiste en un appel statique de la méthode « suivante » qui est visible (mais pas au sens de la section suivante) depuis la classe courante : quel que soit le type dynamique du receveur, ce sera toujours la même méthode qui sera appelée. La seule différence est donc syntaxique[9].

En revanche, `call-next-method` en CLOS (ou `super` de SCALA) correspond à un appel dynamique d'une méthode qui n'est par forcément visible de la classe courante et qui dépend du type dynamique du receveur.

6.7.3 En sélection multiple

Le problème est dans tous les cas assez proche de celui de l'héritage multiple. Dans le produit des types, c'est clairement le même. Avec les multi-méthodes encapsulées, on a le choix entre rechercher une branche plus générale dans la même multi-méthode ou une multi-méthode plus générale dans la super-classe.

6.7.4 Invariants de classe

Les invariants de classe (Section 5.5.3, page 83) mettent en œuvre une combinaison de méthodes un peu particulière dans la mesure où elle repose sur une conjonction de toutes les méthodes héritées.

6.8 Modularité et visibilité

La problématique de la *modularité* est essentielle en génie logiciel et l'histoire des langages de programmation peut se voir comme sa quête.

Nous avons vu en Section 2.4 que la modularité s'exprimait par deux éléments :

— la définition des unités de code qui forment des *modules* à partir desquelles on va construire des applications,

— et la protection du contenu de ces modules vis-à-vis du monde extérieur, pour réduire le couplage entre modules.

La présente section va considérer ce deuxième aspect.

6.8.1 Visibilité, protection, droit d'accès, encapsulation, export, etc.

Toutes ces dénominations variées recouvrent des notions voisines qui consistent à permettre de spécifier, dans un langage, que tel fragment de code (la source) a le droit d'accéder à telle entité du programme (la cible). Sans perte de généralité, la source et la cible sont des classes ou des propriétés. La bonne terminologie est donc celle du droit (point de vue de la source) ou de la protection (point de vue de la cible). Comme en témoigne la remarque 6.3, le terme de visibilité prête à confusion : quelque-chose peut être visible sans être accessible. Ces notions sont donc apparentées aux notions de *portée* (ou *scope*) usuelles, mais cette portée n'est plus uniquement *lexicale*, c'est-à-dire déterminée par l'inclusion textuelle de blocs.

Il est important de noter que ces notions sont toutes statiques et comme elles sont assimilables à des préconditions, elles sont sujettes à la contravariance (Section 5.5.2, page 83). De plus, on observe que le typage statique et les droits d'accès forment des préconditions duales. Le typage statique permet de vérifier statiquement que le receveur a la capacité de traiter le message. Les droits d'accès y rajoutent la vérification de ce que l'émetteur est habilité à l'envoyer.

9. Mais cela ne veut pas dire que la syntaxe de C++ est souhaitable! Que l'appel soit statique ne concerne que le compilateur : le programmeur lui doit se contenter d'appeler la méthode de la super-classe, sans savoir où elle est et sans avoir à la nommer (principe d'anonymat de l'héritage).

L'encapsulation à la Smalltalk

Le principe est excessivement simple et met l'accent sur la séparation entre le soi et les autres :
— toutes les méthodes sont *publiques*, accessibles à tout le programme ;
— tous les attributs sont *privés*, d'usage réservé à `self`.

Il s'agit bien d'un droit *statique* puisque, même dans un langage à typage dynamique, le receveur courant est typé statiquement. On constate aussi que la classe elle-même (ou les instances « sœurs ») n'ont aucun droit particulier.

Clauses d'`export` d'Eiffel

Eiffel conserve une partie de l'encapsulation de Smalltalk en distinguant le receveur courant, tout en permettant une spécification très fine des classes autorisées :
— les accès aux attributs en écriture sont réservés à `Current` ;
— pour les méthodes et les accès aux attributs en lecture[10] le programmeur spécifie l'ensemble des classes qui peuvent y accéder, y compris leurs sous-classes ; en particulier,
— l'ensemble { `Any` } est équivalente à `public`,
— l'ensemble vide `{}` (ou { `None` }) restreint l'accès à `Current`,
— et la liste restreinte à la classe courante équivaut à `private` en Java.

Enfin, les clauses d'`export` peuvent être redéfinies de façon covariante et sont couvertes par la règle dite des *catcalls* (Section 5.5.2, page 83).

En Scala, les droits d'accès sont un mélange entre Eiffel et Java.

En Java

La protection se fait au moyen de différents mots-clés (de `public` à `private`, en passant par d'autres comme `protected`). Si `public` signifie bien « publique », `private` ne fait plus référence au receveur courant mais à la classe, y compris ses sous-classes en Java. L'essence de l'encapsulation est perdue et on retrouve une imbrication de blocs implicite : la sous-classe est censée être imbriquée dans la super-classe. Cette vision est à la base de l'interprétation de l'héritage dans ces langages.

De son côté, `protected` donne des droits d'accès à la totalité du *package* de la classe, ainsi qu'à ses sous-classes.

C♯ est assez proche de Java sur ce point. A part `public`, le sens précis de ces mots-clés diffère suivant les langages.

En C++

C++ reprend les trois mots-clés de Java et y ajoute un quatrième `friend` qui ressemble un peu à ce que propose Eiffel en la matière. Dans une classe, `friend` permet de désigner une autre classe ou une fonction, éventuellement méthode d'une autre classe, pour lui donner les droits d'accès.

Ce mot-clé diffère de trois façons des clauses d'export d'Eiffel :
— faire d'une autre classe un `friend` n'a pas d'effet sur les sous-classes ;
— il n'est pas possible de restreindre le mot-clé à une propriété particulière de la classe qui donne des droits ;
— en revanche, il est possible de ne donner des droits qu'à une fonction particulière.

Héritage protégé de C++. C++ y rajoute en plus une protection de l'héritage : le chapitre 5 ne s'applique sans problème qu'à un héritage `public`. Un héritage `private` revient à faire de l'héritage d'implémentation, sans accepter la substituabilité. Cela revient en fait à se servir de l'héritage pour faire de l'agrégation.

6.8.2 Accès en lecture seule

Un système classique de droits d'accès consiste à distinguer les accès en lecture et en écriture. On souhaite alors que les accès en écriture à certaines données soient réservés à certaines parties du code, tout en laissant des accès en lecture à d'autres parties.

Tous les langages permettent une protection relative, mais aucun n'offre une protection effective et absolue.

Voir aussi Section 8.3.

10. Qui ne sont pas sémantiquement très différents : en Eiffel, une méthode sans paramètre peut être redéfini par un attribut.

Attributs immutables. Les langages disposent souvent de mots-clés pour indiquer qu'un attribut est mutable ou immutable : `var` et `val` en SCALA, `const` en C++ ou C♯, `readonly` en C♯, `final` en JAVA, etc.

Mais cela ne répond pas vraiment à la question de rendre momentanément un objet immutable dans une partie du code. Il s'agit, en effet, de propriétés universelles pour une classe, alors que l'on souhaiterait ici avoir une immutabilité provisoire et locale : dans un contexte de *multi-thread*, on peut imaginer que le même objet soit mutable dans un *thread* et immutable dans un autre *thread*.

Interface immutable. Une première solution consiste à distinguer la partie immutable de l'interface d'une classe, de sa partie mutable : ainsi par exemple, la classe `ImmutableStack` aura les méthodes `isEmpty` et `top`, que sa sous-classe `Stack` complètera avec `pop` et `push`.

On créera une pile avec `new Stack`, mais on l'exportera sous le type `ImmutableStack` aux parties du code qui n'ont un accès qu'en lecture.

Cependant, cette protection est illusoire car le code utilisateur peut caster la valeur d'`ImmutableStack` à `Stack`, et cela marchera toujours.

Jokers avec la généricité. On pourra faire quelque-chose de similaire avec les classes paramétrées (Chapitre 7) avec la notion de *joker* introduite par JAVA. La définition de `ImmutableStack` est alors inutile, et on pourra obtenir quelque-chose d'assez proche avec `Stack⟨? extends Object⟩`, qui représente toutes les piles, mais dont le type formel n'est utilisable qu'en position covariante (Section 7.4.3). Cependant un cast vers `Stack` reste possible. De plus, la restriction à l'interface covariante de la classe laisse les droits sur la méthode `pop` qui n'est pas en lecture seule.

En C++. Le mot-clé `const` permet d'exprimer quelque-chose d'assez proche de la lecture seule, mais un cast reste aussi possible, sous la forme de `const_cast`.

En EIFFEL. Par construction, le droit d'accès en écriture aux attributs d'un objet est réservé à cet objet lui-même (`Current`). Mais les accesseurs en écriture ne sont pas spécifiquement protégés, de même que toute autre méthode modifiant l'état de l'objet. Par ailleurs, la protection en écriture ne doit pas être uniforme pour toutes les instances d'une classe : il s'agit souvent de protéger uniquement certaines.

Mots-clés `immutable` et `mutate`

Une proposition de type immutable reposerait sur les deux mots-clés `immutable` et `mutate`.

- Tout type `T` existe sous une forme normale, et sous la forme `immutable T`, le premier étant un sous-type du second.
- Chaque méthode est déclarée ou détectée comme *mutagène* ou pas, avec le mot-clé `mutate` : seule une méthode mutagène peut modifier l'état de l'objet courant ou appeler une méthode mutagène sur l'objet courant. Ce statut est héritable et immutable.
- Par principe, `immutable T` a pour interface les méthodes non mutagènes de `T`.
- Dans une méthode mutagène, `self: T`, mais dans une méthode non mutagène, `self: immutable T`.
- Enfin, la clé est que la hiérarchie de types mutables et la hiérarchie de types `immutable`, qui sont isomorphes, sont parfaitement étanche ; on a bien `T <: immutable T`, mais il est impossible de faire un cast d'un type immutable vers un type mutable.

Le dernier point est crucial. C'était en effet la faille principale des propositions précédentes, qui était incontournable puisqu'il suffisait de passer par l'intermédiaire du type `Object`. Avec `immutable` la violation de la protection n'est plus possible.

Cette proposition est en attente de preuve ou de réalisation.

6.8.3 Surcharge statique et visibilité

Normalement, la visibilité ne devrait pas avoir d'impact sur un programme, au sens où le même programme devrait se comporter de la même manière avec différents niveaux de visibilité, tant que le compilateur l'accepte. L'existence de la surcharge statique rend néanmoins cette propriété illusoire.

Aussi bien en C++ que JAVA, la surcharge statique est résolue, comme décrit à la section 6.3, mais en se restreignant aux méthodes (propriétés globales) qui sont visibles. Etant donné un programme qui compile et s'exécute normalement on peut obtenir l'un des effets contre-intuitifs suivants, en accroissant la visibilité d'une méthode :

- le programme ne compile plus parce qu'une ambiguïté de surcharge apparaît ;

— le programme ne compile plus parce que la méthode précédemment appelée devient masquée par la méthode qui devient visible (en C++ seulement) ;
— le programme compile mais son comportement change parce que la méthode qui devient visible est plus spécifique que la méthode précédemment appelée.

Le dernier cas est bien sûr particulièrement vicieux : le programmeur peut mettre beaucoup de temps à s'en rendre compte.

Par ailleurs, alors qu'en JAVA les droits d'accès ne peuvent pas être redéfinis, ce n'est pas le cas en C++. On peut se servir de ce trait pour améliorer la simulation de la covariance (Section 6.4), en rendant `private` la méthode redéfinie qui fait le *cast*. Cela aura pour effet d'imposer l'usage des bons types statiques.

6.8.4 Aspects méthodologiques

Dans le principe, les droits d'accès sont facultatifs, et la transformation d'un programme correct obtenue par suppression de toute protection, en rendant tout publique, ne devrait pas modifier le comportement du programme. Ce n'est que partiellement vrai à cause de divers défauts de spécification des langages — dont la question de la surcharge statique discutée plus haut —, mais cette position de principe conduit à s'interroger sur l'utilité des droits d'accès.

Deux arguments généraux peuvent être avancés pour justifier, voire recommander, l'usage de droits d'accès restrictifs :

— la hantise de tous les développeurs ou mainteneurs est que leurs applications se tranforment en plat de spaghetti[11], où tout est en relation avec tout ; les protections constituent un moyen de réduire cette complexité ;
— de nombreux éléments des programmes objet peuvent être considérés comme des *frameworks*, dont les protocoles cachés reposent sur des méthodes essentiellement publiques qui encapsulent des appels à d'autres méthodes qui sont essentiellement privées et ne doivent pas être utilisées en dehors du protocole implémenté par la méthode publique ; la meilleure façon d'assurer que ces méthodes privées ne seront pas accédées hors de propos est qu'elles ne soient pas accessibles.

Exemple. Le protocole d'implémentation des associations UML entre deux classes A et B (Section 6.5) repose sur des méthodes des deux classes, qui doivent pouvoir s'appeler réciproquement, donc `private` serait inapproprié. Mais il est vraisemblable que certaines ne devront pas être appelées en dehors du protocole, donc `public` serait aussi inapproprié. Les clauses d'export d'EIFFEL sont certainement la meilleure solution. En C++ on pourra utiliser `friend`, mais en JAVA on ne pourra pas faire plus précis que `protected`.

6.9 Construction et initialisation d'objets

L'envoi de message est la principale abstraction procédurale des langages à objets mais ce n'est pas la seule. En effet, dans un langage non réflexif, la construction et l'initialisation d'objets sont en général assurées par un autre mécanisme procédural.

6.9.1 Principe

La construction d'objets est l'un des rôles clés de la classe, par lequel elle crée un objet tout en établissant le lien d'instanciation entre cet objet et la classe. Le mécanisme est parfaitement décrit dans le modèle réflexif d'OBJVLISP (Chapitre 11) mais il est plus obscur dans un langage à typage statique comme C++ ou JAVA. Dans ces langages, la création d'objets repose sur la notion de *constructeur* qui est un faux-ami, puisqu'il ne s'agit que d'initialisation.

Dans un langage sans méta-niveau, la construction d'objet ne peut pas reposer sur un envoi de message puisqu'il n'y a pas de receveur : l'objectif est justement d'en créer un. La construction d'objet se fait donc avec une syntaxe spécifique, par exemple avec le mot-clé `new`.

```
new C(...)
```

Dans cette syntaxe, le rôle de construction proprement dite est joué par `new C`, et son effet résulte en un objet normalement constitué mais dont les attributs sont non initialisés.

11. Le terme savant pour le plat de spaghetti serait la *clique* de la théorie des graphes.

Tous les langages proposent, en plus, des fonctions d'initialisation, qui sont malheureusement nommées *constructeur* en C++, Java, C♯, etc. L'expression précédente provoque ainsi l'appel du constructeur `C(...)` sur l'objet créé par `new C`, afin de l'initialiser. La syntaxe n'explicite pas le nom de la fonction car c'est le même que celui de la classe.

De plus, dans ces langages, le constructeur (c'est-à-dire la fonction d'initialisation) n'est pas une méthode, mais une simple fonction statique, et il n'y a pas de sélection à part celle qui concerne la surcharge statique.

Dans un langage comme Eiffel, le mot-clé `create` remplace `new`, mais l'appel de fonction est explicite. L'exemple d'Eiffel ou Prm montre cependant que les fonctions d'initialisation peuvent très bien être des méthodes « normales » d'initialisation (voir la discussion sur les classes paramétrées, Section 7.3.2, page 125). Cependant, dans ces langages, si les constructeurs sont des méthodes normales, héritables et redéfinissables, ce n'est pas le cas pour leur rôle d'initialiseur, qui doit être spécifié explicitement dans chaque classe.

Combinaison des constructeurs. Enfin, les constructeurs doivent en général être combinés, chaque classe étant responsable de l'initialisation des attributs qu'elle introduit. Cette combinaison peut être implicite ou explicite. Dans ce dernier cas, elle peut s'effectuer par un appel explicite dans le code, par exemple un appel à `super`, ou, plus souvent par une déclaration, le compilateur se réservant la façon particulière de générer l'appel.

Par ailleurs, la combinaison des constructeurs peut se faire à l'intérieur de la classe considérée (par exemple un constructeur à deux paramètres appelle un constructeur à un paramètre), ou entre la classe et sa super-classe.

Pattern *Factory*. Bien que la construction d'objet ne repose pas sur un envoi de message, il est souvent utile de pouvoir créer des objets de types différents par envoi de message à un objet préexistant : c'est le rôle du *pattern Factory*.

Initialisation des attributs

Le rôle principal de l'initialisation d'un objet est d'initialiser ses attributs. Lorsque l'initialisation d'un attribut est indépendante du contexte de création de l'objet, cette initialisation peut se faire de façon déclarative, avec une syntaxe de type

```
class C {
    foo : T = new U()
    ...
}
```

Une telle initialisation est possible dans à peu près tous les langages. Elle correspond au mot-clé `:initform` de Clos (Chapitre 12) et assure que l'attribut sera initialisé ainsi dans tous les constructeurs qui ne l'initialisent pas explicitement autrement [12].

En revanche, la plupart des langages considérés ici ne permettent pas de redéfinir, dans une sous-classe, l'initialisation déclarative d'un attribut : en Java, C++, C♯ ou Scala (pour ne citer qu'eux), ce serait considéré comme une surcharge statique, donc la déclaration d'un nouvel attribut. Le langage Nit permet cette redéfinition, avec le mot-clé `redef`, de même qu'Eiffel (à vérifier).

Lorsque c'est possible, une initialisation déclarative est toujours préférable.

6.9.2 Attributs non initialisés

Le rôle d'un initialiseur est d'initialiser les attributs. Lorsque ces attributs sont des références sur des objets (et non pas une valeur immédiate comme un entier), l'absence d'initialisation explicite entraîne un risque d'erreur. La valeur de l'attribut ne peut pas être une séquence aléatoire de 0 et de 1 — celle laissée par la dernière utilisation du mot mémoire considéré — car cela entraînerait des erreurs aléatoires, non reproductibles. L'attribut est donc initialisé implicitement par une valeur distinguée (`void`, `null`, `nil`, suivant les langages). Cette valeur distinguée peut être considérée comme l'unique instance du *type absurde* $\bot$, sous-type de tous les autres types. Un appel de méthode ou un accès aux attributs de cette valeur distinguée provoque une erreur qui peut être assimilée à une erreur de type.

12. Contrairement à la spécification de `:initform`, qui est très précise, les différents langages ne sont pas forcément très précis sur ce qu'il se passe si l'attribut `foo` est aussi initialisé explicitement dans un constructeur, en plus de l'initialisation déclarative. Faire des tests.

C'est la présence ou le risque d'attributs non initialisés qui entraîne une spécification particulière des constructeurs. En effet, si le constructeur d'une classe A était héritable par sa sous-classe B, il ne pourrait pas initialiser les attributs introduits par B.

On verra au chapitre 7 que le problème se pose de façon aiguë lorsque l'on souhaite instancier un type formel ou un type virtuel.

6.9.3 En JAVA

Le constructeur a le nom de sa classe et n'est pas une méthode "normale", ni virtuelle, ni statique, car il s'agit d'un appel statique, mais avec un paramètre `this`. Le nom du constructeur étant fixé, une surcharge statique s'impose dans le cas où on désire plusieurs constructeurs. On combine les constructeurs en utilisant les mots-clés `super` pour appeler celui de la super-classe et `this` pour appeler un autre constructeur de la classe courante. Le programmeur doit impérativement faire figurer cet appel comme première instruction du corps du constructeur — c'est donc une déclaration plus qu'une instruction —, ce qui impose de fait une *linéarisation* des constructeurs, et interdit d'appeler à la fois `this` et `super`.

6.9.4 En C++

A quelques gros détails près, les constructeurs sont spécifiés comme en JAVA, par des méthodes du nom de la classes.

Combinaison des constructeurs. Pour la combinaison, ils sont implicitement linéarisés (Section 4.5.1), et les constructeurs ne sont pas explicitement appelés, mais sont déclarés dans l'en-tête du constructeur. On a donc une syntaxe de type

```
C(T t, U u) : B(t,u), A(t) {...}
```

Entre le corps entre accolades et le ' :' figurent les constructeurs à appeler pour la combinaison. Si l'on suppose que `C` $\prec_d$ `B` $\prec_d$ `A`, on aura les 3 définitions successives :

```
A(T t) { ... }
B(T t, U u) : A(t) { ... }
C(T t, U u) : B(t,u), A(t) {...}
```

On aurait pu imaginer que cette déclaration soit faite dans le fichier d'en-tête (`.h`), et que le code du corps soit réservé au fichier `.cpp`. Le compilateur n'a pas l'air d'accepter cela, ce qui explique pourquoi il est nécessaire de redéclarer les constructeurs des super-classes indirectes, `A(t)` pour `C`. En fait, cette déclaration est nécessaire si l'héritage est virtuel, mais il ne faut surtout pas la faire si l'héritage est non-virtuel. On peut expliquer cela par le fait que, en cas d'héritage non-virtuel, l'implémentation de `B` intègre celle de `A` et que cela est vrai aussi pour le constructeur de `B`. Du coup, le constructeur de `C` ne doit plus mentionner la façon d'initialiser `A`. En revanche, si l'héritage est virtuel, les sous-objets `A`, `B` et `C` sont indépendants et le constructeur de `C` doit spécifier comment il veut initialiser son sous-objet `A`.

Constructeur dit "par défaut". Un piège peut se cacher dans le paysage. Pour certaines classes, un constructeur sans paramètres, dit "par défaut", peut être défini. Si ce constructeur est défini, par exemple dans la classe `A`, il sera appelé par défaut par les constructeurs des sous-classes, et la non-déclaration de `A(t)` dans le constructeur de `C` ne provoquera pas d'erreur de compilation. Comme un constructeur par défaut ne peut être défini dans la sous-classe que s'il est déjà défini dans la super-classe, il est souvent considéré que c'est un "bon" style de programmation de toujours définir un constructeur par défaut, avec un corps vide. Du fait que cette définition peut conduire à masquer une erreur de compilation, il semble que cette définition systématique de constructeurs sans paramètres ni corps soit plutôt une mauvaise habitude.

Liaison tardive dans le constructeur. Par ailleurs, une limitation importante est que, non seulement la liaison tardive ne s'applique pas au constructeur que l'on appelle, mais elle ne s'applique pas aux méthodes qui sont appelées par le constructeur sur l'objet en cours de construction (`this`). C'est une grosse erreur de spécification dont la raison n'est pas claire mais qui fait qu'il ne faut pas appeler une méthode non-statique sur `this` depuis le code des constructeurs.

Constructeur et conversion implicite. Par défaut, lorsqu'une classe `C` a un constructeur avec un unique paramètre de type `T`, ce constructeur peut se servir de conversion implicite lorsqu'une valeur de type `T` est passée là où l'on attend une valeur de type `C`. Ainsi dans le code

```
C c; T t;
c=t;
```

le compilateur interprétera la dernière instruction comme `c=new C(t)`. Pour empêcher ce comportement déraisonnable, il faut annoter le constructeur de `C` avec le mot-clé `explicit`.

Destructeurs. Comme C++ n'est pas doté, en standard, de *garbage collector*, les classes définissent aussi des destructeurs, qui sont combinés de façon similaire.

6.9.5 En Eiffel

Contrairement à C++ et à Java, les constructeurs Eiffel sont de vraies méthodes, de nom quelconque, dont le rôle de constructeur doit être déclaré dans chaque classe. Ce rôle ne s'hérite pas.

Par ailleurs, l'invariant de classe est vérifié en fin d'initialisation.

6.9.6 Nit et le mot-clé `nullable`

Nit reprend pour partie le principe des méthodes d'initialisation d'Eiffel. Il s'y greffe différents mots-clés qui peuvent éviter au programmeur de déclarer explicitement la méthode d'initialisation. Nous décrivons juste, ici, un trait original du langage.

Il y a eu des tentatives pour intégrer la problématique des variables non initialisées dans le système de types des langages. En Nit (Section 10.4), le mot-clé `nullable` dans une annotation de type indique que la valeur considérée peut être nulle. Pour tout type `T`, on a `T <: nullable T` : on peut donc faire une affectation dans un sens. Dans l'autre sens, un *cast* (c'est-à-dire une comparaison à `null`) est nécessaire. Le type `nullable T` ne dispose pas de l'interface de `T`, mais uniquement de l'interface de `null`, qui peut se limiter à quelques méthodes générales comme `equals` ou `toString`.

Dans un tel système, le statut des attributs non-initialisés est particulier dans les constructeurs : un attribut typé par `T` peut être momentanément égal à `null`, mais il doit impérativement être initialisé à la sortie du constructeur.

6.9.7 Spécificité des initialiseurs

Etant donné une classe `C`, muni d'un certain constructeur `C(x,y,z)`, et d'une méthode `init(x,y,z)` sans rôle déclaré d'initialisation, mais contenant le même code que `C(x,y,z)`. Quelle différence peut-il alors y avoir entre

1. l'appel de `new C(x,y,z)`,
2. l'appel de `new C().init(x,y,z)`, où `C()` serait le constructeur dit "par défaut" qui ne ferait rien,
3. l'appel de `new C(x,y,z)` si `C(x,y,z)` ne contenait, cette fois, que l'appel à `init(x,y,z)` ?

La différence dépend bien entendu des langages.

— en C++, si l'on fait `new D` au lieu de `new C`, avec `D` sous-classe de `C`, un appel de méthode virtuelle sur `this` dans `C(x,y,z)` ne provoque pas de liaison tardive : c'est la méthode du type statique (`C`) qui est appelée statiquement ; en revanche, depuis `init(x,y,z)`, le même appel se ferait sur le type dynamique de `this` (`D`) ; dans le cas 3, l'appel de `init` serait statique, donc ne considèrerait pas son éventuelle redéfinition dans `D` ;

— en Nit, tous les attributs qui ne sont pas `nullable` doivent avoir été initialisés à la sortie du constructeur : donc, dans `init`, il n'est possible d'initialiser que des attributs `nullable` ; le cas 3 présenterait la même limitation ;

— dans un langage avec des assertions comme Eiffel (Section 5.5.3, page 83), on peut imaginer que l'invariant de classe est vérifié en sortant de l'initialiseur.

Par ailleurs, dans tous les langages avec des "constructeurs" qui portent le nom de la classe, la combinaison des constructeurs est particulière et partiellement implicite. Par exemple, en C++, les constructeurs sont combinés par linéarisation, donc sans risque de double évaluation, contrairement aux méthodes normales comme `init` qui pourraient en provoquer en cas d'héritage multiple (voir Section 4.3.1, page 46).

6.9.8 Constructeurs et généricité ou méta-programmation

La question de l'initialisation des instances se pose de façon cruciale dans les cas où la classe à instancier n'est pas connue statiquement. On peut distinguer deux cas, lorsque la classe à instancier est un paramètre formel de type, ou un type virtuel (Chapitre 7), ou lorsque l'instanciation se fait par méta-programmation (Chapitre 13).

Dans les deux cas, on se retrouve avec un type inconnu borné supérieurement, et les seuls constructeurs connus sont ceux de la borne.

En pratique, les langages considérés interdisent l'instanciation d'un paramètre formel de type (par exemple, JAVA ou SCALA, à cause de l'effacement de type), ou l'autorise de façon restreinte avec un constructeur sans paramètre (C♯, avec l'annotation `new T`).

Dans le cas de la méta-programmation, la méthode `newInstance` de JAVA ne prend pas de paramètre et appelle le constructeur sans paramètres. Si l'on souhaite utiliser un constructeur différent, il faut alors le rechercher dans la classe et se servir ensuite de la méthode `newInstance` des constructeurs.

6.10 Fonctions et variables statiques

Par ailleurs, beaucoup de langages proposent des méthodes ou fonctions « statiques », car non sujettes à la liaison tardive.

Ces fonctions ou variables diffèrent essentiellement suivant le fait que le paramètre `this` est lié ou pas. Mais dans tous les cas elles ne relèvent pas du modèle objet car elles ne vérifient ni l'Axiome 2.2, page 21, ni le Méta-Axiome 5.1, page 71. De façon générale, la relation entre ces propriétés statiques et les classes où elles sont définies relèvent des mêmes problématiques de rangement et de protection (Section 6.8) que les *packages* JAVA (Section 2.4).

Il n'y a bien entendu aucune nécessité d'avoir des fonctions statiques (sans liaison tardive) lorsqu'elles ont un receveur : ce n'est au mieux qu'une question d'efficacité, l'appel statique étant plus efficace que la liaison tardive (Chapitre 9), mais un bon système d'exécution doit optimiser ça dans la quasi-totalité des cas. En revanche, des fonctions statiques sans receveur ou des variables statiques constituent une fonctionnalité difficilement remplaçable.

En C++. Les fonctions dites non-`virtual` s'utilisent exactement comme des méthodes, avec un receveur explicite qui est lié comme d'habitude à `this`, mais leur sélection est purement statique, basée sur le type statique du receveur.

Il est par ailleurs possible de définir des fonctions extérieures aux classes, qui s'utilisent comme en C.

En JAVA. Les fonctions et les variables `static` sont des fonctions et variables « ordinaires », non objet, définies dans l'espace de noms d'une classe. Il n'y a pas de receveur et `this` n'est pas lié. Ces propriétés statiques sont rangées par commodité dans une classe, mais elles pourraient tout aussi bien être rangées dans une autre, et il n'y a pas de raison profonde de les ranger dans la même classe. Bien entendu, l'accessibilité de ces propriétés, losqu'elles ne sont pas `public`, va être restreinte par la classe et le package où elles sont rangées.

En C♯. Le mot-clé `virtual` a le même rôle qu'en C++. Une méthode déclarée sans ce mot-clé est donc une fonction statique au sens où elle n'est pas appelée par liaison tardive, mais c'est une fonction avec un receveur, comme une méthode normale. Le mot-clé `static` se comporte, de son côté comme en JAVA (avec une différence subtile pour les classes paramétrées, voir Chapitre 7).

En SCALA. Les méthodes et variables statiques n'existent pas en tant que telles, mais elles sont regroupées dans des *singletons* (classes avec une seule instance) déclarés avec le mot-clé `object` à la place de `class`. C'est donc le singleton qui constitue la véritable donnée statique.

Cela revient juste à ranger les méthodes et variables statiques dans des entités dédiées.

En NIT. Aucun trait du langage ne permet de définir de propriétés statiques. Cependant il existe un objet système, instance d'une classe `Sys` singleton de fait, et retourné par la méthode `sys` de la classe `Any` racine de la hiérarchie, donc accessible par n'importe quel objet. Par *raffinement de classe*, il est possible de définir, dans cette class `Sys`, de nouvelles propriétés qui joueront le rôle de propriétés statiques.

En Eiffel. Il n'y a pas non plus d'équivalent de `static` en Eiffel, qui propose plus ou moins l'équivalent de la classe `Sys` de Nit, sans le raffinement. S'ajoutent à cela les méthodes dites `once` qui ne sont activées qu'une seule fois, mémorisent leur premier résultat et le retournent lors des appels suivants : c'est donc une simulation des attributs statiques.

6.11 Programmation fonctionnelle, λ-expressions et fermetures

Il existe de nombreux paradigmes de programmation (impérative, logique, fonctionnelle,...) et les langages à objets sont de plus en plus syncrétiques.

6.11.1 Langage fonctionnel

Un langage peut être fonctionnel de différentes façons :
- par un style de programmation fonctionnel, en opposition à la programmation impérative, donc sans effet de bord ; on peut programmer ainsi, même en C ;
- par le fait qu'il existe un type `function` et donc que des fonctions puissent être des objets de première classe ;
- par le fait qu'il soit possible de prendre la valeur fonctionnelle (de type `function`) d'une fonction ou méthode ; en Common Lisp, c'est la forme syntaxique `function` qui assure cela ;
- par le fait qu'il soit possible de définir des λ-expressions, c'est-à-dire des fonctions anonymes qui peuvent être passées comme des valeurs ;
- par le fait que les λ-expressions sont de véritables fermetures qui capturent leur environnement lexical de définition qui restera accessible même après la fin de l'activation de la fonction courante qui définit justement cet environnement.

Un programme objet consiste à créer des objets et à les faire évoluer par effets de bord : en ce sens, il n'existe probablement pas de langage à objet purement fonctionnel. Au-delà de cet aspect, à part les extensions objets de langages fonctionnels, comme Clos ou OCaml, très peu de langages remplissent intégralement ce cahier des charges. Scala est l'un des rares.

6.11.2 Fonctions de première classe

Définir une fonction comme un objet de première classe dans un langage objet est assez facile pour le programmeur. Il suffit en gros de définir une classe qui comporte une méthode `apply` qui implémente la fonction dont on veut faire un objet de première classe. On pourra en faire une sous-classe d'une classe `function`, qui pourra être paramétrée par les types de paramètres et de retour, avec différentes classes suivant l'arité de la fonction.

Prendre la valeur fonctionnelle d'une fonction ou méthode est plus difficile puisqu'il faut une construction syntaxique particulière. Dans un contexte objet, cela se complique du fait que la fonction peut être considérée à deux niveaux, la méthode globale ou la méthode locale. Dans le premier cas, il faut appliquer la valeur fonctionnelle obtenue à une liste de paramètres qui doit inclure le receveur. Dans le second cas, la méthode locale a été sélectionnée à partir d'un receveur et la valeur fonctionnelle correspondante doit inclure ce receveur. C'est la solution retenue par C♯ et .Net avec la notion de *delegate*. Une troisième solution consistant à désigner une méthode locale pour l'appliquer à un receveur externe présenterait un risque de mauvais typage, puisque la méthode sélectionnée ne serait pas forcément celle qui aurait été sélectionnée pour ce receveur-là.

6.11.3 λ-expressions

Faire des fonctions anonymes nécessite une autre construction syntaxique, similaire au constructeur λ (`lambda`) de Lisp, qui introduit des variables et déclare le corps de la fonction.

Dans un premier temps, cette fonction va capturer les valeurs de variables figurant dans l'environnement courant. C++ 2011 propose une construction de ce type, dans lesquels le programmeur doit expliciter les variables de l'environnement qui figurent comme variables libres dans le corps de la fonction. Le compilateur pourrait bien sûr les trouver tout seul.

Si l'on ne dispose pas de λ-expressions, on peut les remplacer par des classes anonymes. Au lieu de :

```
l:Function<A:B>=Lambda(x: A){ .. } : B
```

on écrira

```
l:Function<A:B>=new Function<A:B> {apply(a:A):B { .. }}()
```

Les méthodes définies dans les classes anonymes de JAVA capturent bien les valeurs des variables de l'environnement.

6.11.4 Fermetures

Pour faire des fermetures, il ne faut pas capturer simplement les valeurs des variables de l'environnement, mais il faut capturer les liaisons variables-valeurs elles-mêmes.

Si l'on dispose de simples λ-expressions, il est très facile de faire cette capture à la main, en réifiant l'environnement à capturer. Soit le pseudo-code suivant :

```
{  x:T
   y:U
   z:V
   l:Lambda(x: A){ .. y .. z ..} : B
   foo(l)
   bar(l)
   y=..
   z=..
   .. }
```

que l'on peut comprendre comme suit : dans l'environnement des variables `{x, y, z}`, on définit une λ-expression qui prend un paramètre de type `A` et retourne une valeur de type `B`. Cette λ capture les variables `y` et `z` de l'environnement. La λ est passée au monde extérieur par les appels des fonctions `foo` et `bar`. Les variables `y` et `z` sont ensuite modifiées, soit directement dans la séquence initiale, soit dans la fermeture elle-même. Chaque modification doit avoir un effet sur l'environnement initial ainsi que sur l'environnement partagé par la fermeture dans les appels de `foo` et `bar`.

Si l'on ne dispose pas de véritables fermetures, on transformera alors le code de la façon suivante par réification de l'environnement :

```
{  x:T
   y:U
   z:V
   class Env{ y:U; z:V; Env(_y:U,_z:V){y=_y; z=_z}}
   e:Env = new Env(y,z)
   l:Function<A,B>=Lambda(x: A){ .. e.y .. e.z ..} : B
   foo(l)
   bar(l)
   e.y=..
   e.z=..
   .. }
```

dans lequel on a défini une classe `Env` qui possède des attributs `y` et `z`, dont on crée une instance. On remplace alors tous les accès aux variables par des accès aux attributs de l'environnement.

La syntaxe est bien sûr fantaisiste. Ce serait mieux de pouvoir faire des classes anonymes, mais pour des raisons de typage, ce serait difficile. Une alternative serait d'avoir des classes de tuples, ayant des noms prédéfinis pour leurs champs, par exemple `f1`, `f2`, etc.. On écrirait alors :

```
{  x:T
   y:U
   z:V
   e:Tuple2<U,V> = new Tuple2(y,z)
   l:Function<A,B>=Lambda(x: A){ .. e.f1 .. e.f2 ..} : B
   foo(l)
   bar(l)
   e.f1=..
   e.f2=..
   .. }
```

Une implémentaion régulière des fermetures pourrait simplement passer par cette transformation source-à-source, et c'est probablement ce que fait SCALA.

6.12 Exercices

EXERCICE 6.9. Vérifier le comportement de la sélection multiple en CLOS et retrouver les résultats des figures 6.3, 6.5, et 6.4. □

EXERCICE 6.10. Sur le même exemple, et sur les exemples d'héritage multiple (Figure 4.13, page 56) vérifier le comportement de `call-next-method` en CLOS. □

EXERCICE 6.11. Implémenter le modèle de graphes d'objets en CLOS en servant de la sélection multiple pour éviter les problèmes de type. □

EXERCICE 6.12. Vérifier le comportement des langages C++, JAVA, C♯, etc. et retrouver les résultats des figures 6.3, 6.5, et 6.4, page 95 qui démontrent qu'il s'agit bien de surcharge statique. □

EXERCICE 6.13. Vérifier le comportement de la redéfinition en EIFFEL et retrouver les résultats des figures 6.3, 6.5, et 6.4, page 95, qui démontrent qu'il ne s'agit pas de surcharge statique. Essayer de déclencher des erreurs de type à l'exécution dans le cas non sûr (Attention le comportement peut dépendre de l'implémentation, SMART EIFFEL vs. EIFFEL STUDIO, et s'éloigner de ce qui est décrit ici.) □

6.13 En savoir plus

Le seul langage courant avec sélection multiple a été longtemps CLOS, donc dans un contexte de typage dynamique. Malgré son intérêt, il n'y a pas eu jusqu'à récemment de langage qui implémente le modèle de Castagna (Section 6.2.2) — ce serait pourtant une extension raisonnable des langages à objets à typage statique, même si cela semble poser des problèmes de typage et de modularité dans le contexte de l'*hypothèse du monde ouvert*. Outre la proposition des *multi-méthodes encapsulées* — qu'il appelle *fonctions surchargées* ce qui prête à confusion — [Castagna, 1997] est une présentation formelle approfondie des problèmes de typage. Le langage FORTRESS est un langage récent, mais déjà abandonné, qui propose (proposait) des *multi-méthodes* dans un contexte de typage statique, en combinant élégamment les multi-méthodes encapsulées de Castagna, avec des multi-méthodes non-encapsulées, à la CLOS, pour résoudre le problème de sélection unique en monde ouvert que pose la règle de Castagna.

Par ailleurs,le seul langage *mainstream* en typage statique dont l'héritage multiple soit basé sur les linéarisations est SCALA. Son typage est invariant et le typage de `call-next-method` reste donc théorique. Finalement, il y a encore moins de langages, en typage statique, avec multi-méthodes encapsulées et linéarisations... Toutes ces combinaisons seraient pourtant intéressantes.

L'implémentation des associations UML offre encore d'autres possibilités : on peut passer par des types *paramétrés* ou *virtuels* (Chapitre 7). On peut même réifier l'association, mais cela ne fait sans doute que rajouter une nouvelle classe associée aux classes d'origine, sans changer les problèmes de typage sous-jacents.

Bibliographie

G. Castagna. *Object-oriented programming : a unified foundation*. Birkhaüser, 1997.

M.A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley, Reading, MA, US, 1990.

J. Gosling, B. Joy, and G. Steele. *The JAVA Language Specification*. Addison-Wesley, 1996.

7

Généricité

La *généricité*, aussi appelée *polymorphisme paramétrique*, consiste en la possibilité de paramétrer des entités de programme, en général par des types. Il s'agit de ce que l'on appelle des *templates*, en C++. On s'intéressera ici essentiellement aux cas où ce sont des classes (ou des types) qui sont paramétrées. Surtout, nous ne considérerons que le cas où le paramètre représente un type.

La généricité n'est pas un trait spécifiquement objet puisqu'elle est présente dans des langages comme Ada qui ont été longtemps purement procéduraux (même si Ada 95 a introduit dans le langage les bases de la programmation par objets et que la version 2005 a repris le système de typage, avec sous-typage multiple, de Java [Taft et al., 2006]). Il s'agit donc plutôt d'une fonctionnalité orthogonale, dont la combinaison avec les traits proprement objet offre toute une gamme de problèmes assez délicats.

On peut résumer les choses en disant que la généricité s'est révélée indispensable aux langages à objets à typage statique, mais qu'elle en constitue l'essentiel des difficultés.

7.1 Classes et types paramétrés

Le type (la classe) paramétré(e) est un type (ou une classe) normal(e), sauf qu'il est paramétré par un ou plusieurs paramètres formels qui peuvent être utilisés comme des types dans sa définition. Nous ne considérerons ici que le cas où ce paramètre formel désigne un type. On désigne souvent les types paramétrés sous le nom de types "d'ordre supérieur" (cf. note 9).

7.1.1 Type paramétré et instanciation

Type formel. Une classe paramétrée, notée $A\langle T\rangle$, est une classe dans laquelle le paramètre formel T est un *type formel*[1] qui peut jouer tous les rôles d'un type dans le code d'une classe normale : annotation de type (`x:T`), instanciation (`new T`), test de sous-typage (*cast*), ou paramètre dans un autre type paramétré $B\langle T\rangle$, par exemple pour l'instancier (`new B<T>`). On peut même a priori paramétrer un type paramétré, par exemple en utilisant $U\langle T\rangle$ dans le corps de $A\langle T, U\rangle$.

Un type formel T n'a de sens que dans le contexte de la classe paramétrée $A\langle T\rangle$ qui l'introduit. Comme pour tous les paramètres, cette introduction sous-entend un mot-clé similaire à `lambda` en Lisp ou aux quantificateurs en logique. En dehors de ce contexte de définition de classe par `class A<T>` le paramètre formel T n'a aucun sens. On peut donc considérer que c'est le mot-clé `class` (ou ses équivalents) qui joue le rôle de `lambda`.

De plus, comme pour tous les paramètres positionnels, les types formels sont toujours à un renommage près. On peut définir une classe $A\langle T\rangle$, puis définir $B\langle U\rangle$ comme sous-classe de $A\langle U\rangle$: le paramètre est noté T dans A mais U dans B, mais c'est "le même". En fait, les types formels (ainsi que les types virtuels que nous verrons plus loin) sont des propriétés au sens du méta-modèle du Chapitre 3, avec une différence principale : ils ne sont pas réellement nommés mais passé de façon positionnelle, comme le sont en général les paramètres des fonctions.

Type concret. Par opposition, on dira qu'un type est concret[2] s'il n'est pas formel et ne contient pas de type formel.

1. Cette appellation relève de l'abus de langage. En anglais on dirait *formal type parameter*, puisqu'il s'agit d'un *formal parameter* qui représente un type. Isoler *formal type* n'est pas forcément une tès bonne idée, et si l'on veut raccourcir, il vaudrait mieux conserver *type parameter*. Cela étant, nous utiliserons *type formel*.

2. Encore une appellation qui n'est pas trop heureuse, puisque ce terme sert en général à désigner l'ensemble des types dynamiques possibles à l'exécution d'un programme.

Les langages ne font en général pas de différence syntaxique entre les types formels et les types concrets : au programmeur de faire en sorte de ne pas créer de conflit de noms et rendre explicite le fait qu'il s'agit d'un type formel tout en ne masquant pas son sens. Dans la suite, nous utiliserons le début de l'alphabet pour les types concrets et la fin pour les types formels.

Instanciation d'un type paramétré. Un type paramétré est donc un générateur de types (classes) — on peut le voir comme une fonction des types vers les types — par instanciation de son(ses) type(s) formel(s), c'est-à-dire par le remplacement du type formel par un type concret. On appelle cela l'*instanciation*[3] du type paramétré.

Cette instanciation revient à substituer un type concret, par exemple C, au type formel T, partout dans le code de la classe paramétrée $A\langle T\rangle$. On obtient alors un type concret $A\langle C\rangle$. Normalement, la sémantique de la classe ainsi obtenue est exactement celle qu'aurait eu le programme résultant d'une substitution textuelle de T par C. On verra qu'il y a néanmoins quelques restrictions.

Comme il n'y a pas de termes pour distinguer $A\langle T\rangle$ de $A\langle C\rangle$, on parlera pour ce dernier d'*instance générique* de type (ou classe) paramétré(e).

Classe ou type ? Qu'est-ce qui fait finalement la différence entre une classe et un type ? Deux choses : les instances, c'est-à-dire les objets que la classe crée, et le code qui constitue la définition de la classe. Curieusement, la généricité va disperser ces deux éléments dans des entités distinctes.

Une classe crée des objets, ses instances, et implémente toutes ses méthodes, alors qu'un type se réduit à des signatures de méthodes. Bien sûr, comme d'habitude, la frontière n'est pas si tranchée : entre les deux, les *classes abstraites* implémentent des méthodes (pas forcément toutes) mais ne permettent pas de créer d'instances. En revanche, il n'existe pas d'entités qui pourraient créer des instances sans implémenter toutes leurs méthodes.

Dans ce sens, une classe paramétrée non concrète ($A\langle T\rangle$) n'est pas une classe : elle ne permet pas de créer directement des objets et ses instances sont en réalité des classes[4]. En revanche, son instanciation $A\langle C\rangle$, où C est un type concret, est a priori une classe, en tout cas si le code de $A\langle T\rangle$ l'a prévu. Si l'on déclare que $A\langle T\rangle$ est abstraite (par `abstract class A⟨T⟩`), cela ne peut concerner que $A\langle C\rangle$ ou plutôt toutes les instances possibles de $A\langle T\rangle$. Il faudra donc définir des sous-classes concrètes de $A\langle C\rangle$ pour pouvoir s'en servir, mais on verra qu'il n'y a souvent qu'une unique sous-classe directe.

Le code `new A⟨C⟩`, où C est concret, va produire un objet instance de $A\langle C\rangle$. Le code `new A⟨T⟩`, où T est formel, n'a de sens que dans le contexte de la définition d'une classe paramétrée par T, par exemple $B\langle T\rangle$ (mais B peut être A). Ce code produira aussi un objet instance de $A\langle C\rangle$, mais dans le contexte d'un objet instance de $B\langle C\rangle$, après application de la méthode idoine. Cela n'a en tout cas aucun sens de considérer que cet objet est une instance de $A\langle T\rangle$.

Le paradoxe de la généricité réside ainsi dans le fait que le code de la classe est détenu par $A\langle T\rangle$, alors que c'est $A\langle C\rangle$ qui a la charge de créer les instances.

7.1.2 Généricité bornée.

En toute généralité, le code de la classe paramétrée peut se servir d'opérations spécifiques de son type formel, ce qui n'est pas le cas des classes paramétrées les plus usuelles comme les collections, piles, files, etc.

Les instanciations 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. On définira ainsi la classe paramétrée $A\langle T <: B\rangle$ dans lequel le type formel T est contraint à être un sous-type d'un type concret B, la borne. Il faut comprendre cela de la façon suivante : $\forall T <: B$, $A\langle T\rangle$ est une classe bien formée et le compilateur est capable de le vérifier sur $A\langle B\rangle$. Par défaut, la borne est la racine de la hiérarchie de classe, `Object` en Java et `Any` en Eiffel.

Par exemple, le type « ensemble ordonné »[5] `OrderedSet` pourrait être typé comme `OrderedSet⟨T <: Comparable⟩`, où le type `Comparable` doit posséder un opérateur de comparaison (ordre).

3. Instanciation est un terme assez surchargé : on peut l'appliquer aux classes pour parler de la création d'objets mais on l'utilise aussi classiquement pour parler de la valuation d'une variable, en Prolog par exemple. Ce deuxième sens produit ici un effet de l'ordre du premier...

