Langages à Objets

Jean-François Perrot
LAFORIA-IBP
Boîte 169, 4 Place Jussieu
75252 Paris Cedex 05
jfp@laforia.ibp.fr

Résumé

Texte de synthèse sur les principes des langages à objets, au sens de la programmation par objets, p. ex. SMALLTALK, SIMULA, EIFFEL et C++.

On part de la technique d'implémentation pour analyser les mécanismes offerts par ces langages (objets réagisssant à l'envoi de messages, classes et instances, héritage entre classes). On insiste sur les ambiguïtés qui s'introduisent inéluctablement et qui sont responsables aussi bien de la fécondité que des limitations de la programmation par objets. On discute au passage les notions connexes d'encapsulation, de généricité, de modularité et de hiérarchie de parties. On situe brièvement les langages ci-dessus mentionnés dans le cadre général ainsi tracé.

Enfin, on tente une évaluation des forces et des faiblesses de la programmation par objets, et on formule quelques observations de bon sens quant à sa mise en œuvre.

English Summary

This paper gives a synthetic presentation of the principles of object-oriented programming languages, such as SMALLTALK, SIMULA, EIFFEL and C++.

We start from the basic implementation techniques to analyse the constructs that these langugages offer (objects that react to messages, classes and instances, inheritance between classes). We put the stress on the ambiguities that creep in and are the source of the power as well as of the problems of object-oriented programming. We discuss also the associated notions of encapsulation, genericity, modularity and part-whole hierarchies. We briefly show how the above-mentioned languages fit in the general framework thus set up.

Finally, we try to assess the strengths and weaknesses of object-oriented programming, and we formulate a few common sense observations about its use.

Langages à objets

par **Jean-François PERROT**

Professeur à l'Université Pierre-et-Marie-Curie

1. Introduction
1,1 Une nouvelle manière de programmer
1,2 Bref historique

2. Les mécanismes de base
2,1 Les objets
2,2 Les classes comme technique d'implémentation
Instanciation
2,3 Les classes comme représentation des concepts
2,4 L'héritage: Principes
2,5 L'héritage: Mise en œuvre
2,6 Notions apparentées

3. Présentation de quelques langages
3,1 Smalltalk
3,2 Eiffel
3,3 Simula
3,4 C++

4. Mise en œuvre
4,1 Limitations
4,2 Nouvelles demandes

5. Bibliographie sommaire

Note : Cet article a été écrit pour le Traité d'Informatique des ***Techniques de l'Ingénieur***, où il doit paraître en mars 1995. Je remercie l'éditeur d'autoriser sa diffusion sous forme de rapport LAFORIA.

1. INTRODUCTION

Dix ans après son apparition en force, la technologie à objets se porte bien. Son succès s'affirme dans presque tous les domaines de l'informatique. Plusieurs congrès internationaux lui sont consacrés chaque année en Europe, aux USA et dans le Pacifique, les revues se multiplient ainsi que les réunions plus ou moins tapageuses à caractère commercial. On y entend souvent parler des technologies à objets, au pluriel. C'est que ce succès s'accompagne d'une confusion notable, les spécialistes des différents domaines ne donnant pas le même sens technique au mot *objet,* mot magique déjà lourdement chargé de sens dans la langue courante. Le barbarisme *orienté objets*, traduction directe de l'anglais *object-oriented,* n'aide pas à clarifier les choses. Nous lui substituons une terminologie plus conforme à la grammaire française : programmation *par objets*, langages *à objets*.

Pour éviter les malentendus, il convient de distinguer les domaines. Comme toute technique de programmation, la *programmation par objets* proprement dite peut se pratiquer en principe dans n'importe quel langage, mais de préférence à travers les *langages à objets.* Elle s'utilise à peu près partout, mais avec un éclat particulier lorsqu'il s'agit d'écrire des systèmes complexes. Elle rencontre alors les préoccupations du génie logiciel et donne naissance à l'*analyse et conception par objets*, qui à travers les multiples méthodes proposées sur le marché (comme OOA, OODLA, HOOD, OMT etc) pose des problèmes profonds d'acquisition des connaissances et rejoint une branche de l'intelligence artificielle en développement rapide. Comme le point de vue des objets est aux antipodes du modèle relationnel, les *bases de données à objets* ont fait leur apparition sur le marché [cf. Chrisment & al, Techniques de l'Ingénieur 6-1992, H 3 840]. Les *systèmes d'exploitation à objets* vont entrer en service d'un jour à l'autre, et on va voir se multiplier les applications distribuées fondées sur des*objets concurrents.* Les objets qui ont cours dans ces derniers domaines doivent affronter des conditions de vie difficiles (persistance, distribution), et ils ne sont plus exactement ceux des langages à objets classiques (cf. le standard CORBA promu par l'organisation OMG). Du côté de l'intelligence artificielle, la *représentation de connaissances* utilise elle aussi des objets, qui sont en fait dérivés des *frames* et se différencient de ceux qui nous occupent ici dans leur intention (p.ex. la classification) comme dans leur structure (absence de la composante procédurale).

Nous nous limitons dans le présent article aux langages (de programmation) à objets dans la perspective de leur utilisation en production de logiciel (et non, par exemple, du point de vue de l'intelligence artificielle). Nous n'aborderons pas l'analyse et la conception par objets, qui demanderaient un développement à part dont celui-ci est un préliminaire indispensable. De même, nous n'examinerons pas les problèmes liés à la permanence, à la concurrence et à la distribution des objets, ni les perspectives de systèmes d'exploitation ou de bases de données.

Notre présentation sera guidée par le principe d'implémentation qui sous-tend la programmation par objets, plutôt que par une analyse *a priori* des besoins qu'elle devrait satisfaire et des moyens qu'elle propose. C'est à notre avis la seule façon d'évaluer sainement les possiblitiés de la technique, les constantes et les variations entre les différentes réalisations, et surtout, pour l'utilisateur, de ne pas prendre ses désirs pour des réalités. Cette dernière tentation est particulièrement forte en programmation par objets, en raison d'une ambiguïté fondamentale que nous analyserons longuement. En fin de compte, c'est sans doute cette ambiguïté qui est la source de la fécondité et de la richesse de la programmation par objets,

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 ?".

Si on se borne aux logiciels qui sont effectivement utilisés de manière industrielle, par opposition aux systèmes à caractère expérimental ou d'usage restreint, la diversité apparente des langages à objets actuellement disponibles se ramène à une architecture unique, le schéma dit *à classes et instances,* expression où le mot *classe* doit être pris en un sens technique précis qui sera détaillé au § 2,2. Ce schéma repose en vérité sur une technique d'implémentation d'une remarquable simplicité, qui a des conséquences multiples et lointaines. Nous l'analyserons en trois temps. D'abord, la structure profonde du modèle à classes et instances, suivant ses deux aspects indissolublement liés de mécanisme intellectuel et de réalisation informatique: l'un ne va pas sans l'autre. Ensuite, nous passerons en revue, à la lumière de ces principes, les langages les plus connus. Enfin, nous ferons une analyse critique en confrontant les moyens de programmation ainsi définis avec les desiderata des utilisateurs, qui leur sont souvent fort étrangers.

1,1 Une nouvelle manière de programmer

1,1,1 Styles de programmation

Programmer, c'est communiquer une requête à une machine: on écrit un programme pour faire exécuter une certaine tâche à un ordinateur. La tâche en question peut être aujourd'hui fort complexe, et sa formulation directe dans le langage de la machine est hors de question. Le programme s'écrit dans un langage de programmation, dont le rôle est de soutenir l'effort intellectuel du programmeur en lui fournissant un cadre conceptuel plus ou moins confortable, et plus ou moins bien adapté à la tâche devant être décrite. Les langages s'organisent selon plusieurs grands styles de programmation, qui diffèrent suivant les mécanismes intellectuels qu'ils mettent en jeu chez le programmeur et suivant les qualités de la formulation obtenue du point de vue de son traitement ultérieur (compilation, exécution, débogage, maintenance).

Le style impératif/procédural classique cherche à formuler la tâche à accomplir d'une manière directe en termes de changements d'états de la machine. Il a l'inconvénient d'enfermer le programmeur dans des structures de données rigides, soumises à des algorithmes plus ou moins compliqués, et de le contraindre à une analyse approfondie du problème à traiter. Cette analyse conduit à une reformulation abstraite en termes de systèmes de variables, et de procédures opérant sur ces variables. Un tel exercice ne peut être pratiqué que par des professionnels rompus à l'algorithmisation et à la manipulation des structures de données, munis d'une culture technique que ne possède pas le commun des mortels. En outre, malgré les efforts prolongés des spécialistes, les textes des programmes obtenus se prêtent mal à une vérification automatique de leur validité.

Le style fonctionnel, de son côté, envisage la définition de la tâche à accomplir comme celle du calcul d'une fonction mathématique. Il a l'avantage de pouvoir mettre en branle une véritable théorie de la programmation, qui conduit par exemple à une forme de vérification sémantique, la synthèse de types. Il a l'inconvénient d'exiger de la part du programmeur un bagage mathématique non négligeable et de nécessiter un retour au style impératif pour traiter bon nombre de problèmes "pratiques" comme les entrées-sorties. La programmation logique

(PROLOG, programmation par contraintes), où la définition de la tâche prend la forme d'une théorie en logique du premier ordre, souffre des mêmes inconvénients.

1,1,2 Objets et outils

La programmation par objets en tant que style de programmation se distingue en faisant systématiquement appel à des mécanismes de pensée très largement répandus, qui en particulier ne sont pas l'apanage des scientifiques. Cet aspect est d'autant plus important que les projets logiciels croissent en taille et en ambition, et qu'ils font plus largement appel aux spécialistes des domaines d'application concernés; ceux-ci, en général, ne sont ni des mathématiciens, ni des programmeurs rompus au dialogue avec la machine, mais peuvent être des ingénieurs, des administrateurs, des commerciaux, des juristes, etc. En outre, le développement de logiciels par des équipes nombreuses vient ajouter le problème de la communication à celui de la compréhension individuelle. Il est donc essentiel de faire appel à des mécanismes intellectuels partagés par le plus grand nombre d'utilisateurs.

L'intuition de base est la suivante : une partie notable de notre activité quotidienne consiste à manipuler des objets matériels complexes, outils et appareils divers - de la voiture au stylo en passant par l'appareil photo. Nous le faisons sans savoir au juste comment fonctionnent ces outils, en nous contentant de les utiliser pour atteindre nos buts: voyager, écrire, faire des photos. C'est, bien sûr, que lesdits outils ont été conçus et construits précisément pour ce faire, et que nous en avons appris le mode d'emploi - mais pas nécessairement les principes physiques ni la technique de réalisation.

Nous sommes aussi capables d'imaginer, pour une tâche quelconque, qu'elle pourrait être effectuée au moyen d'un outil *ad hoc.* C'est même une capacité qui nous distingue des autres primates, chez lesquels la facture et l'utilisation d'outils ne vont guère au delà de prendre un bâton pour atteindre un régime de bananes ou un caillou pour casser des noix. L'outil en question sera très souvent obtenu en combinant d'autres outils préexistants, en procédant le cas échéant par essais et erreurs, sans faire appel à une analyse théorique poussée de la tâche. Nous parlons ici bien entendu de l'expérience commune, non de la construction du bras manipulateur de la navette spatiale.

Il s'agit alors de transposer nos compétences en matière de conception et d'utilisation d'objets à l'univers informatique. En somme, de programmer comme on joue au Meccano, ou comme on conduit sa voiture - entendons par là, en utilisant les mêmes ressources intellectuelles. Réaliser cette transposition n'a rien d'évident, et l'ébauche de solution que propose la programmation par objets est d'une pauvreté navrante. Mais, le peu qui est ainsi gagné s'avère d'une efficacité redoutable. Dans une alliance remarquable avec les progrès récents en matière de taille de mémoire et de périphériques graphiques, il a fait faire un bond en avant à la production du logiciel.

1,1,3 Les trois mécanismes fondamentaux

Voici les trois principes qui sont au cœur de la programmation par objets et qui, selon nous, lui donnent son efficacité.

(a) *Le recours à la simulation, au modèle réduit* : version moderne du célèbre adage selon lequel *un petit dessin vaut mieux qu'un long discours.* L'exécution du programme se ramène à l'activation du modèle. Ceci est l'idée-clef, tant du point de vue du programmeur individuel que d'un point de vue social global : c'est pour construire ce modèle que l'on espère

introduire les composants logiciels proposés en premier lieu par Brad Cox, et qui sont le sujet de tant de débats actuels.

(b) *La distinction entre concept (abstrait) et instance (concrète) du concept,* alias définition/exemple. Comme on verra, le recours à l'abstraction est ici indispensable pour réaliser le modèle du monde. Cette distinction apporte à la fois un mécanisme intellectuel puissant et la clef de la réalisation informatique de l'idée esquissée ci-dessus en (a). En programmation par objets, la réalisation informatique d'un concept s'appelle une *classe,* et un objet individuel a le statut d'*instance.*

(c) *La classification des concepts, fondée sur l'idée intuitive de généralisation.* Le besoin de classer se rencontre dans toutes les sociétés humaines. Il n'est donc pas étonnant que ce troisième mécanisme soit souvent retenu comme la caractéristique majeure de la programmation par objets. Mais, comme nous le verrons, sa réalisation informatique, qui est solidaire de celle du point (b), l'éloigne fort de l'intuition commune.

