1

Objets, classes, héritage : définitions

LE BUT DE CE PREMIER CHAPITRE est de proposer un système de définitions pour la majeure partie du vocabulaire spécifique utilisé dans ce volume. Non pour imposer une norme, mais pour fixer un système de référence par rapport auquel chacun des contributeurs pourra marquer sa différence : on espère ainsi réduire les malentendus qui proviennent de l'emploi des mêmes mots, dans le même contexte, en des sens différents.

La technique de base qu'est le modèle à classes et instances est remarquablement simple. Dès lors, vu la popularité persistante de l'approche objet, on ne peut esquiver l'interrogation : « pourquoi ça marche ? ». Le principe de la réponse se trouve en confrontant les besoins ressentis par les utilisateurs et les moyens techniques censés les satisfaire.

Ce chapitre procède donc en trois temps. Dans un premier temps, sont présentés les deux termes du débat, à savoir les besoins à satisfaire et les conditions techniques de la solution. Dans un deuxième temps, sont définis la solution en question et le vocabulaire assorti, avec les restrictions nécessaires pour simplifier et clarifier les choses. Enfin, dans un troisième temps, la solution avancée est discutée.

1.1 Ouverture

1.1.1 Intentions

Idée générale

Pour qui cherche à décrire l'univers de l'*orienté objet* (pour reprendre un barbarisme familier) le premier caractère observable est l'emploi d'un certain vocabulaire : les mots *classe*, *instance*, *héritage* occupent une position centrale, alors qu'ils sont plutôt rares dans le discours informatique ordinaire ; d'autres comme *envoi de message* et *méthode* sont utilisés en des sens très particuliers ; d'autres enfin comme *type* ou *attribut* ont des acceptions qui recoupent celles de l'usage informatique habituel, mais sont affectés de nuances subtiles ; le cas du mot *objet* lui-même

est particulièrement délicat, car il est manifestement employé en un sens technique précis (mais lequel ?), qui reste compatible avec le sens d'*objet informatique* en usage courant. Malheureusement, si l'enquêteur cherche à en savoir davantage, un rapide coup d'œil à la littérature, depuis les origines jusqu'à aujourd'hui, le convaincra qu'en vingt ans les habitants de la planète des objets ne se sont toujours pas mis d'accord, ni sur le sens exact des termes qu'ils emploient, ni même sur les notions que ces termes sont censés désigner.

Le but de ce premier chapitre est de proposer un système de définitions pour la majeure partie du vocabulaire spécifique utilisé dans ce volume. Non pour imposer une norme, mais pour fixer un système de référence par rapport auquel chacun des contributeurs pourra marquer sa différence : on espère ainsi réduire les malentendus qui proviennent de l'emploi des mêmes mots, dans le même contexte, en des sens différents. Naturellement, pour expliquer la signification d'un terme, il est très utile d'indiquer aussi ce que cette signification ne recouvre pas. Nous proposerons donc une motivation et surtout une discussion critique de ce système de définitions. Ceci nous conduira à mettre l'approche objet en perspective, à discerner ses faiblesses et ses limitations. Au-delà des mots, l'on verra apparaître de nombreuses difficultés conceptuelles, certaines bien connues, d'autres soigneusement cachées. Ce texte prolonge et adapte un exposé antérieur plus détaillé sur certains points [Perrot, 1995], auquel on pourra se reporter le cas échéant.

Nous aurons plusieurs fois à invoquer Aristote et la tradition qui le suit. Pour cela nous utiliserons comme source le petit livre de la collection *Que sais-je ?* qui lui est consacré [Brun, 1988]. En effet, selon nous, l'approche objet reproduit fidèlement, dans le contexte informatique, une partie de la démarche logique et métaphysique du Stagirite, telle qu'elle nous a été transmise à travers les générations de penseurs qui ont forgé les outils intellectuels dont nous nous servons. Cette référence n'est pas innocente : elle manifeste que l'approche objet met en œuvre un système de pensée antérieur à la révolution opérée au XVII^{e} siècle par Galilée et Descartes. Analysant le passage de la physique d'Aristote à la physique moderne (au sens de l'histoire des sciences), Alexandre Koyré note que *la physique d'Aristote [...] s'accorde — bien mieux que celle de Galilée — avec le sens commun et l'expérience quotidienne [...] Elle se refuse à substituer une abstraction géométrique aux faits qualitativement déterminés de l'expérience et du sens commun* (Galilée et la révolution scientifique du XVII^{e} siècle, [Koyré, 1973, page 201]).

Mais l'approche objet ne vise pas la physique, direz-vous, et la critique de Koyré ne s'applique pas à elle. Ce point serait à examiner avec soin. Notons que dans le domaine de la classification des êtres vivants, la démarche d'Aristote révisée au siècle des Lumières a été bouleversée de nos jours par la cladistique, expression sans nuances du triomphe de l'évolutionnisme (voir la section 4 du *chapitre 15*). Il est clair que la démarche des naturalistes offre de nombreux points communs avec l'approche objet : les analystes et concepteurs par objets ont beaucoup à apprendre de l'expérience historique de leurs collègues biologistes systématiciens. La révolution cladistique devrait donc leur donner à réfléchir.

Le fond de la question, selon nous, est que l'approche objet permet de traduire en structures informatiques des opérations intellectuelles qui relèvent du *bon sens* (qui se dit en anglais, ne l'oublions pas, *common sense*). C'est là sa force. C'est

aussi peut-être la source cachée de nombreuses difficultés qui sont aussi celles de l'aristotélisme si, pour citer de nouveau Koyré [Koyré, 1973] le sens commun est — et a toujours été — médiéval et aristotélicien.

Précisions

Notre analyse sera, sans aucun doute, partielle et partiale. Elle sera centrée sur la *programmation par objets* proprement dite et sur les *langages à objets*, qui sont historiquement à l'origine d'une partie des développements dont il est question dans ce volume. Elle prendra en compte, au moins en partie, la problématique de deux domaines voisins, celui des *bases de données à objets* (traitée dans le *chapitre 5*) et celui de la *représentation de connaissances*, issue de l'intelligence artificielle (traitée en particulier dans les *chapitres 10, 11 et 12*). Mais elle laissera de côté les préoccupations du génie logiciel qui ont donné naissance à l'*analyse* et à la *conception par objets* (traitées dans le *chapitre 4*), les questions de parallélisme et de distribution liées aux *objets concurrents* (traitées dans les *chapitres 6 et 7*), ainsi que les *systèmes d'exploitation à objets*.

Enfin, nous nous intéresserons aux langages à objets tels qu'ils sont, tels que l'industrie du logiciel les emploie : SMALLTALK, C++, EIFFEL, JAVA. Il nous semble en effet que ce sont eux, à travers les succès qu'ils ont remportés, qui sont la source de la popularité de l'ensemble de l'approche objet, et que leur architecture commune constitue la référence primaire en la matière. Au grand regret de l'auteur, nous ne dirons rien du *Common Lisp Object System* (CLOS) et surtout nous laisserons de côté OBJECTIVE-CAML, qui sort de notre cadre tant par ses bases implémentatoires que par son ampleur conceptuelle (double système de modules et de classes). Sur CLOS, on dispose d'un excellent texte introductif [Habert, 1995] ; sur OBJECTIVE-CAML, qui n'est pas encore complètement stabilisé, on se reportera à l'URL `http://pauillac.inria.fr/ocaml/`.

Nous prendrons comme point de départ la technique d'implémentation. C'est là procéder au rebours de tout un mouvement de pensée, assez répandu, qui cherche à dégager des concepts et des formalismes supposés indépendants de l'implémentation. Nous estimons que l'approche objet, telle qu'elle se pratique, ne relève pas de ce mouvement, pour des raisons qui apparaîtront plus loin, et qu'à vouloir l'examiner sans référence aux vicissitudes opérationnelles, on est conduit à prendre ses désirs pour des réalités. Cette tentation est particulièrement forte en raison d'une ambiguïté fondamentale — la classe comme extension ou comme intension — que nous analyserons longuement. En fin de compte, c'est sans doute cette ambiguïté qui est la source de la fécondité et de la technique, mais il faut en être bien conscient, pour ne pas se faire prendre aux pièges du vocabulaire et pour ne pas se laisser désorienter par les débats interminables sur le sens des mots, qui surgissent dès qu'on pose la question : « qu'est-ce qu'un objet ? ».

Comme on verra, la technique de base, le modèle à classes et instances, est remarquablement simple — on serait même tenté de la qualifier de « rustique ». Dès lors, vu la popularité persistante de l'approche objet, on ne peut esquiver l'interrogation : « pourquoi ça marche ? ». Cette question rejoint en fait le problème du sens précis donné aux termes du vocabulaire technique, car en fait on parle peu de ce qui

ne marche pas. Le principe de la réponse, croyons-nous, se trouve en confrontant les besoins ressentis par les utilisateurs et les moyens techniques censés les satisfaire.

Nous procéderons en trois temps : les deux paragraphes suivants présenteront les deux termes du débat, à savoir les besoins à satisfaire et les conditions techniques de la solution. Dans un deuxième paragraphe, nous définirons d'un ton dogmatique la solution en question et le vocabulaire assorti, avec les restrictions nécessaires pour simplifier et clarifier (par exemple, supposer que tous nos objets résident en mémoire centrale). Dans la troisième, nous discuterons cette solution, comme annoncé plus haut.

1.1.2 Besoins ressentis par la communauté

L'émergence de l'approche objet se situe dans l'histoire de l'informatique à un moment où les progrès de la technique ont permis de réaliser des systèmes complexes, abordant des domaines d'application très variés, avec une exigence de sûreté renouvelée. Il est devenu indispensable de modéliser explicitement le problème traité, autrement que par un système d'équations, et de confier un rôle de plus en plus important au spécialiste du domaine. C'est dans cette perspective qu'il faut apprécier l'approche objet.

Elle répond clairement à trois préoccupations majeures que nous analysons ci-après, et qui rapprochent les structures informatiques des formes et des intuitions de la pensée ordinaire. Comme nous le verrons, une même technique permet d'apporter une solution approximative, mais efficace, aux trois problématiques à la fois.

Unicité, unité, individualité

Toute l'approche objet repose sur la conviction que les objets du monde réel offrent une base solide à notre intuition. Elle considère que les objets existent, qu'ils « sont bien là », n'en déplaise aux philosophes et à leur critique de la perception. Elle fait au besoin les restrictions nécessaires quant à son domaine de validité : s'il s'agit de manipuler un morceau de cire, on se gardera de l'approcher d'une flamme, pour échapper aux arguments de Descartes. En outre, elle ne les envisage pas en eux-mêmes, mais seulement dans une perspective d'utilisation, selon un certain projet : attitude pragmatique qui permet des simplifications salutaires. Moyennant ces limitations, il faut bien reconnaître que l'héritage culturel de l'*homo faber* nous donne une efficacité extrême dans la manipulation des objets matériels (le plus souvent vus comme des outils) : c'est cette efficacité qu'on souhaite transférer dans le domaine de la programmation.