4. D'où l'analogie très forte qu'il y a entre classes paramétrées et méta-classes (Chapitre 11).

5. 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).

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

Remarque 7.1. On utilise ici le symbole $<:$ pour exprimer la borne mais un langage particulier l'exprimera avec ses propres mots-clés : `implements` et `extends` en JAVA par exemple.

7.1.3 Type paramétré et sous-typage.

Les relations entre les types paramétrés et le sous-typage ne sont guère plus simples.

Relation entre la classe paramétrée et ses instances.

Notons d'abord qu'il n'y a en règle générale aucune relation de sous-typage ou de spécialisation entre $A\langle C\rangle$ et $A\langle T\rangle$. 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. Dans ce dernier cas, si T est borné par B, il peut néanmoins y avoir moyen de se ramener à l'éventuel sous-typage entre $A\langle C\rangle$ et $A\langle B\rangle$, en se ramenant à la quantification universelle rendue possible par la borne : y a-t-il une relation de sous-typage entre $A\langle C\rangle$ et $A\langle T\rangle$, $\forall T <: B$?

Relation entre les instances d'une même classe paramétrée.

Considérons maintenant deux instances $A\langle C\rangle$ et $A\langle D\rangle$ d'une même classe paramétrée $A\langle T <: B\rangle$. Supposons que $D <: C <: B$. Pour fixer les esprits, on peut remplacer A par `Stack`, C par `Vaisselle` et D par `Assiette`. Si l'on examine la chose sur la plan de la spécialisation, il est évident qu'une pile d'assiettes est bien une pile de vaisselle. Les instances dont les types concrets sont en relation de spécialisation, sont elles-mêmes en relation de spécialisation.

En revanche, les types paramétrés posent les mêmes problèmes de sous-typage à cause de la règle de contravariance. Développons l'exemple de la pile, dont le schéma de définition est le suivant :

```
class Stack⟨T⟩{
      pop() : T {...}
      push(T) {...}}
```

Les deux opérations `pop` et `push` de `Stack⟨T⟩` doivent respectivement retourner une valeur et prendre un paramètre de type `T`. Les deux instanciations ont alors les définitions suivantes :

```
class Stack⟨Vaisselle⟩{
      pop() : Vaisselle {...}
      push(Vaisselle) {...}}
```

```
class Stack⟨Assiette⟩{
      pop() : Assiette {...}
      push(Assiette) {...}}
```

Si `Stack⟨Assiette⟩` était un sous-type de `Stack⟨Vaisselle⟩`, le type du paramètre de `push` varierait de façon strictement covariante, ce qui est contraire à la règle de contravariance. Dans un contexte de typage sûr, les 2 types paramétrés sont donc incomparables.

En général, la règle de typage

$$\frac{A\langle T <: B\rangle \quad C' <: C <: B}{A\langle C'\rangle <: A\langle C\rangle} \tag{7.1}$$

est fausse, mais on verra, en Section 7.4, dans quel cas cette règle, ou une règle similaire, peut être sûre.

Remarque 7.2. Cette règle reprend la syntaxe de la *déduction naturelle*, un formalisme logique [Gochet and Gribomont, 1991] beaucoup utilisé pour formaliser les langages de programmation et les types. On la comprendra de la façon suivante. Au-dessus de la barre de fraction, on a une conjonction de prémisses, dont la première affirme que la définition de A est correcte. Sous la barre, une conjonction de conclusions.

Langages covariants

Bien que cette règle de covariance entre instances génériques soit fausse dans le cas général, certains langages l'acceptent. Il s'agit, en général, de ceux qui acceptent la redéfiniton covariante des types de paramètres, EIFFEL en premier lieu. C'est aussi le cas des langages DART et NIT. Les raisons mises en avant pour justifier cette infraction sont de plusieurs types :

— notre intuition aristotélicienne est confortée par le fait qu'un `Set⟨Cat⟩` soit une spécialisation de `Set⟨Animal⟩` ;
— les règles de variance (Section 7.4) sont bien compliquées pour le programmeur « moyen » ;
— des méthodes comme `sort(Collection⟨Object⟩)` sont sûres et très simples dans un cadre covariant : en effet, bien que faisant des lectures et écritures, cette méthode ne fait que réarranger le contenu de la collection, et elle est donc sûre même en typage dynamique. Dans un cadre invariant, il faudrait la paramétrer, sous la forme `⟨T <: Object⟩ sort(Collection⟨T⟩)`.

7.1.4 Spécialisation de types paramétrés

Spécialisation. 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, la spécialisation de $A\langle T <: B\rangle$ par $A'\langle T <: B'\rangle$ est possible, mais elle impose une relation de sous-typage entre les bornes. En effet, cela signifie que $\forall T <: B'$, $A'\langle T\rangle <: A\langle T\rangle$. Or ce dernier terme n'est défini que si $T <: B$. Il faut donc 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 <: Equalable⟩`, à condition que `Comparable` soit un sous-type de `Equalable` [6].

Spécialisation et instanciation. Les classes paramétrées peuvent être reliées par une alternance irrégulière d'instanciations et de spécialisations : les spécialisations peuvent introduire de nouveaux paramètres alors que les instanciations en instancient une partie. Soit, par exemple, la suite de classes paramétrées donc chacune est l'instanciation ou la spécialisation explicite de la précédente :

$A\langle T\rangle$		intro T	
$A'\langle T, U\rangle$	spécialisation	intro U	$\prec A\langle T\rangle$
$A'\langle F, U\rangle$	instanciation	$T = F$	
$B\langle T\rangle$	spécialisation		$\prec A'\langle F, U\rangle$
$B\langle G\rangle$	instanciation	$U = G$	
$C\langle T\rangle$	spécialisation	intro T	$\prec B\langle G\rangle$
$C'\langle T, U\rangle$	spécialisation	intro U	$\prec C\langle T\rangle$
$C'\langle H, I\rangle$	instanciation	$T = H$, $U = I$	

Les deux relations se composent de la même manière que les relations d'instanciation et de spécialisation, par une espèce de transitivité asymétrique. $C'\langle H, I\rangle$ est une instance générique implicite non seulement de $C'\langle T, U\rangle$ avec $T = H$ et $U = I$, mais aussi de $C\langle T\rangle$ avec $T = H$, de $B\langle T\rangle$ avec $T = G$, de $A'\langle T, U\rangle$ avec $T = F$ et $U = G$ et de $A\langle T\rangle$ avec $T = F$. Noter que le T de B n'est pas celui de A, puisque ce dernier a été entre temps instancié par F. D'ailleurs, les noms n'ont qu'une portée locale.

Par ailleurs, les relations de spécialisation ou de sous-typage entre les instances paramétrées à types concrets identiques sont valides :

$$C'\langle H, I\rangle \prec B\langle G\rangle \prec A'\langle F, G\rangle \prec A\langle F\rangle$$

La section 7.10 et la figure 7.3, page 144, développent cela dans un cadre d'héritage multiple.

Remarque 7.3. Les types formels se comportent comme des λ-variables. Il sont introduits dans le nom de la classe en cours de définition, et utilisés dans son corps, y compris dans la déclaration des super-classes. Aussi, tout type formel utilisé dans la super-classe doit être introduit par la classe courante. Il est donc erroné de déclarer `B⟨T⟩` comme une sous-classe de `A⟨U⟩`, où `U` serait aussi un type formel.

Héritage multiple. On peut aussi faire de l'héritage multiple, avec une unique contrainte : il est impossible d'instancier de deux façons différentes le même paramètre, même indirectement, à moins qu'ils ne soient variants. Les justifications seront données en discutant des *types virtuels* (Section 7.9, page 140).

7.2 *Templates* C++ et implémentation hétérogène

En C++, une entité paramétrée est appelée un *template*. La généricité n'est pas bornée et l'implémentation est dite *hétérogène* : la substitution du type concret au type formel est faite textuellement

6. En toute généralité, `Set⟨T⟩` nécessite que le type `T` soit muni de l'égalité.

et chaque instance générique d'un même type paramétré est compilée séparément. Les *templates* de C++ représentent donc une réalisation directe, voire naïve, de la définition de la généricité comme une fonction textuelle, qui prend le code paramétré et y substitue les paramètres par leurs valeurs.

Remarque 7.4. Pour la généricité en C++, les détails syntaxiques et des exemples d'utilisation, il est aussi conseillé de consulter le polycopié de Marianne Huchard sur sa page web : http://www.lirmm.fr/~huchard/.

7.2.1 Avantages et inconvénients

Avantages. Le principal avantage concerne l'efficacité de l'implémentation obtenue, surtout dans le cas où le type concret est un type primitif. De plus, le type formel peut être utilisé comme n'importe quel type concret puisque ce sera un type concret qui sera finalement compilé.

Inconvénients. Les inconvénients sont de plusieurs sortes :

- le premier est dû à l'absence de borne : on ne peut jamais savoir si une instance générique va être correcte avant de la faire ;
- chaque instance générique doit être générée une et une seule fois pour la compilation : on pourrait imaginer que ce soit transparent pour le programmeur mais c'est loin d'être le cas ; l'instanciation du type paramétré perd donc son caractère automatique ;
- le coût en espace peut être rédhibitoire : le code est dupliqué pour chaque instance, ce qui ne sert à rien si toutes les types concrets sont des vraies classes (et non pas des types primitifs) ;
- enfin, l'implémentation ne permet pas de définir de vraies (c'est-à-dire "virtuelles") méthodes paramétrées.

Limitations avec les constructeurs. Le type formel peut a priori être utilisé dans n'importe quel contexte où un type concret est utilisé. On peut donc envisager de faire `new T` sur le type formel T, mais cela va poser un problème pour les constructeurs : il faut que le constructeur utilisé soit défini dans tous les types concrets qui peuvent instancier T. Or ce constructeur ne peut pas initialiser tous les attributs de ces types concrets — typiquement, ceux qui sont introduits par une sous-classe (Section 6.9).

Limitation : types récursifs. L'implémentation hétérogène présente une seule limitation : elles interdit les *types paramétrés récursifs*. Soit le type $A\langle T <: B\rangle$ et supposons que $A\langle T\rangle <: B$. Le type $A\langle A\langle T\rangle\rangle$ est alors licite dans le code d'une classe $A'\langle T\rangle$, par exemple dans $A\langle T\rangle$. Mais T peut donc être instancié par $A\langle T\rangle$, ce qui impose de considérer $A\langle A\langle A\langle T\rangle\rangle\rangle$, et ainsi à l'infini.

Ces types récursifs ont un sens à condition d'être créés dynamiquement, à l'exécution. Une substitution textuelle à la compilation, comme en C++, conduirait à une boucle infinie, ce que le compilateur ne cherche même pas à éviter.

7.2.2 Méthodologie

Pseudo borne. Pour remédier à l'absence de borne, le plus simple est sans doute de faire un modèle de généricité bornée, et de l'implémenter, à la borne près. Cela revient à

1. définir la classe B, éventuellement très abstraite, qui possède toutes les méthodes nécessaires au type formel T de $A\langle T\rangle$, y compris les constructeurs utilisés sur T dans $A\langle T\rangle$;
2. documenter $A\langle T\rangle$ en précisant que T doit être un sous-type de B et posséder des constructeurs de même signature que B (sinon, le programmeur décline toute responsabilité) ;
3. compiler $A\langle B\rangle$ pour vérifier que ça marche.

Simulation de la borne. Lorsqu'on définit une classe paramétrée $A\langle T <: B\rangle$, le rôle de la borne est d'assurer que le type formel T ne sera instancié que par des sous-types de B. Il y a une façon très simple d'assurer cette contrainte lorsque le langage ne permet pas de l'exprimer : il suffit de rajouter dans le code de la classe paramétrée une affectation polymorphe de T vers B.

On écrira le fragment suivant

```
x : T
y : B
y = x
```

dans une portion du code de la classe qui n'est pas réellement atteignable (Si le compilateur optimise trop bien le code mort, cela peut poser un problème, mais ce n'est pas le cas de C++). Cela s'applique aussi à la généricité F-bornée.

Noter qu'en C++ ce fragment ne sera vérifié que lors d'une véritable compilation, donc sur une instance générique, c'est-à-dire un peu tard. Mais cela peut permettre d'obtenir un message d'erreur intelligible (probablement noyé dans des messages d'erreur secondaires).

Alternative en C++ 11 Les versions plus récentes de C++ offrent une construction plus agréable et précise, bien qu'elle repose sur une syntaxe durablement incompréhensible. Cette construction repose sur deux mots-clés, `static_assert` et `is_base_of`. En simplifiant outrageusement la syntaxe, on écrira alors quelque-chose comme :

```
static_assert{is_base_of(B,T)}
```

Compilation et édition de liens. Tout serait simple si l'instance générique $A\langle C\rangle$ n'était utilisée qu'à un seul endroit du programme. Il suffirait alors d'insérer les deux lignes suivantes dans un fichier `.cpp` :

```
template class A<C>;
#include "A.cpp"
```

Comme l'instance générique peut être utilisée par plusieurs classes qui peuvent appartenir aux mêmes programmes ou à des programmes différents, il faut adopter une convention qui permette d'en avoir un et un seul exemplaire lors de l'édition de liens. Deux solutions se présentent :

— encapsuler les deux lignes précédents dans des `#ifndef` ;

— mettre chaque instance générique dans un fichier séparé.

Si l'on doit réutiliser des fichiers compilés venant de sources diverses, la première solution risque de ne pas suffire : il suffit pour cela que $A\langle C\rangle$ soit utilisé par 2 fichiers de 2 sources différentes. Conclusion, seule la deuxième solution marche sans restriction à condition que tout le monde l'applique. Mais cela reste une solution lourde puisque le programmeur ne connait pas forcément toutes les classes paramétrée utilisées par les classes qu'il réutilisent : encore une chose qu'il faut documenter.

7.3 Généricité en Java : implémentation homogène et effacement de type

A part les tableaux, il n'y avait pas de généricité en Java 1.4 jusqu'à la version 1.4 incluse. Deux extensions de Java introduisaient des types paramétrés (et d'autres constructions comme les fonctions de « première classe ») : Pizza et Generic Java. Depuis la version 1.5, la généricité est disponible sous sa forme F-bornée et dans une implémentation dite *homogène*.

Remarque 7.5. Pour la généricité en Java, les détails syntaxiques et des exemples d'utilisation, il est aussi conseillé de consulter le polycopié de Marianne Huchard sur sa page web : http://www.lirmm.fr/~huchard/.

Les tableaux en Java. De façon générale, les spécifications de Java respectent la règle de contravariance, avec une seule exception, les tableaux. En effet, les tableaux sont depuis toujours en parfaite contradiction avec la théorie des types : si X est une sous-classe de Y, $X[]$ est considéré comme un sous-type de $Y[]$. Il en résulte la possibilité d'une exception, à l'exécution, lorsque l'on affecte une instance de X à un tableau de type statique $X[]$ dont le type dynamique est $Y[]$. Aux risques et périls du programmeur, du programme et des passagers.

Par ailleurs, avec la version 1.5 et la généricité, la compatibilité avec les versions précédentes ont conduit les concepteurs à accepter des traits non sûrs, en émettant des avertissements à l'intention du programmeur.

C♯ a repris ce même trait non sûr.

7.3.1 Effacement de type

Le principe de l'implémentation homogène de Java repose sur l'*effacement de type*. Le code effectivement compilé pour la classe $A\langle T <: B\rangle$ est celui de $A\langle B\rangle$ et le type concret C qui est substitué à T est « effacé » : il n'apparaît plus dans l'implémentation de $A\langle C\rangle$.

Avantages. Le principal avantage est que l'on a une vraie compilation séparée de la classe paramétrée, qui est donc vérifiée statiquement, et le code résultant est partagé par toutes les instances, ce qui conduit à un gain potentiellement important en espace mémoire.

Inconvénients. Les inconvénients sont de deux ordres :
- lorsque le type concret est un type primitif, par exemple pour toutes les collections, l'effacement de type conduit à une débauche de *boxing* et *unboxing*, qui sont invisibles par le programmeur mais pénalisent lourdement l'efficacité ;
- le type concret C ne faisant plus partie de l'implémentation de la classe paramétrée, de nombreuses fonctionnalités deviennent impossibles sur le type formel (voir Section suivante).

7.3.2 Limitations dues à l'effacement de type

Du fait de l'effacement de type, la seule chose qui reste possible pour utiliser le type formel est l'annotation de type, $x : T$. En effet, les algorithmes de vérification de types du compilateur ne sont pas affectés par l'effacement.

En revanche, aucune action sur le type formel T n'est possible puisque le type concret C qui lui sera substitué disparaît de l'implémentation : si la manipulation était autorisée, elle porterait en fait sur la borne B.

Instanciation. Il n'est pas possible de faire `new T`. Si l'on désire le faire, il faut contourner le problème par le biais du *pattern Factory* : on introduit une nouvelle méthode, nommée par exemple `newT`, qui est déclarée abstraite dans la classe paramétrée $A\langle T\rangle$, laquelle est donc abstraite. Pour chaque instance générique $A\langle C\rangle$, une sous-classe particulière devra être définie[7], qui définira la méthode `newT` pour qu'elle fasse appel à `new C` avec le constructeur adéquat.

L'un des bénéfices de la généricité est donc perdu puisque l'instanciation ne peut pas être directe et qu'il faut définir une sous-classe concrète à la main. Par ailleurs, on retombe sur le même problème d'initialisation des attributs que dans le cas de la généricité hétérogène : on peut ici utiliser n'importe quel constructeur, mais la méthode `newT` a une signature figée, qui ne permet toujours pas d'initialiser les attributs introduits par une sous-classe. On aura donc le même risque d'erreur causé par des attributs non initialisés.

En revanche, il est possible de faire `new A⟨T⟩` dans le contexte d'une classe paramétrée $B\langle T\rangle$ puisque A est d'une certaine manière concrète. En pratique la « valeur » de `T` est inconnue à l'exécution car déjà effacée, mais ça tombe bien puisqu'il aurait justement fallu l'effacer ! Comme, par ailleurs, `new A⟨T⟩` est bien de type `A⟨T⟩`, tout va bien.

Test de sous-typage. La coercition de type vers un type impliquant le type formel T ou son instanciation est très limitée :
- vers le type T lui-même, ce n'est jamais possible,
- vers le type $A\langle T\rangle$, ce n'est possible que dans certains cas, à partir d'un type $A'\langle T\rangle$ qui serait un super-type du premier. La coercition vers $A\langle C\rangle$ n'est possible que dans les mêmes conditions, à partir d'un type $A'\langle C\rangle$ qui serait un super-type du premier.
- En revanche, la coercition depuis un type non paramétré, `Object` par exemple, vers un type paramétré $A\langle T\rangle$ ou $A\langle C\rangle$ n'est pas possible : il est possible de tester que c'est bien une instance de A, mais pas de vérifier le type concret de l'instance. En pratique, Java accepte le *cast* mais ne fait rien au niveau du paramètre : il faut donc que le programmeur soit sûr de lui, mais on verra que cela entraîne une insécurité permanente.

Il est possible de contourner partiellement ces limitations comme on l'a fait pour le cas de `new T`, en définissant des méthodes abstraites dans $A\langle T\rangle$, à redéfinir dans les sous-classes concrètes, en se servant des types concrets et non des types formels[8]. Cependant, cette solution ne permet plus d'utiliser directement $A\langle C\rangle$: il faut auparavant définir une sous-classe unique, $A\_C \prec A\langle C\rangle$, qui implémente ces méthodes abstraites. Un contexte général d'application de ces techniques est la généricité F-bornée, dont la récursion impose une définition explicite (cf. Section 7.5).

Le défaut supplémentaire est que ces méthodes ne peuvent pas s'implémenter n'importe comment : elles doivent faire l'exact équivalent de ce que ferait le mécanisme impossible sur le type formel. Elles doivent donc être documentées pour les réutilisateurs mais l'idéal serait de les faire générer par méta-programmation.

7. Cette sous-classe $A\langle C\rangle$ est vraisemblablement la seule sous-classe directe de $A\langle T\rangle$ puisque toutes doivent partager la même méthode `newT`.

8. C'est d'ailleurs comme cela qu'il faut comprendre la façon dont est implémenté C♯, mais ces méthodes sont générées par le compilateur et la machine virtuelle.

Compatibilité avec Java 1.4. Pour des raisons de compatibilité avec les versions de Java antérieures à la 1.5, il reste possible d'utiliser le type `A` sans expliciter le paramètre concret `A⟨C⟩` : c'est interprété par le compilateur comme `A⟨B⟩` et c'est une source d'erreurs potentielles innombrables. En théorie, et en pratique hors de cette question de compatibilité, un type paramétré doit toujours être utilisé avec le bon nombre de paramètres.

Effacement de type et typage sûr. En théorie, l'effacement de type est compatible avec le typage sûr. Cependant, l'impossibilité effective de faire des tests de sous-typage (*casts*) vers une instance générique alors que le langage autorise ces casts, ne serait-ce que par compatibilité avec Java 1.4, fait que la généricité n'est jamais sûre.

Du coup, Java traîne en permanence un double boulet :

— du fait de la covariance des tableaux, chaque accès en écriture à un tableau nécessite une vérification dynamique de type, le type dynamique du tableau pouvant être plus précis que son type statique ;

— du fait de l'effacement de type et de la façon dont il est spécifié, tout accès en lecture (retournant le type formel) à une classe paramétrée nécessite une vérification dynamique, le type dynamique pouvant n'avoir aucun rapport avec le type statique.

Bien entendu, ces vérifications sont inutiles dans les cas extrêmes de la hiérarchie des types, comme celui des tableaux dont l'élément est `final`, ou pour les classes paramétrées par `Object`.

7.3.3 Divers

Paramétrer un type formel ? Une autre question est la suivante : peut-on paramétrer un type formel ? Par exemple, on voudrait pouvoir manipuler des `A⟨C,Set⟩`, ou `A⟨B,List⟩` de telle sorte que la classe `A` puisse utiliser des collections (`List` ou `Set`) de n'importe quoi, y compris de son premier paramètre. On pourrait imaginer définir la classe `A⟨T,U⟩` avec `U⟨V⟩ <: Collection⟨V⟩`, dans le code de laquelle on pourrait faire des collections `U⟨Object⟩` comme copie d'un `U⟨T⟩` par exemple. N.B. Si on n'utilise pas d'autre collection que `U⟨T⟩` il suffirait de définir `A⟨T,U <: Collection⟨T⟩⟩`.

Les *templates* de C++ permettent vraisemblablement cela sans aucune restriction : comme `U` n'est pas borné il n'a pas besoin d'être paramétré et le type sera instancié par `A⟨C,Set⟩`. En revanche, cela n'est pas possible en Java, d'abord pour une simple limitation de la grammaire. Un langage muni de clauses `where` permettrait de l'écrire. Mais y aurait-il aussi des raisons d'ordre supérieur ? En effet, comme on peut assimiler un type paramétré à une fonction, paramétrer un type formel mène à des fonctions d'ordre supérieur [9].

7.4 Variance des types formels

Les langages de programmation proposent deux extensions qui permettent d'assouplir la règle de typage sûr qui interdit tout sous-typage entre deux instances génériques. Une extension consiste à annoter les types formels, lors de leur introduction dans la définition d'une classe, pour indiquer qu'ils peuvent varier de façon co- ou contra-variante. L'autre extension consiste à utiliser une classe paramétrée en se restreignant à sa partie co- ou contra-variante. Dans les deux cas, on parle de *variance*.

7.4.1 Paramètres co- ou contra-variants

Troublants trous noirs

Comme pour les attributs, l'interdiction du sous-typage entre instances génériques différentes ne vaut que si le type formel T est utilisé à la fois comme type de paramètre et comme type de retour :

— s'il n'est utilisé que comme type de retour, le type paramétré est une espèce de *trou blanc* [10], qui produit des T sans en accepter en entrée ; le type formel est alors dit covariant, et le sous-typage entre instances génériques s'identifie à la spécialisation.

— en sens inverse, si le type formel n'est utilisé que comme type de paramètre, le type paramétré est une espèce de *trou noir*, qui absorbe des T sans en produire en sortie ; le type formel est alors dit contravariant, et le sous-typage entre instances génériques s'identifie à l'inverse de la spécialisation.

9. Les fonctions ordinaires sont dites “de premier ordre”, et les fonctions de fonctions sont “d'ordre supérieur”.
10. Voir votre cours d'astrophysique.

L'opposition trous blancs/trous noirs reproduit l'opposition lecture/récriture, ou plus généralement producteur/consommateur. La règle est en fait plus compliquée, car les types paramétrés peuvent être imbriqués (voir la remarque 7.6 ci-dessous), leur variance peut dépendre récursivement d'elle-même, enfin, ils peuvent être aussi bivariant, c'est-à-dire à la fois covariant et contravariant, voire être bivariant sans pouvoir être ni co- ni contravariant.

En première approximation, la règle des trous blancs et noirs est néanmoins suffisante, et les règles plus complexes peuvent être laissées aux compilateurs et à leurs concepteurs.

7.4.2 Annotations de variance à la définition

Certains langages comme Scala, ou C♯ version 4, permettent de déclarer, lors de son introduction dans la définition d'une classe, qu'un type formel se comporte de façon covariante (trou blanc) ou de façon contravariante (trou noir). On déclarera ainsi les types `TrouBlanc⟨+T⟩` et `TrouNoir⟨-T⟩`. Si $B <: A$, ces déclarations entraînent que `TrouBlanc⟨B⟩` $<:$ `TrouBlanc⟨A⟩` et `TrouNoir⟨A⟩` $<:$ `TrouNoir⟨B⟩`. La déclaration doit bien entendu être compatible avec l'utilisation du paramètre.

Noter que dans cet exemple, `TrouBlanc` et `TrouNoir` peuvent être considérés comme deux classes distinctes, avec des instances distinctes, aussi bien que comme deux interfaces distinctes de la même classe et des mêmes objets. L'astrophysique hésite entre ces deux interprétations, ces deux phénomènes pouvant être sans relation ou, au contraire, résulter d'une singularité de l'espace-temps qui verrait l'un être l'envers de l'autre.

On réécrira donc les règles précédentes ainsi :

$$\frac{A\langle +T \ <: \ B\rangle \quad C' <: C <: B}{A\langle C'\rangle <: A\langle C\rangle} \tag{7.2}$$

$$\frac{A\langle -T \ <: \ B\rangle \quad C' <: C <: B}{A\langle C\rangle <: A\langle C'\rangle} \tag{7.3}$$

Quelques exemples :

— une collection immutable est définie par `ImmutableCollection⟨+T⟩` ;

— une fonction à un paramètre par `Function⟨-S,+T⟩`, pour une fonction de type $S \to T$ (Section 5.2.3, page 75) ;

— l'interface des objets comparables avec eux-mêmes est définie par `Comparable⟨+T <: Comparable⟨T⟩⟩`, qui possède une unique méthode `CompareTo(T)`.

Mis à part les deux derniers cas, le principal usage pratique concernerait donc des entrées-sorties typées ou des structures de données immutables. Beaucoup de langages ne traitent pas ces cas particuliers. Cependant, le constat à l'origine des annotations de variance est aussi à la base de la notion de *joker* (*wildcard* en anglais) en Java (Section 7.4.3, page 128) : au lieu de considérer la définition du type paramétré, on regarde alors l'usage que l'on en fait localement.

Remarque 7.6. La signification des annotations + et − doit être comprise de façon algébrique : *co-* et *contra*-variances se combinent comme + et − en algèbre. En particulier, un type contravariant en position contravariante devient covariant : la variance du type `A⟨B⟨C⟨T⟩⟩⟩` va donc dépendre de la variance du paramètre de chacune des classes `A`, `B` et `C`.

Le langage C♯ utilise, de son côté, les annotations `in` et `out` qui correspondent aux entrées et sorties des fonctions.

Annotation de variance et spécialisation

La définition d'une sous-classe paramétrée ne peut pas faire apparaître une annotation sur un paramètre formel hérité de la super-classe. En effet, `B⟨+T⟩` $<:$ `A⟨T⟩` n'est licite que si `A⟨+T⟩` est licite.

En revanche, une sous-classe peut faire disparaître une annotation de variance héritée de la super-classe. En effet, `B⟨T⟩` $<:$ `A⟨+T⟩` est licite : dans ce cas, `A` ne définit que des méthodes qui retournent `T`, et `B` y rajoute une méthode qui prend `T` en paramètre. Par exemple, `A` est une collection immutable, et `B` sa variante mutable. Le principe est exactement le même avec une annotation de contravariance.

Les annotations de variance ne sont donc pas des propriétés du type formel global, mais uniquement du type formel local.

Exercices

Exercice 7.1. Définir une classe paramétrée $A\langle T\rangle$ qui fonctionne comme un générateur d'instances de T : le code `x : A<C> ; x=new A<C> ; x.genere() ;` se comporte exactement comme `new C`.

Si $D \prec C$, on devrait pouvoir faire `x=new A<D>` dans le code précédent sans provoquer d'erreur de type. Vérifier en C++, Java, C♯, Eiffel, ...

Attention : la mise en œuvre n'est pas forcément possible dans tous les langages, et elle peut nécessiter des adaptations dans certains. Comment faut-il adapter légèrement ce code à Java pour définir la méthode `genere` ? □

Exercice 7.2. Définir une classe paramétrée $A\langle T\rangle$ qui fonctionne comme une poubelle typée d'instances de T et accepte le code `x : A<C> ; x=new A<C> ; x.jette(new C) ;`.

Si $C \prec D$, on devrait pouvoir faire `x=new A<D>` dans le code précédent sans provoquer d'erreur de type. Expliquer pourquoi ce n'est même pas la peine d'essayer en C++, Java, ou C♯. En Eiffel, la raison est différente mais le résultat identique : pourquoi ? □

Exemple : l'introspection en Java

Un bon exemple est constitué par l'introspection en Java (Section 13.1.1). Dans ce cadre, à chaque classe `C` définie en Java correspond une instance d'une classe `Class` qu'il est possible d'accéder à l'exécution.

La classe `Class` est en fait paramétrée, sous la forme de `Class⟨T <: Object⟩`, de telle sorte qu'une instance de `Class⟨C⟩` corresponde à chaque classe `C`.

Exercice 7.3. Analyser la documentation de la classe `Class⟨T⟩` pour déterminer si le paramètre formel `T` n'est utilisé que dans un contexte covariant, ce qui permettrait d'utiliser une annotation de variance `Class⟨+T⟩` en C♯ ou Scala. □

Le lecteur est renvoyé à la section 13.1.1 et suivantes pour analyser les problèmes posés par le typage de la classe `Class` en Java ou C♯.

7.4.3 Variance à l'utilisation : les *jokers*

Le joker est une construction qui intervient non pas lors de la définition d'une classe paramétrée, comme les annotations de variance, mais lors de son utilisation. On parle de *use-site variance*, par opposition à la *definition-site variance* de Scala ou C♯.

S'il n'est pas possible de faire du sous-typage entre instances d'un même type paramétré (Section 7.1.3), il est néanmoins possible de typer génériquement toute une famille d'instances d'un type paramétré : on utilise pour cela un *type joker*, c'est-à-dire un type qui n'est ni formel, ni concret mais qui désigne une place vide que l'on se contente de contraindre par sous-typage (Voir aussi la remarque 7.1, page 121).

Soit le type paramétré $A\langle T <: B\rangle$ et un type concret $B' <: B$. Il est alors possible d'utiliser les annotations de types $A\langle ? <: B'\rangle$ et $A\langle B' <: ?\rangle$ qui désignent respectivement une famille d'instances bornées par B' :

— $A\langle ? <: B'\rangle$: l'ensemble des $A\langle T\rangle$ tels que $T <: B'$;
— $A\langle B' <: ?\rangle$: l'ensemble des $A\langle T\rangle$ tels que $B' <: T <: B$;
— en combinant les deux bornes, $A\langle B'' <: ? <: B'\rangle$ est l'ensemble des $A\langle T\rangle$ tels que $B'' <: T <: B'$.

Bien entendu, les contraintes de sous-typage n'ont pas disparu et ces types n'autorisent pas tout. Sur un receveur typé par :

— $A\langle ? <: B'\rangle$, il n'est pas possible d'appeler une méthode qui prend un paramètre de type T ; en revanche, une méthode qui retourne un T sera typée comme retournant un B' ;
— $A\langle B' <: ?\rangle$, il est possible d'appeler une méthode qui prend un paramètre de type T, qui devra alors être de type B', et une méthode qui retourne un T sera typée comme retournant un B puisque la borne B reste implicite.

Les types avec *joker* ne peuvent être utilisés que comme annotations de type, y compris comme borne dans la définition d'un type paramétré. Il est en revanche impossible de les instancier (`new`).

Avec les *jokers*, on obtient les deux règles de sous-typage suivantes :

$$\frac{A\langle T <: B\rangle \quad C' <: C <: B}{A\langle ? <: C'\rangle <: A\langle ? <: C\rangle} \tag{7.4}$$

$$\frac{A\langle T <: B\rangle \quad C' <: C <: B}{A\langle C <: ?\rangle <: A\langle C'\rangle ?} \tag{7.5}$$

Sur les jokers et leur utilisation, on lira avec profit [Viroli and Rimassa, 2005].

Collection immutables. De façon générale, si la classe paramétrée considérée est une collection, on obtient avec un *joker* une collection immutable : `Collection⟨? <: B⟩` est une collection de `B` qu'il n'est pas possible d'accéder en écriture. Le *joker* propose donc une solution simple au besoin de collections immutables, par exemple pour implémenter les associations UML (Section 6.5).

Exemple. En reprenant l'exemple des `Stack` (Section 7.1.3), on peut typer une expression par :
- `Stack⟨? <: Vaisselle⟩` : le type couvre les types `Stack⟨Vaisselle⟩`, `Stack⟨Assiette⟩`, `Stack⟨Verre⟩`, etc., la méthode `pop` retourne un type `Vaisselle` et la méthode `push` est interdite ;
- `Stack⟨Verre <:?⟩` : le type couvre les types `Stack⟨Object⟩`, `Stack⟨Vaisselle⟩` et `Stack⟨Verre⟩`, la méthode `push` prend un paramètre de type `Verre` et la méthode `pop` retourne un type `Object`.

Joker borné par un type formel. On peut aussi borner un joker par un type formel. Par exemple, toutes les collections sous-classes de `Collection⟨T⟩` peuvent être munies d'un constructeur par copie `Collection(Collection⟨? <: T⟩)`. En effet, si `c1 : Collection⟨? <: T⟩` et `c2 : Collection⟨T⟩`, il est possible d'itérer sur les éléments de `c1` pour les ajouter à `c2`. L'inverse n'est bien sûr pas possible.

7.4.4 Comparaison

Supposons que le paramètre `T` ne soit utilisé que dans un rôle covariant dans la classe `A⟨T⟩`. Un langage doté d'annotation de variance pourrait alors définir la classe sous la forme `A⟨+T⟩`, et `C' <: C` impliquerait que `A⟨C'⟩ <: A⟨C⟩`.

Les jokers permettent alors une expression équivalente : on utilisera `A⟨? <: C⟩` à la place de `A⟨C⟩`. En effet, si `C' <: C`, alors `A⟨? <: C'⟩ <: A⟨? <: C⟩` (règle 7.4). Le joker est même plus général puisque la relation de sous-typage est valide même si le type formel n'est pas covariant : dans ce cas, les usages contravariant de `T` ne sont pas accessibles sur `A⟨? <: C⟩`.

Cependant, le joker est à la charge des utilisateurs d'une classe paramétrée, alors que l'annotation de variance est à la charge du concepteur de cette classe. Il serait donc déloyal pour le concepteur de faire reposer cette charge sur un grand nombre d'utilisateurs, et l'annotation de variance est de loin préférable lorsqu'elle est possible. Le joker, lui, doit servir pour les types formels qui ne peuvent être ni co- ni contra-variant.

7.4.5 Imbrications des types paramétrés

La covariance d'un type formel n'est pas juste définie par son absence pour typer les paramètres de méthodes ou les attributs. Il faut ausi que le type formel ne soit pas utilisé dans un type de retour qui serait paramétré.

Si dans `A⟨T⟩` il apparait un type de retour `B⟨T⟩`, alors `A⟨+T⟩` n'est pas valide, sauf si `B⟨+T⟩` est valide. On peut s'en tirer, néanmoins, en remplaçant `B⟨T⟩` par `B⟨? <: T⟩`, au prix d'une petite limitation.

De façon plus générale, les types paramétrés peuvent s'emboîter, et la variance d'un paramètre formelle se calcule algébriquement par combinaison des + et − des imbrications. Ainsi, dans la classe

```
class A<T> {
   foo () : D<T>
   bar () : B<C<T>>
   baz (E<T>)
}
```

le paramètre de `A` peut être covariant si

1. le paramètre de `D` est covariant : en effet, dans ce cas,
   `Y <: X ⇒ D⟨Y⟩ <: D⟨X⟩` et `T` pourrait être covariant, mais en aucun cas contravariant ;
2. les paramètres de `B` et `C` sont tous les deux covariants : on aurait en effet,
   `Y <: X ⇒ C⟨Y⟩ <: C⟨X⟩ ⇒ D⟨C⟨Y⟩⟩ <: D⟨C⟨X⟩⟩`,
3. mais aussi si les paramètres de `B` et `C` sont tous les deux contravariants : on aurait en effet,
   `Y <: X ⇒ C⟨X⟩ <: C⟨Y⟩ ⇒ D⟨C⟨Y⟩⟩ <: D⟨C⟨X⟩⟩`, donc `T` pourrait être covariant, mais en aucun cas contravariant ;
4. le paramètre de `E` est contravariant : en effet, dans ce cas,
   `Y <: X ⇒ E⟨X⟩ <: E⟨Y⟩` et, comme `E⟨T⟩` est utilisé en position contravariante, `T` pourrait être covariant, mais en aucun cas contravariant.

Le raisonnement est basé sur des combinaisons multiplicatives de + et − : `T` est contravariant (−) dans `E`, `E⟨T⟩` est en position contravariante (−) dans `A`, donc `T` peut être covariant dans `A` : − par − égale +.

7.5 Généricité F-bornée

La généricité dite *F-bornée*, ou *récursivement bornée*, est une généralisation de la généricité bornée où la borne peut être elle-même paramétrée par le(s) type(s) formel(s). 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 (ce qui n'a rien de trivial puisque cela suppose l'existence d'une structure d'ordre, en général total).

Mis à part les algorithmes de vérification de type qui doivent être adaptés pour gérer cette récursion — en particulier pour éviter les boucles infinies —, le principe même ne change pas, à quelques détails près.

Remarque 7.7. Comme il l'a été dit plus haut, les types formels se comportent comme des λ-variables. Il sont introduits dans le nom de la classe en cours de définition, et utilisés dans son corps, y compris dans la déclaration des bornes. On peut en effet comprendre $A\langle T <: B\rangle$ comme une définition de $A\langle T\rangle$ à laquelle on associe la contrainte $T <: B$, mais cette contrainte appartient au corps de A.

Aussi, tout type formel utilisé dans les bornes doit être introduit par la classe courante. Il est donc erroné de déclarer $A\langle T <: B\langle U\rangle\rangle$, où U serait aussi un type formel. La déclaration $A\langle T <: B\langle U\rangle, U\rangle$ est par contre licite, même si la déclaration de U apparaît après son utilisation : il faut en effet lire cela comme "$A\langle T, U\rangle$ avec $T <: B\langle U\rangle$".

Remarque 7.8. Malgré son aspect séduisant, cette modélisation des ensembles ordonnés reste éminemment naïve, puisqu'elle suppose que la structure d'ordre associée à un ensemble donné est unique. Ce n'est évidemment pas le cas, et il serait sans doute préférable de disposer de constructions plus flexibles offrant l'expressivité des mots-clés `:test` et `:key` dans les fonctions de recherche de Common Lisp.

On pourrait par exemple définir `OrderedSet⟨T where T has compare(T):bool⟩`, avec des instanciations permettant de préciser la méthode `compare`, par exemple `OrderedSet⟨Truc where compare=compareTruc⟩`. On retrouve des constructions analogues dans les `where` clauses de [Day et al., 1995]. C♯ a repris la syntaxe des clauses `where`, mais on ne peut y exprimer que le sous-typage (comme les bornes en Java), et l'existence d'un constructeur par défaut.

7.5.1 Règles syntaxiques

La syntaxe générale d'une définition de classe (ou interface) paramétrée, en se restreignant aux aspects propres à la généricité, est la suivante :

```
classdef = class classname<formaltypedef{,formaltypedef}*>
                    {{ extends| implements} superclassref}*
formaltypedef = formaltype | formaltype extends superclassref
superclassref = classname | classname<formaltype{,formaltype}*>
```

avec la règle additionnelle que tout `formaltype` figurant dans les `superclassref` de la définition de la classe considérée doit avoir été introduit par un `formaltypedef` dans la dite classe : le mot-clé `class` fonctionne comme un constructeur `lambda`, en introduisant les seules λ-variables qui aient le droit d'être utilisées en partie droite, soit après le `extends` ou `implements`. `superclassref` peut bien entendu représenter une interface aussi bien qu'une classe.

Remarque 7.9. Quelques lignes plus loin, on verra que la syntaxe peut être légèrement plus compliquée, puisque `superclassref` peut aussi contenir des *jokers*.

7.5.2 Points fixes

Pour définir `OrderedSet`, il faut bien sûr avoir défini auparavant `Comparable` sous la forme hautement récursive de `Comparable⟨T <: Comparable⟨T⟩⟩`. Un tel type constitue une espèce de *point fixe* dont l'usage diffère légèrement de celui d'un type paramétré « normal ».

L'instanciation d'un tel point fixe est maintenant contrainte. Alors que $A\langle C\rangle$ peut être utilisé dans n'importe quelles circonstances dans un cas de généricité simplement bornée, il n'en est pas de même s'il s'agit d'un point fixe $A\langle T <: A\langle T\rangle\rangle$.

En effet, la validité de $A\langle C\rangle$ suppose que $C <: A\langle C\rangle$: il faut donc que C ait été défini de cette façon. Un point fixe s'instancie donc par un point fixe.

De plus, C est vraisemblablement le seul sous-type direct de $A\langle C\rangle$. En effet, supposons que l'on définisse $C' <: A\langle C'\rangle$: il n'y a aucune relation de sous-typage entre C et C'. Définissons maintenant $C'' <: A\langle C\rangle$: la définition est correcte mais pourquoi ne pas avoir défini $C'' <: C$?

Reprenons l'exemple de `Comparable` : `Comparable⟨T⟩` représente l'ensemble des valeurs comparables avec un type `T`. `T` en est sous-type mais la comparabilité étant commutative, `T` est aussi l'ensemble des valeurs comparables avec `T : T` et `Comparable⟨T⟩` ont la même extension [11]. Il s'en suit que `Comparable` est une classe abstraite et qu'utiliser le type `Comparable⟨Integer⟩` est moins précis qu'utiliser `Integer`.

Cette remarque a des effets concrets que l'on examinera à la section 7.6.4, page 134, mais l'argument pourrait néanmoins ne pas s'appliquer à tous les cas.

7.5.3 F-borné, jokers et annotations de variance

La généricité F-bornée peut produire des situations complexes. Supposons les classes suivantes :

```
Comparable⟨T <: Comparable⟨T⟩⟩
A extends Comparable⟨A⟩
B extends A
C⟨T <: Comparable⟨T⟩⟩
C⟨A⟩
```

`C⟨A⟩` est alors licite, mais `C⟨B⟩` ne l'est pas car `B` n'est pas un sous-type de `Comparable⟨B⟩` mais seulement de `Comparable⟨A⟩`.

```
B <: A  <: Comparable⟨A⟩
```

Pour que `C⟨B⟩` soit aussi valide, il faut définir `C` avec un *joker* :

```
C⟨T <: Comparable⟨? super T⟩⟩
```

Il suffit ainsi que `B` soit un sous-type d'une classe qui implémente `Comparable`.

La définition est possible mais la charge de la complexité repose sur chaque programmeur qui définit une classe comme `C`. En revanche, avec les annotations de variance, on aurait pu identifier immédiatement le fait que `Comparable` n'utilise son type formel qu'en position contravariante. Les bonnes définitions de `Comparable` et de `C` deviennent alors :

```
Comparable⟨-T <: Comparable⟨T⟩⟩
C⟨T <: Comparable⟨T⟩⟩
```

ce qui permet d'obtenir :

```
B <: A  <: Comparable⟨A⟩ <: Comparable⟨B⟩
```

Spécifications alternatives. La solution ci-dessus peut sembler tirée d'un chapeau. On voit bien pourquoi mettre le joker dans la borne de la classe `C` marche. Mais n'aurait-il pas été possible de mettre le joker dans la définition de `Comparable` ?

Exemples alternatifs. Le problème des exemples est qu'ils s'apparentent souvent au « meilleur des cas ». Le cas de `Comparable` est particulier du fait qu'il se définirait avec une annotation de contravariance. L'usage du joker n'est donc pas une restriction.

Pourrait-on avoir les mêmes besoins en remplaçant `Comparable` par une classe paramétrée dont le paramètre n'est pas contravariant ?

7.5.4 Comment utiliser la généricité F-bornée

La généricité bornée est à la fois très puissante mais assez délicate à utiliser. Les sections suivantes en donnent deux exemples pour l'implémentation des associations UML.

7.6 Exemple : implémenter une association UML et ses redéfinitions covariantes

La généricité F-bornée offre un moyen relativement commode pour implémenter les associations UML et leur redéfinition vue au chapitre précédent (Figure 6.8, page 102). On peut voir cela comme un nouveau *pattern* (ou patron) « Implémentation d'associations UML par généricité F-bornée ».

11. Voir la discussion au Chapitre 2 sur l'injectivité de *Ext*.

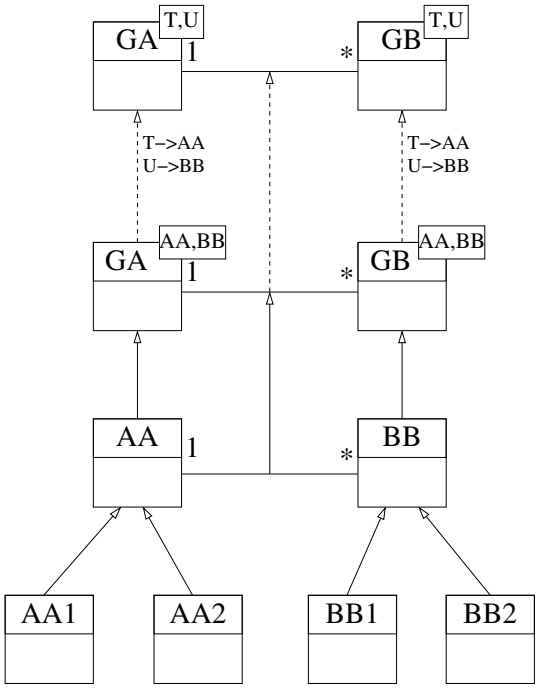


FIGURE 7.1 – L'association de la figure 6.8 avec la généricité.

7.6.1 Principe

Le principe est de définir des classes paramétrées GA et GB, dont A et B seraient respectivement les instances. Le paramétrage consisterait alors à paramétrer GA par GB, c'est-à-dire par un paramètre formel TB borné par GB qui serait instancié par les instances de GB. Et réciproquement.

Or si l'on écrit $GA\langle TB <: GB\rangle$, le paramètre de GB manque. Ce paramètre manquant doit désigner un GA et n'est donc pas TB, puisque TA désigne ici un GB. Il faudrait donc écrire quelque chose comme $GA\langle TB <: GB\langle TA\rangle\rangle$, mais l'usage de TA est ici interdit parce qu'il n'a pas été introduit.