Naturellement, ces idées ne sont pas nouvelles, ni isolées : (b) et (c) remontent à Aristote. On a pu - on peut encore - programmer par objets dans un langage quelconque, comme l'attestent par exemple les boîtes à outils graphiques construites en C dans le système de fenêtrage X (alias *X Window System*). Mais le langage de programmation peut rendre la tâche plus ou moins facile. Il est toujours possible de "penser contre le langage", mais c'est difficile, fatigant, en somme peu efficace.

1,2 Bref historique

1,2,1 Un style impératif

D'un point de vue génétique, la programmation par objets dérive de la programmation structurée des années 70, qui relève elle-même du style impératif. Comme on sait, un programme impératif opère par effets de bord successifs sur l'état de la mémoire de la machine. Une des difficultés de cette manière de faire est que l'état de la mémoire n'est pas une entité dont on puisse facilement avoir une vue globale. Pour aider à sa compréhension, la programmation structurée offre la notion de type de données; la mémoire apparaît comme une collection d'objets inertes sur lesquels opèrent des procédures. Le programmeur travaille ainsi avec une représentation du monde écartelée entre données et procédures, alors que les unes et les autres sont faites pour travailler de concert. En outre, les procédures n'offrent pas un support très sûr à l'intuition, et ce style de programmation conduit souvent à des édifices procéduraux complexes qu'il est bien difficile de construire sans erreur.

La programmation par objets abolit la séparation globale entre données et procédures au profit de la construction d'un modèle du monde homogène, dont les composants sont, à l'instar des objets du monde réel, à la fois supports d'information (données) et susceptibles de comportements qui leur sont propres (procédures). Un tel modèle est mieux intelligible pour deux raisons. D'abord il met en jeu des objets individualisés activables, qui correspondent fidèlement à ceux du monde réel et se prêtent bien à une vision intuitive, par transfert de l'expérience courante. Ensuite la hiérarchie des classes, selon la relation d'héritage, permet une économie de pensée considérable en concentrant l'attention du programmeur sur ce qui est spécifique à chaque concept, les aspects génériques étant "factorisés" par l'héritage.

C'est cette meilleure intelligibilité qui fait la force de la programmation par objets, car au demeurant un programme à objets opère lui aussi par une succession d'effets de bord, de modifications de l'état de son modèle du monde, à travers les changements d'état des objets qui le composent. Et par conséquent la programmation par objets souffre de tous les maux bien connus de la programmation impérative en matière de spécifications abstraites et de preuves de correction. Ce caractère intrinsèquement impératif de la programmation par objets rend aléatoire son mariage (pourtant annoncé à plusieurs reprises) avec des représentants du style fonctionnel et plus encore, du style logique.

1,2,2 Objets et types abstraits

A ses débuts, avec le langage SIMULA-67, rien ne distinguait l'orientation vers les objets de l'orientation vers les types abstraits. Mais cette dernière - théorie algébrique oblige - a évolué vers un style de plus en plus fonctionnel et logique, faisant une large place aux problèmes de définition axiomatique, de contrôle et de vérification de types. La programmation par objets est restée au contraire résolument concrète. Avec ses classes, elle fournit des types concrets plutôt qu'abstraits, donnés par leur implémentation et non par des axiomes, et le souci d'abstraction se réduit chez elle à la recherche de l'encapsulation (cf. § 2,5).

Outre ce caractère "primitif", ce qui distingue la programmation par objets de la programmation par types abstraits, c'est la relation d'héritage. Du point de vue des types, l'héritage se traduit par une relation de sous-typage dont le traitement théorique présente des difficultés substantielles, c'est pourquoi elle n'est pas encore entrée dans l'arsenal de la programmation par types abstraits. Vu son importance dans le développement de logiciels à grande échelle, c'est donc la notion d'héritage qui, selon certains auteurs, marque la transition des premiers langages réputés "à types abstraits" (CLU, ADA) vers les langages dits "à objets" (C++, EIFFEL).

A notre avis, cette transition se trouve aussi dans une orientation plus pragmatique, affirmant que les objets sont des êtres concrets, persistants, modifiables, et abandonnant l'idée d'une spécification fonctionnelle pour se contenter d'une définition procédurale des méthodes. On y a certes perdu en rigueur théorique et en possibilité d'automatisation des preuves, mais on y a gagné en possibilité de réalisation rapide et à grande échelle : le système SMALLTALK-80 a donné une visibilité éclatante à ce fait désormais incontournable, que programmer par classes et instances est facile et efficace. On assiste d'ailleurs aujourd'hui à une nouvelle convergence entre l'approche pragmatique par objets et la démarche plus rigoureuse à base de spécifications par types abstraits, consistant à introduire un mécanisme de contrôle de types dans les langages à objets.

1,2,3 Chronologie

SIMULA (né en 1967 à Oslo, toujours utilisé), dérivé d'ALGOL-60, est aujourd'hui le premier exemple reconnu de langage à objets. On y trouve tous les concepts dont nous avons parlé, mais avec en plus un puissant mécanisme de coroutines: SIMULA est un langage à plusieurs piles ! Il est intéressant de noter comment, au cours de sa genèse, les notions d'objet et de classe se sont introduites comme des moyens pour réaliser les coroutines. Une classe en SIMULA est essentiellement une procédure interruptible et réactivable. Sa forme syntaxique est exactement celle d'une procédure, elle-même directement héritée de la procédure d'ALGOL-

60: déclarations de variables et de procédures locales suivies d'une séquence d'instructions. Cette séquence d'instructions décrit l'activité initiale de chaque instance lors de sa création. On s'attend à ce qu'elle soit interrompue par l'exécution d'une instruction `detach`, pour être ensuite relancée par un `resume`. L'instance apparaît donc comme un état suspendu de la procédure.

En somme pour arriver au schéma classe/instance désormais traditionnel, il a fallu seulement abandonner cette vision dynamique à plusieurs processus et se contenter d'objets statiques, avec un schéma d'interprétation à une seule pile. Tout le reste était en place: les variables locales sont devenues les variables d'instance (suivant la terminologie de Smalltalk), les procédures locales ont été rendues accessibles depuis l'extérieur et sont ainsi devenues des méthodes (idem), la notion d'héritage n'a pas changé. Comme quoi il est parfois historiquement utile de faire compliqué, à condition de revenir ensuite à la simplicité, seule vraiment durable.

La vogue des objets date du lancement de SMALLTALK-80 comme produit logiciel, après une longue maturation dans le laboratoire de Xerox à Palo Alto, marquée par les étapes de SMALLTALK-72 et de SMALLTALK-76. Les livres fondamentaux d'Adele Goldberg et David Robson ont paru en 1983, soit dix ans après les débuts de la recherche. Les idées de SMALLTALK reprennent, en les simplifiant, celles de SIMULA, qui enthousiasmaient un cercle réduit d'aficionados depuis la fin des années 60. Elles rejoignent, en les simplifiant également, celles du système des FLAVORS à l'œuvre sur les Machines LISP dès 1980. Le premier à suivre fut Brad Cox avec son OBJECTIVE-C (1984), les autres concurrents (C++ de Bjarne Stroustrup, EIFFEL de Bertrand Meyer) apparaissent à partir de 1985.

Parallèlement, l'intégration des objets dans l'univers LISP a conduit à de nombreux systèmes expérimentaux, qui ont abouti à deux réalisations majeures : le *CommonLisp Object System* (CLOS), qui fait aujourd'hui partie intégrante de la norme COMMONLISP , et *DYLAN* (Dynamic object-oriented Language), proposé par Apple, Inc., encore en évolution. Nous ne dirons rien de cette branche passionnante, mais passablement différente du "courant principal", et qui demanderait un développement particulier.

1,2,4 Présupposés techniques

Cela dit, il reste à élaborer l'outillage informatique qui permettra de construire ces modèles réduits et de les faire fonctionner. Les choix que nous allons détailler sont fort difficiles à justifier si on ne tient pas compte des moyens informatiques supposés disponibles. Ces hypothèses sont en général passées sous silence alors même qu'elles engendrent le cadre conceptuel sous-jacent à tout le discours de la programmation par objets. Elles sont responsables d'une bonne partie des malentendus entre spécialistes venant d'horizons différents. Par exemple, les systèmes à objets nés dans un univers LISP ne sont pas soumis aux mêmes lois que ceux qui sont construits directement sur une machine abstraite (SMALLTALK) ou encore compilés en C (EIFFEL).

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.

Soulignons ensuite que la distinction traditionnelle entre données et procédures ne peut pas être balayée d'un revers de main, comme certains théoriciens tendent à le faire. La différence est patente si on se trouve en dehors de langages manipulant les véritables valeurs fonctionnelles qui sont l'apanage de SCHEME, COMMONLISP ou ML; même dans ces langages, d'ailleurs, les fonctions ont des propriétés bien particulières comme de ne pas supporter de prédicat d'égalité. Dans le monde où se déroule notre histoire, celui de l'assembleur, de PASCAL ou de C, les valeurs fonctionnelles sont inconnues et il est tout-à-fait illusoire de vouloir traiter données et procédures sur le même pied.

En revanche, nous supposons acquis le partage de la mémoire de travail en trois zones, la zone code, la pile et le tas, suivant le principe de LISP. 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 glanage de cellules (expression moins malodorante que *garbage-collector,* toujours abrégée en GC). 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.

Enfin, 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, et que nous ne nous autorisons à leur égard que trois opérations : la rédaction (écriture dans un fichier), la compilation (ou mise sous forme interne) et l'exécution.

2. LES MÉCANISMES DE BASE

2,1 Objets informatiques

2,1,1 La simulation et la notion d'objet

Comme nous l'avons dit, le but visé est de faire fonctionner un programme non pas comme un algorithme, mais comme l'animation d'un modèle réduit. Ce modèle est construit à partir d'entités informatiques qui reproduisent "fidèlement" les objets du monde réel (voir plus loin). On appellera désormais ces entités des *objets.* Comment définit-on le sens technique ainsi donné au mot "objet", là est la question ! Il nous faut expliquer d'abord comment ces objets fonctionnent (ci-après) et ensuite comment ils sont créés dans la machine (cf. 2,2).

Plus que de fidélité dans la reproduction, il faudrait parler d'approximation suffisante pour atteindre le but assigné au programme. Bien évidemment, les objets informatiques n'épuiseront pas la complexité des objets réels, et d'ailleurs toute une tradition philosophique nous prévient que notre connaissance du réel est imparfaite. Mais s'il s'agit de l'utiliser pour des fins bien définies, l'ingénieur reprend ses droits. Dans toute la suite, nous supposerons donc que l'analyse se fait en fonction d'un projet spécifique et non sous l'aspect de l'éternité.

2,1,2 Définitions

Nos objets n'ayant de sens qu'en raison de leur fonctionnement, il nous faut introduire d'un même coup plusieurs termes qui se définissent l'un par rapport à l'autre.

(a) Un ***objet*** est une entité individuelle, repérée par une adresse unique.

(b) Il est formé de plusieurs ***champs***, connus par leurs noms (en nombre fixé), à la manière d'un enregistrement (*record*) de PASCAL.

(c) Chaque champ contient une valeur qui peut varier au cours du temps (c'est en général l'adresse d'un autre objet). L'ensemble de ces informations constitue à chaque instant ***l'état*** courant de l'objet.

(d) Il reçoit des ***messages*** et traite chacun d'eux en exécutant une ***procédure*** qui correspond au message, l'effet de cette exécution dépendant de son état et pouvant également modifier cet état. L'ensemble des procédures en question définit le ***comportement*** de l'objet.

La nomenclature que nous employons est soumise à deux contraintes : d'une part se séparer des terminologies utilisées par les différents langages, et que l'on trouvera à la section 3, d'autre part mettre en évidence les distinctions que nous estimons essentielles. Nous ne sommes pas parvenu à une parfaite homogénéité, et nous nous excusons des incohérences qui ne manqueront pas de gêner le lecteur. Pour nous le champ est le contenant de l'attribut. D'autre part nous tendrons à confondre procédures et méthodes : la différence étant non de substance, mais de point de vue - la procédure est "pour elle-même" et la méthode (terminologie de Smalltalk) "pour sa classe" .

Nous insistons sur l'idée de messages envoyés aux objets, alors même que ces messages ne font qu'activer des procédures. C'est que le changement de point de vue qui accompagne l'idée de message nous paraît fondamental. L'objet est ainsi perçu comme une entité "respectable", complexe, à laquelle on s'adresse dans son langage, etc. Cette intuition est bien exprimée dans la syntaxe de Smalltalk. La plupart des autres langages utilisent la notation pointée issue des enregistrements (*records* de PASCAL, *structures* de C), et écrivent les envois de messages comme des appels de procédures qui seraient des champs de ces enregistrements. C'est correct du point de vue de l'implémentation - encore que, comme nous l'avons dit, les procédures ne soient pas véritablement des valeurs de champs - mais on perd ainsi le respect de l'objet.

Soulignons que dans cette définition les procédures activées par les messages, qui définissent le comportement de l'objet, ne sont pas traitées comme les valeurs de champs de l'enregistrement, puisque, comme nous l'avons rappelé, nous ne considérons pas que des procédures puissent valablement 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 une batterie de procédures qui définit son comportement.