Or, l'univers du programmeur est peuplé non pas d'objets réels, mais d'entités informatiques, qui ne sont que des amas d'octets en machine. Les objets du domaine d'application sont, comme on dit, représentés en machine. Dans les approches traditionnelles, dès qu'il s'agit d'objets complexes, cette représentation repose sur une analyse qui va faire éclater l'objet en de multiples facettes et finalement le dissoudre complètement pour se ramener à un système de variables à valeurs numériques : l'unité de l'objet, et sa signification, ne se réalisent que dans l'esprit du programmeur. Si l'on a affaire à une collection d'objets complexes, on aboutit avec les bases

de données relationnelles à un jeu de tables contenant des valeurs élémentaires, d'où tout individu complexe est banni.

Il en résulte pour le développeur de logiciels une charge intellectuelle qui devient incompatible avec la conquête des nouveaux domaines d'applications rendus accessibles par les progrès de la technique, comme nous l'avons dit plus haut : la programmation devient trop difficile, les systèmes réalisés sont fragiles, etc. Au contraire, le principe même de l'approche objet est d'établir une correspondance directe, bijective, entre les objets du monde réel et leurs ersatz informatiques, permettant au programmeur de s'exprimer conformément à son intuition. C'est ce que résume la deuxième règle d'or du *Manifeste des Bases de données à objets* [Atkinson *et al.*, 1989] par l'injonction *Thou shalt support object identity*[1]. C'est en apparence un retour au simple bon sens... à condition de savoir le faire !

Ce désir raisonnable de retrouver une unité perdue rencontre un autre courant d'idées, qui est issu de l'intelligence artificielle. Dans ce domaine, l'utilisation de structures complexes réalisées par des jeux de pointeurs et accessibles via une adresse unique est monnaie courante dès l'origine (n'oublions pas que l'intelligence artificielle à ses débuts parlait LISP). Ces structures sont vues comme des réceptacles de connaissances, de compétences, propres à prendre à leur compte certains choix et certaines opérations, et à en décharger *ipso facto* le programmeur. Le problème est alors l'organisation de ces connaissances. Du point de vue qui nous intéresse, deux tendances se dégagent : l'une s'oriente vers la représentation des connaissances incomplètes, avec l'idée des *frames* de Marvin Minsky [Minsky, 1975], dont nous reparlerons ; l'autre rêve d'objets véritablement autonomes, auxquels on s'adresserait en leur envoyant des messages polis : les acteurs du langage PLASMA de Carl Hewitt [Hewitt, 1977], qui, sous l'impulsion de Patrick Greussay, fut soigneusement étudié à Paris et à Toulouse dans les années 80.

Naturellement, c'est la technique d'implémentation qui met une borne aux phantasmes des utilisateurs : dès qu'on quitte l'environnement LISP, les problèmes de réalisation deviennent aigus. Les objets ne sont pas des acteurs — mais la programmation par objets a conservé l'idée de l'envoi de message, et avec elle l'intuition que l'objet est un individu respectable et compétent, auquel on s'adresse dans sa langue.

Abstraction : classes et types

L'attention portée à l'objet individuel « concret » donne une nouvelle coloration à la distinction fondamentale entre concept (abstrait) et instance (concrète) du concept — alias définition/exemple, espèce/individu, etc. Il s'agit d'un très ancien problème, celui du statut des concepts abstraits. Platon les appelait les idées, et il pensait qu'ils existaient réellement dans le monde comme entités séparées. Aristote, critiquant son maître, estime au contraire que les concepts généraux ne sont que des attributs, inséparables des individus auxquels ils s'appliquent [Brun, 1988, page 29]. Ce conflit se poursuit jusqu'à nos jours dans le débat entre nominalistes (qui suivent *grosso modo* Aristote) et réalistes (qui seraient du côté de Platon) avec, au Moyen-

1. Ton système doit gérer l'identité des objets.

Âge, la fameuse « Querelle des Universaux ». Comme nous le verrons, l'approche objet conduit à reposer la même question, dans un contexte renouvelé (voir § 1.3.3).

L'informatique suit une pente plutôt nominaliste. C'est la notion de type qui a occupé ce terrain, d'abord comme indications au compilateur sur l'organisation de la mémoire, ensuite comme annotations systématiques au texte du programme, donnant lieu au contrôle de types, enfin comme spécifications ou interfaces (types abstraits). Il faut souligner qu'un système de types est avant tout un langage qui doit permettre de décrire les entités manipulées par le programme d'une manière qui permette une vérification de cohérence *a priori*, sur le texte du programme (contrôle statique). Le malheur veut que l'exigence de contrôlabilité statique contraigne fortement ce langage, l'empêchant d'exprimer toutes les subtilités de la pensée du programmeur. Certains préfèrent donc ne pas s'encombrer de ces limitations, renoncer au contrôle statique et assumer dynamiquement leur liberté : la tradition SMALLTALK illustre brillamment les possibilités de l'option « non typée ». Toutefois, pour des raisons qu'il conviendrait d'analyser avec soin, la demande de sécurité paraît aujourd'hui majoritaire et, avec elle, la demande de langages fortement typés : EIFFEL est le représentant par excellence de cette orientation. La question des systèmes de types — au sens fort, pour contrôle statique — se pose donc pour les langages à objets. Or, elle se pose dans un contexte nouveau.

La promotion de la notion d'objet va en effet donner le branle à un double mouvement : d'une part, l'accent mis sur la présence d'individus en machine rend désirable leur catégorisation, selon un besoin irrépressible de l'esprit humain ; d'autre part, la technique d'implémentation va engendrer des catégories naturelles : les classes. Du coup, la pratique va s'engouffrer dans la programmation par classes et instances, avant de s'interroger sur le rapport entre les classes et les types.

En d'autres termes, il y a une pression exercée par les objets, ces « nouveaux individus », qui instaurent une nouvelle perception de la concrétude en machine et sont donc « demandeurs d'abstraction ». D'autre part, la « traction » engendrée par la technique implémentatoire des classes vient satisfaire ce besoin d'une manière qui diverge subtilement de la notion traditionnelle de type.

Et le problème des types ainsi renouvelé va se trouver enrichi par le besoin de classification hiérarchique qui représente, lui, une innovation en informatique. En effet, les classes entrent très naturellement dans une structure hiérarchique (via la relation d'héritage, voir le paragraphe suivant), ce que les types traditionnels ont bien du mal à faire. D'où une problématique nouvelle et féconde : d'une part, il faut savoir dans quelle mesure les classes peuvent entrer dans le système des types ; d'autre part, il faut absolument que le système des types accommode d'une manière ou d'une autre la relation d'héritage.

Hiérarchisation : héritage, sous-typage

Le besoin de classer se rencontre dans toutes les sociétés humaines : voir par exemple [Vogel, 1988, page 97] (au début du chapitre sur les taxinomies) Les concepts étant naturellement dotés d'une relation d'ordre, celle de leur plus ou moins grande généralité, nous ne pouvons pas nous empêcher de les comparer sous ce rapport. Un concept `B` est plus général qu'un concept `A` si toute entité qui, comme on

dit, « tombe sous le concept `A` » tombe aussi sous `B` : ce qui se réalise, par exemple, si nous prenons pour `B` le concept de mammifère et pour `A` celui de félin. Ce qu'on exprime en disant *tout* `A` *est un* `B`, forme brève de *toute instance de* `A` *est instance de* `B` : ainsi, *tout félin est un mammifère*. Et en empruntant le langage ensembliste : `A` *est inclus dans* `B`.

En contre-coup du « réveil » du problème de l'abstraction évoqué ci-dessus, nous voulons faire de même avec les images des concepts en machine, d'abord les classes, ensuite les types.

En ce qui concerne les classes, le mécanisme d'héritage vient opportunément satisfaire ce besoin. Il a remporté suffisamment de succès pour être retenu par de bons auteurs comme la caractéristique majeure de la programmation par objets. Dans un grand projet logiciel, l'organisation hiérarchisée du système des classes apparaît vite comme une structure indispensable. Mais, en vérité, le mécanisme d'héritage s'éloigne fort de l'intuition commune — la spécialisation comme inclusion des concepts — qui lui donne sa force attractive, ce qui cause de multiples difficultés.

En ce qui concerne les types, cette idée de hiérarchie n'était pas apparue auparavant en informatique — en tous cas pas au premier plan : la notion de sous-type est absente d'un langage aussi perfectionné que ML. Elle ne s'introduit pas sans mal, sous la pression de l'héritage des classes dont il faut absolument aujourd'hui rendre compte en termes de types. En fait, comme on le verra, la constatation majeure est que l'héritage ne peut pas se traduire simplement par le sous-typage. L'exigence de typage statique va donc entraîner une complication du modèle, avec deux structures pour l'abstraction, d'un côté les classes et leur héritage, de l'autre les types et leur sous-typage.

Du général au particulier

Nous venons de distinguer très nettement deux mécanismes, celui d'abstraction qui oppose un objet (concret) à son concept (abstrait), et celui de hiérarchisation des concepts suivant leur degré de généralité. Dans nos langues « naturelles », ces deux mécanismes ne se distinguent pas : nous affirmons qu'*un chien est un mammifère* de la même manière que nous disons que *Médor est un chien*. La distinction stricte entre classes et instances nous impose de traduire ces deux énoncés par deux mécanismes différents, le premier par un lien de spécialisation entre les classes *chien* et *mammifère*, le second par un lien d'instanciation : l'individu *Médor* est instance de la classe *chien*. Devoir séparer dans la programmation des énoncés que la langue naturelle confond est une contrainte qui peut être jugée salutaire ou insupportable suivant les buts poursuivis ; l'approche objet l'accepte. On sait que d'autres démarches ont cherché à conserver l'ambiguïté de l'expression « X est un Y » : notamment celle des *frames*, avec le lien *is-a* (voir [Masini *et al.*, 1989, section 8.3.1], mais aussi le *chapitre 8*).

En fait, ces deux mécanismes peuvent être vus comme la manière dont l'approche objet aborde un des problèmes de fond des langages de programmation : le passage du général au particulier. Plus précisément, il s'agit de trouver le moyen d'exprimer des informations à caractère général et de les mettre en œuvre dans un cas particulier sans avoir à les reformuler. De ce point de vue, une classe est un

énoncé général, dont les instances sont autant de particularisations ; de même, le lien d'héritage permet de passer d'une classe plus générale à une autre plus particulière.