Une première solution serait de remplacer TA par $GA\langle TB\rangle$, qui serait de la bonne "catégorie", puisqu'on cherche quelque-chose qui représente un GA, sans introduire de nouveau type formel. Cela donnerait :

$$GA\langle TB <: GB\langle GA\langle TB\rangle\rangle\rangle$$
$$GB\langle TA <: GA\langle GB\langle TA\rangle\rangle\rangle$$

Cette définition est, à première vue, syntaxiquement correcte, puisque les types formels sont correctement introduits et utilisés, et de la bonne "catégorie", A ou B. Cependant pour s'assurer de sa correction, il faut vérifier que les contraintes des bornes sont bien respectées. Par exemple, pour définir GA, on utilise une instance de GB comme borne et pour que cette instance soit correcte, il faut que son paramètre $GA\langle TB\rangle$ soit bien bornée par la borne de TA dans la définition de GB, c'est-à-dire $GA\langle GB\langle TA\rangle\rangle$:

$$GA\langle TB\rangle <: GA\langle GB\langle TA\rangle\rangle \tag{7.6}$$

Or deux instances de la même classe paramétrée ne peuvent être en relation de sous-typage que si elles sont identiques (Section 7.1.3). Il s'ensuit que $TB = GB\langle TA\rangle$ et, par symétrie, $TA = GA\langle TB\rangle$. C'est un système à deux équations dont la solution repose sur des termes infinis $GB\langle GA\langle GB\langle GA\langle GB\langle ...\rangle\rangle\rangle\rangle\rangle$. Il n'y a donc pas d'espoir dans cette direction.

Il faut donc un deuxième paramètre, dont l'usage est peu intuitif qu'il représente la classe courante et est borné par elle. On aboutit alors à la double définition suivante :

$$GA\langle TA <: GA\langle TA, TB\rangle, TB <: GB\langle TB, TA\rangle\rangle$$
$$GB\langle TB <: GB\langle TB, TA\rangle, TA <: GA\langle TA, TB\rangle\rangle$$

Comment s'assurer de la correction d'une telle définition ? En uniformisant les noms comme on l'a fait, on retrouve 2 paramètres formels TA qui vérifient la même contrainte $TA <: GA\langle TA, TB\rangle$, et de même pour TB par symétrie. De plus, ces définitions utilisent des bornes bien formées, l'une étant la classe en cours de définition, par exemple $GA\langle TA, TB\rangle$ dans $GA\langle TA, TB\rangle$ (en oubliant les bornes qui sont des contraintes), et l'autre étant la définition de la classe duale.

Instanciation. On définit ensuite A et B par une double instanciation parallèle de GA et GB :

$$A <: GA\langle A, B\rangle$$
$$B <: GB\langle B, A\rangle$$

et de même pour AA et BB :

$$AA <: GA\langle AA, BB\rangle$$
$$BB <: GB\langle BB, AA\rangle$$

On constate que la relation de spécialisation entre le couple A-B et le couple AA-BB a disparu.

EXERCICE 7.4. Définir en JAVA les classes de cet exemple, avec les méthodes permettant d'associer des instances. □

Le cas de OCAML Il est possible de définir en OCAML les couples de classes A-B et AA-BB. Cependant, en tant que types, les 2 couples sont exactement isomorphes, et OCAML est incapable de les distinguer.

Type auto-référentiel. Le type supplémentaire TA de GA sera appelé *auto-référentiel*. Le principe même de ces définitions de classe, comme pour `Comparable`, implique que `self` (ou `this`) est toujours une instance de TA. De la même manière, une instance de $GA\langle A, B\rangle$ sera toujours une instance de A.

Remarque 7.10. Le nom des classes, l'ordre et le nom des paramètres méritent un commentaire. Les classes ainsi définies ne sont pas les classes A et B mais leur modèle générique : d'où le 'G'. Chaque classe ainsi définie est un point fixe sur son premier paramètre. Une autre convention aurait pu être de garder un ordre constant des paramètres : A puis B dans toutes les classes. Le nom des types formels doit enfin être bien choisi, par exemple TA et TB, pour que le caractère formel du type et ce qu'il représente apparaissent bien.

7.6.2 Associations généralisées et *joker*

Bien entendu un modèle objet un peu complexe ne se réduit pas à une association isolée. Il faut donc considérer des graphes d'associations plus complexes. Examinons différents cas, les chaînes, les cycles et les boucles.

Chaîne d'associations. Supposons une chaîne A_1, A_2, ..., A_n d'associations, où chaque classe A_i est associée à la suivante ($i < n$) et à la précédente ($i > 1$). Si $n \geq 3$, A_2 est ainsi liée à 2 autres classes A_1 et A_3 par des associations : il lui faut donc 3 paramètres, le paramètre du point fixe et un pour chacune des 2 autres classes. On définira donc :

$$GA_2\langle T_2 <: GA_2\langle T_2, T_1, T_3\rangle, T_1 <: GA_1\langle T_1, T_2\rangle, T_3 <: GA_3\langle T_3, T_2\rangle\rangle$$

Cette définition est cependant incorrecte si $n \geq 4$ car GA_3 doit alors avoir un 3ième paramètre, T_4, qui devrait être aussi introduit par GA_2. Pourtant cette dernière classe n'a pas à connaître GA_4, puisqu'il n'y a pas d'association directe entre les 2. Par contaminations successives, ce paramètre supplémentaire devrait être ajouté aux classes GA_1 et GA_2, et par généralisation on voit que chaque classe de la chaîne devrait avoir n paramètres, ce qui serait d'une lourdeur syntaxique inacceptable.

On est ici sauvé par les *jokers* : il suffit de remplacer le paramètre supplémentaire par un joker :

$$GA_2\langle T_2 <: GA_2\langle T_2, T_1, T_3\rangle, T_1 <: GA_1\langle T_1, T_2\rangle, T_3 <: GA_3\langle T_3, T_2, ?\rangle\rangle$$

Remarque 7.11. Dans le cas d'une chaîne, on adoptera, par exemple, l'ordre suivant : en premier le point fixe, ensuite les autres paramètres dans l'ordre de la chaîne.

EXERCICE 7.5. Définir en JAVA les classes de cet exemple (avec $n = 4$), avec les méthodes permettant d'associer des instances. Montrer que les *jokers* ne limitent pas le modèle. □

Cycles et cliques. Dans le cas d'un cycle, lorsqu'il y a aussi une association entre A_1 et A_n, on a un modèle uniforme avec 3 paramètres et on définit GA_2 avec un joker supplémentaire :

$$GA_2\langle T_2 <: GA_2\langle T_2, T_1, T_3\rangle, T_1 <: GA_1\langle T_1, ?, T_2\rangle, T_3 <: GA_3\langle T_3, T_2, ?\rangle\rangle$$

Dans le cas d'un graphe plus dense qu'une chaîne, le nombre de paramètres d'une classe est son degré dans le graphe d'associations, plus un. Si le graphe est complet, on retombe sur les n paramètres.

Cas général. Sans *joker*, le nombre de paramètres requis pour une classe est égal à la taille de sa composante connexe dans le graphe dont les sommets sont des classes et les arêtes des associations. Avec les jokers, le nombre de paramètres peut-être simplifié et réduit au degré de la classe considérée, plus un, sauf dans le cas d'un sommet isolé qui n'a pas besoin de paramétrage.

Boucles. Le cas des boucles est intéressant. Les extrémités de l'association peuvent être la même classe A. Mais lors d'une spécialisation, les deux extrémités peuvent devenir distinctes. On ne peut donc pas fusionner les deux paramètres en un seul.

$$GA\langle TA <: GA\langle TA, TB\rangle, TB <: GA\langle TB, TA\rangle\rangle$$

mais les instanciations pourront être de deux formes :

$$A <: GA\langle A, A\rangle$$

ou

$$AA_1 <: GA\langle AA_1, AA_2\rangle$$
$$AA_2 <: GA\langle AA_2, AA_1\rangle$$

On modélise ainsi le fait que les AA_1 sont associés à des AA_2 et réciproquement : le graphe d'instances résultant est biparti.

Exercice 7.6. Définir en Java les classes de cet exemple, avec les méthodes permettant d'associer des instances. □

7.6.3 Spécialisations partielles

L'implémentation de la spécialisation d'association marche bien lorsque le couple AA-BB constitue un point fixe de l'association : BB représente le sous-ensemble des B en association avec les AA et réciproquement.

En revanche, si on veut spécialiser l'association de façon asymétrique, en disant que les AA sont associés à des BB mais les BB peuvent être associés à n'importe quels A, cela ne marche plus.

Exemple : le système de types de Java. Pour étendre le méta-modèle du chapitre 3 dans le cas de Java, on peut décrire plus précisément les classes avec 3 entités et une association de sous-typage (ou spécialisation) :

- `Type` est la classe générale de tous les types et l'association `sous-type-de` est une boucle ; item `Class` est la sous-classe des classes : une classe a pour sous-type des classes et pour super-types n'importe quels types ;
- `Interface` est la sous-classe des interfaces : une interface a pour super-type des interfaces et pour sous-type n'importe quel type.

Le formalisme graphique de Uml montre ici ses limites. On présente les associations (l'association A-B par exemple) comme une arête (non orientée) dans le diagramme, mais cette arête recouvre en réalité 2 arcs (orientés) qui représentent les deux rôles de l'association.

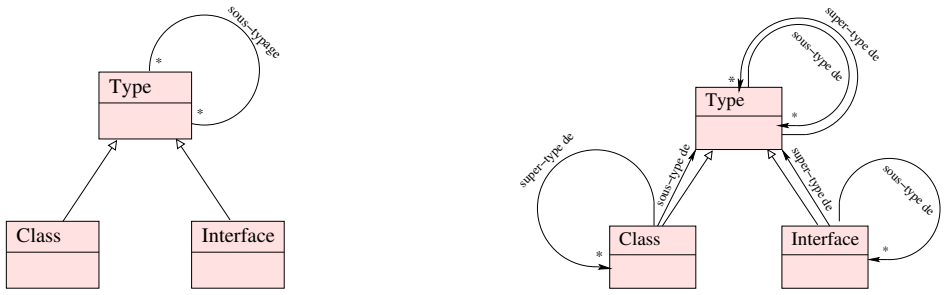
La modélisation de ce système de types au moyen de la généricité ne peut pas marcher pour une raison simple : la généricité fait perdre les relations de spécialisations. Il n'est donc pas possible de définir `Type`, `Class` et `Interface` comme des instances d'une même classe paramétrée tout en conservant leurs relations de spécialisations.

Exercice 7.7. Définir les classes de cet exemple en utilisant la (simulation de la) redéfinition covariante. □

7.6.4 Points fixes : typer par T ou $A\langle T\rangle$?

Dans la définition d'un point fixe $A\langle T <: A\langle T\rangle\rangle$, il faut a priori typer par le type auto-référentiel T et non par $A\langle T\rangle$, parce que le type T est plus précis : il représente un sous-type, en fait l'unique sous-type direct, de $A\langle T\rangle$.

Cependant, dans le code de A, `this` est typé par $A\langle T\rangle$ et non par T : il n'est donc pas possible de passer `this` à une méthode qui attend un T.



(a) Modèle Uml (b) Modèle réel

Figure 7.2 – Spécialisation partielle d'associations : (a) ce qu'Uml permet d'exprimer simplement ; (b) le modèle réel que l'on voudrait exprimer, en remplaçant chaque arête non orientée par deux arcs orientés. En réalité, Uml permet de spécifier un peu plus le diagramme, à coup de `subset`, mais le résultat reste aussi difficilement lisible.

Exemple. Ainsi dans l'exemple de l'association, dans une méthode de GA, il n'est pas possible de passer `this` à une méthode de GB qui attend un TA et non pas simplement un $GA\langle TA, TB\rangle$. Par exemple, les deux classes GA et GB doivent implémenter des méthodes `addB` et `setA` pour associer symétriquement une instance de A à une instance de B. Pour assurer la symétrie, l'une des méthodes doit appeler l'autre sur son paramètre, en lui passant `this` en argument, par exemple :

```
abstract class GA<TA extends GA<TA,TB>,TB extends GB<TB,TA>>{
    void addB(TB b){... b.setB(this) ...}
    }
abstract class GB<TB extends GB<TB,TA>,TA extends GA<TA,TB>>{
    void setA(TA a){... ...}
    }
```

L'appel `b.setB(this)` est mal typé et la coercition `(TA)this` est a priori impossible en Java à cause de l'effacement de type, bien que l'on sache que `this` est forcément une instance de TA. Cependant, cette coercition `(TA)this` est acceptée par Java, qui ne fait rien (ça tombe bien puisqu'il n'y a rien à faire). Au contraire, en C♯ où la coercition est possible, le langage effectue une vérification (qui est donc inutile).

Alternative 1 : relâcher les types. Pour s'en sortir on pourrait adopter le principe de typage suivant : on utilise $A\langle T\rangle$ comme type de paramètre, ce qui permet d'accepter le plus de valeurs possible, y compris `this`, mais T comme type de retour, car c'est plus précis. Mais ce n'est pas possible car la méthode peut retourner potentiellement l'un de ses paramètres, y compris le receveur `this` : il faudrait donc typer aussi le retour par $A\langle T\rangle$. Une méthode `foo` de type $T \to T$ doit donc être typée $A\langle T\rangle \to A\langle T\rangle$.

Cette solution permet de conserver un typage précis mais elle est très lourde syntaxiquement puisque le type $A\langle T\rangle$ devra être traîné dans toutes les instanciations de A et que le nombre de paramètres peut être beaucoup plus important. Ainsi, dans l'exemple des graphes d'objets (Section 8.4), les classes de la chimie devraient être typées par `Ggraphe⟨Molécule,Atome,Liaison⟩` au lieu d'être simplement typées par `Molécule`.

Attention, en Java, pour les raisons de compatibilité évoquées plus haut, il reste possible de typer par `Ggraphe`, sans paramètres, qui est interprété par le compilateur comme `Ggraphe⟨Ggraphe,Gsommet,Garete⟩`, c'est-à-dire un `Ggraphe` de n'importe quoi.

Alternative 2 : méthode abstraite `self`. Une alternative consiste à définir dans $A\langle T\rangle$ une méthode abstraite `self()` de type de retour $A\langle T\rangle \to T$, et à la redéfinir dans chaque instance $C <: A\langle C\rangle$ de $A\langle T\rangle$, pour qu'elle retourne `this`.

Cette technique autorise un typage plus précis, puisque T peut être utilisé partout à la place de $A\langle T\rangle$, mais elle a le double inconvénient

— d'imposer au programmeur de la classe d'utiliser `self()` à la place de `this` comme paramètre[12] de tous les appels de méthodes dans le code de $A\langle T\rangle$;

— d'imposer au réutilisateur un protocole particulier d'instanciation et de spécialisation de $A\langle T\rangle$;

12. Il faut bien entendu étendre ceci à l'affectation de `this` aux variables locales de type T.

— de nécessiter à l'exécution un appel de méthode supplémentaire qui ne fait rien d'autre que permettre le typage statique.

Malgré ces inconvénients, il s'agit certainement de la meilleure technique, car elle localise le problème en un point unique, alors que la solution 1 peut imposer des casts un peu partout, y compris dans le code utilisateur.

En Scala. Dans ce langage, une déclaration mystérieuse permet de résoudre élégamment ce problème. On rajoutera la définition suivante dans le code de la classe :

```
implicit def ev : this.type <:< T;
```

Par la suite, `ev(this)` sera typé par T, alors que `this` est typé par $A\langle T\rangle$. Ce code marche dans le cas présent, et qu'il s'agisse de types formels ou de types virtuels. Mais pourquoi et comment marche-t-il ? Le mystère est assez épais.

Après une enquête approfondie, il semble que dans `this.type`, `type` soit le type virtuel appelé ici `SelfType`. On s'en servira en particulier pour une méthode qui retourne `this`, ce qui est d'un intérêt moyen mais permet, par exemple, d'enchaîner des appels :

```
x.foo.bar.baz
```

au lieu d'avoir à écrire une séquence :

```
x.foo
x.bar
x.baz
```

Spécialiser et instancier un point fixe. Si le typage par un point fixe est problématique, il n'en est pas de même pour sa spécialisation ou son instanciation. Si le point fixe $A\langle T <: A\langle T\rangle\rangle$ est instancié par $C <: A\langle C\rangle$, il ne faut plus jamais spécialiser ou instancier $A\langle C\rangle$, mais uniquement C.

La raison en est que C possède des méthodes propres au domaine de cette instanciation — par exemple la chimie dans l'exemple des `Molécules` —, ou la méthode `self` discutée ci-dessus, alors que $A\langle C\rangle$ ne contient que des méthodes génériques.

Pour l'instanciation on se protégera en définissant A comme une classe abstraite. Pour la spécialisation, le compilateur laisserait passer une définition $D <: A\langle C\rangle$ et seule la discipline du programmeur permet de l'éviter.

Expressivité du langage. Cette question du typage de `this` dans un point fixe vient d'une limitation du langage. On pourrait supposer que le point fixe d'une borne récursive puisse se définir avec un mot-clé, par exemple `fixpoint` et l'on écrirait alors

```
class A<T fixpoint, U extends .., V extends ..>
```

à la place de

```
class A<T extends A<T,U,V>, U extends .., V extends ..>
```

avec la propriété additionnelle que `this : T` si l'on utilise le mot-clé `fixpoint`. Bien entendu, une définition de classe ne peut contenir qu'un seul paramètre `fixpoint`.

Ce mot-clé `fixpoint` pourrait aussi servir à empêcher la spécialisation abusive de `A⟨C, ..⟩` à partir du moment où `C <: A⟨C, ..⟩` a été défini (avec néanmoins une difficulté en monde ouvert).

7.6.5 Limitations

L'approche présente clairement plusieurs limitations :

— le passage à l'échelle apparaît difficile sur le plan syntaxique : les types manipulés pourraient atteindre des tailles inacceptables pour le programmeur et il faudrait que ce soit uniquement réservé à du code généré par programme ;

— la spécialisation à la base du modèle d'origine est perdue : en particulier, l'approche permet d'implémenter des modèles disjoints, par exemple des `Molecules` et des `Reseaux`, mais elle ne permet pas de réelle spécialisation ;

— le langage perd la trace du point-fixe et du fait que le type formel T représente (en général) le seul sous-type direct du type paramétré $A\langle T\rangle$.

7.6.6 Généricité F-bornée en C++

Comme on l'a vu précédemment, le moyen le plus simple pour définir des *templates* en C++ consiste à borner les types formels comme en JAVA, mais en exprimant cette borne en commentaire.

Pour les bornes récursives dans les exemples de `GA` et `GB` on peut ainsi définir :

```
abstract class GA<TA,TB> // TA <: GA<TA,TB>, TB <: GB<TB,TA>
{
    void addB(TB b){... b.setB(this) ...}
    TA foo() { ... }
    }
abstract class GB<TB,TA> // TA <: GA<TA,TB>, TB <: GB<TB,TA>
{
    void setA(TA a){... ...}
    }
```

On définit alors `A` et `B` comme des sous-classes respectives de `GA⟨A,B⟩` et `GB⟨B,A⟩`.

A première vue, il serait possible de se passer du point-fixe, en définissant :

```
abstract class GA<TB> //  TB <: GB<TA>
{
    GA<TB> foo() { ... }
    }
abstract class GB<TA> // TA <: GA<TB>
{
    }
```

On définit alors `A` et `B` comme des sous-classes respectives de `GA⟨B⟩` et `GB⟨A⟩`. Mais si on a besoin d'utiliser le type `TA` dans `GA`, par exemple dans la méthode `foo`, il faudra utiliser `GA⟨TB⟩` qui sera un peu moins précis que `TA`, puisque dans `A`, il s'agira de `GA⟨B⟩` et non de `A`.

Supposons maintenant que l'on définisse dans `A` une méthode `bar()`, alors `new A().foo().bar()` est licite dans la première version, pas dans la seconde. En revanche, le problème du `SetB(this)` aurait disparu.

Nobody's perfect !

7.7 Alternative par le patron Stratégie

L'approche précédente est très élégante par sa symétrie : toutes les classes en relation participent à égalité dans le schéma global. Le prix à payer n'est pas négligeable, par la complexité du paramétrage des classes, avec autant de paramètres que de classes en jeu.

Une alternative toujours basée sur la généricité F-bornée a été proposée en s'inspirant du patron Stratégie.

7.7.1 Principe

Le patron Stratégie permet de définir des comportements spécifiques à différents "contextes".

On va donc l'appliquer à un graphe d'associations en prenant, plus ou moins arbitrairement, l'une des classes de ce graphe en guise de pivot.

On peut alors définir un premier point fixe qui introduit le paramètre, puis les autres classes qui s'en servent et l'instancient :

`GA⟨T` $<:$ $GA\langle T\rangle\rangle$
`GB⟨T` $<:$ $GA\langle T\rangle\rangle$
`A` $<:$ $GA\langle A\rangle$
`B` $<:$ $GB\langle A\rangle$

Les signatures des accès aux rôles de l'association vont alors être les suivantes :

```
GA.getB() : GB⟨T⟩
GB.getA() : T
GA.setB(GB⟨T⟩)
GB.setA(T)
```

La différence fondamentale avec le schéma précédent est que seul le type du point fixe va être précis : `B.getA` est bien de type `A`. Mais `A.getB` n'est que de type `B⟨A⟩`, à moins de le redéfinir de façon covariante dans `A`, mais un cast est nécessaire.

En sens inverse, le code de `GB` peut appeler `GA.setB(this)` sans nécessiter de cast, alors que le code de `GA` doit faire `GB.setA((T)this)`.

7.7.2 Autres alternatives

Il est relativement facile, bien qu'un peu fastidieux, d'énumérer toutes les solutions possibles à la question de la définition des classes `GA` et `GB` en utilisant la généricité.

Le principe consiste à rajouter des paramètres de types, jusqu'à ce qu'on obtienne des définitions correctes des deux classes, c'est-à-dire des définitions qui respectent les propriétés suivantes :

— `GA` et `GB` se référencent mutuellement,
— il est possible de typer `GA.getB()` et `GB.getA()` de façon précise, donc de telle sorte que `A.getB():B` et `B.getA():A`, où `A` et `B` sont définis comme des sous-classes d'instances génériques de `GA` et `GB`.

Par ailleurs, pour que les définitions soient syntaxiquement correctes, il faut encore que :

— tout type formel utilisé "à droite" (c'est-à-dire dans une borne ou un super-type) soit introduit dans la classe considérée,
— tout type paramétré soit utilisé avec le bon nombre de paramètres,
— tout type formel soit correctement typé, c'est-à-dire borné : en pratique, cela revient à vérifier que la borne du i-ème type formel de la classe `GA` ou `GB` est toujours la même (`GA` ou `GB`, mais pas tantôt l'un tantôt l'autre).

Enfin, pour minimiser le nombre de paramètres, on rajoutera la règle suivante :

— on n'introduit un nouveau paramètre avec une certaine borne que si un tel paramètre n'est pas déjà présent dans la classe.

Remarque 7.12. On pourrait aussi envisager une alternative consistant à remplacer le deuxième paramètre par un *joker* :

$$GA\langle TB <: GB\langle ?\rangle\rangle$$
$$GB\langle TA <: GA\langle ?\rangle\rangle$$

Cependant, GA ne pourrait plus typer les appels de méthodes de GB qui prennent ou retournent des GA.

Exercice 7.8. Définir en Java les classes de cette alternative et montrer les limitations causées par les *jokers*. □

7.7.3 Comparaison

L'essence du patron Stratégie consiste à paramétrer le code avec un paramètre de contexte. Cela n'a pas de lien direct avec les associations et on pourrait très bien rajouter des classes de contexte pour cela.

Ainsi, dans l'exemple des graphes d'objets (Section 8.4) on peut appliquer le patron avec une borne récursive sur les classes `Graphe`, `Molécule` et `Réseau`. On pourrait tout aussi bien ajouter une classe `Contexte`$\langle$`T <: Contexte`$\langle$`T`$\rangle\rangle$, avec ses spécialisations `TheorieGraphe`, `Chimie` et `Telecom`.

Dans tous les cas, on voit que le (ou les) points fixes apportent à la fois un complément d'information de type, et un manque puisque ce complément d'information n'est pas valable pour `this`.

7.8 Dans les autres langages

7.8.1 Généricité en .Net ou C♯

En .Net, les spécifications de la généricité retiennent plus ou moins le meilleur de Java et C++ et son implémentation est un compromis entre les implémentations homogènes et hétérogènes [Kennedy and Syme, 2001].

Types dynamiques. L'implémentation de chaque instance générique de la classe paramétrée embarque le type concret de l'instanciation. Il est donc possible de faire des tests de sous-typage précis, pour vérifier qu'un objet est bien instance de T, de $A\langle T\rangle$ ou de $A\langle Truc\rangle$.

Limitations sur les constructeurs. A cause de la limitation sur les constructeurs citée plus haut, il reste impossible de faire `new T` autrement qu'avec un constructeur par défaut, sans paramètre.

Covariance Comme en JAVA, les types de tableaux sont covariants, donc non sûrs. A partir de la version 4, des annotations de variance peuvent aussi préciser le caractère co- ou contra-variant de chaque paramètre, mais cela reste dans un contexte de typage sûr.

Contrairement à JAVA, il n'y a pas de *joker*, mais nous avons vu que c'est partiellement compensé par les annotations de variance (`in` et `out`) des types formels, la définition (manuelle) de `ImmutableSet⟨out T⟩` permettant de faire l'équivalent de `Set⟨? <: T⟩` (Section 7.4.3).

Généricité bornée et clauses `where` En .NET les bornes s'expriment au moyen de clauses `where`. Ainsi, au lieu d'écrire `A⟨T <: B⟩`, on écrira `A⟨T where T : B⟩`. Il ne s'agit néanmoins que d'une variation syntaxique.

La présence de `new()` en fin de clause `where` permet aussi de spécifier la nécessité d'un constructeur par défaut qui sera utilisé dans le corps de la classe pour instancier le type formel.

Ces clauses `where` sont donc loin de l'expressivité proposée dans [Day et al., 1995].

Limitations sur les *mixins* Il est aussi impossible de faire des *mixin*s, c'est-à-dire de déclarer que $A\langle T\rangle$ est une sous-classe de T (Section 4.6, page 59).

La raison de cette limitation est très prosaïque : cette implémentation des mixins ne peut pas se compiler de façon séparée, car la table des méthodes de la super-classe T est inconnue et l'invariant de position des méthodes ne peut pas être assuré (cf. Chapitre 9, page 163). Une implémentation homogène est donc impossible. Cependant, comme les spécifications de .NET autorisent une implémentation hétérogène, cette limitation reste un peu arbitraire.

Coercition On a vu que la règle de contravariance empêche, en règle générale, que $A\langle D\rangle <: A\langle C\rangle$ lorsque $C <: B$. Aussi, il n'est pas non plus possible de faire une coercition entre deux instances génériques. Cependant, avec les annotations de variance, C♯ le permet dans certains cas.

Lorsqu'il n'y a pas de sous-typage entre deux instances génériques, C♯ admet aussi ce *cast* mais `(A<C>)new A<D>()` est en fait du sucre syntaxique pour `new A<C>(new A<D>())`. C'est proprement criminel puisque cela consiste à subtituer une conversion à une coercition, tout en gardant la syntaxe de la coercition.

7.8.2 Généricité F-bornée dans les autres langages

Généricité F-bornée en EIFFEL

Les spécifications récentes d'EIFFEL incluent la généricité F-bornée, qui fonctionne correctement avec EIFFEL STUDIO (en 2007). En revanche, les algorithmes du compilateur SMART EIFFEL ne maîtrisaient semble-t-il pas bien les récursions nécessaires à l'implémentation des graphes d'objets (en 2006).

EXERCICE 7.9. Implémenter en EIFFEL l'association (Exercice 7.4, page 133) par généricité F-bornée. Vérifier avec SMART EIFFEL et EIFFEL STUDIO. □

Généricité F-bornée en C++

Pas plus que la généricité bornée, le langage ne permet pas d'exprimer la généricité F-bornée. Cependant, comme pour la généricité bornée, il est possible de l'appliquer en faisant comme si les paramètre formels étaient bornés. Un schéma de paramétrage qui marche en JAVA marchera aussi en C++, sans les limitations liées à l'effacement de type mais avec celles qui résultent des points-fixes, et bien sûr sans les vérifications statiques par le compilateur.

En particulier, pour la limitation discutée en Section 7.6.4, page 134, l'appel `b.setB(this)` est toujours mal typé mais la coercition `dynamic_cast<TA>(this)` est maintenant possible. Comme on sait que `this` est forcément une instance de TA, il est vraisemblable qu'un `static_cast` marchera aussi, ce qui rendra le test complètement indolore.

EXERCICE 7.10. Implémenter en C++ l'association (Exercice 7.4, page 133) en simulant la généricité F-bornée. □

Circularité des définitions en C++. L'un des innombrables défauts de C++ réside dans son traitement purement linéaire des informations. Tous les cas de circularité réclament des acrobaties plus

ou moins marquées, bien connues des programmeurs C++ mais souvant oubliées par les néophytes. Ainsi, par exemple, si l'on veut définir les deux classes

$$AA <: GA\langle AA, BB\rangle$$
$$BB <: GB\langle BB, AA\rangle$$

dans des fichiers séparés, il faut prévoir le préambule suivant dans le fichier `AA.h` et le symétrique dans `BB.h` :

```
#ifndef AA_H
#define AA_H
#include{GA.h}                          //la super-classe
class AA;                               //la définition anticipée de AA comme une classe
#include{BB.h}                          //pour que AA soit connue comme une classe dans BB
class AA: public virtual GA<AA,BB>      //pour enfin définir AA
...
```

7.9 De la spécialisation à la généricité : les types virtuels

7.9.1 Limites des approches précédentes

L'implémentation des associations UML et de leur spécialisation a trouvé deux approches possibles : la spécialisation covariante (ou sa simulation) et la généricité.

Les deux approches présentent des limites nombreuses :

— La spécialisation covariante permet une implémentation fine d'un modèle UML mais elle rend tout programme potentiellement non sûr : il suffit de rajouter une sous-classe qui fait une redéfinition covariante pour rendre le code précédent non sûr. En compilation séparée, plus rien n'est sûr et des tests dynamiques de type doivent être générés pour quasiment tous les appels de méthodes C'est inacceptable pour des raisons de sûreté et d'efficacité.

— La simulation de la redéfinition covariante présente les mêmes possibilités de modélisation mais elle a l'avantage de restreindre les cas non sûrs aux redéfinitions covariantes effectives. Le test dynamique de type n'est pas généré dans le code de la méthode appelante mais dans celui de la méthode appelée. L'inconvénient est la lourdeur syntaxique pour le programmeur.

— La généricité est appropriée pour implémenter des modèles isomorphes mais sans relation de spécialisation, comme les `Molecules` et les `Reseaux`, dans un cadre de typage sûr. Elle ne convient en revanche pas pour les modèles avec spécialisation, comme le montre l'exemple du système de types de JAVA.

L'approche des types virtuels constitue un intermédiaire entre la spécialisation covariante et la généricité F-bornée, qui préserve la puissance d'expression de la première tout en assurant une sûreté du typage proche de la seconde, sans en avoir les restrictions.

7.9.2 Principe

Définition 7.1 (TYPE VIRTUEL) *Un type virtuel est une propriété des classes dont la valeur est un type concret, exprimée sous la forme $T <: C$ ou $T = C$.*

La première forme indique que le type virtuel peut être redéfini de façon covariante dans les sous-classes. Au contraire, la deuxième forme indique que le type virtuel est final *et qu'il ne peut pas être redéfini.*

Un type virtuel particulier `SelfType` (aussi appelé `MyType`) représente le type du receveur courant : il correspond à une définition implicite `SelfType <: A` dans chaque classe A. Ainsi, de même qu'il existe un mot-clef, variable 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.

Exemples

Dans la définition du couple de classes A-B et de leur association, on définira dans A un type virtuel $TB <: B$ et dans B un type virtuel $TA <: A$. Dans le couple de sous-classes AA-BB, on redéfinira les deux types virtuels par $TB = BB$ et $TA = AA$ si l'on considère que le type est final et ne peut pas être redéfini. Si des redéfinitions ultérieures sont encore possibles, on utilisera alors $TB <: BB$ et $TA <: AA$.

Dans l'exemple du système de types de JAVA, on définira dans `Type` deux types virtuels `Tsuper <: Type` et `Tsub <: Type`. Dans `Class`, on redéfinira `Tsub = Class`, et dans `Interface`, `Tsuper = Interface`.

Sous-typage

Le sous-typage des types virtuels vérifie les règles suivantes :

1. Dans la classe qui définit le type virtuel $T <: B$, le type T n'est sous-type que de B et des super-types de B, et il n'a pas d'autres sous-types que lui-même.
2. Dans la classe qui définit le type virtuel $T = C$, T a les mêmes relations de sous-typage que C.
3. Utilisé en type de paramètre ou en type de retour, un type virtuel n'est pas redéfinissable ;
4. Un envoi de message de la forme `x: A; y: C'; x.foo(y)` est sûr si le type de paramètre de `foo` est un type virtuel T défini par $T = C$ dans A, avec $C' <: C$.
5. En revanche, si le type virtuel a été défini par $T <: C$, l'appel n'est pas sûr : mais il peut être accepté par le compilateur à condition que celui-ci (1) avertisse le programmeur, (2) insère un test de type dynamique.
   En particulier, si le type virtuel est `SelfType`, l'appel `x.foo(y)` n'est pas sûr dans le cas général.
6. Cependant, un envoi de message de la forme `x: A; x.foo(x.bar())` est sûr si le type de retour de `bar` est le même type virtuel T que le type de paramètre de `foo`. Noter que cela n'est vrai que parce que les deux appels ont le même receveur `x`.
   En particulier, si le type virtuel est `SelfType`, l'appel `x.foo(x)` est sûr car il s'agit du même objet.

Types virtuels et typage sûr. Si les types virtuels sont intégrés dans un langage comme seule forme de covariance de type de paramètres, le seul cas d'appel de méthode pas sûr est le (5) (qui inclut le (6)). Pour que le cas (6) soit sûr, il faut que le type de retour soit un type virtuel : cela ne marche pas si ce n'est que la borne du type virtuel. Le meilleur usage des types virtuels consiste donc à les coupler avec des signatures de méthode invariantes.

Usage externe

Un type virtuel T peut avoir tous les usages d'un type dans le cadre de la classe A qui le définit ou en hérite : annotation de type, instanciation (`new`, avec les réserves exprimées plus haut sur la question des constructeurs) et coercition de type.

Ces usages sont aussi possibles à l'extérieur de la classe, sous réserve que l'implémentation le permette : si $x : A$, alors le type $x.T$ peut être utilisé dans tous les usages d'un type.

7.9.3 `SelfType` et méthodes binaires

Le type virtuel `SelfType` permet de traiter deux problèmes de typage, la *perte d'information* et les *méthodes binaires*.

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. Dans l'exemple précédent, il faudrait donc déclarer m_A`(): SelfType` ou, en EIFFEL, m_A`(): like Current`.

En particulier, une méthode de copie (`klone`) doit avoir un type de retour `SelfType`. Dans un langage comme JAVA où `SelfType` n'existe pas, `klone` a le type de retour `Object`, et le programmeur doit rajouter un cast explicite, toujours sûr, pour récupérer le type d'origine :

```
x : A
y : A
y := x.klone()              erreur
y := (A)(x.klone())         ok
```

Sur la copie d'objets voir aussi l'exercice Section 14.4.2, page 218.

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. Le type `SelfType` est à nouveau la bonne façon d'exprimer le type des paramètres de ces méthodes binaires, mais elles restent intrinsèquement non sûres.

Test d'égalité Par ailleurs, une méthode `equals` dont le paramètre serait de type `SelfType` ne peut être considérée comme une implémentation correcte d'un test d'égalité, car il faudrait assimiler l'erreur de type à l'échec du test. En réalité, la bonne spécification d'un test d'égalité devrait passer par le protocole suivant :

1. si les deux opérandes sont égaux physiquement (c'est la même adresse), le test retourne `vrai` ;
2. si les deux opérandes ne sont pas de même type dynamique (par exemple, même table de méthode), le test retourne `faux` ;
3. finalement, la méthode `equals(SelfType)` est appelée.

Dans ce protocole, les étapes (1-2) ne sont pas à proprement parler objet, et l'étape (3) est sûre. Voir aussi la Section 8.2, page 152 qui développe d'autres pièges intéressants posés par l'égalité, ainsi que la Section 8.1, page 151 qui s'intéresse au traitement polymorphe des types primitifs.

7.9.4 Comparaison avec la covariance

Les types virtuels permettent la redéfinition covariante des types de paramètres, tout en conservant des types syntaxiquement invariants. Soit la méthode $m_A(t)$ dans la classe A et sa redéfinition $m_B(t')$ dans la sous-classe B de A, où t' est un sous-type strict de t. On peut remplacer cette redéfinition covariante explicite en définissant le type virtuel $T <: t$ dans A, sa redéfinition $T <: t'$ dans B et en adoptant la signature invariante $m_A(T)$ et $m_B(T)$. Le comportement sera exactement le même qu'en redéfinition covariante.

Si l'on redéfinit le type virtuel en le fixant par $T = t'$, le comportement reste identique à deux détails près : toute redéfinition ultérieure est impossible et le typage est sûr pour les appels de méthodes sur un receveur typé statiquement par B.

Méthodologie

D'un point de vue méthodologique, on procédera de la façon suivante :
— dans la classe qui introduit une méthode ou un attribut que l'on veut redéfinir de façon covariante, on introduit un type virtuel $T <: t$, et on prend T comme type de paramètre ;
— si jamais on veut redéfinir de façon covariante une méthode déjà introduite sans type virtuel, on simule la redéfinition covariante ;
— si plusieurs paramètres ou types de retour doivent être redéfinis en parallèle, on réutilise bien entendu le même type virtuel ;
En adoptant ces règles, le seul cas non sûr est le cas (5), lorsqu'il n'est pas une instance du cas (6).

Types ancrés d'Eiffel

Les types virtuels ressemblent beaucoup aux *types ancrés* d'Eiffel, `like Current` et `like foo`. Le type `like Current` est identique à `SelfType`. Pour `like foo`, la différence est qu'avec les types virtuels il n'est plus nécessaire de définir une propriété de référence et que l'on peut utiliser un type virtuel qui n'est le type de retour d'aucune propriété.

Les types virtuels sont préférables aux types ancrés dans la mesure où ils peuvent être fixés ($T = C$) pour toutes les sous-classes. La règle dite des « *catcalls* polymorphes » d'Eiffel (Erreur 5.5, Section 5.3.4, page 79) correspond à la règle (5) ci-dessus, mais elle ne peut s'appliquer qu'en compilation globale. De plus, les types ancrés ne sont jamais finaux et n'offrent donc pas la possibilité de la règle (4) pour rétablir la sûreté dans certaines sous-classes.

Exercice 7.11. Implémenter en Eiffel l'association (Exercice 7.4, page 133) et les graphes d'objets (Section 8.4, page 161) à l'aide des types ancrés. □

7.9.5 Les associations Uml

L'exemple de la section 7.6, page 131 s'exprime maintenant de façon très simple. On va définir :
— dans A, le type virtuel $TB <: B$;
— dans B, le type virtuel $TA <: A$;

et on va définir AA et BB comme sous-classes respectives de A et B, comme dans le cas de la covariance, avec :

— dans AA, le type virtuel $TB <: BB$;
— dans BB, le type virtuel $TA <: AA$.

Dans les sous-classes, on pourra aussi utiliser des types finaux (= au lieu de <:). C'est une question de sémantique : il faut juste se poser la question de savoir si l'association sera encore spécialisée ou pas.

Le type de `self` La question se pose de savoir s'il est nécessaire de définir un type virtuel *auto-référentiel* pour représenter la classe courante, c'est-à-dire :

— dans A, le type virtuel $TA <: A$;
— dans B, le type virtuel $TB <: B$.

Supposons que l'on veuille appliquer une étape de fermeture transitive à la relation, c'est-à-dire que l'on veuille les A qui sont en relation avec les B qui sont en relation avec un A. Dans A, ce serait typé par A et, dans AA, par AA. Mais ce ne serait pas pour autant typé par `SelfType` puisque qu'une sous-classe $AA1$ de AA ne redéfinit pas forcément TB. Pour avoir une bonne information de type, il faut donc rajouter le type TA dans A. On dispose alors de 3 types différents pour `self`, qui vérifient `SelfType` $<: TA <: A$. Mais la première relation de sous-typage n'est pas connue du compilateur qui, comme avec la généricité, ne sait pas que `self` $: TA$.

L'usage des ces 3 types est différent. TA doit s'utiliser pour tout ce qui est relation avec l'association, donc pour tout ce qui concerne les B. A et `SelfType` doivent s'utiliser comme pour une classe ordinaire, lorsque ça ne concerne pas l'association. L'introduction d'un mot-clé `fixpoint` (cf. Section 7.6.4, page 136) pour qualifier TA permettrait de rétablir l'information sur `self` $: TA$.

Le patron Stratégie. L'alternative avec le patron Stratégie (Section 7.7) s'adaptera de la même manière aux types virtuels.

7.9.6 Comparaison avec la généricité

Un type virtuel est très proche d'un type formel. La définition d'une classe paramétrée $A\langle T <: B\rangle$ équivaut à peu près à la définition de la classe non paramétrée A, avec un type virtuel $T <: B$.

Son instanciation $A\langle C\rangle$, avec $C <: B$, équivaut à la définition d'une sous-classe $A' <: A$, où le type virtuel a été redéfini par $T = C$.

Un type virtuel n'autorise donc pas une instanciation implicite comme un type paramétré. Mais les points fixes de la généricité F-bornée ne l'autorisent pas non plus. Et les instanciations explicites des *templates* C++ ne sont pas plus commodes.

Une autre façon de rapprocher généricité et types virtuels consiste à remarquer que les types formels sont des paramètres positionnels, alors que les types virtuels ressemblent à des paramètres passés par mots-clés, comme en Common Lisp.

Si l'on utilise des types virtuels finaux $T = C$, les types virtuels sont quasiment aussi sûrs que la généricité F-bornée mais, la syntaxe est beaucoup plus légère dans les cas effectivement récursifs et la spécialisation est conservée.

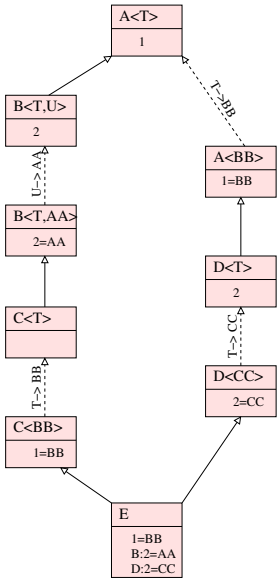
Choisir entre généricité et types virtuels Dans un langage qui propose à la fois la généricité et les types virtuels, comme Eiffel, Scala ou Prm/Nit, il peut être difficile de choisir entre les deux. Comme les type virtuels prennent en charge la covariance (sauf en Scala), on peut sans doute imposer d'abord des règles de typage strictes aux types paramétrés. Ensuite, trois cas se présentent :

— les associations Uml doivent utiliser les types virtuels ;
— les emplois simples de la généricité, quand on n'a besoin ni de covariance, ni de généricité F-bornée, utilisent la généricité ;
— qu'en est-il des autres cas ? quels sont-ils ?

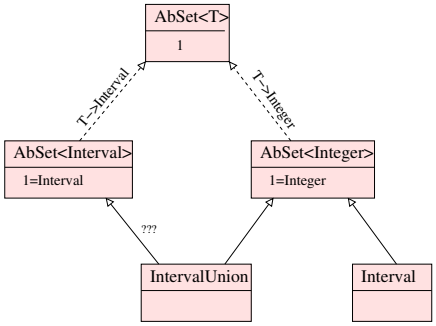
Combinaison de généricité et de types virtuels Que se passe-t-il si l'on souhaite mélanger types virtuels et généricité, par exemple pour définir une class paramétrée par un type dont la borne définit des types virtuels ?

Le cas de `Comparable` et `OrderedSet` peut s'analyser comme suit : soit une classe B (`Comparable`) qui définit un type virtuel auto-référentiel $T <: B$, et $A\langle U <: B\rangle$ une classe paramétrée (`OrderedSet`).

En généricité F-bornée, B serait défini comme $B\langle T <: B\langle T\rangle\rangle$ et le point fixe $B\langle B\rangle$ n'aurait pas de sens, et $A\langle B\rangle$ non plus. Seul les points fixes comme $C <: B\langle C\rangle$ sont licites et peuvent servir à instancier A pour obtenir $A\langle C\rangle$.



(a) Instanciations et spécialisations



(b) Implémentations d'ensembles

FIGURE 7.3 – Héritage multiple et généricité : (a) un premier type formel est introduit dans A, puis un second dans B d'un côté et dans D de l'autre ; (b) une union d'intervalles est-elle un ensemble d'intervalles ?

Avec les types virtuels, $A\langle B\rangle$ est apparemment licite, mais si C est maintenant une sous-classe de B qui fixe $T = C$, $A\langle C\rangle$ n'est pas un sous-type de $A\langle B\rangle$. Or, $A\langle B\rangle$ n'a pas de sens par lui-même : cela n'a pas de sens de faire un ensemble de `Comparable` qui ne sont pas `Comparable` entre eux.

La règle pourrait être la suivante. Si on appelle *classe auto-référentielle* une classe qui définit un type auto-référentiel (dont la borne est la classe considérée) *non final*, il serait interdit d'instancier une classe paramétrée par une classe auto-référentielle.

Noter que `SelfType` ferait de toute classe une classe auto-référentielle d'après la définition précédente. Il faut donc faire une exception pour `SelfType` : un type est auto-référentiel s'il a l'espoir d'être finalisé dans une sous-classe.

*****(à développer...)*****

7.9.7 Classes virtuelles

La notion de classe virtuelle est une fusion entre les notions de types virtuels et de classes internes. Alors qu'un type virtuel est utilisé pour désigner localement une classe définie ailleurs, une classe virtuelle est une classe interne qui peut être redéfinie dans les sous-classes de la classe englobante suivant un principe similaire aux types virtuels. On trouve cette notion dans les langages GBETA et SCALA.

*****(à développer...)*****

7.10 Types virtuels, généricité et héritage multiple

Les types virtuels offrent un bonne base pour analyser l'incidence de la généricité sur l'héritage multiple.

Un type virtuel est une propriété au même titre que les attributs et les méthodes. Les principes de l'héritage multiple valent donc aussi pour les types virtuels et l'on va rencontrer les deux catégories de conflit vues au chapitre 4. Un conflit de propriétés globales se résoudra comme pour les méthodes et les attributs. Le seul point spécifique concerne donc le conflit de propriétés locales.

7.10.1 Conflit de propriétés locales

Dans le cas des types virtuels, le conflit de propriétés locales peut présenter trois formes, suivant que les deux classes en conflit (les classes B et C de l'exemple du losange, Figure 4.1) définissent le type virtuel T par :

— $T <: t$ et $T <: t'$: cela suppose que, dans la sous-classe commune D, $T <: t \cap t'$ (on parle de *type intersection*) et que t et t' aient au moins une sous-classe commune.

— $T <: t$ et $T = t'$: cela suppose que $t' <: t$ et que $T = t'$ dans la sous-classe : sinon, c'est une erreur.
— $T = t$ et $T = t'$: cela suppose que $t' = t$: sinon, c'est une erreur.

Les deux premiers cas d'erreur sont des erreurs fatales à la compilation : il est impossible de définir la sous-classe commune.

Erreur à l'édition de liens. En revanche, dans le premier cas, on se retrouve dans la même situation que pour la covariance dans le cas général (Section 5.7). Une sous-classe commune doit exister mais il n'est pas nécessaire de l'imposer à la compilation : on peut attendre l'édition de liens. Noter que les types virtuels simplifient notablement la discussion : en effet, une sélection par linéarisation retournera une valeur du bon type si le type de retour est virtuel. C'est un autre argument pour utiliser également les types virtuels en tant que type de retour dès qu'il y a de l'héritage multiple.

7.10.2 Application à la généricité

On peut modéliser la généricité en associant à chaque type formel une propriété globale similaire à un type virtuel. La généricité présente cependant une originalité : la propriété globale sous-jacente n'est plus nommée et on ne peut l'identifier qu'à partir de sa position. Cette position doit tenir compte des paramètres formels déjà instanciés dans les super-classes. La figure 7.3a présente un exemple de cet héritage de types formels qui explicite les introductions et instanciations.