2,1,3 Exemple : Un objet représentant un point sur l'écran, dans la perspective d'un projet où on voudrait tracer des trajectoires allant de haut en bas et de gauche à droite *(Fig. 1)*.

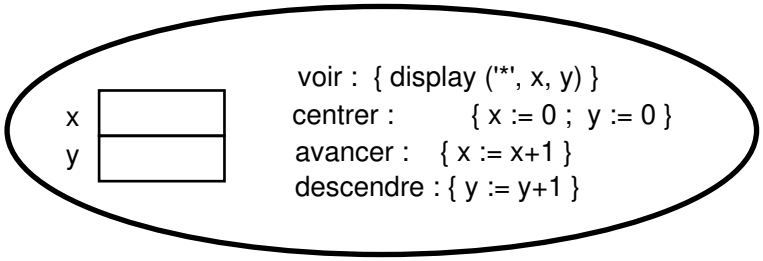

Figure 1.

Il est constitué d'un enregistrement à deux champs nommés `x` et `y`, où sont logées ses deux coordonnées rectangulaires. Il est susceptible de recevoir quatre messages `voir`, `centrer`, `avancer` et `descendre`, en réponse auxquels il exécute l'une des quatre procédures indiquées entre accolades. Le but de ces quatre procédures n'est pas de définir un point en toute généralité, mais par rapport à une intention bien précise (ici, un tracé de trajectoires).

2,1,4 Commentaire

D'abord, on entend par *objet* une entité "concrète", "qui est là", conformément à l'étymologie, "rémanente" par opposition à une fonction ou à une procédure, perçue comme un tout bien individualisé : en termes informatiques, ceci se traduit par sa désignation via une adresse unique. Cette individualité ne signifie pas immutabilité ! Au contraire, nous distinguons parfaitement un objet réel donné et l'état dans lequel cet objet se trouve (par exemple, si une porte est ouverte ou fermée) - même s'il est difficile de définir rigoureusement le point où un objet réel perd son individualité au cours d'une transformation, par exemple lorsqu'on le détruit.

La difficulté, comme dans la réalisation de "vrais" modèles réduits, est de lui conférer un certain degré d'existence. 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. De ce point de vue, un des attributs de l'existence est la réactivité : 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 en dotant nos entités informatiques d'une capacité de réaction que nous les faisons exister.

A condition de se limiter à des messages rudimentaires et à des réponses stéréotypées, c'est facile. Il suffit de considérer que la réaction de l'objet consiste à exécuter une procédure, et que le message qui aura déclenché cette réaction comportait l'indication de cette procédure et de ses arguments éventuels. On demande toutefois que l'effet produit par l'exécution de la procédure dépende de l'état courant de l'objet (sinon, l'individualité de l'objet

serait en cause). Voilà un modèle de communication assurément simpliste, mais réalisable et somme toute fort efficace.

En termes informatiques, cela signifie une entité permanente, qui existe indépendamment du nom qu'on lui donne, que l'on crée explicitement (mais dont la destruction est de préférence automatique), non soumise aux mouvements de la pile. La rémanence suppose que l'on dispose d'un tas et d'un gestionnaire de mémoire qui peut être extrêmement perfectionné. La définition 2,1,2 indique un schéma d'implémentation pour cette entité, il nous faut à présent compléter ce schéma.

2,2 Les classes comme technique d'implémentation

2,2,1 Des objets aux classes

Il reste à expliquer comment les objets que nous avons définis ci-dessus peuvent être créés en machine. Le problème central est celui de la genèse des comportements des objets. C'est ici que les hypothèses que nous avons faites au §1,3 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 les comportements d'un objet. Cette circonstance donne la primauté à la représentation textuelle et conduit tout droit à la notion de classe. Si les comportements pouvaient être obtenus 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 on cherche à les cacher comme dans les langages à prototypes, ne sont donc jamais bien loin.

2,2,2 Définition

(a) Une classe est un texte structuré qui rassemble d'une part une liste de noms de champs, d'autre part un catalogue de procédures (noms et textes). Les noms de champs apparaissent comme les seules variables libres dans les textes de procédures.

Ce texte apparaît sous deux formes, soit comme texte source dans un fichier, soit comme texte compilé en mémoire.

(b) 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 les procédures fournies par la classe en réponse aux messages stipulés par les noms des procédures dans le catalogue détenu par la classe.

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

Le principe de l'interprétation des messages est le suivant :

- le message fournit directement le nom de la procédure à excécuter, et le corps de procédure correspondant est recherché dans le catalogue de la classe ;
- 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).

2,2,3 Exemple : définition de l'objet "point" du § 2,1,3 comme instance d'une classe *(fig. 2)*

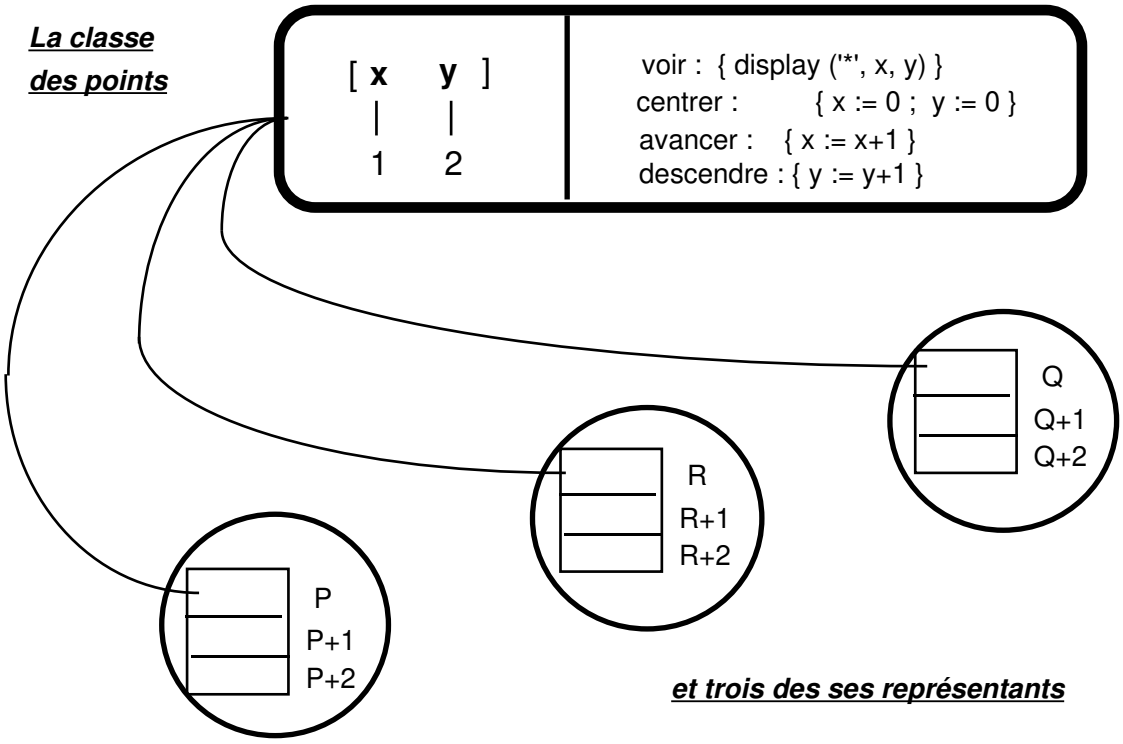

Figure 2.

Lorsque le point repéré par l'adresse `P` reçoit le message "avancer" (par exemple), le corps de procédure `{x := x+1}` est exécuté avec `x` désignant le mot-mémoire d'adresse `P+1` puisque `x` est le *premier* nom de champ figurant dans la liste `[x, y]` détenue par la classe.

2,2,4 Polymorphisme

On observera au passage un aspect extrêmement important de la mécanique ici décrite: le texte d'un message contient le nom de la procédure à exécuter, mais le corps de procédure qui sera effectivement exécuté dépend de la classe de l'objet qui reçoit le message. En termes imagés, à la réception d'un message, c'est l'objet qui décide (via sa classe) quelle opération entreprendre. Le même message envoyé à deux instances de classes différentes conduit donc à des opérations différentes, puisque dans les deux classes le même nom de procédure sera associé à des corps différents. On retrouve ici un phénomène analogue à la "surcharge des opérateurs" (par exemple, le fait qu'on note l'addition des entiers et des flottants avec le même signe "+", bien que les algorithmes soient différents). Dans le jargon des objets on préfère parler de polymorphisme.

Le bon usage de cette tour de Babel des classes veut qu'on désigne par le même nom des opérations analogues, voire homologues : par exemple, on apellera systématiquement `voir` les opérations d'affichage, `avancer` et `descendre` des opérations de translation d'une figure le long des deux axes,etc. Le polymorphisme s'avère alors un instrument extrêmement efficace, grâce à l'économie de pensée qu'il autorise. Il permet d'imaginer que l'objet "sait se débrouiller", qu'il "possède la compétence" pour exécuter dans le détail la tâche qui lui a été

demandée d'un mot. Cette sorte de transfert de responsabilité de l'utilisateur à l'objet utilisé prend tout son intérêt lorsque la classe a été écrite par quelqu'un d'autre (p. ex., une classe standard du système) : l'utilisateur sait ce qu'il veut obtenir, mais il peut se dispenser de savoir exactement comment le résultat sera atteint. En outre, on retrouve à ce niveau quelque chose de l'ambiguïté du langage parlé, et c'est très naturellement qu'on emploie le polymorphisme.

Bien entendu, il n'est pas toujours facile de maintenir une terminologie homogène. C'est affaire de culture, et chaque langage à objets véhicule la sienne. On rencontre parfois des difficultés inattendues, comme d'employer une terminologie française alors que le système bien entendu "parle anglais". Mais somme toute le problème du vocabulaire se résout assez bien, et il est clair que l'emploi d'outils empruntés à l'intelligence artificielle permettra de le résoudre mieux encore dans l'avenir. Avec le polymorphisme, nous venons de voir une des sources majeures de l'efficacité de la programmation par objets. Il va réapparaître un peu plus loin dans le cadre de l'héritage, sous une forme particulièrement percutante.

2,2,5 Instanciation

La classe ainsi définie est d'abord écrite par le programmeur avec une syntaxe concrète *ad hoc*, puis compilée (ce qui n'offre de toute évidence aucune difficulté cachée). Elle doit être compilée avant que soient créées ses instances. Le nombre d'instances d'une même classe n'est limité que par la taille de la mémoire disponible.

Le processus d'instanciation, c'est-à-dire 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, 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.

Il apparaît clairement que, dans ces conditions, un objet ne peut être instance que d'une seule classe, au grand dam de certains utilisateurs qui souhaiteraient exprimer l'évolution d'un objet en le faisant changer de classe au cours de son histoire... Nous reviendrons sur ce genre de limitations.

2,2,6 Commentaire

Soulignons que le procédé consistant à écrire la classe, puis à la compiler et enfin à créer des objets instances de cette classe résout complètement le problème de la création des objets. Le partage en deux étapes écriture-compilation de la classe d'une part, instanciation de l'autre, peut paraître inutilement complexe, d'autant plus qu'il fait apparaître une catégorie nouvelle d'entités informatiques, celle des classes, qui n'était pas requise *a priori*, contrevenant ainsi à la règle du rasoir d'Ockham, qui veut que l'on ne multiplie point les entités sans nécessité.

Mais il n'est pas très facile de proposer une autre solution à ce problème sans faire appel à quelque artifice caché. Tout au plus pourra-t-on décider que chaque texte définit non pas une classe à partir de laquelle on créera un nombre *a priori* illimité d'instances isomorphes, mais un objet unique, contenant ses champs et son catalogue de procédures. C'est évidemment réalisable, voire justifié si on envisage systématiquement chaque objet comme singulier; toutefois, pour y parvenir il faudra trouver un compromis avec une organisation standard de la

mémoire qui tend naturellement à loger le code des procédures dans une zone différente de celle des enregistrements représentant les états des objets, et donc à réaliser le *partage de code* que l'on mentionne souvent comme la raison d'être des classes. Le partage de code n'est pas une optimisation secondaire, c'est la solution naturelle au problème de la représentation des objets.

Nous estimons donc que la réalisation des objets comme instances de classes est la solution la plus simple en l'état de la technique auquel nous nous référons. Une des vertus des classes est d'être des textes, ce qui résout d'entrée de jeu le problème de leur stockage permanent et de la constitution de ces bibliothèques qui jouent un si grand rôle dans la suite de l'histoire. Comme nous allons le voir, cette solution a en outre pour elle d'introduire une représentation des concepts qui fournit un outil intellectuel de grande puissance.

2,3 Les classes comme représentation des concepts

L'efficacité de la programmation par objets est liée à une certaine ambiguïté de la notion de classe. Nous l'avons introduite comme solution technique au problème de la construction des objets. Mais elle acquiert aussitôt un autre statut, qui ne figurait nullement dans son "cahier des charges", celui d'ersatz informatique de la notion de *concept*, au sens ordinairement donné à ce terme en philosophie.