On connaît un autre mécanisme qui répond au même besoin de particulariser des énoncés généraux, la *généricité paramétrique*, connue également sous le nom de *polymorphisme* en théorie des types : si je sais faire la théorie générale des piles dont les éléments sont de type `X`, je peux m'en servir automatiquement pour faire fonctionner une pile d'entiers. Ce mécanisme est en fait la solution au problème de la particularisation que propose l'approche par types abstraits. À son égard, l'approche objet n'a rien de spécifique à dire : l'idée de généricité s'applique banalement à la notion de classe, donnant naissance aux *classes génériques* d'EIFFEL (voir *chapitre 2*) et aux *patrons* (alias *templates*) de C++ (voir la section 6 de *chapitre 3*). D'autres langages préfèrent y renoncer (SMALLTALK, JAVA). Avec un peu d'habileté, on peut proclamer que le mécanisme d'héritage permet de se passer de généricité (voir [Meyer, 1986]). Mais il est certain que pour les langages qui s'offrent le luxe d'incorporer la généricité paramétrique, la présence de l'héritage et l'exigence de sous-typage viennent sérieusement compliquer l'implémentation. On verra plus loin un échantillon des difficultés rencontrées (§ 1.3.2).

1.1.3 Présupposés sur la technique d'implémentation

Les choix d'implémentation que nous allons expliciter, et qui sous-tendent la quasi-totalité des réalisations, sont fort difficiles à justifier si l'on ne tient pas compte des moyens informatiques supposés disponibles. Pour décrire d'une manière adéquate les mécanismes de base des langages à objets, nous nous situons dans un univers très primitif, *grosso modo* celui de la programmation en assembleur, ou celui du *bytecode* d'une machine abstraite — pas l'univers LISP, encore moins celui de ML. Nous souhaitons ainsi exposer les choses comme elles sont, non comme elles devraient être.

L'outillage de base

La première hypothèse est que les seules entités à notre disposition sont de deux sortes, les données et les procédures. Plus précisément, nous nous plaçons dans une architecture mono-processeur, avec une seule pile, et nous ne comptons pas sur le pseudo-parallélisme des processus, ni même des coroutines. Cela exclut notamment des êtres qui seraient extrêmement utiles dans de très nombreuses situations comme les contraintes, mais dont la réalisation informatique offre des difficultés supérieures d'au moins un ordre de grandeur à celle des entités de base que sont les données et les procédures.

Le tas

En revanche, nous supposons acquis le partage de la mémoire de travail en trois zones, la zone code, la pile et le tas. Le tas est administré par un gérant capable d'allouer des blocs de taille variable et, le cas échéant, de les récupérer grâce à un

mécanisme de *ramasse-miettes* (*garbage-collecting*). Nous considérons exclusivement des données structurées hétérogènes, ou enregistrements (*records*), repérées par leur adresse dans le tas (alias pointeur), chaque champ composant étant connu par un déplacement (*offset*) à partir de cette adresse. Le rôle de la pile se limite au traitement de la récursion.

Les procédures ne sont pas des données

Enfin, point essentiel, nous admettons que données et procédures sont des êtres de natures différentes, qui ne sont pas représentés de la même manière et qui ne suivent pas les mêmes lois. C'est la situation commune, à laquelle n'échappent que les langages à fermetures (COMMON LISP, SCHEME, ML), qui sont bien au-dessus de notre modeste atelier.

Nous voyons les procédures comme des textes compilés sous forme exécutable, où tous les noms sont résolus en adresses et en déplacements. Ces textes sont repérés par leur adresse dans la zone code. Ceci signifie que nous renonçons à effectuer des calculs sur les procédures (ce qui demanderait de les considérer comme des données ordinaires), en particulier que nous n'envisageons pas des procédures qui seraient le résultat d'un calcul de notre programme. Nous ne nous autorisons donc à leur égard que trois opérations : la rédaction (écriture dans un fichier), la compilation (ou mise sous forme interne) et l'exécution.

Comme nous allons le voir, on peut proposer, dans ce cadre, une solution simple et efficace aux besoins évoqués au paragraphe 1.1.2 ci-dessus. En revanche, il sera fort malcommode de faire entrer la solution en question (et la pratique qui la met en œuvre) dans une spécification *a priori*.

1.2 Définitions

Comme d'ordinaire en informatique, les objets n'ont de sens qu'en raison de leur fonctionnement. Il nous faut donc introduire d'un même coup plusieurs termes qui se définissent l'un par rapport à l'autre.

1.2.1 Objets

Il y a trois aspects dans la définition d'un objet, qui concourent tous trois à forger son individualité.

Adresse

Un objet est une entité individuelle, repérée par une adresse unique. Pour simplifier, nous ferons l'hypothèse que cette adresse repère une zone mémoire située dans le tas, qui sera soumise au ramasse-miettes. Clairement, cette restriction est insupportable du point de vue des bases de données ! Nous estimons cependant qu'elle a l'avantage de livrer un principe d'implémentation qui peut se décliner en de nombreuses variantes pour raisons d'efficacité, alors que le problème des objets permanents n'admet pas de solution simple. Nous la proposons donc comme référence.

Le besoin d'unité/unicité étant ainsi satisfait d'entrée de jeu, la question est désormais : que trouve-t-on au juste à cette adresse et comment s'en sert-on ?

État

Comme un système physique, un objet possède un état (interne) qui peut varier au cours du temps. C'est donc une entité *mutable* au sens de ML. Nous discuterons ce point plus loin (voir § 1.3.1).

Cet état se réalise de la manière la plus simple, sous la forme d'un enregistrement de PASCAL (`record`) ou d'une structure de C (`struct`), formé de plusieurs *champs* (en nombre fixé), connus par leurs noms. Matériellement, les champs sont repérés par leur déplacement à partir de l'adresse de l'objet prise comme adresse de base.

Chaque champ contient une valeur (en général l'adresse d'un autre objet), qui peut varier au cours du temps. L'ensemble des valeurs des champs constitue à chaque instant l'*état courant* de l'objet. Dans l'analogie avec un système physique, les noms des champs s'identifient aux variables d'état du système : on voit (sans surprise) que nous nous limitons à des systèmes à un nombre fini de degrés de liberté.

Nos champs correspondent aux *slots* de la tradition des *frames* (voir *chapitre 10*). Mais nous ne les analysons pas en *facettes* : nous considérons « bêtement » qu'ils contiennent des valeurs, sans nous demander si ces valeurs sont effectivement valides, etc. Tout au plus attacherons-nous à chaque champ l'indication du type de sa valeur.

Dès qu'on s'éloigne de la matérialité de la réalisation en machine, on préfère parler des *attributs* d'un objet plutôt que des champs d'un enregistrement. Dans notre usage, les deux termes sont synonymes. Les différents langages classiques emploient pour dire la même chose une terminologie riche et diversifiée : en SMALLTALK, on dit *variables d'instance* (et le cas échéant on précise qu'on parle de leur nom) ; en C++, ce sont les *données membres* (*data members*, voir *chapitre 3*) ; en EIFFEL, des *primitives* (*features*, voir *chapitre 2*).

Rappelons qu'il faut distinguer soigneusement le nom de l'attribut (que le compilateur convertit en un déplacement) de la valeur qu'il contient. À un instant donné, donc, l'ensemble (ou plutôt le vecteur) des valeurs des attributs constitue l'état de l'objet.

Mais comment se manifeste cet état ? Et comment s'en sert-on ?

Comportement

L'objet réagit à des sollicitations extérieures. Il est donc doué d'un comportement. Nous reviendrons plus loin sur ce point essentiel. C'est en fait cette capacité réactive qui va donner à ses utilisateurs l'illusion d'une existence concrète ; elle distingue les objets des « structures de données » traditionnelles (voir discussion ci-après, § 1.3.1).

La modélisation du comportement est le point délicat de notre système de définitions. C'est que le comportement est dynamique, et il ne pourra être obtenu, en

dernière analyse, qu'en exécutant du code. Quel code, et comment l'activer ? *Hic jacet lepus*[2] !

À titre de comparaison, rappelons que dans les années 1970, les acteurs de Carl Hewitt avaient leur comportement modélisé comme un script unique activé par filtrage de messages [Hewitt, 1977]. L'approche objet revient à une réalisation plus grossière, mais en pratique fort efficace.

Chaque objet n'a qu'un nombre fini de procédures à sa disposition, chacune portant un nom. Le comportement de l'objet consiste à exécuter l'une de ces procédures, en réponse à un « message » très simplifié, qui ne contient que le nom de la procédure et les arguments de l'appel. Bien entendu, l'effet de cette exécution dépend de l'état courant de l'objet et il peut modifier cet état.

Par rapport à la programmation procédurale classique, le changement vient de ce que le message mentionne le nom de la procédure à exécuter, mais que le choix de la procédure elle-même (du corps de procédure) est effectué par l'objet destinataire du message : deux objets différents peuvent associer à un même nom des procédures différentes (on parle à ce sujet de surcharge ou de polymorphisme). C'est pourquoi nous continuons à utiliser le mot « message », plutôt qu'appel indirect de procédure : on met ainsi en valeur l'autonomie de l'objet, pivot essentiel de toute notre approche.

Soulignons que dans cette définition les procédures activées par les messages, qui définissent le comportement de l'objet, ne sont pas traitées comme valeurs de champs de l'enregistrement, puisque, comme nous l'avons rappelé, nous ne considérons pas que des procédures puissent être traitées comme des valeurs. Un objet apparaît donc comme constitué de deux parties, d'une part l'enregistrement qui définit son état, d'autre part la table de correspondance noms – corps de procédures qui définit son comportement.

Du point de vue de la représentation des connaissances, l'inconvénient de cette solution est qu'il est difficile (pour une machine) de raisonner sur des procédures : par suite, il est difficile de raisonner sur le comportement d'un objet ; et comme, en outre, la structure interne de l'objet (le système des champs) va être traitée comme confidentielle (privée) la plupart du temps, il faut reconnaître que nos objets se prêtent mal au raisonnement informatisé. C'est pourquoi la représentation des connaissances préfère utiliser des *frames*, où le comportement des objets est obtenu par des « attachements procéduraux » : il s'agit bien, en dernière analyse, d'exécuter des procédures, mais dans ce cadre, elles sont appelées à travers des mécanismes de *réflexes* ou *démons*, qui sont censés déclarer une partie de leur sémantique et ainsi donner prise au raisonnement (voir *chapitre 10* et [Masini *et al.*, 1989, section 8.3.5]).