Le conflit de propriétés locales se traduit, lui, par une règle simple : il n'est pas possible de spécialiser deux classes qui instancient le même paramètre formel de façon différente. C'est l'exact analogue du troisième cas de conflit de types virtuels. Cette interdiction formelle s'applique lorsque le type formel est invariant. Dans le cas où le type formel est variant, par exemple dans l'exemple de `Comparable` de la Section 7.5.3, on peut avoir :

```
B <: A  <: Comparable⟨A⟩ <: Comparable⟨B⟩
```

où `A` et `B` sont des sous-types de deux instances génériques différentes de `Comparable`.

Dans le cas d'un type formel `Foo⟨T <: Foo⟨T⟩⟩` covariant, on pourrait alors avoir `A <:Foo⟨A⟩`, et `B` pourrait être déclaré comme un sous-type de `A` et de `Foo⟨B⟩` qui lui même est un sous-type de `Foo⟨A⟩`.

Pour un type formel invariant, une instance générique se traduit par une contrainté d'égalité, par exemple `T=A` ou `T=B`, dont la conjonction est contradictoire. En revanche, lorsqu'il y a variance, il faudrait plutôt considérer une contrainte d'inégalité, par exemple en cas de covariance, `T<:A` et `T<:B`, dont la conjonction est possible si `B<:A`. En cas de contravariance, l'inégalité est inversée. On voit ainsi qu'un type formel covariant se comporte comme un type virtuel, alors qu'un type formel contravariant est proprement irréductible.

Exercice 7.12. Implémenter l'exemple de la figure 7.3a en C++, Java ou Eiffel. En Java on se servira d'interfaces pour éviter la restriction sur l'héritage multiple. □

Exemple. On considère des ensembles. La classe abstraite `AbSet⟨T⟩` implémente l'interface abstraite des ensembles. On considère ensuite les ensembles d'entiers, qui peuvent être implémentés de différentes façons et sont tous sous-types de `AbSet⟨Integer⟩`. Parmi les implémentations intéressantes, il y a les `Intervalle`, lorsque les éléments de l'ensemble sont consécutifs. Lorsque les éléments sont presque consécutifs, une union d'intervalles, sous la forme d'un ensemble ordonné d'intervalles, constitue une implémentation efficace. La figure 7.3b présente ces quelques classes en omettant les ensembles ordonnés qui doivent s'intercaler entre `IntervalUnion` et les `AbSet`.

Exercice 7.13. Essayer d'implémenter l'exemple de la figure 7.3b en C++, Java ou Eiffel. En Java, on se servira d'interfaces pour éviter la restriction sur l'héritage multiple. En C++, on fera attention au mot-clé `virtual`.

Cela marche-t-il et comment se manifestent les éventuelles erreurs ? D'un point de vue mathématique, est-ce qu'une telle modélisation a un sens ? □

Les cas de C++ et de C♯

C++ faisant de la pure substitution textuelle, il n'a pas cette timidité théorique. Etant donné une classe paramétrée `Foo⟨T⟩`, avec 2 méthodes `foo(T)` et `bar():T`, et deux classes `A` et `B` incomparables, il est possible de définir une sous-classe `C` commune à `Foo⟨A⟩` et `Foo⟨B⟩`, à condition d'y redéfinir `bar()` avec un type de retour `D` qui soit un sous-type de `A` et `B`. De côté des méthodes `foo`, il s'agira juste de *surcharge statique*.

C♯ suit l'exemple de C++, mais comme il n'autorise pas la redéfinition covariante du type de retour, la présence de `bar()` n'est pas autorisée.

Dans tous les cas, on n'ose pas vraiment imaginer ce qu'il se passe si `Foo` fait aussi de la surcharge statique sur ... `foo` ! (cf. section suivante)

7.11 Généricité, types virtuels et surcharge statique.

La surcharge statique pose des problèmes intéressants en présence de généricité, et les spécifications de la première dépendent étroitement de celles de la seconde. Deux cas de figure se présentent :

— la classe paramétrée peut appeller une méthode potentiellement surchargée, avec un receveur ou un paramètre typé par un type formel ;

— la classe paramétrée peut définir une méthode surchargée, dont l'une des surcharges a un paramètre typé par le type formel.

Appel de méthode surchargée depuis une classe paramétrée

En effet, quand le receveur ou le paramètre d'un appel de méthode sont typés par un type formel, il n'est pas possible de connaître, statiquement, toutes les méthodes surchargées.

Considérons le type paramétré `Set⟨T⟩` qui spécifie les diverses implémentation des ensembles. Par définition, un ensemble est une collection sans doublons, qui possède une méthode `contains(T)`, pour vérifier l'appartenance d'un élément à l'ensemble, et cette méthode se sert d'un test d'égalité.

La question du test d'égalité est complexe et sera traitée en Section 8.2. Disons juste, ici, que ce test d'égalité peut être physique ou logique, et que la spécification des ensembles en Java utilise un test d'égalité logique, qui est implémenté par la méthode `equals`, définie dans `Object` avec un paramètre de type statique `Object` par l'égalité physique. La programmeur peut redéfinir et/ou surcharger `equals`, par exemple en définissant `equals(Point)` dans la classe `Point`, pour comparer les coordonnées de 2 points.

Le code de `Set⟨T⟩`, par sa méthode `contains`, appelle donc `equals` avec un paramètre de type statique `T`. Statiquement, quand on regarde le code de `Set` et que l'on ne connait pas l'existence de `Point`, on (programmeur ou compilateur) ne voit que `equals(Object)`.

Mais, dans le contexte de `Set⟨Point⟩`, on s'attendrait à ce que `equals(Point)` soit appelé. Et c'est bien ce qui se passerait en C++. En revanche, en Java, le type a été effacé et remplacé par la borne, `Object`. C'est donc `equals(Object)` qui est appelé.

Le cas de C♯ est plus délicat. En théorie, il pourrait faire comme C++, mais il faudrait pour cela que le système reconnaisse, en définissant (c'est-à-dire en le chargeant dynamiquement) `Set⟨Point⟩`, que la méthode `equals` a été surchargée dans `Point`, et qu'il doit donc recompiler les méthodes `add` et `contains` de `Set⟨Point⟩`. C'est probablement trop délicat pour qu'il le fasse et la spécification est donc, sur ce point, celle de Java.

Définition de méthode surchargée dans une classe paramétrée

Dans le cas de C++, l'instanciation générique peut aussi entraîner une ambiguïté de la surcharge. En effet, la classe `A⟨T⟩` peut définir deux méthode `foo(X)` et `foo(T)`, ce qui provoquera une double définition, donc une erreur de compilation, dans l'instance générique `A⟨X⟩`.

Types virtuels et surcharge statique.

Si les types virtuels sont intégrés dans un langage comme seule forme de covariance, les types de paramètres sont syntaxiquement invariants par redéfinition et ils peuvent très bien cohabiter avec la surcharge statique. Un cas d'ambiguïté nouveau peut apparaître si une classe a un type virtuel $T <: C$ et deux méthodes `foo(T)` et `foo(C)`. Comme `foo(T)` est a priori plus spécifique, c'est la signature qui doit être choisie. La coexistence de ces deux signature n'est pas erronée puisqu'il y a des contextes ou les deux peuvent être utilisées (par exemple dans le contexte d'une sous-classe qui redéfinit T). Cependant, comme pour la surcharge statique en général, on ne saurait encourager ce genre d'acrobatie.

7.12 Synthèse sur la méthodologie

De façon abstraite, on dispose ainsi de différents mécanismes qui se recouvrent en partie et ont des pouvoirs de modélisation similaires mais non équivalents :

— la redéfinition covariante ;

	covariance			généricité			
	native	simulée	types virtuels	hétéro	homo	bornée	F-bornée
sûreté (stricte)	−	−	−	+	+	+	+
sûreté (Cardelli)	+−	+	+	+	+	+	+
localité	−	+	+				
instanciation auto	−	−	−	+−	+	+	−
spécialisation / sous-typage	+	+	+	−	−	−	−
expressivité	+	+	+	+	−	−	+−
simplicité	+	−	+	+	+	+	−
efficacité	−	+	+	+−	+−	+−	+−
disponibilité	+−	+	+−	+	+	+	+

FIGURE 7.4 – Comparaison entre la covariance et la généricité

	types	généricité		effacement	annotation		covariance		
	virtuels	bornée	F-bornée	de type	joker	variance	généricité	types virt.	paramètres
C++	−	−	−	−	−	−	−	−	−
JAVA	−	+	+	+	+	−	+− [a]	−	−
C♯	−	+	+	−	−	+	+− [a]	−	−
SCALA	+	+	+	+	+	+	−	−	−
NIT	+	+	−	−	−	−	+	+	−
EIFFEL	+	+	+	−	−	−	+	+	+

a. Pour les tableaux seulement.

FIGURE 7.5 – Comparaison de la généricité suivant les langages

— la simulation de la redéfinition covariante ;
— les types virtuels ;
— la généricité, hétérogène ou homogène ;
— la généricité, bornée ou F-bornée.

On peut analyser et comparer ces mécanismes suivant les critères suivants (Figure 7.4) :

— sûreté du typage ;
— localité des erreurs de type dans le cas où ce n'est pas sûr ;
— instanciation automatique : faut-il définir explicitement les sous-classes ou instances génériques, ou leur définition n'est-elle qu'implicite ?
— la spécialisation ou le sous-typage entre les différents types ou classes produits sont-ils conservés ?
— expressivité : quels programmes peut-on écrire ?
— simplicité syntaxique (passage à l'échelle, discipline du programmeur) ;
— efficacité (voir Chapitre 9) ;
— disponibilité dans les langages *mainstream* actuels.

Enfin, la figure 7.5 compare les langages suivant la façon dont la généricité et la covariance y sont spécifiées.

On peut ainsi tirer diverses conclusions de ces comparaisons :

— les types virtuels sont la bonne façon de faire de la covariance mais ils ne sont pas disponibles dans les langages actuels ; ou plutôt, le seul langage « industriel » qui les propose, SCALA, n'en propose qu'une version sûre ;
— la généricité est très utile en complément des types virtuels, mais elle n'a sans doute pas besoin d'être F-bornée ;
— les types virtuels devraient être préférés à la généricité F-bornée dans la plupart des cas, mais à défaut, la généricité F-bornée est mieux que rien ;
— la covariance des instances génériques n'est sans doute pas utile et devrait être avantageusement remplacée par les annotations de variance et les jokers.

Pour un langage qui présenterait à la fois des types virtuels et de la généricité F-bornée, la généricité pourrait être spécifiée de façon complètement sûre, et les types virtuels serviraient à offrir l'expressivité suffisante, partiellement non sûre. La généricité F-bornée peut éventuellement être conservée pour les `OrderedSet` mais la spécialisation d'associations devrait passer par les types virtuels. SCALA et EIFFEL semblent être les seuls langages à avoir ces deux traits de langage, mais les types virtuels y sont sûrs dans le premier, et rien n'est sûr dans le second.

7.13 Addendum : variables et méthodes paramétrées

Jusqu'ici, on ne s'est intéressé qu'aux classes (ou types) paramétrés.
Les langages permettent souvent de paramétrer d'autres entités que les classes, par exemple :
— paramétrer des méthodes avec des paramètres supplémentaires (autres que ceux de la classe),
— paramétrer des méthodes ou variables statiques.

7.13.1 Méthodes paramétrées

Les méthodes et attributs sont implicitement paramétrés par la classe qui les définit — d'ailleurs, en C++, ce paramétrage doit être tout à fait explicite — et il n'est en général pas utile de paramétrer séparément des méthodes, c'est-à-dire d'ajouter à une méthode un ou plusieurs types formels qui ne seraient pas ceux de la classe englobante. Il y a néanmoins des cas où ce paramétrage est nécessaire, comme dans l'exemple suivant.

Exemple : la méthode `zip`. On définit une classe `Dico⟨T,U⟩` définissant des structures générales d'associations, permettant d'associer une valeur de type `U` à une clé de type `T`. On peut appeler ça des dictionnaires, ou dans le jargon moderne, des *maps*. C'est une généralisation des listes d'associations de Lisp et on peut les spécialiser par exemple par des tables de hachage (ou *hashmap*). Par ailleurs, on a une classe `List⟨T⟩` pour implémenter des listes quelconques. On voudrait alors faire une méthode `zip` de la classe `List⟨T⟩`, qui prenne un paramètre `List⟨U⟩` pour retourner un `Dico⟨T,U⟩` formé des paires successives des éléments des 2 listes [13]. Cette méthode `zip` peut être définie dans `List⟨T⟩`, mais il faut la paramétrer par `U`, soit `zip⟨U⟩(List⟨U⟩)`.

Surcharge statique. Les méthodes paramétrées se prêtent à la surcharge statique, suivant le nombre de types formels. Elles se prêtent aussi à l'inférence de types, les paramètres de types pouvant être déduits du type des paramètres, avec le même genre de règle du plus spécifique que pour la surcharge statique.

Par exemple, on pourra écrire `x:List⟨X⟩; zip(x);` au lieu de `x:List⟨X⟩; zip⟨X⟩(x);` le type formel de la méthode `zip` est déduit du type formel du type statique de son paramètre.

Possible en Java. L'implémentation homogène de la généricité en Java, par effacement de type, permet cette fonctionnalité. La syntaxe à l'appel est `<X>zip(x)`.

Impossible en C++. En revanche, l'implémentation hétérogène des *templates* de C++ ne permet pas de paramétrer les méthodes virtuelles, car chaque "instance" générique de la méthode devrait avoir sa propre implémentation et sa propre entrée dans les tables de méthodes de la classe. En C++, il faudrait donc passer par une fonction paramétrée statique.

En C♯. La spécification de la généricité n'a pas recours à l'effacement de types, mais cela n'empêche pas l'implémentation de pouvoir être homogène. La méthode paramétrée peut donc avoir une unique implémentation et entrée dans la table de méthodes, mais chaque appel doit passer en paramètre quelque-chose qui représente le type formel supplémentaire.

7.13.2 Variable statique dans une classe paramétrée

Les variables et méthodes statiques étaient censées, jusqu'à maintenant, être hors du modèle objet, la classe ne représentant qu'un système de rangement. La généricité impose néanmoins de relativiser cette idée.

En principe. Le principe des variables statiques est de désigner une donnée unique, en quelque sorte partagée par toutes les instances de la classe.

Dans une classe paramétrée, une variable statique dont le type serait fonction d'un type formel pose deux problèmes distincts :
— sa valeur ne peut pas être partagée par toutes les instances génériques de la classe, puisque chacune imposerait son propre type a priori incomparable avec les autres ;
— sa valuation nécessite de pouvoir faire `new T` ou `new A⟨T⟩`.

13. En Lisp, cela s'exprimerait par un `(mapcar #'cons <liste1> <liste2>)`.

Inversement, si le type de la variable n'est pas fonction d'un type formel, rien n'empêche de partager la variable entre toutes les instances génériques, mais il est clair que cela permet 2 spécifications alternatives.

Dans tous les cas, c'est l'implémentation qui va dicter en grande partie la spécification du langage.

En Java. Une variable statique ne peut pas être fonction d'un type formel et reste partagée par toutes les instances génériques de la classe. C'est une conséquence directe de l'effacement de type et de l'implémentation homogène.

En C♯. Une variable statique reste propre à chaque instance générique, même si elle n'est pas fonction d'un type formel. L'implémentation hétérogène permet d'avoir une variable statique par instance générique et donc de traiter le cas où un type formel sert à typer la variable statique. Le fait d'imposer cette hétérogénéité au cas où aucun type formel n'est utilisé résulte vraisemblablement d'une volonté d'homogénéité, mais c'est au prix d'une autre hétérogénéité, puisqu'une classe paramétre ne se comporte plus comme une classe normale. La spécification alternative aurait été possible.

7.13.3 Méthode statique dans une classe paramétrée

Paramétrer une méthode statique semble a priori possible sans autre limitation que celles de l'implémentation sous-jacente de la généricité. Ce paramétrage peut aussi bien être restreint aux types formels de la classe englobante qu'ajouter des types formels. Et ceci aussi bien dans une implémentation homogène que dans une implémentation hétérogène comme en C♯. Dans ce dernier cas, l'implémentation hétérogène peut conduire à générer dynamiquement des instances génériques de la méthode, mais c'est inhérent à la compilation dynamique.

En Java. Malgré cette possibilité théorique, les spécifications de Java ne permettent pas d'utiliser dans une méthode statique les paramètres de type de la classe englobante. On peut trouver cela bizarre, puisqu'il suffit de les déclarer explicitement pour obtenir le résultat souhaité. C'est cependant relativement logique, ne serait-ce que parce qu'il n'y a aucune logique dans les propriétés statiques : leur présence dans une classe est une question de rangement, et éventuellement de droits d'accès. Une méthode statique paramétrée (ou non) peut donc être rangée dans une classe paramétrée (ou non), et les deux paramétrages sont complètement indépendants. Le cas des variables statiques est similaire, en Java, en ce qu'ils ne sont pas paramétrés.

En C♯. En revanche, en C♯, les attributs sont explicitement spécifiés comme dépendants de chaque instance générique, donc avec le paramétrage implicite de la classe englobante. Il en est de même pour les méthodes statiques, et c'est logique si l'on raisonne en termes d'équivalence attribut-méthode au travers des accesseurs. Du coup, si le programmeur veut définir une méthode statique avec un autre paramétrage, qu'il aille la ranger ailleurs !

7.14 En savoir plus

Indépendamment de la programmation par objets, on retrouve la généricité dans un certain nombre de langages à typage statique, Ada, déjà nommé, ou Haskell [Peyton Jones, 2003]. La confusion des concepts fait beaucoup de programmeurs génériques croient qu'ils programment objet.

La conception de langage de programmation est un art difficile : le choix entre les diverses alternatives présentées dans ce manuel n'est pas évident. Une solution serait de les inclure toutes, mais le choix se reporterait alors sur le malheureux programmeur...

La distinction entre les implémentations homogène et hétérogène de la généricité a été faite par [Odersky and Wadler, 1997]. Il reste bien entendu un grand espace entre ces deux extrêmes, que .Net semble avoir rempli (mais c'est à vérifier en détail).

Les types virtuels ont été introduits dans le langage Beta [Madsen et al., 1993], mais ce langage est relativement peu utilisé et, en tout cas, peu connu en France. Ils n'existent pas en C++ et les concepteurs de Java leur ont préféré la généricité pour la version 1.5. Nous les avons présentés dans la version proposée par [Torgersen, 1998, Thorup and Torgersen, 1999]. Les types virtuels sont proches des *types ancrés* d'Eiffel mais, avec ces derniers, l'absence de types finaux et la covariance généralisée ne permettent pas d'en apprécier tout l'intérêt. Ne pas confondre aussi *type virtuel* et *classe virtuelle* : les langages de la famille Beta combinent les fonctionnalités des types virtuels et des classes internes (*inner classes*). Noter qu'il s'agit ici du seul cas où nous considérons qu'une entité puisse être spécifiée comme

finale. C'est en effet une infraction au Méta-axiome 2.2, page 21, que nous considérons comme légitime compte-tenu du coût d'implémentation et du risque d'erreur causés par une covariance généralisée.

[Altidor et al., 2012] analyse les usages des *jokers* de JAVA et des annotations de variance de C♯ ou SCALA, et en propose l'intégration.

Ce chapitre clos l'analyse du typage et de la covariance présentée dans les chapitres précédents et recoupe pour l'essentiel celle de [Shang, 1996]. La section sur les types virtuels doit être considérée comme un *addendum* ou un *erratum* de [Ducournau, 2002b,a] — leur potentiel pour traiter la covariance et remplacer les types ancrés d'EIFFEL m'avait à l'époque échappé.

Bibliographie

John Altidor, Christoph Reichenbach, and Yannis Smaragdakis. Java wildcards meet definition-site variance. In *Proc. ECOOP'12*, pages 509–534. Springer-Verlag, 2012. doi : 10.1007/978-3-642-31057-7_23.

M. Day, R. Gruber, B. Liskov, and A. Myers. Subtypes vs. where clauses. constraining parametric polymorphism. In *Proc. OOPSLA'95*, SIGPLAN Not. 30(10), pages 156–168. ACM, 1995.

R. Ducournau. "Real World" as an argument for covariant specialization in programming and modeling. In J.-M. Bruel and Z. Bellahsène, editors, *Advances in Object-Oriented Information Systems, OOIS'02 Workshops Proc.*, LNCS 2426, pages 3–12. Springer, 2002a.

R. Ducournau. Spécialisation et sous-typage : thème et variations. *Revue des Sciences et Technologies de l'Information, TSI*, 21(10) :1305–1342, 2002b.

P. Gochet and P. Gribomont. *Logique, méthodes pour l'informatique fondamentale*, volume 1. Hermès, Paris, 1991.

A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Proc. PLDI'01*, SIGPLAN Not. 36(5), pages 1–12. ACM, 2001.

O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

M. Odersky and P. Wadler. Pizza into Java : Translating theory into practice. In *Proc. POPL'97*, pages 146–159. ACM, 1997.

Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

David L. Shang. Are cows animals ? http://www.visviva.3d-album.com/transframe/papers/covar.htm, 1996. URL http://www.visviva.3d-album.com/transframe/papers/covar.htm.

S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, and P. Leroy, editors. *Ada 2005 Reference Manual : Language and Standard Libraries*. LNCS 4348. Springer, 2006.

K.K. Thorup and M. Torgersen. Unifying genericity : Combining the benefits of virtual types and parameterized classes. In R. Guerraoui, editor, *Proc. ECOOP'99*, LNCS 1628, pages 186–204. Springer, 1999.

Mads Torgersen. Virtual types are statically safe. In *Elec. Proc. of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL 5)*, 1998.

Mirko Viroli and Giovanni Rimassa. On access restriction with java wildcards. *Journal of Object Technology*, 4(10) :117–139, 2005.

8

Mélanges typés

En plus des grands principes énumérés dans les chapitres précédents, on trouve dans les langages de programmation par objets de nombreuses fonctionnalités qui mettent en jeu, simultanément, différents aspects du typage statique. C'est un catalogue à la Prévert, qui traite aussi bien de l'intégration des types primitifs, en particulier dans la généricité, à la spécification conjointe des tests d'égalité logique et du hachage d'objets, en passant par l'implémentation de collections immutables.

Le chapitre se termine par une application cohérente de l'implémentation des associations UML, les graphes d'objets.

8.1 Intégration des types primitifs

Dans les langages objets idéaux comme SMALLTALK, toutes les valeurs sont des objets, y compris lorsque leur type est primitif (`int`, `float`, etc.) Comme, par définition, les valeurs primitives ne sont pas des objets construits, il s'agit donc à la fois d'un artifice d'implémentation et d'une astuce du compilateur qui fait les conversions nécessaires lorsque cela s'impose.

Les langages comme EIFFEL ou PRM présentent une intégration des types primitifs dans les classes qui réalise cet idéal. En revanche, C++ maintient 2 mondes complètement distincts, et JAVA et C♯ les intègrent très superficiellement.

8.1.1 Traitement polymorphe des types primitifs (*boxing*)

Le problème fondamental des types primitifs est de pouvoir les manipuler de façon polymorphe, dans les deux grandes formes que l'on a vues du polymorphisme, le sous-typage et la généricité. Dans le premier cas, on veut faire des classes qui généralisent les types primitifs, par exemple une classe `Number` qui généralise `int` et `float`. `Object` serait bien entendu la généralisation ultime.

Dans le second cas, on veut faire des classes paramétrées générales, qui s'appliquent aussi aux types primitifs. On veut pouvoir faire des piles d'entiers (`Stack⟨int⟩`), mais aussi des vecteurs (au sens mathématique du terme) d'entiers (`Vector⟨int⟩`) et de flottants (`Vector⟨float⟩`) qui reposent sur une implémentation générique de l'algèbre linéaire.

Principe du *boxing*. En pratique, pour qu'un `int` puisse être considéré comme un `Object` (par exemple en affectant le premier à une variable typée par `Object`), il faut transformer sa représentation de façon à l'aligner sur celle des objets construits (voir Chapitre 9). Cela consiste à mettre l'entier dans une « boîte », c'est-à-dire un objet muni d'un unique attribut de type `int`, pour en faire un objet « normal » équipé d'une table de méthodes. Il s'agit ici de conversions et non pas de coercition (*cast*) que l'on appellent *boxing* et *unboxing*.

Dans le système de types des langages considérés, un *type emboîté* correspond à chaque type primitif, et le code compilé doit effectuer les conversions de *boxing* et *unboxing*.

Test d'égalité des boîtes. Dans le contexte des types emboîtés, le test d'égalité se complique puisqu'il faut tester non pas l'égalité des contenants, mais celle des contenus. S'il s'agissait simplement d'égalité logique, il suffirait d'appliquer, dans un cas particulièrement simple, la méthode `equals` suivant le protocole décrit en Section 8.2, page 152. Mais il s'agit ici d'égalité physique, qui est supposée très efficace et ne peut pas passer par un appel de méthode. Il faut donc certainement se résoudre à ce que le compilateur génère des tests d'égalité physique plus compliqués lorsque les arguments peuvent être des boîtes, c'est-à-dire lorsque les types statiques sont des super-types des types primitifs, en testant successivement

1. l'égalité physique,
2. sinon, si l'argument est une boîte,
3. et si les deux arguments sont de même type,
4. et si les contenus sont égaux.

Comme le dernier test dépend de la taille du contenu, il est sans doute plus simple de procéder comme suit : on implémente l'égalité physique comme une égalité logique pour les boîtes, avec une méthode par type de boîte, définie par défaut comme l'égalité physique dans `Object`. Et le compilateur remplace l'appel à cette méthode par un test d'égalité physique lorsque les types statiques ne sont pas super-types des types primitifs.

8.1.2 Types emboîtés implicites en Eiffel et Prm

En Eiffel et Prm, ces types emboîtés ne sont pas explicités et les conversions restent implicites : c'est le compilateur qui s'en charge. De plus, il existe différentes classes de nombres permettant de définir une arithmétique générique qui peut ensuite être utilisée dans des classes paramétrées. On peut donc supposer que, dans ces langages, le test d'égalité physique sur les boîtes est implémenté correctement.

Remarque 8.1. Pour les types primitifs qui s'implémentent sur strictement moins qu'un mot mémoire, il peut être plus efficace de recourir à la technique d'implémentation des valeurs immédiates en typage dynamique : quelques bits du mot mémoire, par exemple un octet, sont utilisés pour encoder le type de la valeur.

8.1.3 Types emboîtés explicites en Java et C♯

En Java et C♯, cette intégration est loin d'être aussi transparente.

Jusqu'en Java 1.4, les types primitifs comme `int` coexistaient avec leur version emboîtée (`Integer`), charge au programmeur de tout expliciter, depuis l'instanciation de `Integer` jusqu'aux conversions dans les 2 sens.

A partir de Java 1.5 et de l'introduction de la généricité, le compilateur Java rend les conversions implicites (on appelle cela *autoboxing*), au moins dans certains cas, mais les types primitifs ne sont toujours pas de vrais sous-types de `Object`, la racine de la hiérarchie. De plus, la généricité ne permet pas d'instancier un type formel par un type primitif : il faudra utiliser `Stack⟨Integer⟩` au lieu de `Stack⟨int⟩`, mais on pourra lui passer directement la valeur `92`, sans avoir à expliciter `new Integer(92)`.

Enfin, en C♯, la situation est identique à celle de Java en ce qui concerne le sous-typage : les types primitifs ne sont pas des sous-types de `Object`. De plus, les types emboîtés sont explicites mais les conversions sont implicites, comme en Java 1.5. En revanche, comme la généricité est spécifiée sans effacement de type, la définition de `Stack⟨int⟩` est possible et elle n'implique aucune conversion implicite ou explicite.

En Java, il aurait été possible d'accepter `Stack⟨int⟩`, mais cela aurait été du sucre syntaxique pour `Stack⟨Integer⟩`.

Quant au test d'égalité physique, il est à craindre qu'il ne soit réalisé sur les boîtes comme pour les objets ordinaires.

8.2 Test d'égalité et hachage

> [...] dire d'une chose quelconque qu'elle est identique à elle-même est évidemment trivial.
> W.V. Quine, « Identité » in *Quiddités. Dictionnaire philosophique par intermittence*

8.2.1 Test d'égalité

Les prédicats d'égalité sont nécessaires dans tous les langages de programmation, mais leur spécification précise pose de nombreuses difficultés. On distingue en gros deux catégories de prédicats d'égalité, pour l'égalité physique et l'égalité logique :

— le prédicat d'égalité physique teste en gros l'égalité de référence, c'est-à-dire si c'est le même objet ;
— le prédicat d'égalité logique teste s'il s'agit d'objets similaires ou isomorphes.

Ainsi en Lisp, on dispose de différents prédicats :

— le prédicat d'égalité physique est (`eq`),
— le prédicat d'égalité logique sur les listes (`equal`) teste l'égalité de structure de deux listes,

— un prédicat spécifique à chaque type ou famille de types teste l'égalité spécifique à la famille (égalité numérique, de chaîne de caractères, etc.)

De façon générale, et si l'on exclut les cas particuliers comme l'égalité des boîtes (Section 8.1), l'égalité physique consiste à comparer deux chaînes de bits qui représentent une valeur immédiate ou une référence. Quant à l'égalité logique, ses spécifications imposent des propriétés mathématiques (c'est une relation d'équivalence) qui sont non triviales. En particulier, par réflexivité, l'égalité physique doit rester un cas particulier de l'égalité logique.

Dans les langages de programmation par objets, la spécification se complique du fait que

— les égalités logiques spécifiques sont unifiées en un prédicat unique, spécifié dans la racine de la hiérarchie et redéfini dans les sous-classes ;

— en typage statique, cette redéfinition peut donner lieu aux problèmes de surcharge statique et de redéfinition covariante du Chapitre 6 ;

— les classes de la bibliothèque (les collections par ex.) peuvent se servir de l'un ou l'autre de ces prédicats d'égalité, ce qui met en jeu la généricité (Chapitre 7) ;

— en particulier, on peut se retrouver avec les problèmes combinés de la généricité et de la surcharge statique (Section 7.11), qui ne se résoudront pas de la même manière en Java et en C++, C♯ s'alignant sur ce point sur le premier.

Rien que sur ce point, la spécification et l'usage du test d'égalité est d'une grande complexité. Le risque est donc grand qu'il soit mal compris, donc mal utilisé, que ce soit par les programmeurs de base ou par les concepteurs/programmeurs de bibliothèques ou de *framework*.

8.2.2 Hachage

Le test d'égalité est d'autant plus compliqué qu'il est utilisé dans le hachage.

Le hachage est une technique qui permet d'implémenter des collections avec une efficacité qui peut se caractériser comme un temps constant en moyenne, et donc sous réserve d'une heureuse distribution des objets ou valeurs hachés.

Le principe du hachage consiste à associer un petit entier `hash(x)` à chaque objet `x`, en essayant de vérifier des propriétés de distribution qui réduisent au maximum les collisions, c'est-à-dire le fait que `hash(x)=hash(y)` lorsque `x`≠`y`.

Une table de hachage, un `HashSet` par exemple, sera alors implémentée de façon à ce que la recherche de `x` dans l'ensemble se fasse principalement sur les éléments `y` tels que `hash(x)=hash(y)`. En pratique, il y a de nombreuses variantes mais ce n'est pas le lieu d'en discuter ici.

Le point crucial est double :

— les structures de hachage ont besoin d'un test d'égalité ;

— la hachage doit être robuste à la gestion automatique de la mémoire.

Hachage et test d'égalité. Le test d'égalité utilisé peut être l'égalité physique ou logique. On peut imaginer faire 2 classes différentes (mais cela entraînerait de l'héritage multiple).

En Common Lisp, le prédicat d'égalité peut être passé en argument lors de la création d'une table de hachage. On pourrait donc imaginer une solution similaire, mais elle serait inadaptée en Java où il n'est pas possible de passer une fonction en argument. En Scala, où les fonctions sont des objets de première classe, ou en C♯, avec la notion de *delegate*, cela pourrait être possible.

Sinon, dans les langages actuels, la plupart des structures de hachage utilisent le test d'égalité physique. Java semble la seule exception, où les `HashSet` utilisent le test d'égalité logique.

Dans tous les cas, si un objet est placé dans une table de hachage à l'instant t, il doit pouvoir y être retrouvé à l'instant $t+1$, c'est-à-dire que sa valeur de hachage doit être immutable. Le principe essentiel du hachage d'objets est donc qu'il repose sur des propriétés immutables de l'objet, que ce soit son adresse, si l'objet n'est pas déplaceable, ou certaines de ses propriétés.

Hachage et *garbage collection*. La gestion automatique de la mémoire repose sur un *garbage collector* dont il existe différents types. Du point de vue qui nous intéressent, il y en a deux : ceux qui déplacent les objets et ceux qui ne les déplacent pas. La propriété principale est la suivante : si les objets sont déplacés en mémoire, ils ne doivent pas être déplacés dans une table de hachage.

Lorsque le *collector* ne déplace pas les objets, il n'y a pas de problème et le hachage peut utiliser l'adresse des objets.

En revanche, si le *collector* déplace les objets, leur adresse n'est plus *immutable* et le hachage ne peut pas être basé sur les adresses. Dans ce cas, le test d'égalité physique utilisé par le hachage se sert à la place d'un numéro immutable (mais pas forcément injectif) qui est affecté aux objets lors de leur création. Cela induit bien sûr un surcoût, puisque chaque objet hachable doit posséder un

attribut supplémentaire. Le surcoût pourrait être réduit en spécifiant une interface `IHashable` qui serait indispensable pour qu'un objet puisse être utilisé comme clé dans une structure de hachage, mais cette usage dépend sans doute plus de l'utilisation des classes que de leur spécification intrinsèque.

8.2.3 En JAVA : cohérence des méthodes `equals` et `hashCode`

En JAVA, une méthode `boolean equals(Object)` est définie dans la classe `Object` où elle se comporte comme un test d'égalité physique (==). La redéfinition de `equals` dans les sous-classes permet d'implémenter le prédicat d'égalité logique spécifique à chaque type.

En parallèle, la méthode `int hashCode()` est définie dans `Object` pour calculer la valeur de hachage d'un objet. Cette valeur de hachage est utilisée par les tables de hachage (par exemple `HashSet`) et la méthode est redéfinissable dans toutes les classes. Dans la classe `Object`, la définition de `hashCode`, suivant les implémentations, hache l'adresse de l'objet ou retourne un numéro immmutable affecté à l'objet lors de sa création (cela dépend essentiellement du type de *garbage collector*).

La spécification du langage impose explicitemnt que ces deux méthodes soient cohérentes, c'est-à-dire que `p.equals(q)` implique `p.hashCode() == q.hashCode()`. C'est une conséquence inéluctable de toute la discussion qui précède.

Lorsque le programmeur redéfinit `equals`, il doit redéfinir `hashCode` en parallèle.

Contrainte 8.1 *Les méthodes `equals` et `hashCode` doivent être redéfinies en parallèle dans chaque classe où elles sont redéfinies et faire appel aux valeurs des mêmes attributs. `p.equals(q)` implique `p.hashCode() == q.hashCode()`*

Exemple : la classe `Point`. On définira par exemple la classe `Point` des points à 2 dimensions, avec 2 attributs entiers `x` et `y`, et des accesseurs `int getX` et `setX(int)` (et idem pour `y`).

L'égalité logique de 2 points peut alors se spécifier comme l'égalité 2 à 2 des attributs `x` et `y`, et se définir par la méthode

```
public boolean equals(Point p) {
   return (getX() == p.getX() &&  getY() == p.getY())}
```

Bien entendu, la surcharge statique va provoquer des ambiguïtés. Ainsi, `new Point(1,2).equals(new Point(1,2))` va retourner vrai, alors que `new Point(1,2).equals((Point)new Point(1,2))` ou `(Point)new Point(1,2).equals(new Point(1,2))` vont retourner faux. Surcharger `equals` n'est donc qu'une solution partielle, et il faut impérativement pratiquer la simulation de la covariance, en retournant faux quand le test de sous-typage échoue (NB `equals` est un prédicat, c'est-à-dire une fonction à valeur booléenne, qui n'a aucune raison de signaler une exception).

Contrainte 8.2 *Les redéfinitions de `equals` doivent se faire avec simulation de la covariance.*

En parallèle, le programmeur doit redéfinir `hashCode` dans `Point`, pour que la valeur de hachage soit une fonction de `getX()` et `getY()`.

Les pièges ne sont pas terminés pour autant. En effet, si les coordonnées `x` et `y` du point sont mutables, les modifier va changer la valeur de hachage du point, ce qui va invalider sa position dans une structure de hachage. Modifier les coordonnées d'un élement d'un `hashSet` va faire qu'on ne peut plus le retrouver dans le `hashSet`. C'est bien sûr inacceptable.

Contrainte 8.3 *Les définitions des méthodes `equals` et `hashCode` ne doivent faire référence qu'à des attributs immutables.*

8.2.4 Spécification correcte (mais spéculative) de `equals`

Symétrie

Le problème n'est pas encore complètement réglé, car nous n'avons pas encore considéré la question de la symétrie. On pourrait ainsi vouloir tester `p1.equals(p2)`, où `p1` serait une instance directe de `Point` et `p2` une instance de `point3D`, une sous-classe de `Point`. Par symétrie, la réponse devrait être faux, même si les 2 points ont les mêmes coordonnées `x` et `y`.

Contrainte 8.4 (Symétrie) *`equals` ne retourne vrai que pour des instances d'une même classe.*

Cela n'assurera pas une totale symétrie (rien ne peut empêcher le programmeur d'implémenter le prédicat d'égalité comme une inégalité par exemple), mais cela impose les contraintes minimales à assurer quelles que soient les classes considérées. On suppose donc qu'il existe un opérateur ou une fonction statique `hasSameType`, à valeur booléenne.

Réflexivité

De plus, il est inutile de faire des comparaisons coûteuses sur les attributs (`equals` peut être récursif, comme `equal` en LISP), lorsque les deux objets sont le même. Outre l'obligation de respecter la réflexivité des relations d'équivalence, des raisons d'efficacité font que le test d'égalité logique devrait toujours commencer par un test d'égalité physique.

Contrainte 8.5 (Réflexivité) *`equals` doit commencer par un test d'égalité physique.*

Transitivité

Bien entendu, l'égalité doit aussi être *transitive*. Cependant, cette propriété découlera vraisemblablement naturellement de la transitivité des opérations d'égalité des définitions spécifiques de `equals` : en toute logique, la définition d'une méthode `equals` particulière devrait faire appel récursivement à `equals` ou à des prédicats d'égalité plus spécifiques [1]. Là encore, il n'est pas possible de forcer le programmeur à une implémentation transitive.

Le cas de `null`

Enfin, il faut traiter le cas où l'une des deux valeurs est `null`. En effet, `equals` étant un prédicat, il serait logique qu'il retourne une valeur booléenne dans le cas où l'un des deux objets, voire les deux, est nul : `null.equals(null)` devrait ainsi retourner vrai.

Protocole d'égalité

Au total, la façon correcte de spécifier `equals` serait donc de définir, dans `Object`, une méthode statique `equals`, plus une méthode redéfinissable `logicEquals`. La première fait successivement :

— un test d'égalité physique, pour retourner vrai,
— une comparaison avec `null` du receveur et du paramètre, pour retourner faux,
— un test d'inégalité des types, pour retourner faux,
— enfin, l'appel de `logicEquals`.

La méthode `logicEquals`, de son côté, peut supposer que son paramètre a toujours le même type que le receveur et elle est redéfinissable dans toutes les sous-classes. Idéalement, on utilisera un paramètre typé par le type virtuel `SelfType`. Bien que ce soit en général non sûr, son utilisation dans le cas présent serait parfaitement sûre, puisque l'appel de `logicEquals` est protégé par un test d'égalité de types [2].

De plus, un bon compilateur aura tout intérêt à expanser en ligne (*inlining*) l'appel de `equals`, et les deux premiers tests pourront économiser l'appel effectif de `logicEquals` dans de nombreux cas.

Dans les langages actuels, qui n'autorisent pas `Selftype`, le type du paramètre de `equals` devrait être `Object`, ce qui va imposer un test de sous-typage coûteux, mais qui réussira toujours :

EXERCICE 8.1. Définir `equals` dans `Object`, ainsi que `logicEquals` dans `Object`, `Point` et `Point3D`. □

Un dernier point : pour empêcher le programmeur d'utiliser directement `logicEquals` à la place de `equals`, le compilateur doit détecter que l'appel n'est pas sûr à cause de la covariance de `Selftype`. L'argument échoue, bien entendu, si `Selftype` est remplacé par `Object`. Une alternative consisterait à étendre les mécanismes de droits d'accès statique, par exemple les clauses d'`export` d'EIFFEL, aux méthodes : L'appel de `logicEquals` serait déclarée réservée à `equals` dans `Object`, alors que la méthode resterait connue et redéfinissable par toutes les classes.

Remarque 8.2. Du fait du nombre différent de leurs paramètres, la surcharge statique permettrait d'utiliser le même nom, `equals`, pour la méthode `static Object.equals(Object,Object)` et pour la méthode "virtuelle" `Object.equals(Selftype)`.

Egalité partielle

Beaucoup de programmeurs se servent de la méthode `equals` de façon purement adhoc, pour pouvoir faire une recherche spécifique dans des collections. Nous en avons même vu utiliser `equals` pour faire un test d'inégalité !

Il est néanmoins possible de définir d'autres prédicats d'égalité, voire d'inégalité. On peut ainsi souhaiter définir une méthode d'égalité partielle, par exemple pour vérifier qu'un `Point2D` et un `Point3D`

1. Mais dans ces derniers cas, le compilateur devrait être capable de remplacer un appel générique à `equals` par l'appel d'un prédicat spécifique à partir du type statique des arguments.
2. Il faudra néanmoins convaincre le compilateur de l'accepter...

ont la même projection. On pourrait faire une méthode `partialEquals(SelfType)`, mais les erreurs de covariance feraient leur retour, puisque cela revient à enlever la garde `hasSameType`. Une solution serait de spécifier ce test d'égalité partielle de la façon suivante :

— si les types dynamiques des 2 paramètres ne sont pas comparables, on retourne faux ;
— sinon, on applique `sup.logicEquals(sub)`, où `sup` est le paramètre de type le plus général, et `sub` celui de type le plus spécifique.

Cette garde éliminera tout risque d'erreur de type.

On suppose qu'il existe un opérateur (ou une méthode statique) `hasSubType` qui teste que les premier opérande a un type dynamique plus spécifique que le second. En pratique, étant donné deux paramètres `x` et `y` dont on veut tester l'égalité partielle, on aboutit au protocole suivant :

— si `x hasSubType y`, alors `x.logicEquals(y)` ;
— si `y hasSubType x`, alors `y.logicEquals(x)` ;
— sinon le test échoue.

Remarque 8.3. Bien entendu, la présence simultanée de plusieurs méthodes d'égalité différentes suppose que l'on soit capable de

— créer des collections dont les éléments utiliseront un test d'égalité particulier,
— faire des recherches sur une collection donnée, en utilisant un test d'égalité différent de celui qui a été spécifié lors de la création de la collection.

Le premier cas revient à paramétrer des classes par autre-chose que des types : c'est commun avec les *templates* C++, mais impossible avec Java ou C♯. Le deuxième cas peut se faire avec des méthodes de recherche dotées d'un paramètre supplémentaire. On simplifierait beaucoup de choses en introduisant des paramètres optionnels dans les langages.

8.2.5 Synthèse des contraintes

`x.equals(x) = true`	*réflexivité*
`x.equals(y) = y.equals(x)`	*symétrie*
`x.equals(y) & y.equals(z) ⇒ x.equals(z)`	*transitivité*
`x.equals(y) ⇒ x.hashCode()=y.hashCode()`	*hash code*
`x.equals(y) ⇒ x.hasSameType()`	*même type dynamique*

Pour être complet, il faut souligner que ces contraintes s'appliquent aussi au cas où `x=null`, sans entraîner d'exception. Il faut donc supposer que les méthodes `equals`, `hashCode`, `hasSameType` font partie de l'interface de `null`, de même que des méthodes comme `toString`.

8.2.6 Cas des structures circulaires

La spécification précédente d'`equals` s'applique à des structures arborescentes, ou plus généralement avec des cycles (non orientés) mais pas des circuits (orientés). Comme la fonction Lisp `equal`. En cas de structure circulaire, ce qui est courant en programmation objet avec les associations, la définition précédente partira en boucle infinie sur des objets isomorphes, ou même partiellement isomorphes. Il faut donc spécifier deux nouvelles méthodes, `circEquals` et `logicCircEquals`, qui sont dans un rapport analogue à `equals` et `logicEquals`.

En entrant dans `logicCircEquals`, il faut d'abord vérifier que l'on n'est pas en train de boucler, c'est-à-dire de déjà tester l'égalité des 2 paramètres. Pour cela, il faut utiliser une table de hachage dans laquelle on associera les 2 paramètres en entrant dans `logicCircEquals`. Lorsqu'en entrant dans la méthode `logicCircEquals` les deux paramètres sont déjà associés, la méthode retourne vrai. Attention ! pour éviter à nouveau une boucle infinie, cette table de hachage doit utiliser une test d'égalité physique, pas logique.

Enfin, cette table de hachage doit être unique durant toute la récursion de `logicCircEquals` : la table de hachage doit donc être allouée dans `circEquals`, puis passée comme paramètre supplémentaire à `logicCircEquals`. Par défaut, dans toutes ses redéfinitions qui vont provoquer un appel récursif, `logicEquals(x,y,ht)` doit retourner vrai si `y` est déjà associé à `x` dans `ht` ; sinon, l'association doit être faite, la condition spécifique est testée. Avant de sortir, on peut se poser la question de défaire l'association. Ce dernier point est discutable : au mieux, c'est une question de compromis entre la taille de la table, si on stocke tout, et les éventuels recalculs, si on repasse par les mêmes paires d'objets.

Enfin, l'égalité des structures circulaires pose un problème de définition non trivial : c'est une question de morphisme, mais on peut vouloir vérifier un isomorphisme, c'est-à-dire une relation bijective entre les objets des deux structures circulaires, ou pas. Par exemple, une liste circulaire de 1, en Lisp peut se faire avec un nombre quelconque de cellules : les listes `(1)`, `(1 1)` et `(1 1 1)` peuvent toutes est

rebouclées sur elles-mêmes. Elles sont en relation d'égalité logique, mais pas isomorphes. La spécification précédente doit être complétée pour avoir un isomorphisme, il faut que la table de hachage ne soit jamais vidée, que `x` soit aussi associé à `y` dans une deuxième table de hachage, et que la présence d'une autre association de l'un des deux arguments fasse retourner faux.

Il s'agit au total d'une opération coûteuse dont la sémantique précise est multiple et l'implémentation est délicate.

8.2.7 Objet vs. valeur

La question de l'égalité recouvre en fait celle de l'identité des objets, et celle-ci n'est pas sans rapport avec la distinction entre valeurs et objets. Considérés comme des valeurs, c'est-à-dire des objets mathématiques, les objets devraient être immutables et interchangeables à une égalité logique près. Les implémentations des nombres, qu'il s'agissent de petits entiers, de `bignum`, de rationnels ou de complexes, reposent en général sur des structures (au sens C du terme) immutables, dont il pourra exister autant d'instances identiques que l'on veut. C'est le contenu de la structure qui détermine l'identité de l'objet, c'est-à-dire sa valeur. On pourrait ainsi définir la classe `Point` (vue d'un point de vue mathématique) comme la classe `Complex`.

Une alternative consiste à ne créer qu'un unique représentant de la classe d'équivalence. Cela nécessite un protocole de construction différent, où l'allocation de l'objet ne se fait que si la valeur n'existe pas déjà. Cela présente sans doute peu d'intérêt pour les nombres, mais c'est une technique couramment utilisée pour les `string`, ce qui en fait des `symbol`. Cela nécessite une table de symboles qui mémorise tous les symboles rencontrés.