2,3,1 Du concret à l'abstrait

A l'intérieur de la machine, classes et instances sont également abstraites ou concrètes. C'est notre interprétation qui assigne aux unes un statut d'abstractions (concepts) et aux autres un statut d'objets concrets (instances). Le mécanisme qui sous-tend cette interprétation est de nature analogique : comme le concept abstrait par rapport à ses réalisation concrètes, 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 cette opposition abstrait/concret, vouant les classes au pôle "abstrait".

Ainsi sommes-nous conduits inéluctablement à considérer, par exemple, la classe des points figurée au § 2,2,3 comme la représentation de la notion restreinte de point dont nous avons besoin pour notre projet de tracé de trajectoires.

2,3,2 Extension et intension

En plus de sa signification littérale, qui prescrit comment ses instances sont faites et comment elles se comportent, la classe se voit donc chargée d'une autre valeur sémantique : elle représente l'ensemble de ses instances, effectivement présentes à un moment donné dans le système ou seulement potentielles. Notre classe des points représente donc aussi l'ensemble des points dont nous nous sommes servis, ou dont nous nous servirons, dans l'exécution de notre projet. Pour être précis, nous parlerons de l'*extension* de la classe pour désigner cet ensemble, conformément à l'habitude des logiciens.

Soulignons que ce glissement sémantique se produit de manière inconsciente, comme de nombreux phénomènes langagiers, et qu'il faut 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.

Soulignons aussi que cette ambiguïté est 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), en sociologie (les classes sociales), etc. Pourquoi donc avons-nous repris un terme aussi chargé de sens ? C'est qu'en informatique la notion d'ensemble pose problème. Pour représenter un ensemble infini en termes finis acceptables 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*, néologisme forgé pour s'opposer à *extension*. Par rapport aux classes traditionnelles, qui sont des ensembles, vus comme des extensions, les classes de la programmation par objets sont des programmes, descriptions intensionnelles, les seules choses que nous sachions réaliser avec nos drôles de machines... Et bien entendu, notre pensée passe naturellement de la description à la chose décrite. On voit que l'ambiguïté à laquelle nous nous heurtons est profonde.

2,3,3 Tout discours est nécessairement abstrait, donc général

Muni de cette interprétation de la classe comme abstraction, nous pouvons mieux comprendre la solution qu'elle apporte au problème de la construction des objets. 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'intention 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 à rapprocher de la règle de généralisation en logique du premier ordre : si on a $P(x)$, où P est un prédicat et x une variable libre, alors on a universellement $\forall x\, P(x)$.

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 définireraient par "recopie différentielle" : on y voit toujours reparaître des constructions 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.

2,4 L'héritage : Principe

Pour de nombreux auteurs, la notion d'héritage entre classes est la pierre de touche de la programmation par objets. Force est de constater qu'elle pose quelques problèmes.

2,4,1 Définitions

(a) 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 incluse dans celle de A et si le catalogue des procédures de B est contenu dans celui de A.

(b) Lorsque A hérite de B, on dit aussi que A est une *sous-classe* de B, et que B est une *sur-classe* de A.

(c) La relation d'héritage est une relation d'ordre, elle se représente par un graphe sans circuits appelé *le graphe d'héritage*.

Les termes de sous-classe et de sur-classe se justifient en adoptant le point de vue extensionnel : si on admet que l'appartenance d'un objet à une classe (vue en extension) est caractérisée par la possession de tous les champs et de tous les comportements définis (intensionnellement) par cette classe, avec éventuellement quelques autres attributs supplémentaires, alors il est licite de poser que toutes les instances de la classe-héritière A sont aussi des instances de la classe-héritée B. Dans ces conditions, l'extension de B contient celle de A, on peut donc dire que A est sous-classe de B.

Du point de vue intensionnel, celui des textes des classes, la relation s'inverse : le texte de la sous-classe "contient" celui de la sur-classe. Il est significatif que la terminologie courante reflète le point de vue extensionnel !

En ce qui concerne l'écriture des textes, en pratique on définira une sous-classe A en partant de la sur-classe B supposée déjà connue, et 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 sur-classe.

2,4,2 Exemple : La classe des points colorés comme sous-classe de celle des points *(fig. 3).*

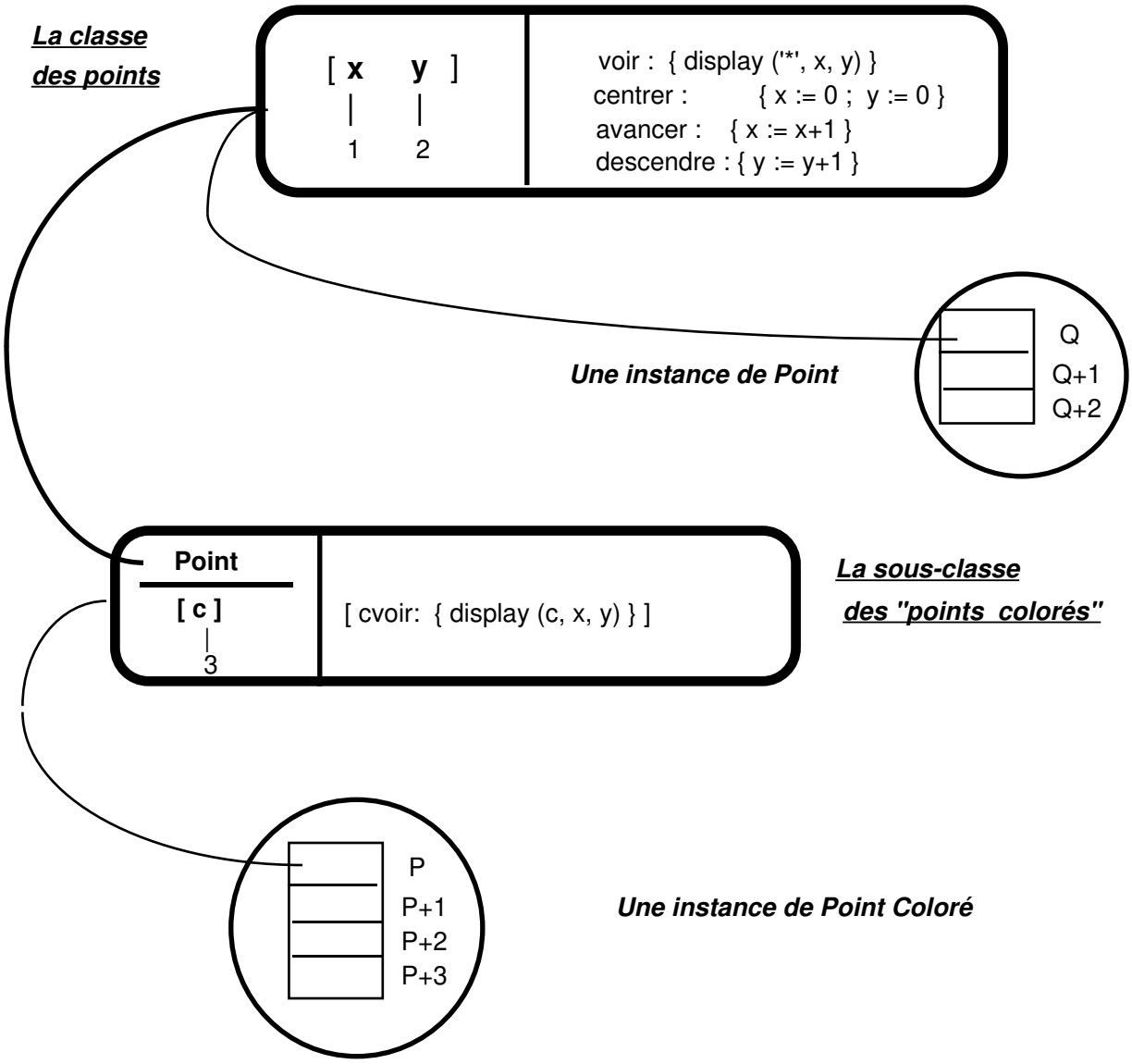

Figure 3.

Un point coloré est un objet plus compliqué qu'un point : il possède une variable supplémentaire *c* et une procédure d'affichage perfectionnée *cvoir* qui montre sa couleur. Par suite, la classe des points colorés hérite de la classe des points. Conformément à la discussion précédente, nous considérons que tout point coloré *est* aussi un point ordinaire, et nous disons donc que la classe des points colorés est une sous-classe de celle des points.

La déclaration de la classe des points colorés se fait comme indiqué, en précisant qu'elle hérite de la classe des points et en donnant seulement le nom de la variable *c* et le couple (nom, texte) de la procédure *cvoir.* On note que la variable supplémentaire prend naturellement le numéro 3 dans la liste complète des champs des instances, ce qui permet de ne rien changer à l'interprétation des messages définis dans la sur-classe. Cette facilité appréciable est liée au fait que la sur-classe est ici unique (on parle en ce cas d'héritage simple) ; en présence de plusieurs sur-classes (héritage multiple) il faut gérer en plus une table indiquant comment se répartissent les champs hérités, ce qui complique la réalisation sans changer le principe. Nous exposerons ci-après le mécanisme de l'héritage simple, et nous reprendrons ensuite la discussion de l'héritage multiple.

2,4,3 Mécanisme d'exécution : le *look-up* et la redéfinition

Pour que toute instance d'une sous-classe puisse être considérée comme une instance de la sur-classe, le mécanisme d'interprétation des messages (§ 2,2,2) doit être complété comme suit :

si le nom de la procédure apparaît dans le texte de la sous-classe, le corps de procédure correspondant est exécuté comme ci-dessus ;

sinon, la recherche se poursuit dans la sur-classe : si le nom de la procédure y figure, le corps de procédure associé est exécuté, sinon la recherche se poursuit dans la sur-classe de la sur-classe (si elle existe), etc, jusqu'à arriver à une classe qui n'a pas elle-même de sur-classe, ce qui déclenche une erreur à l'exécution.

Ce processus de recherche en remontant dans le graphe d'héritage (*look-up*)est très facile à implémenter sous l'hypothèse que l'héritage est simple : l'unicité de la sur-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 entrée unique, qui correspond à une sur-classe générale exprimant les propriétés communes à tous les objets du système, de sorte que le graphe d'héritage est connexe, et que lorque l'héritage est simple ce graphe est un arbre.

On peut immédiatement étendre le procédé au cas où on donne dans la sous-classe une nouvelle définition à un nom de procédure déjà défini dans la sur-classe : en général, pour associer à ce nom un comportement légèrement différent de celui que spécifie la sur-classe, par exemple pour tenir compte d'un attribut supplémentaire. Ainsi, dans notre exemple de la figure 3, au lieu de doter nos points colorés d'un nouveau comportement `cvoir`, on peut souhaiter *redéfinir* la procédure d'affichage `voir` de manière à faire apparaître la couleur du point. Il suffit pour cela de rédiger la sous-classe comme le montre la fig. 4. Conformément au mécanisme de *look-up*, quand on adressera le message `voir` à un point coloré, c'est la nouvelle procédure qui sera exécutée. Les points "ordinaires", bien entendu, ne changent pas leur manière de (se faire) voir.

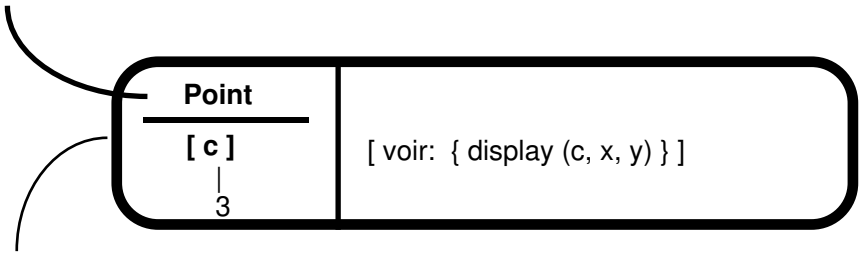

Figure 4.

Cette technique de redéfinition des procédures dans les sous-classes est d'une importance fondamentale. Elle prend souvent la forme extrême où une procédure ne reçoit dans la sur-classe qu'une définition sommaire, explicitement destinée à être raffinée dans les sous-classes par redéfinition, mais suffisante pour que cette procédure soit utilisée par d'autres comportements définis dans la sur-classe, et qui seront hérités sans modifications. De telles procédures sont appelées *virtuelles*, ou *retardées* (*deferred*) suivant les langages. Nous reprendrons cette question plus loin.

2,4,4 Commentaire

Les concepts ne sont pas des entités inertes, ils s'ordonnent naturellement par la relation de généralisation. Nous avons tous une idée intuitive de ce que signifie "tel concept est plus général que tel autre", mais en donner une définition mécanisable, c'est une autre affaire. Les chercheurs en représentation des connaissances, avec leurs réseaux sémantiques et leurs algorithmes de classification, en savent quelque chose. Pour nous le problème se pose en ces termes : la relation de généralisation entre concepts peut-elle se lire dans l'approximation fournie par le texte des classes ? Certainement pas d'une manière générale, comme le montrent des exemples simples : ainsi, en définissant un carré par un point et une longueur (celle des côtés, supposés parallèles aux axes), et un rectangle par deux points (le coin supérieur gauche et le coin inférieur droit), comment "voir" que la classe des rectangles est plus générale que celle des carrés ? Ou encore, comment exprimer que la classe des unijambistes est plus particulière que celle des hommes "ordinaires" ?