Nous avons insisté sur la nature procédurale du comportement de nos objets, mais dans le langage courant on préfère l'oublier. On parle alors de *méthodes*, dans le vocabulaire de SMALLTALK, le nom de la méthode étant appelé *sélecteur* (à ce sujet, on notera que les conventions très particulières de la syntaxe de SMALLTALK font que la forme du sélecteur dénote l'arité de la méthode, à l'opposé des possibilités de surcharge qu'offrent C++ et JAVA). EIFFEL pour sa part préfère parler de

2. Ou : *c'est là que les Athéniens s'atteignirent ...!*

routines (*chapitre 2*), C++ de *fonctions membres* (*member functions*, voir *chapitre 3*).

1.2.2 Classes

Il reste à expliquer comment les objets que nous avons définis ci-dessus peuvent être créés en machine. Nous y arriverons en introduisant une nouvelle entité, la classe, qui va aussitôt se trouver chargée de représenter les concepts abstraits, bien qu'elle n'ait pas été définie dans ce but.

Le problème central est celui de la genèse du comportement des objets. C'est ici que les hypothèses que nous avons faites plus haut sur la nature des procédures pèsent de tout leur poids. Puisque dans notre système une procédure ne peut pas être le résultat d'un calcul, elle doit provenir directement de la compilation de son texte. Ceci vaut en particulier pour les procédures qui définissent le comportement d'un objet. Cette circonstance donne la primauté à la représentation textuelle et conduit tout droit à la notion de classe. Si le comportement pouvait être obtenu par un véritable calcul, la classe perdrait peut-être de son importance. Mais il faut se rappeler que, même en programmation fonctionnelle, le calcul des procédures se réduit à fixer les valeurs des variables libres qui apparaissent dans leur texte. Les classes, même si l'on cherche à les cacher comme dans les langages à prototypes (voir *chapitre 8*), ne sont donc jamais bien loin.

Texte

Une *classe* est un texte structuré qui rassemble, d'une part, une liste de noms de champs, et, d'autre part, un *dictionnaire* ou *catalogue de procédures*, appelé encore *catalogue de méthodes*, (noms et textes). Les noms de champs apparaissent comme variables libres dans les textes des procédures. Ce texte apparaît sous deux formes, soit comme texte source dans un fichier, soit comme texte compilé en mémoire.

Une *instance* d'une classe est un objet constitué des champs dont la liste est donnée par la classe et dont le comportement est défini par l'interprétation des messages, comme expliqué ci-dessous.

Cet objet possède un champ supplémentaire qui contient l'adresse du code compilé de sa classe.

Interprétation des messages

Son principe est le suivant :

1. le message fournit directement le nom de la procédure à exécuter et le corps de procédure correspondante est recherché dans le catalogue de procédures de la classe ;
2. ce corps est ensuite exécuté avec la convention que chaque nom de champ qui y figure sera interprété comme désignant le champ correspondant de l'instance qui traite le message (par exemple, en interprétant les noms de champs comme des déplacements à partir de l'adresse de l'objet prise comme adresse de base).

On peut résumer le mode d'exécution du corps de procédure en disant que l'exécution a lieu « dans le contexte local de l'objet ». En effet, l'état courant de l'objet, à savoir le système des couples « (nom de champ,valeur du champ) », peut être vu comme un contexte d'exécution fixant les valeurs des variables libres du corps de procédure. Deux objets différents, instances de la même classe, définissent des contextes isomorphes mais différents, dans lesquels l'exécution d'un même corps de procédure peut donner des résultats différents : c'est ainsi que l'exécution du corps de procédure dépend de l'état de l'objet destinataire du message ; c'est ainsi également que cette exécution peut modifier l'état en question.

Si nous revenons aux considérations du paragraphe 1.2.1 sur l'autonomie de l'objet dans le choix de la procédure à exécuter en réponse à un message, nous voyons que deux objets qui sont instances de la même classe feront le même choix : pour que le même message adressé à deux objets différents provoque un comportement différent, il faudra que les catalogues des méthodes des deux classes associent au même nom (sélecteur) des procédures différentes.

Instanciation

La création d'une instance à partir de sa classe se décompose en deux phases :

1. l'allocation d'une zone-mémoire de taille $k + 1$, où k est le nombre des champs, le premier mot recevant l'adresse de la classe, suivie de

2. l'affectation à chaque champ d'une valeur initiale.

La première phase est confiée au gestionnaire de mémoire ; elle échappe au contrôle du programmeur. La seconde, en revanche, fait l'objet de procédures d'initialisation qui doivent être explicitement rédigées (à moins qu'on ne dispose d'initialisations par défaut grâce à une valeur universelle `nil`). Le statut exact de ces procédures d'initialisation diffère selon les langages : elles sont souvent désignées par le nom trompeur de « constructeurs » (tradition de C++ malheureusement reprise par JAVA).

Avec cette mécanique extrêmement simple, nous avons le principe d'une réalisation complète de notre idée d'objet. Il est difficile de faire mieux, et même de faire autrement : on finit toujours par voir réapparaître la notion de classe sous des déguisements divers.

Deux remarques :

- on ne peut créer d'instance que si la classe est présente en mémoire, et l'instance ne peut fonctionner que tant que la classe reste présente,

- un objet ne peut être instance que d'une seule classe, et, en principe, il ne change pas de classe au cours de son existence (sauf manœuvre exorbitante comme la primitive `become:` de SMALLTALK ; le recours à la réflexivité — voir § 1.3.3 — permet d'aller plus loin [Rivard, 1997]).

Les classes comme représentation des concepts

À l'intérieur de la machine, classes et instances ont la même consistance matérielle, à savoir des octets en mémoire. Les unes ne sont pas plus abstraites ou concrètes que les autres. C'est l'interprétation que nous faisons de l'implémentation esquissée ci-dessus qui assigne aux classes un statut d'abstractions (concepts) et aux autres un statut d'objets concrets (instances). Cette interprétation s'appuie sur une analogie : de même que le concept abstrait est unique vis-à-vis de ses réalisations concrètes, qui sont multiples, de même la classe est une, et ses instances sont (potentiellement) multiples. En outre, comme nous l'avons dit, les instances (objets informatiques) ont pour vocation de représenter des objets concrets du monde réel, elles occupent donc le pôle « concret » de l'opposition abstrait/concret, vouant les classes au pôle « abstrait ».

Or, comme nous l'avons observé, les programmeurs ressentent un besoin d'abstraction aigu. Du coup, nos classes, bien qu'elles aient été introduites comme une technique de réalisation pour notre projet d'objets, vont devoir jouer le rôle de représentation des concepts. Bien évidemment, elles présenteront à cet égard des insuffisances nombreuses. Mais il est honnête de prendre conscience de la difficulté de cette nouvelle tâche ! Tout un courant de recherches en intelligence artificielle lui est consacré, commençant par les réseaux sémantiques (dès 1968, voir *chapitre 10*), passant par KL-ONE [Brachman et Schmolze, 1985] et aboutissant aujourd'hui aux logiques de descriptions (voir *chapitre 11*).

Il s'ensuit un glissement sémantique de la plus haute importance : étant donné un concept, on lui associe naturellement l'ensemble de ses instances, que l'on appelle l'*extension* du concept. Pris en lui-même, un concept (par exemple le concept de chat) est certes autre chose que son extension (qui, elle, est un ensemble : l'ensemble des chats). Mais très souvent le nom du concept sert à désigner son extension, et, en somme, le concept est vu comme représentant son extension (quand on dit « le chat est un animal domestique »).

Dans la mesure où elles représentent des concepts, nos classes se voient donc chargées d'une valeur sémantique ensembliste en plus de leur signification littérale, qui prescrit comment leurs instances sont faites et comment elles se comportent : chacune représente l'ensemble de ses instances, effectivement présentes à un moment donné dans le système ou seulement potentielles.

Soulignons que ce glissement sémantique se produit de manière inconsciente, comme dans de nombreux phénomènes langagiers, car notre pensée passe naturellement de la description à la chose décrite. Il faut donc faire un effort de réflexion pour s'en rendre compte. Il en résulte pour la notion de classe une ambiguïté essentielle, qui est précisément la source de sa fécondité. Cette ambiguïté est d'ailleurs corroborée par la polysémie du mot classe. Dans plusieurs branches du savoir plus anciennes que l'informatique, une classe désigne une espèce particulière d'ensemble : ainsi en mathématiques (classes d'équivalence), dans les sciences de la nature (la classe des mammifères, voir *chapitre 15*), en sociologie (les classes sociales), etc.

Nos classes viennent ainsi s'ajouter à la panoplie dont disposent les informaticiens pour représenter des ensembles. On sait de reste qu'en informatique la notion d'ensemble pose problème. Pour représenter un ensemble infini en termes finis ac-

ceptables par une machine, il faut recourir à un programme descriptif, qui définit de manière algorithmique tous les éléments de l'ensemble (sous sa forme la plus simple, ce programme va expliciter les sous-entendus cachés sous les points de suspension qui apparaissent si souvent dans les notations mathématiques). Ce programme est appelé une *intension*, mot emprunté au vocabulaire de la philosophie [3]. Et les classes de la programmation par objets sont bien des programmes, descriptions intensionnelles, les seules choses que nous sachions réaliser avec nos drôles de machines...

Réflexion *a posteriori*

Pour construire un nouvel objet, il faut indiquer à l'ordinateur comment il se compose, et pour ce faire, en attendant les interfaces de programmation graphique, il faut rédiger à l'usage de la machine un texte décrivant le procédé constructif. Dans ce texte apparaîtront les noms des objets composants, avec le statut de variables libres. Et, du coup, notre texte ne décrira plus un objet singulier, construit à partir de composants eux-mêmes singuliers, mais toute la famille des objets qui peuvent être produits par le même procédé en donnant aux variables libres d'autres valeurs : il devient bel et bien une classe.

Cette incapacité d'un texte à décrire une entité singulière sans automatiquement décrire une famille d'entités isomorphes est bien connue de la tradition philosophique : Aristote notait déjà qu'il n'y a pas de définition de l'individu [Brun, 1988, page 36]. On peut la rapprocher de la règle de généralisation en logique du premier ordre : si l'on a $\mathtt{P}(\mathtt{x})$, où `P` est un prédicat et `x` une variable libre, alors on a universellement $\forall \mathtt{x}\mathtt{P}(\mathtt{x})$.

Ceci explique les difficultés infinies que rencontrent les tentatives de réaliser en informatique une approche basée sur des prototypes, qui seraient des instances concrètes à partir desquelles d'autres instances concrètes se définiraient par « recopie différentielle » (voir *chapitre 8*) : on y voit toujours reparaître des objets constructeurs qui, en fait, sont très analogues aux classes. Ce phénomène ne perd de son importance qu'en présence d'une grande quantité d'objets déjà existants, dont on peut oublier l'histoire, et qui fournissent la base d'un « concret préalable ». Mais cette situation n'est pas encore courante.