8.2.8 Clés dans les bases de données

Cette problématique n'est pas très éloignée de la notion de *clé* dans les bases de données. Adaptée à l'approche objet, une clé est un ensemble d'attibuts qui permet d'identifier de façon unique un objet.

Du coup, la bonne spécification de `equals` et `hashCode` devrait passer par une déclaration explicite des clés. Par exemple un mot-clé `key` désignerait chaque attribut appartenant à la clé. Et le compilateur se chargerait de générer les méthodes `equals` et `hashCode` qu'il n'est pas sérieux de laisser à la charge du programmeur.

EXERCICE 8.2. La génération des méthodes `equals` et `hashCode` par méta-programmation est un bon exercice du Chapitre 13. On remplacera le mot-clé `key` par une *annotation* JAVA. □

8.2.9 Faire une table de symboles

Supposons que l'on veuille faire une application qui manipule des termes et des expressions formées de ces termes, de telle sorte que deux occurrences d'un terme puissent être considérées comme identiques et qu'il soit nécessaire de mémoriser ces termes dans des structures de hachage.

On va donc faire une classe `Term` munie d'un attribut qui référencera la `String` associée au terme, plus d'autres informations spécifiques.

```
class Term {
   thestring: String;
   Term (s: String) { thestring=s }}
```

A partir de là, il y a deux façons de faire.

La mauvaise consistera à considérer q'un langage comme JAVA fournit tout ce qu'il faut, avec des `HashMap`, et des méthodes `equals` et `hashCode`.

Dans ce cadre, le lecteur avisé de ce qui précède va redéfinir `equals` et `hashCode` dans la classe `Term`, de telle sorte que deux termes de même `String` soient `equals` et que le `hashCode` d'un `Term` soit égal à celui de sa `String`.

Le programme pourra alors lire les expressions en entrée et pour chaque terme rencontré construire une instance de `Term` correspondante, et stocker ces instances de `Term` dans la `HashMap`.

Avantages : ça marche !

Inconvénients : la taille des expressions peut-être très grande par rapport au nombre de termes, donc la consommation mémoire, en `Term` et `String`, sera élevée. De plus, les accès aux tables de hachage seront coûteux à tous les niveaux : calcul de `hashCode` et test `equals`, tous les deux basés sur le contenu d'une chaîne de caractères.

La bonne consistera à identifier le problème dès le début et à définir une table de termes et un constructeur de termes (au sens propre du terme!) qui retourne le terme déjà archivé où en construit un nouveau :

```
class Term {
   thestring: String;
   Term (s: String) { thestring=s }
   static private table = new HashMap<String,Term>
   static public newTerm (s: String)
            { table.get(s) || table.set(s,new Term(s) }
   }
```

Avantages : il n'y a plus qu'une instance de chaque terme, et les méthodes `equals` et `hashCode` reposant sur l'égalité physique et l'adresse de l'objet sont les plus efficaces possibles.

Noter que cette façon de faire consiste à encoder les termes de l'applications comme des objets. Avec des langages comme FORTRAN ou C on aurait en général utilisé un encodage par des entiers. La morale de l'histoire est que les langages à objets peuvent bien présenter des fonctionnalités de haut niveau comme les tables de hachage, cela n'empêche pas le programmeur de réfléchir à la bonne façon d'encoder ses données.

8.2.10 En savoir plus

Pour en savoir plus sur l'égalité logique et le hashage dans le contexte de langage comme JAVA ou SCALA : http://www.artima.com/lejava/articles/equality.html.

La question de l'égalité des objets est aussi traitée dans le contexte de CLOS, section 12.2.8.

La question de l'identité des objets a été au cœur de nombreuses réflexions philosophiques, mais aussi physiques. Outre la référence au logicien W.V. Quine en exergue de cette section, Leibniz est ainsi connu pour son principe d'*identité des indiscernables*, qui dit bien ce qu'il veut dire.

La physique quantique propose une intéressante variante de ce principe. Il existe deux grandes catégories de particules élémentaires, les fermions et les bosons, qui diffèrent par leur *spin*, de valeur respectivement fractionnaire ou entière. Les premiers, par exemple les électrons, vérifient le *principe d'exclusion* de Pauli qui veut qu'il n'y ait pas 2 particules dans le même état. En revanche, ce principe ne s'applique pas aux bosons. On voit ainsi que la question des valeurs et des objets se pose aussi dans la nature, et qu'elle n'a pas une réponse univoque.

On rencontre des problèmes similaires avec les classes paramétrées `Comparable` et `OrderedSet` qui devraient pouvoir être paramétrées par la relation d'ordre à utiliser (Chapitre 7).

8.3 Collections immutables

L'immutabilité des objets est une propriété intéressante sur plusieurs plans :

— c'est d'abord une question de protection : pour la logique des programmes, elle garantit que l'état de certains objets ne va pas être altéré par des procédures qui n'en ont pas le droit ;
— c'est ensuite une question d'expressivité du sous-typage : l'immutabilité autorise la covariance ;
— enfin, c'est aussi une question de performance, l'immutabilité permettant des analyses plus fines, une gestion mémoire plus efficace, et plus de parallélisme.

L'immutabilité peut se comprendre de trois façons différentes :

— c'est une propriété essentielle de l'objet (par exemple, les `String` en JAVA) : l'objet ne peut jamais être modifié ;
— c'est une propriété momentanée de l'objet référencé, qui ne peut pas être modifié durant l'activation courante de la méthode ;
— c'est une propriété locale de l'expression qui référence l'objet : la méthode courante ne peut pas lui appliquer de méthodes susceptible de modifier l'objet.

Dans le premier cas, le compilateur et le programmeur peuvent faire des hypothèses d'invariance sur des propriétés de l'objet : par exemple la chaîne de longueur "abc" est de longueur 3, et le restera. Ce cas ne pose pas de problème pratique puisqu'il suppose juste que la bonne classe immutable a été définie (manuellement ou automatiquement à partir de la classe mutable correspondante).

Le deuxième cas est censé apporter les mêmes garanties, mais la concurrence rend la preuve difficile.

Dans le troisième cas, ces hypothèses d'invariance sont hasardeuses, même sans concurrence, puisque l'objet considéré peut être accessible autrement par une méthode appelée localement, et il est nécessaire de recalculer cette longueur à chaque usage.

Enfin, l'immutabilité peut être superficielle (l'objet courant) ou profonde (les objets référencés par l'objet courant), et ce à une profondeur quelconque.

Nous considérerons, ici, l'immutabilité locale et superficielle, vue comme une protection des objets similaire au droits d'accès statique. Une méthode n'a aucun droit de modification sur un objet lorsqu'il est vu, localement, comme immutable.

La question de l'immutabilité se pose pour tous les objets, mais plus particulièrement pour les collections. Il y a en gros deux façons de définir des collections immutables, en faisant des classes ou interfaces dédiées, ou avec le joker de JAVA (voir aussi Section 6.8.2).

8.3.1 Classes ou interfaces dédiées

Le principe consiste à dupliquer la hiérarchie des classes et interfaces de collection pour faire des versions mutables et immutables. En principe, la version mutable devrait être un sous-type de la version immutable, puisque la version immutable n'a d'accès qu'en lecture alors que la version mutable y rajoute les accès en écriture. On aurait donc `Mutable⟨T⟩ <: Immutable⟨T⟩`. Cela suppose bien sûr que les collections immutables n'aient pas de propriété *positive*, par exemple le fait qu'elles soient protégées en écriture.

Par ailleurs, on a une hiérarchie classique de collections, par exemple `HashSet⟨T⟩ <: Set⟨T⟩ <: Collection⟨T⟩`. On fait le produit des deux ordres, ce qui nous donne de l'héritage multiple. Dans un langage comme JAVA et C♯ qui n'ont que du sous-typage multiple, il faut donc que les collections immutables soient des interfaces. Le principe consistera donc à faire des implémentations mutables (par exemple `HashSet`) et à avoir des interfaces immutables. Lorsqu'une classe veut rendre immutable une collection, elle la crée mutable et la retourne sous un type immutable. Par exemple :

```
public getImmutableCollection() : ImmutableSet<T> {
            hs =  new HashSet<T>;
            ...
            return hs; }
```

On pourra de plus utiliser les annotations de type, en C♯ ou SCALA par exemple, pour rendre les collections immutables covariantes. Le produit d'ordre passe alors à une troisième dimension, puisqu'il faut alors y rajouter le sous-typage covariant `Immutable⟨Assiette⟩ <: Immutable⟨Vaisselle⟩ <: Immutable⟨Object⟩`.

Cette approche par définition d'une hiérarchie de collections immutables est en gros celle suivie par SCALA.

8.3.2 Avec *joker*

Le *joker* évitera complètement le besoin de définir des collections immutables, simplifiant d'autant la hiérarchie de classes et interfaces.

```
public getImmutableCollection() : Set<? extends T> {
            hs =  new HashSet<T>;
            ...
            return hs; }
```

Ici, la covariance vient naturellement de l'usage du *joker*.

Dans les deux cas, il faut noter que l'immutabilité n'est qu'un point de vue sur un objet intrinsèquement mutable. L'inconvénient de cette approche par rapport à des collections immutables qui seraient des classes à part entière, mais en héritage multiple, est qu'un *cast* maladroit peut dévoiler le caractère effectivement mutable de l'objet considéré. Noter que le cast n'a pas besoin d'être explicite : un test de sous-typage dû à la covariance suffirait à rendre mutable un objet typé comme immutable. Bref, la protection n'est pas sûre, même si le programmeur n'y met pas de malignité. Sans parler du fait que le joker n'exclut que les méthodes utilisant le paramètre formel de type en position contravariante. Par exemple, `add(T)`. Mais le programmeur pourra vider la collection, avec `clear()`, en toute impunité.

Mais on ferait difficilement mieux sans réel héritage multiple.

8.3.3 Approche par annotation de type

Une troisième approche consiste à avoir une annotation de type spécifique, par exemple `immutable`. Pour n'importe quel type `T`, on peut utiliser le type `immutable T` qui repose sur les deux règles de sous-typage :

```
T <: immutable T
immutable T = immutable immutable T
```

Par ailleurs, pour que l'immutabilité soit garantie, il doit être impossible de caster un `immutable T` vers un `T`.

Par principe, l'interface de `immutable T` ne contient que les méthodes de `T` qui ne modifient pas le receveur courant, directement ou indirectement. Pour cela, il y a trois solutions :

— à la C++, où le mot-clé `const` joue le rôle de `immutable` : dans ce cas, chaque méthode de l'interface de `const T` est elle-même étiquetée par `const`, et une méthode `const` ne peut pas modifier son receveur ou appeler une méthode non-`const` ; deux gros défauts à cette spécification :
  — il y a surcharge statique entre les méthodes `foo()` et `foo() const` ;
  — `const_cast` permet de décoller l'étiquette d'immutabilité en castant un `immutable T` en un `T` ;

— en OCaml, le mot-clé `immutable` a en gros les spécifications que nous lui donnons ici, mais il n'y a pas d'annotation particulière des méthodes : si `foo()` de la classe `A` ne modifie pas le receveur courant, il appartient à l'interface de `immutable A` ; si dans une sous-classe `B` *spec* `A`, `foo()` est redéfinie et modifie le receveur courant, alors `foo()` ne fait pas partie de l'interface de `immutable B` et `immutable B` n'est pas un sous-type de `immutable A` ;

— une alternative consisterait à définir une annotation optionnelle, par exemple `mutate`, pour les méthodes (en tant que propriétés globales, dès leur introduction) indiquant que la méthode peut modifier le receveur courant : l'interface de `immutable T` ne contient que les méthodes non-`mutate` de `T` ; `mutate` est optionnel car il peut être déduit par le compilateur du fait que la méthode, à son introduction, modifie le receveur courant ; en revanche, il est obligatoire, lorsque la méthode définie lors de l'introduction ne le modifie pas mais qu'il est prévu qu'une de ses redéfinitions le fasse : il est interdit de redéfinir une méthode non-`mutate` par une méthode `mutate`.

8.3.4 Collections quasi-immutables covariantes

On pourra utiliser ces patrons pour définir des collections dites *quasi-immutables*, qui sont publiquement immutables mais privativement mutables. C'est le cas des collections utilisées pour implémenter les associations Uml. On en trouvera un autre exemple avec les classes qui mémorisent leurs instances, Section 14.2.2, page 217.

On suppose que l'on a des classes A, B, C, etc., avec $A \prec B \prec C$..., ainsi que des classes T, U, V, etc., avec $T \prec U \prec V$.... Les deux hiérarchies peuvent être confondues.

A chaque classe A, B, C, .. est associée une collection `Set⟨T⟩`, `Set⟨U⟩`, `Set⟨V⟩`, etc., par exemple par le biais de variables statiques. Dans chaque classe A, B, C, .., on définit alors une méthode (non statique) d'accès à ces collections, nommée par exemple `getSet` et typée par `Set⟨? <: T⟩`, dans A, `Set⟨? <: U⟩`, dans B, etc. Cela constitue le principe du « patron :`allocation` :`class` » (page 210).

Enfin, dans A, et uniquement dans A, on définit la méthode `addSet` qui va ajouter un élément à la collection retournée par `this.getSet()`.

Dans un contexte où `y` est une instance de T, le compilateur va refuser l'expression `this.getSet().add(y)`. En effet, le *joker* interdit toute écriture car `this.getSet()` retourne un `Set` de n'importe quel sous-type de T.

Grâce à l'effacement de type, si le programmeur est sûr que `y` est une instance de T (resp. U, V) lorsque `this` est une instance de A (resp. B, C), on peut contourner le problème avec un *cast* que le compilateur ne peut pas vérifier et qu'il va donc accepter : `(Set<A>)(this.getSet()).add(y)`

Il est important de comprendre que cela marche en Java grâce à la conjonction de deux traits de langages :

— le *joker* permet de désigner des instances génériques de façon assez générale ;

— l'effacement de type oblige le compilateur à accepter des *cast* qu'il ne peut pas vérifier.

Sans joker. En C♯, aucun de ces traits de langage n'existe. Cependant, l'extension du langage par ajout de *joker* ne permettrait toujours pas cette solution, car C♯ peut implémenter le *cast*, et il échouerait toujours.

En revanche, la version 4 de C♯ permet d'annoter les types formels suivant leur caractère in- co- ou contra-variant. On pourrait ainsi définir `ImmutableSet⟨+T⟩`, la classe des ensembles immutables, qui n'implémente que les accès en lecture et dont le paramètre est covariant : `ImmutableSet⟨U⟩ <: ImmutableSet⟨T⟩` si `U <: T`. Le type Java `Set⟨? <: U⟩` est en réalité strictement équivalent au type C♯ `ImmutableSet⟨U⟩`, où l'interface de `ImmutableSet` est constituée exactement des méthodes de `Set` qui ne prennent pas le type formel en paramètre. La méthode `getSet` aurait alors un type de retour `ImmutableSet⟨U⟩` au lieu de `Set⟨? <: U⟩`.

Dans la méthode `addSet`, un cast de `ImmutableSet` vers `Set` est alors nécessaire, mais il ne peut pas se faire dans une unique méthode définie dans A, parce que le paramètre T courant n'est qu'un super-type de celui de l'ensemble réel, U par exemple. La solution serait alors de définir `addSet` dans

chaque sous-classe de A, en faisant le *cast* sur le paramètre de `addSet` et en accédant directement à l'attribut statique : `B.set.add((U)y)`.

Cela fait néanmoins beaucoup de choses à redéfinir dans chaque sous-classe : la variable statique `set` et ses deux accesseurs `addSet` et `getSet`.

8.4 Application des associations Uml : les graphes d'objets

On peut appliquer les principes généraux de l'implémentation des associations Uml à un exemple typique : les graphes d'objets, qui sont des `Graphes` composés de `Sommets` et `Arêtes`. Ces graphes peuvent être spécialisés dans différents domaines, par exemple des `Molécules` en chimie et des `Réseaux` en informatique.

L'objectif est de définir des classes qui interdiront, par construction, l'existence de *chimères*, c'est-à-dire d'objets hybrides qui mélangeraient des caractéristiques issues de différents domaines d'application (chimie et télécommunications par exemple).

Cette application constitue le projet associé au module *Langages à Objets à Typage Statique*. L'objectif du projet est d'implémenter les graphes d'objets dans différents langages et avec différentes techniques, pour mettre en valeur les problèmes de typage statique.

8.4.1 Spécification des graphes d'objets

Ce sont des `Graphes` composés de `Sommets` et `Arêtes`, les 3 constituant des classes à parité. Ces graphes peuvent être spécialisés dans différents domaines, par exemple des `Molécules`, composées d'`Atomes` et `Liaisons`, en chimie et des `Réseaux`, composés de `Noeuds` et de `Liens`, en informatique.

L'objectif est de définir des classes qui interdiront, par construction, l'existence de *chimères*, c'est-à-dire d'objets hybrides qui mélangeraient des caractéristiques issues de différents domaines d'application (chimie et télécommunications par exemple). Il s'agit donc de mettre en valeur les erreurs de types qui peuvent survenir, soit à la compilation, soit à l'exécution, pour montrer que le code protège efficacement contre ces erreurs.

Spécifications détaillées des associations A ces spécifications générales s'ajoutent le fait que les spécifications doivent être implémentées dnas les règles de l'art :

- elle doivent être navigables dans les deux sens, chaque objet connaissant les instances des deux autres classes avec lesquelles il est lié ;
- les associations ne sont pas des agrégations : on peut par exemple enlever un sommet d'un graphe pour le mettre dans un autre ;
- pour simplifier et pour traiter le cas où le rôle d'une association est mono-valué, on considérera qu'un sommet ou arête n'appartient qu'à un seul graphe ;
- du fait du caractère bidirectionnel des associations, la mise à jour doit vérifier des contraintes de symétrie : si l'objet `a` référence l'objet `b`, alors l'inverse doit être vrai aussi ; en particulier, par contrapposition, si l'objet `a` ne référence plus l'objet `b`, alors `b` ne doit plus référencer `a` ; toute mise à jour est donc double !
- lorsque les techniques d'implémentation permettent des erreurs de type à l'exécution, il faut veiller à ce que l'erreur de type de survienne pas alors que la mise à jour est faite à moitié.

Invariants L'objectif est de faire en sorte de disposer d'instance de graphes valides, où toutes les contraintes seront respectées, et que toute mise à jour (ajout ou retrait d'élément) d'un graphe valide aboutisse à un graphe valide.

On définira dans chaque classe `Graphe`, `Sommet` et `Arete` une méthode `invariant` qu'on testera systématiquement lors des mises à jour d'un graphe.

- l'invariant d'une arete est que son graphe doit être identique à celui de ses deux sommets extrémités :

```
arete.getGraphe() == arete.getSommet1().getGraphe() &&
      arete.getGraphe() == arete.getSommet2().getGraphe()
```

- l'invariant d'un sommet est que toutes ses arêtes incidentes respectent leur invariant ;
- l'invariant d'un graphe est que tous ses sommets et arêtes respectent leur invariant et que leur graphe soit le graphe lui-même.

En Eiffel, on utilisera bien entendu les mots-clés existant pour les assertions, `require`, `ensure` et `invariant`.

Constructeurs Il y a en gros deux façons de construire les éléments d'un graphe, sommets ou arêtes. On peut faire des créations contextuelles, avec les signatures :

```
Sommet(Graphe)
Arete(Sommet,Sommet)
```

ou bien n'utiliser que des constructeurs hors contexte, qui créent des éléments « sur étagère », avant de les assembler dans un graphe. On poura faire les deux, mais dans le premier cas, on s'assurera que l'on peut déplacer des éléments d'un graphe dans un autre.

8.4.2 Techniques à implémenter

Il faut utiliser toutes les techniques vues en cours, c'est-à-dire :

covariance pure : seul le langage Eiffel la permet ;

simulation de la covariance : l'alternative pour tous les autres langages, en se basant sur la surcharge statique pour les langages qui le permettent (Java, C++, C♯, Scala, ..) ;

généricité F-bornée : elle peut être utilisée dans tous les langages considérés, sauf Prm/Nit ; en Eiffel, la covariance s'applique aussi aux instances génériques, donc son utilisation peut être un peu différente ;

types virtuels : c'est la solution la plus simple et élégante, mais peu de langages la proposent, Prm/Nit, Eiffel (types ancrés) et Scala ; notez que la spécification des types vituels en Scala est plus stricte que dans les deux autres langages ;

patron Stratégie on pourra aussi considérer l'alternative du patron Stratégie (Section 7.7 au schéma symétrique de la généricité F-bornée ou des types virtuels.

8.4.3 Langages considérés

Sont à considérer :
— les langages à objets *mainstream* comme Java, C++ et C♯ ;
— les langages plus exotiques, qu'ils soient en perte de vitesse comme Eiffel, en pleine ascension comme Scala, ou confidentiels comme Prm/Nit ; on pourra aussi regarder Fortress ;
— des langages moins directement ciblés par le cours comme Ada 2005 ou OCaml : pour ces derniers langages, il faudra voir ce qui est ou n'est pas faisable mais, dans les deux cas, ce sera réservé à des étudiants qui ont déja fait de l'Ada ou du Caml.

8.4.4 Cahier des charges

Chaque étudiant doit respecter les deux contraintes suivantes :
— implémenter les 3 techniques principales, par simulation de la covariance, généricité F-bornée et types virtuels ;
— le faire en Java, C++, plus un ou deux (ou plus) autres langages.

Noter que le choix de C♯ comme troisième et dernier langage ne permet pas de respecter la première contrainte : prenez le en quatrième langage.

Attention, il ne s'agit pas de faire une implémentation complète mais uniquement le squelette : il faut juste le petit nombre de méthodes nécessaire à la mise à jour des 3 associations, pour construire et décconstruire des graphes, et ce qu'il faut pour afficher les objets de façon satisfaisante.

Le projet est personnel, même si vous pouvez, bien sûr, travailler à plusieurs. Le rendu du projet se fera par démonstration individuelle en salle de TP la semaine après les examens de première session. Inutile de rédiger un rapport.

8.5 Autres problèmes de typage

La section 13.1 décrit les problèmes de typage statique, en particulier de généricité, provoqués par l'introspection et la réflexivité dans le contexte des langages Java et C♯.

9

Implémentation

Si l'implémentation ne peut pas et ne doit pas servir à spécifier un langage, elle peut néanmoins aider à le comprendre, surtout quand les spécifications sont nébuleuses.

9.1 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 9.1). 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 deux invariants suivants.

Invariant 9.1 (Référence) *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 9.2 (Position) *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 τ_s et τ_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 [1], 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 9.1 résume cette implémentation.

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

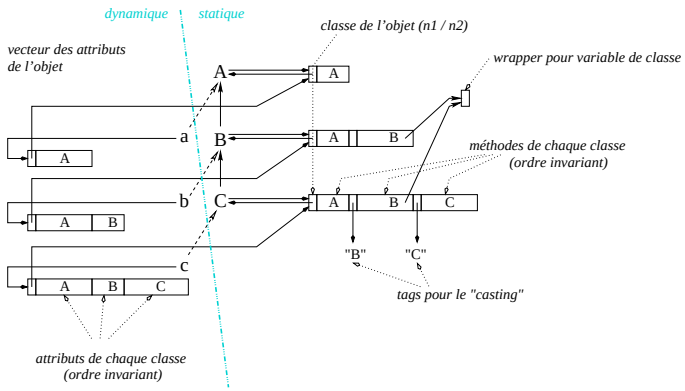


FIGURE 9.1 – Structure des objets et des tables de méthodes en héritage simple : 3 classes A, B et C et leurs instances respectives, a, b et c.

9.2 En héritage multiple

Avec l'héritage multiple, il devient nécessaire de préciser dans quel contexte pratique on se place. L'implémentation en héritage simple est compatible avec la compilation séparée et l'*hypothèse du monde ouvert* : la connaissance des futures sous-classes est inutile. L'héritage multiple pose un problème majeur si l'on souhaite conserver cette hypothèse.

9.2.1 Le problème de l'héritage multiple

En héritage multiple, le problème se complique considérablement comme le montre [Ellis and Stroustrup, 1990, 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 [Meyer, 1992, 1997].

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 9.2), 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 indirection dans une table, un pointeur sur un objet n'est plus invariant suivant son type statique (invariant 9.1).

9.2.2 Principe d'implémentation

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

Invariant 9.3 *Chaque attribut a un indice non ambigu et invariant dans le contexte du type statique qui l'introduit, donc indépendamment du type dynamique.*

Invariant 9.4 *Chaque méthode a un indice non ambigu et invariant dans le contexte d'un type statique qui connait 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 9.2) :

Invariant 9.5 *Toute entité de type statique T est liée au sous-objet correspondant à T, muni de sa propre table de méthodes.*

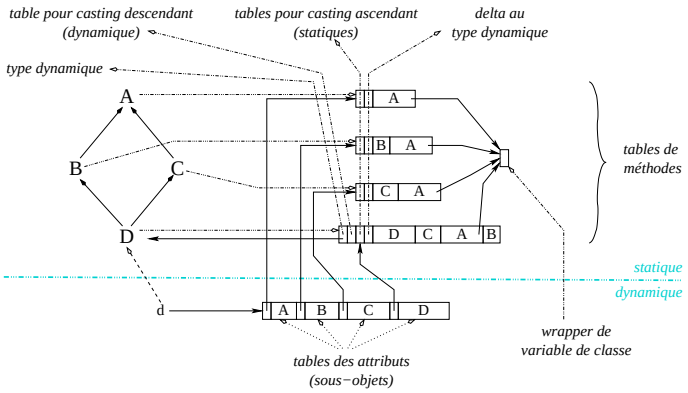


FIGURE 9.2 – Tables des attributs et des méthodes en héritage multiple : contrairement à la figure 9.1, une seule instance est décrite.

Cet invariant est trivialement vérifié par l'implémentation de l'héritage simple (invariant 9.1). En héritage multiple, il est nécessaire de rajouter une propriété de non trivialité, qui est la source du surcoût de cette implémentation :

Invariant 9.6 *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 9.3).

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 9.5 impose de recalculer la valeur de `self` à chaque envoi de message : lorsque le receveur est une entité de type statique τ_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 τ_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 [3] :