Ce qu'on peut exprimer au niveau des classes, c'est *une forme très particulière de généralisation* , que donne la relation "être plus compliqué".

Disons qu'un objet *x* est plus compliqué qu'un objet *y* si *x* possède plus d'attributs et/ou plus de procédures de comportement que *y* (au sens que tous les attributs et toutes les procédures de *y* se retrouvent dans *x*). Pour être tout-à-fait précis, nous sommes amenés à introduire les classes A et B des objets *x* et *y*, et à dire que tous les attributs et toutes les procédures que définit B se retrouvent dans A, c'est-à-dire que A hérite de B, au sens de la définition 2,4,1.

On peut alors considérer que la classe B est plus générale que la classe A. Dans ce cas, en effet, les instances de A *ont toutes les propriétés* des instances de B, et on peut dès lors estimer qu'elles *sont* aussi - à toutes fins utiles - instances de A, ce qui nous ramène à l'inclusion extensionnelle de B dans A. Soulignons que cette inclusion résulte de notre interprétation du mécanisme d'exécution, il est "dans notre tête" et non point dans le code. On peut se contenter d'une lecture intensionnelle du graphe d'héritage, strictement fidèle au texte vu comme la trace du processus historique d'écriture des classes par adjonctions et complications successives. Mais la classification des concepts selon leur généralité est un outil intellectuel tellement puissant, et si bien partagé, que la lecture extensionnelle s'impose à nous.

Nous allons voir que cette interprétation de l'héritage comme une spécialisation donne elle-même lieu à de multiples variations. La question "du bon usage de l'héritage" est loin d'être épuisée.

2,4,4 Le "lien dynamique"

Il faut se débarrasser ici d'un faux problème qui encombre inutilement de nombreux exposés sur les objets, celui de l'opposition entre lien statique et lien dynamique. Il ne se pose que dans la perspective d'un compilateur maladroit prenant au pied de la lettre les indications de type qu'on lui fournit (cf. § 2,6,2). Dans ce paragraphe, nous identifions le *type* d'une variable et la *classe* des objets que cette variable peut contenir comme valeur.

Considérons une variable `v` déclarée de type `C`,où `C` est une classe qui définit une procédure `p` et qui possède une sous-classe `C'` où la procédure `p` est redéfinie. Soit `x`, l'objet

valeur de la variable *v*. à un moment donné de l'exécution. Il peut se faire que cet objet soit une instance de `C'`, d'après l'interprétation extensionnelle de l'héritage. Supposons que ce soit le cas, et qu'on doive envoyer le message `p` à *x*. Clairement, la procédure devant être activée est celle qui résulte de la redéfinition de la procédure `p` dans `C'`, ce qui exige la présence d'une table d'indirection dans le code exécutable au lieu d'un simple transfert. Cette manière de faire, la seule correcte du point de vue sémantique, est appelée traditionnellement le "lien dynamique".

Un compilateur peu scrupuleux pourra se targuer de plus d'efficacité et invoquer un prétendu "lien statique", en n'engendrant pas de table d'indirection et en exécutant directement la procédure de la sur-classe, rendant ainsi la redéfinition inopérante. Qu'importe que ce soit faux pourvu que ça aille vite ! Comme une telle politique est manifestement aberrante, les langages qui la pratiquent permettent au programmeur d'insister en exigeant explicitement la génération de la table grâce à une syntaxe idoine (p.ex., en C++, il faut qualifier la procédure de *virtual* dans la sur-classe !). Nous n'en dirons pas plus sur ce chapitre peu reluisant.

2,5 L'héritage : Mise en œuvre

2,5,1 Les deux aspects de l'héritage.

Revenons d'abord sur les deux aspects intensionnel et extensionnel de l'héritage simple. Du point de vue intensionnel, la branche qui porte une classe dans l'arbre d'héritage exprime la genèse de cette classe au cours d'un processus de développement par raffinements successifs tout-à-fait classique en programmation. Il peut fort bien arriver que la classe résultat de ce processus n'ait plus grand'chose à voir avec telle ou telle classe intermédiaire, par exemple lorsque la construction des nombres aléatoires se fait à partir de la notion de flot : on aboutit à des quantités qui sont intuitivement plus proches des nombres que des fichiers, contrairement à leur phylogenèse, un peu comme l'évolution des cétacés a conduit ces mammifères à ressembler à des poissons.

Au contraire, le point de vue extensionnel exprime que toute sous-classe représente un concept qui spécialise celui de chacune de ses prédécesseurs le long de la branche. Attention à ne pas ranger les baleines parmi les poissons ! Mais il faut surtout prendre garde à ce que, comme nous venons de le dire, l'héritage ne permet de traiter qu'une notion de généralisation / spécialisation très particulière, celle qui provient de l'adjonction d'attributs et/ou de procédures.

Les difficultés viennent de ce que, poussés par nos habitudes de pensée, nous sommes tentés de demander à la relation d'héritage bien plus qu'elle ne peut donner. On voit souvent les débutants en programmation par objets invoquer l'héritage dès qu'une relation d'ordre apparaît dans le problème traité, même lorsque cette relation n'a rien à voir avec la généralité des concepts en jeu, par exemple lorqu'il s'agit d'une hiérarchie de parties.

2,5,2 Classes abstraites, méthodes virtuelles, etc.

On peut lire l'arbre d'héritage d'une autre manière, en considérant que chaque sur-classe possédant plusieurs sous-classes résulte de la factorisation des éléments communs à toutes ses sous-classes. Cette "mise en facteur commun" peut résulter d'un plan délibéré, dans un développement descendant, mais aussi bien d'une constatation *a posteriori*, qui aura

conduit à un remaniement de l'écriture de classes déjà réalisées. Il arrive alors souvent que la sur-classe ait un caractère inachevé, en ce sens qu'elle ne contient pas assez d'informations pour spécifier des instances "viables" puisque précisément les informations complémentaires doivent être apportées dans ses sous-classes. Ce genre de situation est extrêmement fréquent : par exemple, il n'existe pas de mammifère qui soit seulement un mammifère, il sera nécessairement instance d'une espèce particulière de mammifère, chien, chat, vache, cochon...

Une telle classe "incomplète", qui ne doit pas créer d'instances mais seulement servir de base à des héritages, est appelée *abstraite* dans la tradition de SMALLTALK et de C++, *virtuelle* en SIMULA et *retardée* (anglais : *deferred*) en EIFFEL. Ce qualificatif s'oppose à "concret" qui s'applique aux classes "ordinaires" capable d'engendrer des instances en état de marche. Grâce à l'analogie avec des concepts généraux comme celui de mammifère, cette notion de classe abstraite est d'un emploi très naturel et intuitif.

Les choses se compliquent lorsque le compilateur prétend vérifier que les classes abstraites ne sont effectivement pas instanciées. Il faut alors introduire une syntaxe spéciale pour marquer ces classes. D'abord une syntaxe pour permettre d'introduire des noms de procédures avec leur profil sans leur attribuer de corps (formes `is deferred` en EIFFEL, et *pure virtual,* i.e. `virtual...=0` en C++). La présence d'une telle forme dans le texte d'une classe suffit à la dénoncer comme abstraite auprès du compilateur. EIFFEL demande en outre, pour plus de sûreté, que le programmeur confirme ses intentions en qualifiant la classe elle-même de `deferred`. Tout ce dispositif doit s'entendre dans une perspective de contrôle statique, qui permet par exemple au compilateur de s'assurer dans les sous-classes réputées concrètes que toutes les procédures ont bien fait l'objet d'une (re)définition avant toute instanciation.

C'est dans ce contexte que le polymorphisme évoqué en 2,2,4 prend toute sa puissance. L'emploi de procédures retardées permet de fixer toute une terminologie, et d'exprimer certaines relations de manière définitive au niveau d'abstraction convenable, les (re)définitions ultérieures devant se conformer au cadre ainsi tracé. On obtiendrait le même résultat avec des langages comme SCHEME ou ML en utilisant des fonctionnelles d'ordre supérieur, de sorte que la technique des classes abstraites peut se voir comme un palliatif à l'absence d'une sémantique des valeurs fonctionnelles dans les langages courants.

Une autre interprétation de cette technique permet de simuler dans une certaine mesure la généricité dans les langages qui en sont dépourvus comme SMALLTALK ou SIMULA (cf. § 2,6,3).

2,5,3 Héritage simple vs héritage multiple

Le mécanisme d'interprétation de l'héritage des classes par *look-up* est particulièrement simple à réaliser lorsque chaque classe possède une seule sur-classe. Cette limitation n'est pas choquante si on adopte le point de vue intensionnel : l'arbre d'héritage reflète fidèlement le processus de construction par adjonctions successives d'attributs et de procédures. Elle devient difficile à admettre si on prend le point de vue extensionnel, car elle revient à imposer que nos classifications ne soient que des taxinomies, à l'image de la classification des êtres vivants en embranchements, classes, ordres, familles, genres, espèces, sous-espèces et variétés.

L'héritage multiple supprime cette limitation, en autorisant chaque classe à avoir plusieurs sur-classes. Du point de vue intensionnel, on obtient la possibilité de construire des classes par composition de leurs sur-classes, par blocs et non plus seulement par adjonction d'éléments (ce qui reste toujours possible). Du point de vue extensionnel, les classes ressemblent encore plus à des concepts, susceptibles d'entrer dans des relations de spécialisation complexes, visualisées par un graphe d'héritage qui n'est plus un arbre, mais qui reste sans circuit.

L'algorithme d'interprétation demande alors quelques éclaircissements : *quid* lorsque la même procédure - ou plutôt le même nom - se retrouve dans plusieurs sur-classes ? Comment résoudre l'ambiguïté qui apparaît alors dans le processus de *look-up* ?Il y a deux réponses, suivant les langages.

Les uns (au premier rang desquels CLOS) spécifient un ordre total dans le parcours du graphe d'héritage, revenant ainsi à un choix par précédence, comme dans l'héritage simple. Les critères définissant cette *linéarisation du graphe d'héritage* font l'objet de toute une littérature.

Les autres suppriment le problème à sa racine, en exigeant du programmeur qu'il lève lui-même l'ambiguïté soit en renommant la procédure en cause dans la sous-classe (cas d'EIFFEL), soit en indiquant explicitement à quelle sur-classe on se réfère lors de chacun de ses emplois (cas de C++).

L'emploi le mieux maîtrisé de l'héritage multiple correspond au cas où les sur-classes emploient des vocabulaires disjoints. Plus précisément, l'usage a consacré le terme de "mixin" pour désigner une classe abstraite définissant une fonctionalité "orthogonale" à la signification principale de la classe. On procède alors en écrivant d'abord l'essentiel d'une classe, avec les héritages nécessaires, puis en la complétant en la faisant hériter en plus d'une ou plusieurs "mixins" préparées à part.

La qualité de la communication et du consensus obtenu sur le texte des programmes est une préoccupation centrale en programmation par objets. La puissance du polymorphisme tient entièrement à l'économie de pensée qu'il autorise et à la généralité de ce phénomène. Cet aspect consensuel sous-tend la réussite des bibliothèques de classes prédéfinies : quelles que puissent être leurs divergences sur le détail de leur réalisation, les programmeurs sont instinctivement d'accord sur ce que représente une classe, et sur l'usage qu'on en peut faire, sans avoir aussitôt envie de la réécrire.

2,6 Notions apparentées

Comme nous l'avons dit plus haut (2.2.1), l'émergence de la programmation par objets a rencontré, puis absorbé, des courants de pensée provenant d'autres sources, comme la représentation des connaissances, la théorie de la programmation et la pratique du génie logiciel. Ces courants charrient avec eux un vocabulaire touffu, qui recouvre des préoccupations qu'il faut absolument expliciter, pour voir à quel point une réalité informatique peut être chargée d'intentions différentes. Voici trois termes qui apparaissent très souvent, on pourrait allonger la liste (métaclasses, concurrence, etc).

2,6,1 Encapsulation

C'est l'un des mots-clés qui reviennent régulièrement lorsqu'on parle de programmation par objets. Il recouvre l'idée qu'il est sain pour le programmeur de voir l'objet comme un tout *fermé,* dont la structure interne doit être cachée à l'observateur extérieur. Cette idée est également connue sous le nom bizarre d'*abstraction* (dans *abstract data type*). Elle oppose l'interface permettant de manipuler l'objet, connue de l'extérieur, à l'implémentation "interne". Ce n'est nullement une nécessité logique, mais une préoccupation propre au génie logiciel, nous ne la discuterons pas ici. Notons qu'elle suppose en premier lieu la distinction entre un espace intérieur et un espace extérieur : pour ce qui nous concerne, l'intérieur est en général le texte de la classe, l'extérieur le reste du texte du programme.

Comme l'objet fermé doit néanmoins rester manipulable de l'extérieur, on ne peut pas tout cacher. Les langages divergent sur la distinction entre ce qui doit être caché et ce qui doit être divulgué, et sur les modalités de la chose. SMALLTALK considère que les noms d'attributs sont systématiquement cachés, et les noms de procédures divulgués. EIFFEL version 2, reprenant la tradition de CLU, donne une liste explicite de noms d'attributs et de procédures divulgués, les autres étant cachés. EIFFEL version 3 introduit un dispositif syntaxique élaboré dans le même but. C++ introduit de subtiles gradations, liant de plus les modalités de la divulgation à celles de l'héritage. A travers toutes ces variations, l'intention demeure, la même qu'en CLU et en ADA.