1.2.3 Héritage

La dualité entre intension et extension apparaît au grand jour avec la notion d'héritage. Bien entendu, les langages classiques ont pour en parler chacun leur vocabulaire et leurs mots-clés : EIFFEL et SMALLTALK parlent d'héritage, C++ de *dérivation* et JAVA d'*extension* (avec le mot-clé `extends`).

Définitions : ajouter des attributs et des méthodes

1. On dit qu'une classe `A` *hérite* d'une classe `B` si tous les attributs définis par `B` se retrouvent dans `A`, c'est-à-dire si la liste des noms de champs de `B` est

3. Terme sorti de l'usage courant, synonyme de *compréhension*, dit le dictionnaire Lalande [Lalande, 1926].

incluse dans celle de `A` et si le catalogue des méthodes de `B` est contenu dans celui de `A`.

2. Lorsque `A` hérite de `B`, on dit aussi que `A` est une *sous-classe* de `B` et que `B` est une *super-classe* de `A`.

3. La relation d'héritage est une relation d'ordre, qui se représente par un graphe sans circuit appelé le *graphe d'héritage*.

En ce qui concerne l'écriture des textes, en pratique, on définira une sous-classe `A` à partir de la super-classe `B` supposée déjà connue, en donnant seulement les noms de champs et les procédures supplémentaires « propres à `A` », sans reproduire le texte de `B`, accessible via un pointeur *ad hoc*. Matériellement, une sous-classe se présente donc comme un « fragment de classe », incomplet du point de vue sémantique, et qu'il faut compléter par sa super-classe.

Les termes de sous-classe et de super-classe demandent une justification. En effet, du point de vue intensionnel, le texte de la sous-classe contient celui de la super-classe ! En quoi est-elle donc *sous*-classe ? Parce que son extension est un *sous-ensemble* de l'extension de sa super-classe. Mais cette inclusion ensembliste ne va pas de soi ! Elle repose sur l'interprétation que nous donnons au fonctionnement du système. Elle se traduit au niveau des types par l'apparition de la relation de sous-typage. Voyons en quoi consiste cette interprétation.

A priori, un objet appartient à (l'extension d') une classe si et seulement si il a été créé à partir de cette classe. Les extensions de deux classes différentes sont donc tout simplement disjointes, et toute hiérarchie est exclue. Il faut par conséquent adopter une démarche plus raffinée que cette vision simpliste. Il serait désirable de pouvoir définir l'extension d'une classe comme l'ensemble des objets qui possèdent tous les champs et le comportement définis (intensionnellement) par cette classe. Les instances d'une sous-classe, qui possèdent tout cela avec quelque chose de plus, feraient ainsi de plein droit partie de cette extension, et nous aurions l'inclusion souhaitée. On pourrait épiloguer sur le bien-fondé de cette définition par condition nécessaire et suffisante portant sur la structure, mais nous admettrons ici qu'elle est, dans le présent contexte, « raisonnable ».

Malheureusement, elle est trop difficile à implémenter pour nos faibles moyens. Chaque instance d'une classe possède évidemment tous les champs et le comportement en question. Mais pour la réciproque, rien, dans les mécanismes que nous avons décrits, ne permet d'analyser la structure d'un objet. La seule information que nous sachions exploiter est la « marque de fabrique » de la classe, sous la forme de son adresse (ou de son nom) qui fait partie de l'objet. En fait, on peut très bien écrire deux fois la « même » classe sous deux noms différents, et le système les traitera comme deux classes totalement différentes, aux extensions disjointes. Ce problème est abordé de front, avec des moyens plus puissants, par les logiques de descriptions dans la théorie de la subsomption (voir *chapitres 10, 11 et 12*).

Nous revenons à une définition aisément implémentable en disant qu'un objet appartient à l'extension d'une classe si et seulement s'il a été créé à partir de cette classe *ou d'une de ses sous-classes*. L'inclusion cherchée est ainsi assurée par

définition. Quant à l'implémentation, il suffit de prévoir dans la structure des classes un champ supplémentaire contenant l'adresse de la super-classe pour pouvoir répondre à la question.

Interprétation des messages : la recherche ascendante *look-up*

Pour que toute instance d'une sous-classe puisse être considérée comme une instance de la super-classe, le mécanisme d'interprétation des messages doit être complété :

- en ce qui concerne les champs supplémentaires introduits dans la sous-classe, ils sont simplement ajoutés à la fin de la liste donnée par la super-classe ;
- en ce qui concerne les messages :
  - si le sélecteur — nom de la méthode — apparaît dans le catalogue de la sous-classe, le corps de procédure correspondant est exécuté comme expliqué dans le paragraphe précédent (naturellement, seules les méthodes définies dans la sous-classe sont habilitées à utiliser les champs supplémentaires) ;
  - sinon, la recherche se poursuit dans la super-classe : si le nom y figure, le corps de procédure associé est exécuté, sinon la recherche se poursuit dans la super-classe de la super-classe (si elle existe), etc., jusqu'à arriver à une classe qui n'a pas elle-même de super-classe, ce qui déclenche une erreur à l'exécution.

Ce processus de recherche en remontant dans le graphe d'héritage (en anglais *look-up*) est très facile à implémenter sous l'hypothèse que l'héritage est simple : l'unicité de la super-classe se traduit par la structure du graphe d'héritage en forêt, ou ensemble d'arbres. En fait, on considère la plupart du temps que le graphe d'héritage possède une racine unique, qui correspond à une super-classe générale exprimant les propriétés communes à tous les objets du système. Le graphe d'héritage est alors connexe, et lorsque l'héritage est simple, il s'agit d'un arbre.

Si l'héritage est multiple, il faut définir un parcours ascendant du graphe d'héritage, ce qui peut donner lieu à des algorithmes savants [Ducournau et Habib, 1989] [Ducournau *et al.*, 1995]. En outre, il faut recourir à un système un peu plus compliqué qu'une simple numérotation pour assurer la correspondance entre les noms des champs et les déplacements en mémoire.

Redéfinitions

L'interprétation extensionnelle de l'héritage est légitime si l'on se borne à ajouter dans la sous-classe des attributs et des méthodes qui portent des noms ne figurant pas dans la super-classe : le procédé d'interprétation ci-dessus fonctionne alors sans ambiguïté, et l'on peut affirmer qu'une instance de la sous-classe se comporte en tout point comme une instance de la super-classe (sauf en ce qui concerne les erreurs à l'exécution) et que par conséquent, il est licite de la considérer comme étant elle aussi une instance de ladite super-classe. Plus précisément, on peut dire que tout le

comportement significatif (c'est-à-dire non erroné) des instances de la super-classe est reproduit sans changement par celles de la sous-classe, qui ainsi « se qualifient » pour appartenir à la super-classe.

Les difficultés commencent si l'on veut introduire des homonymes : on parle alors de redéfinition ou de masquage de l'attribut ou de la méthode déjà connu sous le même nom. Il s'agit en fait d'une idée fort naturelle, très fréquemment employée dans la pratique. Il n'est malheureusement pas facile d'en rendre compte sur un plan théorique.

La redéfinition d'attributs est possible dans certains langages typés. Le plus souvent, le compilateur impose une certaine cohérence entre les deux définitions, à savoir que le type de l'attribut redéfini (dans la sous-classe) soit compatible avec le type déclaré dans la super-classe. Pour un langage non typé (ou « typé dynamiquement ») comme SMALLTALK, la redéfinition d'attributs n'a pas de sens évident (il faut se méfier de l'ingéniosité des programmeurs SMALLTALK, ils sont fort capables d'en trouver un !). Aussi les différentes versions du langage ont-elles des attitudes qui vont de la tolérance sans commentaire à l'interdiction brutale.

Mais c'est la redéfinition de méthodes qui est universellement employée et c'est elle qui pose de graves problèmes. On peut immédiatement étendre le procédé de recherche ascendante (*look-up*) au cas où l'on donne dans la sous-classe une nouvelle définition à un nom de procédure déjà défini dans la super-classe : en général, on souhaite associer à ce nom un comportement légèrement différent de celui que spécifie la super-classe pour l'adapter au perfectionnement introduit par la sous-classe, par exemple pour tenir compte d'un attribut supplémentaire.

La difficulté saute aux yeux : en présence de redéfinition, le comportement significatif des instances de la sous-classe va être différent de celui des instances de la super-classe, et, en bonne logique, l'on ne pourra donc plus conclure comme précédemment que l'extension de la sous-classe est incluse dans celle de la super-classe. Pour maintenir cette conclusion, il faudrait étendre la notion de « différence significative de comportement » pour pouvoir dire que les comportements des deux classes ne diffèrent pas significativement ... Pas très commode ! On devine les difficultés qui nous attendent quand nous voudrons confier cette tâche à un contrôleur de types (voir plus loin, en 1.3.2 les questions de covariance et de contravariance).

Sous-typage et lien dynamique

Du point de vue du compilateur, la recherche ascendante (*look-up*) est source d'inefficacité : le compilateur travaille non sur les objets eux-mêmes, comme l'interprète que nous avons esquissé ci-dessus, mais sur des expressions où les objets apparaissent comme valeurs de variables, le compilateur lui-même n'ayant connaissance que de la classe des variables, supposée déclarée par le programmeur. En l'absence d'héritage, la connaissance de la classe suffit au compilateur pour déterminer « statiquement » la procédure à appliquer en réponse à un message.

Mais si la classe déclarée (disons `B`) possède des sous-classes (disons `A`), alors une variable déclarée de classe `B` peut très bien contenir en fait, à un moment ou à un autre du calcul, une instance de la sous-classe `A` : puisque toute instance de `A` est aussi instance de `B` ! Attention ! C'est ici que l'interprétation extensionnelle de

l'héritage se transforme en exigence de sous-typage : dire que toute instance de `A` est aussi instance de `B` entraîne que partout où une instance de `B` peut légitimement apparaître, on peut lui substituer une instance de `A` sans changer le comportement du programme. Dans le langage de la théorie des types, cette possibilité de substitution signifie que, en tant que type, la sous-classe `A` est sous-type de `B` : c'est la définition même du sous-typage. On verra plus loin les redoutables conséquences de cette exigence, laissons de côté pour l'instant le problème du contrôle de types.

Dès lors, pour peu que la méthode visée par le message à compiler ait été redéfinie dans la sous-classe `A`, le compilateur doit activer soit la procédure déclarée dans la super-classe `B`, soit celle qui provient de la redéfinition dans la sous-classe. Et ce n'est que dynamiquement, à l'exécution, que cette incertitude sera levée. Ce problème est connu sous le nom de *liaison dynamique*, notamment dans la tradition de C++ (voir *chapitre 3*). Il a fait l'objet de tout un travail d'implémentation, voir par exemple [Ducournau, 1997] ou l'article [Zendra *et al.*, 1997] sur le compilateur SMALL EIFFEL.

Classes abstraites

On s'est aperçu très tôt que les classes jouaient au moins deux rôles, à la fois créatrices d'objets utilisables (instances) et détentrices d'informations exploitables par héritage (sous-classes). Et de plus, que certaines classes ne pouvaient jouer utilement que le second rôle : conçues pour être complétées par des sous-classes, elles ne possédaient pas assez d'informations pour engendrer des instances « viables ». Dans la tradition SMALLTALK, on les appelle des classes *abstraites*, en JAVA aussi, alors qu'en EIFFEL elles sont dites *retardées* (*deferred*). Les classes destinées à la procréation directe sont au contraire dites *concrètes*.

C'est l'exigence de contrôle statique qui a introduit un traitement particulier pour ces classes. En SMALLTALK, une classe abstraite est traitée par le compilateur comme une classe ordinaire : si le programmeur lui fait engendrer une instance, il prend le risque d'une erreur ultérieure s'il essaye de faire fonctionner cette instance — tant pis pour lui. Le mécanisme d'héritage suffit, que l'on interprète la complétion de la classe abstraite par un *ajout* de nouvelles méthodes ou par la *redéfinition* de méthodes qui étaient définies de manière non opératoire (définitions « bidon », se bornant à fixer le sélecteur, comme la définition standard `self SubclassResponsibility`).

Il est évident qu'un contrôle statique est possible, empêchant le programmeur de créer des instances qui ne jouiraient pas de toutes leurs facultés : les classes possédant des méthodes insuffisamment définies sont marquées « abstraites », avec interdiction de les instancier, et dans les sous-classes prétendues concrètes, le compilateur est en mesure de vérifier que les définitions sont bien complètes. Il suffit pour cela d'étendre la syntaxe afin de donner un statut aux définitions « bidon » (mot-clé `deferred` en EIFFEL, expression `= 0` en C++ (voir *chapitre 3*).

Cette variation très simple par rapport à l'héritage standard se révèle être un merveilleux outil de conception, voir par exemple *chapitre 3.* On notera que la distinction classe abstraite / classe concrète recoupe exactement la distinction aristotélicienne entre le genre et l'espèce [Brun, 1988, page 36]. Elle répond à l'obser-

vation courante selon laquelle on ne rencontre jamais un mammifère, mais toujours soit un chat, soit un chien, soit un homme, etc. L'espèce apparaît la dernière dans la hiérarchie des genres (en systématique, on dirait "des taxons", voir *chapitre 15*), auquel on ne peut plus ajouter que des accidents qui définissent des individus.

Limitations

La relation d'héritage entre classes, telle que nous l'avons décrite, est censée satisfaire le besoin de hiérarchisation des concepts. Il est manifeste qu'elle n'y réussit que d'une manière très partielle, puisque les seuls cas de particularisation qu'elle traite sont ceux qui proviennent d'une complication (ou d'un perfectionnement !), avec ajout d'attributs et/ou de méthodes. Le problème général se pose ainsi : étant donné deux ensembles d'objets `A` et `B`, avec `A` contenu dans `B`, peut-on trouver des définitions intensionnelles pour ces deux ensembles, sous forme de classes telles que nous les avons décrites ici, sur lesquelles le compilateur (ou le contrôleur de types) pourra effectivement conclure que toute instance de `A` est aussi instance de `B` ? En général, ce problème se pose alors qu'on a déjà une classe décrivant `B`, et il s'agit de trouver une description de `A` sous forme de sous-classe.

La réponse est en général non. Voici deux contre-exemples.

1. Prenons pour `B` la classe des rectangles et pour `A` celle des carrés. La manière intensionnelle ordinaire de décrire un rectangle est de le définir par deux points, alors qu'un carré se définit par un point et une longueur : il faudrait un démonstrateur de théorèmes puissant pour conclure automatiquement sur ces bases que tout carré est un rectangle ! La difficulté vient de ce qu'un carré n'est pas un rectangle « avec quelque chose en plus », mais bien un rectangle « tel que deux côtés consécutifs soient égaux », cette dernière restriction échappant complètement à nos possibilités descriptives.
2. À partir d'une hypothétique classe `Homme`, on imagine comment écrire une sous-classe `Barbu` en ajoutant un attribut et les méthodes afférentes. Mais comment décrire par des sous-classes les notions d'unijambiste ou de manchot ? Nous savons programmer la présence, mais non l'absence. Voyez dans un autre registre les problèmes de la négation en PROLOG ; il s'agit essentiellement de la même difficulté.

Pourtant, on considère depuis Aristote que toute définition est constituée du genre, qui est commun à plusieurs espèces, et des différences par lesquelles se distinguent les espèces [Brun, 1988, page 36]. Dans notre vocabulaire, ceci revient à dire qu'une classe se définit par la donnée de sa super-classe (son genre) et par ses différences spécifiques : or, Aristote aurait sans doute accepté comme différence valable la propriété que deux côtés consécutifs soient égaux, ou qu'un membre soit absent, alors que justement ces exigences sont inexprimables dans notre langage de programmation. Le problème est donc moins la structure générale de la définition par genre et différences que le domaine des différences exprimables dans le langage, qui pour nous se limite à l'ajout de méthodes et d'attributs.

On voit donc que le besoin de hiérarchisation n'est que très imparfaitement satisfait par le mécanisme simple de l'héritage. Mais ce besoin est tellement fort que

cette solution partielle a été saisie avidement, quitte à épiloguer sur les difficultés qu'elle engendre.

1.3 Discussion

1.3.1 Objets et valeurs

Il s'agit d'une distinction fondamentale, qui mérite d'être mise au premier plan de nos préoccupations. Qu'elle ait été jusqu'ici largement occultée par les tenants de la programmation par objets est en soi un intéressant sujet de réflexion.

Que les objets sont essentiellement réactifs, donc mutables

Qu'un objet soit doué de capacités réactives de par son comportement est une propriété fondamentale, indispensable pour lui donner l'individualité, la personnalité, qui vont créer chez le programmeur l'illusion d'existence concrète qui est le ressort de toute l'approche objet. Bien que notre objet informatique se réduise à une séquence de bits en mémoire, il faut qu'il déclenche les mêmes processus psychologiques qui fonctionnent dans notre intellect lorsque nous manipulons des objets réels. Nous éprouvons l'existence des objets réels parce qu'ils nous résistent, parce qu'ils réagissent, parce qu'ils répondent à nos enquêtes sensorielles en nous renvoyant des stimuli (visuels, tactiles, etc). Dans l'univers de la machine, l'acte de « toucher » ou de « sentir » un objet se transpose en un envoi de message, et la sensation correspondante se traduit par la réponse de l'objet. C'est donc en dotant nos entités informatiques d'une capacité de réaction que nous les faisons exister : *il fonctionne donc il existe*.

Notons ici que cette réactivité entraîne la plupart du temps que l'état de l'objet puisse changer au cours du temps. On pourrait penser que certains objets rigides ne changent jamais d'état, mais, en fait, la modélisation informatique ajoute souvent à l'état intrinsèque des variables comme la position de l'objet : un point est un bon exemple d'objet immuable, sauf si justement son état inclut ses coordonnées, et si on veut le faire bouger sur l'écran ... Nos objets sont donc de manière essentielle des entités mutables, ce qui entraîne des conséquences déplaisantes pour la théorie des types, comme on le sait (voir notamment la thèse de X. Leroy [Leroy, 1992]).

Mais il y a plus grave : vu qu'il existe de toute évidence, dans notre univers mental, des entités non réactives et non mutables (par exemple les nombres, plus généralement les abstractions mathématiques), l'observation précédente a comme corollaire que, dans un système de représentation équilibré, il doit y avoir aussi des structures qui ne sont pas des objets, ce que, dans une certaine tradition, on appelle des valeurs — notamment en bases de données (voir *chapitre 5*). Cette opposition entre les concepts d'objet et de valeur a été parfaitement analysée dans par B. Mac Lennan en 1982 [MacLennan, 1982], donc au moment même où l'approche objet entamait sa carrière.

Objets et valeurs : halte au « tout objet » !

C'est d'abord une affaire de modélisation : l'idée que l'objet est un individu connu par son adresse, et dont l'état peut changer, ne s'applique pas également bien à toute espèce d'entités. Par exemple, s'il est clair qu'une personne — ou une organisation — est un objet à part entière, identifié de manière unique, qu'en est-il de la date de naissance de la personne — ou de l'adresse de l'organisation ? En faire un objet à part entière expose au grave danger de voir une rectification anodine modifier les données relatives à d'autres personnes — ou organisations — ayant la même date de naissance (ou la même adresse), sauf à s'astreindre à une discipline particulière de recopie dans l'attribution des dates et des adresses. Or une telle menace est intolérable dans l'éthique des bases de données. Comme on le voit, la question de fond est : qu'entend-on au juste par « avoir la même date de naissance » (ou « avoir la même adresse ») ? Est-ce bien la même chose que « avoir la même mère » ou « travailler dans la même compagnie » ? On trouvera dans *chapitre 8* (section 6) une analyse des problèmes liés à l'identité des objets qui prolonge la présente.

Il vaudra donc mieux, dans certains cas, considérer que dates et adresses ne sont pas des objets, mais des valeurs, et construire une mécanique adéquate pour les traiter (implémentant une sémantique de valeur, dit-on en bases de données, voir *chapitre 5* et [Capponi, 1995]). Ainsi, des difficultés seront évitées, qui ne sont que désagréables pour les uns, mais carrément intolérables pour d'autres.

Évidemment, cette distinction complique le langage, et elle va à l'encontre du principe du « tout objet » revendiqué par des langages comme SMALLTALK et EIFFEL. Historiquement, ce principe simplificateur a pu être, il y a quelques années, générateur de progrès. Mais aujourd'hui nous pouvons nous permettre d'observer que, dans les faits, il ne s'applique jamais d'une manière complète : il existe toujours des classes bizarres (les nombres, les booléens, les caractères), dont l'implémentation échappe au modèle standard et qui n'entrent dans le cadre syntaxique uniforme que par une pétition de principe abusive. La distinction entre objets et valeurs se rencontre partout. Mais, dans certaines traditions, elle est mise au premier plan (notamment, dans celle des bases de données), et dans d'autres, on choisit de la cacher soigneusement (SMALLTALK), ou de la déguiser sous des considérations d'efficacité (voir la notion d'*objet expansé* en EIFFEL, *chapitre 2*, section 4). *Caveat programmator*[4] !