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

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 [Ellis and Stroustrup, 1990]). 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
```

2. Précisons que la notation $\Delta_{t,u}$, ainsi que toutes les notations Δ qui vont suivre, sous-entend un type dynamique τ_d donné, dont l'explicitation alourdirait trop la notation.

3. En italiques, les instructions supplémentaires par rapport à l'héritage simple.

type statique → / ↓ dynamique	A	B	C	D
A	| A	—	—	—
B	| A	| A | B	—	—
C	| A	—	| A | C	—
D	| A	| A | B	| A | C	| A | B | C | D

FIGURE 9.3 – Tables des méthodes pour l'exemple de la figure 9.2, 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.

On économise ainsi une indirection dans une table, au prix d'un saut constant [4].

9.2.3 L'héritage multiple non virtuel de C++

L'implémentation de C++ telle qu'elle est décrite dans [Ellis and Stroustrup, 1990, Lippman, 1996] 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 [5] :

— 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 [6], 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 9.7 *Le sous-objet associé à τ_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 τ_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 9.2, lorsque l'héritage de la classe A par B 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* [7]. Dans

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

5. 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.

6. 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.

7. [Meyer, 1997] a un usage un peu différent du terme d'héritage répété, qui désigne la figure d'héritage en losange (Figures 9.2 et 9.4) : 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`

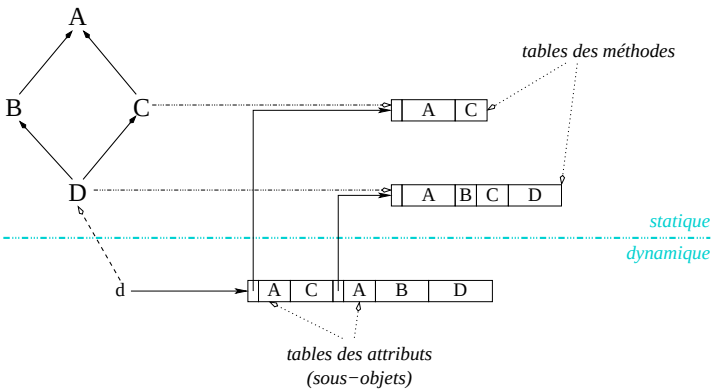


FIGURE 9.4 – Tables des attributs et des méthodes en héritage multiple « non virtuel »

type statique → ↓ dynamique	A	B	C	D
A	| A |	—	—	—
B	B	| A | B |	—	—
C	C	—	| A | C |	—
D	C/D	D	| A | C |	| A | B | C | D |

FIGURE 9.5 – 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.

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 9.4). Si l'héritage est arborescent, le nombre total de tables est égal au nombre de super-classes sans super-classes. Mais dans le cas général, le nombre de tables est égal au nombre de chemins de la classe considérées vars les racines : dans le pire des cas, ce nombre peut-être exponentiel.

EXERCICE 9.1. Donner un exemple de hiérarchie de classes impliquant un nombre exponentiel de tables. □

9.2.4 La combinaison de méthodes

Dans la *combinaison de méthodes* (voir Section 6.7), les appels à `super` ou `Precursor` sont des appels statiques sans problèmes. En revanche, `call-next-method` est un appel dynamique qui dépend du type dynamique du receveur. Il est donc nécessaire de passer par la table des méthodes. Pour une méthode `foo` définie dans la classe C et qui appelle `call-next-method`, il faut introduire dans A une méthode `cnm_foo_A` dont l'adresse est la méthode suivante.

Dans une implémentation par sous-objets, cette méthode n'est nécessaire que dans la table des méthodes du sous-objet correspondant au type statique A.

EXERCICE 9.2. Vérifier par *introspection* la façon dont `super` est implémenté en SCALA où il fonctionne comme `call-next-method`. □

9.3 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.

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.

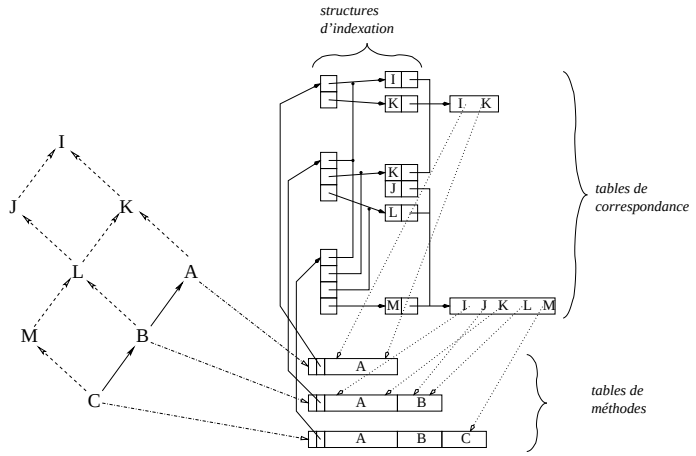


FIGURE 9.6 – Sous-typage multiple et héritage simple : variante du sous-typage simple, avec tables de conversion (variante 1).

La spécification du problème est très exactement celle de JAVA [Arnold and Gosling, 1997, Grand, 1997] : 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 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'implémente. L'invariant 9.2 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.

9.3.1 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. C'est le choix effectué par la spécification des machines virtuelles JAVA, où 2 opérations coexistent :

— `invokevirtual` est utilisé lorsque le receveur est typé par une classe, et il est implémenté comme en sous-typage simple ;

— `invokeinterface` est utilisé lorsque le receveur est typé par une interface, et il est implémenté de façon *ad hoc*.

On conserve alors de l'héritage simple les deux invariants 9.1 et 9.2, 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 9.6) ;
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.

9.3.2 Application aux *mixins*

On a vu en Section 4.6 comment le modèle du sous-typage multiple permettait d'implémenter un modèle d'héritage multiple asymétrique comme les *mixins* du langage Scala.

9.4 Compilation globale

L'analyse qui précède repose sur l'hypothèse du monde ouvert. Certains systèmes font l'*hypothèse* inverse *du monde clos* et pratiquent une compilation globale. C'est le cas en particulier des implémentations d'Eiffel comme Smart Eiffel ou Eiffel Studio. Les implémentations en sont notablement différentes de ce qu'elles sont en compilation séparée (voir par exemple [Ducournau, 2011b]).

Une autre alternative consiste à reposer sur l'hypothèse du monde ouvert à la compilation, puis sur l'hypothèse du monde clos à l'édition de liens. C'est la solution adoptée par les concepteurs du langage Prm. Elle est rapidement décrite dans le chapitre suivant.

9.5 Limites de l'indépendance des spécifications vis-à-vis de l'implémentation

En théorie, les langages de programmation sont spécifiés indépendamment de toute implémentation. Cependant, il est naturel d'avoir une idée de l'implémentation pour bien spécifier une fonctionnalité, car une spécification non implémentable n'est guère recevable.

On est cependant ici dans le domaine des grands principes et, en pratique, les langages sont souvent indissociables de leur implémentation. Un sous-ensemble des langages est toujours indépendant des implémentations, mais il y a souvent une fonctionnalité qui ne l'est pas. Trois exemples :

— en Eiffel, la covariance non contrainte des types de paramètres réclame des vérifications systématiques en compilation séparée : n'importe quel paramètre peut être redéfini, à moins que son type soit *final*, parce exemple si c'est un type primitif, ou s'il a été déclaré `frozen`. La compilation séparée d'Eiffel rendrait donc le langage à la fois peu sûr et inefficace. La règle des *catcall*s aggrave le problème puisqu'elle n'est vérifiable qu'en compilation globale.

— en C++, la plus grande partie de la spécification du langage est basée presqu'explicitement sur l'implémentation par sous-objets, en particulier l'héritage non-virtuel et sa combinaison avec l'héritage virtuel.

— Java et C♯ sont indissociables de leur machine virtuelle et du chargement dynamique.

9.6 En savoir plus

[Ducournau, 2011b] propose une synthèse sur les différentes implémentations possibles des mécanismes propres aux langages à objets. Plusieurs techniques ont été proposées pour répondre aux problèmes posés conjointement par l'héritage multiple et le chargement dynamique, en particulier la *coloration* (dont [Ducournau, 2011a] présente une synthèse) et le *hachage parfait* qui s'applique bien à l'implémentation des interfaces de Java [Ducournau, 2008]. [Ducournau et al., 2009] présente par ailleurs des résultats expérimentaux sur ces différentes implémentations.

Nous n'avons considéré dans ce chapitre que le cadre du typage statique. Les langages à typage dynamique comme Smalltalk et Clos posent des problèmes encore différents, même en héritage simple.

Bibliographie

K. Arnold and J. Gosling. *The JAVA programming language, Second edition.* Addison-Wesley, 1997.

R. Ducournau. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 30(6) :1–56, 2008. doi : 10.1145/1391956.1391960.

R. Ducournau. Coloring, a versatile technique for implementing object-oriented languages. *Softw. Pract. Exper.*, 41(6) :627–659, 2011a. doi : 10.1002/spe.1022.

R. Ducournau. Implementing statically typed object-oriented programming languages. *ACM Comp. Surv.*, 43(4), 2011b. doi : 10.1145/1922649.1922655.

R. Ducournau, F. Morandat, and J. Privat. Empirical assessment of object-oriented implementations with multiple inheritance and static typing. In Gary T. Leavens, editor, *Proc. OOPSLA'09*, SIGPLAN Not. 44(10), pages 41–60. ACM, 2009. doi : 10.1145/1639949.1640093.

M.A. Ellis and B. Stroustrup. *The annotated C++ reference manual.* Addison-Wesley, Reading, MA, US, 1990.

M. Grand. *JAVA Language Reference.* O'Reilly, 1997.

S. B. Lippman. *Inside the C++ Object Model.* Addison-Wesley, New York, 1996.

B. Meyer. *Eiffel : The Language.* Prentice-Hall, 1992.

B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, second edition, 1997.

10

Quelques langages

Les langages C++ et Java sont universellement utilisés et considérés ici comme connus. Le langage C♯ (et ses avatars .Net) est plus récent et ne tardera pas à être aussi connu que Java, dont il diffère apparemment assez peu — en tout cas, au niveau d'abstraction auquel on les considère ici. Cependant les détails qui les différencient sont révélateurs des degrés de liberté que s'offrent, à tort ou à raison, les concepteurs de langages.

En revanche, le langage Eiffel est moins connu, en tout cas des générations actuelles, et Prm reste purement confidentiel. Le premier est pourtant un grand classique, mais il n'a pas réussi à se transformer en langage de production. Aussi est-il de moins en moins utilisé et son enseignement se réduit à quelques poches de résistance. Le second quant à lui est nouveau mais ne prétend pas devenir un langage de production : c'est un langage de recherche destiné à tester des constructions de langage et des implémentations.

Finalement, Scala est un langage émergent qui devrait durer.

Nous ne décrirons de ces langages que les traits saillants, renvoyant pour le reste à la documentation existante.

10.1 C♯ vs Java

On se contente ici de lister les différences répertoriées au long des différents chapitres, entre C♯ et Java :

objets vs C alors que Java a abandonné toute compatibilité avec C, C♯ maintient une coexistence avec de nombreux traits non-objet de C++ : cohabitation des classes et des `struct`, mot-clé `virtual`, etc.

généricité alors que Java pratique l'effacement de type dans une implémentation homogène, C♯ conserve des types exacts à l'exécution, avec une implémentation mixte homogène/hétérogène, fonctionnellement très proche de C++ (avec quelques différences mineures quand même) (cf. Section 7.8.1). En revanche, C♯ n'a pas repris la notion de *joker*.

héritage multiple alors que Java ne voit pas les conflits de propriétés globales pour les méthodes qui sont introduites dans plusieurs interfaces implémentées par une même classe, C♯ a identifié le problème et propose une construction syntaxique partielle pour y remédier (cf. Section 4.3.4).

covariance C♯ reprend la règle de covariance inutilement stricte de Java 1.4, avec un type de retour invariant (cf. Section 5.3).

types primitifs : ils sont un peu mieux intégrés en C♯, mais c'est surtout grâce à la généricité (cf. Section 8.1).

surcharge statique Encore une subtile différence entre C♯ et aussi bien Java 1.4 que Java 1.5 : le cas d'ambiguïté de Java 1.5 qui n'existe pas en Java 1.4 est résolu différemment en C♯ (cf. Section 6.3.2).

redéfinition et surcharge C♯ utilise trois mots-clés pour la définition de méthodes : `virtual` est implicite et obligatoire en Java, `override` qui est implicite en Java et `new` dont l'effet n'est pas possible en Java.

10.2 Le langage Scala

On peut caractériser Scala par les différents éléments suivants :

— malgré une syntaxe très différente, on peut le voir comme une extension de JAVA du fait qu'il compile vers le *bytecode* JAVA (JVM) et qu'il peut ainsi réutiliser toutes les bibliothèques JAVA.
— il complète JAVA par différents aspects comme un quasi-héritage multiple par *mixin* — appelés *traits* dans le langage — (cf. Section 4.6).
— une intégration de différents aspects fonctionnels dans le langage, comme des fermetures.

10.3 Le langage EIFFEL

10.3.1 Traits distinctifs

Les caractéristiques principales d'EIFFEL, qui le différencient de la plupart des autres langages, sont les suivantes :

typage statique à la sûreté équivoque, sans *surcharge statique* ;

généricité F-bornée (Chapitre 7) ;

covariance à peu près généralisée, sur les types de retour (bien sûr !), les types de paramètres, voire les droits d'accès statiques et les contrats ; à cette covariance généralisée, s'ajoute la notion de *type ancré* avec le mot-clé `like`, qui constitue un premier pas vers les *types virtuels* (dans les dernières versions d'EIFFEL, `like` n'est plus obligatoire et la syntaxe devient très proche des types virtuels, mais sans type final) ; la règle des *catcall*s est censée rendre le typage sûr (au sens strict) mais elle n'est pas appliquée (Chapitre 5) ;

héritage multiple qui se rapproche le plus du méta-modèle tout en permettant des aberrations relevant de l'héritage répété de C++ (Chapitre 4) ;

assertions de type *précondition*, *postcondition* et *invariant* permettant de définir des *contrat*s, avec les mots-clés `require`, `ensure` et `invariant` (Chapitre 5) ;

droits d'accès statiques à grain fin, au niveau de la classe ou au niveau du receveur courant (Chapitre 6) ;

constructeurs héritables, mais leur rôle de constructeur ne l'est pas ;

méthodes `once` pour remplacer les variables statiques (Chapitre 14) ;

syntaxe très éloignée de celle de C++ et JAVA, assez verbeuse ;

vocabulaire très idiosyncrasique en version anglaise (voir *feature*, *deferred*, etc. dans le glossaire, Chapitre 16) ; la traduction française aggrave souvent les choses.

La discussion de certains points a été faite dans les différents chapitres de ce cours. Pour le reste, le lecteur est renvoyé à la littérature sur EIFFEL (voir ci-dessous).

Quelques clones d'EIFFEL ont vu le jour mais ont aujourd'hui disparu, comme SATHER [Omohundro and Stoutamire, 1995, Szyperski et al., 1994]. Deux implémentations d'EIFFEL sont disponibles, EIFFEL STUDIO et SMART EIFFEL. Pour des raisons qui apparaissent vite évidentes, la compilation du langage est globale : en effet, la covariance non restreinte rendrait le langage partout non sûr et inefficace (Chapitre 9). Cependant, en mode développement, EIFFEL STUDIO propose un mélange (dit *melting ice*) d'interprétation et de compilation qui évite de tout recompiler lors des ajouts ou modifications.

10.3.2 En savoir plus

EIFFEL est avec C++ le pionnier des langages à objets en typage statique avec héritage multiple. Malgré des avancées conceptuelles indéniables, le langage a pourtant perdu la bataille et ne survit qu'avec difficultés. D'un point de vue bibliographique, on se référera à [Meyer, 2001] comme première introduction, à [Meyer, 1997] pour une méthodologie et à [Meyer, 1994] (mis à jour sur `http://se.ethz.ch/~meyer/#Progress`) pour le manuel de référence. [Gautier et al., 1996] est un cours de programmation par objets, prenant comme support les langages C++ et EIFFEL et les comparant.

10.4 Les langages PRM et NIT

Le langage PRM a été développé au LIRMM par Jean Privat dans le cadre de sa thèse [Privat, 2006b]. Une version plus opérationnelle en a été dérivée par Jean Privat à l'UQAM sous le nom de NIT (`http://nitlanguage.org`).

Le langage a pour double objectif de servir de plate-forme pour tester
— des spécifications alternatives de différents mécanismes relatifs au typage, à l'héritage ou à la modularité ;

— des techniques d'implémentation et de compilation originales.

EIFFEL a été une première inspiration pour certains traits, la covariance par exemple, mais la version courante s'en éloigne maintenant beaucoup.

10.4.1 Spécifications du langage

PRM signifie *Programmer avec des Modules et du Raffinement*, ce qui sous-entend *Programmer objet avec des Modules et du Raffinement de classes*. Les traits originaux du langage concernent l'héritage multiple, le typage statique et les modules et le raffinement.

Héritage multiple PRM implémente directement le méta-modèle du chapitre 3 et traite les conflits de propriétés globales comme dans le chapitre 4. La résolution des conflits de propriétés locales et la combinaison de méthodes seront à terme basées sur des linéarisations, probablement C3 (Subsection 4.5.3).

Typage statique Le typage est statique et sûr, au moins au sens de Cardelli [2004]. La philosophie est basée sur la covariance mais au moyen des types virtuels (Subsection 7.9) : les types de paramètres sont donc syntaxiquement invariants. Contrairement à [Torgersen, 1998], des erreurs de type sont acceptées dans le cas où le type virtuel du type statique du receveur est simplement borné. Mais, contrairement à EIFFEL STUDIO, un test dynamique de type est généré par le compilateur et le programmeur reçoit un avertissement.

Bien que les types virtuels soient souvent considérés comme une alternative à la généricité, le langage propose aussi la généricité bornée et, dans un avenir proche, F-bornée. L'idée est de réserver les types virtuels à la modélisation de la redéfinition covariante, par exemple pour des associations UML, alors que la généricité serait utilisée pour les types génériques les plus simples comme les collections. Bien entendu, la généricité assure un typage parfaitement sûr, contrairement aux types virtuels.

Modules et raffinement de classes PRM propose enfin une double notion de modules et de raffinement de classes, qui lui donne son nom, et qui est basée sur un métamodèle isomorphe à celui des classes et propriétés (Chapitre 3) et présenté dans [Ducournau et al., 2007].

10.4.2 Modèle de compilation

Comme l'a montré le chapitre 9, l'implémentation des mécanismes objet est difficile en héritage multiple et sous l'*hypothèse du monde ouvert*. La compilation séparée est souhaitable puisque c'est elle qui correspond le mieux au Méta-axiome 2.2, page 21, et que c'est ce qui permet de vérifier une unité de code avant de s'en servir. Cependant, l'*hypothèse du monde clos* permet de nombreuses optimisations qui rendent le programme résultant beaucoup plus efficace : on l'obtient en général par la compilation globale, comme dans SMART EIFFEL ou EIFFEL STUDIO.

Aussi, le schéma de compilation de PRM a été conçu pour essayer de concilier ces deux extrémités, en proposant une compilation séparée, sous l'hypothèse du monde ouvert, avec des optimisations globales qui sont effectués à l'édition de liens (Figure 10.1). Les optimisations sont de deux catégories :

— une analyse de types permet de déterminer les *types concrets*[1] de chaque expression : on peut en déduire 2 informations précieuses, les appels monomorphes, qui peuvent être traités comme des appels statiques, et le code mort, qui peut être éliminé ;

— l'*hypothèse du monde clos* permet d'optimiser l'implémentation des objets, par exemple avec la technique de coloration [Ducournau, 2011].

L'unité de compilation est le module. Le compilateur génère du code C, qui est compilé séparément. Une étape globale est appliquée avant l'édition de liens (`ld`) pour construire le schéma global vivant du programme et lier les différents fichiers de code issus de la compilation C.

Erreurs de type Ce schéma de compilation présente l'avantage d'offrir une réponse graduée à la sûreté du typage :

— à la compilation, les règles de typage garantissent un typage sûr, à l'exception du cas où le type du paramètre est virtuel et non final ($T <: t$) ;

1. Le *type concret* (*concrete type*) désigne ici l'ensemble des types dynamiques qu'une expression peut prendre lors de toutes les exécutions possibles d'un programme. Il n'y a pas de rapport avec l'utilisation de ce terme dans le cadre de la généricité (par opposition à type formel) ou en tant que classe, par opposition à classe abstraite.

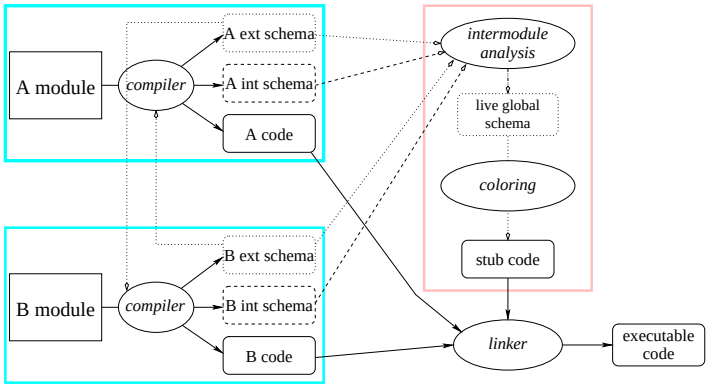


FIGURE 10.1 – The PRM compilation scheme, with local modular separate compilation phases (left) and a final global linking phase (right)

— à l'édition de liens, pour chaque appel de méthode non sûr, l'analyse de types permet de vérifier s'il y a un risque d'erreur à l'exécution ; un test dynamique est inséré dans ce cas, ce qui rend le typage sûr au sens de Cardelli [2004].

10.4.3 Perspectives

Le travail actuel sur PRM consiste à consolider le compilateur dit *autogène* (ou *bootstrapé*), écrit en PRM— voir l'exergue de Spinoza et la notion de *bootstrap* au Chapitre 11.

Les étapes suivantes considéreront la sélection multiple et un méta niveau d'introspection, et plus si affinités. Un schéma d'implémentation de la généricité, intermédiaire entre les schémas homogène et hétérogène, est aussi planifié.

Un *plug-in* Eclipse est en cours de développement.

10.4.4 Exercices

La plupart des exercices des chapitres 4 à 7 peuvent être adaptés à PRM.

10.4.5 En savoir plus

Une présentation générale du langage PRM est donnée dans [Privat, 2006a] et toutes les informations complémentaires sont disponibles sur le site `http://www.lirmm.fr/prm`. Le compilateur PRM a été utilisé pour expérimenter diférentes techniques d'implémentation et les tester dans un environnement de *méta-programmation* [Ducournau et al., 2009].

PRM a été développé par des étudiants de l'Université Montpellier 2 issus de l'IUP3 et du DEA, ou du Master. Ce développement a eu lieu au cours de leur DEA et thèse (Jean Privat, 2002-2006), stage de M2R et thèse en cours (Floréal Morandat 2006-), stage d'été de M1 (Simon Allier et Guillaume Artignan, 2007), stage de M2R (Simon Allier et Yang Mengxi, 2008). Aux suivants ! PRM est utilisé au LIRMM pour expérimenter des implémentations. Une version plus opérationnelle en a été dérivée par Jean Privat à l'UQAM sous le nom de NIT (`http://nitlanguage.org`).

Bibliographie

L. Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, 2nd edition, 2004.

R. Ducournau. Coloring, a versatile technique for implementing object-oriented languages. *Softw. Pract. Exper.*, 41(6) :627–659, 2011. doi : 10.1002/spe.1022.

R. Ducournau, F. Morandat, and J. Privat. Modules and class refinement : a metamodeling approach to object-oriented languages. Technical Report LIRMM-07021, Université Montpellier 2, 2007.

R. Ducournau, F. Morandat, and J. Privat. Empirical assessment of object-oriented implementations with multiple inheritance and static typing. In Gary T. Leavens, editor, *Proc. OOPSLA'09*, SIGPLAN Not. 44(10), pages 41–60. ACM, 2009. doi : 10.1145/1639949.1640093.

M. Gautier, G. Masini, and K. Proch. *Cours de programmation par objets. Applications à Eiffel et comparaison avec C++*. Masson, Paris, 1996.

B. Meyer. *Eiffel : The Language*. Prentice-Hall, 1992.

B. Meyer. *Eiffel, le langage*. InterEditions, Paris, 1994. Traduction française de [Meyer, 1992].

B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

Bertrand Meyer. An Eiffel tutorial. Technical report, ISE, 2001.

S.M. Omohundro and D. Stoutamire. The Sather 1.0 specification. Technical Report TR-95-057, Int. Computer Science Institute, Berkeley (CA), USA, 1995.

Jean Privat. PRM—the language. 0.2. Technical Report 06-029, LIRMM, Montpellier, 2006a.

Jean Privat. *De l'expressivité à l'efficacité, une approche modulaire des langages à objets — Le langage PRM et le compilateur prmc*. PhD thesis, Université Montpellier II, 2006b.

C. Szyperski, S. Omohundro, and S. Murer. Engineering a programming language : The type and class system of Sather. In *Proc. of First Int. Conference on Programming Languages and System Architectures*, LNCS 782. Springer Verlag, 1994.

Mads Torgersen. Virtual types are statically safe. In *Elec. Proc. of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL 5)*, 1998.

Troisième partie

Méta et Cie

11

Méta-programmation et réflexivité

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.

Dans un premier chapitre (2), nous avons abordé les concepts de la programmation par objets, essentiellement les classes et la spécialisation, à la lumière d'Aristote. Le chapitre 3 sur la méta-modélisation nous a permis d'approfondir les notions de propriétés et d'héritage. Malgré ces approfondissements, la notion même de classe est restée assez floue, de même que les rapports des classes avec leurs instances. Ce chapitre est donc consacré aux classes, en les examinant à la lumière de notre outil universel, c'est-à-dire comme des objets.

11.1 Introduction

11.1.1 Du méta à la réflexivité, en passant par la méta-programmation

La notion de *méta* a été introduite au chapitre 3, dans un contexte de méta-modélisation. Le passage à la méta-programmation est naturel : le compilateur objet d'un langage objet en est l'exemple le plus intuitif. Cependant, la programmation d'un compilateur (ou d'un interprète) n'est pas donnée à tous : le progammeur de base n'est pas autorisé à programmer ou reprogrammer son propre compilateur, il en a bien assez avec ses propres programmes.

Le terme de *méta-programmation* prend donc le sens très particulier d'une programmation de niveau *méta* qui est accessible au programmeur de base. Cette méta-programmation peut s'exercer dans différents contextes :

— dans un cadre interprété, on parlera alors souvent de *réflexivité* ;
— dans un cadre compilé, la méta-programmation permettra d'intervenir sur la compilation, par des manipulations sur l'*arbre syntaxique* proche de *transformations source-à-source* ;
— dans un cadre de chargement dynamique, la méta-programmation permet d'intervenir à la volée lors du *chargement* d'une classe ;
— enfin, une version plus limitée de la méta-programmation, appelée *introspection*, permet d'accéder au niveau méta, celui des classes, sans pouvoir réellement le programmer.

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 *Compilation* de Master 1 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. La figure 11.10 en donne un exemple particulièrement éclairant.

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.

11.1.2 La réflexivité dans les langages de programmation

Objets de première classe. La littérature sur LISP, en particulier COMMON LISP [Steele, 1990], 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 (voir Chapitre 13).

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 comportement 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*.

Protocole de méta-objets. Un système objet extensible est en général décrit par un *protocole* consituté de classes, de leurs méthodes principales (une API) et d'une spécification minimale du graphe d'appel de ces méthodes. En termes d'UML, cela correspondrait à un double diagramme de classes et de séquences, le dernier restant assez abstrait. Dans les systèmes réflexifs, on parle de *protocole de méta-objets* (ou MOP).

11.2 La réflexivité et le modèle OBJVLISP

> Il doit y avoir quelque part une poubelle où s'amoncellent des explications.
> Une seule chose inquiète dans ce panorama :
> ce qui va arriver le jour où quelqu'un pourra expliquer aussi la poubelle.
> Julio Cortazar, *Destino de las explicaciones* in *Un tal Lucas*, 1979.

Le niveau de base est constitué des classes et de leurs instances, ces dernières n'ayant pas besoin d'être différenciées. Le méta-niveau va donc avoir pour rôle de modéliser les classes comme des objets, en décrivant la classe des classes, comme dans le méta-modèle du chapitre 3. La réflexivité va alors consister à plonger ce méta-niveau dans le niveau de base. Un modèle réflexif est donc son propre méta-modèle.

Axiome 11.1 (Réflexivité) *Les classes sont des objets comme les autres, donc instances d'une classe de classes appelée* méta-classe.

Les classes étant maintenant des objets, il est alors possible de modéliser leurs rapports avec leurs instances, c'est-à-dire le mécanisme d'instanciation. Autrement dit, de spécifier le protocole de construction de façon objet.

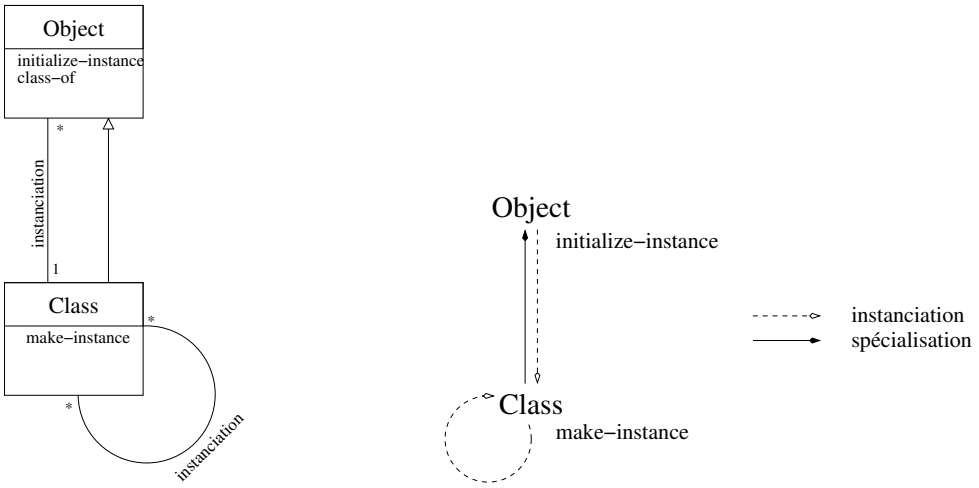


Figure 11.1 – Modèle réflexif ObjVlisp : Diagramme de classes UML (à gauche) et diagramme simplifié d'instances (à droite).

11.2.1 Le modèle ObjVlisp

Le modèle ObjVlisp [Cointe, 1987] est le modèle objet réflexif le plus simple[1]. Il est caractérisé par 2 catégories d'objets — les objets et les classes — modélisées par 2 classes, `Object` et `Class`. Le modèle ObjVlisp est schématisé dans la Figure 11.1.

Bien entendu,

- les classes sont des objets, ce qui se traduit par le fait que `Class` est une sous-classe de `Object` (par simple inversion de l'Axiome 2.1) ;
- `Object` et `Class` sont des classes, donc des instances de `Class` (par définition de `Class`) ;
- toute classe est sous-classe de `Object` (par définition de `Object`) ;
- toute méta-classe est sous-classe de `Class` (par définition 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 », puis l'initialise 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`.

Enfin, `class-of` permet à un objet d'accéder à sa classe et de s'introspecter.

L'ensemble est résumé par les diagrammes de la figure 11.1. Noter que le diagramme Uml est impuissant à exprimer l'essence du méta : la flèche de spécialisation entre `Class` et `Object` est en réalité une instance de l'association `spécialisation`.

Extension du modèle. Le modèle ObjVlisp est ouvert, 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 11.2). Chaque plan constitue le méta-niveau du plan inférieur. 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.

11.2.2 L'œuf et la poule : un problème d'amorçage

> [...] je ne connais pas de formalisme qui permette d'exprimer clairement ce qui se passe lors d'un amorçage ; cela traduit bien le manque d'intérêt actuel vis-à-vis de ce mécanisme qui devrait être plus sérieusement étudié par les théoriciens.
>
> J. Pitrat, *Métaconnaissance*, 1990

1. 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.

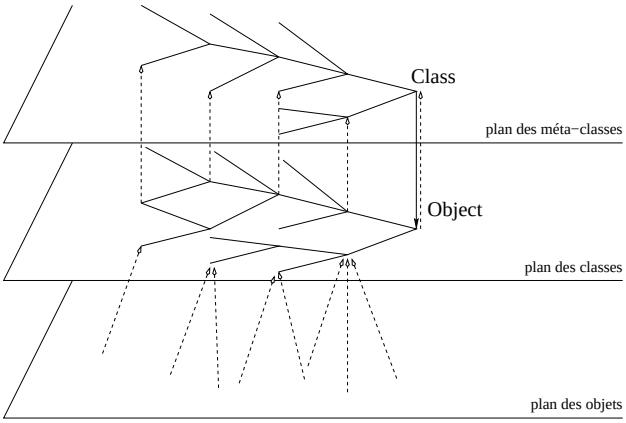


FIGURE 11.2 – Vue par plans et extension du modèle réflexif de OBJVLISP

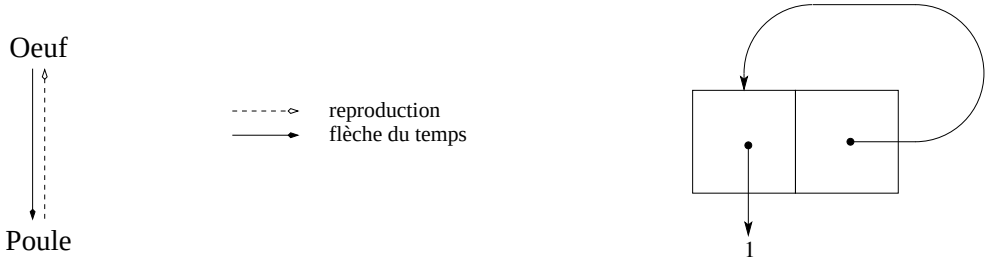


FIGURE 11.3 – Structures circulaires : l'œuf et la poule (gauche) et liste circulaire `(1 1 1 ..)` en LISP (à droite).

La réflexivité provoque un problème classique d'*amorçage* (en anglais *bootstrap*[2]), 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[3] en développant une suite de compilateurs.

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 11.3) :

— 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 ».

2. Le terme de *bootstrap* désigne classiquement en informatique le procédé par lequel un ordinateur s'initalise 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 « fairait allusion au baron de Münchausen 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 Interactif du 28 mars 2001).

3. Spinoza mentionnait déjà, au XVII^{e} siècle, que c'était un faux problème : voir exergue du présent chapitre.

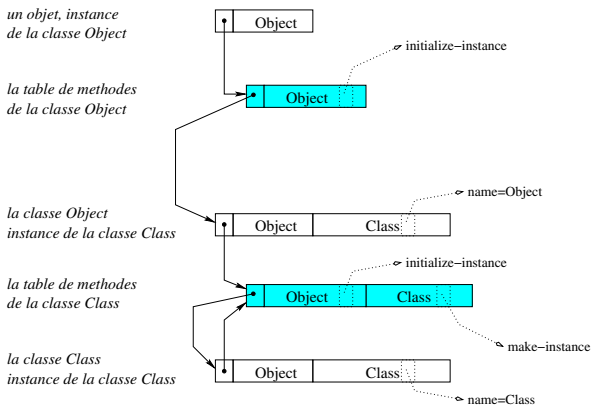


Figure 11.4 – Amorçage du noyau ObjVlisp suivant l'implémentation en héritage simple du chapitre 9 : en grisé (cyan) les tables de méthodes, en blanc les objets.

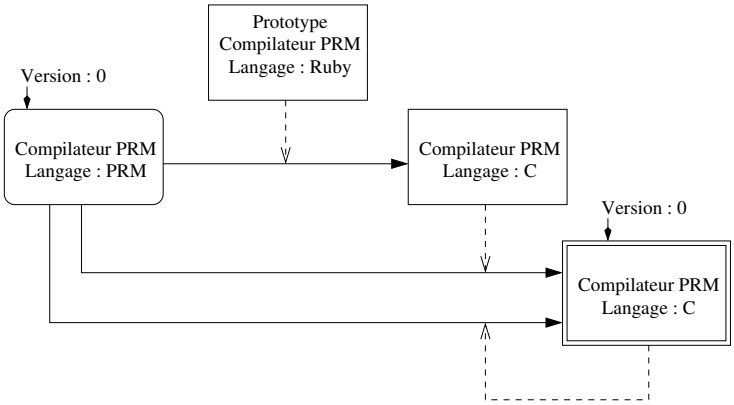


Figure 11.5 – Bootstrap du compilateur de Prm (d'après [Morandat, 2010])

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.

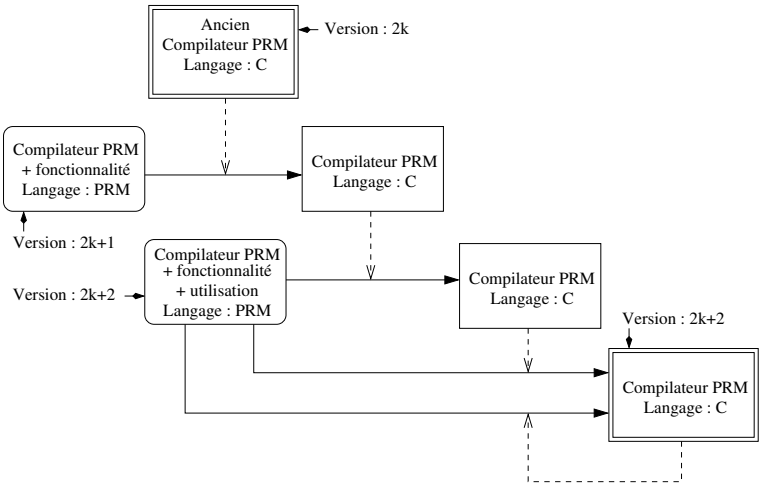
De façon similaire à la construction d'une liste circulaire, bien que plus complexe, l'amorçage de ObjVlisp va consister à construire les structures de données « à la main » — c'est-à-dire au niveau du langage sous-jacent, Lisp en l'occurrence — les structures de données qui implémentent les objets `Object`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

Si l'on suit le principe de l'implémentation de l'héritage simple décrit dans le chapitre 9, on obtiendrait alors le schéma de la figure 11.4, où chaque table de méthodes contient, en plus des adresses de méthodes, un pointeur vers la classe (l'instance de méta-classe) dont elle est la table.

Bootstrap de compilateur autogène. Créer un compilateur d'un nouveau langage, développé dans ce nouveau langage — on appelle cela un compilateur *autogène* —, est un autre cas typique d'amorçage. La figure 11.5 décrit ainsi le *bootstrap* du compilateur de Prm et la figure 11.6 montre comment le langage et le compilateur sont étendus à de nouvelles spécifications.

Bootstraps simples. Certains amorçages peuvent paraître vertigineux mais sont assez simples à réaliser. Ainsi par exemple de *la classe `Auteur` appartient à la classe des classes qui référencent leur auteur*. La classe `Auteur` doit référencer une de ses instances, ce qu'elle ne peut pas faire lors de la création de la classe, mais peut faire a posteriori, après avoir créé l'instance nécessaire.

L'amorçage est ici sans problème parce que la référence à l'auteur n'a pas de rôle actif dans l'instanciation (`make-instance`) : elle sert juste à documenter la classe. Voir l'exercice Section 14.3.



Face à une nouvelle construction du langage, l'évolution se fait en deux temps. Dans une première version, le compilateur est prêt à compiler la nouvelle construction. Il faut attendre la version suivante pour que le source du compilateur puisse utiliser cette nouvelle construction.

FIGURE 11.6 – Principe d'évolution d'un compilateur autogène (d'après [Morandat, 2010])

Bootstrap et astrophysique. Après les trous blancs et noirs de la généricité (Section 7.4.1), le bootstrap offre une bonne occasion de revenir à l'astrophysique. La meilleure analogie que l'on peut faire pour expliquer la notion de bootstrap, c'est le *big bang*. Avant, il n'y a "rien" ; après, il y a tout, c'est-à-dire un univers en ordre de marche.

Assez curieusement, le dual du *big bang*, le fameux et attendu *big crunch*, présente une forme assez similaire, qui atteint à la perfection dans l'épigramme suivant :

> Les cannibales ont disparu de Papouasie depuis que les autorités locales en ont mangé les derniers.
> A. Vialatte, cité par Simon Leys à propos des maoïstes dans *La forêt en feu*

11.2.3 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 (l'attribut est en pratique *immutable*) et partagée par 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 et l'attribut pourra être implémenté comme un champ dans la table de méthodes de la classe.

Attributs d'instance de la classe. Les classes étant des objets comme les autres, il est 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 (avec `:allocation :class`, voir Chapitre 12), mais Smalltalk propose seulement le second [4].

La classe comme attribut de l'instance. On a vu Section 5.2.3 que le type ou la classe d'un objet peut être considéré comme un attribut immutable de l'objet [5] Comme tout attribut d'instance, cet attribut est héritable, mais il est par définition redéfini dans toute sous-classe.

En revanche, la classe de la classe (c'est-à-dire la méta-classe de l'instance) ne peut pas être considérée comme un attribut de l'instance : elle n'est donc pas héritable et il n'y a aucune nécessité à vouloir qu'une classe hérite sa méta-classe d'une super-classe.

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 [6]. 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.

Cependant, 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. Notons qu'il ne faut pas non plus confondre variables statiques et *variables globales* : ces dernières sont connues globalement dans tout le code du programme, mais elles ne sont pas forcément statiques.

Pour une variable statique de C++ ou Java, seule la classe intervient — si l'on y accède au moyen d'une variable, c'est le type statique de la variable qui est considéré — et son rôle n'est que celui d'un espace de noms, l'héritage correspondant alors à une imbrication de ces espaces de noms.

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.

Instanciation et constructeurs 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 (`make-instance`), 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 (`initialize-instance`), pour (dont le receveur est) l'instance que l'on est en train

4. Pour le premier point, cela dépend des implémentations : la plupart n'autorisent pas la redéfinition de variables de classes.

5. Il est censé être partagé par toutes les instances de la classe, mais l'instance en a besoin pour accéder à la classe : le partage est donc virtuel.

6. 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*.

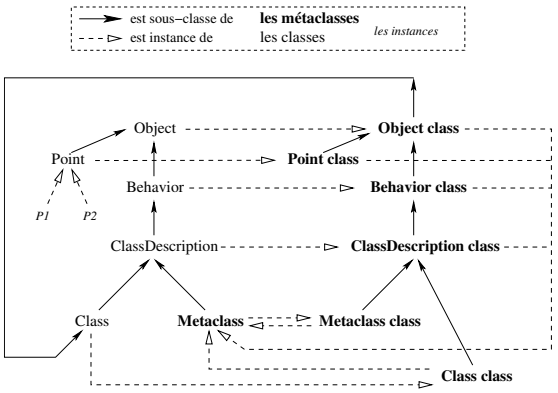


FIGURE 11.7 – Le modèle réflexif de SMALLTALK (d'après G. Pavillet)

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 (Section 6.9, page 110).

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.

Pour une méthode statique, seule la classe intervient — si l'on y accède au moyen d'une variable, c'est le type statique de la variable qui est considéré — et son rôle n'est que celui d'un espace de noms. En particulier, même si on accède à la fonction par le biais d'un objet, cet objet n'est pas un receveur et `this` n'est pas lié dans la méthode.

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[7]), il faudrait effectivement que les instances de la classe et la classe des instances soient des attributs typés, ce qui poserait un problème intéressant, dont une esquisse de solution est donnée Section 13.1.2.

11.3 Modèles réflexifs alternatifs

Pour les pionniers, l'objectif de la réflexivité était relativement clair (Axiome 11.1) mais la mise en œuvre était moins simple. Aussi, avant de converger vers un modèle unique satisfaisant — celui d'OBJVLISP— différentes variantes ont été examinées, dont seule ne reste que celle de SMALLTALK 80. Par ailleurs, des modèles réflexifs comme celui de CLOS constituent des complexifications du noyau OBJVLISP.

11.3.1 Le modèle SMALLTALK

Historiquement, SMALLTALK est le premier langage à avoir introduit la réflexivité. Son modèle actuel est le résultat d'une évolution qui s'est terminée en 1980 et il reste 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`), qui est en réalité une classe *singleton*. Comme le montre la figure 11.7, chaque méta-classe a alors, 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.

Enfin, on remarque que la réflexivité ne forme pas une boucle au niveau de la racine des classes, mais un circuit 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 11.2 si l'on met de côté la boucle de réflexivité : qui plus est, les deux plans sont parfaitement isomorphes (Figure 11.8).

7. Le problème de la théorie des types dans un langage réflexif est un problème ouvert.

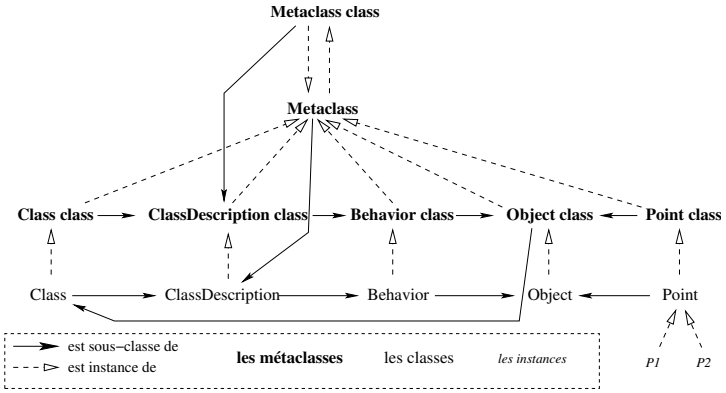


FIGURE 11.8 – Vue par plan du modèle réflexif de SMALLTALK (d'après G. Pavillet)

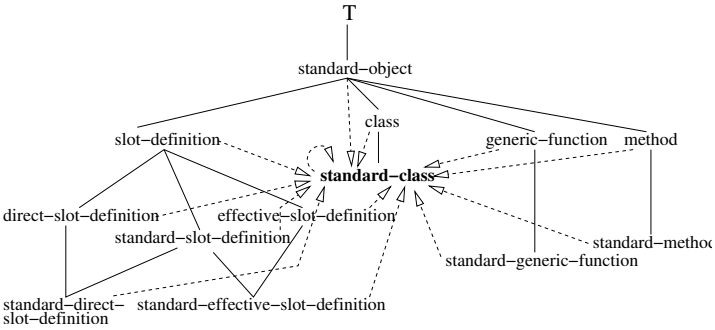


FIGURE 11.9 – Le modèle réflexif de CLOS (d'après G. Pavillet)

Le langage OBJECTIVE-C, qui peut se définir comme une implémentation de SMALLTALK dans une syntaxe proche de C, reprend un noyau réflexif analogue à celui de SMALLTALK.

11.3.2 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 11.9).

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 $\top$ (le type universel de toutes les valeurs) 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 à la section 6.2, 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 : les deux termes recouvrent sensiblement la différence entre les propriétés locales et globales du méta-modèle (Chapitre 3), la sélection multiple (Section 6.2) en plus.

Le modèles des *slots*. Les attributs sont méta-modélisés en CLOS sous la forme de *slot-definitions*, qui correspondent assez exactement à la description de *slot* qui est faite dans `defclass`. Ces *slot-definitions* sont de deux types : les *direct* correspondent à la déclaration explicite faite dans une classe alors que les *effective* sont produits par la combinaisons des descriptions du *slot* dans toutes les super-classes. Ces deux niveaux de descriptions sont sans rapport avec les 2 sortes de propriétés, locales et globales, du méta-modèle du chapitre 3 : les deux sont locaux.

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 et doit retourner `true` pour chacune. La fonction `validate-superclass` exprime donc une compatibilité entre méta-classes : celle de la classe et celle de la super-classe.

11.3.3 Le modèle PYTHON

PYTHON 2. L'instanciation repose en PYTHON sur le couple de méthode `__new__` et `__init__` dont on peut supposer qu'il prend la place des méthodes `make-instance` et `initialize-instance` de CLOS. Cependant, plusieurs indices laissent à penser qu'il s'agit de faux amis et que le modèle réflexif de PYTHON, utilisé ainsi, est très incorrect :

- les deux méthodes `__new__` et `__init__` sont appelées en séquence (au lieu que la première appelle la seconde) ;
- les exemples de méta-classes fournis dans [van Rossum] ne redéfinissent pas `__new__` mais `__init__` ;
- au contraire, `__new__` peut être redéfini dans une classe normale, par exemple :

```
class Singleton(object)
    def __new__(cls, *args, **kwds):
        ...
```

Finalement, il semble que la bonne façon de se servir des méta-classes en PYTHON repose non pas sur `__new__` mais sur `__call__`. En effet, en PYTHON, au lieu d'écrire `new Person(..)`, on écrit juste `Person(..)`. Une classe est assimilée à une fonction, et son instanciation revient à appeler cette fonction. Il existe donc une méthode `__call__`, et c'est cette méthode qu'il faut redéfinir, avec un appel à `super`, pour implémenter de nouvelles méta-classes. En corollaire, la méthode `__new__` a l'air inutile.

Classic classes vs. new-style classes. Le langage PYTHON a connu des évolutions marquées et la compatibilité avec les programmes existants a contraint les concepteurs du langage à faire coexister anciennes et nouvelles versions. Un changement majeur a été l'intégration des types et des classes dans la version 2.2. Les anciennes classes non intégrées aux types s'appellent *classic classes*, et les nouvelles *new-style classes*. Seules ces dernières — qui se reconnaissent ou se déclarent par le fait qu'elles sont sous-classes de `object` — présentent un intérêt. Malgré tout, le modèle réflexif de PYTHON reste très insatisfaisant.

En fait, le langage PYTHON est un langage de prototypes, c'est-à-dire sans classes, avec une distinction superficielle entre les classes et les instances. Ainsi la fonction `type` retournera la même valeur pour toutes les instances de *classic classes*, en indiquant juste que ce sont des instances. En revanche, la même fonction retournera une espèce d'instance, spécifique à chaque classe, pour les *new-style classes*.

L'autre intérêt des *new-style classes* est qu'elles utilisent la linéarisation C3 (Section 4.5.3).

PYTHON 3. Ce qui précède vaut pour PYTHON 2. La version 3 du langage représente une évolution majeure dont l'évaluation doit être reprise à zéro. Un premier constat est qu'il n'y a plus que des *new-style classes*. La fonction `type`, qui correspond grossièrement à `class-of` en CLOS ou `getClass` en JAVA, laisse penser que le noyau réflexif est plus proche de celui de CLOS que de celui de SMALLTALK. Ainsi, deux classes « normales » ont la même méta-classe `type`, qui représente l'équivalent de `Class` dans le modèle OBJVLISP.

Voir [Simionato, 2008] sur les nouveautés de PYTHON 3. Il ne semble pas, néanmoins, qu'il y ait eu d'amélioration notable à part la disparition des *classic classes*, mais peut-être est-elle cachée derrière une question de compatibilité.

Attention ! Pour des raisons de compatibilité, les versions 2 et 3 de PYTHON sont installées ensemble dans les installations standard, et le piège est que la commande `python` appelle en réalité `python2` et non pas `python3` (Ubuntu 13.04) ! Utilisez `python3` ! ! !

*****(à développer...)*****

11.3.4 Autres langages

De nombreux autres langages disposent de fonctionnalités de méta-programmation plus ou moins évoluées et pertinentes. Les énumérer tous est sans espoir et une recherche sur le web à partir des bons mots-clés (*reflection*, *introspection*, *metaclass*, *metaprogramming*), et les pages correspondantes des encyclopédies en ligne donnent accès à des listes plus ou moins à jour. Citons par exemple, parmi les langages non encore cités dans ce chapitre, le langage RUBY qui concourt dans la même catégorie

que Python, mais avec un noyau réflexif inspiré de Smalltalk, mais qui ressemble assez furieusement à celui de Clos : il y a même une distinction entre des objets standards et non-standard...

11.4 Des exemples canoniques

11.4.1 Définir de nouvelles méta-classes

La définition de nouvelles méta-classes suppose souvent que l'on souhaite spécifier de nouvelles relations entre une classe et ses instances. On peut imaginer différents comportements spécifiques, par exemple :

— des classes qui n'ont qu'une seule instance (*singleton*),
— des classes qui comptent leurs instances,
— des classes dont les instances sont persistantes,
— ou tout simplement résident sur une machine distante,
— ou des classes qui mémorisent leurs instances.

On peut aussi souhaiter modifier le comportement par défaut d'un langage pour des fonctionnalités qui ne reposent pas sur l'instanciation. On voudra par exemple :

— redéfinir la linéarisation du langage : celle de Clos n'est pas parfaite et on devrait préférer C3 (Section 4.5.3) mais celle-ci est apparue trop tard pour pouvoir être imposée aux programmeurs. Python a réussi à faire passer des modifications de cet ordre en faisant cohabiter deux catégories de classes différentes ;
— implémenter des associations Uml au moyen de *slots* dont les accesseurs assurent la symétrie ;
— corriger les défauts de l'héritage multiple de Clos qui ne reconnaît pas les propriétés globales.

Le chapitre 14 développe tous ces exemples et on traitera en priorité le suivant.

Exercice 11.1. La section 14.2, page 216, développe l'exemple des classes qui mémorisent leurs instances. Le traiter en Clos. □

11.4.2 Introspection

L'introspection constitue un mode de méta-programmation dans lequel on accède aux classes existantes sans avoir besoin pour autant de créer de nouvelles méta-classes. L'objectif est alors de pouvoir naviguer dans les objets indépendamment de leur type. Voir la section 13.1.1 sur l'introspection en Java.

L'introspection se révèle très utile pour implémenter des fonctionnalités génériques comme l'inspection, la copie, la comparaison et la sérialisation d'objets. Toutes ces fonctionnalités nécessitent en effet de parcourir la structure des objets, en effectuant sur chacun des attributs des traitements qui sont de fait indépendants de la classe. Bien entendu, s'il ne s'agissait d'implémenter que ces 4 fonctionnalités, cela pourrait être fait dans l'implémentation du langage, sans qu'il soit nécessaire, ni même utile, de le doter de capacités de méta-programmation.

Exercice 11.2. La section 14.4.1, page 217, développe l'exemple d'un inspecteur d'objets qui permet d'afficher le contenu d'un objet — nom et valeurs de tous les attributs — quelle que soit la classe de l'objet. Le traiter en Clos. □

Exercice 11.3. La section 14.4.3, page 218, développe l'exemple de la sérialisation d'objets qui permet d'écrire (ou sauvegarder) le contenu d'un ensemble d'objets reliés entre eux et de les relire (ou restaurer). Le traiter en Clos. □

11.5 En savoir plus

Le sage montre la lune, l'imbécile regarde le doigt.
Proverbe chinois

La réflexivité a été introduite pour implémenter l'une des premières versions de Smalltalk (1973-1976). Elle a été ensuite adoptée par presque toutes les extensions objet de Lisp, dont Clos est l'un des aboutissements, mais elle n'a trouvé une forme satisfaisante qu'avec ObjVlisp [Cointe, 1987] et Clos. [Kiczales et al., 1991] et [Forman and Danforth, 1999] sont deux ouvrages importants consacrés à la méta-programmation, pour le premier dans le cadre de Clos, et pour le second dans le cadre de SOMobjects Toolkit, avec un noyau réflexif similaire à celui d'ObjVlisp. Les schémas des divers noyaux réflexifs sont extraits de [Pavillet, 2000].

Ce chapitre s'est pour l'essentiel focalisé sur ce qu'on appelle la *réflexion de structure*. La *réflexion de comportement* (en anglais *behavioral reflection*) consiste à implémenter les mécanismes primitifs des objets par envoi de message à des méta-objets ... en essayant d'éviter une régression infinie.

Ceux qui désirent approfondir les notions de réflexivité, sur un plan aussi bien logique que philosophique ou artistique, trouveront leur bonheur dans le mythique [Hofstadter, 1985]. Enfin, d'un point de vue purement théorique, on ne connait pas de formalisation de la réflexivité. C'est certainement l'une des raisons qui bloquent son apparition en typage statique au delà de la simple introspection à la JAVA. Comme l'interprétation usuelle et intuitive des objets est ensembliste (Chapitre 2), une piste pourrait être celle des *ensembles non-fondés* de [Aczel, 1987] qui permet à un ensemble d'être son propre élément.

Bibliographie

P. Aczel. *Non-well-founded sets*. Number 14 in CSLI Lecture Notes. CSLI, 1987.

P. Cointe. Metaclasses are first class : the ObjVlisp model. In *Proc. OOPSLA'87*, SIGPLAN Not. 22(12), pages 156–167. ACM, 1987.

I. R. Forman and S. H. Danforth. *Putting Metaclasses to Work*. Addison-Wesley, 1999.

D. Hofstadter. *Gödel, Escher, Bach, les brins d'une guirlande éternelle*. InterEditions, Paris, 1985.

G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

F. Morandat. *Contribution à l'efficacité de la programmation par objets : évaluation des implémentations de l'héritage multiple en typage statique*. Thèse d'informatique, Université Montpellier 2, 2010.

G. Pavillet. *Des langages de programmation par objets à la représentation des connaissances à travers le MOP : vers une intégration*. Thèse d'informatique, Université Montpellier 2, 2000.

J. Pitrat. *Métaconnaissance, futur de l'intelligence artificielle*. Hermès, Paris, 1990.

M. Simionato. Metaclasses in Python 3.0, 2008. URL http://www.artima.com/forums/flat.jsp?forum=106&thread=236260. //www.artima.com/forums/flat.jsp?forum=106&thread=236234.

G. L. Steele. *Common Lisp, the Language*. Digital Press, second edition, 1990.

Guido van Rossum. Unifying types and classes an Python 2.2. URL http://www.python.org/download/releases/2.3//.

(a) In Le Monde du vendredi 13 janvier 2012

(b) In Le Monde du mercredi 16 janvier 2013

FIGURE 11.10 – Le méta et la réflexivité ne sont pas qu'une question scientifique...

12

Bref manuel CLOS

Ce chapitre complète le modèle de CLOS présenté en section 11.3.2.

12.1 Définition de classes

Outre la réflexivité, l'originalité principale de CLOS repose sur la façon dont les méthodes y sont spécifiées, sous la forme de *fonctions génériques* dont la sélection repose sur tous les paramètres et non plus sur un unique receveur (Section 6.2). Du fait de la sélection multiple, ou plutôt de sa spécification dans le langage, les fonctions génériques et leurs méthodes ne sont pas encapsulées dans les classes, comme dans les langages à objets classiques, mais définies à l'extérieur.

12.1.1 La macro DEFCLASS

La macro `defclass` permet de définir les classes, en en spécifiant uniquement les super-classes, la méta-classe et les attributs. Sa syntaxe est la suivante :

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

Chaque description d'attribut est une liste alternant mots-clés et valeurs, où les mots-clés sont des `keyword`s COMMON LISP, c'est-à-dire des symboles litéraux préfixés par « :» (voir [Ducournau, 2013]) :

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

Parmi les options de classe, nous ne considérerons que la déclaration d'une méta-classe (par défaut, `standard-class`), par le mot-clé `:metaclass` :

```
(:metaclass <nom-meta-classe>)
```

Exemple : On pourra ici reprendre l'exemple des polygones déjà cité au chapitre 4 :

```
(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. Voir la Section 12.2.7.

Remarque 12.1. Attention ! Dans la version courante (2.49) de CLISP, la méthode `validate-superclass` n'est pas appelée sur la super-classe par défaut (`standard-object`) lorsque celle-ci n'est pas explicitée. Il s'agit d'un défaut de spécification ou d'implémentation. Pour éviter tout problème, il faut absolument expliciter la super-classe lorsqu'on utilise une méta-classe différente de `standard-class`.

Héritage multiple. Les classes sont en héritage multiple et cet héritage est basé sur une linéarisation dont le principe est décrit en Section 4.4.

Description d'attribut

Chaque attribut peut être décrit par les différents mots-clés suivants, dans un ordre quelconque et tous étant optionnels :

```
(<nom-d-attribut> :allocation {:class | :instance}
                  :init-form <expr-eval>
                  :initarg <keyword>
                  :accessor <generic-function-name>
                  :reader <generic-function-name>
                  :writer <generic-function-name>
                  :type <expr>)
```

On distingue :

— des mots-clés d'initialisation lors de la création d'une instance :
`:initform` est suivi d'une expression évaluable dont la valeur servira à l'initialisation de l'attribut si aucune autre initialisation n'est faite ; l'expression est évaluée au moment de l'initialisation, pas du `defclass`, mais dans l'environnement lexical du `defclass` ;
`: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 ce qui se rapproche le plus d'un attribut `static` en C++ ou JAVA.

— le mot-clé `:type` n'est pas traité et sert principalement de commentaire.

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, de façon indéterminée ;

— 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.

Il ne faut malheureusement pas compter sur le système pour signaler une anomalie dans le cas d'indétermination.

Options de classe.

La seule qui nous concerne réellement est l'option qui permet de choisir une méta-classe particulière, avec le mot-clé `:metaclass` :

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

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

En cas d'absence de cette option, c'est la méta-classe `standard-class` qui est utilisée par défaut. En particulier, il n'y a pas d'héritage par défaut de la méta-classe des super-classes [1].

12.1.2 Héritage et redéfinition d'attributs

Contrairement aux langages objets « classiques », 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 (au sens de la linéarisation) 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.

12.2 Définition de méthodes et fonctions génériques

12.2.1 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 syntaxe de `defmethod` est la suivante :

```
(defmethod <method-name> [ <option> ] <λ-list> <body>)

<option> = :before | :after | :around

<λ-list> = (<param-sel> [ &key <param-opt> [ &allow-other-keys ]]
                        [ &optional <param-opt> ] [ &rest <param> ])
<param-sel> = <param> | (<param> <class-name>)
```

Les mots-clés en `<option>` seront expliqués plus loin. Dans cette syntaxe, la différence principale avec celle de `defun` est que la sélection de méthode porte sur les paramètres obligatoires, appelés ici `<param-sel>`, qui sont en général constitués d'un symbole `<param>` et d'une annotation de type `<class-name>`. Par défaut, `<class-name>` est remplacé par `t`, le type universel de toutes les valeurs. La syntaxe des paramètres optionnels `&optional`, par mots-clés `&key`, ou du paramètre restant `&rest` est la même qu'en COMMON LISP, à l'unique exception du mot-clé supplémentaire `&allow-other-keys` dont la présence indique au système qu'un mot-clé imprévu ne doit pas être traité comme une erreur, ce qui est nécessaire pour combiner les méthodes.

La macro `defgeneric` sert à définir une fonction générique : son emploi est facultatif car elle est appelée automatiquement par `defmethod` en cas de besoin. La syntaxe de `defgeneric` est restreinte à la signature de la fonction, plus quelques options que nous ne considérerons pas ici :

```
(defgeneric <method-name> <λ-list>)
```

`Defmethod` a la même syntaxe que `defun`. La seule différence est qu'il est possible de typer tous les paramètres obligatoires, en faisant figurer le type dans une liste `(<paramètre> <classe-du-paramètre>)`. L'absence de typage d'un paramètre revient à le typer par le type universel $\top$. 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 doit différer). Lors d'une définition de méthode, si une méthode de la même

1. La classe de la classe n'est pas une propriété des instances : elle n'est pas censée s'hériter (Section 11.2.3). Malgré tout, le langage aurait pu prévoir une spécification de la méta-classe par défaut qui serait héritable des super-classes, avec forcément des conflits à envisager. C'est une question de conception.

fonction générique a déjà été définie avec les mêmes types de paramètres, la nouvelle définition écrase la précédente.

Une fonction générique peut être vue comme une fonction LISP normale, dont le rôle consiste à analyser les types de ses différents paramètres pour déterminer les méthodes à appeler et combiner, et dans quel ordre. Il s'agit donc d'une implémentation de l'envoie de message complètement différente de celle qui a été décrite au Chapitre 9.

12.2.2 Création et initialisation d'instances

Pour créer des instances d'une classe, on appelle la fonction générique `make-instance` sur la classe ou sur le nom de la classe. L'implémentation par défaut de cette méthode appelle `initialize-instance` sur l'objet créé, avant de le retourner. La syntaxe de la définition de ces deux méthodes serait la suivante :

```
(defmethod make-instance ((c standard-class) &rest initargs) ...)
(defmethod initialize-instance ((c standard-object) &rest initargs) ...)
```

`c` étant la classe à instancier et `initargs` une liste alternée de mots-clés et de valeurs qui sera passée à `initialize-instance`. Les mots-clés sont ceux qui sont associés aux slots par `:initarg` dans la définition de la classe doivent être traités par le corps de `initialize-instance` (ou sont ignorés). Par défaut, le comportement de `initialize-instance` consiste à traiter les initialisations effectuées au moyen des mots-clés `:initform` et `:initarg`.

Une syntaxe alternative de la λ-liste de ces méthodes est la suivante :

```
(defmethod make-instance ((c standard-class) &key ... &allow-other-keys) ...)
(defmethod initialize-instance ((c standard-object)  &key ... &allow-other-keys) ...)
```

On appelle donc ces deux méthodes avec la syntaxe suivante :

```
(make-instance <classe> {<mot-cle> <expr>}*)
(initialize-instance <objet> {<mot-cle> <expr>}*)
```

Sur l'exemple des polygones du début de chapitre, on pourra ainsi écrire :

```
(make-instance 'polygone :sides '(1 2 3 4) :angles '(30 60 120 150))
```

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)))
```

12.2.3 Principes et conseils

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))
```

ou

```
(defgeneric initialize-instance (o &key &allow-other-keys))
```

et toute nouvelle méthode doit 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`)[2] que la fonction générique.

2. Avec la possibilité d'en accepter d'autres, par `&allow-other-keys`, qui est indispensable pour la redéfinition de `initialize-instance`. Noter que `&rest initargs` et `&key &allow-other-keys` sont à peu près équivalents.

`defgeneric` implicite ou explicite ? Lorsque `defgeneric` n'est pas appelé explicitement, le premier `defmethod` — c'est-à-dire un `defmethod` portant sur un symbole qui n'a pas encore été utilisé comme nom de fonction — provoque l'appel de `defgeneric` avec, comme arguments, la liste de paramètres de la méthode. Il est cependant conseillé de faire un `defgeneric` explicite : cela permet de recharger un fichier en repartant de zéro, alors qu'une définition implicite aurait pu conserver quelques définitions de méthodes obsolètes.

Convention de nommage. La convention de nommage des accesseurs consiste à préfixer leur nom par le nom ou une partie du nom de la classe : on pourra se restreindre à un préfixe qui rappelle la grande catégorie, pour éviter des noms trop longs. Exemples : `poly` pour les polygones (Section 4.5.2), `class` ou `slot` pour les méta-objets correspondants.

Pour les fonctions génériques qui ne sont pas des accesseurs, mais dont la sélection s'effectue sur un seul paramètre, on peut adopter une convention similaire. Mais le MOP de CLOS utilise souvent des noms de fonctions commençant par `compute`, le nom de la classe figurant plutôt dans le suffixe.

Enfin, lorsque la sélection porte sur plus d'un paramètre, il faut soit essayer d'y faire figurer 2 noms de classes, soit identifier un paramètre principal qui serait la classe dans le cas d'une multi-méthode encapsulée.

12.2.4 Combinaison de méthodes et sélection multiple

Le principe de l'appel de méthode en CLOS repose sur la sélection multiple dans le produit des types qui a été décrite à la section 6.2.

Différents moyens permettent de « combiner » les méthodes d'une même fonction générique. L'appel de la fonction `call-next-method` (avec ou sans argument) permet de faire appel à la méthode suivante dans la *linéarisation* (cf. Chapitre 4 et Section 6.7). En complément, il est possible d'ajouter aux méthodes usuelles (dites *primaires*) des méthodes secondaires (ou démons) qui sont décrits par des modalités `:before`, `:after` et `:around`.

Le principe général de la combinaison des méthodes est le suivant :

- on détermine les classes (types dynamiques) des n paramètres de l'appel, puis la linéarisation du produit de leur graphes d'héritage suivant la méthode décrite ci-dessous ;
- on recherche et évalue d'abord les méthodes `:around`, dans l'ordre de la linéarisation du maximum au minimum, en les chaînant par `call-next-method` ;
- après la dernière méthode `:around` (s'il y en a), ou en premier appel (s'il n'y en a pas),
  - on déclenche toutes les méthodes `:before`, dans l'ordre de la linéarisation,
  - puis la méthode primaire (sans option), recherchée dans l'ordre de la linéarisation du minimum au maximum, dont on retourne la valeur, après chaînage éventuel par `call-next-method`,
  - puis toutes les méthodes `:after`.
- en cas d'appel depuis une méthode `:around`, le contrôle est alors repassé aux méthodes `:around` successives qui ont appelé la première méthode primaire.

Les méthodes `:before` et `:after` ne sont pas fonctionnelles : elles doivent faire un effet de bord, ne retournent rien et ne peuvent pas appeler `call-next-method`.

Le lecteur trouvera la description plus précise dans le manuel du langage. La combinaison de méthodes peut aussi être modifiée en définissant de nouvelles classes de `method-combination`.

12.2.5 Exemple de sélection simple : `initialize-instance`

La méthode `initialize-instance` par défaut ne doit jamais être masquée car elle automatise les initialisations basées sur les mots-clés `:initform` et `:initarg`. Par ailleurs, c'est une méthode d'effet de bord qui ne retourne rien.