Notons au passage que ce désir de cacher la structure interne des objets aux regards indiscrets rend la tâche plus difficile à ceux qui souhaitent non pas utiliser lesdits objets, mais raisonner à leur sujet. Les éléments accessibles de l'extérieur sont en général des procédures de mise en œuvre, qui offrent peu de prise au raisonnement - en l'état actuel des techniques, du moins. Aussi les langages à objets dont nous traitons ici n'ont-ils que peu de succès en intelligence artificielle auprès des spécialistes de représentation des connaissances, qui leur préfèrent des formalismes issus des modèles de *frames*. Nouis reprendrons ce point en 4,2,2.

2,6,2 Types et typage

Le problème du typage en programmation par objets se pose d'une manière originale du fait de la présence des classes, qui fournissent un système naturel de types. Si bien qu'un langage comme SMALLTALK peut se targuer d'un contrôle de types dynamique, à la différence de langages comme LISP ou SCHEME.

Les problèmes se posent lorsqu'on veut effectuer un 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, principalement dûes à l'interprétation de l'héritage en termes de sous-types, comme le savent bien les théoriciens et comme l'ont illustré les discussions autour du système de types d'EIFFEL

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". En effet, considérons un fragment de code exprimant une opération qui doit pouvoir s'appliquer à des

instances de 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. A 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 y 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*.

2,6,3 Généricité

Ce mot et son associé, l'adjectif "générique", est aujourd'hui mis à toutes les sauces, souvent comme simple ersatz de *général* et de *généralité*. Dans le contexte des langages de programmation, du moins, on peut lui donner le sens précis de "généralisation au moyen d'un paramètre de type", conformément à la tradition inaugurée par CLU et par ADA, aujourd'hui poursuivie par EIFFEL et par C++ : l'exemple traditionnel est la pile générique définie comme `Pile[T]` et qui se particularise en `Pile[Integer]`, `Pile[Char]` ou `Pile[Real]`. Ce mécanisme-là est résolument étranger au modèle à classes et instances. Il est d'ailleurs, si on veut bien y penser, beaucoup plus difficile à élaborer. En particulier, il n'est concevable que dans un langage possédant une théorie des types.

Ce qu'offre la programmation par objets, c'est la relation d'héritage, qui peut être employée pour *simuler* la généricité, comme on l'enseigne dans les cours de programmation en SMALLTALK. Si on veut avoir les deux mécanismes simultanément, comme en EIFFEL, la situation devient intéressante (avec la *généricité contrainte*), mais complexe, et l'utilisateur rencontre parfois des problèmes épineux. C'est qu'on a énormément compliqué le modèle : l'utilisateur a-t-il encore une claire perception de la manière dont il est implémenté ?

Voici un exemple très simple montrant combien les exigences de typage cohérent en présence de généricité 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, `U` héritant de `V`, alors le type `A[U]` est sous-type de `A[V]` (hypothèse dite de covariance). Supposons que la classe `A[X]` soit celle de la pile générique, et possède une procédure `push(`*x*`)` à un argument qui doit être de type `X`. Soit *p,* instance de `A[U]`, une "pile de `U`". Si *v* est une instance de `V` qui n'est pas aussi instance de `U`, l'exécution de "*p* `.push(`*v*`)`" doit provoquer une erreur ; mais il suffit que l'objet *p* soit logé dans une variable *q* qui a été déclarée de type `A[V]`, ce qui est licite en vertu du sous-typage, pour que cette opération, dès lors libellée "*q* `.push(`*v*`)`" devienne innocente aux yeux du contrôleur de types... Faut-il abandonner l'hypothèse de covariance ?

3. LANGAGES

Dans cette section, nous ne prétendons pas offrir une introduction à chacun des langages mentionnés, mais seulement montrer comment les principes généraux exposés dans la section précédente se réalisent dans chacun d'entre eux - en particulier, quelle est la

terminologie employée pour désigner les différents composants du système. En conséquence, nous avons renoncé à toute indication sur la syntaxe et, par suite, à donner des exemples. Les langages officiellement reconnus comme "à objets" se répartissent en deux groupes, ceux où toutes les données ont le statut d'objet et ceux qui ont été conçus comme des extensions de langages procéduraux classiques, et qui offrent donc la possibilité de combiner l'utilisation des objets avec la programmation traditionnelle dans le langage de base. Nous traiterons d'abord des langages du premier groupe, en commençant par SMALLTALK, car il est le plus simple et le plus proche du modèle théorique esquissé ci-dessus, suivi par EIFFEL, qui s'en distingue sur plusieurs points et notamment par l'adoption d'un système de typage statique. Dans le second groupe, nous passerons en revue SIMULA, en raison de son rôle historique, et C++ vu sa diffusion actuelle. Nous ne parlerons pas de la version de PASCAL à objets (Turbo-Pascal v. 6), car elle nous amènerait pour l'essentiel à répéter l'exposé sur C++.

3,1 SMALLTALK

Développé de 1970 à 1980 au centre de recherches de Xerox à Palo Alto. Il en existe actuellement plusieurs versions sur le marché, de SMALLTALK-80, issu de la maison mère, à SMALLTALK-V qui débuta sa carrière sur IBM-PC. Par sa simplicité et par la qualité de l'environnement standard - qui varie suivant les versions - il représente l'exemple le plus achevé de programmation par objets.

3,1,1 Instanciation : Objets, classes, métaclasses

En SMALLTALK les attributs s'appellent *variables d'instance*, les procédures *méthodes* et leurs noms *sélecteurs*. L'encapsulation s'obtient en rendant accessibles de l'extérieur toutes les procédures, mais aucun des attributs.La présence d'une valeur nulle universelle `nil` permet de distinguer clairement la création d'un objet (sous la forme d'un objet vide) de son initialisation qui doit le rendre "opérationnel".

Le système classes-instances de SMALLTALK ne diffère du modèle que nous avons présenté en 2,2 que par l'adjonction de *variables de classe*, qui sont des variables déclarées dans la classe et partagées par toutes ses instances. Le trait le plus intéressant du système est de considérer les classes elles-mêmes comme des objets, instances de classes spéciales appelées *métaclasses*. Du coup il devient possible de doter les classes de leurs propres variables d'instance et de leur propres méthodes - notamment des méthodes pour créer des instances de diverses manières - ce qui rend de multiples services. Dans ces conditions, l'opposition terminologique entre variables de classe et variables d'instance est malencontreuse puisque les variables de classe ne sont nullement les variables d'instance de la classe en tant qu'objet...

3,1,2 Héritage

L'héritage en SMALLTALK est simple. La redéfinition des méthodes se fait sans cérémonie. Les classes abstraites ne se différencient pas des classes normales par la syntaxe. Pour exprimer la virtualité d'une méthode on lui donne comme corps la formule standard `self subclassResponsibility.`

En cas de redéfinition, la méthode de la sur-classe reste accessible grâce au mot-clef `super`.

3,1,3 Bibliothèque standard

Le langage sert de base à une très riche bibliothèque standard de plusieurs centaines de classes, où figurent un certain nombre de constructions puissantes qu'on ne trouve pas dans les autres langages à objets:

- la notion de bloc (classe `BlockContext`) qui permet de capturer sous forme d'un objet un fragment de code dans son environnement courant. C'est l'analogue exact des fermetures dans les langages fonctionnels. On provoque l'exécution d'un objet bloc en lui envoyant le message `value`. Grâce à cette notion, les structures de contrôle habituelles se réduisent à des messages ordinaires définis dans les classes standard `Boolean` et dans toutes les sous-classes de `Collection` ;

- un bloc peut voir son exécution suspendue et reprise par un mécanisme de file d'attente, il devient alors un `Process`;

- la notion de symbole, héritée de LISP, bien distincte de celle de chaîne de caractères, permet un certain degré de réflexivité : on peut ainsi manipuler les sélecteurs de méthodes, et par le message standard `perform`, exécuter une méthode dont le sélecteur a été calculé. Les symboles apparaissent aussi pour donner des noms aux classes, et pour créer un environnement global très commode à utiliser : à sa création, chaque classe est donnée comme valeur (puisqu'elle est elle-même un objet de plein exercice) à une variable globale qui a le même nom qu'elle.

3,1,4 Environnement de programmation

Le développement historique de SMALLTALK est inséparable de celui des écrans BitMap. L'interface utilisateur graphique de SMALLTALK-80 a fortement marqué son époque, avec l'idée géniale du `Browser`, instrument de communication bidirectionnelle entre l'utilisateur et le système.

Cette interface utilisateur est bâtie sur une philosophie des interfaces bien définie, connue sous le nom de MVC (pour Modèle, Vue, Contrôleur, qui désignent les trois composants de l'interface). Les structures de base de la technique MVC sont disponibles dans la bibliothèque standard de SMALLTALK-80 sous la forme de trois classes abstraites et de leurs sous-classes, prêtes à être réutilisées par le programmeur d'applications.

3,2 EIFFEL

Langage conçu par Bertrand Meyer en 1986 comme outil de génie logiciel employant à fond l'approche par objets, sans perdre les acquis de la conception d'ADA. C'est un langage à classes et instances fortement typé, avec héritage multiple et généricité. La première version à faire une carrière commerciale, EIFFEL 2, est aujourd'hui remplacée par la version 3, dont le standard est géré par le *Nonprofit International Consortium for EIFFEL* (NICE).

3,2,1 Instanciation : Objets et classes

En EIFFEL les attributs et les procédures sont uniformément appelés *traits* (en anglais *features*).L'idée est que l'utilisateur ne doit pas distinguer les accès directs aux champs et les méthodes sans arguments, la différence étant une question d'implémentation (règle de la *référence uniforme*). L'encapsulation est explicite et la syntaxe permet d'ndiquer précisément où tel et tel *trait* est utilisable.

Les classes sont vues comme des types (plus précisément, comme des implémentations de types abstraits). Elles ne sont pas des objets ordinaires : l'auteur en fait même une question de principe, car selon lui *"Classes do not need to exist at run-time. ... Classes are not conceptually needed during execution"*.

Cette attitude est cohérente avec le refus de toute entité globale : l'exécution d'un programme EIFFEL résulte de la pseudo-création d'une instance d'une classe-racine, qui introduit en cascade tout l'univers du programme. Il en résulte une parfaite homogénéité, mais un certain nombre de rigidités. Le langage prévoit différentes constructions pour contourner ce problème, par exemple les fonctions `once`.

3,2,2 Héritage et contrôle de types

EIFFEL offre un héritage multiple, et résout les problèmes de conflits entre les noms d'attributs qui apparaîtraient dans plusieurs sur-classes en exigeant que les noms en question soient changés, à moins qu'ils ne proviennent d'un ancêtre commun *(repeated inheritance)*. De même la redéfinition d'une méthode doit être annoncée. Ces changements de nom permettent également d'adapter la terminologie des sur-classes à celle de la sous-classe.

La compatibilité entre le type déclaré d'une variable et celui de l'objet qu'elle contient (plus exactement, dont elle contient l'adresse) repose sur l'héritage : la classe de l'objet doit être sous-classe de la classe déclarée. Dans le cas de classes génériques, elle exige aussi la compatibilité des types-paramètres.

3,2,3 Assertions, exceptions, invariants

EIFFEL permet de donner aux procédures des *préconditions* et des *postconditions* , introduites respectivement par les mots-clés `require` et `ensure` suivies d'*assertions ,* qui sont essentiellement des séquences d'expressions booléennes. Ces assertions sont vues comme un *contrat* liant l'implémenteur et l'utilisateur. Elles peuvent aussi faire l'objet d'un contrôle à l'exécution, grâce à des options de compilation. Les erreurs détectées peuvent être récupérées par un mécanisme d'*exceptions,* comme en PL/I, CLU et ADA.

A un niveau plus global, chaque classe peut être munie d'un *invariant* , qui est une assertion portant sur les attributs qui doit être vérifiée à chaque instant par chaque instance de la classe. L'invariant d'une classe est automatiquement valable pour toutes ses sous-classes.

Enfin, suivant la tradition de Simula et de Smalltalk, Eiffel comporte une *bibliothèque standard* où sont définies les structures de données usuelles : tableaux, listes de diverses sortes, arbres.

3,3 SIMULA

Le doyen des langages à objets a été défini par Dahl et Nygaard au Centre de Calcul Norvégien (Oslo, 1967) comme extension d'ALGOL-60 en vue d'applications à la simulation. Il est toujours vivant, avec un centre de développement à l'Université de Lund (Suède), et commercialisé par plusieurs sociétés scandinaves.

3,3,1 Instanciation : Objets et classes

Les classes de SIMULA ont gardé quelques vestiges de leur forme originale de procédures, où les variables locales sont devenus les attributs et les procédures locales définissent le comportement des instances (cf. § 1,2,3): elles peuvent recevoir des paramètres, et chacune possède une séquence d'instructions exécutables, résidu du corps de procédure. Cette séquence s'exécute à la création de chaque instance, dans l'environnement de l'instance: on peut la voir comme une séquence d'initialisation.En fait, dans l'intention des concepteurs, cette séquence sera interrompue, puis reprise, par un mécanisme de coroutines (instructions `detach` et `resume`).