1.3.2 Classes et types

Comme nous l'avons indiqué plus haut (§ 1.1.2), leur interprétation extensionnelle donne aux classes une vocation évidente à servir de types. Et, dès lors, la relation d'héritage induit une relation de sous-typage (définie par la règle de substitution, voir paragraphe 1.2.3). EIFFEL a été le premier langage à tenter sur ces bases l'aventure d'un typage fort (voir *chapitre 2*). Le résultat est, disons, plus compliqué que prévu. Si l'on veut conserver des distinctions simples, on arrive avec [Cook *et al.*, 1990] à la conclusion que l'héritage n'est pas du sous-typage (*Inheritance Is Not Subtyping*).

4. Que le programmeur prenne garde !

On est ainsi conduit à séparer les notions de classe et de type (appelé *interface* en JAVA) et à les relier par un lien d'implémentation : un type se réduisant à une signature de méthodes [5], une classe qui implémente ce type (il peut y en avoir plusieurs) doit fournir un système d'attributs et les corps de procédure idoines réalisant la signature. Nous ne discuterons pas cette démarche, qui marque un retour à l'approche par types abstraits. Nous répéterons seulement quelques observations élémentaires attestant que, si l'on s'en tient au point de vue naïf que nous avons adopté ici, les choses ne se passent pas comme on le voudrait [Cook *et al.*, 1990].

Contrôle statique

Rappelons l'idée du contrôle de types statique, à la compilation : les objets apparaissent dans le texte d'un programme sous la forme d'expressions, dont on voudrait pouvoir prédire le type indépendamment de l'exécution du code. Il faut pour cela édifier une théorie des types qui doit rendre possible la détermination du type d'une expression à la compilation. L'entreprise est remplie d'embûches si l'on veut un typage sûr, qui garantisse qu'à l'exécution aucune erreur due à une incompatibilité de type ne se produira. Par opposition, on parle d'un *typage dynamique* (par exemple pour SMALLTALK) lorsque la non conformité des classes des objets avec les messages qui leur sont envoyés n'est détectée qu'à l'exécution.

Quoi qu'il en soit, il est clair que le typage statique impose au programmeur de s'expliquer beaucoup plus à fond que ne le demande le typage dynamique. Cette exigence peut être ressentie comme vexatoire ou comme salutaire, suivant le contexte. En particulier, elle rend pratiquement indispensable le recours à l'héritage multiple et aux *mixins*. Rappelons qu'on appelle *mixin*, après [Stefik et Bobrow, 1986] une classe qui ne spécifie qu'un genre de comportement très limité (par exemple, le fait de porter un nom), et que l'on peut à volonté insérer dans un graphe d'héritage multiple pour « ajouter » ce comportement à toute une hiérarchie de classes définies par ailleurs. On s'attend à ce que le vocabulaire correspondant soit unique dans le système, pour éviter les conflits qui sont la plaie de l'héritage multiple.

En effet, considérons un fragment de code exprimant une opération qui doit pouvoir s'appliquer à des instances appartenant à des classes différentes — par exemple, une visualisation. Dans ce fragment de code, une même variable doit pouvoir contenir lesdites instances et recevoir les messages adéquats — dans notre exemple, le message `voir`. Pour pouvoir déclarer cette variable, il faudra lui attribuer un type qui garantisse au compilateur que les envois de messages en question sont bien licites. À moins de disposer d'opérations sur les types comme la réunion ensembliste, il faudra donc créer une classe abstraite où seront déclarés lesdits messages, avec les attributs afférents, et dont devront hériter les classes concrètes des instances candidates à l'utilisation.

En d'autres termes, il faudra expliciter par une classe *mixin* le point de vue particulier sous lequel notre fragment de code regarde les objets — dans notre exemple, la visualisation. C'est là une technique classique, que le typage statique rend obligatoire.

5. Les interfaces de JAVA autorisent en plus la définition de constantes partagées par toutes les classes qui implémentent l'interface ; ce supplément ne change pas le fond de la question.

Contravariance des arguments

Cette question touche le type que l'on peut assigner à une méthode lorsqu'on la redéfinit dans une sous-classe. Elle fait apparaître un conflit entre la cohérence de la théorie, garante de l'économie de pensée du programmeur, et les intentions que l'on souhaite exprimer à travers les déclarations de types.

Rappelons que l'interprétation extensionnelle de l'héritage, qui dit que chaque instance d'une sous-classe est aussi instance de sa super-classe, se traduit du point de vue des types par la règle de substitution : étant donné une classe `B`, une sous-classe `A` de `B`, dans toute construction du langage où apparaît une instance `b` de `B`, soit `C[b]` (avec `C` comme *contexte*) et qui est déclarée correcte par le contrôleur de types, on peut substituer à l'objet `b` n'importe quelle instance `a` de la sous-classe `A`, sans que le contrôleur de types trouve à redire à la nouvelle construction `C[a]`.

Soit alors `p(x)` une procédure à un argument de type `X` définie dans `B` et redéfinie dans `A` avec `X'` comme type de son argument. Considérons la construction (fragment de texte de programme) `C[b] = b.p(u)`, où `u` est un argument effectif pour la procédure `p`. Cette expression est supposée correcte, l'argument `u` appartient donc au type `X` déclaré dans `B`. D'après la règle de substitution, l'expression `C[a] = a.p(u)` doit aussi être correcte. Or dans cette dernière, `p` désigne la procédure redéfinie (en vertu du lien dynamique), et, par conséquent, le contrôleur de types exige que `u` appartienne au type `X'`. Ceci étant vrai pour tout $u \in X$, cette exigence se traduit par l'inclusion de `X` dans `X'`.

On déduit de ce raisonnement fort simple que, lorsqu'on redéfinit une procédure dans une sous-classe, les nouveaux types de ses arguments doivent contenir les types stipulés dans la définition initiale. Cette règle, bien connue en théorie des types, est appelée « règle de *contravariance* des arguments ». Elle énonce en effet que, pour être « plus spéciale », une procédure doit avoir une domaine de définition « plus général ».

En revanche, le même raisonnement appliqué à une fonction au lieu d'une procédure montre que le domaine du *résultat* de la méthode redéfinie dans la sous-classe doit être inclus dans le domaine déclaré initialement, ce qui se traduit par la règle de « *covariance* des résultats ».

Or, l'intuition commune voudrait plutôt que la spécialisation d'une procédure redéfinie dans une sous-classe s'exprime en restreignant son domaine de définition, c'est-à-dire en déclarant `X'` inclus dans `X`, selon la règle dite de *covariance*. La plupart des concepteurs de langages ont décidé d'adopter cette règle, ce qui les oblige à introduire un appareillage spécifique pour éviter les incohérences dans le contrôle de types. Le meilleur exemple est fourni par la distinction entre *class-level* et *system-level* en EIFFEL, avec la théorie des *catcalls* (voir *chapitre 2*, section 8.2).

Cette contradiction entre la règle théorique simple qui gouverne le sous-typage et l'usage qu'on veut en faire explique la prudence de JAVA en ce qui concerne le profil des méthodes redéfinies : ce langage interdit tout simplement de changer de profil par rapport à la déclaration initiale, ni quant aux arguments, ni quant au résultat.

Types récursifs

La contravariance des arguments est la cause de maux innombrables. En voici un exemple bien connu.

L'expression « types récursifs » fait immédiatement penser aux arbres. Mais, dans le contexte de l'approche objet, le type associé à une classe devient récursif dès que le nom de la classe intervient dans son code, ne serait-ce que comme type d'argument ou de résultat dans une méthode. L'exemple classique de l'égalité mérite d'être médité.

Supposons que nous souhaitions doter une de nos classes `B` d'une méthode destinée à tester l'égalité de deux de ses instances. Elle aura tout naturellement pour signature : `egale(x:B):bool`, et le type associé à `B` sera récursif. Si, à présent, nous écrivons une sous-classe `A` de `B`, nous allons souhaiter redéfinir la méthode `egale`, pour tenir compte d'un supplément de structure, en `egale(x:A):bool` — ce qui est contraire à la contravariance !

Outre les remèdes généraux qui ont été proposés, il faut mentionner ici l'approche par *surcharge* : doter la classe `A` de deux méthodes `egale`, l'une de signature `egale(x:B):bool` (héritée de `B`), l'autre de signature `egale(x:A):bool` (redéfinie dans `A`), le choix entre les deux méthodes étant fait sur le type de l'argument à l'appel. On est alors dans le cadre des *multi-méthodes* de CLOS, où l'objet destinataire d'un message n'a plus le privilège du choix de la procédure à employer, et n'est plus que l'un des arguments de l'appel, le choix de la procédure effective se faisant sur la base des classes de tous les arguments. Cette généralisation qui semble au premier abord purement technique modifie en fait profondément le style de programmation : on revient au style fonctionnel, cette fois dans un cadre élargi, où les entités manipulées ne sont plus des données inertes, mais des objets complexes, le cas échéant réactifs, et classés de manière hiérarchique (voir [Habert, 1995]).

Mutabilité et généricité

Voici pour finir un exemple très simple montrant comment les exigences de typage cohérent en présence de généricité paramétrique (voir § 1.1.2) peuvent conduire à des situations inattendues. L'intuition ordinaire pousse à croire que, si `A[X]` est une classe générique, et si `U` et `V` sont deux classes concrètes, U héritant de `V`, alors le type `A[U]` est sous-type de `A[V]`.

Supposons que la classe `A[X]` soit celle de la pile générique et qu'elle possède une procédure `push(x)` à un argument. Soit `q` une variable déclarée de type `A[V]`, `v` une variable déclarée de type `V`, et considérons l'instruction `q.push(v)` : elle doit certainement être bien typée. Soit `p`, instance de `A[U]`, une « pile de `U` » : puisque `A[U]` est sous-type de `A[V]`, l'objet `p` peut être donné comme valeur à la variable `q`. Supposons à présent que la variable `v` ait pour valeur une instance « propre » de la super-classe `V` (c'est-à-dire, qui n'est pas instance de la sous-classe `U`). L'exécution de l'instruction `q.push(v)` aura pour effet de rompre l'homogénéité de la pile `p` en y insérant un élément qui n'est pas de type `U`. C'est précisément le genre d'erreurs que le contrôle de types est censé éviter !

La morale de l'histoire est que `A[U]` ne peut pas être sous-type de `A[V]`. La cause en est que l'argument de `push` (déclaré dans la classe générique `A[X]`) est

contraint (par l'effet que produit la procédure, qui modifie l'état de son destinataire) à être de type `X`, ce qui est incompatible avec la contravariance des arguments.

1.3.3 Classes et métaclasses

Un des aspects les plus stimulants de l'approche objet est la tournure que prend la recherche en matière de réflexivité. Comme on sait, cette manière d'aborder les langages de programmation est née en programmation fonctionnelle. Avec les objets, elle s'introduit d'une manière très naturelle, comme on va l'esquisser ici.