Il y a donc deux façons possibles de la redéfinir, par une méthode primaire qui commence par appeler `call-next-method` :

```
(defmethod initialize-instance ((p une-classe) &key ... &allow-other-keys)
  (call-next-method)
  ...
  )
```

ou par une méthode `:after` :

```
(defmethod initialize-instance :after ((p une-classe) &key ... &allow-other-keys)
  ...
  )
```

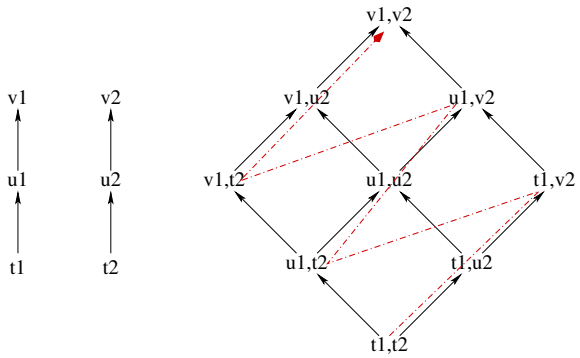


FIGURE 12.1 – Linéarisation du produit de deux hiérarchies de types.

Finir par l'appel à `call-next-method` ou faire une méthode `:before` pourrait être une grave erreur, puisque l'initialisation spécifique à la sous-classe peut nécessiter que l'initialisation spécifique à la super-classe soit déjà effectuée. En C++ ou JAVA, l'ordre de combinaison des constructeurs est imposé ainsi.

12.2.6 Linéarisation du produit d'ordre

En cas de sélection sur plusieurs arguments, c'est-à-dire lorsque plus d'un paramètre a une annotation de types, on applique la démarche suivante :

- on détermine d'abord les classes (types dynamiques) des n paramètres de l'appel,
- on construit le produit d'ordres des linéarisations de chacune de ces classes : on obtient un graphe avec un maximum, le n-uplet formé de n fois la racine des types, et un minimum, le n-uplet constitué des classes en question.
- On linéarise ce produit d'ordre avec la méthode illustrée par la Figure 12.1 qui décrit le principe de la linéarisation par défaut du produit d'ordres de la Figure 6.1. Dans l'exemple, la paire `(v1,v2)`, où `v1=v2` est la racine des types, constitue le maximum, et `(t1,t2)` est le minimum constitué des classes des arguments de l'appel.
  La linéarisation consiste à fixer le premier paramètre puis à faire varier le suivant, comme avec des boucles imbriquées. On généralise aisément à un nombre quelconque de paramètres.
- Les méthodes candidates sont alors ordonnées suivant cette linéarisation.

12.2.7 Exemple de sélection multiple sans combinaison : `validate-superclass`

La méthode `validate-superclass` offre un exemple simple de sélection sur deux paramètres, sans combinaison puisque le corps de cette méthode est en général réduit à retourner `t` ou `nil`.

Le principe de `validate-superclass` est le suivant. La méthode prend deux paramètres, la classe en cours de définition et une de ses super-classes directes :

```
(defgeneric validate-superclass (class super))
```

La méthode est appelée par le protocole de définition de classes (`defclass`) sur la classe en cours de définition et chacune de ses super-classes directes. Si l'un de ces appels retourne `nil`, une exception est signalée. Sinon, le protocole de définition de classe se poursuit.

Par défaut (au lancement du système), une seule méta-classe est définie, `standard-class`, et la définition de la fonction générique `validate-superclass` est constituée d'une unique méthode :

```
(defmethod validate-superclass ((class standard-class) (super standard-class))
    t)
```

Lorsqu'une nouvelle méta-classe est définie, par exemple :

```
(defclass my-class (standard-class)
   (...)
   (:metaclass standard-class))
```

le protocole de définition de classes bloque toute instanciation possible de cette méta-classe en définissant la méthode :

```
(defmethod validate-superclass ((class my-class) (super standard-class))
    nil)
```

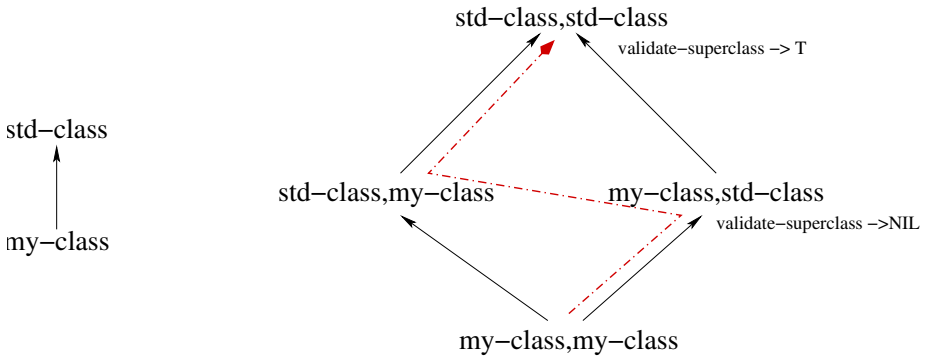


FIGURE 12.2 – Linéarisation du produit : exemple de `validate-superclass`, avec ses deux définitions pas défaut.

Noter qu'il n'est pas nécessaire de définir une méthode dans laquelle `super` serait typé par `my-class`, puisque `my-class` ne peut pas, en l'état être instanciée. Le concepteur de `my-class` doit donc définir dans quelle condition une première instance de `my-class` peut être définie, en redéfinissant la méthode qui bloque tout :

```
(defmethod validate-superclass ((class my-class) (super standard-class))
    <condition spécifique à vérifier>)
```

Il lui faut ensuite déterminer à quelle condition une instance de `my-class` peut être une sous-classe d'une autre instance de `my-class` :

```
(defmethod validate-superclass ((class my-class) (super my-class))
    <condition spécifique à vérifier>)
```

ainsi que la condition pour qu'une classe "normale" puisse être instance de `my-class` :

```
(defmethod validate-superclass ((class standard-class) (super my-class))
    <condition spécifique à vérifier>)
```

En règle générale, il faut donc définir 3 méthodes `validate-superclass` pour chaque nouvelle méta-classe. Cela peut se compliquer avec le nombre des méta-classes. Dans le cas où les 4 méthodes (en comptant celle par défaut sur `standard-class`) ne seraient pas définies, la Figure 12.2 montre la linéarisation qui serait suivie pour remplacer la ou les définitions manquantes.

Pour une application pratique, voir l'exemple sur les classes qui mémorisent leurs instances (`memo-class`) en Section 14.2.

Tests. Pour tester que `validate-superclass` marche bien, il faut essayer avec la combinatoire exhaustive de méta-classes concernées, comme dans la Figure 12.2.

Cas particulier de CLISP. La section précédente expose la théorie qui permet d'obtenir ce que l'on veut et rien que ce que l'on veut. Cependant, les implémentations du *meta-object protocol* s'éloignent souvent de la théorie. Ainsi CLISP a un défaut différent et considère que

```
(defmethod validate-superclass ((class my-class2) (super my-class1))
    t)
```

lorsque `my-class2` est sous-classe de `my-class1` [CLISP, Chapitre 29].

Cela n'empêche pas le protocole que nous proposons de marcher, mais cela peut donner l'impression fausse qu'il n'est pas utile en totalité. Si l'on souhaite être portable d'une implémentation de CLOS à une autre, il est préférable d'éviter de se reposer sur les défauts.

Par ailleurs, l'implémentation du protocole de compatibilité contient manifestement un bug lorsqu'aucune super-classe n'est explicitée : les spécifications de CLOS considèrent que la super-classe est alors, par défaut, `standard-object`, mais `validate-superclass` ne sera pas appelée car l'implémentation n'itère que sur les super-classes explicites. Il faut donc toujours expliciter les super-classes lorsqu'on instancie une méta-classe spécifique.

12.2.8 Exemple de sélection multiple et combinaison : `equals`

On a discuté plus haut de la difficulté de spécifier et implémenter proprement les tests d'égalité logique (Section 8.2). Dans ce cas, on a affaire à une méthode purement fonctionnelle, et il est hors de question d'utiliser des méthodes `:before` ou `:after`. Restent donc les méthodes primaires et `:around`.

En CLOS, on pourra définir une fonction générique `equals` de la façon suivante :

```
(defgeneric equals (x y))

(defmethod equals ((x t) (y t))
      (eq x y))

(defmethod equals ((x cons) (y cons))
      (equal x y))

(defmethod equals ((x number) (y number))
      (= x y))

(defmethod equals ((x string) (y string))
      (string-equal x y))

(defmethod equals :around ((x standard-object) (y standard-object))
      (or (eq x y)
          (and (eq (class-of x) (class-of y))
               (call-next-method))))
```

Cela revient à spécifier `equals` comme étant la fonction LISP d'égalité physique (`eq`) par défaut, la fonction d'égalité logique `equal` sur les cellules, la fonction d'égalité numérique sur les nombres, et la fonction `string-equal` pour les `string`. Rappelons que `t` est le type universel de toutes les valeurs, super-type de `standard-object`. Pour les objets standard, on définit juste une méthode `:around` qui impose que les deux arguments aient le même type, en laissant une méthode primaire plus spécialisée prendre le relais. Par défaut, ce sera la méthode définie dans le couple `<t,t>`.

On peut alors définir des classes `Point2D` et `Point3D`, avec des attributs `x`, `y` et `z`, et les méthodes `equals` suivantes :

```
(defmethod equals ((p point-2D) (q point-2D))
      (and (= (point-x p) (point-x q))
           (= (point-y p) (point-y q))))

(defmethod equals ((p point-3D) (q point-3D))
      (and (= (point-z p) (point-z q))
           (call-next-method)))
```

Si l'on compare 2 `point-3D` par `equals`, on va d'abord appeler la méthode `:around` qui va vérifier l'égalité des classes avant d'appeler, éventuellement, `call-next-method`, qui appellera la méthode des `point-3D`, laquelle appellera enfin celle des `point-2D` si c'est nécessaire.

Remarque 12.2. Il est important de noter qu'il n'y a quasiment pas d'autres solutions que d'utiliser cette méthode `:around`, du fait que l'on veut maintenir simultanément différentes propriétés :

— une condition nécessaire : l'égalité des types ;

— une condition suffisante, l'égalité physique, qui constitue également un comportement par défaut.

L'obligation de vérifier la condition nécessaire interdit de court-circuiter la méthode `:around`, ce qui garantit de plus une factorisation maximale ainsi que l'usage d'un nom unique.

En effet, faire de la méthode `:around` une méthode primaire aurait forcé l'égalité physique pour tous les objets (si l'on garde le `call-next-method`), ou aurait au contraire traité l'égalité logique comme une simple égalité de type pour les classes ne redéfinissant pas `equals`. D'autre part, en faire une fonction ordinaire aurait imposé deux noms de fonction différents, avec la possibilité d'un court-circuit. Enfin, enlever purement et simplement la méthode `:around` aurait conduit le programmeur à s'imposer la discipline d'assurer à la main la vérification de l'invariant.

On peut en tirer un constat intéressant : pour spécifier proprement quelque-chose d'aussi universel que l'égalité logique, les langages à objets, fussent-ils à typage statique, sont assez désarmés. La présente solution est basée sur des fonctions génériques à 2 paramètres, définies en dehors des classes, et disposant d'un équivalent de `call-next-method` ainsi que des méthodes `:around`. Dans un langage plus classique, des problèmes d'encapsulation se poseraient immanquablement pour l'accès aux attributs.

En Section 8.2.4, nous proposons une spécification alternative, basée sur une méthode statique, qui joue le rôle du `:around` ici, et d'une méthode covariante, avec un paramètre typé par le type virtuel

`Selftype`, ici exceptionnellement sûr grâce au test d'égalité des types dynamiques effectué auparavant. Mais cette implémentation n'empêche pas forcément les court-circuits.

Toutes ces solutions restent cependant hétérodoxes.

12.3 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 donné. Soit, par exemple, la définition suivante :

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

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

Alors, pour une instance `<object>` de `<classname>`, les 3 expressions suivantes sont équivalentes :

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

Dans les 3 cas, l'accès en lecture retournera la même valeur si le slot a bien été initialisé, mais il signalera une exception dans le cas contraire. Pour éviter cette exception il faut vérifier l'initialisation du slot avec la fonction `slot-boundp` :

```
(slot-boundp <object> '<slot-name>)
```

En écriture, on aura la même équivalence, sans les risques d'exception :

```
(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 *immutable*) ou *en lecture seule*.

Noms des attributs et des accesseurs. Le nom des attributs peut être considéré comme interne à la classe et il n'y a généralement pas de nécessité de chercher à éviter les conflits de nom en héritage multiple. En cas d'un tel conflit, les usages de `slot-value` et `slot-boundp` seront cependant ambigus, mais les accesseurs doivent permettre de désambiguïser.

En effet, les accesseurs sont des fonctions génériques usuelles, donc globales. Il faut donc utiliser les conventions de nommage usuelles en héritage multiple et typage dynamique, en incluant une partie du nom de la classe dans le nom de l'accesseur.

Conventions d'accès aux *slots*. On adoptera de préférence les conventions suivantes. Par défaut, tous les attributs sont définis avec `:accessor` et accédés par cet accesseur. Si l'attribut est immutable, on ne définira qu'un `:reader` et `slot-value` sera réservé pour les cas exceptionnels d'écriture.

12.4 Fonctions utiles

Outre les macros de définitions, un programmeur CLOS a besoin d'utiliser les fonctions suivantes :

(`call-next-method`) appelle la méthode de la fonction générique courante suivant celle-ci dans la linéarisation de ses paramètres, en lui passant les mêmes arguments que ceux de la méthode courante : déclenche une erreur en cas d'absence de méthode suivante (voir `next-method-p`) ;

il est aussi possible d'appeler `call-next-method` en lui passant une liste d'arguments compatible avec la signature de la fonction générique, mais le résultat est vraisemblablement indéterminé si les paramètres sur lesquels portent la sélection ne sont pas ceux qui ont été reçu en entrée ;

(`class-name` class) retourne le nom (`symbol`) de la classe `class` (Figure 12.3) ;

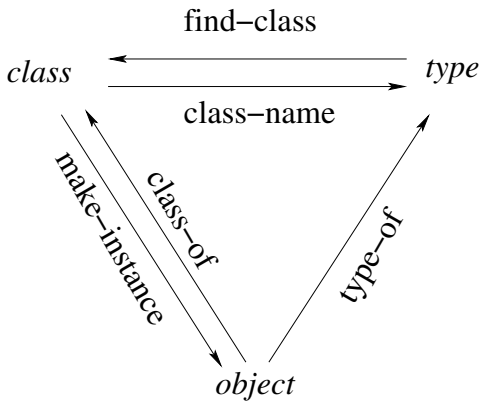


Figure 12.3 – Relations entre classes, types et objets.

(`class-of` objet) retourne la classe de l'objet, c'est-à-dire l'instance d'une méta-classe, pas son nom (Figure 12.3) ;

(`find-class` nom-de-classe) retourne la classe, c'est-à-dire l'instance d'une méta-classe, à partir de son nom, un `symbol` (Figure 12.3) ; signale une exception si aucune classe ne porte ce nom ;

(`initialize-instance` (o standard-object) &rest initargs) est la fonction d'initialisation des instances, appelée par `make-instance`. Par défaut la liste d'arguments suivants (`initargs`) est une liste mot-clé/valeur qui pourrait être décrite par `&key &allow-other-keys`.

(`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'accesseur (erreur de style) et surtout si `slot` est calculé ;

on peut faire des `setf` sur `slot-value` ;

(`subtypep` type1 type2) teste si `type1` est bien un sous-type de `type2` ; le test s'applique non seulement aux classes Clos, mais aussi au système de types de Common Lisp qui est assez compliqué ;

(`type-of` objet) retourne le type de l'objet, c'est-à-dire le nom de sa classe (Figure 12.3) ;

(`typep` objet type) teste si `objet` est bien une instance de `type` ; `(typep objet type)` est approximativement équivalent à `(subtypep (type-of objet) type)` ;

(`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 au concepteur d'une nouvelle méta-classe de spécifier cette compatibilité, en définissant les diverses combinaison de méthodes entre cette méta-classe et les autres. Par défaut, toute méta-classe est incompatible avec les autres et avec elle-même. Voir Section 12.2.7.

Le diagramme de la Figure 12.3 commute dans tous les sens :

```
type-of = class-name ∘ class-of
class-of = find-class ∘ type-of
class-name = type-of ∘ make-instance
identity = find-class ∘ class-name = class-name ∘ find-class
identity = make-instance ∘ class-of = class-of ∘ make-instance
```

12.5 Méta-programmation en Clos

12.5.1 Comment définir de nouvelles méta-classes

Le principe consiste, en toute généralité, à définir un ensemble de classes et de méta-classes qui forment un diagramme de classes, étendu à l'instanciation, connexe. La démarche générale de définition

de nouvelles méta-classes, par exemple les classes qui mémorisent leurs instances (voir Section 14.2), suit les quelques étapes suivantes dont l'ordre est à peu près imposé :

1. commencer par spécifier correctement l'objectif (étape banale qu'il n'est pas complètement inutile de rappeler) ;
2. définir les classes et méta-classes en jeu : en général, une seule méta-classe suffit, mais on peut imaginer des exemples complexes impliquant un diagramme de classes non trivial ;
3. spécifier les règles de compatibilité des méta-classes, entre elles et avec les méta-classes préexistantes, puis définir les méthodes `validate-superclass` pour chaque paire de méta-classes (x, y) où x est une nouvelle méta-classe et $x \prec y \preceq$ `standard-class` ;
4. tester ces règles de compatibilité, en définissant les classes de test pour toute la combinatoire des paires (x, y), y compris bien sûr les cas qui doivent échouer ;
5. définir les méthodes de ces classes et méta-classes, en commençant en général par `make-instance` pour les méta-classes ;
6. définir des classes "applicatives" qui instancient ces méta-classes ;
7. instancier ces classes applicatives, en vérifiant leur comportement.

12.5.2 *Meta-Object Protocol*

Le *meta-object protocol* (MOP) de CLOS est souvent mal documenté : [Kiczales et al., 1991] en décrit le principe mais les implémentations ne sont pas forcément fidèles, en particulier sur le plan du vocabulaire. Pour CLISP, il est décrit dans les notes d'implémentation [CLISP, Chapitre 29].

En pratique, on adoptera la démarche suivante :

— Le nom des fonctions accesseurs est en général préfixé par l'entité visée par les fonctions, c'est-à-dire la classe du premier argument (le « receveur » si le terme s'appliquait) : `class-` pour les classes, `slot-` pour les slots, etc. La complétion de symbole de CLISP permet donc d'obtenir aisément la liste de toutes les fonctions, par exemple pour les classes : "`(class-` " suivi de tabulation.

— les fonctions de calcul du MOP sont en général préfixées par "`compute-` " ;

— L'inspecteur d'objets (voir Section 14.4.1) permettra d'un autre côté de connaître le nom des *slots* et de leurs accesseurs. Encore un joli problème d'*amorçage* !

12.6 Lancement de CLOS

CLOS est en général intégré dans les implémentations standard de COMMON LISP, mais les détails du MOP peuvent changer d'une implémentation à une autre.

Sur les machines du réseau d'enseignement de la FDS, on utilise CLISP, que l'on lance depuis un terminal avec la commande `clisp`. Voir le polycopié de LISP [Ducournau, 2013] pour l'utilisation de CLISP.

Avec une autre implémentation que CLISP, il est conseillé de faire des essais, en utilisant CLISP comme référence.

12.7 En savoir plus

COMMON LISP résulte de l'unification de la plupart des grands dialectes LISP au début des années 1980. De la même manière, CLOS résulte de la fusion de deux surcouches objet de LISP, FLAVORS et LOOPS, et de leurs évolutions respectives, NEW FLAVORS et COMMON LOOPS. Du fait de cette histoire compliquée, COMMON LISP et CLOS sont de gros langages pleins de redondances. Aussi, les années 1990 ont vu apparaître quelques concurrents qui ont cherché à tirer parti de l'expérience de CLOS en proposant des langages plus compacts et mieux spécifiés : on peut citer DYLAN d'Apple [Shalit, 1997] et ILOGTALK et POWER-CLASSES d'ILOG [ILO, 1995, ILOG]. Malheureusement, ces langages ont été très vite lâchés par la société qui les a conçus. Il semble cependant que DYLAN existe toujours (http://www.opendylan.org/). Cependant, DYLAN semble avoir un méta-niveau similaire à celui de JAVA et restreint à l'introspection.

Les spécifications de CLOS sont incluses dans le manuel de COMMON LISP [Steele, 1990] qui est disponible en ligne, mais elles ne détaillent pas le *meta-object protocol*. [Ducournau, 2013] peut être une bonne piqure de rappel sur le langage COMMON LISP. [Keene, 1989, Habert, 1996] sont des initiations à la programmation objet en CLOS dont l'un au moins est disponible à la BU. [Kiczales et al., 1991] est la bible théorique du niveau méta et [CLISP, Chapitre 29] le procès-verbal de sa réalisation en CLISP. Pour d'autres implémentations de LISP, il faudra trouver la documentation correspondante.

Bibliographie

CLISP. Implementation notes for CLISP, 1997. URL http://clisp.cons.org/impnotes.html.

R. Ducournau. Petit Imprécis de LISP. Université Montpellier 2, polycopié de Master Informatique, 85 pages, 2013.

B. Habert. *Objectif : CLOS*. Masson, Paris, 1996. (à la BU de l'UM2).

ILOG TALK *reference manual, Version 3.2*. ILOG, Gentilly, 1995.

ILOG. *Power Classes reference manual, Version 1.4*. ILOG, Gentilly, 1996.

S.E. Keene. *Object-Oriented Programming in* COMMON LISP. *A programmer's guide to* CLOS. Addison-Wesley, Reading, MA, 1989.

G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

A. Shalit. *The Dylan Reference Manual : The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, 1997.

G. L. Steele. *Common Lisp, the Language*. Digital Press, second edition, 1990.

13

Méta-programmation et typage statique

Le chapitre 11 s'est intéressé à la réflexivité à l'exécution, dans des langages à typage dynamique. L'objectif du présent chapitre est de considérer la méta-programmation lorsqu'elle se pratique soit en typage statique, soit dans le cadre d'une génération de code réalisée de façon statique, avant l'exécution.

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.

L'introspection représente une version restreinte de la réflexivité, dans laquelle les classes existent à l'exécution, mais où les méta-classes ne peuvent pas être programmées explicitement par le programmeur.

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 13.1).

On pourra parler de *méta-programmation directe* lorsque l'espace-temps est unique, et de *méta-programmation indirecte* sinon. La méta-programmation directe regroupe la réflexivité, telle que nous l'avons présentée dans le cadre de CLOS, et l'introspection dont JAVA donne le meilleur exemple. Enfin, la méta-programmation indirecte procède par génération de code, comme la compilation, mais elle peut avoir lieu à la compilation ou au chargement (ou édition de liens).

Ce chapitre va donc examiner successivement l'introspection telle qu'elle est mise en œuvre en JAVA ou C♯, et la méta-programmation indirecte, à la compilation ou au chargement, pour finir par la programmation par aspects.

13.1 L'introspection en typage statique

13.1.1 L'introspection en JAVA et le *package* `reflect`

Le langage JAVA repose sur une machine virtuelle (la JVM [Meyer and Downing, 1997]) 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.

Le principe de l'introspection repose, ou pourrait reposer, sur les éléments suivants :

— la définition d'une classe pour représenter les classes, `java.lang.Class` en JAVA, sous-classe directe de `java.lang.Object` : cette classe décrit tous les constituants d'une classe (liste d'attributs et de méthodes), permet d'accéder à leurs noms, à la valeur d'un attribut dans un objet, et même d'invoquer une méthode (seulement si l'attribut ou la méthode sont `public`) ;

— deux méthodes permettant d'obtenir la classe d'un objet (`getClass`) et de retrouver une classe à partir de son nom (`Class.forName`) ;

— chaque classe `C` a aussi un attribut statique `C.class` qui retourne l'instance de `Class` correspondant à `C` : `C.class` et `Class.forName("C")` sont identiques, mais la première expression est beaucoup plus efficace ;

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

FIGURE 13.1 – Les 3 moments de la méta-programmation : suivant le cas, l'espace-temps des classes et des instances est le même ou pas.

CLOS	JAVA	
`T`	`java.lang.Object`	*racine de la hiérarchie*
`standard-object`	`--`	
`standard-class`	`java.lang.Class`	*méta-classe*
`class-of`	`getClass`	*introspection*
`find-class`	`Class.forName`	
`typep`	`instanceof`	*sous-typage*
`--`	`isInstance`	

FIGURE 13.2 – Equivalence entre JAVA et CLOS

— un mot-clé `instanceof` et une méthode `isInstance` permettent de faire un test de sous-typage : dans le premier cas, la classe est passée de façon statique, alors que c'est une valeur (le receveur) dans le second cas.

Toutes ces fonctions sont les équivalents des fonctions CLOS (Figure 13.2).

L'implémentation de JAVA correspond alors assez exactement au schéma de la figure 11.4 : un objet pointe sur sa classe par l'intermédiaire de sa table de méthodes, ce qui revient à faire de la classe un attribut partagé par toutes les instances (`:allocation :class`).

Sur l'introspection en JAVA le lecteur consultera l'ouvrage de [Forman and Forman, 2005].

Introspection et typage sûr. Il est possible de programmer en JAVA en restant toujours au niveau méta. C'est bien entendu d'une inefficacité absolue : (i) un appel de méthode redouble tout accès, et (ii) la propriété accédée doit être recherchée à partir de son nom.

C'est par ailleurs particulièrement non sûr, puisque les méthodes d'accès aux attributs ou d'invocation de méthodes ne permettent pas de typer plus précisément que par `Object`. Le niveau méta a donc une interface typée dynamiquement et les types de paramètres et de retour doivent être vérifiés systématiquement. Il faut même vérifier l'existence de la propriété cherchée. L'inefficacité en est donc redoublée.

EXERCICE 13.1. Développer en JAVA l'exemple de l'inspecteur d'objets (Section 14.4.1, page 217, et Exercice 11.2, page 189). □

EXERCICE 13.2. Se servir de l'introspection de JAVA pour faire un inspecteur d'objets pour SCALA (Section 4.6, page 59) qui tienne compte des *mixins*. En profiter pour analyser l'implémentation de SCALA. □

Rajouter l'introspection à un langage. Rajouter l'introspection à un langage est quelque-chose d'à la fois simple et compliqué.

— Il faut d'abord faire un méta-modèle du langage que l'on veut appareiller, au niveau d'introspection que l'on désire, et sur le "modèle" du méta-modèle présenté au Chapitre 3 ; on implémente ensuite ce méta-modèle dans le langage considéré, et cela constituera l'API de réflexion du langage.

— Il faut ensuite modifier le compilateur du langage, de façon à ce qu'il génère le code nécessaire à l'instanciation du méta-modèle pour chaque classe compilée.

Si la première partie est simple puisqu'elle peut se faire à un niveau de programmation usuel, la deuxième partie nécessite une opération à cœur ouvert du compilateur, qui peut se révéler beaucoup plus difficile. Par ailleurs, si l'introspection est actuellement une spécificité de systèmes d'exécution dynamiques comme les machines virtuelles JAVA ou .NET, il n'y a pas de raison de la cantonner à ce rôle. En théorie, rien n'empêche de rajouter une couche d'introspection à C++, EIFFEL ou PRM.

Noter que le méta-modèle d'introspection peut logiquement constituer un sous-modèle du méta-modèle du compilateur. Si le compilateur du langage est *autogène*, c'est-à-dire écrit dans le langage source considéré, les classes du compilateur peuvent être des extensions des classes de l'API d'introspection.

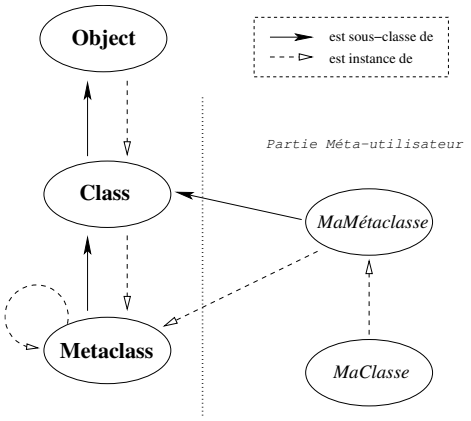


FIGURE 13.3 – Le modèle réflexif de OPENC++ (d'après G. Pavillet)

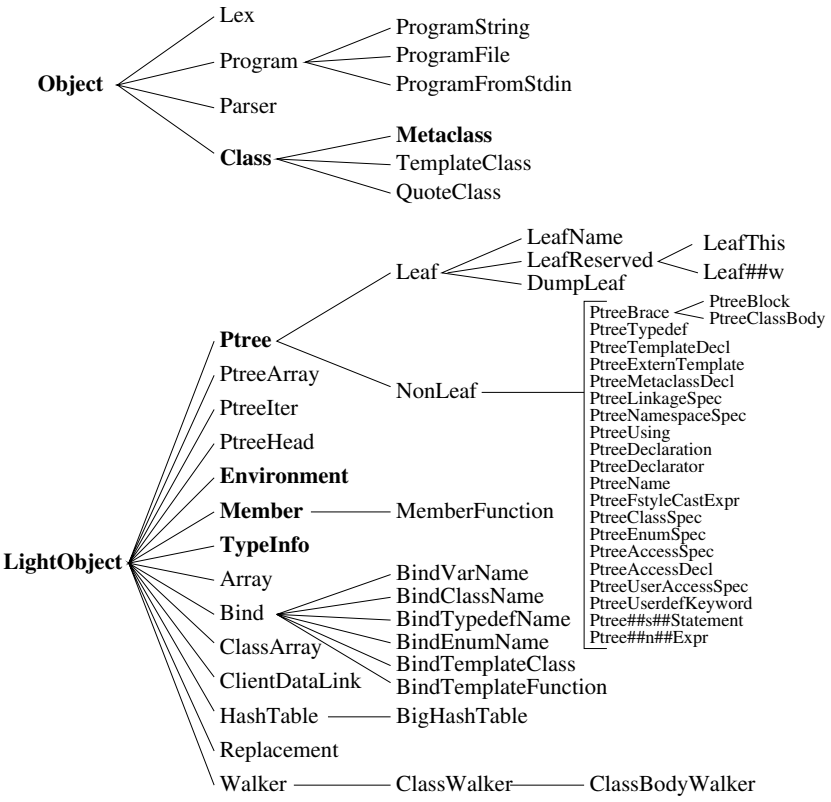


FIGURE 13.4 – Les classes de OPENC++ (d'après G. Pavillet)

Les types dans l'introspection JAVA ou C♯

En JAVA, on dispose d'une unique méta-classe `Class`. Le typage de cette classe est intéressant dans la mesure où la classe `Class` possède une méthode `newInstance ()` qui permet d'instancier ses instances. En l'état, le type de retour de cette méthode serait `Object`, ce qui n'est pas très informatif.

Pour remédier à cette triste situation, la classe `Class` est en réalité définie comme une classe paramétrée `Class⟨T⟩`. Le type de `newInstance ()` peut alors être `T`, et `C.class` est un attribut *statique* typé par `Class⟨C⟩`. De plus, chaque instance générique de `Class` est une classe *singleton*.

Lorsque l'on manipule une classe inconnue, il faut la typer avec un *joker*, par `Class⟨? <: B⟩`.

Au total, la classe `Class` et une classe `C` quelconque ont les interfaces suivantes :

```
class Class<T> {
     static Class forName(String) {}
     T newInstance() {};
     bool isInstance(Object) {} ;
}

class C {
     static Class<C> class;
     Class<? extends C> getClass() {};
}
```

Ce typage en JAVA n'est cependant possible que grâce à l'effacement de type ! Considérons la classe de `Class` : c'est `Class⟨Class⟩`, dont il est impossible d'expliciter le paramètre de type du paramètre de type. Impossible d'être plus précis que `Class⟨Class⟨? <: Object⟩⟩`. Le problème est général : pour toute classe paramétrée `A⟨T <: B⟩`, on ne peut considérer qu'une unique instance de `Class⟨A⟨? <: B⟩⟩`.

Si l'on considère l'attribut statique `class`, qui est généré implicitement dans chaque classe `C` de telle sorte que `C.class` retourne une valeur générée par `new Class⟨C⟩`, il apparait que l'attribut `class` ne peut pas être paramétré dans une classe paramétrée. Donc, dans `A⟨T <: B⟩`, `class` est typé par `Class⟨A⟨? <: B⟩⟩`.

Constructeurs et introspection

La méthode `newInstance` de JAVA n'accepte aucun paramètre, donc elle ne peut appeler que le constructeur sans paramètre, dit "par défaut". Si l'on souhaite utiliser un autre constructeur, il faut alors le rechercher par méta-programmation dans la classe considérée, puis lui appliquer la méthode `Constructor.newInstance` avec les arguments adéquats. Il faudra appliquer la même technique si le constructeur sans paramètres n'est pas `public`.

En C♯. En C♯, comme le type formel `T` n'apparait pas en paramètre de méthodes dans la définition de `Class⟨T⟩`, on pourrait considérer que le paramètre `T` est covariant, et définir la classe `Class⟨T+⟩`.

Cependant, comme il n'y a plus, en C♯, d'effacement de type ni de *joker*, il n'est pas possible de considérer `Class⟨Class⟩` comme un type bien formé. De façon plus générale, pour toute classe paramétrée `A⟨T⟩`, il faudrait considérer une instance de `Class⟨A⟨C⟩⟩` par instance générique `A⟨C⟩`. L'attribut `class` de `A⟨T⟩` est alors typé par `Class⟨A⟨T⟩⟩`, et il y a un attribut par instance générique, dont la valeur est l'instance générique. Même si cette instanciation peut être réalisée de façon paresseuse, elle peut être considérée comme trop coûteuse.

L'alternative (effectivement mise en œuvre en C♯) consiste à revenir à une classe non paramétrée, avec la perte d'information qui en résulte puisque la classe représentée ne peut plus être typée statiquement que par `Object`.

13.1.2 La réflexivité dans un langage à la JAVA ou C♯

Supposons que nous voulions étendre l'introspection de JAVA pour en faire une véritable réflexivité, avec possibilité de définir des méta-classes (sous-classes de `Class`) et de spécialiser la méthode de création d'instances.

Typage de `getClass`. Chaque objet aurait donc, au moins implicitement, un attribut immutable représentant sa classe, que l'on accèderait par `getClass()`. Etant immutable, le type de l'attribut peut être covariant. Supposons que la classe `A` soit définie avec une méta-classe `meta-A`, par exemple en faisant `class A instantiate meta-A`. Cela signifie que `A.class` serait typé par `meta-A⟨A⟩`. On peut alors supposer que le système redéfinit `getClass` de façon implicite, en le typant par `meta-A⟨? <: A⟩`. On

voit que `getClass` est doublement covariant, `meta-A` sous-classe de `Class` et `meta-A⟨? <: B⟩` sous-type de `meta-A⟨? <: A⟩` lorsque `B <: A`.

Pour résumer, le typage implicite des classes `A` et `meta-A` serait :

```
class meta-A<T> extends Class<T> {
     T newInstance() {};
}

class A instantiate meta-A {
     static meta-A<A> class;
     meta-A<? extends A> getClass() {};
}
```

Les sous-classes de `A` doivent donc être des instances de `meta-A`.

En sens inverse, le programmeur aurait cependant un degré de liberté : la classe `meta-A` pourrait aussi mentionner `A` sous la forme `meta-A⟨T <: A⟩`, en en faisant la borne de son type formel. Cela imposerait que les instances de `meta-A` soient des sous-classes de `A`.

Typage de `new`. Bien sûr, `new A()` serait alors du sucre syntaxique et un court-circuit pour `Class.forName("A").newInstance()` ou `A.class.newInstance()`, et la méthode `newInstance()` de la classe A aurait un type de retour statique A. Mais cela doit reposer sur une règle de typage particulière, puisque la méthode `newInstance` est la même pour toutes les instances d'une même méta-classe. Une solution est celle de JAVA, qui impose que pour chaque classe `C`, `C.getClass()` soit de type `Class⟨C⟩`, et que `newInstance` soit typé par le type formel de `Class`.

De plus, les instances de la méta-classe doivent être des sous-types du type de retour de sa méthode `newInstance`.

Si l'on fait `cl.newInstance`, où `cl` est une classe (`cl:Class`), on obtiendra le type de retour `Object`, ou un sous-type si le type statique de `cl` est un sous-type de `Class`, par exemple `meta-A`. `Class` doit en effet être compris comme `Class⟨? <: Object⟩`.

Maintenant, soit `MyObject` une sous-classe de `Object`. Si l'on définit une méta-classe particulière `MyClass⟨T <: MyObject⟩`, sa méthode `newInstance()` serait de type `MyObject` et la méthode `getClass()` de `MyObject` pourrait avoir `MyClass` au lieu de `Class` comme type de retour. La compatibilité des méta-classes dans un tel système pourrait être assez semblable à ce qu'elle est en SMALLTALK, au détail près qu'il ne semble pas nécessaire que chaque méta-classe soit une classe *singleton*. Cependant, cela ne semble pas être une obligation, et le type de `getClass` pourrait n'être qu'un super-type de `MyClass`.

Si jamais on souhaite que les classes mémorisent leurs instances, `MyClass` pourra définir un ensemble d'instances typé par `ImmutableSet⟨MyObject⟩` (en supposant que l'on dispose de paramètres formels covariants, comme en C♯ 4).

En JAVA ou C♯. Comme discuté pour l'introspection, la principale différence entre JAVA et C♯ résulte des spécifications différentes de la généricité. On pourra donc avoir trois spécifications différentes de ce noyau réflexif,

- à la JAVA, avec une classe `Class⟨T⟩` et des métaclasses paramétrées, de l'effacement de type et un usage de *joker* pour le typage ;
- sans généricité, avec la perte d'information que l'on observe actuellement en C♯ ; si l'on redéfinit `newInstance` dans une méta-classe `MyClass` avec un type de retour `MyObject`, il n'y a pas de perte d'information pour `MyObject`, mais il y en aura pour toutes ses sous-classes ; La seule façon d'éviter la perte d'information est que chaque méta-classe soit un *singleton* ; On retrouverait alors le modèle réflexif de SMALLTALK.
- avec une généricité à la C♯, avec une instance de `Class⟨A⟨C⟩⟩` par instance générique `A⟨C⟩` ; les instances directes de `Class⟨T⟩` doivent être générées paresseusement.

13.2 La méta-programmation par génération de code

Lorsque la méta-programmation ne se déroule pas à l'exécution, elle doit forcément procéder par génération de code. Il y a alors plusieurs possibilités.

13.2.1 La méta-programmation indirecte à la compilation

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, en réalité de faire des transformations source-à-source.

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.

Contrairement à Javassist et à l'introspection, ces outils de méta-niveau permettent de développer des applications en étendant le langage par méta-programmation.

Démarche générale

Par rapport à la réflexivité, la démarche générale de la méta-programmation à la compilation consiste à générer le code à ajouter aux classes « normales », au lieu de le programmer dans les méta-classes.

On peut donc décomposer le problème en deux étapes :

— la première consiste à concevoir la façon dont le code généré doit se comporter. Comme pour toute génération de code (voir par exemple la méthodologie de développement des macros dans [Ducournau, 2013]), on commence par simuler à la main les transformations que l'on veut faire, pour ensuite écrire le code qui les génère. Il faut donc trouver comment transformer les méta-classes en code objet de base et, pour cela, il faut analyser en quoi consiste une méta-classe. Comme toute classe, elle se définit par sa structure de données (ses attributs) et son comportement (ses méthodes).

— Une fois que la cible est identifiée, il faut automatiser la transformation, ce qui peut passer par la définition de nouveaux mots-clés, ainsi que de méta-classes spécifiques pour les traiter. Il faut alors programmer la transformation, c'est-à-dire la génération du code et son insertion.

Structure de données Une bonne façon d'aborder le problème consiste à l'examiner d'abord en Clos, en supposant qu'il n'y ait pas de méta-classes. Au lieu d'utiliser un attribut « normal », c'est-à-dire alloué dans chaque instance (`:allocation :instance` en Clos), mais défini dans la méta-classe, on pourrait alors utiliser un attribut défini et alloué dans la classe, donc avec `:allocation :class` (voir le Chapitre 12). Les deux sont équivalents en pratique, même si les sémantiques diffèrent, très exactement, d'un niveau méta.

Mais il n'y a pas d'équivalent de `:allocation :class` en Java. La seule façon d'allouer un « attribut » dans la classe est d'en faire une *variable statique* avec le mot-clé `static`. Mais ce n'est alors plus objet, dans le sens où la variable ne dépend plus du type dynamique d'un receveur. Il est cependant très simple lever cet obstacle : on appellera cela le « patron `:allocation :class` ». Il faut et il suffit d'encapsuler cette variable statique par un ou deux accesseurs, suivant que l'on désire y accéder en lecture seule, ou en écriture aussi.

Remarque 13.1. Lorsqu'il n'est pas utilisé dans un contexte de méta-programmation, ce patron pose un problème de discipline important.

— La variable statique et le ou les accesseurs doivent impérativement être définis en parallèle.

— La variable statique ne doit pas être utilisée en dehors de l'accesseur. L'annoter `private` n'est pas suffisant, puisque le programmeur qui la redéfinit pourrait être tenter de l'utiliser directement sur place.

— La consigne n'est pas héritable automatiquement : le programmeur qui introduit ce patron dans une classe doit passer la consigne aux programmeurs qui vont spécialiser cette classe !

— Ces derniers devront comprendre et appliquer la consigne.

Bref, la génération par méta-programmation est quasiment indispensable pour ce genre de patron.

Comportement Le comportement des méta-classes consiste d'abord à créer des instances, par `make-instance`. Or ce rôle est joué dans les langages sans niveau méta par une boîte noire cachée dans le `new`. Il faut donc transférer le code de méta-programmation de `make-instance` à `initialize-instance`, donc le mettre dans le constructeur. Attention, en C++, les constructeurs posent un problème (Section 6.9, page 110).

Le comportement des méta-classes consiste aussi dans des méthodes normales, qu'il faut transposer de la méta-classe dans des classes normales, ce qui ne pose pas de problème particulier.

Génération de code proprement dite

La transformation va reposer d'abord sur un protocole qui mêle un méta-modèle plus ou moins évolué à un parcours de l'arbre syntaxique du programme source, pour identifier les parties où il faut

intervenir. Dans ce contexte, le code généré est créé plus ou moins manuellement par instanciation des classes de l'arbre syntaxique. Une génération textuelle (du code entier d'une méthode) ou directement au niveau du méta-modèle (pour créer une entité complète comme un attribut ou une interface) est aussi possible.

Exercices

EXERCICE 13.3. Traiter en OPENC++ ou OPENJAVA l'exemple des classes qui mémorisent leurs instances (Section 14.2, page 216, et Exercice 11.1, page 189). □

EXERCICE 13.4. Etendre par méta-programmation JAVA ou C++ avec un mot-clé `covariant` qui permette de simuler la redéfinition covariante (Section 6.4, page 99). □

13.2.2 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 [Chiba, 1998]. ASM offre des fonctionnalités du même ordre mais serait plus efficace (`http://asm.ow2.org`).

Il est donc aussi possible d'écrire des applications qui peuvent traiter des objets de type encore inconnu, par exemple pour les espionner, voire pour vérifier qu'ils ne mettent pas en jeu la sécurité du système. Ce n'est vraisemblablement pas dans une optique de développement. De fait, il s'agit plus de logiciels qui permettent de manipuler du bytecode que de véritable méta-programmation.

13.2.3 La programmation par aspects

La programmation par aspects (ou *Aspect-Oriented Programming*, AOP) peut-être considérée comme une forme déclarative mais restreinte de méta-programmation. Elle s'intéresse à la « séparation des préoccupations » (en anglais *separation of concerns*) dans les programmes, chaque préoccupation formant un *aspect* qui doit être traité séparément avant d'être assemblé dans le programme final par un processus qui s'appelle le « tissage » et qui s'apparente à l'édition de liens.

Dans la programmation par aspects, le programmeur déclare des *points de coupe* (*pointcut*), qui lui permettent de définir, en intension, les points du flot de contrôle du programme (appelés point de jonction ou *join points*) où il désire insérer du code.

Les points de coupes représentent le versant statique, c'est-à-dire des expressions dans le code, alors que les points de jonction représentent le versant dynamique, à l'exécution. Dans le contexte statique d'un point de coupe, il est possible d'accéder au point de jonction courant, par `thisJoinPoint`, par lequel on peut alors accéder au receveur courant, par `getThis` ().

A chaque point de coupe est associé un *avis* (*advice*), qui représente le code à insérer.

Un point de coupe se décrit par une expression proche d'une expression régulière qui détermine si un point de jonction satisfait le point de coupe ou non. Les points de coupe s'expriment avec

— des mots-clés spécifiques qui décrivent le type de points de coupe (appel d'une méthode, du côté de l'appelante ou de l'appelée, ..),
— le type statique ou dynamique du receveur,
— le nom de la méthode concernée.

Dans les noms de classe ou de méthode, une * permet une quantification universelle.

Des systèmes existent comme ASPECTJ, HYPERJ et GLUONJ, en JAVA. GLUONJ est, par exemple, basé sur JAVASSIST. Des analogues doivent aussi exister en .NET.

13.2.4 Méta-programmation *a priori* vs *a posteriori*

Parmi les différents systèmes survolés ici, on peut faire la distinction suivante :

— d'une part, les systèmes comme CLOS ou OPENJAVA qui permettent une méta-programmation *a priori* : le programmeur définit des méta-classes et programme en mélangeant méta-programmation et programmation de base ;
— d'autre part, les systèmes comme JAVASSIST ou les aspects qui offrent une sorte de méta-programmation *a posteriori*, le code « métier » ne faisant pas appel expliciteemnt à la méta-programmation qui est effectuée ultérieurement et appliquée de l'extérieur.

La méta-programmation *a priori* permet en général d'étendre le langage, en y rajoutant des mots-clés, qui doivent être interprétés au niveau méta. En revanche, la méta-programmation *a posteriori* ne permet rien de tel.

13.2.5 Annotations Java

Le langage Java spécifie la notion d'*annotation*, qui consiste en un symbole préfixé par `@`, éventuellement paramétré, qui sert à annoter diverses entités de programme Java (une classe, une méthode, une instruction ou expression). Ces annotations peuvent être définies par le programmeur et, par défaut, elles ne font rien. Le compilateur se contente de les recopie fidèlement dans le *bytecode*, mais un compilateur particulier peut aussi générer des annotations.

Pour qu'une annotation ait un effet, il faut donc que le système d'exécution (machine virtuelle, compilateur JIT) ou les manipulateurs de bytecode (Javassist) les connaissent et leur associent un traitement particulier.

Les annotations offrent dont un moyen assez efficace de faire de la méta-programmation *a priori*, en utilisant des moyens *a posteriori*, tout en étendant le langage, sans avoir à en étendre la syntaxe. Le programmeur définit ses propres annotations et les insère dans le code métier, et le système de méta-programmation *a posteriori* interprète les annotations en y insérant le code idoine.

Exercice 13.5. Reprendre différents exercices du chapitre 14, en les traitant avec des annotations Java. □

Exercice 13.6. Reprendre les différentes implémentations des associations Uml des chapitres 6 et 7, en les traitant avec des aspects et des annotations Java. □

Les attributs C♯. En C♯, les annotations existent aussi, mais elle s'appellent ... attributs !

13.3 Macros, méta, généricité, etc.

La notion de méta est ubiquitaire en informatique, et il est intéressant de voir à quel point des notions apparemment étrangères sont en fait très proches dès lors qu'on les considère au niveau méta :

- les *macros* Lisp sont des fonctions qui produisent du code : c'est l'essence même de la méta-programmation ; en Lisp, les fonctions `quote` et `eval`, sont les deux fonctions qui permettent de franchir la barrière méta, c'est-à-dire de passer du niveau de base des valeurs au niveau méta de la syntaxe et du code, ou inversement ;
- dans les langages plus classiques comme C ou PL/1, ces macros sont remplacées par un *préprocesseur* ;
- l'instanciation du code générique (ou paramétré, Chapitre 7) correspond ainsi à l'*expansion* de macros, et c'est exactement comme cela que le conçoivent les compilateurs C++ ;
- lorsque ce code générique est constitué de classes paramétrées, son instanciation est similaire à l'instanciation de méta-classes : une classe paramétrée peut donc être considérée comme une sorte de méta-classe non standard.

13.4 En savoir plus

OpenC++, OpenJava (aussi appelé OJ), Javassist puis GluonJ ont été développés, dans l'ordre, par la même équipe au Japon (http://www.csg.ci.i.u-tokyo.ac.jp/) [Chiba, 1998, Tatsubori et al., 2000]. Le premier combine donc les lourdeurs de C++ et d'une première expérience de méta-programmation à la compilation. Par opposition, les autres bénéficient à la fois de Java et des expériences successives.

Depuis quelques années, la méta-modélisation a pris beaucoup d'importance car c'est la base d'Uml, des ADL comme Eclipse et de l'ingénierie des modèles.

La méta-programmation est pour sa part beaucoup utilisée dans les architectures à base de composants au-dessus de Java. D'après Guillaume Grange, ingénieur consultant et ancien étudiant :

> Javassist a été repris par la communauté JBoss, un des acteurs principaux dans le monde des applications d'entreprises. (http://jboss.org/javassist). Javassist est aussi utilisé dans le *framework* de *mapping* objet/relationnel Hibernate (http://www.hibernate.org/). Il sert en particulier à gérer les chargements de données à la demande (*lazy loading*) lors de parcours de graphes d'objets persistés en base de données. Aujourd'hui, Hibernate est le principal *framework* d'accès aux données dans les applications. Le requêtage de SGBD directement par des requêtes Sql n'est quasiment plus utilisé.

Côté C++, si OpenC++ n'a plus guère qu'un intérêt historique, sa bibliothèque a été réutilisée pour développer la bibliothèque VivaCore, un *framework* d'analyse et de manipulation de programme C++ (http://www.viva64.com/vivacore-library/).

Le terme de *separation of concerns* a été introduit par E. W. Dijskra [1982] non pas à propos du génie logiciel et de la programmation, où il est abondamment utilisé aujourd'hui, mais comme un principe général de décomposition des problèmes dans une approche scientifique.

Bibliographie

S. Chiba. Javassist—a reflection-based programming wizard for Java. In *Proc. ACM OOPSLA Workshop on Reflective Programming in C++ and Java*, 1998.

E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing : A Personal Perspective*, page 60–66. Springer-Verlag, 1982.

R. Ducournau. Petit Imprécis de LISP. Université Montpellier 2, polycopié de Master Informatique, 85 pages, 2013.

I. R. Forman and N. Forman. *Java Reflection in Action*. Manning, 2005.

J. Meyer and T. Downing. *JAVA Virtual Machine*. O'Reilly, 1997.

M. Tatsubori, S. Chiba, M.-Ol. Killijian, and K. Itano. Openjava : A class-based macro system for java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, LNCS 1826, pages 117–133. Springer-Verlag, 2000.

14

Exercices de niveau méta

A) animaux sur lesquels on fait des paris,
B) animaux dont la chasse est interdite du 1^{er} avril au 15 septembre,
C) baleines échouées,
D) animaux dont l'entrée sur le territoire national est soumis à quarantaine,
E) animaux en copropriété, F) animaux empaillés, G) *et cætera*[1],
H) animaux susceptibles de communiquer la lèpre, I) chiens d'aveugle,
J) animaux bénéficiaires d'héritages importants,
K) animaux pouvant être transportés en cabine, L) chiens perdus sans collier,
M) ânes, N) juments présumées pleines.

Georges Perec, *Penser/Classer*, 1982

Les divers problèmes de modélisation et d'implémentation décrits ci-dessous forment un tout cohérent, et la plus grande partie est traitable 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.

A part le premier exemple, tous ces exercices ciblent un niveau méta mais l'application à d'autres langages comme C++, Java ou Eiffel est en général possible, au prix de plus grandes acrobaties. Enfin, une partie de ces exercices ne concerne que les langages à typage statique, qu'il s'agit d'étendre par méta-programmation.

14.1 Les polygones

Objectif. Développer l'exemple esquissé au Chapitre 12 et dans la figure 4.13, page 56, 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[2], 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.

14.1.1 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 π, etc.), ou au calcul de certains attributs.

1. *Cet « etc. » n'a rien de surprenant en soi ; c'est seulement sa place dans la liste qui le rend curieux* (N.d.A.).
2. Dont on se rappellera qu'elle vaut $(n-2)\pi$, où n est le nombre de côtés.

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 (Section 12.4, page 201).

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

14.1.2 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.

En Eiffel.

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

14.2 Classes qui mémorisent leurs instances

On considère un système objet comme une base de données (un SGBD), à laquelle on peut soumettre des requêtes pour retrouver les objets possédant telle ou telle propriété. Le principe fondamental est de définir des classes *qui mémorisent leurs propres instances*, et uniquement celles-ci.

14.2.1 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 d'abord pouvoir définir `memo-object`. Une fois que c'est fait, on peut se préoccuper de n'accepter comme instances de `memo-class` que des sous-classes de `memo-object`, et réciproquement.

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`.

Avec héritage multiple. La spécification de `validate-superclass` pose un problème en cas d'héritage multiple. En effet, le protocole impose que la fonction retourne `true` pour chacune des super-classes (quantification universelle) alors que, dans le cas des `memo-class`, il suffit qu'une des super-classes soit une `memo-class` (quantification existentielle).

Définir les méthodes `validate-superclass` en conséquence (Voir aussi la section 14.8.2).

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.

14.2.2 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 CLOS sans méta-programmation. Curieusement, CLOS se prête très bien à cet exemple sans utiliser la méta-programmation (ni méta-classe, ni modification de `make-instance`). C'est un très bon exercice qui doit donner des pistes pour le faire dans d'autres langages sans niveau méta.

La solution réflexive passe par `make-instance` et un attribut d'instance de la classe : par quoi les remplacer ? La réponse est donnée, implicitement, dans le chapitre 11.

L'une des conclusions à en tirer est que l'automatisme de la méta-programmation doit être remplacé par un *pattern* subjectif que doit suivre le programmeur.

En JAVA, C♯ ou C++. La transposition de CLOS sans méta-programmation à JAVA ou C++ doit passer par les constructeurs et des variables `static`. Il faut définir, 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), en accédant dynamiquement à cette variable statique au moyen du « patron `:allocation :class` » (page 210).

En C++, un problème particulier résulte des limitations des constructeurs.

En JAVA, on pourra se servir de *joker*s pour assurer la covariance des ensembles d'instances, au prix d'un *cast* toujours sûr mais invérifiable dynamiquement à cause de l'effacement de type.

En OPENC++ ou OPENJAVA. Le principe va être de générer automatiquement le code nécessaire pour C++ ou JAVA, tel que spécifié ci-dessus.

14.3 Classes référençant leur auteur

La réflexivité et la théorie des ensembles sont riches en paradoxes, réels ou apparents. On peut ainsi vouloir définir :

— une classe `Personne`, et sa sous-classe `Auteur` qui représente le créateur d'une classe ;
— définir une méta-classe `Author-Class` qui serait la classe des classes référençant leur auteur ;
— tout en souhaitant que les deux classes `Personne` et `Auteur` soient elles-mêmes des instances de `Author-Class`.

Définir ces trois classes en CLOS, ainsi que l'auteur qui les définit.

14.4 Introspection

L'introspection permet l'affichage du contenu d'un objet, la navigation d'un objet à un autre, la copie d'objets ou encore la sauvegarde d'un ensemble d'objets sur un support externe — JAVA a popularisé cette dernière fonctionnalité sous le nom de *sérialisation*.

14.4.1 Inspecteur d'objets

C'est un outil indispensable pour débugguer et on aimerait bien l'avoir déjà pour le faire — encore un problème d'amorçage !

Objectif. Il s'agit d'abord 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.

Dans un deuxième temps, on peut inclure une interface navigationnelle, en numérotant les attributs et en permettant d'inspecter récursivement les objets qui sont valeurs des attributs.

Remarque 14.1. 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.

La navigation nécessite d'ouvrir une boucle `read-inspect`, dont les commandes sont le numéro du slot à inspecter, `up` pour remonter et `quit` pour sortir de l'inspection. Il faudra aussi étendre `inspect-object` aux listes et tableaux pour pouvoir naviguer simplement partout.

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'objets ne pose aucun problème. En JAVA on pourra ajouter aux spécifications de l'inspecteur les fonctionnalités graphiques interactives permettant de naviguer d'un simple clic.

14.4.2 Copie d'objets

La copie d'objets consiste à créer une instance de la classe de l'objet original pour y recopier son contenu.

Remarque 14.2. La copie d'un objet peut enfreindre les contraintes d'arité d'une association dans un diagramme UML. En réalité, il faudrait souvent faire la copie d'une ensemble d'objets, d'une façon un peu similaire à la sérialisation.

14.4.3 Sérialisation d'objets

La *sérialisation* consiste à écrire le contenu d'un ensemble d'objets sur un support externe : c'est une sauvegarde. Inversement, il est ensuite possible de restaurer cet ensemble d'objets dans une application incluant les classes considérées.

*****(à développer...)*****

14.5 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 7.1, page 119) 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[3] : 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)
```