Rappelons qu'en ALGOL-60 la notion de "type défini par l'utilisateur" ne s'était pas encore imposée (il faut attendre PASCAL). Les classes ne sont donc pas directement interprétées comme des types, mais presque. Par rapport aux valeurs "ordinaires" provenant d'ALGOL-60, les objets ont un statut particulier : les instances d'une classe `C` sont de type "référence de `C`" `ref(C)`, et il faut un opérateur d'affectation spécial "`:-`" pour les manipuler. La création d'une instance se fait par une procédure standard `new` appliquée à la classe : une déclaration comme

```
ref(Machin1) x
```

(qui énonce que la variable `x` est du type des instances de la classe `Machin1`) autorise l'affectation spéciale :

```
x :- new Machin1.
```

Et bien entendu, en 1967 il n'était pas question de considérer les classes comme des objets !

3,3,2 Héritage

En SIMULA l'héritage est simple. Il prend la forme syntaxique particulière de la *préfixation*: pour exprimer qu'une classe `C2` hérite d'une classe `C1`, on fait précéder la définition de `C2` par la mention (du nom) de `C1`. Par exemple, une des plus célèbres classes de l'univers SIMULA s'appelle `Simulation` (elle définit tous les mécanismes nécessaires à la gestion de processus concurrents), et d'innombrables classes en héritent grâe à des définitions du genre

```
Simulation class Machin1...
Machin1 class Machin2 ...
```

En fait, SIMULA interprète l'héritage comme mettant le matériel défini dans la sur-classe (variables et procédures) à la disposition de la sous-classe. Dès lors, on peut généraliser, et envisager de faire hériter d'une classe `C` une séquence d'instructions anonyme *s* (plus précisément, un bloc au sens d'Algol-60), en écrivant le nom de la classe `C` en tête de la séquence *s* : cela signifie que les déclarations et les instructions composant *s* peuvent utiliser le matériel de `C`. En revanche, comme la préfixation utilise le nom de la sur-classe, on ne peut hériter que d'une entité nommée, en l'occurrence une classe et non un bloc.

La syntaxe de SIMULA n'autorise à écrire qu'un seul nom comme préfixe d'une classe (ou d'un bloc), l'héritage simple est ainsi imposé sans discussion. Mais il serait facile d'étendre cette syntaxe à l'héritage multiple, en autorisant une liste de noms (séparés par des virgules, par exemple, à la manière de C++).

Comme SIMULA est un langage typé, son compilateur cède à la tentation d'utiliser le lien statique (cf. § 2,4,4) hélas ! Pour l'en dissuader, il faut employer une syntaxe spéciale

mentionnant explicitement la classe dont on veut faire exécuter la procédure grâce à la notation `qua` dont voici le principe :

Supposons que la méthode `voir` soit définie dans la classe `Machin1` et redéfinie dans sa sous-classe `Machin2`, et considérons les déclarations suivantes :

```
ref(Machin1) x ;  ref(Machin2) y ;
```

suivies des instructions :

```
y :- new Machin2 ;      *** normal ***
x :- y                  *** compatible ***
```

L'appel *x.voir* (notation pointée, § 2,1,2) utilisera la méthode `voir` de la sur-classe `Machin1`, l'appel *y.voir* celui de la sous-classe `Machin2`, conformément aux déclarations, bien qu'il s'agisse en fait du même objet dans les deux cas ! Pour que *x* (resp. *y*) puisse utiliser la méthode de la sous-classe (resp. de la sur-classe), il faut écrire

```
x qua Machin2.voir
```
(resp. `y qua Machin1.voir`)

3,3,3 Bibliothèque standard

SIMULA a été le premier langage à apporter avec lui une substantielle bibliothèque de classes prédéfinies permettant d'étendre le langage grâce à l'héritage. En l'occurrence, cette extension prend un caractère exemplaire et mérite qu'on s'y attarde.

Rappelons que Dahl et Nygaard, ses auteurs, voulaient construire un langage propre à la simulation de processus temporels discrets. Une première version du langage, publiée en 1966, atteste clairement de leurs préoccupations. Pourtant, on chercherait vainement, dans la description de SIMULA, une allusion quelconque à la simulation à événements discrets... De spécialisé qu'il était, le langage est devenu universel, exprimant, comme nous l'avons vu, les opérations générales de la programmation par objets, et elles seules. Le but initial aurait-il été perdu de vue ? Certes non, mais au lieu d'être figées dans des primitives du langage, les opérations de base de la simulation à événements discrets sont construites explicitement *au moyen du langage,* sous la forme du matériel contenu dans la classe standard `Simulation` déjà mentionnée. Dès lors, il suffit pour les mettre en œuvre de préfixer par `Simulation` la classe ou le bloc devant les utiliser.

C'est un changement d'une portée considérable. La classe en question peut être utilisée telle quelle, comme si elle fournissait une batterie de primitives, sans plus. Même à ce niveau la possibilité de lire le code la définissant vient opportunément compléter et préciser la documentation disponible, permettant un usage plus sûr, parce que mieux informé. Mais surtout l'analyse de ce code permet de l'imiter, de l'adapter en tout ou en partie à des besoins légèrement différents, bref de mettre en œuvre la somme de connaissances qu'il représente autrement que sous sa forme exécutable. Cette nouvelle manière d'utiliser - ou plutôt, comme on dit maintenant, de réutiliser - des programmes est en matière de génie logiciel la grande innovation de la programmation par objets. Nous y reviendrons au § 4,2,3.

3,4 C++

Malgré son nom, et le fait qu'il ait été développé aux Bell Labs, C++ n'est pas une stricte extension de C. C'est un langage proche de C, mais qui n'est pas tout-à-fait compatible avec C standard. En outre, il est *directement compilé* , sans passer par le compilateur C. Il existe

aujourd'hui un nombre considérable de compilateurs C++, avec des bibliothèques d'importance et de qualité diverses.

3,4,1 Instanciation : Objets et classes

En C++ les éléments déclarés dans une classe s'appellent ses *membres* (terminologie usuelle pour parler des *structures* de C : la classe est vue en C++ comme une variante de la structure). Les attributs sont les *data members*, et les procédures les *member functions.*

La création et l'initialisation des instances sont confiées à une sorte très particulière de fonctions membres (déclarées sans résultat), les *constructeurs*. Une même classe peut posséder plusieurs constructeurs, qui portent tous le même nom, à savoir celui de la classe, grâce au mécanisme de surcharge hérité de C. Ils doivent donc se différencier par le nombre et les types de leurs arguments.

L'encapsulation est tout particulièrement élaborée. De ce point de vue, les classes de C++ apparaissent comme une variante syntaxique des *packages* d'ADA. Tous les membres d'une classe sont réputés privés (`private`), sauf mention expresse des qualificatifs `public` et `protected` (voir § suivant). Les procédures d'accès (*member functions*) se définissent dans un fichier à part, en les préfixant par le nom de la classe.

3,4,2 Héritage

Il s'appelle ici *dérivation,* la sous-classe est la *classe dérivée* et la sur-classe la *classe de base* . Il se marque dans la syntaxe des déclarations par une forme de préfixation, analogue à celle de SIMULA, mais adaptée à l'héritage multiple : la liste des noms des sur-classes est insérée entre le nomde la sous-classe et son corps : pour dire que la classe `Chose` hérite des classes `Machin1`, `Machin2` et `Machin3`, on écrira

```
class Chose : Machin1, Machin2, Machin3
      {...corps de la classe Chose...} ;
```

.

Il est affecté de deux subtilités supplémentaires relative à l'encapsulation.

(1) Les attributs publics de la sur-classe seront-ils publics ou privés dans la sous-classe ? Cet aspect fait l'objet de déclarations explicites. Par défaut, tous les attributs de la sur-classe sont privés dan la sous-classe. Pour obtenir que ce qui était public dans une sur-classe le reste dans la sous-classe, il faut affecter le nom de la sur-classe en question de la mention `public` :

```
class Chose : Machin1, public Machin2, Machin3
      {...corps de la classe Chose...} ;
```

(2) La déclaration de la sous-classe est vue comme un espace extérieur par rapport à la déclaration de la sur-classe (cf. § 2,6,1), et par suite les attributs privés de la sur-classe ne sont pas disponibles pour définir la sous-classe, ce qui peut être fort gênant. Pour ne pas être obligé de rendre publics les attributs nécessaires, cequi pourrait trahir les intentions du programmeur, il convient de les déclarer `protected` dans la sur-classe. L'écriture d'une classe n'est donc pas indépendante des projets d'héritage que l'on caresse à son endroit...

Il existe aussi en C++ un moyen de violer l'abstraction des données en permettant à certaines fonctions et à certaines classes d'accéder à tous les éléments d'une classe : il suffit de déclarer que ces fonctions ou classes sont *amies* (`friends`) de la classe ainsi rendue accessible. Cette déclaration se fait dans le corps de la classe visée. Attention, les fonctions

"amies" sont des fonctions ordinaires, tandis que les fonctions "membres" sont des *méthodes,* qui s'adressent toujours à un destinataire de message. Une classe `A` amie d'une autre classe `C` aura toutes ses fonctions-membres amies de `C`.

3,4,3 Bibliothèque standard

Un des inconvénients majeurs de C++ est la multiplicité de ses réalisations. Seule la définition du langage est standardisée (standard ANSI). Les différents compilateurs sur le marché ne réalisent le plus souvent qu'une partie du standard publié. Par exemple, les classes génériques - au sens de la généricité du § 2,6,3 - sont appelées `templates` dans le document officiel, mais elles sont encore rarement implémentées. Au moins peut-on penser que l'avenir verra les compilateurs en vente se rapprocher du standard. C'est ce qui s'est passé pour l'héritage multiple, qui ne figurait pas dans le langage à l'origine et n'a été effectivement disponible que bien après son introduction officielle.

Mais l'hétérogénéité est encore plus grande en matière de bibliothèques de classes prédéfinies. Dès lors, aucune de ces bibliothèques, quels qu'en soient les mérites, ne s'impose comme une référence (par exemple, dans le cadre des nombreux enseignements de C++ qui tentent de répondre à la demande insustrielle). Il en résulte que C++ est encore largement perçu comme un langage - au demeurant fort complexe - et non comme un environnement de programmation. Du coup, la culture véhiculée par C++ reste assez peu pénétrée des idées de réutilisation de composants logiciels, contrairement à la culture Smalltalk.

4. MISE EN ŒUVRE

Dans cette dernière partie, nous examinons comment l'outil informatique esquissé ci-dessus se comporte vis-à-vis des exigences de ses utilisateurs. Les difficultés sont de deux ordres, celles qui sont liées aux limitations intrinsèques du modèle, et celles qui viennent de demandes sans rapport avec le modèle.

4,1 Limitations du modèle

Il est important de répéter les limitations inhérentes au modèle à classes et instances, de manière à rester bien conscient de la différence entre ce qui s'exprime directement dans ce cadre et ce qui nécessite le recours à des techniques particulières et la construction de superstructures complexes. Répétons que la programmation par objets permet en général d'aborder beaucoup plus facilement des problèmes difficiles, mais que certains d'entre eux lui résistent aussi bien qu'à la programmation classique. Il ne faut pas tout lui demander !

Nous n'essaierons pas de caractériser ici ces problèmes rebelles. Nous nous contenterons de souligner trois aspects du modèle à classes et instances qui apparaissent souvent comme des limitations qu'on aimerait surmonter. Les solutions connues à ces difficultés sont partielles et onéreuses. Il faut en être prévenu ! *Caveat emptor...*

4,1,1 Rien que des attributs et des procédures

On souhaiterait bien souvent pouvoir employer des entités plus évoluées, comme des *contraintes* (pour pouvoir, par exemple, déclarer que la classe `Carré` est sous-classe de la classe `Rectangle` avec la contrainte "`longueur = largeur`"). En ce cas, il faudra les

construire soi-même - ou les acheter -, le modèle à objets ne les fournit pas. Ce problème important subit actuellement une attaque en règle de la part des chercheurs.

De même, dans l'analyse de systèmes importants on désire presque toujours exprimer qu'un objet complexe est composé de sous-objets, eux-mêmes complexes, etc., ce qu'on appelle une *hiérarchie de parties.* Ce problème bien naturel soulève d'épineuses difficultés techniques, et il n'y a pas actuellement de façon standard pour représenter ce type de hiérarchie en programmation par objets. Ce n'est pourtant pas faute d'avoir essayé !

On voudrait aussi, dans de nombreux cas, analyser la complexité d'un objet en disant qu'on peut l'envisager sous *différents points de vue.* Là non plus, aucune technique standard n'est disponible, malgré des années de recherches, et il faut se forger sa propre solution. Ce problème apparaît en général couplé avec le précédent.

Observons ici que le modèle standard ne permet même pas d'exprimer le lien de spécialisation qui résulte de l'affectation d'une valeur à un attribut : étant donné une classe `Roue`, munie d'un attribut `diamètre`, comment en déduire la classe des roues de diamètre 20 ? A partir d'une classe, nous ne savons que créer des instances ou compliquer sa structure via héritage. La multiplicité potentielle des roues de diamètre fixé échappe à notre formalisme...