Les classes comme objets

Le vieux problème du statut des Universaux (en ce qui nous concerne, celui des concepts abstraits) se pose en des termes renouvelés dans le cadre de l'approche objets. Il s'agit de savoir si les concepts existent réellement dans le monde (position dite réaliste) ou s'ils n'apparaissent que dans notre discours sur le monde (position nominaliste), de savoir s'ils sont des choses ou seulement des mots. Dans notre image informatique, cela revient à savoir si les classes sont des objets (manipulables comme tels) ou seulement des textes (plus ou moins compilés). Sur ce débat qui remonte à la critique de Platon par Aristote, on trouvera une information exhaustive dans le livre d'Alain de Libera « La querelle des Universaux » [Libera, 1996]. La question est difficile : Aristote lui-même, tout en déniant aux concepts abstraits la qualité de sujet, leur accordait celle de substance — substances secondes, certes, mais néanmoins substances [Brun, 1988, page 32].

Dans le cadre de l'approche objets, le contexte se précise. Partant de la position nominaliste standard en informatique, on observe deux mouvements conjoints vers le réalisme. D'une part, dans la pratique de la programmation, on a souvent besoin de traiter les classes comme des objets — disons qu'il est commode d'adopter la même syntaxe pour les classes et pour les instances (cela simplifie l'écriture), et que dès lors il est souhaitable d'unifier aussi la sémantique. D'autre part, le fait que les classes soient des entités informatiques ne laisse aucun doute (surtout pour des programmeurs ayant subi l'influence de LISP, langage éminemment réaliste) sur la possibilité matérielle de les manipuler, pour peu qu'on en prenne la décision. Il y a donc un besoin, la possibilité technique est là, le réalisme devrait triompher ... Pourtant, le débat n'est pas clos.

Il y a d'abord des arguments « idéologiques » sur la nécessité ou l'inutilité de manipuler les classes à l'exécution. Si les classes sont vues avant tout comme des types, donc comme des instruments destinés à un contrôle statique effectué par le compilateur, on peut légitimement considérer qu'elles ont « disparu » au moment de l'exécution, et que l'idée même de les modifier ou d'en créer de nouvelles par programme est aussi incongrue que celle du mouvement en l'absence de moteur pour un disciple d'Aristote. Si, au contraire, on voit en elles des outils faits pour construire des objets, il en va tout autrement : les outils sont des objets particuliers, que l'on répare, perfectionne, voire que l'on construit dans l'atelier même. Dès lors, rien ne s'oppose à ce que la structure de ces outils soit décrite sous forme de classes. Comme les instances de ces classes sont elles-mêmes des classes, on les distingue

en les appelant *métaclasses*, par un emploi du préfixe *méta* courant en informatique. Sur le modèle du rapport entre physique et métaphysique, mais plus précisément en parallèle avec la distinction que les logiciens font entre langage et métalangage, le « niveau méta » désigne un niveau de représentation où sont définis les cadres de la représentation du niveau de référence [6]. Les classes sont ainsi au « niveau méta » par rapport aux instances, et les métaclasses par rapport aux classes.

Ces différences que nous qualifions d'idéologiques viennent en fait recouper le partage entre interprétation et compilation, ou plus exactement (car plus aucun langage opérationnel n'est strictement interprété) entre compilation incrémentale et interactive, façon SMALLTALK, et compilation séparée style EIFFEL ou C++. Dans le premier cas, où la tradition LISP est vivace, la matérialité de la représentation des classes à l'exécution ne pose point de problème, ni par conséquent leur élévation au rang d'objets de plein exercice. Dans le second, l'intervention du compilateur réduit les classes à des tables de *look-up* pour la liaison dynamique ; il est donc beaucoup moins facile de reconnaître leur existence comme objets véritables. Il est intéressant de noter que JAVA, qui dans sa première version se rangeait du côté nominaliste, devient nettement réaliste avec le JDK 1.1 et le *package* `Java.reflect`.

Il se pose aussi des problèmes d'ordre strictement logique. Notamment, celui de la régression à l'infini : si toute classe est un objet, une métaclasse est elle aussi un objet, instance d'une méta-métaclasse, etc. Il est intéressant de noter que cet argument a été employé par Aristote dans sa critique de Platon (argument dit « du troisième homme », [Brun, 1988, page 30] [Libera, 1996, page 75]). Dans la formulation informatique, ce problème devient celui du *bootstrap* (amorçage). Tout système qui admet les métaclasses doit le résoudre d'une manière ou d'une autre (voir [Masini *et al.*, 1989, sections 2.5 et 5.4], et en dernier lieu la thèse de Fred Rivard [Rivard, 1997]).

Réification et réflexivité

Dans un langage à objets, il n'y a pas que les classes dont on puisse se demander ce qu'elles sont. Les messages, notamment, ne sont en général pas des objets, et pourtant il est souvent utile de pouvoir les « attraper au vol » pour les analyser et, le cas échéant, leur appliquer un traitement adapté. À l'intérieur des classes, le statut des méthodes n'est pas clair : peut-on demander à une méthode, par exemple, combien elle accepte d'arguments ?

Mieux qu'avec le problème des métaclasses, on voit apparaître avec ce genre de questions la notion fondamentale de réification, clef de voûte de toute l'approche objet. Un message par exemple est une entité qui n'a pas normalement le statut d'objet. Pure forme syntaxique dans le texte, indiscernable de l'opération d'envoi qui le met en œuvre, il est traduit en une séquence d'instructions exécutables et n'a pas d'individualité repérable à l'exécution : le message n'apparaît que dans un envoi de message, donc dans un acte. Toutefois, nous concevons son existence objective :

6. Rappelons que *meta* est une préposition grecque ($\mu\epsilon\tau\grave{\alpha}$) 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 — $\mu\epsilon\tau\grave{\alpha}\ \tau\grave{\alpha}\ \phi\upsilon\sigma\iota\kappa\acute{\alpha}\ \beta\iota\beta\lambda\iota\acute{\alpha}$.

à nos yeux, le message a une structure, un objet destinataire, des objets arguments, un sélecteur. Dans certains cas, nous souhaitons pouvoir changer le point de vue du système d'exécution et obtenir qu'il considère effectivement le message comme un objet, conformément à la vue que nous en avons, afin de l'examiner et de décider sur son sort (par exemple, pour le ranger dans la boîte à lettres d'un acteur avec la priorité convenable).

L'opération qui fait passer du message « en acte » à l'objet analysable qui le représente s'appelle la *réification* (du latin *res*, chose : on pourrait dire *chosification*). Elle est plus ou moins disponible suivant les systèmes. En SMALLTALK, le compilateur réifie le message en cas de refus par le destinataire, et il utilise l'objet-message ainsi synthétisé dans son traitement d'erreur : il existe toute une technique d'implémentation fondée sur les possibilités ainsi ouvertes (par redéfinition de la méthode `doesNotUnderstand:`).

L'opération inverse, qui replonge un objet dans le flot de calcul, s'appelle *réflexion*. Elle apparaît dans la démarche générale connue sous le nom de réflexivité, notamment en programmation fonctionnelle, mais dans l'approche objet elle reste invisible. En effet, les objets du « niveau méta » sont activés comme les objets du niveau de référence, sans distinguer différents niveaux d'interprétation.

Dans le monde des objets *stricto sensu*, la mécanique est simple, et il n'y a pas grand'chose à réifier. La tentative extrême dans ce domaine reste le *Meta-Object Protocol* (MOP) de CLOS [Kiczales *et al.*, 1991]. En revanche, dans le monde des acteurs et des objets distribués, l'envoi de message revêt des formes variées (envoi avec ou sans attente de réponse, *broadcast*, envoi différé, etc), donnant à l'approche réflexive une vaste carrière (voir en dernier lieu la thèse de Claude Michel [Michel, 1997]).

1.4 Conclusion

Nous venons d'introduire le terme de *réification* dans un contexte strictement technique, dans l'espoir de faire « toucher du doigt » la nature de cette transformation. Bien évidemment, cette notion est d'un emploi très général : toute l'approche objet repose sur la possibilité de représenter les entités du domaine d'application envisagé par des objets. Mais certaines entités ne prennent pas « naturellement » le statut d'objet, de chose ! Un geste, un mouvement, sont-ils des objets ? Un avion est clairement un objet (compliqué), mais un vol (au sens des agences de voyage) ? une réservation ? Question de point de vue ? En vérité c'est une question de modélisation (voir *chapitre 4*). Une modélisation réussie réifie à bon escient, ce qui n'est pas toujours facile.

Le progrès des techniques rend accessibles des applications de plus en plus complexes, abordant des domaines de plus en plus éloignés des territoires bien balisés du calcul scientifique et de la gestion. Ces ambitions nouvelles conduisent souvent à des réifications hasardeuses. Plutôt que de chercher un exemple « réel » qui demanderait de longues justifications, je voudrais illustrer mon propos par une sorte d'apologue tiré du livre [Libera, 1996, pages 51–52]. Il s'agit d'un passage du Ménon de Platon, où Socrate exerce sa maïeutique sur un esclave et arrive à lui faire dire que les

abeilles ont une propriété en commun à l'égard de laquelle elles sont indistinguables — nous dirions, qu'elles sont instances d'une même classe. Et Socrate ajoute : *Eh bien c'est pareil pour les vertus ! Même s'il y en a beaucoup et de toutes sortes, elles possèdent du moins une seule forme caractéristique identique chez toutes sans exception, qui fait d'elles des vertus. Une telle forme caractéristique est ce qu'il faut bien avoir en vue pour répondre à qui demande de montrer en quoi consiste la vertu* (Traduction de M. Canto, collection Garnier-Flammarion).

En d'autres termes, pour Socrate les vertus sont elles aussi des objets, instances de la classe « Vertu » ! Il me semble que notre expérience collective nous suggère qu'écrire une classe « Abeille » en vue d'un projet informatique bien défini est chose *a priori* faisable, mais qu'il en va autrement pour les vertus, plus précisément que réifier la vertu est un procédé si épouvantablement réducteur qu'on ne peut l'envisager hors d'un projet informatique lui-même tellement réducteur qu'il vaut mieux ne plus parler de vertu ...

Plus profondément, il nous faut réfléchir sur les limitations intrinsèques de la démarche aristotélicienne induisant les concepts généraux à partir de l'expérience sensible que nous avons des individus, et dont la modélisation par objets est l'expression informatique directe. Les révolutions qui ont marqué l'histoire de la pensée scientifique et qui dans certains domaines ont disqualifié l'aristotélisme (notamment en physique) nous informent que le sens commun n'a pas toujours raison, et qu'il faut parfois passer par des cheminements cachés.