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`. On peut enfin se poser la question du paramétrage des fonctions, par exemple de la méthode `zip` (Section 7.13).

Types virtuels Alternativement, on pourra adopter l'approche des types virtuels (Section 7.9).

En Smalltalk.

14.6 Attributs 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.

14.6.1 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.

14.6.2 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 diagramme explicitant les zones mémoire affectées à ces différentes entités) ;

3. 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.

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.

14.7 Combinaison d'exemples et variations

Cette section énumère un certain nombre de variations autour (i) des mémo-classes et des classes qui ont un comportement spécifique à l'instanciation, (ii) du noyau OBJVLISP et des méta-classes comme objets de première classe, (iii) de la combinaison de diverses méta-classes par héritage multiple.

Dans tous les exercices qui suivent, faites des diagrammes (sur le modèle d'OBJVLISP), en incluant les relations de spécialisation et d'instanciation, ainsi que les méthodes et attributs qui sont définis dans chaque classe. Certains exemples se traitent bien en CLOS, mais d'autres ne doivent pas dépasser la modélisation.

14.7.1 Méta-méta-classes

Le modèle réflexif d'OBJVLISP ou de CLOS distingue seulement les classes et les objets. Si `Class` (ou `standard-class`) est une méta-classe, les méta-classes ne sont pas des « objets de première classe » au sens où il n'y a pas de classes de méta-classes.

EXERCICE 14.1. Etendre le modèle OBJVLISP/CLOS pour y rajouter la méta-méta-classe `metaclass`. Le faire d'abord en tant qu'utilisateur, puis en tant que créateur du noyau réflexif. □

EXERCICE 14.2. Intégrer les mémo-classes dans ce noyau réflexif étendu. □

Remarque 14.3. Attention à la marche méta. Les 3 classes `object`, `class` et `meta-class` ont une définition stricte au sens mathématique du terme, c'est-à-dire une condition nécessaire et suffisante. Ce sont respectivement, les classes de tous les objets, de toutes les classes et de toutes les méta-classes. Une méta-classe ne peut donc jamais être instance de `class` sans être instance de `meta-class`.

14.7.2 Classes qui clonent leur dernière instance

Une variation sur le thème des mémo-classes consiste à définir la classe des classes qui mémorisent leur dernière instance, et dont l'instanciation consiste à cloner cette dernière instance. Un exemple pratique pourrait être celui d'un *gestionnaire de versions* qui permet de modifier un objet, jusqu'à ce qu'on décide de le figer et de dériver une nouvelle version à partir d'un clone.

EXERCICE 14.3. Développer cet exemple en CLOS sur le même schéma que les mémo-classes, en incluant un chaînage explicite des versions dans un arbre. Traiter aussi le cas particulier de la libération d'instances. □

EXERCICE 14.4. Inclure dans l'exemple la méta-méta-classe `meta-class`. □

14.7.3 Héritage multiple de méta-classes

On peut étendre l'exercice précédent en considérant que le gestionnaire de version devrait aussi mémoriser ses instances, donc être une mémo-classe.

EXERCICE 14.5. Intégrer par héritage multiple les méta-classes `memo-class` et `memo-last-class`. Inclure aussi la méta-méta-classe `meta-class`. Analyser les questions d'héritage multiple. □

Remarque 14.4. Attention à l'héritage multiple : c'est tellement facile en CLOS que l'on croit qu'il n'y a rien à faire. Il faut quand même vérifier qu'il n'y a pas de conflit de propriétés globales (de noms), et que l'ordre imposé par la linéarisation est compatible avec ce que l'on veut faire.

14.7.4 Gestion de la persistance ou des objets distants

L'interopérabilité des systèmes à objets (programme, bases de données, etc.) peut être mise en œuvre assez facilement au travers de la méta-programmation. On se place dans le contexte d'un programme CLOS qui communique (par ex. par RPC) avec un programme distant, en C++ par exemple.

On suppose pour cela que l'on a d'abord deux types de données LISP

— le type `external` représente l'adresse d'un objet externe ;

— les types externes eux-mêmes, par exemple les classes C++, sont simplement des `symbol`.

ainsi que 2 primitives LISP qui permettent l'interfaçage avec le monde extérieur

— `(get-external type key)` recherche l'objet externe associée à la clé (au sens des bases de données) `key` et soit retourne l'addresse (par exemple un objet de type LISP `external`) de l'objet en question, soit retourne `nil` s'il n'est pas trouvé ;

— `(make-external type key)` crée un objet externe correspondant à la clé, et retourne son adresse, c'est-à-dire ce que retournera `(get-external type key)` aux appels suivants : si un tel objet existe déjà, une exception est levée.

Dans les deux fonctions, `type` est un `symbol` qui représente le nom de la classe externe considérée et la clé `key` est considérée comme une valeur atomique quelconque, par exemple une chaîne ou un entier.

On veut définir des classes CLOS de telle sorte que chaque type externe (C++) corresponde à une classe CLOS.

EXERCICE 14.6. Sur un modèle proche des mémo-classes, définir les classes nécessaires pour étendre le noyau réflexif à ces objets externes. Penser à spécifier les attributs nécessaires pour référencer les objets et les types externes. □

EXERCICE 14.7. Définir les méthodes nécessaires (`make-instance`, etc.) de façon à ce que le monde extérieur et les fonctions `get-external` et `make-external` restent cachés au programmeur. □

Pour que l'association entre le monde CLOS et le monde extérieur soit injective, c'est-à-dire qu'un seul objet CLOS corresponde à chaque objet extérieur, il faut que les classes locales mémorisent leurs instances.

EXERCICE 14.8. Intégrer `extern-class` dans le schéma (et diagramme) des mémo-classes. □

Si l'on navigue en CLOS au travers des objets externes, on va récupérer des adresses d'objets distants dont il va falloir retrouver le représentant CLOS s'il existe. Comment peut-on faire ?

On peut aussi considérer des classes externes qui sont paramétrées.

EXERCICE 14.9. Intégrer `extern-class` dans le schéma (et diagramme) des classes paramétrées (cf. Section 14.5). □

Réalisation pratique. En pratique, on peut imaginer le protocole suivant (qui n'est pas de l'ordre de l'imagination, puisque l'auteur l'a mis en œuvre au début des années 1990.

On dispose d'un programme en C++ dont on va analyser les fichiers d'en-tête `.h` pour générer :

— le code C d'interface RPC entre les programmes LISP et C++ ;

— la définition des classes CLOS correspondant aux classes C++ ;

— la définition des méthodes CLOS qui vont appeler les fonctions C d'interface RPC.

14.7.5 Méta-classes qui mémorisent leurs instances

La gestion de la persistance donne l'envie de traiter les classes externes comme des mémo-classes, au niveau des objets eux-mêmes (les objets externes doivent être mémorisés), mais aussi au niveau méta, car les classes externes doivent aussi être mémorisées :

EXERCICE 14.10. Développer ces mémo-méta-classes, qui mémorisent les classes instances, qui sont elles-mêmes des mémo-classes. Inclure aussi la méta-méta-classe `meta-class`. □

Remarque 14.5. Attention à l'héritage multiple : ici les super-classes ne sont pas au même niveau méta. Est-ce un problème ?

Dans l'exercice précédent sur l'interopérabilité, la question de l'injectivité ne se pose pas que pour les objets finaux : elle se pose aussi pour les classes elles-mêmes, car à chaque type externe ne doit pas correspondre plus d'une classe CLOS. Encore une fois, la solution est du ressort des mémo-classes.

EXERCICE 14.11. Intégrer les `extern-class` dans les mémo-classes, aux deux niveaux des classes et des objets. □

14.7.6 Classes finales

De nombreux langages ont un mot-clé comme `final` (JAVA), `sealed` (C♯) ou `frozen` (EIFFEL), pour indiquer qu'une classe ne peut pas être spécialisée. Cela peut se faire facilement par méta-programmation.

EXERCICE 14.12. Définir, en CLOS, la méta-classe `final-class`. Définir en particulier les méthodes `validate-superclass` qui vont assurer à ses instances leur comportement spécifique. □

14.7.7 Classes singleton et méthodes `once` d'EIFFEL

Les langages objets proposent différentes entités « à un coup », dont l'activation n'a lieu qu'une seule fois.

Classes singleton. Les classes *singleton* n'ont qu'une instance unique. Dans certains langages comme SCALA, un mot-clé `object` remplace le mot-clé `class`. La méta-programmation d'une classe singleton est très facile en CLOS. Elle est beaucoup plus imparfaite en JAVA et C++ car elle ne peut pas se simuler dans le constructeur.

EXERCICE 14.13. Définir, en CLOS, la méta-classe `singleton-class`. Etudier la possibilité (ou l'impossibilité) de spécialiser une instance de `singleton-class` par une autre, ou par une quelconque classe. Définir les méthodes `validate-superclass` en conséquence. □

Méthodes `once` d'EIFFEL. Il n'y a pas de variables statiques en EIFFEL et les méthodes `once` sont une façon de les simuler : une telle méthode n'est activée qu'une fois pour la classe et son résultat est mémorisé et retourné lors des appels ultérieurs.

Sa méta-programmation est assez simple.

Noter qu'il y a deux spécifications distinctes de cette fonctionnalité : on peut activer la méthode une seule fois pour la classe et toutes ses sous-classes — sauf si la méthode y est redéfinie. Alternativement, chaque sous-classe peut avoir sa propre activation unique de la méthode, avec sa propre valeur stockée.

Réflexes `si-besoin` Une spécification alternative serait d'activer la méthode une fois par instance. On pourrait alors définir la `surface` d'un `polygone` par une méthode qui mémoriserait son résultat. Mais dans ce cas on doit aussi pouvoir recalculer la `surface` si les dimensions du polygone ont été modifiées. De tels mécanismes se retrouvent dans les langages de *frame*s sous le nom de *réflexe* `si-besoin`.

14.7.8 Test d'égalité, méthode de hachage et clé

On a vu à la Section 8.2 la difficulté qu'il y avait à spécifier et implémenter proprement les tests d'égalité logique et le hachage d'objets.

La solution serait de déclarer explicitement une *clé*, c'est-à-dire une ensemble d'attributs permettant d'identifier de façon unique un objet. A partir de cette clé, on peut facilement générer automatiquement les méthodes d'égalité logique (`equals`) et de hachage (`hashCode`).

EXERCICE 14.14. Définir les classes et méta-classes `hashable-object` et `hashable-class` qui implémentent ces fonctionanlités. On pourra le faire en étendant la spécification de `equals` faite en Section 12.2.8. □

On peut ensuite combiner cette classe avec celles qui mémorisent leurs instances de façon à ce qu'il ne soit pas possible de créer deux objets de même clé : lors de la tentative de création du second, la méthode `make-instance` retournera la première instance créée avec cette clé.

14.8 Etendre ou modifier les spécifications de CLOS

La méta-programmation peut servir à étendre un langage, voire à le modifier si ses spécifications semblent erronées. Dans tous les cas, il ne faut pas modifier les méta-objets existants et leurs méthodes, mais on peut définir une nouvelle méta-classe « standard » que l'on utilise ensuite à la place de `standard-class`. Cela revient à faire coexister des classes de différents styles, un peu comme en PYTHON.

14.8.1 Changer la linéarisation de CLOS

Définir la méta-classe `C3-class` qui implémente la linéarisation C3 (Section 4.5.3). On procédera pour cela à la redéfinition de la fonction générique `compute-class-precedence-list`.

On pourra faire plusieurs variantes sur ce thème :

— vérifier la monotonie de la linéarisation de CLOS, et avertir dans le cas contraire (dans `standard-class`) ;
— vérifier la monotonie de la sélection de méthode (contre-exemple de la figure 4.12b) ;
— vérifier les cas de non associativité de l'opérateur $\boxplus$;
— comparer C3 à la linéarisation de CLOS et avertir quand elles divergent ;
— faire de même avec d'autres linéarisations.

On pourra adapter tous ces problèmes au contexte d'autres langages réflexifs avec héritage multiple, par exemple PYTHON.

14.8.2 Changer le protocole de compatibilité des super-classes

Le protocole de compatibilité des super-classes est basé sur la fonction générique `validate-superclass` qui doit être vérifié pour chaque super-classe. On a vu (Section 14.2) que certaines méta-classes demandent plutôt que la fonction soit vérifiée pour au moins une super-classe directe. On peut encore relâcher cette contrainte en se contentant d'exiger qu'au moins une super-classe indirecte vérifie la condition. C'est suffisant si l'on souhaite simplement que la nouvelle classe hérite certaines fonctionnalités d'instances antérieures de la méta-classe.

Examiner le protocole d'appel de `validate-superclass` pour voir s'il est possible de le modifier. NB Les sources de `clisp` sont disponibles avec la distribution.

Si ce n'est pas possible par modification du protocole qui appelle `validate-superclass`, il est toujours possible d'arriver à nos fins en implémentant `validate-superclass` comme si c'était le protocole en question. Il suffit alors d'oublier le paramètre qui désigne la super-classe, pour à la place récupérer à partir de la classe en cours de définition l'ensemble de ses super-classes, directes ou indirectes, pour implémenter le comportement désiré.

14.8.3 Définir des catégories de slots particulières

Le modèle des slots de CLOS est assez primitif et de nombreuses extensions sont naturelles : certaines ont été examinées dans d'autres chapitres et d'autres contextes. L'application à d'autres langages réflexifs est bien sûr envisageable.

Propriétés globales pour les slots et les accesseurs

L'héritage multiple de CLOS ne respecte pas le méta-modèle. Cela concerne essentiellement les slots puisque les fonctions génériques sont définies en-dehors des classes.

Pour intégrer le méta-modèle, il faut donc définir de nouvelles classes de `slot-definition`, une nouvelle classe de méta-objets pour des *slots globaux* et redéfinir partiellement les mécanismes d'héritage pour garder la trace des classes d'introduction et gérer les conflits de propriétés gloables.

Si, en toute généralité, les fonctions génériques ne sont pas concernées, les accesseurs le sont. Il faudrait donc inclure le traitement des accesseurs dans cette extension.

Associations UML

On a vu que l'implémentation des associations UML posait des problèmes de typage (Sections 6.5 et 7.6). La question primordiale n'a été cependant qu'à peine effleurée. Elle repose sur les points suivants :

— l'implémentation d'une association UML binaire repose sur la définition d'un *rôle* dans chacune des 2 classes partenaires : ce rôle est un attribut don la valeur doit être une instance ou un ensemble d'instances de l'autre classe ;
— lorsque l'arité d'un rôle est potentiellement supérieure à 2 — on dit que le rôle est multi-valué — sa valeur a une sémantique ensembliste stricte et la définition d'accesseurs de modification (ajout et retrait) est souhaitable ;
— le point crucial est que les deux rôles doivent être en permanence cohérents : si a pointe sur b par le rôle 1, b doit pointer sur a par le rôle 2.

Il faut donc définir des classes particulières de `slot-definition` qui permettent de spécifier un rôle en précisant :

— la classe partenaire (on pourra utiliser le mot-clé `:type` si cela ne provoque pas de problème ;
— le rôle inverse (`:inverse`) ;
— l'arité (`:arity`) : la valeur pourra être un entier ou un intervalle ;

— le nom des accesseurs de modification (`:adder`, `:deleter`).

Il faut enfin, modifier le protocole pour intégrer ces nouveaux mots-clés et générer les méthodes associées. On pourra envisager de se servir de méthodes `:before`, `:after` ou `:around`.

Implémentation des attributs booléens

Le modèle d'implémentation de Lisp est basé sur des pointeurs implicites et des valeurs immédiates. Chaque valeur occupe donc un mot, ce qui peut-être beaucoup si la valeur considérée est booléenne. La définition d'une classe ayant plusieurs attributs de valeurs booléennes entraînera un certain gaspillage si l'on considère que ces attributs booléens auraient pu être alloués sur un seul mot.

L'idée des attributs booléens repose sur ce constat. Il faut définir une classe de `slot-definition` dotée d'un mot-clé `:boolean` (par exemple comme valeur du mot-clé `:type`), de telle sorte que ces attributs booléens n'aient pas d'implémentation en propre mais soit alloués dans un attribut supplémentaire, à concurrence d'environ 30 attributs booléens par attribut supplémentaire. Il faut donc aussi définir la classe de `slot-definition` pour ces attributs supplémentaires, dotée d'un attribut qui indique le nombre de booléens alloués. Inversement chaque attribut booléen doit connaître sa position.

Pour compléter le schéma, il faut définir les accesseurs et gérer l'héritage multiple.

Noter que le problème de l'ajout dynamique d'un nouvel attribut est assez délicat : le protocole est rarement assez précis.

Implémentation des attributs distants

Dans l'exemple des `extern-class` de la Section 14.7.4, page 221, un objet Clos représente un objet distant. Cet objet distant peut avoir des attributs auxquels on souhaite accéder depuis Clos, comme s'il s'agissait d'attributs Clos.

Une solution automatique est possible, bien qu'elle dépasse largement le cadre de ce polycopié. On commence par analyser le schéma des classes externes (C++ par exemple), pour générer automatiquement le code RPC pour y accéder ainsi que le schéma de classe Clos correspondant.

Du côté Clos, un mot-clé spécial (par ex. `:external`) précise que le slot ne doit pas être implémenté dans l'objet et permet de générer un accesseur qui fait appel au code RPC.

Nous avions procédé ainsi, au début des années 90, entre le langage Yafool et une application en C++.

14.9 Méta-modélisation du monde réel

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 des applications dans le « monde réel ».

14.9.1 Modélisation d'animaux et d'espèces animales

La modélisation des espèces animales et des animaux en offre un bon exemple. Les trois plans de la figure 11.2, page 182, 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.).

14.9.2 Modélisation d'automobiles et de marques

Si l'on considère que les automobiles individuelles (la mienne, la vôtre, etc.) sont des instances, on peut faire des modèles d'automobile des classes et, pourquoi pas ? des marques des méta-classes.

14.10 Faire une simulation de noyau réflexif typé

L'objectif est de simuler dans un langage à typage statique le noyau réflexif esquissé en Section 13.1.2. Il s'agit pour cela de définir deux classes `StdObject` et `StdClass` qui reproduisent le noyau réflexif en simulant dans le langage ce qui devrait être implémenté au niveau système. Pour ce faire, on peut envisager 3 langages :

— en C♯ : on dispose pour cela de la généricité non effacée, d'annotations de variance (mais seulement sur les interfaces) ; inconvénient supplémentaire : pas de covariance sur le type de retour ; au total, la simulation est possible avec quelques lourdeurs et des casts pour compenser l'absence de covariance ;

— en Scala : on dispose alors d'annotations de variance sur toutes les classes et de la covariance du type de retour, mais la généricité est effacée ; on n'arrivera donc pas à simuler la boucle récursive, mais tout le reste sera plus simple qu'en C♯ ;
— en Nit : on dispose de la covariance généralisée sur les génériques ; la simulation sera plus facile à écrire mais la vérification statique sera moins sûre.

En Scala comme en Nit on peut envisager de se servir aussi de types virtuels.

14.11 En savoir plus

[Forman and Danforth, 1999] développe de nombreux autres exemples d'utilisation de la réflexivité. L'exemple des attributs booléens a été traité par Gabriel Pavillet dans sa thèse pour étudier les capacités expressives de différents MOP (ceux de Clos, IlogTalk/Power-Classes, OpenC++) [Pavillet, 2000].

Dans les années 1980 sont apparus une famille de langages à objets dits « de *frames* » destinés à la représentation des connaissances et permettant de faire de la programmation à objets avec des mécanismes de haut niveau comme l'implémentation des associations Uml avant la lettre — voir par exemple [Masini et al., 1989]. Les langages à objets modernes ont presque complètement perdu ces mécanismes de haut niveau, que l'on retrouve partiellement dans Power-Classes [ILOG], qui a malheureusement disparu corps et biens, au début des années 2000. Il comprenait un MOP très détaillé pour les *slots*, ainsi que la génération automatique d'accesseurs pour gérer les associations (Section 14.8.3). Un langage comme Arom a poursuivi cette tradition, jusque vers 2005 (http://www.inrialpes.fr/romans/arom/).

Bibliographie

I. R. Forman and S. H. Danforth. *Putting Metaclasses to Work.* Addison-Wesley, 1999.

ILOG. *Power Classes reference manual, Version 1.4.* ILOG, Gentilly, 1996.

G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Les langages à objets.* InterEditions, Paris, 1989.

G. Pavillet. *Des langages de programmation par objets à la représentation des connaissances à travers le MOP : vers une intégration.* Thèse d'informatique, Université Montpellier 2, 2000.

Quatrième partie

Et cœtera

15

Conclusion provisoire

Les prévisions sont difficiles,
surtout lorsqu'elles concernent l'avenir.
Pierre Dac

La programmation par objets représente aujourd'hui, depuis une vingtaine d'années, le paradigme dominant de programmation. Il n'y a pas raison que cela change à court ou moyen terme. Cependant, les langages eux-mêmes vont changer, pour le meilleur ou pour le pire.

Malgré leur intérêt, les langages à objets sont loin d'être parfaits. Il reste des difficultés sans doute insurmontables et incontournables. Pour n'en citer que deux, la combinaison des méthodes en cas de conflit de propriétés locales en héritage multiple, et la question de la covariance. Dans les deux cas, les progrès apportés par la linéarisation C3 ou par les types virtuels n'ont pas fait disparaître le problème : ils en ont juste atténué le caractère critique. Le concepteur-programmeur devra rester vigilant.

D'autre part, dans le détail, les langages à objets sont encore aisément perfectibles, c'est un euphémisme. Des défauts de spécification, plus ou moins nombreux suivant les langages, devraient finir par être corrigés, au moins dans les nouveaux langages, car l'évolution de chaque langage est contrainte par une nécessaire compatibilité avec les programmes existants. Les longs débats qui ont présidé à l'introduction de la généricité en Java 1.5 montrent bien l'importance de cette compatibilité — en Java, l'un des défis était d'augmenter le langage sans toucher à la Jvm. A contrario, un langage comme IlogTalk est mort de n'avoir pas pu assurer une compatibilité minimale avec le dialecte LeLisp dont il était censé prendre la relève. Les langages réflexifs posent finalement moins de problèmes. Rien n'empêche par exemple de programmer en Clos en utilisant la linéarisation monotone C3 et Python 2.2 a institutionnalisé un langage à deux vitesses, qui fait cohabiter les horreurs des premières versions (*classic classes*) avec les spécifications plus conformes mais encore imparfaites des *new-style classes*.

Les progrès cités plus haut tardent encore à être intégrés à un langage complet et grand public. Si le compilateur est contraint au pessimisme, le programmeur a droit à un optimisme prudent. Le langage Prm est une réponse partielle à ces problèmes il est probable qu'il prendra une place plus importante dans les années qui viennent, ... au moins dans ce cours.

Par ailleurs, des besoins nouveaux aussi imprévisibles que le Web il y a 20 ans, ou des bouleversements techniques, peuvent survenir, qui renouvellent complètement la problématique de la programmation.

Aussi les (apprentis-)programmeurs d'aujourd'hui doivent s'attendre à voir apparaître des syntaxes ou des concepts qu'il faudra bien qualifier de nouveaux, même si certains existent depuis la nuit des temps ... des objets.

D'un point de vue plus prosaïque, ce support de cours continuera à s'enrichir au fil du temps et des étudiants. Des notions comme les aspects et la modularité devraient finir par prendre une plus grande place.

16

Glossaire

Mal nommer les choses, c'est ajouter aux malheurs du monde
Albert Camus

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.

abstract **:** le terme anglais pour **abstrait**.

abstrait : Lorsqu'abstrait (*abstract*) qualifie une classe ou une méthode, le terme désigne le fait que la classe n'est pas instanciable, donc n'a pas d'instances directes, ou que la méthode n'est pas implémentée dans la classe courante, qui est de fait abstraite.

Les langages l'expriment la plus souvent par le mot-clé `abstract` (Java, Smalltalk), mais aussi par `deferred` (Eiffel), voire `virtual`.

Le terme est bien entendu utilisé aussi dans d'autres contextes, par exemple pour *type abstrait de données*.

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

amorçage : Le procédé transcendant par lequel on crée un système réflexif.

aspect : Dans la programmation par aspects, le terme d'aspect désigne une « préoccupation », c'est-à-dire un ensemble de traitements spécifiques, qui peuvent être injectés, de façon déclarative, dans un programme existant.

association : Dans la terminologie Uml, ce terme désigne des relations entre classes qui peuvent être interprétées dans le sens ensembliste, comme partie du produit cartésien des extensions des classes considérées.

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

Les langages utilisent des termes très variés pour désigner les attributs : variables d'instance (Smalltalk, Java), *data member* (C++), *slot* (CLOS), *champ*, etc.

Inversement, *attribute* est utilisé, en Java, pour désigner des annotations (préfixées par '@') des classes et des propriétés.

bootstrap **:** Le nom anglais pour **amorçage** et la boucle de chaussure pour aller dans la Lune.

cast **:** le petit mot familier pour **coercition** de type.

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

coercition : son principal usage en programmation par objets est un test de sous-typage dynamique (*downcast*) dont l'usage devrait être strictement encadré et qui représente une certaine hypocrisie du typage sûr. Son usage ascendant ne peut servir qu'à la désambguïsation de la surcharge statique.

conflit : (ou *ambiguïté*) terme générique qui désigne le fait qu'un fragment de programme ne dénote pas une entité unique bien spécifiée. On peut distinguer les conflits de **propriétés globales**, ceux de **propriétés locales** — les deux catégories étant liées à l'**héritage multiple** — ainsi que les conflits de résolution de la **surcharge statique**.

constructeur : fausse méthode et faux-ami : le constructeur est en fait une méthode d'initialisation des instances, lors de l'instanciation. Dans les langages qui utilisent le terme, c'est une fonction statique.

Dans les langages qui n'utilisent pas le terme (EIFFEL ou PRM) ces initialiseurs sont des méthodes normales, mais leur fonction d'initialiseur doit être déclarée explicitement et ne s'hérite pas. En CLOS, la fonction générique `initialize-instance` est une méthode normale dont le rôle est fixé par le *meta-object protocol*.

contravariance : indique simplement une évolution monotone, en sens contraire (par exemple contravariance des intensions), par opposition à une évolution dans le même sens, dite **covariante**.

La *règle de contravariance* impose la contravariance des types de paramètres mais la covariance du type de retour.

covariance : indique simplement une évolution monotone, dans le même sens (par exemple covariance des extensions), par rapport à une relation d'ordre de base.

La *règle de contravariance* impose une covariance des types de retour.

`deferred` **:** le terme EIFFEL pour **abstract**.

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

droits d'accès voir **visibilité**.

dynamique : par opposition à **statique**, désigne de façon générale ce qui se fait ou se détermine à chaud, en présence de valeurs, à l'exécution.

Dans un contexte objet, le terme s'applique aussi aux instances (par opposition aux classes, qui sont statiques).

La dynamique représente enfin tout ce qui relève de l'évolution du système, par création d'instances et envois de messages, et qui peut s'exprimer au travers d'un **protocole**.

encapsulation : le fait que les données d'un objet ne puissent 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.).

envoi de message : la métaphore qui désigne l'abstraction procédurale dans les langages à objets : l'objet **receveur** décide de son propre comportement pour réagir aux messages qu'il reçoit.

`extends` **:** le mot clé de JAVA pour la **spécialisation** entre classes ou entre interfaces.

extension : ensemble des instances d'une classe, son interprétation.

Le terme est aussi utilisé dans la littérature (mais pas ici), dans le sens de spécialisation, pour désigner la définition d'une sous-classe relativement à sa super-classe : par exemple en JAVA avec le mot-clé `extends`.

feature **:** le terme EIFFEL pour **propriété**.

final : se dit d'une propriété qui n'est pas redéfinissable dans les sous-classes (méthode), d'une classe qui ne peut pas être spécialisée, voire d'un attribut *immutable*, c'est-à-dire en lecture seule (voir aussi Section 6.8.2). Les langages peuvent proposer un mot-clé spécique pour cela, par exemple `final` en JAVA, `sealed` en C♯ et `frozen` en EIFFEL. `sealed` est aussi utilisé en SCALA pour interdire de définir d'autres sous-classes que celles qui sont définies comme classes internes.

C'est une décision importante, qui nie les évolutions ultérieures et le Méta-axiome 2.2, page 21, et qu'il ne faut donc pas prendre à la légère. Nous ne l'avons en fait considéré que dans le cas des *types virtuels*, parce que, dans ce contexte, *final* ne bloque pas les redéfinitions ultérieures mais uniquement certains usages (Section 7.9).

Dans le contexte de la réflexivité, on parle aussi d'*objet final* pour désigner un objet qui n'est pas une classe.

framework **:** désigne, de façon assez générale, un système ou un programme conçu de façon à être le plus générique possible pour pouvoir être appliqué, spécialisé, étendu, etc. à des domaines variés. On peut voir un *framework* comme une bibliothèque de classes, doublé d'un **protocole** qui décrit les enchaînements de méthodes et qui spécifie informellement la façon dont on peut le spécialiser ou l'étendre.

généricité : désigne ici le fait de pouvoir définir des types ou classes paramétrés, en général 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 **propriétés globales** (ou *noms*) et de **propriétés locales** (ou *valeurs*).

immutable : Anglicisme caractérisant ce qui n'est pas modifiable (objet, attribut, etc.) au moins dans un contexte donné (voir aussi Section 6.8.2).

`implements` **:** le mot clé de JAVA pour la **spécialisation** entre classes et interfaces.

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 désigne aussi la relation entre une classe et ses instances.

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.

interface : désigne en JAVA des classes très abstraites qui ne comportent que des signatures de méthodes et des variables statiques.

Le terme est utilisé plus généralement dans la littérature pour désigner les signatures de fonctions ou méthodes exportées par un module ou une classe. Dans ce sens, interface est proche d'**intension**, mais cette dernière comporte toutes les propriétés, alors que l'interface ne comporte que les propriétés exportées (visibles).

introspection : la possibilité pour un objet d'accéder à son niveau **méta**, c'est-à-dire à sa classe, en lecture seule, c'est-à-dire sans pouvoir la programmer à proprement parler.

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

liaison tardive : (ou *late binding*) 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**.

linéarisation : technique de traitement de l'héritage multiple qui consiste à calculer un ordre total sur les super-classes de chaque classe, l'héritage et le **masquage** s'effectuant dans cet ordre.

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 ou fonction.

Les différents langages utilisent des termes variés : routine (EIFFEL), *function member* ou fonction virtuelle (C++), etc.

mixin **:** modèle d'héritage multiple asymétrique dans lequels les classes sont en héritage simple et les *mixins* sont des espèces de classes abstraites qui peuvent être héritées en héritage multiple avec quelques contraintes.

modèle : : ce terme est omniprésent dans ce manuel, où on parle aussi bien du *modèle objet*, au sens de l'ensemble des concepts à la base de la programmation par objets, que des modèles, de type UML, mettant en pratique ce modèle objet sur des prolèmes particuliers.

monomorphe : se dit d'une expression pour indiquer qu'elle n'est pas **polymorphe**, c'est-à-dire que sa valeur de peut avoir qu'un seul type dynamique à l'exécution, en général le type statique de l'expression. Par extension, on dira qu'un appel de méthode est monomorphe si son receveur l'est, ou — par abus — si l'appel de méthode peut se résoudre par un appel statique, même si le receveur n'est pas monomorphe.

MOP : trigramme pour *meta-object protocol*.

multi-méthode : méthodes qui sont sélectionnées suivant le type dynamique de tous leurs arguments.

objet : c'est bien l'objet de tout ça.

objet de première classe ou *first-class object* : désignait, avant l'apparition de la programmation par objets et en programmation fonctionnelle en particulier, le fait que certaines entités soient des valeurs de plein droit, instance d'un type particulier — par opposition à des entités qui n'ont d'existence que syntaxique.

Une fonction dans un langage fonctionnel, ou une classe dans un langage réflexif sont des objets de première classe.

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

overriding **:** voir **redéfinition**.

patron : un *patron de conception* (ou *design pattern*) est un motif récurrent qu'il est avantageux de réutiliser dans diverses situations pour accélérer le développement et rendre le code plus compréhensible pour d'autres programmeurs. Dans un contexte objet, on peut assimiler un patron à un petit modèle UML composé de 2 ou 3 classes, parfois plus, de quelques méthodes et attributs et d'un **protocole** qui spécifie la **dynamique** du système. Ce polycopié présente divers patrons, par exemple pour la simulation de la covariance (Chapitre 6) ou pour l'implémentation des associations UML (Chapitres 6 et 7).

Le terme 'patron' est aussi une bonne traduction de '***template***'.

pattern **:** voir **patron**

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

`private` **:** : le mot-clé pour **privé**.

privé : se dit d'un élément de programme qui n'est visible et accessible que depuis l'élément courant : le terme est ambigu puisqu'il peut désigner le receveur courant ou la classe courante, voire ses sous-classes.

Le caractère privé d'une entité peut restreindre son catactère objet : ainsi, (en C++) une méthode privée n'est plus virtuelle, et l'héritage privé ne supporte plus le sous-typage.

propriété : la troisième notion primitive, avec les classes et les objets. Les propriétés appartiennent aux objets mais sont factorisées dans les classes. Dans tous les langages, il y en a deux catégories, les attributs (données) et les méthodes (procédures). Certains rares langages y ajoutent des *types virtuels*.

Leur bonne compréhension nécessite de les modéliser avec des propriétés *globales* (ou *génériques*) et des propriétés *locales*.

Les différents langages utilisent des termes variés : *member* en C++, *features* en EIFFEL. Le terme de propriété est aussi utilisé dans d'autres sens, par exemple en C♯, pour désigner les **attributs** lorsqu'ils sont munis d'**accesseurs**.

`protected` **:** : un mot-clé batard pour désigner, en C++ ou en JAVA un intermédiaire entre privé et public. La propriété reste visible pour les sous-classes ou le *package* (en JAVA).

protocole : désigne une spécification, en général informelle, de l'aspect **dynamique** d'une partie d'un système sous la forme de l'enchaînement de créations d'instances et d'appels de méthodes. Un protocole correspond, en gros, à ce que propose UML sous les noms de diagramme de séquences, de collaborations ou d'interactions.

Les **patrons** de conception reposent généralement sur un diagramme de classes couplé à un protocole.

`public` **:** le mot-clé pour **publique**.

publique : se dit d'un élément de programme qui est visible et accessible depuis la totalité du programme.

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.

redéfinition : phénomène qui consiste à donner, dans une sous-classe, une définition à une propriété qui 'masque' la définition préexistante de la même propriété dans les super-classes et la remplace pour les instances de la sous-classe. A ne pas confondre avec la **surcharge statique**.

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) « consiste à transformer ou à transposer une abstraction en un objet concret, à appréhender un concept comme une chose concrète » (Wikipedia).

Le procédé est courant en informatique, en particulier avec la programmation par objets dont la méthodologie consiste à transformer des entités abstraites en « objets ». Le méta-modèle objet est ainsi la réification du modèle objet.

Le terme est utilisé dans de nombreux domaines (philosophie, sociologie, psychologie, littérature) à propos ou par des auteurs comme K. Marx, G. Lukács, E. Zola, etc.

schéma : Le terme de schéma est très général et un peu surchargé : on peut s'en servir pour parler d'un dessin, d'une définition schématique, d'un schéma d'algorithme ou de classe, voire d'un **patron**. On gardera ici le terme de **diagramme** pour les dessins, et on utilisera schéma pour des définitions schématiques, par exemple dans un langage de programmation donné. Le schéma d'une classe JAVA correspond à peu près à son interface, c'est-à-dire à la définition que l'on en ferait si c'était une interface JAVA. Le schéma d'une classe C++ correspondra à peu près à ce que contient son fichier `.h`.

sélection de méthode c'est une problématique en soi, liée à la redéfinition et à la surcharge statique, qui repose sur la métaphore de l'**envoi de message** ou **liaison tardive**. La sélection (en anglais *dispatch*) peut aussi être *multiple*.

sérialisation : désigne le procédé d'entrée-sortie par lequel un graphe d'objets clos, c'est-à-dire l'équivalent d'un diagramme d'instances, peut être écrit ou lu dans un système d'archivage quelconque (fichier, SGBD, etc.)

***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 : relation entre classes qui se définit comme l'inclusion des extensions et supporte l'héritage.

`static` **:** caractérise dans les classes ce qui n'est pas objet, même en JAVA, car c'est alloué une fois pour toutes (attributs) ou appelé de façon statique (méthodes) ! La classe n'a alors plus qu'un rôle d'espace de noms.

statique : par opposition à **dynamique**, désigne de façon générale ce qui se fait ou se détermine à froid, en l'absence de valeur, à la compilation.

Dans un contexte objet, le terme s'applique aussi aux classes (par opposition aux classes, qui sont dynamiques).

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 désigne, en général, le fait que le même nom puisse être utilisé dans des contextes de types différents pour désigner des entités différentes.

En programmation par objets, les entités sont généralement des propriétés qui sont distinguées, dans le contexte de la même classe, par les types statiques et le nombre des paramètres (pour les attributs il faut considérer le type statique du receveur).

***template* :** désigne les entités paramétrées de la généricité en C++ (Chapitre 7).

typage : désigne une approche ou un système de types. Il peut être *statique* ou *dynamique*, suivant que le langage comporte des annotations de type ou pas. Dans le premier cas, il peut aussi être *fort* dans le sens où la correction des types statiques sera vérifiée, et même *sûr*, c'est-à-dire qu'il exclut toute erreur de type à l'exécution ou bien, suivant une acception alternative, qu'il rattrape toute erreur de type.

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*, simple annotation dans les programmes qui sert de commentaire pour le compilateur et le programmeur, du type *dynamique*, type des valeurs à l'exécution, qui gouverne effectivement le comportement des programmes.

Un type peut aussi être *formel* ou *virtuel* (Chapitre 7).

variable de classe désignation ambigüe, 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` **:** c'est toute la différence entre C++ et la programmation par objets. C++ impose un mot-clé spécial pour désigner un comportement « objet » normal, c'est-à-dire quelque chose qui est **virtuel**.

Le mot-clé s'utilise pour les méthodes (C♯ souffre du même mal) et pour la déclaration des super-classes.

virtuel : désigne en général dans la littérature et dans les langages, par le mot-clé `virtual`, le fait qu'une entité dépende d'un objet **receveur** et soit désignée par **liaison tardive**. L'entité est alors **redéfinissable**. Cela peut désigner une fonction (**méthode**), l'héritage (dans ce cas, c'est la position d'un sous-objet qui est redéfinissable) ou même un **type** ou une **classe**. Le principe de base de la programmation par objets (Méta-axiome 5.1, page 71) peut se reformuler en : *tout est virtuel.*

On rencontre aussi quelquefois '*virtual*' dans le sens d'*abstract*, mais c'est un vrai faux-ami, puisqu'une méthode abstraite est forcément virtuelle, alors que l'inverse est évidemment faux.

visibilité : (ou mieux, **droits d'accès**) le fait que les propriétés d'un objet (d'une classe) ne soient pas accessibles depuis n'importe quelle autre classe. Normalement, la visibilité ne devrait pas avoir d'impact sur un programme, au sens où le même programme devrait se comporter de la même manière avec différents niveaux de visibilité, tant que le compilateur l'accepte. L'existence de la textbfsurcharge statique rend néanmoins cette propriété illusoire : en accroissant la visibilité on peut obtenir un programme qui se comptorte différemment, ou même qui ne compile plus.

Index

Table des matières

末尾