4,1,2 Obligation de conceptualiser

Le modèle à classes et instances exige que tout objet soit instance d'une classe. Il interdit donc la création d'objets dont on ne connaîtrait pas la classe, ou pour lesquels on jugerait inutile de la définir, par exemple si un tel objet ne doit exister qu'en un seul exemplaire et s'il doit apparaître dès la création du système. Le modèle ne fait point de place à l'objet singulier, à l'*unicum.* Or de tels objets sont souvent fort utiles. Cette exigence d'abstraction conduit à des distorsions variées suivant les systèmes, qui utilisent des subterfuges divers pour définir les objets singuliers dont ils ont besoin.

Symétriquement, un objet n'est instance que d'une seule classe. Le contraire, à savoir qu'un objet puisse appartenir à plusieurs classes simultanément, n'est pourtant pas contraire à l'intuition (*la voiture de Jacques* appartient à la classe `Voiture` et à la classe `PossessionsDeJacques` ; plus généralement, il suffit d'avoir des classes dont les extensions ont une intersection non vide). On parle alors de *multi-instanciation.* Mais cela sort de notre cadre d'implémentation, qui bon gré mal gré nous impose de faire apparaître (fût-ce de manière cachée) la classe `VoituresPossédéesParJacques`, sous-classe des deux précédentes par héritage multiple (et représentant leur intersection), dont l'objet visé sera instance.

Et que se passe-t-il si on ne sait vraiment pas à quelle classe appartient un objet donné, par exemple si on veut justement le classer, c'est-à-dire déterminer sa classe ? Alors la programmation par objets au sens où nous l'entendons ici ne s'applique pas, il faut se tourner vers des représentations à bases de *frames*, comme le font les spécialistes de représentation des connaissances.

4,1,3 La réification obligatoire

Les problèmes évoqués ci-dessus sont bien connus, et ont suscité de nombreux travaux. Mais un phénomène plus caché vient souvent perturber la bonne mise en œuvre de la programmation par objets. Il tient à la nature même des entités du monde réel que l'on veut

représenter. Certaines ne sont nullement des objets au sens usuel du terme, mais des actions, des mouvements, des intentions, etc. Et les représenter par des objets, instances de classes, comme nous y contraint notre outil informatique, suppose de leur faire une grande violence. En transformant en objet une entité comme une action, ce qu'on appelle la *réifier*, on la prive nécessairement de certaines de ses propriétés essentielles : par exemple, de son caractère temporel - il ne suffit pas de lui donner deux attributs `date` et `durée` pour représenter adéquatement le temps.

Bien sûr, nous invoquerons ici, comme en 2,1,1, l'idée du projet, du but poursuivi par rapport auquel nous faisons nos approximations : et si ces deux attributs suffisent à notre besoin ? En ce cas, tant mieux, ne cherchons pas plus loin, notre réification sera réussie. La difficulté est que nous avons un penchant naturel à la réification, sans prendre toujours garde aux erreurs qu'ell nous fait commettre. Une réification abusive peut engendrer des mécomptes, surtout si le projet change ultérieurement et qu'on soit amené à reprendre l'analyse.

Or, justement le but d'un logiciel en cours d'analyse est souvent une action à effectuer. On cherche à définir cette action. L'expérience montre que nous n'avons que peu d'instruments intellectuels pour ce faire, alors que, en bons ingénieurs, nous disposons d'une panoplie très étendue pour parler des objets matériels. Nous allons donc, parfois sans même nous en apercevoir, réifier. La programmation par objets correspond à cette pente de l'esprit humain. Elle nous dit :

*"Plutôt que de chercher à définir **l'action** que vous visez,*
*construisez donc **l'acteur** qui exécutera cette action ".*

Car, une fois l'acteur construit, l'action sera déterminée par sa structure, son mécanisme, etc.

La difficulté profonde réside dans la différence essentielle entre l'action et l'acteur, ou plus généralement entre l'action et l'objet représentant l'action. Le passage de l'un à l'autre est le thème de toute une recherche dans le domaine des langages de programmation, sous le nom de réflexivité.

Il arrive que cette représentation soit déjà faite par les spécialistes du domaine eux-mêmes. Par exemple, dans le domaine bancaire, il est généralement admis qu'un *mouvement* (au sens de mouvement comptable) est caractérisé par une date, un montant, un compte origine et un compte destinataire: un objet, donc, instance d'une classe `Mouvement` qu'on peut immédiatement se mettre à écrire. Et pourtant, quoi de plus immatériel qu'un mouvement ? Quoi de plus éloigné d'un objet rémanent ? C'est la *trace* du mouvement qu'on peut observer, pas le mouvement lui-même ! Pouvons-nous valablement nous contenter de cette trace ? En pratique oui, parce que la communauté des employés de banque a de longue date effectué l'analyse qui conduit à représenter le mouvement comptable *réel,* tel qu'il se produit et affecte la vie des personnes, sous la forme de cette abstraction simplifiée, sa trace. Au passage, on notera que cette observation explique pourquoi l'exemple standard donné dans les livres d'initiation à la programmation par objets est si souvent celui du compte en banque...

Mais lorsque cette analyse n'a pas encore eu lieu, il faut la faire, et le risque est grand de perdre en chemin des aspects essentiels. En fait nous voilà revenus à un problème de communication, de groupe social concerné, en somme un problème de sociologie appliquée.

On voit ainsi que la mise en œuvre de la programmation par objets nous pousse sournoisement à orienter nos analyses d'une manière qui peut être contre la nature des choses, sans que nous en prenions clairement conscience. Avec le risque que la nature se venge, et que les systèmes ainsi produits déçoivent leurs utilisateurs...

4,2 Nouvelles demandes

4,2,1 Génie logiciel : classes et modules

Le génie logiciel cherche des moyens pour franchir le gouffre qui sépare la pensée des donneurs d'ordres et l'expression écrite des programmes. Sa problématique est fortement tributaire de la complexité des problèmes traités et de la nature des langages de programmation employés. Sa doctrine de base s'est constituée avec pour référence les langages procéduraux classiques. Elle s'appuie sur une division du "cycle de vie" en trois phases d'analyse (fonctionnelle), de conception, et d'implémentation (procédurale). Du point de vue des langages, elle a fait naître le besoin d'une structure de *modules*, qui devraient représenter des entités intelligibles, si possible réutilisables et le cas échéant susceptibles de compilation séparée.

L'adoption de langages à objets pour la programmation a fait très normalement apparaître une branche de "génie logiciel à objets". En élevant d'un cran le "niveau intellectuel" du code écrit, les objets sont venus bousculer la division tripartite ci-dessus, au profit dun processus "sans couture" (en anglais *seamless*) de l'analyse à l'implémentation. Du même coup, le caractère fonctionnel de la phase d'analyse s'est trouvé mis en question. Suivant la métaphore des "composants logiciels" identifiés avec des classes, la réutilisation du code est devenue une pratique courante, du moins pour les langages qui sont accompagnés d'une bibliothèque de classes standard largement diffusée.

Mais le problème général de la modularité n'est pas résolu pour autant. La structure de classe est en effet trop rudimentaire pour satisfaire tous les besoins en maitère de modules. Plus précisément, on constate que les entités significatives sont presque toujours formées de plusieurs classes reliées entre elles de diverses manières. En d'autres termes, le grain de modularité fourni par les classes est trop fin, on a besoin d'un niveau supérieur d'intégration. Lequel ? Il n'y a pas de réponse décisive. Chaque langage propose son approche et sa terminologie (les *clusters* en EIFFEL, les *frameworks* en SMALLTALK). Trouver une structure de module de niveau adéquat et qui jouisse de bonnes propriétés du point de vue de son implémentation reste un problème ouvert.

4,2,2 Intelligence artificielle : classes et réseaux sémantiques

L'intelligence artificielle, sur son versant représentation des connaissances, privilégie elle aussi l'orientation vers les objets, mais elle se pose des problèmes bien différents. Son but est de construire des bases de connaissances qui seraient exploitables de plusieurs manières, et notamment grâce à des raisonnements, conformément à l'exigence de déclarativité qui veut que la manière de représenter les connaissances soit indépendante de l'usage qui en sera fait. Ceci demande un minimum d'accès aux structures sur lesquelles le raisonnement doit s'exercer, ainsi qu'une sémantique raisonnablement stable.

Or, comme nous l'avons vu en 2,6,2, les langages à objets tendent à présenter les instances comme des entités fermées, capables d'exhiber un certain comportement mais pas

de fournir la base d'un raisonnement. D'une part, les mécanismes d'encapsulation cachent la structure des objets - c'est leur raison d'être - et d'autre part les procédures qui définissent le comportement sont, par nature, rebelles au raisonnement. Enfin, les perversions de la relation d'héritage viennent ruiner tout espoir de stabilité sémantique. On comprend dès lors pourquoi les chercheurs en représentation des connaissances se détournent des langages à objets classiques et préfèrent utiliser des modèles d'objets différents, essentiellement des systèmes d'attributs-valeurs d'où sont bannies les procédures. Ce qui différencie ce modèle des *frames* traditionnels est le maintien de deux niveaux bien distincts, celui des classes, support d'une relation d'héritage, et celui des instances.

Ce qui reste commun aux deux tendances de l'intelligence artificielle et du génie logiciel est le rôle central de la modélisation des concepts, que l'on retrouve au cœur de l'approche connue désormais sous le nom de logiques de descriptions, issue des travaux sur les réseaux sémantiques de type KL-ONE. Il s'agit de définir une algèbre des concepts : opérations ensemblistes d'union et d'intersection, définitions par propriétés caractéristiques, etc. Cette algèbre est ordonnée par la relation de généralisation-spécialisation (en jargon *subsomption*). La grande question, étant donné deux concepts obtenus comme valeurs de deux expressions, est de savoir si l'un est plus général que l'autre. Naturellement, en-dehors de tout contexte d'application, il est facile de produire quelque paradoxe pour mettre en défaut toute approche de ce genre, et il est manifestement illusoire d'espérer trouver un modèle informatique du concept en toute généralité.

Ce n'est qu'en restreignant le champ d'application en fonction d'un projet défini (cf. 2,1,1) et en précisant le cahier des charges en conséquence, qu'on a des chances d'aboutir : ainsi, la formalisation du concept par la classe, au sens des langages à objets, s'est-elle avérée remarquablement efficace dans le domaine de la programmation, et en particulier de la programmation d'interfaces graphiques, mais fort peu satisfaisante, comme nous l'avons dit, du point de vue de la représentation des connaissances. Il est clair que le graphe d'héritage des langages à objets joue le rôle d'un précurseur de réseau sémantique. A cet égard l'insistance des bons auteurs sur la relation d'héritage, malgré ses multiples défauts, se trouve justifiée par l'histoire.

4,2,3 Réutilisation, bibliothèques et "frameworks"

Le succès des expériences historiques de l'environnement de programmation de SMALLTALK et du système de fenêtrage de la Machine Lisp écrit en FLAVORS ont lancé l'idée de composants logiciel, démarquée de celle de composant électronique disponible sur catalogue, et utilisable, puis réutilisable à volonté. C'est Brad Cox qui a été le premier propagandiste de cette vision d'un développement du logiciel qui se ferait enfin sur le mode industriel et non plus artisanal. Et c'est dans ce sens qu'œuvrent les rédacteurs de bilbiothèques de composants standard en SMALLTALK, EIFFEL ou C++.

Il faut observer que la mutation de l'industrie du logiciel qui devrait s'ensuivre n'a pas encore eu lieu. A cela deux ordres de causes . D'une part l'analogie entre composants matériels (en particulier, électroniques) et composants logiciels bute sur la différence profonde et mal comprise entre objets concrets et objets abstraits. Et cette différence rend imprévisible le comportement humain face à ces objets de pensée que sont les logiciels. D'autre part, si on sait à peu près commercialiser des applications logicielles complètes - encore que la nature exacte de ces biens immatériels, copiables, transmissibles par réseau, ne soit pas encore bien établie, voyez les problèmes de copyright - on n'a pas une idée bien précise du statut marchand des

composants logiciels. On a proposé des réseaux de distribution, des redevances pour usage, mais actuellement le frein principal à la pratique du composant logiciel est sans doute économique.

Ajoutons que la mise au point de bibliothèques réutilisables relève d'un art de la programmation très élaboré - puisqu'il ne s'agit pas de satisfaire un cahier des charges donné, mais d'anticiper sur les utilisations qui pourront être faites du composant en cours d'écriture. Et que par conséquent les bonnes bibliothèques sont rares.

Comme on voit, la programmation par objets nous fait aller des fondements de la réflexion individuelle aux structures de commercialisation des logiciels, en passant par de superbes réalisations. Elle est un des acteurs principaux des révolutions opérées par l'informatisation de la société.

5 Bibliographie sommaire

Adele GOLDBERG & David ROBSON : *Smalltalk-80, The Language,* Addison-Wesley 1989.

MASINI, G., NAPOLI, A., COLNET D., LÉONARD D., TOMBRE, K. : *Les langages à objets,* InterEditions 1989.

Bertrand MEYER, . : *Conception et programmation par objets, pour du logiciel de qualité,* InterEditions 1989.

Erich GAMMA, Richard HELM, Ralph JOHNSON, John VLISSIDES : *Design Patterns, Elements of Reusable Object-Oriented Software,* Addison-Wesley 1994.

ERROR: undefined
OFFENDING COMMAND:

STACK: