USTL FLANDRES ARTOIS

LIFL U.A 369 du C.N.R.S

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

N° d'ordre : 322

THESE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

par

Bernard CARRE

METHODOLOGIE ORIENTEE OBJET POUR LA REPRESENTATION DES CONNAISSANCES CONCEPTS DE POINT DE VUE, DE REPRESENTATION MULTIPLE ET EVOLUTIVE D'OBJET

Thèse soutenue le 31 janvier 1989 devant la commission d'Examen

Membres du jury :

M. DAUCHET	Président
C. CARREZ	Rapporteur
J-F. PERROT	Rapporteur
G. COMYN	Directeur de thèse
F. RECHENMANN	Examinateur

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PAREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CONSTANT Eugène	Electronique
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie Appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean-Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilitéss
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BREZINSKI Claude	Analyse Numérique

M. BRIDOUX Michel	Chimie-Physique
M. CELET Paul	Géologie Générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. DAUCHET Max	Informatique
M. DEBOURSE Jean-Pierre	Gestion des Entreprises
M. DHAINAUT André	Biologie Animale
M. DOUKHAN Jean-Claude	Physique du Solide
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
M. FAURE Robert	Mécanique
M. FOCT Jacques	Métallurgie
M. FRONTIER Serge	Ecologie Numérique
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. LABLACHE-COMBIER Alain	Chimie Organique
M. LACOSTE Louis	Biologie Végétale
M. LAVEINE Jean-Pierre	Paléontologie
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique Atomique et Moléculaire
M. LEROY Jean-Marie	Spectrochimie
M. LHOMME Jean	Chimie Organique Biologique
M. LOMBARD Jacques	Sociologie
M. LOUCHEUX Claude	Chimie Physique
M. LUCQUIN Michel	Chimie Physique
M. MACKE Bruno	Physique Moléculaire et Rayonnements Atmosph.
M. MIGEON Michel	E.U.D.I.L.
M. PAQUET Jacques	Géologie Générale
M. PETIT Francis	Chimie Organique
M. POUZET Pierre	Modélisation - calcul Scientifique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. SALMER Georges	Electronique
M. SCHAMPS Joel	Spectroscopie Moléculaire
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. TILLIEU Jacques	Physique Théorique
M. TOULOTTE Jean-Marc	Automatique
M. VIDAL Pierre	Automatique
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne	Composants Electroniques
M. ANDRIES Jean-Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BART André	Biologie animale
M. BASSERY Louis	Génie des Procédés et Réactions Chimiques

Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique Atomique et Moléculaire
M. BERTRAND Hugues	Sciences Economiques et Sociales
M. BERZIN Robert	Analyse
M.BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie Végétale
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie Civil
M. BOIVIN Jean-Claude	Spectroscopie
M. BOUQUELET Stéphane	Biologie Appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BRASSELET Jean-Paul	Géométrie et Topologie
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie Animale
M. CATTEAU Jean-Pierre	Chimie Organique
M. CAYATTE Jean-Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie Structurale
M. CHIVE Maurice	Composants Electroniques Optiques
M. COMYN Gérard	Informatique Théorique
M. COQUERY Jean-Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques et Sociales
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique Nucléaire et Corpusculaire
M. COUTURIER Daniel	Chimie Organique
M. CRAMPON Norbert	Tectolique Géodynamique
M. CROSNIER Yves	Electronique
M. CURGY Jean-Jacques	Biologie
Melle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	Géologie Appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean-Paul	Informatique
M. DELORME Pierre	Physioliogie Animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
M. DENEL Jacques	Informatique
M. DE PARIS Jean Claude	Analyse
M. DEPREZ Gilbert	Physique du Solide - Cristallographie
M. DERIEUX Jean-Claude	Microbiologie
Melle DESSAUX Odile	Spectroscopie de la réactivité Chimique
M. DEVRAINNE Pierre	Chimie Minérale
Mme DHAINAUT Nicole	Biologie Animale
M. DHAMELINCOURT Paul	Chimie Physique
M. DORMARD Serge	Sciences Economiques
M. DUBOIS Henri	Spectroscopie Hertzienne
M. DUBRULLE Alain	Spectroscopie Hertzienne
M. DUBUS Jean-Paul	Spectrométrie des Solides
M. DUPONT Christophe	Vie de la firme (I.A.E.)
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FAUQUAMBERGUE Renaud	Composants électroniques

M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie Sturcturale
M. GAMBLIN André	Géographie urbaine, industrielle et démog.
M. GLORIEUX Pierre	Physique moléculaire et rayonnements Atmos.
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie Physique
M. GOURIEROUX Christian	Probabilités et Statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean-Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie Organique
M. GUILBAULT Pierre	Physiologie animale
M. HENRY Jean-Pierre	Génie Mécanique
M. HERMAN Maurice	Physique spatiale
M. HOUDART René	Physique atomique
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et Statistiques
M. Jean Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme (I.A.E.)
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KREMBEL Jean	Biochimie
M. LANGRAND Claude	Probabilités et statistiques
M. LATTEUX Michel	Informatique
Mme LECLERCQ Ginette	Catalyse
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
Melle LEGRAND Solange	Algèbre
M. LEGRAND Pierre	Chimie
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne
M. LE MAROIS Henri	Vie de la firme (I.A.E.)
M. LEROY Yves	Composants électroniques
M. LESENNE Jacques	Systèmes électroniques
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique théorique
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU Jean-Marie	Optique-Physique atomique
M. MAIZIERES Christian	Automatique
M. MAURISSON Patrick	Sciences Economiques et Sociales
M. MESMACQUE Gérard	Génie Mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie Organique
M. MORTREUX André	Chimie Organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
Mme MOUYART-TASSIN Annie Françoise	Informatique
M. NICOLE Jacques	Spectrochimie
M. NOTELET Françis	Systèmes électroniques
M. PARSY Fernand	Mécanique

M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. STEEN Jean-Pierre	Informatique

Je remercie Max Dauchet d'avoir accepté de présider le jury. Je lui suis très reconnaissant de son accueil au sein du Laboratoire d'Informatique Fondamentale de Lille (LIFL) et des encouragements constants qu'il m'a offerts dès ce jour.

Je remercie Christian Carrez et Jean François Perrot d'avoir accepté d'être rapporteurs de cette thèse et pour le temps qu'ils ont consacré à son examen. Que Jean François Perrot reçoive également toute ma gratitude pour avoir motivé mes projets initiaux lors de l'école d'été de Montréal, en juillet 1986.

Je remercie François Rechenmann d'avoir accepté d'être examinateur et pour l'intérêt qu'il manifeste envers nos travaux.

J'exprime toute ma reconnaissance à Gérard Comyn pour avoir initié, dirigé et critiqué mon étude. Ces années de travail en commun ont été agrémentées et facilitées par ses grandes qualités humaines, son amitié et toute la confiance qu'il m'a toujours manifestée.

Je tiens à associer à ces remerciements les membres du LIFL avec qui j'ai pu entretenir de nombreuses discussions, sources d'idées et d'inspiration. En particulier que soient remerciés Jean Marc Geib et Michel Mériaux, qui a dirigé mes premiers travaux de thèse.

Je remercie Philippe Mathieu pour sa contribution essentielle à la réalisation du projet ROME.

Un grand merci à Jeanine Descarpentries, qui a tapé cette thèse, pour l'efficacité, la patience et la sympathie dont elle a fait preuve.

Je remercie enfin tous ceux, proches et amis, qui participent à l'ambiance et au soutien indispensables au bon déroulement de ces années de recherche.

A ma seconde famille,
Thérèse, Sylvie, Sandrine et Robert Rose
à qui je dois beaucoup.

TABLE DES MATIERES

Cette thèse est structurée en deux parties.
La première, intitulée ***Les Langages Orientés Objet,*** *regroupe les chapitres II et III. Elle constitue une synthèse et une réflexion sur l'existant des points de vue conceptuel, méthodologique et technique.*
La seconde partie, intitulée ***R.O.M.E., Représentation d'Objet Multiple et Evolutive****, correspondant au chapitre IV, présente nos contributions à ce domaine.*
Ces deux parties peuvent être abordées de façon relativement indépendante.

CHAPITRE I

*****

I N T R O D U C T I O N

La représentation des connaissances est devenue une branche à part entière de l'INTELLIGENCE ARTIFICIELLE depuis la prise de conscience de ses chercheurs du fait que les performances d'une traduction informatique d'un problème réel dépendent autant de la puissance algorithmique des techniques de résolution employées que de la représentation de la masse de connaissances du domaine :

"Les chercheurs commencent à comprendre qu'en matière de programmation les tours d'adresse sont intéressants, mais non généralisables... On commence à reconnaître, dans les sphères de l'I.A., que le noeud du problème est de savoir comment les humains exploitent leur bagage de connaissances, et sous quelle forme ils l'entreposent..."

R. SCHANK
(tiré de [DREYFUS. 79])

De nombreux problèmes se posent dès lors pour extraire, structurer et exploiter cette connaissance détenue par l'homme. Ferber dans [FERBER.83] expose à juste titre le problème de la représentation des connaissances à travers deux questions :

"1 - Quelles sont les catégories conceptuelles nécessaires à l'établissement d'une base de connaissance ?

2 - Quelles sont les formulations informatiques les plus adéquates pour représenter ces catégories conceptuelles ?"

La première question est posée à toute la communauté interdisciplinaire de l'Intelligence Artificelle regroupant tant les informaticiens que les philosophes, psychologues, linguistes, etc. C'est le problème de la "conceptualisation de la connaissance".

La seconde question est posée plus directement aux informaticiens, la réponse étant l'objectif ultime d'un courant de recherche informatique auquel nous espérons apporter notre contribution par ce travail.

Deux approches essentielles peuvent être adoptées pour résoudre ce problème :

"There are two basic approaches in artificial intelligence, the performance oriented and the psychologically oriented approach. In the performance oriented approach, the goal is essentially to solve a particular problem. In the psychological approach the goal is also to solve the problem but with the additionnal constraint, that the way in which the program solves the problem should reflect in some sense the way a human being would do the job."

R. PFEIFER,
[PFEIFER.86]

L'approche orientée performance s'attache avant tout à fournir une technique de résolution du problème posé (à résoudre le "quoi"), dans un souci d'optimisation en termes de coût informatique (complexité en temps et en espace). Cette quête de la performance est évidemment à moduler par le respect de certaines qualités logicielles, telles que la structuration, communément reconnues et recherchées actuellement.

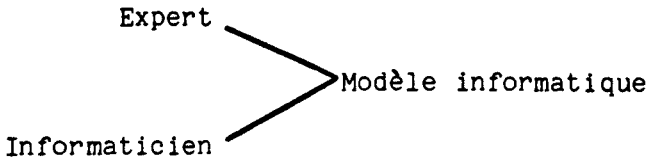
La quête supplémentaire de l'approche psychologique ou orientée connaissances se situe dans le souci de rapprocher au maximum la représentation artificielle (informatique) du problème (le "comment") de sa représentation naturelle, en tant que problème du monde réel. Cette approche cherche avant tout la proximité entre les concepts informatiques manipulés et les concepts naturels qui ressortent des connaissances. Elle tend à répondre conjointement aux deux questions initiales en rapprochant conceptualisation réelle et conceptualisation artificielle des connaissances.

Cette proximité peut se mesurer par la facilité avec laquelle les experts non informaticiens disposant de la connaissance appréhendent les concepts informatiques d'un modèle de représentation et l'intuition naturelle qu'ils peuvent en avoir. Nous pouvons parler alors de puissance déclarative d'un modèle informatique et des outils l'implémentant.

La puissance déclarative d'un outil informatique de représentation des connaissances permet de passer d'un schéma de conception d'un système faisant intervenir l'informaticien (le "cogniticien") comme "filtre" entre l'expert et le modèle informatique :

Expert → Informaticien → Modèle informatique

à un schéma dans lequel l'expert, cette fois-ci simplement aidé par l'informaticien, peut directement formaliser sa connaissance dans le modèle informatique [COMYN.88] :



Dans le premier schéma, correspondant à l'approche orientée performance, l'informaticien est indispensable pour traduire, au sens fort de coder, le problème posé par l'expert, contrairement au second où le modèle de représentation informatique est, du moins en intention, le plus proche possible de la représentation naturelle détenue par l'expert.

En dehors des nombreux avantages de l'approche orientée connaissances, bien mis en évidence par Pfeifer, notamment pour l'acquisition des connaissances, celle-ci nous semble être la plus adaptée dans le contexte de l'I.A., ce domaine de recherche n'aspirant-il pas avant tout à copier l'intelligence humaine ?

Notre souci, dans cette thèse, est donc de contribuer à cette approche de la représentation des connaissances, précisèment en essayant d'étudier et de développer la contribution du paradigme orienté objet à ce problème. Plusieurs modèles de représentation des connaissances ont été définis depuis les réseaux sémantiques jusqu'au modèle orienté objet qui nous concerne, en passant par les règles de production largement utilisées par les outils de développement de systèmes experts. Ces modèles se classent essentiellement en deux familles :

- le formalisme relationnel de la logique, des règles de production et, dans une moindre mesure, des réseaux sémantiques.

- le formalisme objet des langages orientés objet, des frames (schémas), des acteurs et des scripts, sous lequel peuvent également être placés les réseaux sémantiques.

Nous n'exposerons pas ces différents modèles, ceci étant fait largement dans [PINSON.81], [NILSSON.82], [RICH.83], [FERBER.83], [LAURIERE.86], [GALAMBOS.al.86], [VOYER.87], [CERCONE McCALLA.87]. Ferber notamment expose clairement l'opposition entre les formalismes relationnel et objet, tant d'un point de vue conceptuel et philosophique que d'un point de vue informatique. Le formalisme relationnel privilégie les énoncés de relations entre entités de la connaissance, l'entité elle-même n'ayant pas d'existence propre, son identité est dispersée à travers les informations en vrac. A l'opposé, le formalisme objet laisse une identité physique complète aux entités, les objets, responsables de leurs caractéristiques et disposant d'un comportement propre. Cette thèse est purement dédiée à ce dernier formalisme et à ses fondements.

Dans un premier temps, nous présentons et étudions le modèle orienté objet dans sa conception la plus pure qui nous semble être bien définie dans [COINTE.86]. Après un bilan du courant de pensée qui a mené au concept d'objet issu, de manière assez parallèle, de domaines tels que la simulation, l'I.A., la programmation et l'étude des systèmes informatiques, nous présentons les concepts de base qui caractérisent l'approche orientée objet sur laquelle se fonde cette thèse. Ces concepts sont définis métacirculairement dans un modèle de base à la OBJVLISP [COINTE.86], métacircularité (voir l'"auto-description" ou l'"auto-référence" [HOFSTADTER.79] ou encore la "réflexivité" selon Cointe) qui est une garantie de sa qualité :

"[un langage] doit être capable de représenter facilement et adéquatement toute connaissance, y compris lui-même".

J. Ferber
[FERBER.85].

Puis nous présentons une étude méthodologique de l'approche orientée objet, en mettant l'accent sur les propriétés de celle-ci qui en font un "bon" formalisme tant d'un point de vue conceptuel que d'un point de vue informatique à proprement parler. Nous insistons notamment sur l'opposition entre hiérarchie conceptuelle des classes à laquelle est attaché le mécanisme d'inférence propre à ce modèle qui est l'héritage et la hiérarchie structurelle d'objets par agrégation. Nous montrons toute la richesse, mais aussi la grande hétérogénéité de la notion de classe et de hiérarchie d'héritage à travers leurs diverses interprétations et utilisations notamment comme moyen de conceptualisation par affinement/généralisation. Cette hétérogénéité nous semble nécessiter une étude de "garde-fou" méthodologiques qui sont analysés. Les problèmes de stratégie d'héritage, en particulier d'héritage multiple, sont également étudiés pour montrer qu'il n'existe pas actuellement de solution satisfaisante à tous points de vue.

Nous présentons ensuite les fondements de la représentation de hiérarchies structurelles d'entités par agrégation, notamment les problèmes de représentation d'une hiérarchie des parties. Nous montrons que le modèle orienté objet ne propose pas de mécanisme d'inférence particulier pour cette structuration, ce qui est une critique couramment opposée à cette conception. Nous terminons en présentant comment il est aisé de définir et développer la notion de frame métacirculairement à partir des concepts de base du modèle, en tant qu'application de l'agrégation.

Hiérarchies conceptuelle et structurelle sont ensuite confrontées pour insister sur l'importance de leur séparation conceptuelle, méthodologique et technique.

Le dernier chapitre présente notre noyau orienté objet ROME, pour Représentation d'Objet Multiple et Evolutive. Il s'agit d'un outil d'expérimentation de concepts orientés objet issus de nos réflexions présentées précédemment. ROME est défini à la base par un noyau méta-circulaire dans la lignée du modèle orienté objet étudié dans les chapitres précédents mais original par l'apport de notions nouvelles, notamment le concept de point de vue associé à la notion de classe. Une stratégie d'héritage multiple est dès lors bâtie sur cette notion et apporte des solutions à certains problèmes introduits au chapitre III.

Puis est présentée la notion originale de Représentation Multiple et Evolutive. L'un des aspects les plus déclaratifs du modèle étudié jusque là et mis en évidence dans les chapitres précédents, réside dans sa capacité à représenter des hiérarchies conceptuelles déduites naturellement d'une démarche très courante de classification dans tout domaine de connaissances :

"Classification arises from the universal need, in any domain of discourse, to describe uniformities of collections of instances. It is a basic activity of infants in organizing sense impressions, of scientists in organizing knowledge within scientific disciplines, and of application programmers in organizing domain knowledge and behaviour."

P. Wegner
[WEGNER.86]

Nous montrons cependant que cette conceptualisation naturelle des connaissances requiert les concepts de représentation multiple et d'évolution d'objet absents du modèle orienté objet classique.
La représentation multiple consiste en la possibilité de déclarer l'appartenance d'un objet à plusieurs classes. Ce concept est nécessaire pour faciliter l'expression de classifications multiples des mêmes entités, notamment sous des points de vue indépendants.
L'évolution consiste en l'exploitation de l'interprétation d'une hiérarchie de classe comme modèle de généralisation/affinement de concepts pour les objets eux-mêmes. Elle réclame la possibilité de faire évoluer les liens d'appartenance précédents.

Nous montrons que l'absence de ces concepts dans le modèle orienté objet classique est due aux contraintes induites par le mécanisme d'instanciation comme seul moyen d'utiliser la hiérarchie conceptuelle des classes pour les objets eux-mêmes. C'est pourquoi nous introduisons le concept de Représentation Multiple et Evolutive d'objet comme solution homogène au problème précédent. Elle permet de rattacher un objet à plusieurs classes et ceci dynamiquement. Nous étudions sa sémantique et ses conséquences sur l'approche orientée objet de la représentation des connaissances. Nous espérons ainsi augmenter la puissance déclarative de ce modèle.

LES LANGAGES ORIENTES OBJET

Nous n'avons pas inventé la notion d'objet, nous l'avons découverte.

K. Nygaard
European Conference on Object-Oriented Programming,
Paris, 1987.

C H A P I T R E II

*****

II - LE MODELE ORIENTE OBJET

II.1.- INTRODUCTION AUX CONCEPTS DE BASE

Aujourd'hui le langage orienté objet le plus connu est certainement SMALLTALK [GOLDBERG-ROBSON. 83]. Il faut cependant remonter à la création de SIMULA [DAHL-NYGAARD. 65] [NYGAARD-DAHL. 78] pour voir apparaître la notion centrale d'objet telle qu'elle est connue actuellement. SIMULA, extension de ALGOL-60, fut conçu pour traiter des problèmes de simulation qui resteront un domaine d'application privilégié de la programmation par objet [BEZIVIN. 84]. Le but était de développer un outil de description et de simulation de systèmes. La notion de système est à prendre ici selon la définition suivante de NYGAARD :

> "A system is a part of the world that a person (or group of persons) chooses to regard as a whole consisting of components, each component characterized by properties that are selected as being relevent and by actions related to these properties and those of other components"
>
> [NYGAARD. 86]

Dès lors une approche naturelle pour simuler de tels systèmes est de décrire directement ses composants par des objets possédant leurs propriétés et sachant réaliser certaines actions.

Le concept d'objet est né et prend la forme d'une entité rémanente regroupant, au sein d'une même structure, données et actions.

objet = <données , actions>

Le choix d'une entité très générale fait suite à une recherche de l'uniformité commune aux créateurs de SIMULA :

> "A tool both for the person writing the description and for people with whom he wanted to communicate about the system".
>
> [NYGAARD. 86]

puis aux concepteurs de son héritier direct SMALLTALK. Les origines de SMALLTALK remontent à 1969 quand A. KAY cherche un formalisme unique pour décrire toutes les composantes de la machine FLEX, notamment son système d'exploitation et son environnement de programmation. Ceci le mène à un minimum de concepts dont celui très général d'objet similaire à l'objet SIMULA. Ce souci d'uniformité est lié à celui de simplicité :

> "the purpose of the SMALLTALK project is to support children of all ages in the world of information. The challenge is to identify and harness metaphors of sufficient simplicity and power to allow a single person to have access to, and control over; information which ranges from numbers and text through sounds and images."
>
> [INGALLS. 78]

Il suffit aujourd'hui d'observer des incarnations du concept d'objet aussi diverses qu'un entier, un rectangle, une fenêtre graphique ou encore un flot dans l'environnement SMALLTALK 80 pour constater toute sa généralité.

Autant que ces travaux, c'est tout un courant de pensée fondamentalement orienté objet qui s'est développé et a contribué au rôle actuel de cette approche. Les langages d'acteurs [HEWITT-BISHOP-STEIGER. 73] [PUGH. 84] [AGHA. 86] qui trouvent leur origine au MIT autour de C. HEWITT mettent également en avant la notion d'objet sous la forme d'acteur indépendant et responsable de sa propre connaissance

> "Actors are potentially active chunks of knowledge which communicate by exchanging polite messages."
>
> [HEWITT-SMITH. 75]

De façon similaire les travaux de MINSKY en psychologie cognitive [MINSKY. 75] menèrent à la notion proche de frame qui, rétrospectivement, peut apparaître comme une extension de la notion d'objet comme nous le verrons dans le paragraphe III.4.4.

> "a frame is a data structure for representing a stereotyped situation, like being in a certain kind of living room, or going to a child's birthday party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expend to happen next. Some is about what to do if these expectations are not confirmed."
>
> [MINSKY. 75]

La diversité de ces travaux montrent l'importance d'une certaine approche orientée objet en informatique qu'il peut paraître difficile de cerner exactement, ce qui est bien traduit par cette phrase de D.G. BOBROW à OOPSLA'86

"The subject of O.O.P has no future; O.O.P is not a subject; it is a conglomerate".

où O.O.P. signifie object oriented programming et peut être étendu à object oriented approach. Il semble cependant que le principe de généralité et d'uniformité autour d'une seule entité, l'objet, soit le fil conducteur de ce courant de pensée. Cette idée est même poussée à l'extrême dans la communauté orientée objet jusqu'à la maxime : "tout est objet". FERBER dans [FERBER. 83] met l'accent sur la vision très ancienne de la connaissance autour d'objets.

"Depuis Aristote les choses de l'univers (puis les idées avec Platon), sont considérées comme des "objets" bénéficiant d'un certain nombre d'attributs : l'essence, la qualité, la relation, le temps, la situation, l'action, la passion et l'avoir".
[FERBER. 83]

Le paradigme orienté objet suit directement cette hypothèse de structuration de la connaissance autour d'entités disposant de leurs propres caractéristiques et responsables de celle-ci

"knowledge should be organized around conceptual entities with associated descriptions and procedures."
[BOBROW-WINOGRAD. 77]

L'importance de ce paradigme en Intelligence Artificielle et plus particulièrement en représentation des connaissances tient à cette hypothèse assez naturelle et intuitive. Pratiquement, il s'agit de copier cette vision en modélisant, en simulant les entités du monde réel, les choses au sens large, par des entités informatiques proches, les objets.

"La programmation par objet permet de traduire directement une vision intuitive. Il s'agit d'une simulation de la réalité plutôt qu'une élucidation théorique.
J.F. PERROT
Ecole d'Eté d'Informatique de l'A.F.C.E.T.,
MONTREAL, 1986

"L'un des avantages de la conception orientée objet réside dans son aptitude à refléter, dans la structure du logiciel, la structure du monde réel de l'application ou du moins de l'abstraction que le concepteur s'en fait. Les objets logiciels définis dans les programmes correspondent à une implémentation des objets du monde réel."
[PITETTE. 86]

L'aspect naturel et intuitif de l'approche orientée objet confère aux outils bâtis sur celle-ci une puissance d'expression de haut niveau qui fait leur intérêt pour l'Intelligence Artificielle.

II.2 - LES CONCEPTS DE BASE

Nous allons introduire les concepts minimaux d'une famille de langages à laquelle appartient SMALLTALK et beaucoup d'autres outils orientés objet. Précisons dès maintenant que la limitation à cette famille fait référence à une controverse sur le moyen de décrire et créer les objets. En effet, le modèle classe/instance/héritage présenté ici à partir du § II.2.2. n'est pas la seule solution à ce problème et se trouve concurrencé par le modèle prototype/délégation [LIEBERMAN. 86] propre à une autre famille de langages proches des langages d'acteurs et qui sera présenté succinctement au § II.3.5. Nous n'insisterons pas sur le dernier modèle, notre étude étant centrée sur le premier. Notons que fondamentalement aucun des deux modèles n'est "plus orienté objet" que l'autre. Il s'agit uniquement d'un choix entre deux approches philosophiques et conceptuellement différentes de la représentation des connaissances par objet qui a des conséquences techniques et méthodologiques non négligeables.

II.2.1.- Objet et envoi de message

L'objet est l'entité de base qui possède sa propre caractérisation. Celle-ci est un ensemble de caractéristiques de deux types :

Les caractéristiques factuelles nommées attributs, variables d'instance, slots, champs,... contiennent des données locales à l'objet.

Les caractéristiques comportementales appelées méthodes, procédures, opérations, sont des actions que sait réaliser l'objet

objet = <attributs , méthodes>

Prenons tout de suite des exemples informels. Une porte logique par exemple, un And à 2 entrées, peut être caractérisée par ses états d'entrée/sortie dans les attributs respectifs : entrée1, entrée2, sortie et les méthodes :

- de mise à jour des états d'entrée, appelées entrée1:unétat , entrée2:unétat,
- de lecture des entrées/sorties : entrée1, entrée2, sortie
- de calcul de l'état de sortie : calcule-sortie.

On peut lui ajouter une caractérisation graphique par les attributs position et orientation et une méthode de dessin appelée "dessine-toi":

Un-And

attributs : entrée1 true
entrée2 false
sortie false
position (10,20)
orientation ouest

méthodes : entrée1:un_état
entrée2:un_état
sortie
calcule-sortie
dessine-toi

Un rectangle peut être caractérisé par deux points (à la SMALLTALK) origin, corner et plusieurs méthodes telles que origin: unpoint, corner: unpoint, dessine-toi, calcule-largeur, calcule-surface.

un-rectangle

attributs : origin (10,20)
corner (110,200)

méthodes : origin: unpoint affectation de son origine
corner: unpoint affectation de son coin
dessine-toi : méthode de dessin
calcule-largeur : méthode de calcul de sa largeur
calcule-surface : méthode de calcul de sa surface

Un objet souvent cité en introduction aux langages orientés objet est la tortue logo caractérisée par sa position, son cap, sa couleur, la position de sa plume et les méthodes avance, tournedroite, leve-crayon,...

Une première constatation suite à cette définition de l'objet est le **regroupement, au sein d'une seule entité des données et procédures.**

Un apport essentiel de l'approche orientée objet est en effet de supprimer la dichotomie structures de données d'une part, procédures manipulant ces données d'autre part. Ici, données et procédures sont regroupées, **encapsulées**, au sein d'une même entité, l'objet, autonome et responsable de ces informations.

Le seul moyen d'activer un objet est l'envoi de message (ou transmission) : **on ne manipule pas un objet, on communique avec lui.** Ce mécanisme de communication entre objets (y compris l'utilisateur !) consiste à demander à un objet d'effectuer une action, i.e. d'appliquer une de ses méthodes. Reprenant la terminologie SMALLTALK, à chaque méthode est associé un sélecteur que l'on peut assimiler à son nom et un certain nombre de paramètres. L'envoi de message prend alors la forme :

<objet-destinataire sélecteur arguments>

A la réception d'un message, l'objet destinataire réagit en appliquant la méthode associée au sélecteur spécifié, s'il en dispose, sinon ceci entraîne une erreur "message not understood...".

Reprenant les exemples précédents, en réponse aux messages "Un-And dessine-toi" "un-Rectangle dessine-toi" ou "une-tortue tournedroite", respectivement Un-And et Un-Rectangle se dessineront, Une-Tortue tournera à droite.

Cette approche est à opposer à l'approche procédurale classique où l'objet est manipulé par une procédure qui le prend pour paramètre :

approche orientée objet envoi de message	approche procédurale appel procédural
Un-And dessine-toi	dessin (Un-And)
Un-Rectangle dessine-toi	dessin (Un-Rectangle)
Une-Tortue (tournedroite)	tournedroite (Une-Tortue)

Tout objet est vu de l'extérieur par son interface, son protocole qui est l'ensemble des sélecteurs de méthodes qui peuvent être spécifiés dans un envoi de message. Un objet ne peut communiquer avec un autre objet qu'à travers le protocole de ce dernier qui définit les messages auxquels il sait répondre. Bien que l'envoi de message ne soit pas le mécanisme général d'application d'une méthode dans les outils orientés

objet (*) penser en ces termes contribue fortement à mieux percevoir l'approche orientée objet dans laquelle l'objet est un acteur responsable activé par envoi de message et non une entité passive et manipulée.

La notion d'envoi de message induit tout naturellement le polymorphisme [CARDELLI-WEGNER. 85] puisque la réponse dépend avant tout de l'objet receveur. Un même sélecteur peut appartenir au protocole de plusieurs objets lui associant des méthodes différentes. Par exemple, le message "dessine-toi" peut être envoyé à un rectangle, Un-And et en général tout objet qui a un comportement graphique, chacun répondant évidemment de façon différente. Cette propriété est également essentielle et implicite dans l'approche orientée objet.

Enfin, un autre principe propre à la notion d'objet est la rémanence ou le "reactive principle" de INGALLS :

> "The valient feature of SMALLTALK is that all objects are active, ready to perform in full capacity at any time."
> [INGALLS. 78]

Un objet conserve son état d'une activation à une autre. Cet état est représenté par son environnement local défini par ses attributs et manipulé uniquement par ses propres méthodes. Ainsi, tout changement de l'état d'un objet et donc d'un de ses attributs, passe par l'application d'une méthode, i.e. un envoi de message. Par exemple le changement de l'état de l'entrée 2 du And précédent se fait par l'envoi du message :

```
Un-And  entrée2:  true
```

Après le message :

```
Un-And  calcule-sortie
```

son état de sortie sera alors true.

(*) En Flavors par exemple [MOON.86] la notion de fonction générique correspond à celle de sélecteur. L'envoi de message est remplacé par une généralisation de l'appel fonctionnel Lisp classique qui prend un objet comme premier argument et qui recherche la méthode à appliquer pour cet objet. Il n'y a pas de différence syntaxique entre un appel fonctionnel classique et un appel de fonction générique, par exemple (dessine-toi un_And) correspond à l'envoi de message "un_And dessine-toi" . L'envoi de message adopté par SMALLTALK et d'autres L.O.O. cache d'ailleurs l'appel fonctionnel par l'équivalence [BOBROW and al.86]

```
          objet  sélecteur  argt1 ... argtn
<=> (funcall(méthode_associée sélecteur objet) objet argt1...argtn)
```

où la fonction méthode_associée recherche la méthode associée au sélecteur pour l'objet qui doit être appliquée et le paramètre objet permet de définir l'environnement d'application. Cette correspondance est classiquement à la base de l'implémentation de l'envoi de message (primitive send) pour le développement de L.O.O. au dessus de Lisp.

Nous reviendrons largement sur ces principes et les propriétés qu'ils induisent.

II.2.2.- Classe et instance

Comment définir la multitude d'objets qui peuvent intervenir dans un problème donné ? C'est SIMULA qui, le premier, introduit la notion de classe (SIMULA I introduit l' "activity" qui est reprise et généralisée ensuite par la notion de "class" en SIMULA 67) reprise par SMALLTALK.

L'idée centrale est que **tous les objets semblables par leur caractérisation, qui ont les mêmes propriétés, sont des représentants, des concrétisations, des instances d'une même entité abstraite, la classe.** La classe définit un modèle d'objet en spécifiant une caractérisation générale de la forme :

attributs : nom_attribut_1 , nom_attribut_2 ...

méthodes : sélecteur1 méthode associée
sélecteur2 méthode associée.

Nous pouvons ainsi parler de la classe des entiers, des tableaux, des chaînes de caractères, des rectangles, des personnes ou des composants électroniques. (SMALLTALK-80 définit une centaine de classes à la base). Par exemple, la classe And va définir les portes and telles que notre objet and du paragraphe II.2.1. :

Classe And

Attributs : entrée1, entrée2, sortie, position, orientation

Méthodes : entrée1: un_état ... (écriture)
entrée1 ... (lecture)

calcule-sortie
dessine-toi

Les concepts d'objet et de classe sont liés par l'instanciation ou mécanisme de création d'un objet à partir d'une classe, dont il devient instance. Toutes les instances d'une même classe disposent des mêmes attributs et méthodes définies dans celle-ci. Elles ne diffèrent que par les valeurs (valeurs d'instance) associées à leurs attributs.

Une remarque importante est qu'**un objet n'existe pas sans sa classe** qui le caractérise complètement : tout objet est instance d'une classe à laquelle il est lié à l'instanciation par un lien du type "is-a", "est-un". Tous les objets ont cette propriété et peuvent donc déjà être caractérisés par une première classe appelée généralement OBJECT :

OBJECT

attributs : est-un

L'attribut est-un aura pour valeur la classe d'instanciation de l'objet.

Se pose alors le problème du statut de ces nouvelles entités que sont les classes. En SIMULA, le statut d'une classe est proche de la notion de type abstrait [BRIOT. 85] dont nous donnerons la définition suivante [CARDELLI-WEGNER. 85] :

> "An abstract data-type defines a class of abstract objets which is completely characterized by the operations available on those objects"

La déclaration d'une classe en SIMULA apparaît comme la déclaration d'un type en tête du bloc dans lequel elle sera utilisée. La référence à des objets de ce type se fait par la déclaration de variables de type ref (ou pointeur) qui pourront référencer des instances de cette classe créees par la primitive "new".

C'est SMALLTALK 76 qui unifie la notion de classe et d'objet selon la maxime d'uniformité "tout est objet". La classe a alors le statut d'objet et, comme tout objet, est instance d'une classe, appelée méta-classe : **Une méta-classe est une classe qui caractérise des classes.**

Dans cette approche, l'instanciation d'un objet est alors vue naturellement comme un comportement de la classe en tant qu'objet doué du pouvoir d'abstraction et de création d'autres objets, ses instances. Toute classe comprend le message "new" dont le sélecteur correspondant est associé à une méthode implémentant le mécanisme d'instanciation. A la réception de ce message, une classe répond en créant un exemplaire, une instance du modèle d'objet qu'elle spécifie et établit le lien "est-un". Par exemple, la création d'un and par la classe And, se fera par l'envoi de message "And new".

Ceci mène à la définition d'objets particuliers que sont les classes. Tout comme la classe OBJECT caractérise tous les objets, nous pouvons définir la classe de toutes les classes, nommée CLASS :

CLASS

attributs : attributs, méthodes

méthodes : new, méthode d'instanciation

Cette définition de la notion de classe est à interpréter comme suit : toute classe est un objet caractérisé par les attributs "attributs" et "méthodes" qui contiendront une caractérisation, un modèle d'objet, et par la méthode "new" par laquelle elle sait créer des objets, ses instances qui auront la caractérisation précédente.

Quelle est alors la position de la classe CLASS par rapport à la classe OBJECT ? Rappelons que OBJECT caractérise tous les objets, nous pouvons donc lui associer l'ensemble O de tous les objets, selon une interprétation ensembliste assez intuitive. De la même façon, à CLASS est associé l'ensemble C de toutes les classes. Toute classe étant un objet, c'est un sous-ensemble de O. Ce qui introduit la notion de sous-classe présentée dans le paragraphe suivant et qui positionne la classe CLASS par rapport à OBJECT :

CLASS est sous-classe de OBJECT

II.2.3.- Sous-classe et héritage

Ces notions ont été introduites par SIMULA.67.

Nous avons vu que le concept de classe permet de regrouper tous les objets semblables par leur caractérisation. Les concepts de sous-classe et d'héritage permettent intuitivement de définir des objets "plus ou moins semblables" :

> "Inheritance is the concept in object languages that is used to define objects that are almost like other objects".
> [STEFIK-BOBROW. 85]

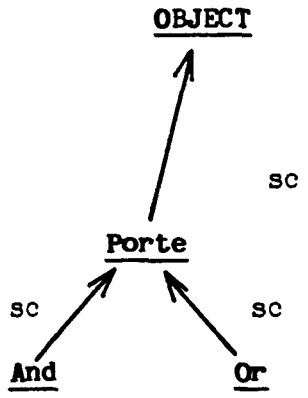
Reprenons nos "and" à deux entrées que nous avons caractérisés par les attributs entrée1 entrée2 sortie... et plusieurs méthodes dont calcule-sortie. De façon similaire les or à deux entrées peuvent être caractérisés par les mêmes attributs mais une méthode calcule-sortie différente. Les and et les or sont donc semblables par leurs attributs et le fait qu'ils aient une méthode d'évaluation de leur sortie, calcule-sortie. Nous pouvons généraliser cette caractérisation à toute porte à deux entrées et une sortie autour du concept représenté par la classe Porte et qui définit la caractérisation générale suivante :

Porte

attributs : entrée1 , entrée2 , sortie

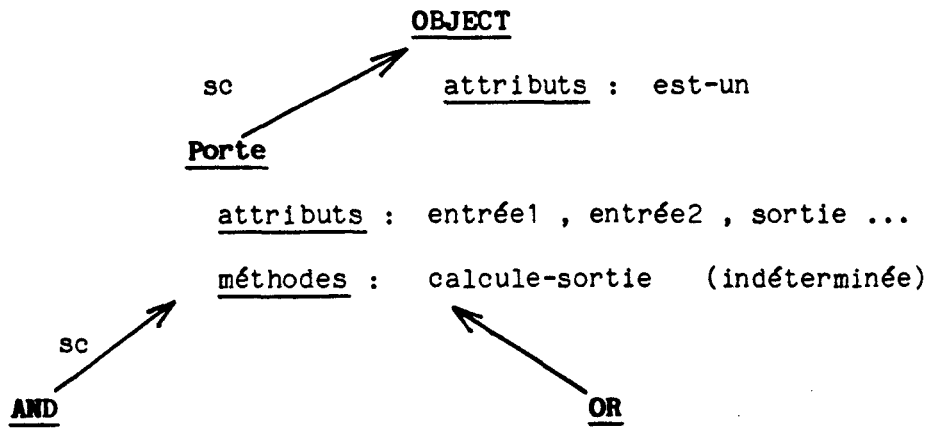
méthodes : calcule_sortie

sachant qu'il est impossible de définir la méthode calcule-sortie, en général, qui est donc indéterminée à ce niveau. Selon l'interprétation ensembliste déjà introduite dans le paragraphe précédent, à cette classe est associé l'ensemble de toutes les portes à 2 entrées et 1 sortie dont deux sous-ensembles sont les "and" et les "or" définis respectivement par les classes "And" et "Or". La notation de sous-classe permet de représenter cette relation d'inclusion : And et Or sont sous-classes de Porte. Inversement, Porte est sur-classe des classes And et Or . Enfin, et de la même façon, les portes étant des objets, la classe Porte est sous-classe de la classe OBJECT ce qui est représenté par le schéma suivant :



où $x \xrightarrow{sc} y$ est la notation retenue pour "x est sous-classe de y"

La méthode calcule-sortie peut maintenant être définie au niveau de chaque sous-classe de Porte selon le schéma informel suivant :



attributs :

méthodes : calcule-sortie
sortie ← entrée1 ∧ entrée2

attributs :

méthodes : calcule-sortie
sortie ← entrée1 ∨ entrée2

Concernant la caractérisation des objets représentants de ces classes, nous constatons que le lien classe-sous-classe s'interprète intuitivement comme un lien d'affinement, de spécialisation et, inversement, de généralisation ou d'abstraction.

Cette relation classe-sous-classe induit donc implicitement un mécanisme d'inférence des caractéristiques appelé héritage qui assure que

> tout objet représentant d'une classe est caractérisé par celle-ci et par ses sur-classes selon l'ordre classe-sur-classe, c'est-à-dire du plus affiné au plus général.

Ainsi sur l'exemple, la classe PORTE définit une caractérisation générale commune à toutes les portes, notamment les AND et les OR, selon un processus de factorisation des propriétés partagées par ces objets. Cette caractérisation est affinée dans les sous-classes AND et OR. En particulier, la méthode calcule-sortie, indéterminée au niveau plus abstrait de la classe PORTE, reçoit une définition propre à chaque sous-classe. Par le mécanisme d'héritage, toute porte and, objet représentant de la classe AND, est caractérisée par la méthode calcule-sortie la plus affinée, définie dans AND, par les attributs entrée1 , entrée2 , sortie, hérités de PORTE, et enfin par l'attribut est_un hérité de OBJECT.

Nous verrons par la suite qu'une classe peut être sous-classe de plusieurs autres classes, ce qui induit un mécanisme d'héritage multiple.

Cette nouvelle caractéristique de toute classe est naturellement définie dans la classe CLASS de toutes les classes par un attribut "sorte de" qui contiendra la (ou les) sur-classe(s) d'une classe. Ceci complète la définition déjà donnée de CLASS comme suit :

CLASS

attributs attributs méthodes sorte-de

méthode new

L'attribut sorte-de implémente la relation classe-sur-classe sur laquelle opère l'héritage.

II.2.4.- Métaclasses

Les notions de base que nous avons présentées dans les paragraphes précédents suffisent à définir celle de métaclasse. Une classe est un objet et donc est caractérisée par une classe, appelée pour commodité une métaclasse. La métaclasse CLASS définit la caractérisation générale partagée par toute classe. Toute sous-classe de CLASS est donc une métaclasse puisqu'elle définit des classes. Plus généralement, toute sous-classe d'une métaclasse est une métaclasse pour les mêmes raisons. D'autre part, toute métaclasse est une classe puisqu'elle définit des objets, des classes. Toute métaclasse est donc instance d'une métaclasse, au minimum CLASS. Synthétiquement,

toute métaclasse est à la fois instance et sous-classe d'une métaclasse, au minimum CLASS.

Ainsi une métaclasse n'est rien d'autre qu'une classe et donc un objet. Tout comme définir une sous-classepermet de caractériser des objets particuliers, définir une "sous-métaclasse" permet de définir des classes particulières. Nous reviendrons dans le paragraphe suivant sur cette notion qui peut paraître compliquée au premier abord mais qui simplifie en fait le modèle de base en le définissant par lui-même (métacircularité). Nous verrons des exemples de métaclasses notamment pour la définition de méthodes particulières d'instanciation puisque celle-ci est typiquement un comportement des objets classes. Les métaclasses n'ont pas toujours été considérées comme de simples classes. Comme nous l'avons précisé au § II.2.2., cette notion a été introduite en SMALLTALK 76 pour unifier l'univers des entités divisées en objets d'une part et classes d'autre part en SMALLTALK 72 (*1).

CLASS était cependant la seule métaclasse et cette notion a été généralisée en SMALLTALK-80 où toute classe a sa propre métaclasse, sous-classe de l'ultime métaclasse CLASS. SMALLTALK-80 gère automatiquement les métaclasses qui n'apparaissent pas directement à l'utilisateur et selon un schéma assez complexe que nous ne présenterons pas [BRIOT. 85] (*2)

Les caractéristiques propres de la classe, en tant qu'objet instance de sa métaclasse, sont appelées variables de classe et méthodes de classe par opposition aux variables d'instance et méthodes d'instance des objets qu'elle caractérise (*3)

Après la présentation du modèle de base dans le paragraphe qui suit nous reprendrons cette terminologie SMALLTALK qui permet d'oublier superficiellement la notion de métaclasse, sachant qu'elle est implicite.

(*1) Où cette notion n'existait pas, ni même en SIMULA; ni même actuellement dans beaucoup de L.O.O. comme FLAVORS, COMMONOBJECTS [SNYDER. 86a]. Elle peut paraître superflue pratiquement mais a l'avantage de valider les concepts par eux-mêmes. Si les concepts définis par un modèle se veulent assez puissants pour représenter toute forme de connaissance, ils doivent commencer par se représenter aux-mêmes.

(*2) CLASS est elle-même instance d'une métaclasse Métaclass dont la métaclasse, (Métaclass Class) est instance d'elle-même

(*3) En fait, ses variables de classe sont accessibles par toutes ses instances (et les instances de ses sous-classes...) et définissent donc un environnement global partagé par celles-ci. Nous négligeons cet aspect et considérerons les variables de classe comme de simples variables d'instance de la classe en tant qu'objet.

II.3.- LE MODELE METACIRCULAIRE DE BASE

Les concepts vus précédemment et leurs relations peuvent être synthétisés dans un modèle métacirculaire selon le principe d'uniformité cher à la communauté orientée objet :

> "one way of stating the SMALLTALK philosophy is to choose a small number of general principles and apply them uniformly."
> [KRASNER. 83]

Ce principe d'uniformité se révèle à travers la métacircularité du modèle, par lequel les concepts de base objet, message, classe, instanciation, sous-classe et héritage se définissent en fonction d'eux mêmes.

> "So it is almost the reader must know everything before knowing anything".
> [GOLDBERG. 83]

Le modèle de base présenté ici est inspiré de OBJVLISP [BRIOT-COINTE. 86] en tant que noyau minimum qui définit une sémantique générale des langages orientés objet de la famille SMALLTALK [COINTE. 86] que nous utiliserons par la suite.

(*) Nous n'entrerons pas dans les détails du langage OBJVLISP en ne retenant que ses concepts d'une façon assez superficielle.
En particulier, nous essaierons de cacher au maximum le LISP sous-jacent qui nous ferait entrer dans des détails techniques et syntaxiques sans intérêt à ce niveau conceptuel.
Par la suite nous adopterons un pseudo-langage de type SMALLTALK pour nos exemples.

II.3.1.- Les cinq postulats d'OBJVLISP [BRIOT-COINTE. 86]

P1 : Toute entité du langage est un objet.
Un objet représente un fragment de connaissance et un ensemble de potentialités :

objet = <données , procédures>

P2 : La seule structure de contrôle est l'envoi de message : le protocole pour activer un objet.
Un message spécifie quelle procédure appliquer selon la forme :

(send objet destinataire sélecteur arg1 ... arg2)

où le sélecteur est le nom de la méthode et arg1 ... arg2 ses arguments.

P3 : Tout objet appartient à une classe qui spécifie ses données (attributs appelés champs ou variables d'instance) et son comportement (procédures appelées méthodes). Les objets sont créés dynamiquement à partir de ce modèle, ce sont les instances de la classe. Respectant la philosophie de Platon, toutes les instances d'une classe ont la même structure et la même forme, mais diffèrent par les valeurs associées à leurs attributs.

P4 : Une classe est aussi un objet, généré à partir d'une classe, appelée métaclasse (car elle décrit les classes). Ainsi (P3) à chaque classe est associée une métaclasse.
La métaclasse initiale et primitive est la classe CLASS.

P5 : Une classe peut être définie comme sous-classe d'une ou plusieurs autres classe(s). Ce mécanisme de sous-classement (subclassing) permet l'héritage des attributs et des méthodes et est appelé héritage. La classe OBJECT représente le comportement commun à tous les objets.

Le postulat P3 doit en fait être affiné pour les langages respectant ce modèle. Cette précision est relative à la distinction nécessaire entre les notions d'instance et de représentant d'une classe :

Définitions : instance (ou représentant direct)

Nous appelons instance d'une classe (appelée une-classe) tout objet (appelé un-objet) créé (instancié) par celle-ci par l'envoi de message :

(send 'une-classe 'new 'un-objet...)

Cette classe est appelée classe d'instanciation, classe génératrice de l'objet.

représentant :

tout objet appartient à, est représentant d'une classe s'il est instance de cette classe ou d'une sous-classe de cette classe (*1)

Reprenons l'exemple des portes logiques, soit un-and instance de la classe And : (send 'AND 'new 'un-and...) Un-And est instance, représentant direct de And, sa classe d'instanciation, et est représentant de PORTE, OBJECT, surclasses de celle-ci.

Le postulat P3 doit être complété par le principe d'instanciation unique : (*2)

Tout objet est instance d'une classe et une seule.

"Each object is associated with a single class"
[GOLDSTEIN-BOBROW. 80]

Précisons également le postulat P2 : l'envoi de message est le seul moyen d'activer un objet. L'activation d'un objet se traduit par l'application de la méthode associée au sélecteur spécifié dans le message, comme nous l'avons montré dans le paragraphe II.2.1. L'envoi de message doit donc être vu plus précisément dans ce modèle comme le seul moyen d'appliquer une méthode. En particulier un objet, pour appliquer ses propres méthodes, passe par l'envoi de message à lui-même, ce qui introduit la notion de self-message. Il existe une variable particulière appelée généralement self ("this" en SIMULA) qui référence l'objet en cours d'activation. Les self messages ont donc la forme :

(send self sélecteur arguments)

Dans la définition d'une méthode, au niveau d'une classe, une référence à self correspond donc au représentant de celle-ci qui exécutera cette méthode, c'est-à-dire l'objet destinataire spécifié dans l'envoi de message qui entraîne l'application de cette méthode. Nous en verrons un exemple dans le paragraphe III.3.3.

(*1) La relation "être représentant de" est transitive comme l'est la relation d'appartenance aux ensembles d'objets associés à chaque classe.

En SMALLTALK :

UnObjet est instance de Une-classe
<=> (UnObjet isMemberOf: UneClasse) = true
UnObjet est représentant de UneCclasse
<=> (UnObjet isKindOf: UneClasse) = true

(*2) Ce principe est respecté par les outils bâtis sur le modèle classe/instance/héritage présenté ici. Nous verrons les conséquences qui sont à la base de notre étude sur la représentation directe multiple et évolutive et seront exposées dans le chapitre IV.

II-3-2.- Objet et classe : OBJECT et CLASS

Nous avons vu au paragraphe II.2 que les concepts d'objet et de classe ne peuvent se définir l'un sans l'autre :

- tout objet est instance d'une classe
- toute classe est elle-même un objet.

Nous avons introduit les deux classes de base OBJECT et CLASS qui définissent ces concepts. OBJECT définit la caractérisation commune à tous les objets notamment la caractéristique "est-un" relative au postulat P3 et qui traduit le premier énoncé précédent : tout objet est instance d'une classe. Est-un est un attribut de tout objet; il est donc spécifié dans la classe OBJECT :

OBJECT

attributs : est-un

CLASS définit la caractérisation commune à toutes les classes dont les attributs :

attributs, méthodes : par lesquels une classe spécifie une caractérisation, un schéma, un moule, un modèle d'objet défini par les caractéristiques factuelles qui sont les attributs (variables d'instance, champs...) et les caractéristiques comportementales que sont les méthodes (procédures, actions ...) nommées par leurs sélecteurs.

sorte-de : par lequel une classe est sous-classe d'autres classes et qui permet l'héritage des attributs et méthodes définies dans celles-ci.

et la méthode :

new : par laquelle une classe est un générateur d'objets, ses instances, qui lui seront liées par leur attribut est-un.

Ceci traduit les deux fonctionnalités fondamentales d'une classe [CARRE-COMYN. 87a] :

* la fonctionnalité d'abstraction

  par laquelle une classe représente un concept général, abstraction de tous les objets qu'elle caractérise.

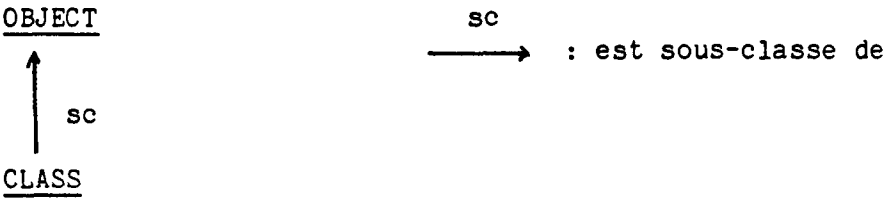
* la fonctionnalité de génération, ou instanciation par laquelle elle crée des objets (instances) disposant de la caractérisation qu'elle décrit et **inférée par héritage** sur ses sur-classes.

CLASS

attributs : attributs, méthodes, sorte-de

méthodes : new.

Selon le deuxième énoncé et le postulat P4 toute classe est elle-même un objet, ce qui se traduit par le fait que CLASS est sous-classe de OBJECT.



OBJECT est la seule classe qui n'est sous-classe d'aucune autre classe puisqu'elle est la classe minimum de tout objet.

"The reverse question is : is every object a class ?
whose answer is no".

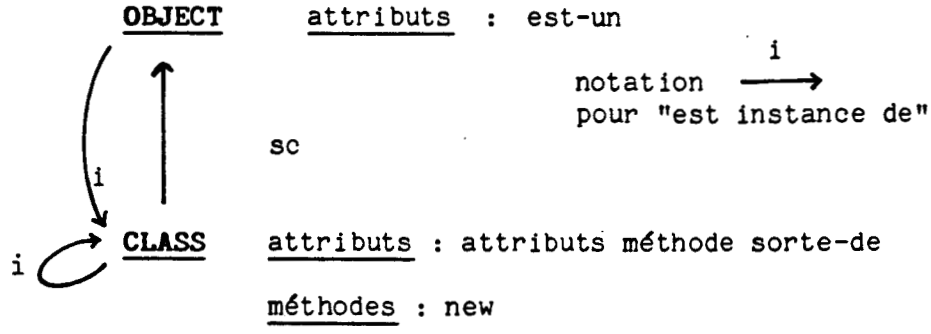
[BRIOT-COINTE. 86]
[BRIOT. 85]

En effet, seuls certains objets, les classes, représentants de CLASS, ont les deux fonctionnalités précédentes, contrairement aux objets représentants de OBJECT et non de CLASS, appelés instances terminales. Par exemple, un And particulier, instance de la classe And ne peut être considéré comme l'abstraction d'autres objets. (*). C'est pourquoi la classe CLASS n'est que sous-classe de OBJECT et non confondue avec celle-ci et ce schéma doit être interprété comme suit avec les définitions d'instance et de représentant précédentes :

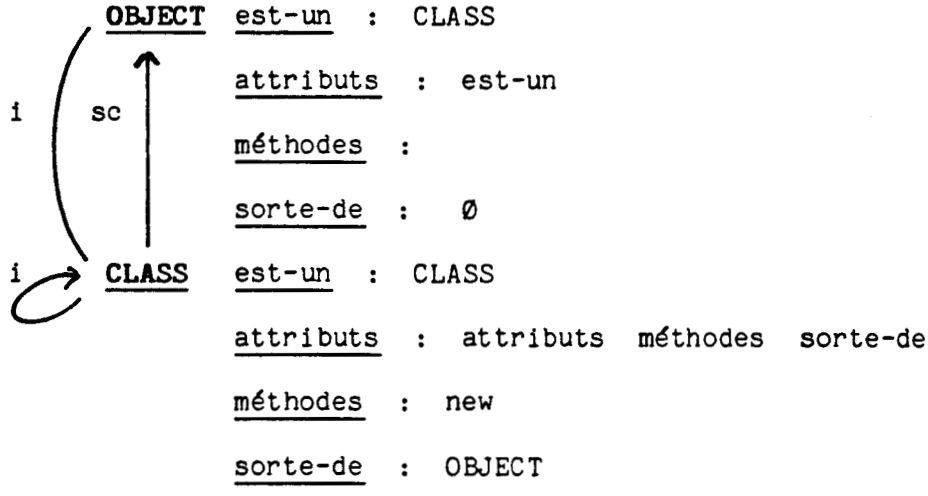
* tout objet est représentant de OBJECT
* tout objet représentant de CLASS est une classe
* tout objet non représentant de CLASS est une instance terminale.

(*) C'est ici la différence essentielle avec le modèle prototype/délégation qui sera discuté ensuite et dans lequel tout objet peut être considéré comme le prototype d'autres objets.

OBJECT et CLASS sont des classes, instances de la (meta)classe CLASS, ce qui termine le schéma métacirculaire du modèle de base :



Ainsi, en appliquant le mécanisme d'héritage, nous obtenons une description développée de ces deux objets :



Remarquons que cette description des classes de base est incomplète mais suffit à définir conceptuellement le modèle et les concepts dans leur acceptation la plus pure (*)

Les notions de classe et de métaclasse sont alors unifiées autour de ces deux classes de base, d'après les énoncés suivants :

(*) Par exemple le protocole de méthodes partagé par tout objet en SMALLTALK et défini dans OBJECT comprend des méthodes telles que "printOn:" , d'affichage, , "error:" , de retour de message d'erreur, "class" qui renvoie sa classe d'instanciation,, "isKindOf : , is MemberOf:" déjà vues... De même en Objvlisp, "is, print, error ..." sont équivalentes respectivement aux méthodes précédentes "class, printOn :, error: ...".

* toute sous-classe (*) de OBJECT caractérise des objets
* toute sous-classe de CLASS caractérise des classes
* toute sous-classe de OBJECT non sous-classe de CLASS caractérise des instances terminales.

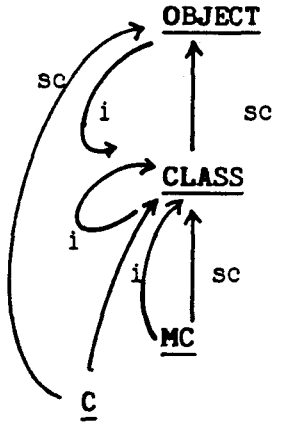
Donc, toute métaclasse est à la fois :

* représentant de CLASS, puisque c'est une classe; elle caractérise des classes qui sont des objets.
* sous-classe de CLASS, puisque les objets qu'elle caractérise sont des classes.

Enfin, toute classe non métaclasse caractérise des objets qui ne sont pas des classes, et qui sont donc des instances terminales, elle ne peut donc être sous-classe de CLASS, elle est :

* représentant de CLASS, puisque c'est une classe
* sous-classe de OBJECT et non de CLASS, puisque les objets qu'elle caractérise ne sont pas des classes mais des instances terminales.

Soient MC une métaclasse, C une classe (non métaclasse); nous pouvons résumer les liens dans le schéma suivant :



(*) sous-classe est pris généralement au sens large sauf précision quand il s'agit d'une sous-classe directe :

- une classe C est sous-classe directe de C' si C' apparaît dans son attribut sorte-de
- une classe C est sous-classe de C' si elle est sous-classe directe de C' ou de toute sous-classe de C'.

La relation "être sous-classe de" est transitive comme l'est la relation "être représentant de".

II.3.3.- Exemple

Posons tout d'abord le formalisme assez général pour l'envoi de message et les méthodes dans notre modèle informel proche de OBJVLISP

Toute méthode est définie par son nom, appelé sélecteur, et une Lambda expression Lisp qui définit notamment ses paramètres :

méthode = <sélecteur , lambda-expression>

= <sélecteur , lambda(arguments)(corps de la méthode)>

L'envoi de message se fait par l'application de la primitive send :

(send un-objet sélecteur arguments)

L'objet destinataire, un-objet, répond en appliquant la méthode associée au sélecteur, s'il en dispose.

L'affectation par un objet de ses variables d'instance se fait simplement par la primitive lisp (setq variable valeur).

L'instanciation est le mécanisme de création d'un objet par une classe, par l'envoi du message "new". L'objet créé dispose alors de la caractérisation spécifiée par cette classe et inférée par héritage sur les sur-classes. Supposons la méthode new définie comme suit :

new (lambda(valeurs)(...méthode d'instanciation...))

où valeurs est une liste d'initialisation des attributs de la forme : ... :nom_attributi vali... . En particulier, la création d'une classe, instance de la classe CLASS se fera selon la forme suivante

```
(send CLASS 'new
 :name      <nom de la classe>
 :supers    <liste des surclasses>
 :i.v       <liste des attributs>
 :méthodes <liste des méthodes>)
```

Remarquons que l'attribut "est-un" n'est jamais à préciser puisque sa valeur est la classe d'instanciation spécifiée dans le message d'instanciation (*). Ainsi la création des classes Porte, And et Or de l'exemple déjà pris en référence se fait comme suit :

```
(send CLASS 'new
 :name 'PORTE
 :supers '(OBJECT)
 :i-v '(entrée1 entrée2 sortie)
 :methods
   '(entrée1= (lambda (état) (setq entrée1 état))
     entrée2= (lambda (état) (setq entrée2 état))
     entrée1? (lambda () entrée1)
     entrée2? (lambda () entrée2)
     sortie? (lambda () sortie)
     calcule-sortie (lambda () (print "indéterminée"))))

(send CLASS 'new
 :name 'AND
 :supers '(PORTE)
 :i-v ()
 :methods
   '(calcule-sortie (lambda () (setq sortie (and entrée1 entrée2)))))

(send CLASS 'new
 :name 'OR
 :supers '(PORTE)
 :i-v ()
 :methods
   '(calcule-sortie (lambda () (setq sortie (or entrée1 entrée2)))))
```

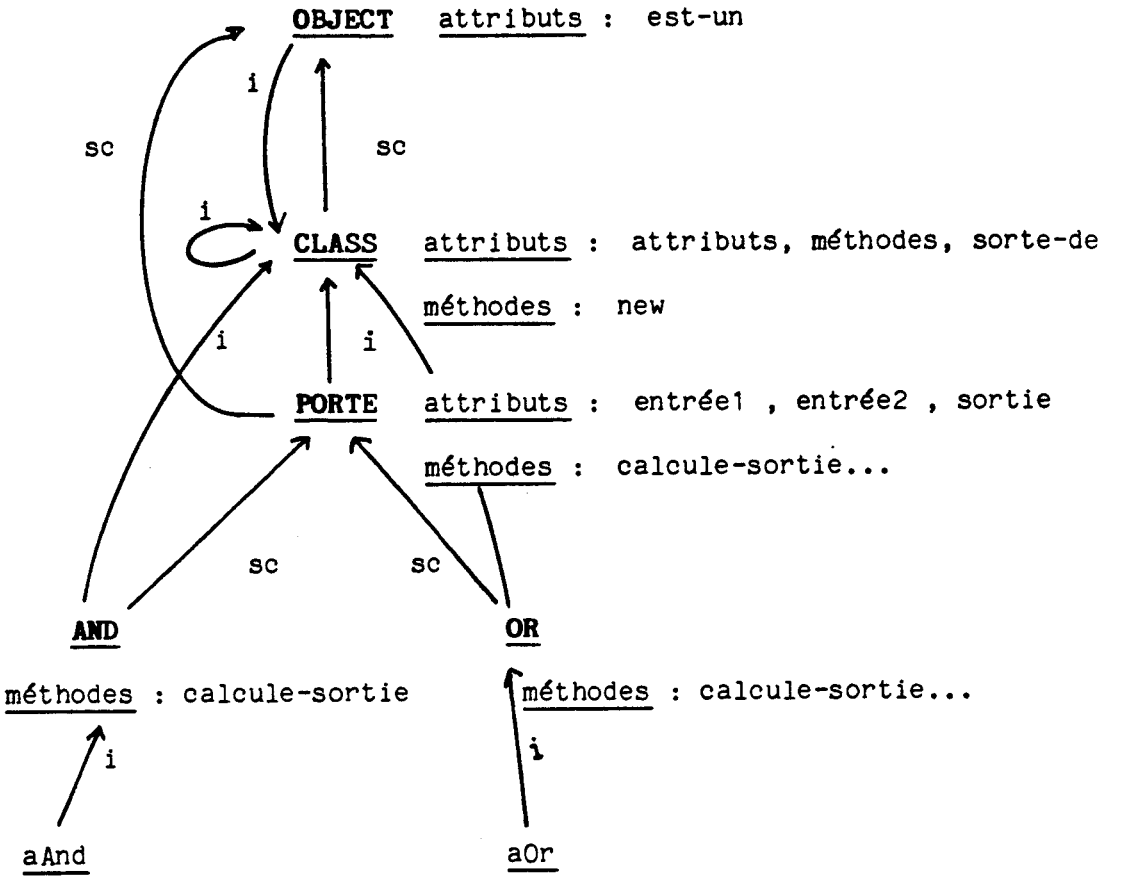
Puis l'instanciation d'un And, soit aAnd et d'un OR, soit aOR

```
(setq aAnd (send AND 'new))
(setq aOr (send OR 'new))
```

Le schéma suivant présente les différents objets définis jusqu'alors avec les liens d'instanciation et d'héritage :

(*) Il est évident que cet attribut a un statut particulier et est géré automatiquement par le système même si, conceptuellement, il s'agit d'un attribut comme un autre.

OBJVLISP gère implicitement ce "pseudo-champ" nommé 'isit' ainsi que "self" et "métaisit" definis pour tout objet et qui référencent respectivement l'objet lui-même et la métaclasse de sa classe d'instanciation.



La lecture de ce schéma montre les principes énoncés précédemment :

* tout objet est représentant de OBJECT, par exemple :

aAnd —i→ AND —sc→ PORTE —sc→ OBJECT

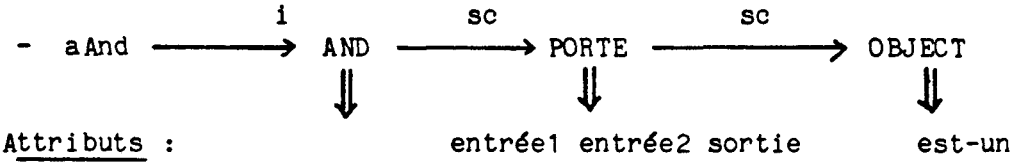
AND —i→ CLASS —sc→ OBJECT

* les objets représentants de CLASS sont des classes sinon des instances terminales :

  aAnd n'est pas une classe, il n'est pas représentant de CLASS

  AND est une classe.

* un objet a les caractéristiques définies par sa classe d'instanciation et par héritage sur les sur-classes de celle-ci, c'est-à-dire toutes les caractéristiques rencontrées sur le chemin d'héritage qui part de sa classe d'instanciation jusque l'ultime classe OBJECT dans cet ordre :

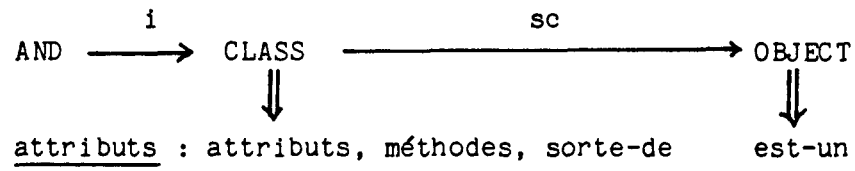


Méthodes (protocole) : calcule-sortie

notamment il n'a pas les fonctionnalités d'abstraction et de génération définies dans CLASS, c'est pourquoi c'est une instance terminale :

(send aAnd 'new) provoque une erreur du type "message : not understood" puisque new n'appartient pas à son protocole.

* AND est une classe et non une instance terminale (ce n'est notamment par un and !) caractérisée par :



méthodes : new

* tout objet est instance d'une classe. Tout objet étant représentant de OBJECT il est caractérisé par l'attribut est-un qui le lie à sa classe d'instanciation (i)

Donnons maintenant un exemple d'utilisation de self-messages. Supposons que l'on veuille qu'à chaque écriture des entrées 1 et 2 la porte calcule automatiquement sa sortie, c'est-à-dire fasse appel à sa méthode calcule-sortie. Les méthodes d'écriture entrée1= et entrée2= se définissent donc simplement comme suit dans la classe porte (AND et OR restent inchangées) :

```
(send CLASS 'new
 :name 'PORTE
 :supers '(OBJECT)
 :i-v '(entrée1 entrée2 sortie)
 :methods
  '(entrée1= (lambda (état) (setq entrée1 état)
                            (send self 'calcule-sortie))
    entrée2= (lambda (état) (setq entrée2 état)
                            (send self 'calcule-sortie))
    entrée1? (lambda () entrée1)
    entrée2? (lambda () entrée2)
    sortie? (lambda () sortie2)
    calcule-sortie (lambda () (print "indéterminée"))))
```

Insistons sur le fait que l'héritage garantissant que c'est toujours la méthode la plus affinée qui est appliquée, pour des instances des sous-classes AND et OR, les méthodes respectives calcule-sortie seront prises en compte. Par exemple l'envoi de message

(send aAnd 'entrée1= t)

où aAnd est instance de And, résultera en l'application de la méthode entrée1= définie dans porte qui fera appel à la méthode calcule-sortie définie dans AND..

Enfin, voici deux exemples classiques de métaclasses pour définir des classes particulières. Dans le premier exemple, nous voulons rapprocher la notion de classe de la notion d'ensemble en définissant les classes qui conservent les instances qu'elles ont créées dans une liste. Une telle classe sera donc caractérisée par une métaclasse CLASS-ENS définissant :

* un attribut appelé instances dont la valeur sera la liste de ses instances

* une nouvelle méthode new (*1) qui, après instanciation, (send self 'new...) ajoutera à la liste instances la nouvelle instance crée.

D'où la création de CLASS-ENS

```
(send CLASS 'new
 :name 'CLASS-ENS
 :supers '(CLASS)
 :i-v '(instances)
 :methods
  '(newe (lambda ()
    (let ((inst (send self 'new))) (newl instances inst) inst))))
```

Nous pouvons alors définir la classe AND qui conservera tous les and créés comme suit (en supposant que la classe Porte (*2) reste définie comme précédemment) :

```
(send CLASS-ENS 'new
 :name 'AND
 :supers '(PORTE)
 :i-v ()
 :methods
   '(calcule-sortie (lambda () (setq sortie (and entrée1 entrée2)))))
```

(*1) pour les spécialistes, nous n'introduirons le super que dans le paragraphe III.3.4.2.

(*2) La classe Porte est une classe abstraite : comme nous le verrons par la suite, elle n'est donc pas susceptible d'être instanciée. il n'y a donc pas d'intérêt à la définir comme instance de CLASS-ENS. On pourrait cependant, en supposant qu'une classe ait connaissance de ses sous-classes, fonctionnalité qui n'a pas été introduite ici, étendre CLASS-ENS pour qu'une classe puisse connaître tous ses représentants au sens large, c'est-à-dire ses instances et les instances de ses sous-classes... (Voir par exemple la métaclasse E-CLASS de FOVL [CARRE.88]

Après la création de deux ands and1 et and2 par :

```
(setq and1 (send AND 'newe))

(setq and2 (send AND 'newe))
```

l'état de la classe And sera :

AND

méthodes : (calcule-sortie ...)

sorte-de : PORTE

instances : (and1 and2)

Le deuxième exemple est la définition des classes qui initialisent leurs instances par une métaclasse CLASS_INIT définissant la méthode newi qui envoie le message init à son instance nouvellement créée. Ces classes devront donc définir le message d'initialisation, init, pour leurs instances.

```
(send CLASS 'new
 :name 'CLASS_INIT
 :supers '(CLASS)
 :i-v ()
 :methods
  '(newi (lambda () (send(send self 'new) 'init))));  (*)
```

La classe And peut alors être définie comme suit :

```
(send CLASS_INIT 'new
 :name 'AND
 :supers '(PORTE)
 :i-v ()
 :methods
  '(calcule-sortie (lambda () (setq sortie (and entrée1 entrée2)))
    init
     (lambda ()
      (setq entrée1 t)
      (setq entrée2 t)
      (setq sortie t)
      self)))
```

La création avec initialisation d'un and par :

```
(setq aAnd (send AND 'newi))
```

provoquera l'initialisation du and :

And1

entrée1 true
entrée2 true
sortie true.

(*) Il s'agit là d'un enchaînement de messages (send self 'new) renvoie l'instance créée à laquelle on envoie le message 'init.

II.4.- Controverse classe/prototype

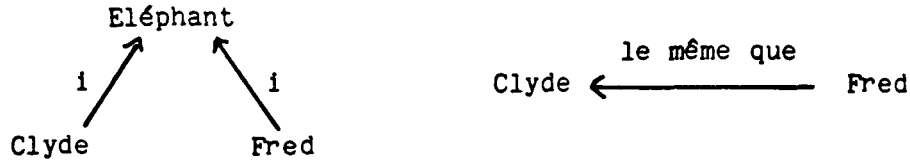
Le modèle classe/instance/héritage présenté jusqu'à maintenant est à opposer au modèle prototype/délégation adopté notamment par les langages d'acteurs [LIEBERMAN. 86-a] [AGHA. 86]. Ce modèle ne sera pas présenté en détail étant donné que notre étude est centrée sur la première approche. Nous allons seulement le présenter intuitivement à travers la controverse classe/prototype qui a des fondements conceptuels et philosophiques discutés dans [LIEBERMAN. 86-b]. Le problème est celui de la représentation d'objets semblables et notamment le partage de connaissance entre ces entités. Reprenons l'exemple de LIEBERMAN à propos des éléphants Clyde et Fred. Supposons que Clyde soit le premier éléphant que vous rencontrez et que vous pouvez caractériser par sa couleur, sa taille, la longueur de sa trompe, le nom que vous lui donnez.... Puis vous rencontrez un deuxième éléphant, que vous appelez Fred qui ressemble fortement à Clyde quant aux caractéristiques qui le décrivent mais diffère par les valeurs associées à ces caractéristiques. Quelles sont alors les représentations mentales de ces deux éléphants et notamment comment partagent-ils le même schéma pour se caractériser ?

Selon le modèle basé sur la notion de classe, tous ces éléphants peuvent être regroupés autour du concept abstrait "Eléphant" représenté par la classe du même nom. Cette classe spécifie une caractérisation générale des éléphants dont les caractéristiques précédentes, la couleur, la taille, la longueur de la trompe, son nom,... Clyde et Fred sont alors vus comme des représentants (des exemplaires, des concrétisations) de cette classe.

L'approche par prototype, par contre, suit l'expérience que vous avez eue du concept d'éléphant. Clyde étant le premier éléphant que vous avez rencontré, il lui est immédiatement associé une entité, un objet caractérisé comme précédemment avec ses valeurs propres associées. Jusque là c'est le seul objet de ce type (!) que vous connaissez et il correspond à l'image mentale du concept d'éléphant sans que ce concept abstrait ait une existence indépendamment de l'entité concrète qu'est Clyde. A la rencontre de Fred, vous vous direz "tiens, cette chose ressemble fortement à Clyde" et établirez un lien entre Fred, un nouvel objet, et Clyde, parce qu'il partage les mêmes caractéristiques que ce dernier aux valeurs près qui seront définies dans l'objet Fred. Clyde apparaît alors comme le prototype de Fred et de toutes les autres choses qui lui ressemblent, appelées des extensions. Une différence importante est donc que l'approche par classe impose la création du concept abstrait avant les exemplaires concrets que seront ses instances inversement à l'approche par prototype.

Cette dernière approche conduit à un modèle orienté objet à un seul niveau d'entités, les objets, sans introduire la notion de classe : un objet peut être créé directement, et existe indépendamment de toute abstraction que serait la classe. Par ailleurs, tout objet est potentiellement un prototype d'autres objets qui partagent une partie de sa caractérisation. Inversement, dans le modèle centré classe, un objet n'existe pas sans sa classe d'instanciation.Les classes sont des objets particuliers capables de caractériser d'autres objets.

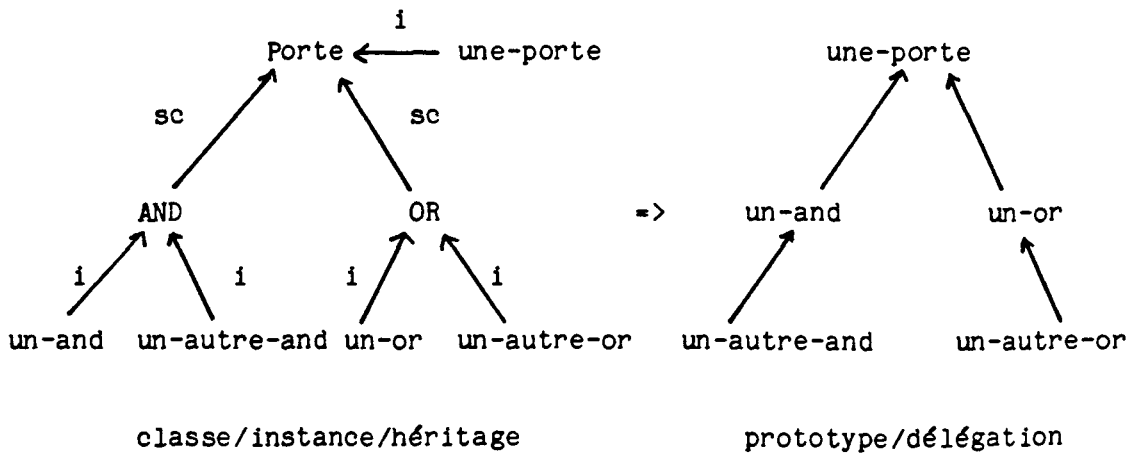
Un objet est relié à ses prototypes par un lien du type "je suis le même que". Ce lien direct entre deux objets semblables n'existe pas dans notre modèle mais se traduit par l'énoncé "je suis instance de la même classe que", les deux objets ayant un lien "est-un" (i) vers la même classe :



Dans l'approche par prototype, le mécanisme de partage de connaissances entre les objets est la délégation. Tout objet est caractérisé par la liste de ses prototypes en tant qu'extension de ceux-ci et une partie personnelle qui contient ses propres caractéristiques. Le principe de la délégation est le suivant :

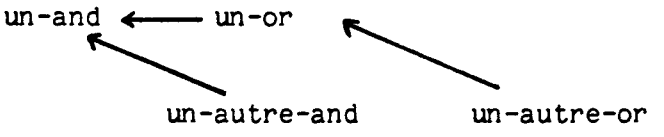
Quand un objet extension reçoit un message, il essaie d'abord d'y répondre en utilisant sa connaissance personnelle, sinon il reporte (délègue) le message sur ses prototypes et récursivement.

Par exemple, supposons que Clyde ait quatre pattes et que Fred n'ait pas redéfini en propre cette caractéristique. Si l'on demande à Fred combien il a de pattes, il répondra 4 après avoir reporté la question à son prototype Clyde puisque lui-même ne peut y répondre. Par contre, alors que Clyde, a la couleur grise, supposons que Fred ait sa couleur propre, rose, définie dans sa partie personnelle. Fred répondra rose directement si on lui demande sa couleur. Ainsi ce mécanisme de partage est basé essentiellement sur la caractérisation par défaut, les caractéristiques d'une extension sont par défaut celles de son prototype. L'approche par classe est basée sur la caractérisation abstraite, deux instances d'une même classe partagent exactement le même modèle abstrait spécifié par celle-ci. Enfin, il est assez intuitif que l'héritage entre les classes est remplacé ici par la délégation entre les objets(*). Pour constater ceci il suffit d'associer à chaque classe un prototype qui est l'une de ses instances et de substituer les liens "sorte-de" et "est-un" par l'unique lien de prototype. Reprenons une partie de notre exemple sur les portes :



(*) Insistons cependant sur le fait que la délégation induit un mécanisme de propagation de valeur d'instance entre un prototype et ses extensions qui n'a pas d'équivalent immédiat avec l'héritage.

Ce schéma est cependant la traduction d'une abstraction a priori. Il ne montre pas l'approche par expérimentation des concepts propres au modèle par prototype. L'expérimentation des objets dans l'ordre un-and , un-or , un-autre-and , un-autre-or conduirait par exemple tout simplement au schéma :



où, notamment, un-or redéfinirait la méthode calcule-sortie par défaut de son prototype un-and.

En conclusion de cette controverse, insistons sur le fait que l'approche par prototype suit l'expérience de concepts et peut paraître plus adaptée aux problèmes d'apprentissage :

> "people seem to be a lot better at dealing with specific examples first, then generalizing from them than they are at absorbing general abstract principles first".
> [LIEBERMAN. 86-b]

Comme nous l'avons précisé, ce modèle est plus proche de langages d'acteurs comme Act1 [LIEBERMAN. 81] et des travaux sur la programmation concurrente et le parallélisme comme ABCL [YONEZAWA and al. 86].

Enfin, remarquons que les deux notions de classe et de prototype sont complémentaires et ont été utilisées conjointement, notamment pour représenter les valeurs par défaut. En Thinglab [BORNING. 79] par exemple, à toute classe peut être associé un prototype, une instance type dont les valeurs d'instance de ses attributs sont les valeurs par défaut pour les autres instances.

Dans [MITTAL and al. 86] la notion de prototype est utilisée pour le développement de versions successives d'un problème en loops [BOBROW-STEFIK. 81] langage bâti sur le modèle classe/instance/héritage.

Le modèle prototype/délégation ne sera pas développé plus en détail et, sauf précision, le modèle classe/instance/héritage sera toujours notre référence de base.

II.5.- LES OUTILS ORIENTES OBJET

OBJVLISP permet d'introduire de manière uniforme toute une famille d'outils bâtis sur le modèle classe/instance/héritage dont les caractéristiques sont les suivantes :

* L'objet est l'entité de représentation de base.

* La notion de classe est utilisée pour définir les caractéristiques d'objets semblables.

* Les classes sont reliées entre elles par une relation "sorte-de" de généralisation/affinement qui induit un mécanisme d'inférence des caractéristiques qu'elles définissent, l'héritage. Cette relation peut être monovaluée (une classe n'a qu'une sur-classe directe), les classes formant alors un arbre de sommet la classe la plus générale Object, et l'héritage est dit simple. Elle peut être multivaluée (une classe peut avoir plusieurs sur-classes directes), les classes forment alors un treillis (*) de sommet object et l'héritage est dit multiple.

* Un objet est créé par instanciation d'une classe, ce qui détermine ses caractéristiques définies par celle-ci et inférées par héritage sur ses sur-classes.

Le premier de ces outils déjà cité est SIMULA 67 qui introduisit les notions d'objet, de classe, d'instance, de sous-classe et d'héritage simple par préfixage d'une classe par une sur-classe. L'héritage multiple a été étudié par la suite en SIMULA dans [KROGDHAL. 85]. La programmation par envoi de message a été introduite par SMALLTALK qui connut trois versions successives SMALLTALK 72, 76, 80. On pourra trouver dans [KRASNER. 83] une présentation de son évolution et ses différentes implémentations. Nous avons vu également comment l'application du principe d'uniformité a conduit à la notion de métaclasse unifiée ensuite avec celle de classe en Objvlisp. L'héritage en SMALLTALK.80 standard est simple mais plusieurs travaux expérimentent l'héritage multiple comme [GOLDSTEIN-BOBROW. 80] et [BORNING-INGALLS. 82].

SMALLTALK est aujourd'hui l'un des langages orientés objet les plus utilisés et souvent pris comme paradigme de l'approche orientée objet. Le soin de sa conception métacirculaire et donc orientée objet est une preuve en soi de la puissance de cette approche. Ce langage offre à l'utilisateur un environnement graphique très convivial et à la base une hiérarchie d'une centaine de classes qui en fait un environnement de programmation très riche.

De nombreux outils orientés objet ont été implémentés au dessus de Lisp, en tant qu'outils privilégiés pour le développement d'autres langages :

(*) C'est le terme habituellement utilisé, il s'agit, en fait, d'une structure munie d'un élément maximal, OBJECT.

CHAPITRE III

*****

III - ETUDE METHODOLOGIQUE DE L'APPROCHE ORIENTEE OBJET

Après cette présentation générale des concepts du paradigme orienté objet, nous allons en donner une caractérisation de nature conceptuelle, méthodologique et technique. Comme nous l'avons précisé au paragraphe II.1, l'approche orientée objet est un sujet en pleine évolution et ce chapitre a pour but de présenter l'état de certains thèmes, sans avoir l'ambition d'être exhaustif. Cette synthèse nous permettra de faire référence à ces aspects dans le chapitre IV qui présente notre modèle orienté objet ROME qui expérimente des concepts originaux pour la représentation des connaissances.

III.1.- MODULARITE

La modularité est une propriété communément admise comme nécessaire pour la représentation de systèmes de connaissance au sens large suivant :

> "A system is a part of the world that a person (or group of persons) chooses to regard as a whole consisting of components, each component characterized by properties that are selected as being relevent and by actions related to these properties and those of other components";
>
> [NYGAARD. 86]

Elle permet de maîtriser la complexité en rendant locale, interne, à un module ses propres caractéristiques et fonctionnalités dont il est responsable et dont il gère la cohérence. Le principe de modularité est énoncé comme suit par INGALLS :

> "THE PRINCIPLE OF MODULARITY
> No part of a complex system should depend on the internal details of any other part".
>
> [INGALLS. 78]

L'approche modulaire fait une forte distinction entre l'aspect interne et l'aspect externe d'un module. L'aspect interne traduit son implémentation c'est-à-dire comment il réalise les fonctionnalités qui

le caractérisent. Cette représentation interne lui est propre et est encapsulée, cachée pour l'extérieur, ce qui lui permet d'en assurer la cohérence. L'aspect externe est l'ensemble des requêtes qu'un autre module peut lui adresser et représente donc son interface avec l'extérieur. Cette vue externe du module constitue donc l'abstraction de son implémentation interne.

Idéalement, un module est donc une boîte noire encapsulant son implémentation et vue de l'extérieur par son abstraction. Ce principe est à la base des outils de représentation modulaire que l'on pourrait qualifier de "orientée module", dont la représentation orientée objet n'est qu'un modèle (*). Elle se retrouve dans des outils tels que ADA, MODULA 2 ou CLU [BISHOP.86], [LE VERRAND.82], [LISKOV and al. 77] en informatique ou dans les outils de CAO [BEGG.84] tels qu'en conception hiérarchique de VLSI [NIESSEN.86] où les propriétés d'abstraction et d'encapsulation sont requises :

> "An abstraction created for some module M serves as an interface to all parent modules P which may use M . The abstraction should only include that information relevant for the interaction between module M and its parent... As a consequence the determination of the correctness of module detail of M should be done within the module M ... The points mentioned,
> - abstraction in intermodular interface and thus determined by mutual negociation,
> - absence of interference from a parent module with the module M other than via the abstraction and vice versa from a module to its parent,
> - local determination of correctness within M
>
> can be considered to be a requirement specification for the definition or required primitives for abstraction".
>
> [NIESSEN.86]

Nous allons voir qu'à la base, l'objet des langages orientés objet répond à ces spécifications et apparaît donc comme un bon outil pour la décomposition modulaire.

(*) Peut-être par phénomène de mode, le terme objet tend même à remplacer le terme module et on parle d' "orienté objet" dès qu'il y a modularité. Cette assimilation mène à des rapprochements tels que ADA et les langages orientés objet [PITETTE.86] [KROGDHAL-OLSEN.86] et montre toute l'importance du courant de pensée orienté objet.

III.2.- ENCAPSULATION ET ABSTRACTION

> "Objects provide an elegant mechanism for modular decomposition of real world systems".
>
> [AGHA.86]

où système doit être pris au sens très général de NYGAARD défini précédemment. Le principe de modularité énoncé par INGALLS pour SMALLTALK est garanti par l'encapsulation définie ainsi par SNYDER :

> "Encapsulation is a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces".
>
> [SNYDER.86b]

Elle se révèle dans le concept d'objet, en tant qu'incarnation de la notion de module, à travers la séparation des caractérisations interne et externe. **La caractérisation interne formée des attributs et des méthodes est strictement encapsulée dans l'objet. La caractérisation externe est l'ensemble des sélecteurs de méthodes appelée protocole ou interface.** La "manipulation" d'un objet, ou plutôt le dialogue avec un objet n'est possible qu'à travers son protocole par le mécanisme d'envoi de message qui spécifie un sélecteur ("quoi"), l'objet répond en appliquant localement sa méthode ("comment") associée. En particulier, la manipulation d'un attribut n'est possible que si une méthode correspondante est définie, l'objet peut ainsi assurer sa cohérence interne en permettant un accès contrôlé à ses caractéristiques. D'autre part, cette approche permet de rendre indépendantes les spécifications externes (le protocole) de l'implémentation interne qui peut prendre des formes différentes selon des choix de conception différents. Indépendamment de son implémentation, l'objet est vu de ses utilisateurs, les clients, à travers la même interface, abstraction de ses implémentations possibles :

> "The external interface of a module serves as a contract between the module and its clients... If clients depend only on the external interface, the module can be implemented without affecting any clients, so long as the new implementation supports the same (or an upward compatible) interface".
>
> [SNYDER.86b]

Considérons par exemple une réalisation des points qui peuvent être représentés en coordonnées cartésiennes ou polaires. Supposons les spécifications externes d'un point définies par les sélecteurs suivants en SMALLTALK (*) :

(*) Les sélecteurs en SMALLTALK sont de 3 types.
Les sélecteurs unaires sans argument sont simplement des mots, par exemple x,y,z , angle; les sélecteurs binaires à un argument sont des mots terminés par :, par ex. at: de lecture d'un élément d'une collection : "un-tableau at:2", ou les sélecteurs déjà vus isKindOf: une classe; is MemberOf: une classe;pour les sélecteurs à mots clés (keywords sélectors) à plus de 2 arguments, chaque argument est précédé d'un mot-clé de la forme mot terminé par : , par ex. $x:x_0$; $y:y_0$ ou at: indice put: valeur d'écriture dans une case d'une collection.

x: x_0 y: y_0 affectation de ses coordonnées cartésiennes

r: r_0 angle: a_0 -"- -"- polaires

x quelle est ton abscisse ?

y quelle est ton ordonnée ?

r quel est ton rayon ?

angle quel est ton angle ?

Comment implémenter les objets points répondant à ce prototole ?

Plusieurs implémentations correspondant exactement aux mêmes spécifications sont possibles. Une première solution est de caractériser un point par les attributs x,y,r, angle représentant respectivement ses coordonnées cartésiennes et polaires. Sachant les équations de passage, la cohérence interne sera assurée par la mise à jour des coordonnées polaires quand les coordonnées cartésiennes sont affectées par la méthode implémentant le sélecteur x : x_0 y : y_0 et inversement. Une première version de la classe Pt(point) est donc :

```
Object subclass: #Pt
        instanceVariableNames: 'x y r angle '

Pt comment:
'Pt caracterise les points representes en coordonnees cartesiennes et polaires.'


Pt methodsFor: 'accessing'

angle
        "renvoie son angle"

        ^angle

r
        " renvoie son rayon"

        ^r

r: r0 angle: a0
        "affectation de r et angle et propagation de x et y"

        r <- r0.
        angle <- a0.
        x <- r0 * a0 cos.
        y <- r0 * a0 sin

x
        " renvoie son absice"

        ^x

x: x0 y: y0
        "affectation de x et y et propagation de r et angle"

        x <- x0.
        y <- y0.
        r <- (x0 squared + y0 squared) sqrt.
        x0 ~= 0
                ifTrue: [angle <- (y0 / x0) arcTan]
                ifFalse: [y0 = 0
                                ifTrue: [angle <- 0.0]
                                ifFalse: [y0 < 0
                                        ifTrue: [angle <- Pi / 2 negated]
                                        ifFalse: [angle <- Pi / 2]]]

y
        "renvoie son ordonnee"

        ^y
```

Cette version montre que la cohérence interne est assurée par la stricte encapsulation des attributs x , y , r , angle. Leur accès en écriture, notamment, n'est permis qu'à travers les messages $x:x_0$ $y:y_0$ et $r:r_0$ $angle:a_0$ dont les méthodes associées garantissent leur équivalence.

Une deuxième solution repose sur le fait que, puisque l'une des représentations peut être déduite de l'autre, il suffit d'en choisir une pour implémenter le point, soit l'implémentation en coordonnées cartésiennes (PtC) :

```
Object subclass: #PtC
        instanceVariableNames: 'x y '
PtC comment:
'PtC (C comme Cartesien) caracterise les points, repondant au meme protocole que
 les representants de Pt, mais implementes en coordonnees cartesiennes.'

PtC methodsFor: 'accessing'

angle
        "lecture virtuelle de l'angle, calcule"

        x ~= 0
                ifTrue: [^(y / x) arcTan]
                ifFalse: [y = 0
                                ifTrue: [^0.0]
                                ifFalse: [y < 0
                                                ifTrue: [^Pi / 2 negated]
                                                ifFalse: [^Pi / 2]]]

r
        " lecture virtuelle de r, calculee"

        ^(x square + y square) sqrt

r: r0 angle: a0
        " tranformation en x et y"

        x <- r0 * a0 cos.
        y <- r0 * a0 sin

x
        " renvoie x"

        ^x

x: x0 y: y0
        " simplement affectation de x et y"

        x <- x0.
        y <- y0

y
        " renvoie y"

        ^y
```

De façon similaire, le choix de l'implémentation en coordonnées polaires pourrait être fait. Ici, en particulier, les attributs correspondant aux coordonnées polaires n'existent pas et leur "lecture virtuelle" est réalisée par les méthodes associées aux sélecteurs r et angle.

Toutes ces versions de Point implémentent le même protocole indépendamment de la représentation interne. Les "clients" de tels points n'ont pas à connaître l'implémentation effective dans l'un ou l'autre ou les deux repères.

L'uniformité du dialogue avec un objet uniquement à travers son protocole est une garantie de l'indépendance spécifications externes/implémentation effective et donc de l'abstraction. En particulier, l'accès à un attribut peut être virtuel dans le sens où celui-ci n'existe pas concrètement sans que le client le sache. D'autre part, si l'attribut existe effectivement, son accès par méthode permet à l'objet de gérer sa cohérence ou d'effectuer des vérifications. Il est possible ainsi dans les langages non typés tels que SMALLTALK, d'effectuer un typage dynamique des attributs lors de leurs manipulations. En SMALLTALK par exemple, les messages isMemberOf:une-classe et isKindOf:une-classe compris par tout objet (ces méthodes sont définies dans Object) permettent de tester son appartenance (directe ou indirecte) à une-classe. Ainsi, l'écriture effective des attributs x et y par x_0 et y_0 peut être conditionnée dans la méthode x:x_0 y:y_0 par leur appartenance à la classe Float :

```
x:x0  y:y0

((x0  isKindOf: Float) and : [y0 isKindOf: Float])

      ifTrue:  [ x ← x0 . y ← y0...]

      ifFalse: [self error: 'arguments non Float!']
```

L'accessibilité des attributs par méthodes est donc primordiale dans ces langages, ne serait-ce que vis à vis de ces vérifications explicites de type.

Bien que nous ayons axé notre présentation de l'encapsulation des attributs sur une approche uniforme de type SMALLTALK, celle-ci n'est pas la solution adoptée par tous les outils orientés objet. Nous pouvons distinguer deux types de solutions que nous appellerons respectivement non-accessibilité et accessibilité par défaut des attributs.

La non-accessibilité par défaut est l'approche de type SMALLTALK (*)

(*) Pour les spécialistes, nous ne tiendrons pas compte des sélecteurs instVarAt: et instVarAt: put: définis pour tout objet et réservés à des usagers de type système (debug...) ou pour le développement d'applications très spécifiques.

que nous avons présentée : un attribut n'est accessible que si une méthode correspondante est définie. Certains langages comme Flavors ou Common Objects permettent de déclarer un attribut accessible en lecture ou en écriture (:readable-instance-variables, :writable-instance-variables, ou même initable-instance-variables en Flavors) et génèrent automatiquement les méthodes d'accès correspondantes selon un sucre syntaxique fonction du nom de l'attribut (le nom lui-même pour la lecture en Flavors par exemple). Ceci est satisfaisant si ces méthodes sont traitées comme toute autre et peuvent ainsi être modifiées pour garantir la cohérence ou effectuer des vérifications telles que le typage (*).

L'accessibilité par défaut considère que tout attribut est accessible par défaut; il faut alors un mécanisme pour contrôler ou interdire cet accès si nécessaire. En Treillis/Owl par exemple pour tout attribut, soit att, d'un objet sont définies deux méthodes (opérations) de lecture et écriture de la forme : get_att(me) et put_att(me, value : type) (où "me" correspond à self). Ce langage garantissant un contrôle de type, ces méthodes suffisent dans bien des cas où l'attribut est effectivement accessible et qu'il n'y a pas d'autres vérifications ou actions à effectuer. Par contre, si la simple méthode de lecture ou d'écriture ne suffit pas, comme pour notre première version de Point, l'opération peut être redéfinie. Enfin, si l'attribut n'est pas accessible en lecture ou en écriture, il est possible de déclarer ses opérations get et put privées (private) à l'objet concerné. D'autres mécanismes de contrôle de l'accessibilité par défaut sont implémentées par les outils orientés objet comme les valeurs actives de loops et Kee, les attachements procéduraux ou les facettes de typage et les réflexes des frames. Nous ne détaillerons pas ces dernières techniques qui seront exposées au paragraphe III.4.4. Retenons simplement que si l'accessibilité par défaut des attributs est autorisée, il faut des mécanismes pour la contrôler et assurer la cohérence interne de l'objet.

(*) Nous verrons en particulier, au paragraphe III.3.4.3., qu'en Flavors, il est possible d'attacher des méthodes "before" et "after" qui sont appliquées respectivement avant et après l'application de la méthode elle-même ("main").
La méthode "before" peut être typiquement utilisée pour les vérifications de type ou de contraintes avant écriture d'un attribut. La méthode "after" permet de propager d'autres actions comme la mise à jour des coordonnées polaires (resp. cartésiennes) après écriture des coordonnées cartésiennes (resp. polaires) dans notre première version de Point.

III.3.- HIERARCHIE CONCEPTUELLE : L'HERITAGE

III.3.1.- Fondements

Nous avons vu que dans le modèle de base présenté au chapitre II :

* tout objet est instance d'une classe

* toute classe est sous-classe d'autres classes, au minimum OBJECT.

Ainsi, tout objet est relié à sa classe d'instanciation par son attribut "est-un" et toute classe à ses sur-classes par son attribut "sorte-de". Ce dernier lien construit ainsi une hiérarchie de classes que nous appelons hiérarchie conceptuelle sur laquelle agit le mécanisme d'inférence des caractéristiques appelé héritage :

> "inheritance provides a means of conceptually organizing information about the world."
>
> [AGHA.86]

Nous allons détailler cet aspect du modèle des points de vue conceptuel, technique et méthodologique. Nous verrons notamment les nombreuses interprétations que l'on peut donner à cette hiérarchie.

Les deux liens précédents ("est-un" et "sorte-de") sont deux formes d'un lien généralement appelé IS-A bien connu en représentation des connaissances et qui est né dès les réseaux sémantiques

> "The idea of IS-A is quite simple. Early in the history of semantic nets, researchers observed that much representation of the world was concerned with the conceptual relation expressed in English sentences such as "John is a bachelor" and "A dog is a domesticated carnivorous mammal". That is, two predominant forms of statements handled by A.I. knowledge representation systems were the predication, expressing that an individual (e.g. John) was of a certain type (e.g. bachelor), and the universally quantified conditional, expressing that one type (e.g. dog) was a subtype of another (e.g. mammal). The easiest way to get such statements into a semantic-net scheme was to have a link that directly represented the "is-a" parts of such sentences. Thus, the IS-A link was born".
>
> [BRACHMAN.83]

Dans de tels énoncés apparaissent dès lors deux types d'entités que l'on peut qualifier de concrètes et abstraites. Les **entités concrètes**, par exemple John, sont nos instances terminales et sont appelées par ailleurs "individuals", concretisations. Les **entités abstraites**, par exemple bachelor, dog, mammal, sont nos classes, ou types, "generics". On associe ainsi le lien "est-un" à "prédication" et "sorte-de" à "universally quantified conditional".

Il existe de nombreuses interprétations du lien IS-A [BRACHMAN.83], comme on pourra s'en rendre compte à travers ce chapitre. On peut cependant distinguer deux interprétations fondamentales, qui aident fortement à la compréhension et à l'utilisation des notions de classe, sous-classe et héritage.

L'**interprétation ensembliste** associe à une classe l'ensemble de ses représentants au sens de la définition donnée au paragraphe II.3.1. Elle est le regroupement des objets semblables qu'elle caractérise. La relation représentant → classe traduit l'appartenance et la relation sous-classe → sur-classe traduit l'inclusion, ordre partiel sur l'ensemble des représentants d'une sur-classe, à la sur-classe étant associée l'union des ensembles des représentants de ses sous-classes.

Soit O un objet et C , C' deux classes :

O est représentant de $C \equiv O \in C$

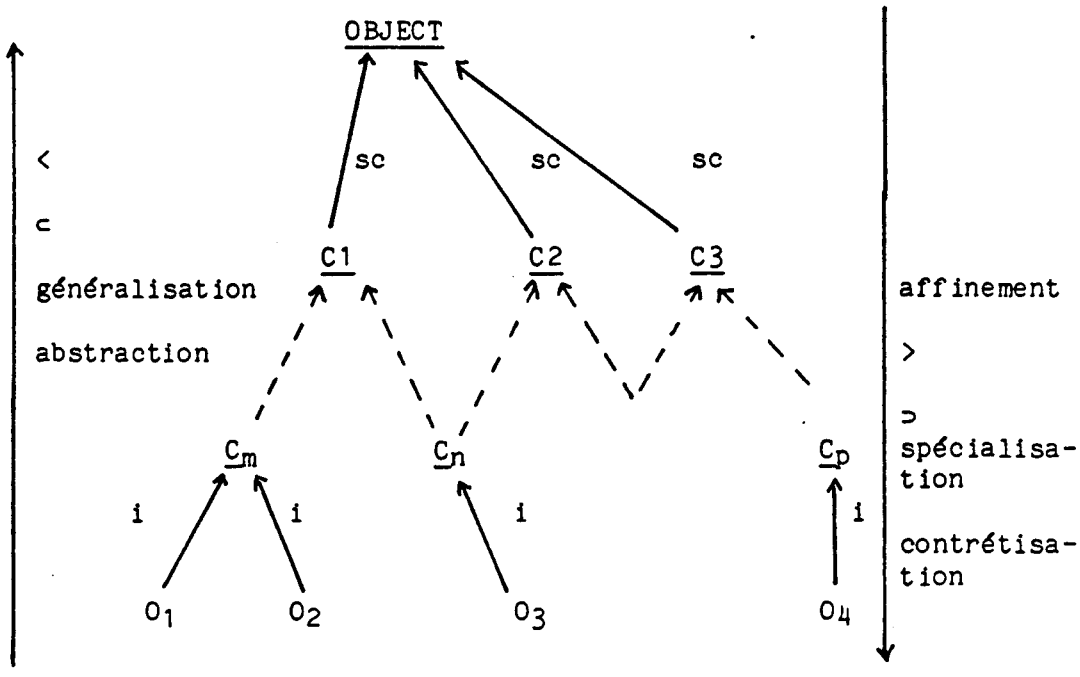
C est sous-classe de $C' \equiv C \subset C' \equiv C < C'$.

Cette interprétation assez intuitive est à la base de modèles algébriques des langages orientés objet [CARDELLI-WEGNER.85], [BRUCE-WEGNER.86].

L'**interprétation conceptuelle** considère la classe comme un concept général et abstrait, abstraction d'objets semblables, ses représentants. Elle spécifie le modèle, le "moule" [BRIOT.85] de ces objets, c'est-à-dire la structure commune qui définit les caractéristiques partagées par ceux-ci, la relation sous-classe → sur-classe s'interprète alors comme la généralisation, l'abstraction, inversement elle traduit l'affinement, la spécialisation de concepts. Classe et sous-classe sont donc des outils d'abstraction des langages orientés objet comme nous le verrons au paragraphe III.3.5.2 :

> "A programming language contains abstraction mechanisms-tools for describing the common structure of similar phenomena";
> [NYGAARD.86]

L'ordre partiel, défini entre les classes, d'inclusion des ensembles d'objets associés ou de généralisation/affinement de concept induit dans sa version la plus simple une hiérarchie arborescente des classes. Dans cette version, une classe est sous-classe d'une seule autre classe et le mécanisme d'inférence des caractéristiques est l'héritage simple. Le sommet de la hiérarchie est la classe OBJECT, ensemble de tous les objets ou concept le plus général. C'est le schéma de l'arbre d'héritage en SIMULA ou en SMALLTALK Standard.



"It was quickly noted that the IS-A connections formed a hierarchy (or, in some cases, a lattice) of the types being connected - that is, the IS-A relation is roughly a partial order".

[BRACHMAN.83]

Une version plus générale, mais aussi plus problématique est l'extension de l'organisation arborescente à l'arrangement en treillis et de l'héritage simple à l'héritage multiple introduit par Flavors. Dans cette version, une classe peut être sous-classe (directe) de plusieurs autres classes.

Selon l'interprétation ensembliste, un objet peut ainsi être représentant de plusieurs classes sur-classes de sa classe d'instanciation.

"A class may have any number of super-classes; however an instance is always instance of precisely one class".

[BORNING-INGALLS.82] (*)

(*) où le terme instance doit être remplacé par la notion de représentant selon nos définitions. En effet, un objet reste instance (au sens de l'instanciation) d'une seule classe, sa classe d'instanciation qui peut avoir plusieurs sur-classes (directes). C'est par sa classe d'instanciation que l'objet est représentant de plusieurs classes. Cette approche pose des problèmes non négligeables qui seront présentés au chapitre IV et qui sont à la base de notre langage ROME.

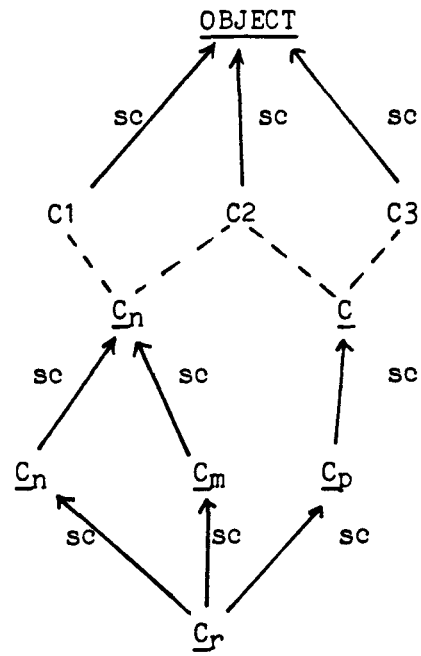
Une classe apparaît alors comme l'intersection de ses super-classes.

Selon l'interprétation conceptuelle, l'héritage multiple permet la composition de concepts de plus haut niveau et des caractéristiques qu'ils spécifient :

> "Multiple inheritance increases sharing by making it possible to combine description from several classes".
> [STEFIK-BOBROW.85]

Une sous-classe est l'affinement de chacune de ses sur-classes prise individuellement par combinaison avec les autres sur-classes.

Le schéma de représentation est alors un treillis (*) de sommet toujours la classe OBJECT



(*) ou plus exactement un sup demi treillis. En général, les structures d'ensembles de classes proposées n'ont pas d'élément minimal qui corresponde à une contradiction ("nothing" en Omega [AGHA.86], c'est-à-dire une classe qui ne décrit aucun objet.

La majorité des outils proposent aujourd'hui leur héritage multiple. Cette notion pose cependant des problèmes et les stratégies utilisées diffèrent fortement selon les langages comme nous le verrons au paragraphe III.3.3.

III.3.2.- Héritage et caractérisation

Nous allons introduire le rôle de l'héritage comme mécanisme d'inférence ou de partage de caractéristiques pour présenter ensuite les différentes stratégies utilisées (III.3.3.). Puis nous essaierons de préciser les différentes techniques de composition de méthodes qui utilisent l'héritage (III.3.4).

III.3.2.1.- Affinement de caractérisation

L'affinement d'une classe par une sous-classe se traduit par celui des caractérisations d'objet qu'elles définissent. On distingue couramment deux types d'affinement [STEFIK-BOBROW.85] [RECHENMANN. 86] que nous nommerons :

l'affinement par ajout : par lequel une sous-classe ajoute des caractéristiques à celles définies par une sur-classe.

l'affinement par substitution par lequel une sous-classe substitue une définition de caractéristique d'une sur-classe par une nouvelle définition. Cet affinement est spécifié par masquage; i.e. en réutilisant le même nom de caractéristique définie dans la sur-classe et substitué dans la sous-classe.

L'héritage est alors le mécanisme d'inférence des caractéristiques qui traduit l'affinement. Concernant l'affinement par ajout, il garantit qu'un objet dispose des caractéristiques définies par sa classe d'instanciation plus toutes celles définies par les sur-classes de celles-ci, jusqu'à OBJECT. Concernant l'affinement par substitution, il garantit ce que nous appellerons la règle du plus affiné : pour une caractéristique donnée, c'est toujours sa définition la plus affinée qui est prise en compte. Celle-ci est la première rencontrée selon un parcours des classes dans le sens de la généralisation, i.e. de la classe d'instanciation de l'objet à la classe la plus générale, OBJECT.

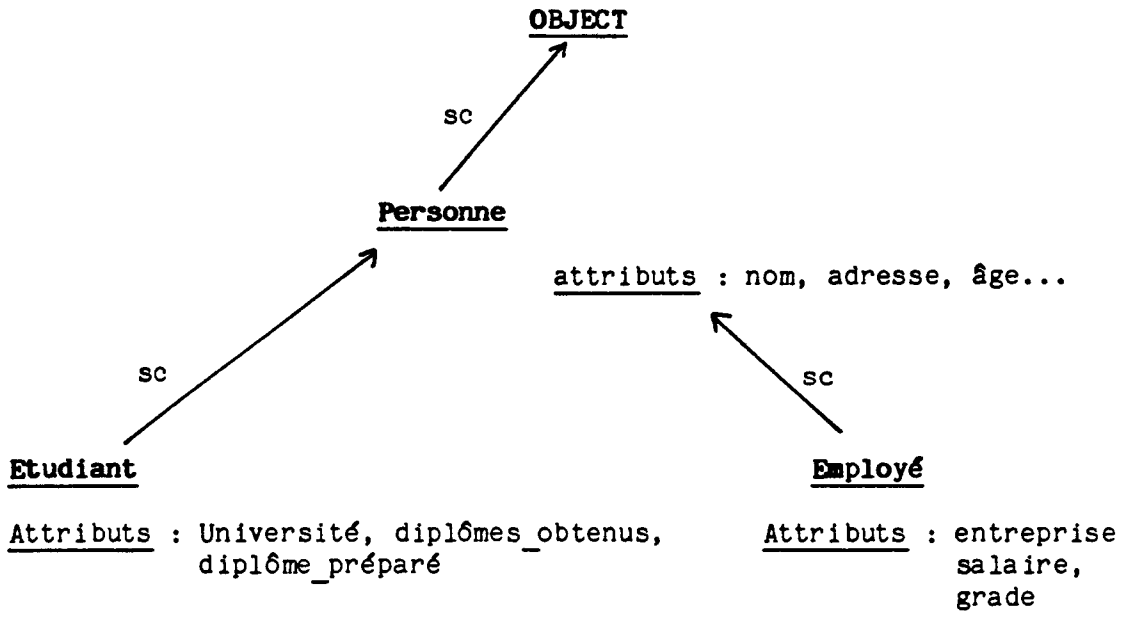
Dans le cas où les classes forment un arbre (héritage simple), ce parcours est unique et ne pose pas de problème particulier. Dans le cas où les classes forment un treillis (héritage multiple), ce parcours n'est pas unique et les stratégies implémentées diffèrent, ce qui sera vu au paragraphe III.3.3.

Rappelons que les caractéristiques sont de deux types, attributs (variables d'instance, slots, champs,...) et méthodes (action, procédures, opérations, fonctions,...) associées à leur sélecteur.

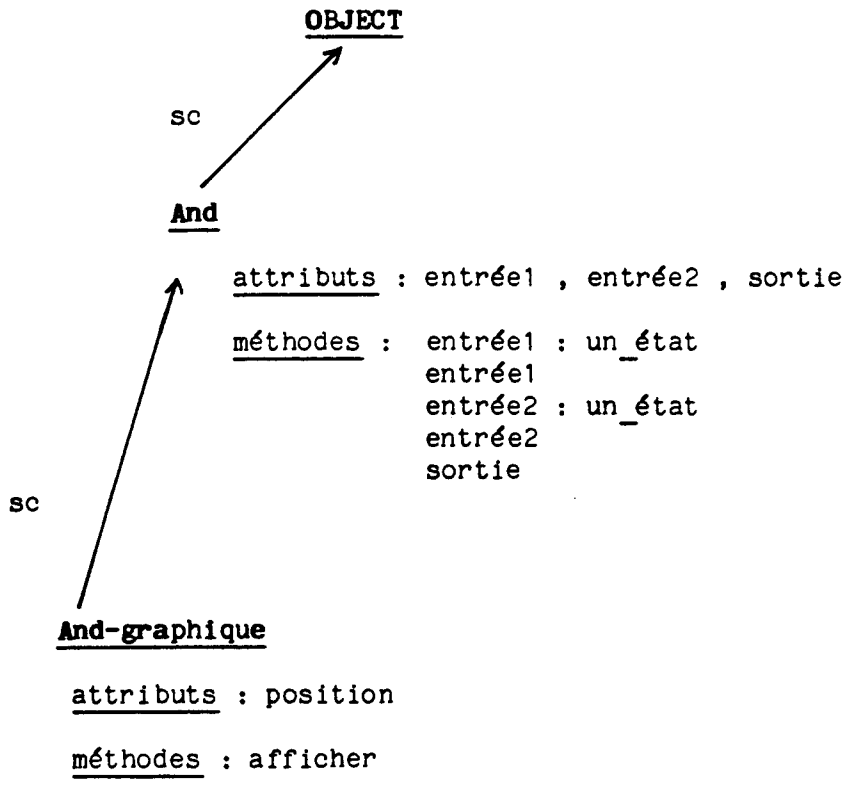
L'affinement par ajout s'applique à ces deux types de caractéristique. L'affinement par substitution n'a de sens que si la caractéristique dispose d'une définition associée. Il s'applique donc aux couples (sélecteur méthode), la méthode étant considérée comme la définition associée à un sélecteur, c'est-à-dire l'implémentation de la fonctionnalité spécifiée par celui-ci. Par contre, concernant les attributs, les outils diffèrent selon qu'ils leur associent ou non des informations. Pour les outils comme SMALLTALK, les attributs (variables d'instance) sont spécifiés au niveau d'une classe par leur nom et aucune information ne leur est associée. L'affinement par substitution n'a donc pas de sens pour de tels attributs. Par contre, certains outils attachent des informations aux attributs. Celles-ci peuvent être plus ou moins complexes, et vont d'une simple valeur par défaut comme en LOOPS, un type dans les langages typés ou des facettes dans les langages de frames, comme nous le verrons au paragraphe III.4.4. L'affinement par substitution a alors un sens pour ces outils, par exemple pour redéfinir une nouvelle valeur par défaut, le type, ou de nouvelles facettes (par ajout ou par substitution ! (*)). Notre modèle de base étant proche de SMALLTALK nous ne considérerons que l'affinement par substitution de méthode sachant que les stratégies d'héritage étudiées par la suite valent pour tout type de caractéristique. Montrons ces deux types d'affinement sur les exemples simples suivants :

(*) Nous verrons que dans les langages de frames un attribut peut être vu lui-même comme un objet défini par une classe. Les facettes sont alors les caractéristiques des attributs et récursivement. L'affinement entre classes s'applique donc en particulier pour les classes d'attributs et suit idéalement les même lois.

- Affinement par ajout :

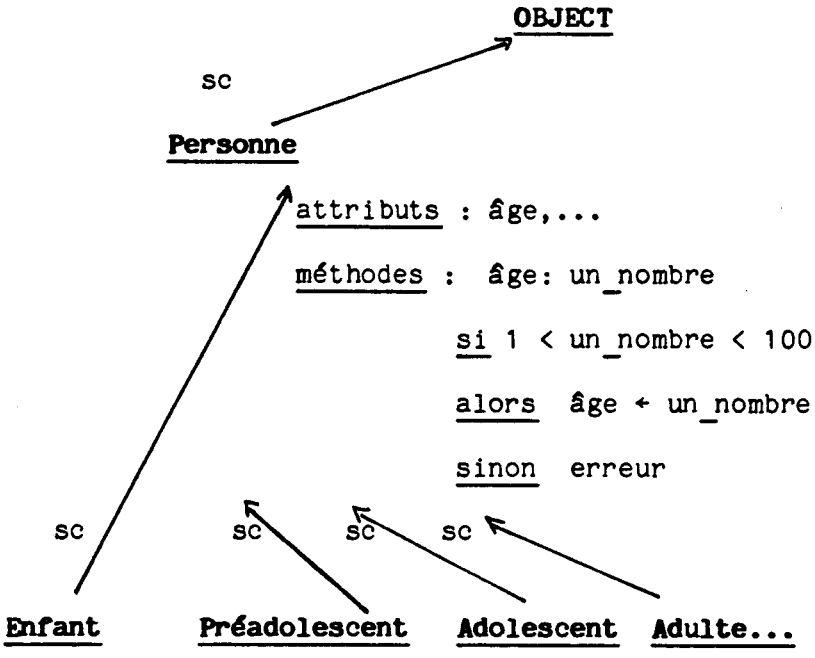


chacune des classes traduit un affinement par ajout de caractéristiques, en l'occurrence d'attributs.



La classe And-graphique ajoute les caractéristiques graphiques à celles définies par la classe And.

- Affinement par substitution



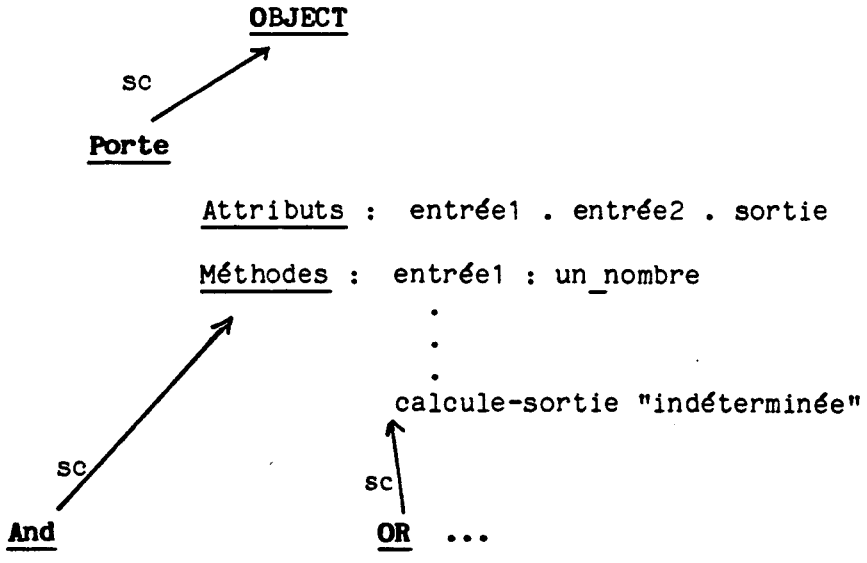
méthodes âge: un_nombre

si 1 < un_nombre < 10

alors âge ← un_nombre

sinon erreur.

La méthode âge: un_nombre de Personne est substituée dans chacune des sous-classes.



méthodes calcule-sortie
sortie ← entrée1 et entrée2

La méthode calcule-sortie indéterminée au niveau de Porte est substituée dans chacune de ses sous-classes pour lui attacher sa définition appropriée.

III.3.2.2.- Partage de caractérisation

D'un point de vue plus pratique, l'héritage peut être vu comme un simple mécanisme de partage d'informations communes selon un certain processus de factorisation des caractéristiques :

> "L'héritage est le mécanisme de partage de la connaissance dans les langages orientés objet".
>
> [DUCOURNEAU-HABIB.87]

> "One use of the class hierarchy of SMALLTALK is to factor and share code".
>
> [SANDBERG.86]

Il permet ainsi d'éviter une redondance d'information entre classes et d'optimiser l'écriture des "programmes" orientés objet :

> "Mechanisms like this are important because they make it possible to declare that certain specifications are shared by multiple parts of a program. Inheritance helps to keep programs shorter and more tightly organized".
>
> [STEFIK-BOBROW.85]

Reprenons notre exemple des portes logiques. Les portes and et or étant caractérisées comme suit :

And

attributs : entrée1 , entrée2 , sortie

méthodes :
```
entrée1: un_état
entrée1
entrée2: un_état
entrée2
sortie
calcule-sortie
        sortie ← entrée1 et entrée2
```

OR

attributs : entrée1 , entrée2 , sortie

méthodes :
```
entrée1: un_état
  .
  .
  .
calcule-sortie
        sortie ← entrée1 ou entrée2
```

L'héritage permet de factoriser les caractéristiques communes à ces deux classes dans une sur-classe Porte selon le schéma du III.3.2.1.. On peut donc définir l'héritage comme le mécanisme qui restitue une définition complète de chaque classe à partir d'une définition factorisée. Il peut être vu également comme un mécanisme de "copie virtuelle" [FAHLMAN.79] des caractéristiques d'une sur-classe dans chacune de ses sous-classes moyennant les substitutions. Cette copie virtuelle se fait selon un parcours des classes qui dépend des stratégies employées et que nous allons présenter dans le paragraphe suivant.

III.3.3.- Stratégies d'héritage

Quelle que soit son interprétation en tant que mécanisme d'inférence, de partage ou de copie virtuelle, l'héritage revient à un algorithme de parcours du graphe des classes appelé généralement lookup.

L'héritage peut être statique ou dynamique selon les outils. En général, dans les langages interprétés de type SMALLTALK ou les extensions de LISP, l'héritage des attributs est statique et celui des méthodes dynamique. L'héritage statique des attributs se fait par l'application du lookup une seule fois à l'instanciation. Il ramène tous les attributs qui caractérisent l'objet créé depuis sa classe d'instanciation jusque OBJECT. L'instanciation construit alors une structure d'objet dont les champs correspondent aux attributs auxquels seront associées les valeurs d'instance :

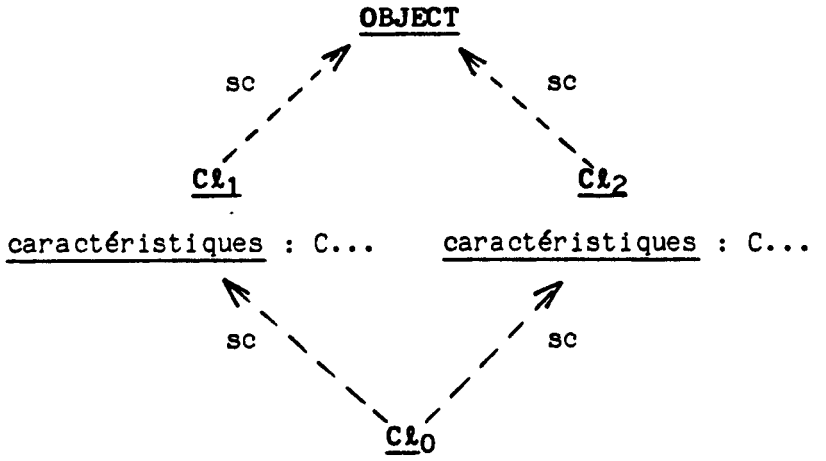
```
objet = <nom_d'attribut_1  valeur_1

        nom_d'attribut_2  valeur_2
           .
           .
           .

        nom_d'attribut  valeur_n> (*)
```

(*) Beaucoup d'extensions de LISP utilisent la structure de A-list ((nom d'attribut_1 . valeur_1)...(nom d'attribut_n . valeur_n)) ou de plist en associant à chaque objet un symbole. Dans cette dernière représentation, les noms de propriétés correspondent aux noms d'attributs et les primitives de lecture et écriture utilisent simplement les fonctions getprop et putprop d'accès à la plist.

L'héritage dynamique des méthodes se fait à l'exécution. Chaque fois qu'une méthode est invoquée (par envoi de message) le lookup recherche la définition associée pour l'objet concerné (*). Cette différence est dûe principalement au fait que les instances ne diffèrent que par les valeurs (valeurs d'instances) associées aux attributs dont ils ne partagent que les noms, contrairement aux méthodes complétement partagées. Il leur faut donc un espace particulier à chacune pour mémoriser leurs couples (attributs, valeurs) tandis qu'elles peuvent partager leurs méthodes physiquement en mémoire dans la structure de classe sous l'attribut "méthodes".

L'héritage peut être simple ou multiple respectivement si le graphe des classes est un arbre ou un treillis. L'héritage simple ne pose pas de problème particulier. Le parcours est linéaire sur la branche [classe d'instanciation...OBJECT] sur laquelle les classes sont totalement ordonnées de la plus affinée, la classe d'instanciation, à la plus générale, OBJECT. Pour une caractéristique d'un nom donné, la définition associée à la première occurrence de ce nom sur le parcours est prise en compte, c'est la plus affinée.

L'héritage multiple pose des problèmes essentiellement dûs aux conflits de caractéristiques de même nom définies par plusieurs sur-classes d'une même classe.



Quelle est la caractéristique C prise en compte pour les représentants de $C\ell_0$?

La réponse dépend en fait des stratégies employées.

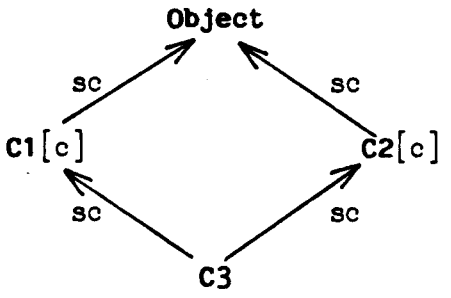
(*) (send unObjet sél argt1 ... argtn)

<=> (funcall (méthode-associée sél unObjet) unObjet argt1..argtn)

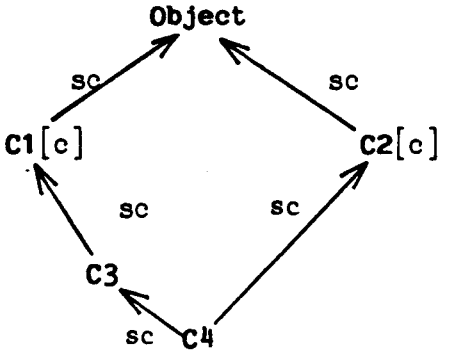
où méthode-associée consiste en l'application du lookup.

Les classes étant organisées en graphe, il n'en existe pas de parcours trivial comme dans le cas de l'héritage simple, l'ordre est partiel. L'héritage multiple fait l'objet de nombreux travaux comme sa récente formalisation dans le cadre de la théorie des graphes [DUCOURNEAU-HABIB.87] et les stratégies employées diffèrent selon les outils. On peut distinguer deux types d'approches :

* l'approche linéaire qui est la plus courante, est basée sur les parcours classiques en profondeur d'abord pour Flavors ou Loops par exemple et en largeur d'abord pour Mering. Ces stratégies utilisent l'ordre de la liste des sur-classes directes d'une classe, valeur de son attribut sorte_de, pour tranformer l'ordre partiel sur ces sur-classes en un ordre total. Elles linéarisent donc le graphe d'héritage d'une classe en une chaîne et ramènent ainsi l'héritage multiple à l'héritage simple de telle sorte qu'une seule caractéristique conflictuelle est héritée. Montrons ceci sur le schéma suivant où C1, C2 , C3 sont des classes et c une caractéristique définie par C1 et C2.

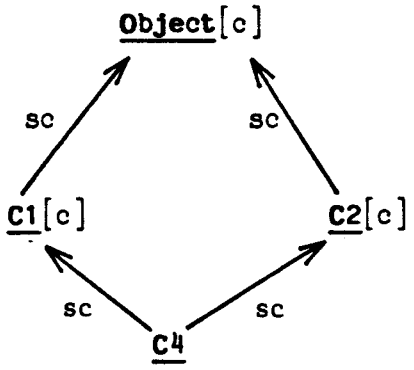


Il y a donc un conflit d'héritage de la caractéristique c pour la classe C3. L'ordre de la liste de ces sur-classes directes est (C1, C2), c'est donc c définie par C1 qui sera héritée par C3. Si la liste avait été (C2,C1) , c définie dans C2 aurait été prise en compte. Ainsi, lors de la définition d'une classe, la déclaration de l'ordre de ses sur-classes a une grande importance. Même si le cas précédent ne fait pas de différence entre profondeur d'abord et largeur d'abord, il est évident que ces deux stratégies ne sont pas équivalentes comme on peut le constater sur le schéma suivant :



où c héritée par C4 sera retrouvée dans C1 en profondeur d'abord et dans C2 en largeur d'abord. Ceci montre l'importance de la profondeur du graphe sur l'héritage.

Enfin, le problème de l'affinement par substitution se pose sur le schéma suivant



Considérant que la règle du plus affiné doit être respectée avant tout (* 1) , la caractéristique la plus affinée pour C4 est celle définie par C2. Un simple parcours en profondeur ne suffit donc pas puisqu'il retrouverait c au niveau de OBJECT. Il doit être complété comme suit : parcours en profondeur d'abord en vérifiant avant tout la règle du plus affiné. Cette solution est adoptée en particulier par Flavors. Le graphe précédent peut se traduire par les relations d'ordre partiel suivantes :

$$C4 < C1 < Object$$

$$C4 < C2 < Object.$$

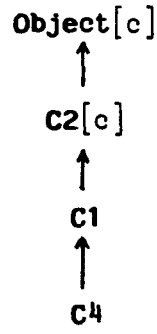
complétées par la relation d'ordre sur la liste des surclasses de C4 (*2)

$$C1 < C2.$$

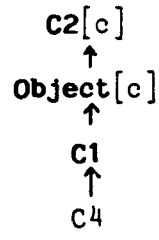
(*1) Ce qui peut mener à débat. En effet, l'interprétation même d'un tel schéma n'est pas simple. S'agit-il pour C4 d'hériter de c définie dans OBJECT par C1 et ainsi de passer outre la redéfinition de c dans C2 ? S'agit-il au contraire de respecter avant tout l'affinement par substitution, c'est-à-dire considérer que c est redéfinie dans C2 et donc que cette définition est la plus affinée ? Ou s'agit-il encore d'hériter des 2 définitions ? La réponse n'est pas évidente. Les stratégies respectent généralement la substitution avant tout, moyennant les mécanismes spécifiques du SUPER ou plus généralement d'héritage explicite qui seront vus aux III.3.4.2. et III.3.4.3.

(*2) Cette relation supplémentaire est traduite dans [DUCOURNEAU-HABIB.87] par ce qu'ils appellent un arc de multiplicité reliant C1 à C2.

et conduit donc à l'ordre total : C4 < C1 < C2 < Object qui transforme le graphe précédent en la chaîne d'héritage simple :

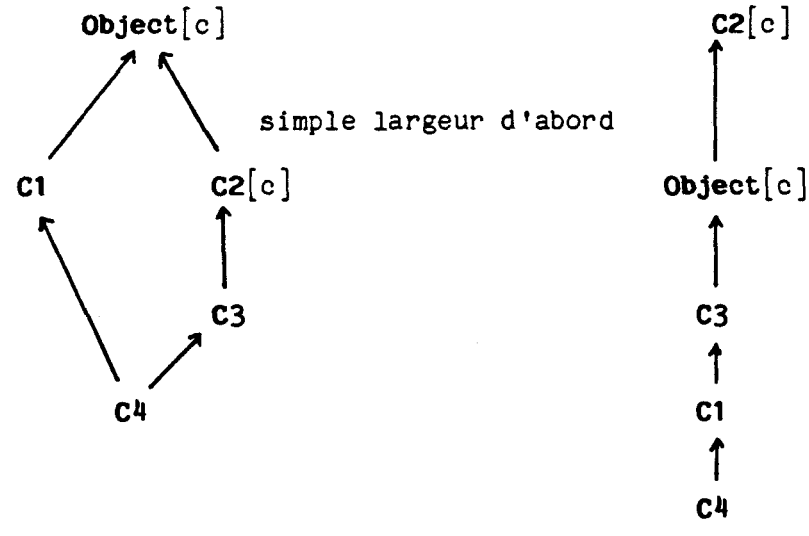


contrairement au simple parcours en profondeur d'abord qui aurait mené à la chaîne :



et à l'incohérence Object < C2. La caractéristique de C2 est alors héritée.

Le parcours en largeur d'abord semble résoudre ce problème puisque la chaîne résultante est justement la première donnée ci-dessus. Malheureusement ce n'est pas le cas sur le schéma suivant :



qui montre une fois de plus l'influence de la profondeur du graphe sur ce type de stratégie. La solution à ce problème est évidemment similaire : parcours en largeur d'abord en véfifiant avant tout la règle

du plus affiné. Elle conduit de façon similaire à la transformation du graphe précédent en la chaîne d'héritage simple :

C4 → C1 → C3 → C2 → Object.

Nous terminerons là l'exposé de ces solutions linéaires. Remarquons cependant qu'une analyse plus poussée montrerait d'autres problèmes encore plus pointus, ce qui prouve l'intérêt d'une étude théorique de l'héritage telle qu'elle est réalisée dans [DUCOURNEAU-HABIB.87]. En conclusion, rappelons et insistons sur plusieurs points qui caractérisent cette aproche. L'ordre de la liste des sur-classes directes d'une classe et la profondeur du graphe sont d'une grande importance car ils déterminent le parcours réalisé par l'héritage. Le graphe d'héritage d'une classe est linéarisé en une chaîne par transformation de l'ordre partiel des sur-classes en un ordre total. L'héritage multiple est ainsi ramené à un héritage simple.Ainsi, et c'est la propriété essentielle, une seule définition d'une caractéristique conflictuelle est héritée.

* l'approche graphique opère directement sur le graphe sans transformation. Le principe est qu'une classe hérite de toutes les caractéristiques définies dans ses super-classes indépendamment des conflits de noms. Il est évidemment à l'opposé de l'approche linéaire où les conflits sont "résolus" en ne retenant qu'une seule caractéristique conflictuelle, celle-ci dépendant de l'ordre de parcours. Cette stratégie a été étudiée pour étendre SMALLTALK à l'héritage multiple [BORNING-INGALLS.82] puis reprise par plusieurs langages comme COMMON OBJECTS et TREILLIS/OWL [SNYDER.86ab] [SHAFFERT and al.86]. Dans le cas de conflit ces outils offrent un mécanisme pour nommer explicitement la sur-classe à partir de laquelle la caractéristique doit être recherchée, c'est-à-dire un mécanisme d'accès direct en tout point du graphe. En Extended SMALLTALK par exemple (nom couramment donné à la version de SMALLTALK étendue à l'héritage multiple et présentée dans [BORNING-INGALLS.82]) la notion de sélecteur a été étendue à celle de sélecteur composé (compound selector) par préfixage par le nom d'une sur-classe : classe.sélecteur (* 1). La méthode associée au sélecteur est alors recherchée à partir de la sur-classe spécifiée en préfixe. Si cette méthode est unique, elle est appliquée. Sinon le problème se pose de façon similaire, i.e. plusieurs sur-classes de la classe préfixe initiale définissent la même méthode et l'ambiguité doit être levée explicitement par sélection d'une de ces surclasses (*2). Il est assez clair que dans cette approche toute définition de caracté-

---

(*1) Le mécanisme d'envoi de message n'est pas modifié. La première fois qu'un message spécifiant un sélecteur composé est envoyé, celui-ci ne faisant pas partie du protocole de l'objet, il y a erreur "message not understood" qui est trappée pour analyser ce sélecteur et extraire le préfixe qui spécifie la classe à partir de laquelle le lookup doit commencer sa recherche. La méthode trouvée est alors compilée sous le sélecteur classe.sélecteur et le message est envoyé à nouveau résultant en l'application de cette méthode.

(*2) Il y a erreur à l'exécution en SMALLTALK et à la compilation en TREILLIS/OWL.

ristique est potentiellement accessible par désignation directe et explicite de la classe sélectionnnée, c'est pourquoi nous l'appellerons par la suite héritage explicite.

En dehors de son intérêt pour lever les conflits, ce type de stratégie permet, d'un point de vue pratique, d'augmenter l'efficacité de l'héritage, puisqu'un point de départ du parcours lui est spécifié explicitement. Ceci peut être déterminant, surtout si l'héritage est dynamique, notamment pour les interpréteurs.

Bien que ce type de stratégie paraisse satisfaisant et résolve d'une manière élégante les conflits, il pose des problèmes non négligeables qui seront évoqués au paragraphe III.3.4.3. Nous retiendrons ici son principe de base qui est de permettre à une classe d'hériter de toutes les caractéristiques définies dans ses sur-classes indépendamment des conflits.

En conclusion, insistons sur le fait que l'héritage multiple est un sujet en pleine évolution et qu'il n'y a pas aujourd'hui de stratégie standard, tout n'est pas possible de manière équivalente avec n'importe quel outil.

III.3.4.- Techniques de combinaisons de méthodes

Chaque sous-classe définit sa propre caractérisation d'objet en affinant par ajout ou par substitution celles spécifiées par ses surclasses. En particulier, une classe définit des méthodes qui peuvent faire appel à d'autres méthodes définies par elle-même ou héritées : c'est la combinaison de méthodes.

III.3.4.1.- SELF

Nous avons vu, au paragraphe II.3.1. que le seul moyen d'appliquer une méthode dans notre modèle de base est l'envoi de message (postulat P2). Un objet, pour appliquer ses propres méthodes, passe donc par l'envoi de message à lui-même, référencé par la pseudo-variable SELF dans le corps d'une méthode. Les self messages constituent donc a priori le seul moyen de combiner les méthodes. Nous avons vu un des exemples de combinaison par self message au paragraphe II.3.3. Prenons ici l'exemple des portes not définies par la classe Not suivante en SMALLTALK (*)

(*) Nil est interprétée comme l'état indéterminé d'où la méthode d'évaluation de la sortie, calcule-sortie. En particulier, l'instanciation SMALLTALK initialise les variables d'instance à Nil. Initialement, les états d'un not seront donc indéterminés.

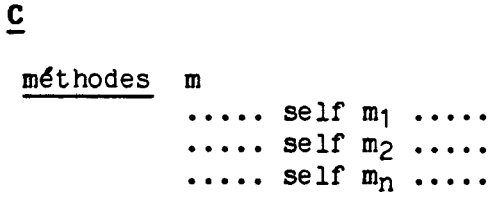
```
Object subclass: #Not
        instanceVariableNames: 'entree sortie '

Not comment:
'Not caracterise les portes not.'

Not methodsFor: 'evaluating'

calculeSortie
        " evalue et ecrit la sortie"

        entree isNil
                ifTrue: [sortie <- nil]
                ifFalse: [sortie <- entree not]


Not methodsFor: 'reading'

entree
        "retourne l'etat d'entree"

        ^entree

sortie
        "retourne l'etat de sortie"

        ^sortie


Not methodsFor: 'writing'

entree: etat
        " affecte l'entree et propage l'evaluation de la sortie par self
          calculeSortie si l'etat est valide sinon une erreur par self error,
          heritee de Object"

        ((etat isKindOf: Boolean) or: [etat isNil])
                ifTrue:
                        [entree <- etat.
                        self calculeSortie]
                ifFalse:
                        [self error: ' etat booleen ou indetermine (nil)']
```

Les instances de Not, à chaque modification de leur état d'entrée par l'envoi du message "entrée: un_Etat" évalueront automatiquement leur sortie par "self calcule_sortie" si un_Etat est valide (Un_Etat = true ou false ou nil). La méthode "entrée:" de Not est une combinaison par self messages de la méthode "calcule-sortie" définie par cette même classe et de "error:" héritée par celle-ci de Object :

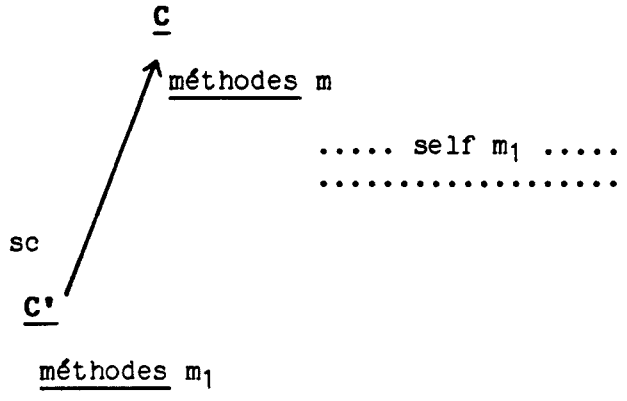
> SELF et l'héritage permettent à une classe de définir des méthodes par combinaisons d'autres méthodes qu'elle définit ou hérite.

Une conséquence de cette propriété de SELF est dûe au fait suivant.

Soit une méthode m , nommée par la suite méthode appelante, combinaison d'autres méthodes m_1 , m_2 ,..., m_n , nommées par la suite méthodes appelées. La méthode m est définie par la classe C et les méthodes m_i sont définies ou héritées par C selon le principe précédent.



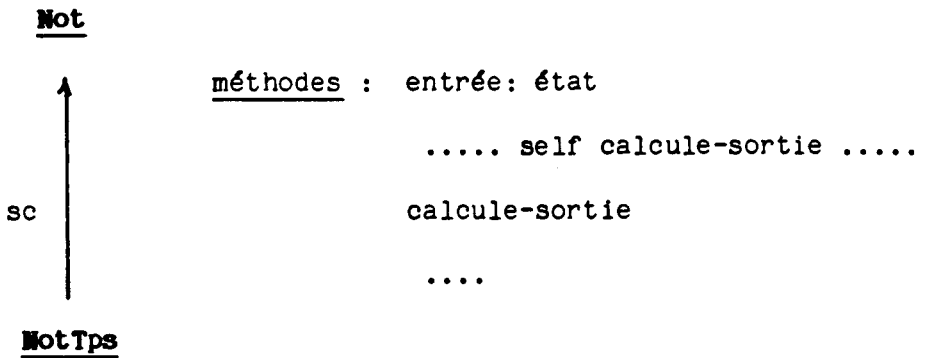
Toute méthode m_i étant susceptible d'être affinée par substitution dans les sous-classes de C , il en résulte un affinement implicite de la méthode m appelante sans que celle-ci soit effectivement redéfinie :



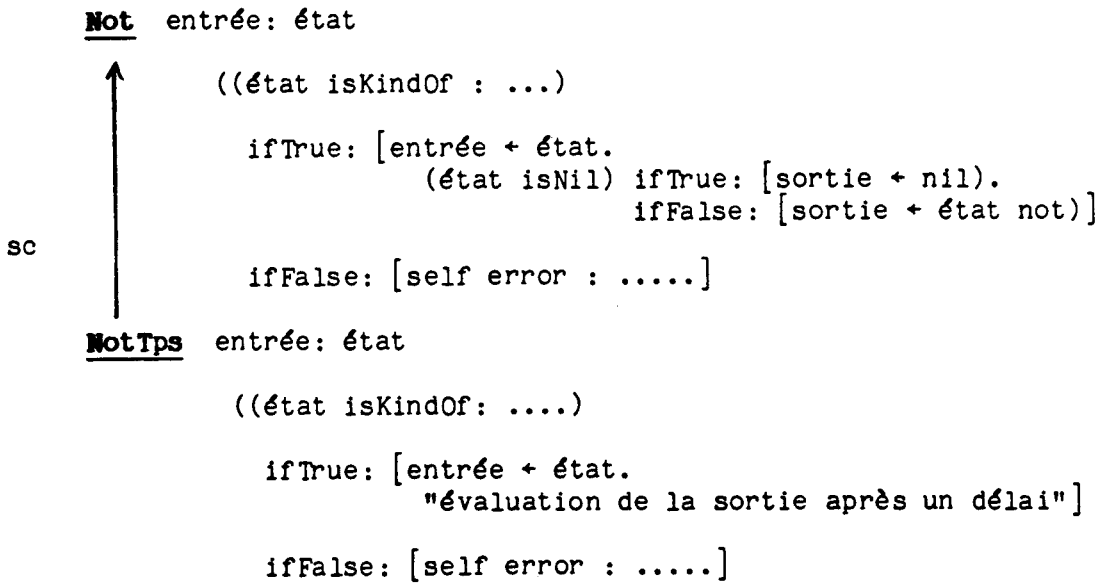
"affinée par substitution"

L'héritage garantissant la règle du plus affiné, l'application de la méthode m pour une instance de C' résultera en l'application de la méthode m_1 , telle que définie dans C'.

Par exemple, soit la sous-classe NotTps de Not qui tient compte des problèmes de délais temporels (Tps pour temps) dans l'évaluation de la sortie après l'écriture de l'entrée. Supposons que l'affinement de la classe Not en NotTps se fasse par l'affinement par substitution de la méthode calcule-sortie :



Ainsi l'affinement de la méthode calcule-sortie pour NotTps a pour conséquence l'affinement implicite de la méthode entrée : état pour cette même classe. Rétrospectivement, l'affinement de cette dernière aurait dû être explicitée si elle n'avait pas fait appel à calcule-sortie par self message :



SELF et l'héritage permettent un affinement implicite de la méthode appelante par l'affinement de ses méthodes appelées.

Plus généralement une méthode peut être volontairement décomposée en d'autres méthodes par self messages, laissant ainsi la possibilité de les affiner ultérieurement dans les sous-classes. Par exemple, la méthode calcule-sortie de NotTps peut être décomposée en :

- méthode de temporisation appelée délai
- évaluation effective de la sortie

NotTps

méthodes calcule-sortie
```
self délai.
(entrée isNil) ifTrue: [sortie ← Nil]
               ifFalse: [sortie ← entrée not]
```

délai
```
"méthode de temporisation"
```

La méthode "délai" pouvant être redéfinie par des sous-classes de NotTps, par exemple en fonction de la technologie, calcule-sortie sera implicitement affinée. Prenons un autre exemple simple, la classe Personne définit la méthode âge: un Nombre, avec un Nombre qui doit être compris entre 1 et 100. Le fait de décomposer cette méthode en :

- méthode de test de la validité de unNombre appelée valideAge
- écriture effective de l'âge.

laisse la possibilité d'affiner implicitement cette méthode par affinement de la méthode valideAge dans les sous-classes telles que Enfant, Préadolescent, Adolescent, Adulte :

```
Object subclass: #Personne
        instanceVariableNames: 'age '

Personne methodsFor: 'accessing'

age: unNombre
        " affectation de l'age, si unNombre verifie les contraintes
        imposees par la methode valideAge: . Cette methode est
        redefinie dans les sousclasses"

        (self valideAge: unNombre)
                ifTrue: [age <- unNombre]
                ifFalse: [self error: ' age non valide ']

valideAge: unNombre
        " au moins entre 1 et 100 "

        ^ unNombre between: 1 and: 100

Personne subclass: #Enfant
        instanceVariableNames: ''

Enfant methodsFor: 'controling '

valideAge: unNombre
        ^unNombre between: 1 and: 10
```

et de façon similaire pour les autres sous-classes.

> SELF permet la décomposition d'une méthode m en d'autres méthodes dont l'affinement dans les sous-classes provoquera l'affinement implicite de m par héritage.

Enfin, SELF permet la récursivité. Par exemple la factorielle (!) est définie comme suit en SMALLTALK dans la classe Integer :

```
factorial

(self = 0) ifTrue: [^ 1]

          ifFalse: [(self < 0) ifFalse: [^ self * (self -1)factorial]

                               ifTrue: [self error: 'invalid
                                                    factorial']
```

III.3.4.2. SUPER

Bien que self soit à la base de la combinaison de méthodes, un problème se pose quand il y a à la fois substitution et combinaison.

On veut, par exemple, définir la sous-classe NotGraphique de Not, qui ajoute les caractéristiques graphiques suivantes de définition évidente :

V.I. : x,y

méthodes : afficher
"méthodes d'affichage d'un not par dessin du symbole à la position (x,y) et visualisation des états d'entrée/sortie"
.....

Supposons que l'on veuille qu'à chaque mise à jour de son état un notgraphique se dessine automatiquement. La mise à jour de son état se faisant sur écriture de son état d'entrée, c'est-à-dire par la méthode entrée: état, il s'agit d'affiner cette méthode par substitution pour décrire ce comportement.

Une approche légitime serait de vouloir exprimer la combinaison :

nouvelle méthode entrée: état

= ancienne méthode entrée: état

puis afficher.

Cependant le masquage (substitution) de l'ancienne méthode par la nouvelle interdit son accès et elle ne peut être appliquée par simple self message :

NotGraphique :

méthodes : entrée: état

self entrée: état "en pensant à l'ancienne méthode"
self afficher

conduirait évidemment à un bouclage de l'héritage.

La plupart des langages offrent donc la possibilité de passer outre le masquage par le mécanisme des super-messages introduit par SMALLTALK et repris par d'autres langages comme MERING (super:) LOOPS (←Super) COMMONLOOPS (RunSuper), OBJVLISP (runsupers)...

En SMALLTALK la pseudo variable super référence comme self l'objet lui-même, i.e. l'objet en cours d'activation qui applique la méthode concernée, mais force l'héritage à partir de la sur-classe de la classe définissant cette méthode (*1), c'est donc une technique d'échappement à la substitution.

Notre exemple s'écrit alors :

NotGraphique

```
entrée: état

    super entrée: état.
    self afficher.
```

La propriété essentielle du SUPER est de permettre l'affinement incrémental des méthodes dans les sous-classes, chacune d'elles apportant des fonctionnalités supplémentaires :

> "The incremental specialization of methods - that is - the ability to make incremental addition to inherited methods".
> [STEFIK-BOBROW.85]

Nous allons essayer de montrer les aspects de ce mécanisme.

Insistons tout d'abord sur le statut de super en SMALLTALK en le comparant à self. Un point important est que self et super référencent le même objet puisque l'évaluation de la méthode doit se faire dans le contexte de self, c'est-à-dire du receveur du message. Un super message cache donc fondamentalement un envoi de message à self (*2) et non à un autre objet, comme on pourrait le penser naïvement. C'est un "sucre syntaxique" qui a l'apparence d'un simple envoi de message (*3). Super, tout comme self, est réservé à l'implémentation et ne peut apparaître que dans le corps d'une méthode.

(*1) et non de la super-classe de la classe d'instanciation de l'objet, ce qui serait équivalent sur notre exemple mais ne résoudrait évidemment pas le problème si l'objet est instance d'une sous-classe de cette classe.

(*2) Ceci est plus apparent syntaxiquement en Loops par exemple. Dans ce langage la primitive d'envoi de message est (← objet sélection arguments), un self message s'écrit donc (← self sélecteur argts) contrairement à un "super message" qui fait appel à la primitive (←Super self sélecteur arguments)

(*3) Remarquons le parallèle avec la délégation où "super" référencerait effectivement un not en tant que prototype d'un not Graphique. En mettant en parallèle les notions de prototype et de classe, on pourrait penser que super référence ici la super-classe Not en tant qu'objet. Ceci est évidemment faux, ne serait-ce que parce que l'objet Not ne peut comprendre le message entrée: état; ce n'est pas un not, il n'est pas représentant de lui-même !

Il ne permet donc un accès à l'ancienne méthode que de manière interne à un objet. En effet, le protocole externe de l'objet est inchangé et vue de l'extérieur l'ancienne méthode est effectivement masquée. Tout envoi de message "unNotGraphique entrée: un_état" ou un NotGraphique est instance de NotGraphique provoquera l'application de la méthode la plus affinée, c'est-à-dire celle définie dans cette dernière classe. Seule l'évaluation "interne" de cette méthode fera appel à l'ancienne méthode et "super entrée: état" est équivalent à "self entrée: état", vue de l'intérieur, comme si l'ancienne méthode n'avait pas été masquée (*).

L'importance du masquage réside dans le fait que l'ancienne méthode ne doit plus être applicable de l'extérieur de l'objet. En effet, rien n'empêcherait d'utiliser un nom de méthode différent si l'ancienne devait toujours faire partie du protocole de l'objet. Sur notre exemple, il suffirait alors de renommer la méthode entrée: état définie dans NotGraphique en gentrée: état (g comme graphique). la solution serait alors :

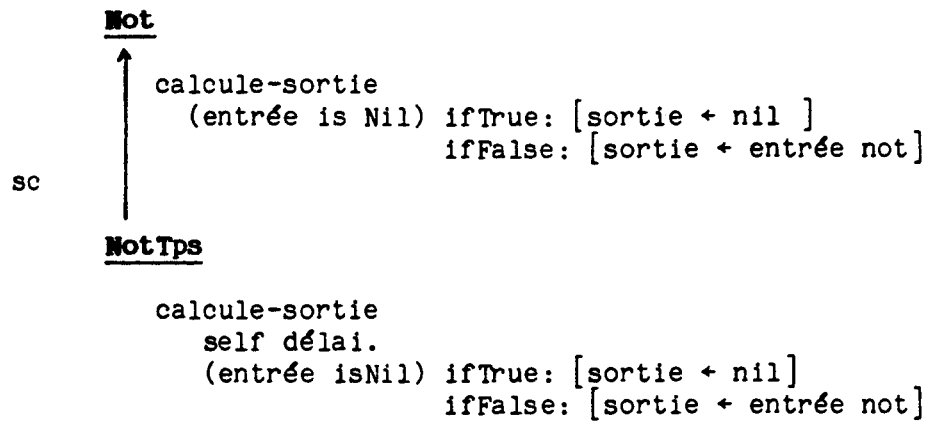
Not-graphique : gentrée: état

self entrée: état
self afficher

Remarquons le remplacement de "super entrée: état" en "self entrée: état" qui montre bien que self et super référencent le même objet (et toute la subtilité de ce mécanisme). Les méthodes"entrée: état" non graphique et "gentrée : état" feront alors toutes deux partie du protocole d'un NotGraphique et pourront faire l'objet d'envoi de message "UnNotGraphique gentrée: état" propagera en plus sa visualitation automatique. Le choix de masquer ou non, et donc, dans le cas présent, d'utiliser SUPER ou non, dépend évidemment de l'application, c'est-à-dire des spécifications de l'objet.

Par exemple, reprenons nos classes Not et NotTps du paragraphe précédent et, en particulier, leur définition respective de la méthode calcule-sortie :

(*) C'est là toute l'ambiguité du Super : on veut masquer tout en ne masquant pas ou masquer pour l'extérieur, c'est-à-dire pour les utilisateurs externes à l'objet, mais ne pas masquer pour l'objet lui-même. Dans ce cas, ce problème ne se pose pas pour les langages qui font une séparation plus nette entre privé et public, ou interne et externe. Comme Treillis/Owl, C++ ou CommonObjects. Dans ces langages, il est possible de déclarer des opérations (méthodes) publiques, c'est-à-dire faisant partie du protocole externe, et privées, c'est-à-dire réservées à l'implémentation interne. C'est le cas également pour ROME comme on pourra le voir par la suite, qui fait la séparation entre les notions de méthode et de sélecteur. Toute méthode est privée, c'est-à-dire applicable par l'objet, seules celles associées à des sélecteurs sont activables de l'extérieur de l'objet.



grâce au Super, cette dernière version de calcule-sortie peut s'écrire :

NotTps

```
calcule-sortie
  self délai.
  super calcule-sortie
```

Super permet donc de montrer clairement l'affinement réalisé par la sous-classe NotTps qui tient compte des problèmes de délai. Son utilisation peut être essentielle ici pour interdire l'accès à l'ancienne méthode calcule-sortie, dans une version plus élaborée de la simulation au niveau temporel.

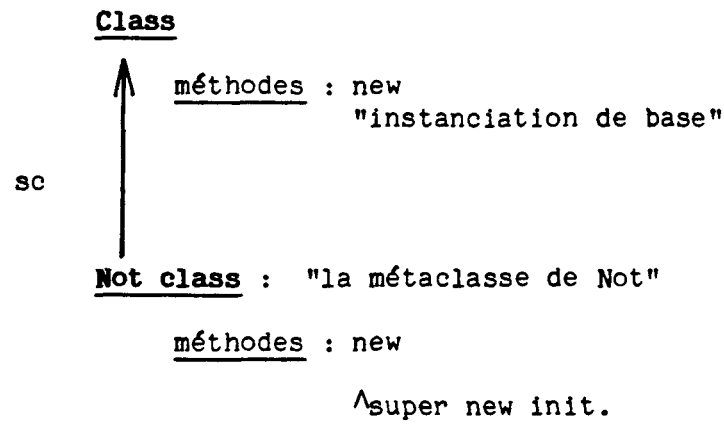
Super est souvent utilisé pour définir de nouvelles méthodes d'instanciation par masquage du new telles que celles présentées en II.3.3. Reprenons l'exemple d'une classe qui initialise ses instances, soit Not dont les variables d'entrée et sortie sont respectivement initialisées à false et true :

Not

```
surclasse : Object
variables d'instance : entrée, sortie
méthodes d'instances : entrée: état
                          .
                          .
                          .
                       init
                          entrée ← false
                          sortie ← true
méthodes de classe :  new
                          ^super new init
```

Rappelons que les méthodes de classe sont les méthodes d'instances de la classe, en tant qu'objet et sont donc définies dans la classe de celle-ci, c'est-à-dire sa métaclasse. En particulier, la méthode new

de base est définie dans la sur(méta)classe de toutes les métaclasses, c'est-à-dire Class (*). Le schéma d'héritage simplifié suivant :



montre que Super permet d'hériter de la méthode new de base tout en la masquant. Le masquage est important ici si toute instance de Not doit être initialisée et donc que l'ancienne méthode new ne puisse plus être appliquée par envoi de message. L'utilisation d'un autre nom de méthode qui pourrait faire référence à l'ancienne par un simple self message (telle que la méthode newi du § II.3.3) laisserait applicable l'ancienne méthode, ce qui pourrait mener à des incohérences dans un problème de simulation (certaines instances seraient initialisées par init, d'autres non).

D'un point de vue plus pratique, super a l'avantage de rendre plus concis le code d'une méthode affinée. Il permet d'éviter de recopier le code de l'ancienne méthode dans la nouvelle, si celle-ci y fait référence. Il suffit de comparer les deux versions de calcule-sortie de NotTps, sans et avec super.

Enfin, la technique du super peut être utilisée pour des besoins tout autres d'efficacité de l'héritage puisque la classe de départ pour son parcours est de niveau plus élevé dans la hiérarchie que la classe d'instanciation, comme c'est le cas pour self.

Reprenons la dernière version de notre exemple, elle peut s'écrire comme suit :

(*) Sans entrer dans les détails, le modèle métacirculaire SMALLTALK est plus complexe et new est seulement héritée par Class de la sur-métaclasse (abstraite) Behavior. Celle-ci décrit le comportement commun à toutes les classes, dont les classes, leurs classes (métaclasses), leur classe (la méta-métaclasse "Métaclass"), sa classe (la méta-méta-métaclasse "Metaclass class") qui héritent toutes de Class (et donc de Behavior, elle même sous-classe de Object).

NotGraphique

gentrée: état

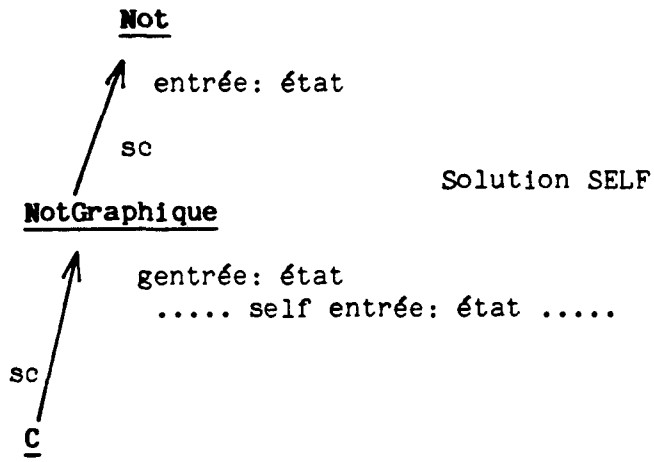
super entrée: état

même s'il n'y a pas de problème de masquage, super et self référencent le même objet destinataire du message gentrée: état. La recherche d'entrée: état commencera ici à la classe Not au lieu de la classe d'instanciation de l'objet concernée qui peut être la sous-classe NotGraphique elle-même ou toute autre sous-classe éventuellement plus profonde qui hérite de celle-ci. Remarquons cependant que ces deux dernières versions ne sont équivalentes que pour :

* les instances de NotGraphique

* les instances d'une sous-classe de NotGraphique telle que la méthode "entrée: " ne soit pas redéfinie sur le chemin entre cette sous-classe (comprise) et NotGraphique.

En effet, supposons qu'une sous-classe de NotGraphique, soit C (*) redéfinisse la méthode "entrée:" :

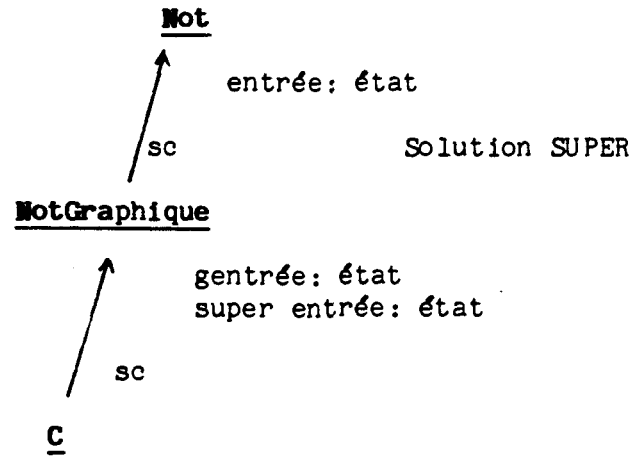


entrée: état.

Pour une instance de C , la méthode entrée: appelée par gentrée: sera effectivement la plus affinée, c'est-à-dire celle définie dans C.

(*) on s'assurera facilement que le fait que C soit une sous-classe directe de NotGraphique ne restreint pas la généralité de ce problème.

La méthode gentrée: est donc implicitement affinée pour les instances de C , grâce à l'utilisation de self (*).



Cette fois, pour une instance de C , la méthode entrée: appelée par gentrée: sera retrouvée dans Not et non dans C , l'affinement de entrée: n'aura donc aucun effet sur gentrée:

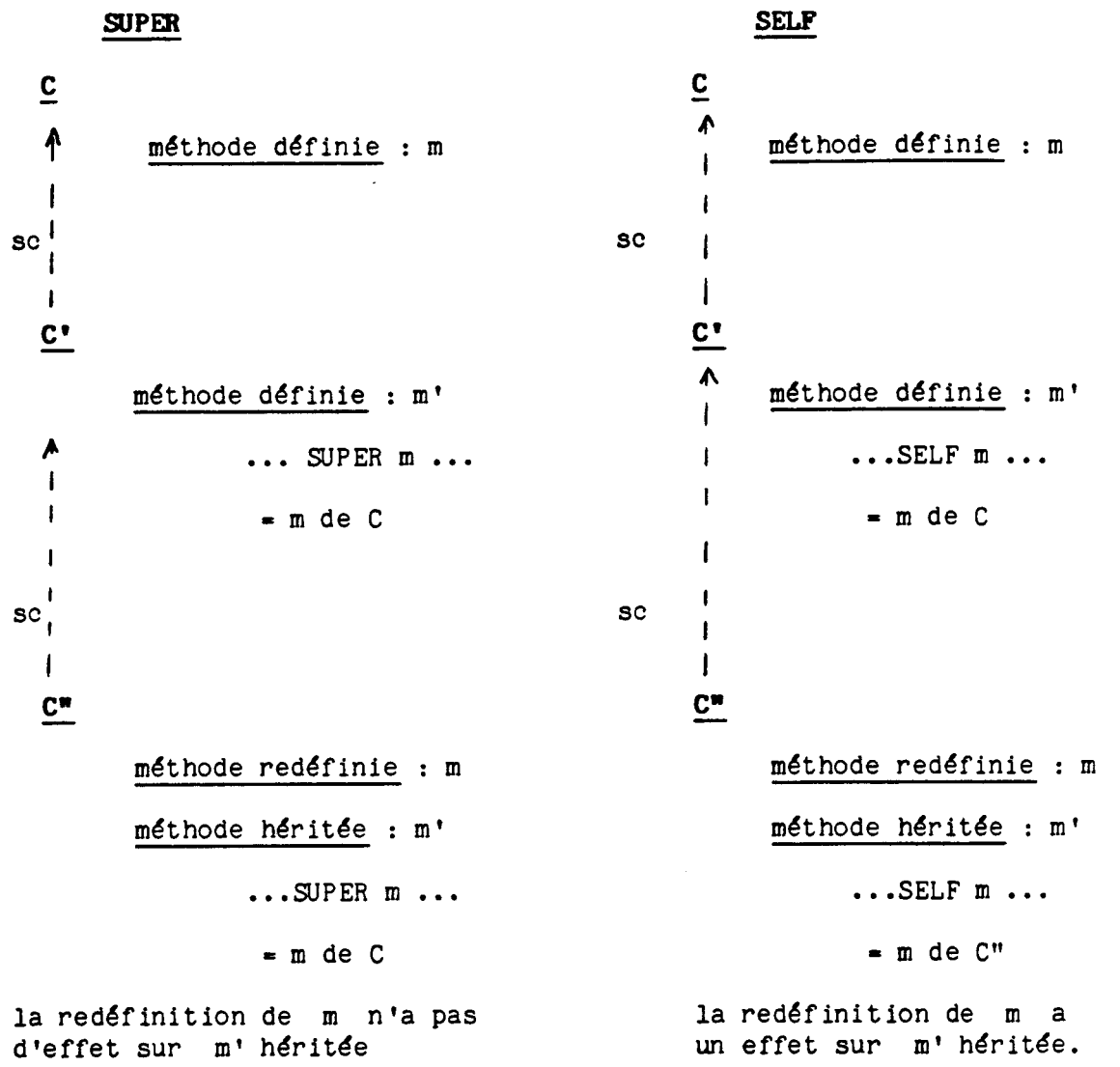
Ce dernier point met l'accent sur un problème général à l'utilisation du super que l'on appellera l'inhibition de l'affinement et que l'on retrouvera dans le paragraphe suivant pour les stratégies graphiques. Le problème est le suivant :

> soient m et m' deux méthodes définies respectivement par les classes C et C', C' étant sous-classe de C et m' invoquant m par l'expression super m , tout affinement par substitution ultérieur de m dans une sous-classe C" de C', n'aura aucun effet sur m' pour les représentants de C".

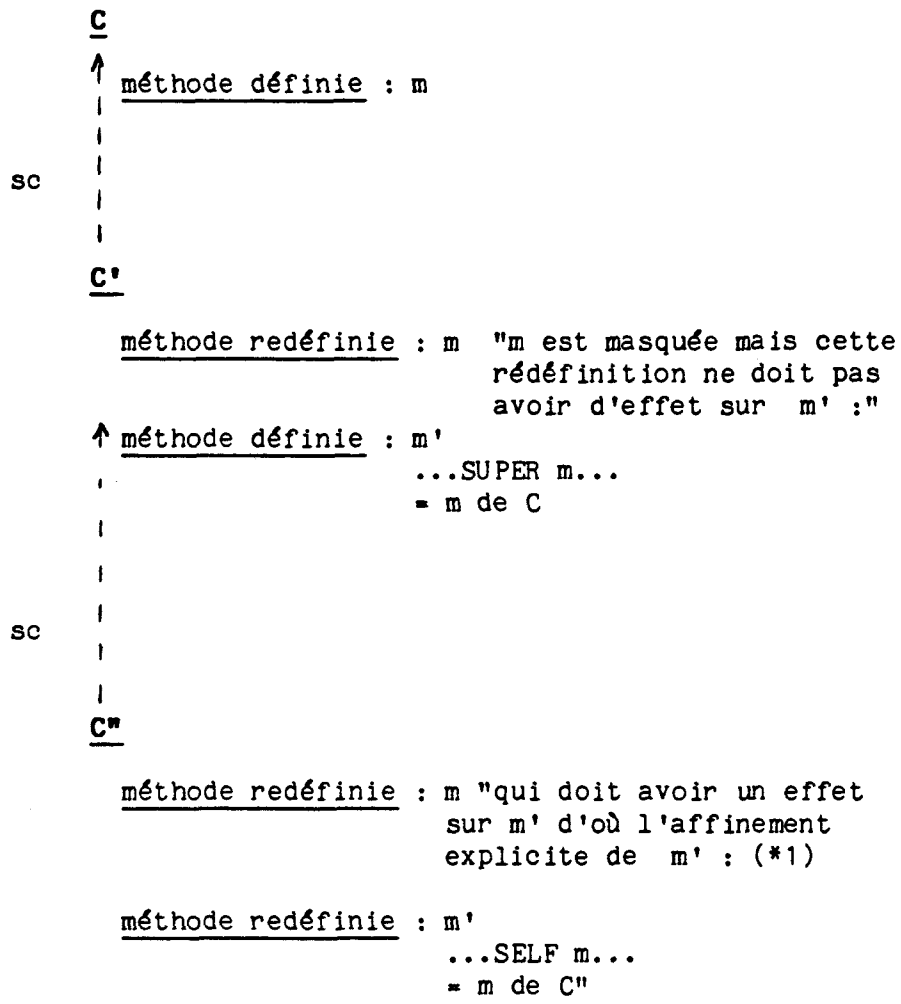
En d'autres termes, la définition de m prise en compte dans le contexte de m' par SUPER est figée, c'est celle de C , et on ne pourra bénéficier de l'affinement implicite de m' par affinement ultérieur de m comme ce serait le cas si self avait été utilisé.

(*) Cette propriété de self a été montrée dans le paragraphe précédent.

En résumé :



Si l'utilisation du super est indispensable, pour des raisons de masquage, il faut alors affiner explicitement la méthode appelante, m' en l'occurrence. Par exemple :



Il est assez difficile de donner des règles générales de résolution de ce problème qui dépendent fortement des cas (*2). En conclusion, la notion de super message est assez délicate et un soin particulier doit être pris pour son utilisation; nous retiendrons cependant son intérêt pour l'affinement incrémental des méthodes.

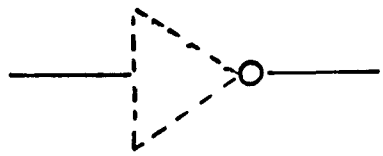
(*1) Remarquons que la méthode m peut très bien ne pas être redéfinie, c'est-à-dire que m de C' convient et qu'au niveau de C" elle peut maintenant devoir être prise en compte dans le contexte de m'. De la même façon, m' y fera référence par SELF m et c'est effectivement m de C' qui sera appliquée, grâce à l'héritage.

(*2) Examiner par exemple le cas où m' est elle-même un affinement de m(!).

III.3.4.3. et les autres...

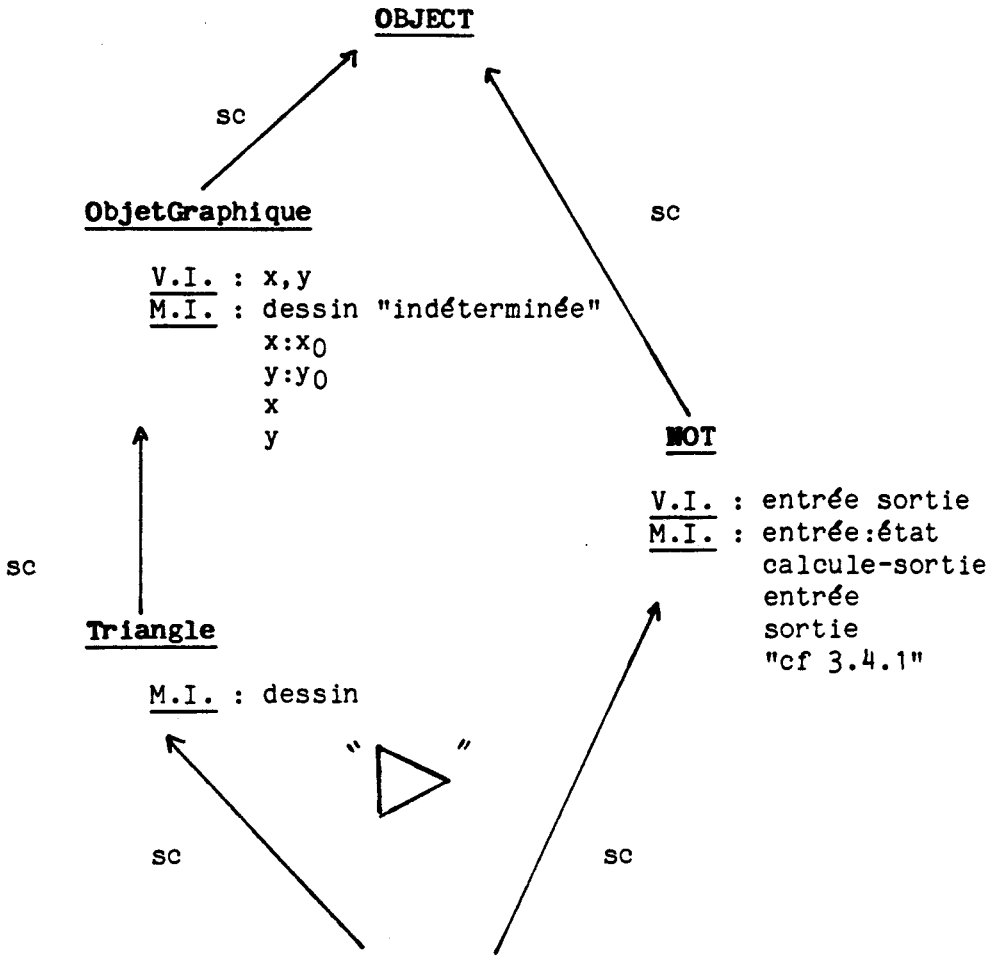
Nous avons vu que self, et sa forme voisine super, est à la base de la définition de méthode par combinaison d'autres méthodes. D'autres mécanismes de combinaison de méthodes sont parfois offerts par les outils orientés objet notamment pour les problèmes d'héritage multiple.

Remarquons, tout d'abord, que bien que les principes de base de combinaison de méthodes aient été montrés sur des exemples ne nécessitant que l'héritage simple, ceux-ci sont généraux, du moins en intention, quelque soit l'héritage. Dans le cas de l'héritage simple, une méthode peut être la combinaison par self (ou super) messages de méthodes disponibles le long de la branche d'héritage à laquelle appartient la classe qui la définit. En héritage multiple, ce principe est généralisé à toutes les branches du treillis d'héritage auxquelles appartient sa classe de définition. S'il n'y a pas de conflits d'héritage, il est assez clair que ceci ne pose pas de problèmes particuliers quelle que soit la stratégie employée (linéaire ou graphique). Montrons ceci sur l'exemple des not graphiques dans un langage de type SMALLTALK, ou plutôt extended SMALLTALK [BORNING-INGALLS.82] la solution serait similaire avec d'autres outils à héritage linéaire ou graphique. Considérons la classe Objet Graphique définissant les caractéristiques communes à tous les objets graphiques, telles que leur position par les attributs x,y et les méthodes "dessin", indéterminée à ce niveau (*) , $x:x_0$, $y:y_0$, x , y de manipulation de coordonnées. La sous-classe Triangle définit de façon évidente sa propre méthode de dessin. La classe NotGraphique peut alors être définie comme sous-classe de Triangle dont elle va affiner la méthode dessin pour ajouter le symbole d'inversion et les 2 fils d'entrée/sortie.



Elle reste sous-classe de Not ("d'un point de vue fonctionnel") dont elle affine la méthode entrée:état pour propager automatiquement l'affichage de la porte à chaque modification de son état. Le treillis d'héritage est donc le suivant :

(*) La classe DisplayObject en SMALLTALK est définie comme une classe abstraite, notion que nous verrons au paragraphe III.3.5


OBJECT
sc
sc
ObjetGraphique
V.I. : x,y
M.I. : dessin "indéterminée"
x:x0
y:y0
x
y
NOT
V.I. : entrée sortie
M.I. : entrée:état
calcule-sortie
entrée
sortie
"cf 3.4.1"
sc
Triangle
M.I. : dessin
sc
sc
NotGraphique
M.I. : dessin
... self super.dessin ...
afficher
"dessin puis affichage des états"
...self dessin ...
entrée:état
self super.entrée:état.
self afficher

Remarquons qu'en extended SMALLTALK, un super message a la forme d'un message à sélecteur composé avec le préfixe réservé "super" et est traité de la même façon (*1). Cette forme fait d'ailleurs plus clairement apparaître qu'un super message est un self message moyennant le forçage de l'héritage à partir de la super-classe. Ainsi le mécanisme du super reste utilisable avec l'extension à l'héritage multiple, s'il n'y a pas d'ambiguïté comme c'est le cas ici dans NotGraphique :

- "self super.dessin" , la méthode dessin de Triangle sera appliquée.

- "self super.entrée:état", la méthode entrée:état de Not sera appliquée.

Super peut paraître cependant superflu pour les stratégies graphiques puisque toute méthode est toujours accessible par le mécanisme de préfixage par une classe. Ainsi, dans NotGraphique :

- "self Triangle.dessin" est équivalent à "self super.dessin"

- "self Not.entrée:état" est équivalent à "self super.entrée:état" (*2)

Ceci termine le cas où il n'y a pas de conflits entre les sur-classes et qui ne fait pas de différence essentielle entre stratégies linéaire et graphique.

A l'inverse, si plusieurs méthodes conflictuelles doivent être combinées un problème se pose pour les stratégies linéaires. En effet, étant donné que seule la première méthode rencontrée par le lookup est accessible, il n'est pas possible en général de réaliser de telles combinaisons. C'est ici que certains mécanismes complémentaires sont nécessaires comme les fonctionnelles de combinaison de méthodes en Flavors appelées "method-combinations". Leur fonctionnement est le

---

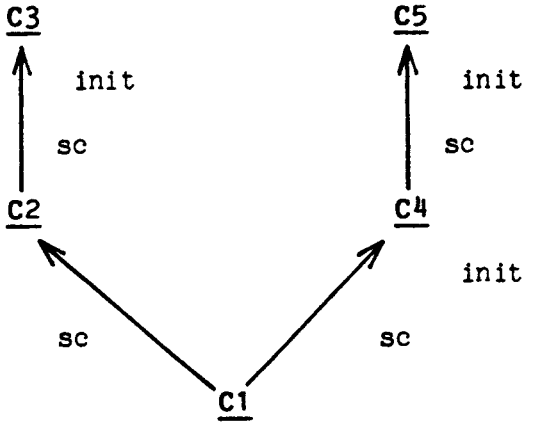
(*1) La forme générale d'un sélecteur composé est "classe.sélecteur". Rappelons que le méthode retrouvée est ajoutée au dictionnaire des méthodes de la classe qui définit la méthode appelante sous ce sélecteur composé. De la même façon, ici la méthode "dessin" de Not sera ajoutée au dictionnaire des méthodes de NotGraphique sous le sélecteur "super.dessin". Notons que le super classique reste valable pour la branche d'héritage la plus à gauche (sur laquelle opère l'héritage simple classique), par exemple "self super.dessin" peut être remplacé par "super dessin" contrairement à "self super.entrée:état".

(*2) En Treillis/Owl et CommonObjects dont l'héritage est graphique, il n'y a d'ailleurs pas d'équivalent du super et ces invocations de méthodes s'écriraient respectivement : Triangle's dessin (me), Not's entrée: (un état) et (call-méthod (Triangle dessin)), (call-méthod (Not entrée:)état) implicitement sur self.

suivant : les méthodes sont d'abord toutes recherchées puis ordonnées selon l'ordre présenté au paragraphe III.3.3. de la plus affinée à la plus générale. La "method-combination" gère alors leur application et leurs résultats. Il existe plusieurs types de "method-combination" prédéfinis par Flavors, chacune ayant son fonctionnement particulier :

- appliquer seulement la plus affinée (par défaut)
- toutes les appliquer, la plus affinée avant ou inversement
- toutes les essayer, la plus affinée avant jusqu'à ce qu'une d'entr'elles rende un résultat différent de nil
- appliquer toutes les méthodes before, puis la méthode principale puis toutes les méthodes after (*1)
- rassembler tous les résultats en liste
- en faire la "somme arithmétique".
  ... [MOON.86]

Cette technique permet évidemment une grande liberté mais peut poser des problèmes notamment méthodologiques. En effet, toute méthode étant potentiellement accessible, elle laisse peu de sémantique au masquage et par conséquent à l'affinement par substitution. Certaines de ces fonctionnelles se retrouvent dans d'autres outils sous des formes différentes. Par exemple en Loops une classe C peut définir une méthode m , combinaison de toutes les méthodes m les plus affinées dans les sur-classes de C, c'est-à-dire qui n'ont pas été elles-mêmes redéfinies avant C. De telles méthodes sont appelées "fringe methods" et la fonctionnelle de combinaison est la primitive ← SuperFringe appliquée sur self. Par exemple, la définition de la méthode "init" qui applique toutes les fringe-méthodes "init" héritées de plusieurs sur-classes se ferait comme suit :




```
init
(←SuperFringe self init)
```

qui consiste en l'application de "init de C3" et "init de C4".

(*1) En Flavors, à toute méthode (main ou primary method), peuvent être attachées deux méthodes (démons) "before" et "after" qui sont respectivement appliquées avant et après la méthode principale, ce qui est une autre façon de combiner des méthodes. Vis à vis de l'affinement par substitution de ces méthodes, elles sont gérées différemment : par défaut la "main method" suit la règle du plus affiné tandis que toutes les parties before et after sont appliquées dans l'ordre d'héritage. Ces méthodes sont une généralisation des réflexes des langages de frames (Cf. III.3.5).

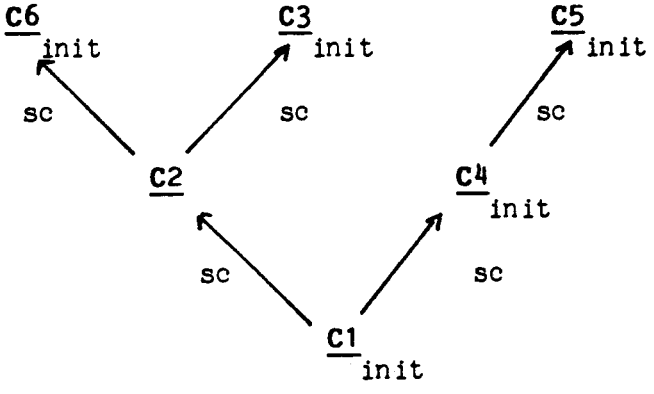
En ce qui concerne les stratégies graphiques, les conflits ne posent pas de problème a priori puisque toute méthode est potentiellement accessible par préfixage par sa classe de définition. Ainsi en reprenant l'exemple précédent, la méthode init de C1 serait définie comme suit en extended SMALLTALK :

```
C1

 init

     self C3.init...
     self C4.init...
```

et même

```
C1

 init

     self C2.init
     self C4.init
```

s'il n'y a pas d'ambiguité sur C2.init. En effet pour ce message composé, le lookup est lancé à partir de C2; si la seule méthode retrouvée est init de C3, C2.init et C3.init sont équivalents. Par contre, si init est définie également dans une autre sur-classe de C2, il y a conflit (erreur) qui doit être levé explicitement (*1)



self C2 . init . ...→ erreur ...→ self C3 . init
self C4 . init (ou C6 . init ou les deux selon le problème).

(*1) plus généralement le message "self m" avec m héritée par plusieurs classes provoque une erreur sauf dans le cas où la méthode est la même, c'est-à-dire si ces classes l'ont héritée d'une même super-classe. Treillis/owl réagit de la même façon contrairement à CommonObjects qui provoque dans tous les cas une erreur.

Nous en venons aux problèmes posés par cette stratégie. Le premier est la remise en cause de la règle du plus affiné qui n'a plus beaucoup de raison d'être ici puisque toute définition d'une méthode est accessible indépendamment du masquage. Par exemple C1 peut faire référence à init de C5 par self C5.init . Tout comme les "méthod-combination" de Flavors, ce mécanisme offre une grande liberté mais enlève une grande part de l'intérêt à l'affinement par substitution. Nous n'insisterons pas sur cet aspect que nous négligerons par la suite, notre approche laissant un rôle important à l'affinement. Comme transition au second problème, il est assez clair que l'héritage explicite est une généralisation du super. En effet, retrospectivement, le super est un cas particulier de cette technique où seules la (ou les, par ←SuperFringe à la loops) sur-classes directes d'une classe peuvent être désignées. Nous avons vu que cet échappement à la règle du masquage était nécessaire pour l'affinement incrémental des méthodes par petits pas mais qu'il posait le problème d'inhibition de l'affinement. Ce problème se pose d'une façon encore plus nette ici puisqu'il s'agit d'une généralisation à la référence directe, explicite et figée de toute méthode définie dans les surclasses (indirectes) de la classe.

> "This explicitness can cause problems because methods build in as constants information about the class hierarchy, which may change".
>
> [BOBROW and al. 86]

Nous ne parlerons même pas de primitives telles que DoMethod, TryMethod ou ApplyMethod en Loops qui sortent du paradigme orienté objet [STEFIK and BOBROW. 85], ou plutôt du paradigme orienté héritage (*) et qui ne nous concernent pas ici.

Dans notre contexte, l'utilisation de l'héritage explicite est intéressant, en tant que mécanisme de résolution de conflits restreint aux fringe-methods par respect de la règle du plus affiné : c'est cette approche que nous considérerons implicitement.

(*) L'héritage n'a plus son rôle implicite pour de telles primitives contrairement à la primitive d'envoi de message. Rappelons que pour celle-ci l'héritage apparaît à travers la fonction (méthode-associée...) avec la définition :
(send object selecteur argt1 ... argtn)
<=> (funcall(method-associée sélecteur objet) objet argt1 ... argtn)
(Cf. II.2.1). Ici method-associée est remplacée par une simple fonction d'accès directe à la classe, soit méthode-définie-dans :
(DoMethod objet sélecteur classe argt1...argtn) <=>
(funcall (méthode-définie-dans-classe sélecteur objet)
objet argt1...argtn).

III.3.5.- Intérêt et utilisation de l'héritage

Après cette introduction un peu technique sur l'héritage, nous allons essayer de dégager l'intérêt des notions de sous-classe et d'héritage. De cette présentation se dégageront quelques interprétations de la hiérarchie conceptuelle des classes, ses propriétés essentielles et certaines règles méthodologiques pour sa construction. Dans la première partie (III.3.5.1 à III.3.5.4) nous considérerons l'héritage comme un simple mécanisme d'inférence implicite induit par la notion de sous-classe, approche que nous pourrions qualifier de conceptuelle et qui nous concerne directement dans un contexte de représentation des connaissances. Puis nous verrons succinctement (III.3.5.5) que cette utilisation de l'héritage est en fait une restriction et que celui-ci peut être détaché de la notion de sous-classe dans d'autres contextes. Nous n'insisterons pas sur cette dernière approche qui sort de notre cadre conceptuel.

III.3.5.1.- Classification simple et multiple, points de vue

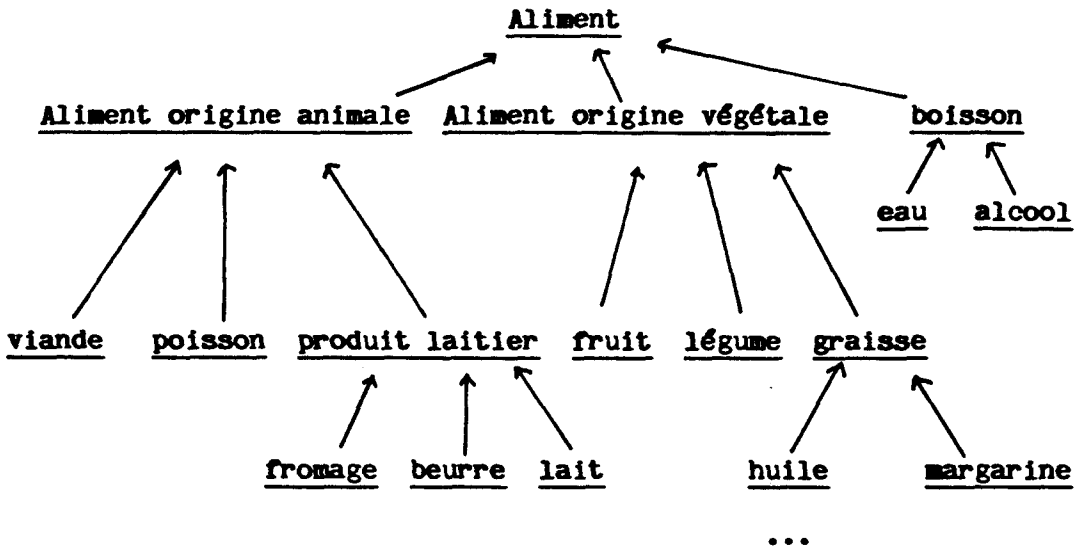
La notion de sous-classe est évidemment un outil privilégié pour construire des classifications, des taxonomies d'entités.

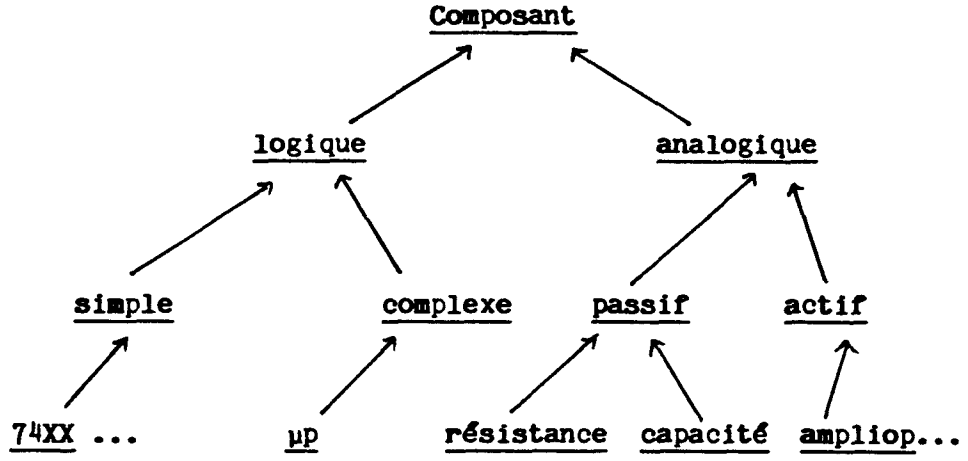
> "In object oriented system, classes are used to make category distinctions, and the class lattice is used as a taxonomy of types."
>
> [MITTAL and al. 86]

De nombreux exemples sont présentés dans la littérature, ne serait-ce que la classification des objets en SMALLTALK qui fournit une hiérarchie de base d'une centaine de classes. Dans [WEGNER. 86] est montrée l'intérêt philosophique de la classification et toute son universalité qui en fait une activité essentielle que l'on retrouve dans tous les domaines, comme en biologie (DARWIN) ou en mathématiques.

> "Classification arises from the universal need, in any domain of discourse, to describe uniformities of collection of instances. It is a basic activity of infants in organizing sense impressions, of scientists in organizing knowledge within scientific disciplines, and of application programmers in organizing domain knowledge and behaviour".
>
> [WEGNER. 86]

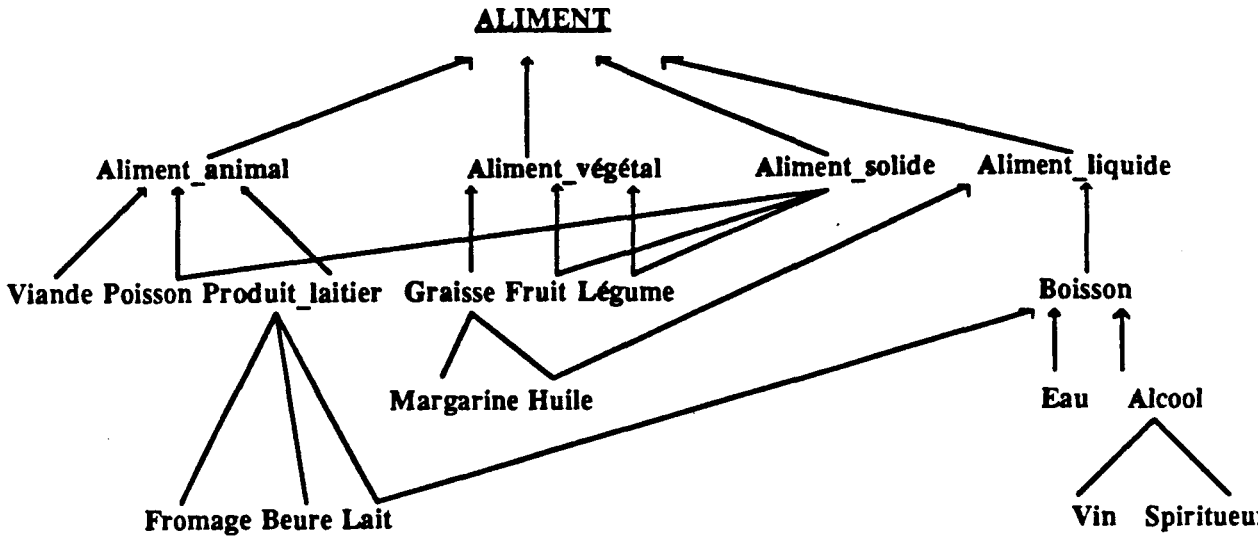
Cette généralité montre tout l'intérêt et l'aspect naturel de la notion de sous-classe des outils orientés objet, qui est à la base un concept puissant pour la représentation des connaissances. Nous ne prendrons que deux exemples simples :





Ces classifications seront traduites directement en héritage simple. L'héritage multiple permet des classifications multiples, liées à une multiplicité des liens IS-A. entre concepts (*). Le premier exemple peut être étendu comme suit :

(*) entre classes ! dans le modèle de base classe/instance/héritage. Rappelons que tout objet est instance d'une classe unique (Cf. postulat P3, II.3.1) par laquelle il peut être représentant de plusieurs classes, les sur-classes de sa classe d'instanciation. Cette contrainte est à la base de nos travaux et sera discutée au chapitre IV pour être remise en cause dans notre noyau orienté objet ROME.

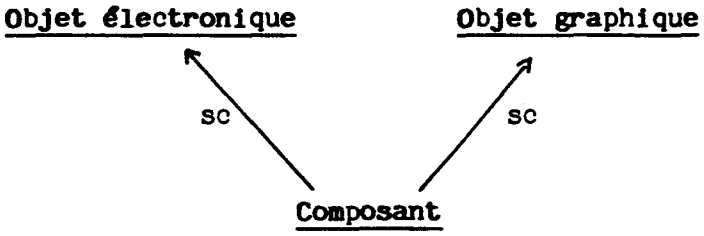


De la même façon, en plus des classifications logique/analogique des composants, nous pourrions les classifier par technologie, par boîtier, grâce à l'héritage multiple.

On constate ici l'aspect déclaratif d'une représentation orientée objet puisque ces regroupements "naturels" pour un expert sont copiés directement dans la représentation informatique. Cette représentation lui assure par l'héritage une inférence implicite des caractéristiques qu'il aura spécifiées pour chaque classe. L'héritage multiple permet en plus une approche du type "multi-expertise", chaque expert ayant sa propre classification selon son point de vue sur les objets. Chaque point de vue concerne un aspect particulier de l'objet, une perspective sur l'objet. Ces termes ont été introduits la première fois dans [GOLDSTEIN-BOBROW. 80], article sur leur environnement expérimental P.I.E. (Personal Information Environment) qui étendait SMALLTALK-76 selon plusieurs directions. Ici, il s'agissait d'une première expérimentation de l'héritage multiple en SMALLTALK-76.

> "SMALLTALK-76 does not support multiple inheritance. Classes are organized into a strict hierarchy and an instance can be associated with only one class, at a single position in the hierarchy. However, there are situations in which one desires greater descriptive power. For example consider an environment for hardware design. Objects in this environment represent circuit elements... resistors, chips, wires, etc... There are at least two points of view from which one may wish to examine these objects. The first is as circuit elements with associated electrical behavior; the second is as display objects that know how to draw pictures of themselves".
>
> [GOLDSTEIN-BOBROW.80]

Ces notions ont été reprises ensuite en Loops (*1). Reprenant leur exemple, un composant, d'un point de vue électronique (expert électronicien) est un élément de circuit et d'un point de vue graphique (expert informaticien graphique) est un objet graphique. Ce qui peut être représenté en héritage multiple comme suit :



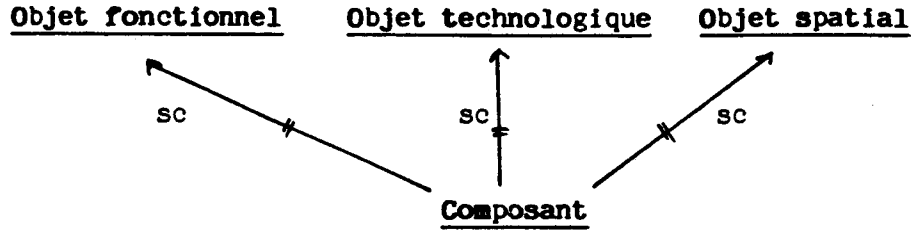
Cet exemple peut être étendu à d'autres points de vue. Dans la conception d'un projet électronique [NIESSEN.86] [BEGG.84]. Les différents niveaux de conception considèrent un composant selon des points de vue différents :

* fonctionnel
* technologique (contraintes électriques : délais, temps de réponse, dissipation)
* spatial (occupation du boîtier pour le placement-routage)

d'où une solution qui serait le schéma d'héritage multiple simplifié (*2)

(*1) Nous verrons en fait, au paragraphe III.5.2. qu'il ne s'agit pas à proprement parler d'héritage multiple entre classes, mais d'une simulation par agrégation d'objets, contrairement à la solution des sélecteurs composés [BORNING-INGALLS.82], Cf. III.3.3. Nous retiendrons cependant le terme de point de vue pour l'héritage multiple.

(*2) Nous verrons en IV que la solution n'est cependant pas aussi simple, ce qui entraînerait des débats qui n'ont pas d'intérêt direct à ce niveau de la présentation.



Cette approche met l'accent sur l'interprétation de la **notion de classe en tant que point de vue sur les objets qu'elle décrit** (*) [CARRE-COMYN. 87ab]. Considérer la classe objet fonctionnel, c'est regarder un composant d'un point de vue fonctionnel et hériter des caractéristiques correspondant à cet aspect. D'autres exemples intuitifs permettent de constater cet intérêt de l'héritage multiple comme :

- les points de vue du nutritionniste et du biologiste sur un nutriment.
- considérer une personne des points de vue professionnel, loisirs, social...
- considérer un cheval en tant que moyen de locomotion ou simple animal.
- considérer le bois en tant que matériau de chauffage ou matériau de construction.

Nous reviendrons assez souvent sur cette interprétation, notamment aux paragraphes III.3.5.4, III.5.2 et surtout aux chapitres IV la notion de point de vue étant centrale dans notre modèle orienté objet ROME.

III.3.5.2.- Abstraction

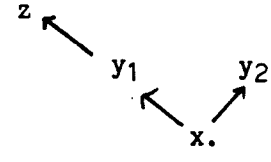
Nous avons pu constater les propriétés d'abstraction du modèle orienté-objet à travers plusieurs notions :

- la notion d'objet elle-même grâce à l'encapsulation par laquelle l'implémentation interne, attributs et méthodes, est abstraite par l'interface, le protocole externe :

  "Objects that are data abstractions with an interface of named operations and a hidden local stage".

  [CARDELLI-WEGNER.85]

(*) Cette interprétation est d'ailleurs reprise intuitivement dans la littérature pour considérer les caractéristiques héritées d'une sur-classe particulière. Par exemple dans [SNYDER.86b], en rapport au schéma d'héritage multiple l'auteur considère une caractéristique héritée de y_1 par x en exprimant ceci par "from y_1's point of view"



- la notion de classe, outil d'obstraction d'objets semblables

  "abstraction mechanism-tools for describing the common structure of similar phenomena".
  [NYGAARD.86]

- la notion de sélecteur et l'envoi de message grâce au polymorphisme

  "polymorphism, a notion that may be defined as the ability to define program entities that may take more than one form".
  [MEYER.86]

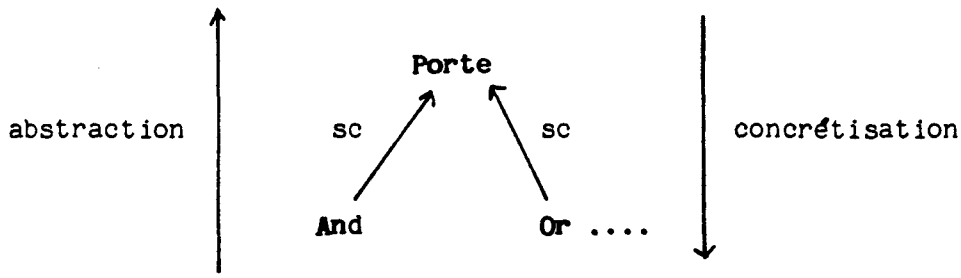
  "in the context of object-oriented programming, it refers to the capability for different classes of objects to respond to exactly the same protocol".
  [STEFIK-BOBROW.85]

Un sélecteur est un nom générique abstrait (generic function en Flavors) auquel peuvent être associées plusieurs implémentations de méthodes.

Nous allons examiner ici l'abstraction supportée par la hiérarchie de classes et l'héritage. L'héritage étend l'abstraction réalisée par la notion de classe puisqu'il permet, comme nous l'avons déjà précisé, de définir des objets plus ou moins semblables :

"inheritance is the concept in object oriented languages that is used to define objects that are almost like other objects".
[STEFIK-BOBROW.85]

Dans bien des cas de classification, une sur-classe est l'abstraction de ses sous-classes. Par exemple :



La classe Porte est l'abstraction des classes And, OR... . La hiérarchie de classes s'interprète alors comme une hiérarchie d'abstraction/concrétisation. L'abstraction est réalisée par le processus de factorisation dans la sur-classe des caractéristiques communes aux sous-classes.

Concernant les spécifications externes, la sur-classe factorise les sélecteurs communs, c'est-à-dire le protocole partagé par les sous-classes. La hiérarchie de classes est donc implicitement un support pour le polymorphisme :

> "Thus polymorphism is intimately related to the notion of inheritance, and we can say that the expressive power of object-oriented types systems is due in large measure to the polymorphism they facilitate".
>
> [CARDELLI-WEGNER.85]

Ce type de polymorphisme est appelé dans [CARDELLI-WEGNER.85] polymorphisme d'inclusion, en vertu de l'interprétation ensembliste de la hiérarchie de classes. Inversement, les sélecteurs spécifiés par une sous-classe forment un sur-ensemble de ceux de ses sur-classes, un représentant d'une sous-classe répond en particulier au protocole défini dans les sur-classes de celles-ci grâce à l'héritage. C'est une propriété essentielle des notions de sous-classe et d'héritage. Soient A et B deux classes, B sous-classe de A alors :

> "At any point where we expect an object of class A, an object of class B will satisfy our needs, because it will accept all the messages an object of class A would accept".
>
> [AMERICA.87](*1)

Par exemple, supposons que la classe Porte spécifie le sélecteur "print". Au message correspondant, toute porte répondra en imprimant ses caractéristiques. Pour le tirage de la nomenclature d'un circuit, il est alors aisé d'envoyer le message print à toutes les portes indépendamment du fait que ce soient des portes and, or, ... De la même façon, en supposant que la classe Porte spécifie le sélecteur "calcule-sortie", d'évaluation de la sortie, le message correspondant peut être envoyé à toute porte dans une phase de simulation, indépendamment de son identité plus fine en tant que and, or,... Qu'en est-il alors des méthodes associées aux sélecteurs ? Nous pouvons distinguer 3 cas principaux selon qu'au sélecteur partagé dans la sur-classe peut être associée :

1 une implémentation totale

2 une implémentation partielle

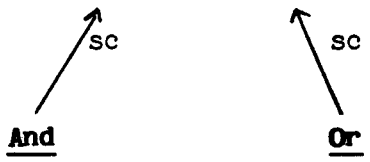
3 aucune implémentation

de la méthode. Nous exposerons succinctement les deux premiers cas pour insister plus particulièrement sur le dernier, typiquement lié à l'abstraction.

(*1) C'est le cas de l'héritage avec "sous-classement" (subclassing) que nous considérons implicitement dans notre approche conceptuelle. Nous verrons cependant au III.3.5.5. que ces deux notions ne sont pas forcément liées.

1 : Le premier cas est assez simple, il s'agit d'une méthode complétement partagée par les sous-classes. La méthode implémentée au niveau de la sur-classe est simplement héritée par les sous-classes. C'est le cas par exemple des méthodes : entrée1: , entrée2: , dans l'exemple suivant :

```
Porte

   attributs   entrée1 , entrée2 , sortie

   méthodes    entrée1: état

               (état isKindOf Boolean)
                  ifTrue: [entrée1 ← état]
                  ifFalse: [self error:
                              'état non booléen']

               entrée2: état

                  "similaire"

               entrée1

                  ^entrée1

               entrée2

                  ^entrée2

               ...
```



Nous n'insisterons pas sur ce cas en précisant cependant un sous-cas particulier quand la méthode partagée apparaît comme une méthode par défaut valide pour toutes les sous-classes. Le masquage peut alors être utilisé par les sous-classes pour définir une implémentation plus appropriée aux objets particuliers qu'elles caractérisent, c'est masquer pour adapter. Supposons par exemple qu'une classe Polygone définisse une méthode de calcul de la surface associée au sélecteur "surface" selon un algorithme très général pour tout polygone. Cette méthode pourra être redéfinie par masquage dans les sous-classes Rectangle, Triangle, Carré,... en utilisant de façon évidente un algorithme plus approprié à leurs caractéristiques propres. Cette redéfinition de la méthode surface n'est cependant pas absolument nécessaire, puisque la méthode héritée de Polygone convient en particulier pour ses sous-classes, mais peut l'être, ne serait-ce que par souci d'efficacité(*).

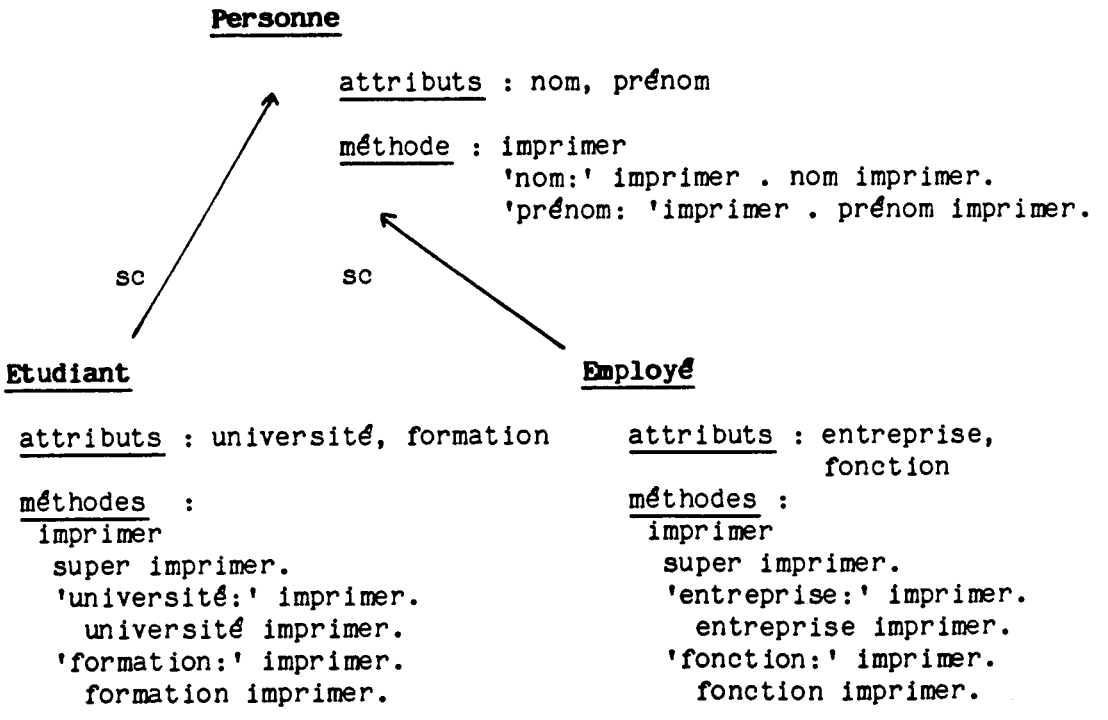
(*) voir notamment en SMALLTALK l'exemple assez révélateur de l'optimisation des méthodes booléennes de BOOLEAN dans les 2 sous-classes True et False (!).

La méthode surface de Polygone apparaît donc comme une implémentation par défaut du calcul de la surface pour tous ses représentants.

2 L'implémentation partielle d'une méthode pour un sélecteur factorisé est directement liée à l'utilisation du SUPER tel que nous l'avons introduit précédemment. SUPER permet une factorisation partielle du code d'une méthode dans la sur-classe qui, inversement, sera complétée dans chacune des sous-classes par masquage :

SUPER ≡ masquer pour compléter.

Prenons l'exemple simple suivant. Supposons que tout étudiant doive répondre au message "imprimer" en imprimant ses nom, prénom, université et formation et que tout employé au même message réponde en imprimant ses nom, prénom, entreprise et fonction. Il est clair que l'implémentation de la méthode "imprimer" (*) peut être partiellement factorisée au niveau d'une sur-classe, soit Personne, comme suit :

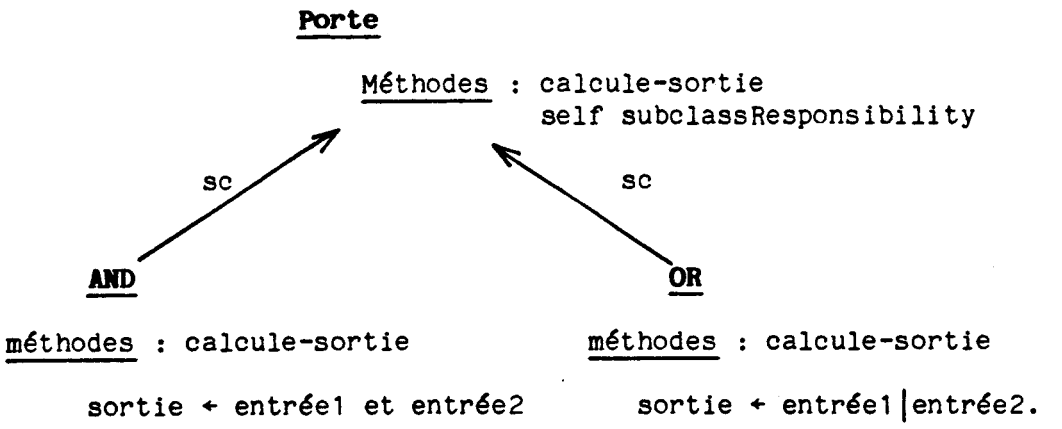


La méthode "imprimer" de Personne apparaît comme une implémentation partielle des méthodes de même nom de Etudiant et Employé.

(*) Nous n'entrerons pas dans les détails d'une implémentation en SMALLTALK qui compliquerait inutilement l'exemple. Sachons simplement que la méthode imprimer devrait être paramétrée par un stream précisant le support de l'impression, fenêtre, fichier, imprimante... (Cf. notamment la méthode printOn: aStream de Object).

3 Insistons maintenant sur le dernier cas, révélateur de la propriété d'abstraction de la hiérarchie de classes. Il s'agit du cas où aucune implémentation de méthode ne peut, au niveau de la sur-classe, être associée au sélecteur partagé par les sous-classes. Cette méthode est donc indéterminée à ce niveau et chaque sous-classe devra l'implémenter par masquage, c'est le masquage pour l'implémentation. En effet, la spécification du sélecteur au niveau de la sur-classe impose que tous ses représentants soient capables de répondre au message correspondant, or aucun comportement (notamment par défaut) n'est défini à ce niveau. Le masquage est donc requis dans ce cas. La méthode de même nom (sélecteur) spécifiée par la sur-classe aura donc des implémentations différentes dans chaque sous-classe, c'est le polymorphisme d'inclusion. Beaucoup de langages offrent la possibilité de déclarer une telle méthode comme indéterminée au niveau de la sur-classe. En Simula ou C++ par exemple, ceci correspond à la notion de fonction virtuelle déclarée "virtual" dans la sur-classe, ce terme signifie "doit être implémentée par les sous-classes" (* 1). Les langages compilés tels que ceux-ci vérifient alors à la compilation que les sous-classes implémentent effectivement ces méthodes.

En SMALLTALK (interprété) pour une méthode indéterminée, le message "self subclassResponsibility" est associé au sélecteur dans la super-classe. Ce self message génère une erreur à l'exécution si la méthode n'a pas été implémentée par la sous-classe (*2) et la vérification est donc dynamique. Revenons sur notre exemple des portes logiques. Toutes les portes doivent savoir calculer leur état de sortie, en réponse au message "calcule-sortie" et donc partagent le même sélecteur calcule-sortie. L'action associée dépend cependant du type de porte concerné, and ou or.... La méthode est donc spécifiée au niveau de la classe Porte en "self subclassResponsibility", et devra être redéfinie par chaque sous-classe :



(*1) "virtual (the Simula 67 and C++ term for "to be defined later in a class derived from this one")" [STROUSTRUP.86]
(*2) Le sélecteur "subclassResponsibility" est défini pour tout objet (donc dans OBJECT); le code associé est le self message "self error: 'my subclass should have overriden one of my messages'" (en SMALLTALK/V ce sélecteur est équivalent à "implementedBySubclass messages" qui ouvre une fenêtre de label "should be implemented by subclass").

Toute nouvelle sous-classe, par exemple NAND, devra procéder de la même façon. Remarquons cependant que ni la spécification de calcule-sortie au niveau de Porte, ni sa déclaration en subclassResponsibility ne sont obligatoires d'un point de vue technique. En effet, si toute sous-classe définit la méthode calcule-sortie, le masquage a pour effet d' "oublier" sa spécification au niveau supérieur. D'autre part, s'il n'est pas spécifié au niveau de Porte et qu'une sous-classe ne le définit pas, tout envoi de message correspondant résultera de toute façon en une erreur "message not understood" à l'exécution.

Cette technique est donc plus un guide méthodologique et une aide au concepteur d'une sous-classe. Si celui-ci oublie de définir la méthode associée, il sera prévenu (à l'exécution) par une erreur "this method should have been implemented by my subclass". D'un point de vue méthodologique, elle garantit la complétude des spécifications à chaque niveau d'abstraction correspondant à une classe : Une méthode définie par une classe ne peut être la combinaison par self message que de méthodes spécifiées ou héritées par cette classe, même si elles sont indéterminées à ce niveau (*).

Ce principe permet notamment une bonne lisibilité des définitions et assure une conception incrémentale des classes, chacune d'elles n'ayant connaissance que d'elle-même et des classes supérieures. Par exemple, supposons que la méthode calcule-sortie soit référencée par les méthodes entrée1: un_état et entrée2: un_état par self message au niveau de Porte. La méthode calcule-sortie doit donc être spécifiée au même niveau même si elle est indéterminée, son implémentation ultérieure dans les sous-classes sera implicitement prise en compte pour les représentants de celles-ci :

(*) Bien qu'il n'y ait aucune vérification statique de ce genre par l'interpréteur.

```
Object subclass: #Porte
        instanceVariableNames: 'entree1 entree2 sortie '

Porte comment:
'Classe abstraite des portes.'

Porte methodsFor: 'writing'

entree1: etat
        "ecriture de l'entree1 et propagation de la sortie"

        (etat isKindOf: Boolean)
                ifTrue: [entree1 <- etat. self calculeSortie]
                ifFalse: [self error: 'etat non booleen']

entree2: etat
        "ecriture de l'entree2 et propagation de la sortie"

        (etat isKindOf: Boolean)
                ifTrue: [entree2 <- etat. self calculeSortie]
                ifFalse: [self error: 'etat non booleen']

Porte methodsFor: 'initializing'

init
        entree1 <- false.
        entree2 <- false.
        sortie <- false

Porte methodsFor: 'evaluating'

calculeSortie
        "indeterminee a ce niveau abstrait"

        self subclassResponsibility

Porte methodsFor: 'reading'

entree1
        "lecture de l'entree1"

        ^entree1

entree2
        "lecture de l'entree2"

        ^entree2

sortie
        "lecture de la sortie"

        ^sortie
```

```
Porte class

Porte class comment:
'metaclasse de Porte'

Porte class methodsFor: 'instanciation'

new
        ^super new init



Porte subclass: #And
        instanceVariableNames: ''

And methodsFor: 'evaluating'

calculeSortie
        sortie <- entree1 & entree2



Porte subclass: #Or
        instanceVariableNames: ''

Or methodsFor: 'evaluating'

calculeSortie
        sortie <- entree1 | entree2
```

Ce type de factorisation est tout à fait caractéristique de l'abstraction supportée par la notion de sur-classe et nous amène à considérer un type particulier de classes appelées classes abstraites

> "Abstract superclasses are created when two classes share a part of their descriptions and yet neither one is properly a subclass of the other. A mutual superclass is created for the two classes which countain their shared aspects. This type of superclass is called abstract because it was not created in order to have instances".
>
> [GOLDBERG-ROBSON.83]

Par exemple, les classes And et Or partagent une partie de leur description, notamment leur protocole complet. L'une n'est cependant pas à proprement parler sous-classe de l'autre (nous reviendrons sur cet avis au III.3.5.5) et une approche naturelle est de les abstraire en une sur-classe commune, Porte. Porte est un exemple de classe abstraite, elle n'a que le rôle d'abstraction de ses sous-classes et

n'est pas susceptible d'être instanciée (*1). Une instance d'une telle classe disposerait d'une caractérisation incomplète notamment si certaines de ses méthodes sont indéterminées. De nombreux exemples de classes abstraites peuvent être trouvés dans la hiérarchie SMALLTALK comme Collection, abstraction de ses sous-classes Set, Bag,... ou Number, abstraction des sous-classes Float, Fraction, Integer,... Donnons un autre exemple inspiré de [STROUSTRUP.86] qui concerne les figures graphiques, cercles, rectangles, triangles, carrés. La classe abstraite Figure va définir les caractéristiques communes à toutes les classes notamment, les attributs position, color et les méthodes where, moveto: et draw de définition évidente :

```
Figure

  sur-classe : objet

  attributs  : position, color

  méthodes   : where

                 ^position

               draw

                 self subclassResponsibility

               moveTo: aPoint

                 position ← aPoint.
                 self draw.
```

(*1) Or, toute classe a la fonctionnalité d'instanciation définie par CLASS! Nous n'insisterons pas sur ce problème ici, il sera évoqué au chapitre IV. Sachons simplement que les classes abstraites n'ont pas de statut spécial en SMALLTALK et dans la plupart des outils apparentés, ce qui découle du modèle de base. Tout au plus peut-on redéfinir la méthode new pour de telles classes, telles qu'elle renvoie un message d'erreur explicite, ce qui est peu satisfaisant d'un point de vue fondamental. L'utilisateur est en fait supposé ne pas instancier de telles classes (essayer pourtant "Number new" en SMALLTALK!). Le problème est alors d'identifier de telles classes. Un signe particulier est la présence du subclassResponsibility dans le code des méthodes qu'elles définissent (ou héritent sans les implémenter pour une classe abstraite, sous-classe d'une classe abstraite) qui se repère par une lecture attentive du code, ce qui est peu satisfaisant :

> "since there is no syntactic difference between an abstract superclass and any other class, a careful examination of the code must be made to determine if a superclass is an abstract superclass and to find the abstraction operation that must be implemented by the subclass"
>
> [SANDBERG.86]

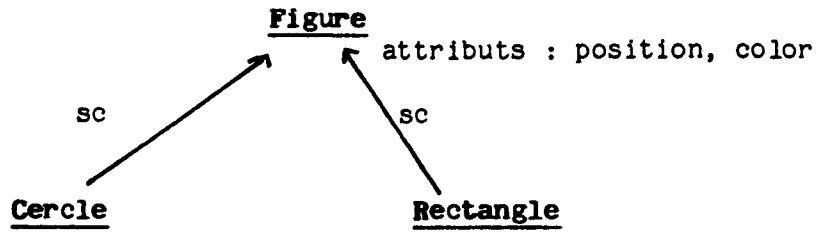
La méthode draw est évidemment indéterminée et sera implémentée par les sous-classes Cercle, Rectangle, Triangle, Carré,...

Une classe abstraite spécifie donc un protocole et éventuellement une implémentation partielle partagés par les sous-classes. A l'extrême, une telle classe ne peut que spécifier un protocole sans aucune implémentation, toutes les méthodes étant indéterminées. L'exemple typique est la description des piles qui peuvent être implémentées par un tableau ou une liste. La classe abstraite Pile spécifie le protocole bien connu : empiler:v , dépiler, vide, pleine dont l'implémentation est indéterminée à ce niveau (subclassResponsibility) et sera définie par les sous-classes décrivant les piles implémentées par un tableau et les piles implémentées par une liste. Ce cas extrême établit un parallèle avec les packages Ada, la sur-classe apparaissant comme la partie spécification du package (sur-classe de spécification), une sous-classe étant la partie implémentation ("body", sous-classe d'implémentation). C'est le cas de l'utilisation de l'affinement pour l'implémentation de spécifications, "subclassing" ou "subtyping for implementation" [HALBERT-O'BRIEN.87]. Ceci montre bien l'abstraction que peut supporter la hiérarchie de classes, l'affinement traduisant l'implémentation, cas extrême de la concrétisation.

Enfin, précisons rétrospectivement une propriété remarquable de l'héritage. Considérons un message correspondant à une méthode indéterminée au niveau d'une sur-classe abstraite et implémentée par les sous-classes (polymorphisme d'inclusion). Tout représentant de la sur-classe comprend ce message mais répond différemment selon la sous-classe à laquelle il appartient. Un tel message résulte donc en une discrimination, requise par le polymorphisme d'inclusion, pour rechercher la méthode à appliquer en fonction de la sous-classe d'appartenance de l'objet destinataire.

Il est évident, à ce niveau de la présentation, que cette discrimination est garantie par l'héritage. Par exemple, pour les figures graphiques, soit une_Figure, un objet représentant de Figure (et non instance de celle-ci (*). Le message "une_Figure draw" résultera en l'application de la méthode draw définie dans Cercle, Rectangle,... selon qu'une figure est instance de l'une ou l'autre de ces sous-classes. La même discrimination opère sur les attributs qui caractérisent les représentants d'une même sur-classe en fonction de leur sous-classe. La sur-classe détermine leurs attributs communs, tandis que chacune des sous-classes peut définir des attributs propres aux objets particuliers qu'elles caractérisent. Par exemple :

(*) (une_Figure isKindOf: Figure) = true ≠ (une_Figure is MemberOf: Figure).



De la même façon, chaque sous-classe peut ajouter ses propres méthodes non partagées par les autres classes (ne seraient-ce que les méthodes d'accès aux attributs). De manière générale, l'héritage assure la discrimination des caractéristiques des figures en fonction de leur sous-classe d'appartenance :

l'héritage est un mécanisme de discrimination implicite de caractérisation.

Cette propriété qui peut paraître évidente à ce niveau de la présentation (*) est pourtant remarquable par rapport à d'autres langages sans héritage, tels que Ada, Modula 2 ou CLU. Dans ce tels langages, autant il est aisé de définir des types d'objets à la façon des classes, autant il est difficile de représenter une hiérarchie naturelle de ces types telle qu'une hiérarchie de classes construite par factorisation. La seule possibilité est la structure de records variables ("variant records") introduite d'ailleurs dès PASCAL. La hiérarchie est alors codée explicitement à travers un discriminant (ou tag) et autant de discriminants sont nécessaires que de niveaux de la hiérarchie. Montrons ceci sur l'exemple des figures dans une syntaxe Ada-like assez intuitive :

```
type TYPE_FIGURE is (CERCLE,RECTANGLE)

type FIGURE(TAG : TYPE_FIGURE) is

    record

        POSITION : POINT;
        COULEUR  : COLOR;

        case TAG is

          when CERCLE => RAYON : REAL;
          when RECTANGLE => LARGEUR, LONGUEUR : REAL;

        end case;

    end record
```

(*) En fait, on peut dire que le problème de la discrimination ne se pose même pas dans une telle approche, c'est pourquoi nous ne présentons cette prorpiété que rétrospectivement.

La discrimination des attributs caractérisant une figure en fonction de son type particulier est explicitement programmée à travers le discriminant TAG. De la même façon, chaque méthode commune (correspondant ici à des procédures) devra gérer cette discrimination en fonction de la valeur du tag de l'objet concerné. Ainsi la procédure draw paramétrée par F de type Figure comportera dans son code une discrimination explicite des cas (*) :

```
case F.TAG is

   when CERCLE => -- dessiner un cercle
   when RECTANGLE => -- dessiner un rectangle

end case
```

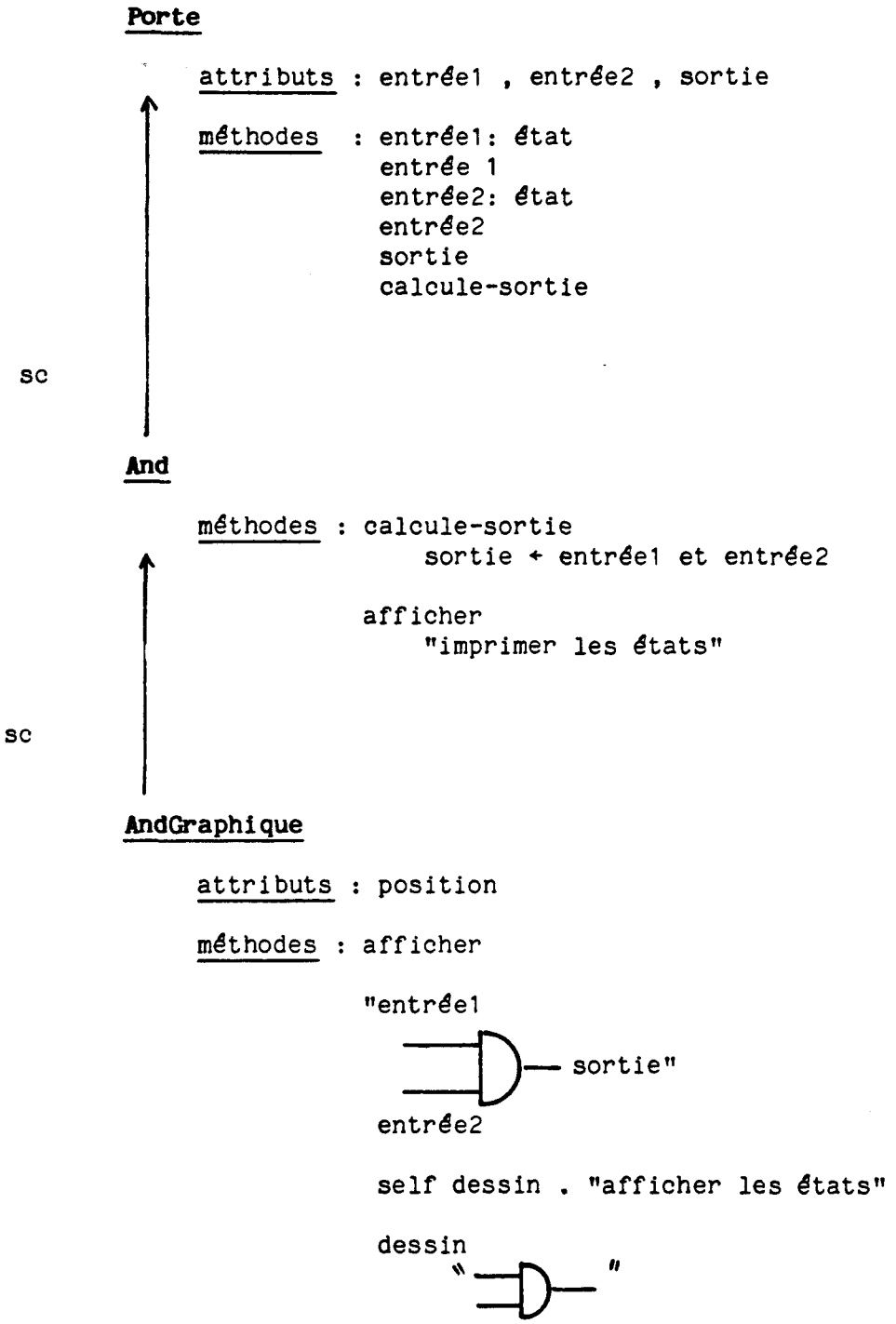
De façon générale dans cette approche, dès qu'une figure est concernée et qu'une discrimination des cas est nécessaire, celle-ci doit être gérée explicitement. D'autre part, il est aisé de constater les contraintes qu'impose cette dernière approche. Par exemple, l'ajout d'un nouveau type de figure oblige à modifier profondément le code partout où une discrimination des cas est nécessaire alors qu'il suffit d'ajouter une sous-classe à la classe Figure sans la modifier et de laisser la discrimination implicitement à la charge de l'héritage dans notre modèle.

Nous n'insisterons pas davantage sur la supériorité évidente de l'héritage face à ce problème qui est largement présenté par ailleurs, voir par exemple [MEYER.86] [STROUSTRUP.87] [HALBERT-O'BRIEN.87]. Nous retiendrons simplement que l'héritage est un mécanisme de discrimination implicite, propriété qui montre une fois de plus l'aspect déclaratif et naturel du modèle classe/instance/héritage.

III.3.5.3.- Spécialisation

Nous avons vu, dans le paragraphe précédent, que la hiérarchie de classes pouvait traduire une relation d'abstraction/concrétisation, la concrétisation étant une forme d'affinement. Une autre forme d'affinement, présentée ici, est la spécialisation d'une classe par une sous-classe. La sous-classe est alors définie pour spécialiser les objets représentants de la sur-classe en rapport à une utilisation dans un contexte plus spécifique. C'est le cas, par exemple, des sous-classes Not_Graphique ou Not_Tps de Not (III.3.4) qui spécialisent respectivement les not dans un contexte d'application graphique ou de simulation temporelle. Prenons ici l'exemple semblable pour les portes And telles que définies dans le paragraphe précédent. De la même façon, nous pouvons définir la sous-classe And_Graphique de And :

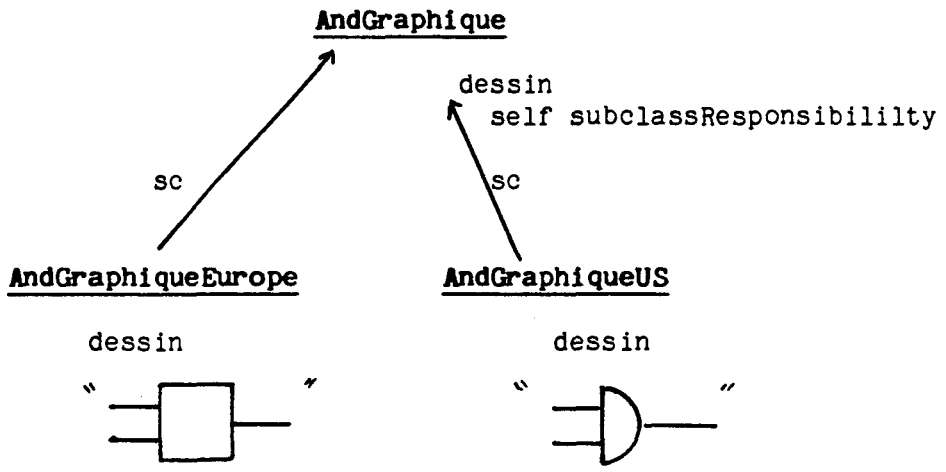
(*) ce type de polymorphisme est appelé polymorphisme paramétrique dans [CARDELLI-WEGNER.85].



Les classes And et AndGraphique correspondent à des niveaux de conception différents de l'application, moins élaboré pour And, par exemple pour une version simplement texte de l'application, plus élaboré pour AndGraphique dans une version graphique du simulateur. De la même façon, AndTps spécialiserait les and dans une version temporelle du simulateur (en tenant compte des problèmes de délais, de la technologie employée,...). L'intérêt de l'héritage pour la spécialisation est de permettre une conception incrémentale d'une application.

Une sous-classe définie par spécialisation réutilise (*) les caractéristiques définies dans la sur-classe et les affine par ajout ou par substitution.

Il est intéressant de constater les rôles différents joués par les classes dans le schéma d'héritage précédent et de placer la spécialisation par rapport à l'abstraction. La sur-classe And ne peut être considérée à proprement parler comme une abstraction de la classe And Graphique. Ce n'est d'ailleurs pas une classe abstraite au sens où nous l'avons présenté précédemment, ne serait-ce que parce que des instances de And peuvent tout à fait être créées, notamment à un niveau assez élémentaire de l'application. Ces instances disposent d'une caractérisation complète. AndGraphique n'est qu'une spécialisation de la classe And qui ajoute les caractéristiques graphiques. Elle pourrait elle-même être une classe abstraite (!). Il suffit de considérer les graphismes différents selon les normes américaine ou européenne, la méthode dessin serait alors indéterminée au niveau de AndGraphique et implémentée par chaque sous-classe respective :



La frontière peut paraître assez floue entre concrétisation et spécialisation bien qu'il soit assez clair que le lien "sous-classe" entre AndGraphique et And n'ait pas la même interprétation que celui entre And et Porte. Concrétisation et spécialisation sont en fait deux formes d'affinement et utilisent les mêmes notions de sous-classe et d'héritage. Nous pouvons cependant attacher ces concepts à des niveaux différents de la conception d'une hiérarchie de classes.

Plus que la concrétisation, c'est l'abstraction qui est importante dans cette première forme. La création d'une sur-classe abstraite

---

(*) c'est la propriété de "réutilisabilité" de l'héritage [MEYER.86].

résulte d'une analyse précise des classes d'objets semblables intervenant dans le problème et utilise la factorisation des caractéristiques communes de leur description respective.

Pour la deuxième forme d'affinement, c'est effectivement la spécialisation qui est importante, plus que son inverse que l'on pourrait nommer assez vaguement la généralisation.

La création d'une sous-classe résulte du besoin de définir des caractéristiques en plus de celles décrites par la sur-classe, utilisant l'héritage. **L'abstraction est liée à un processus d'analyse ascendante, la spécialisation à un processus de conception descendante.** La sur-classe sous laquelle vient se greffer une sous-classe de spécialisation est en général créée et utilisée depuis longtemps. C'est le cas pour notre exemple, la classe And est conçue et utilisée dans une version élémentaire du simulateur. Puis le concepteur de la version graphique la spécialise en And Graphique par affinement de la caractérisation définie par And. Un autre exemple serait la définition des points en trois dimensions en SMALLTALK, par spécialisation de classe Point (2D) existant dans la hiérarchie SMALLTALK de base et largement utilisée pour l'environnement graphique du langage. C'est le cas également de la classe Pen spécialisée en la classe QDPen (QuickDrawPen) dans la version Mac Intosh de l'environnement.

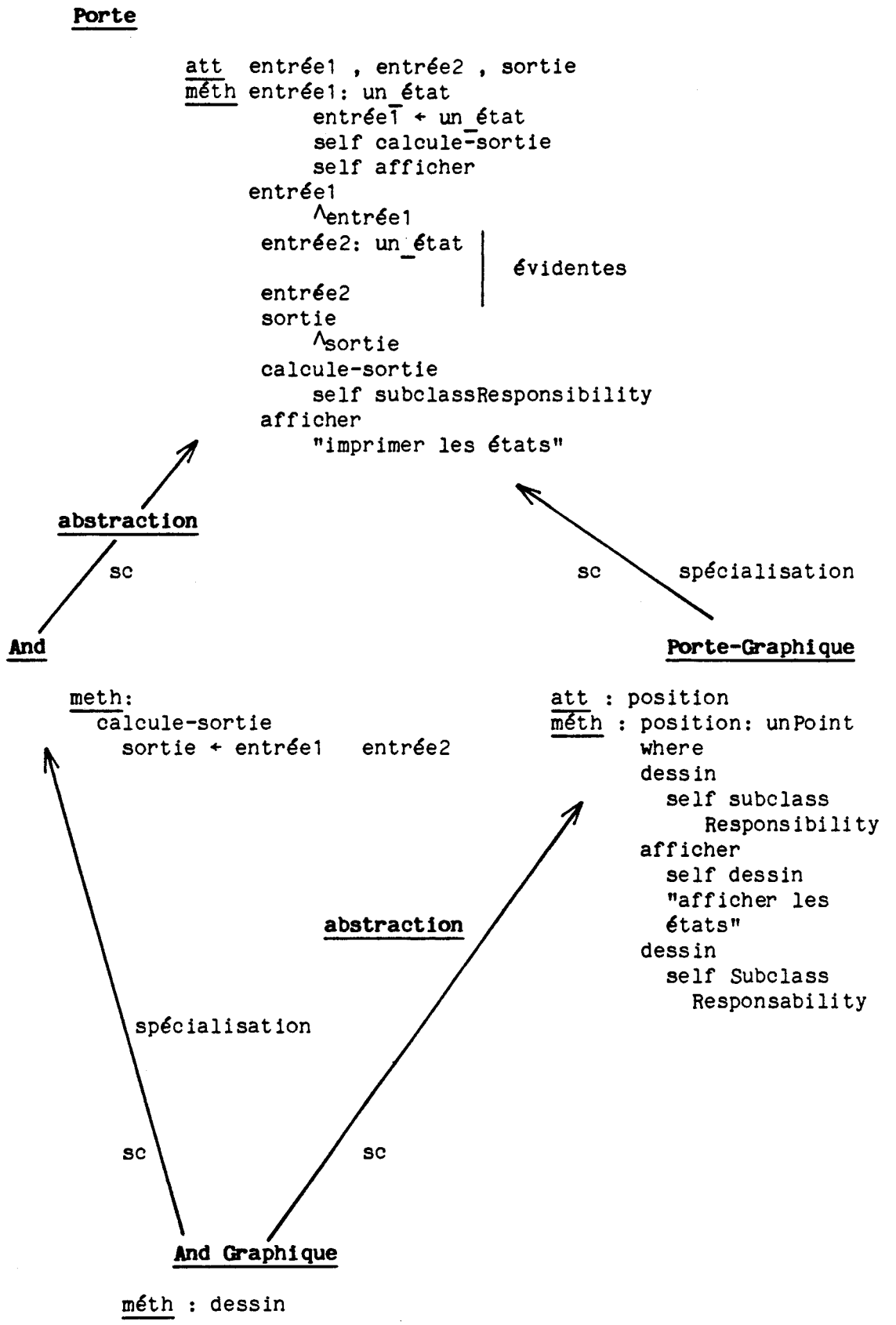
III.3.5.4.- Combinaison

Cette possibilité est propre à l'héritage multiple et vient s'ajouter aux principes précédents généraux aux deux formes d'héritage. L'héritage multiple permet de créer une classe par combinaison de plusieurs autres ou, inversement, d'effectuer des factorisations multiples de classes.

Si l'héritage multiple n'existait pas, il faudrait... recopier. C'est le cas, par exemple [BORNING-INGALLS.82] pour les classes ReadStream, WriteStream et ReadWriteStream, un stream étant un tampon pour l'accès aux fichiers, périphériques et d'autres objets internes. ReadWriteStream partage, de façon assez évidente, les protocoles des 2 classes ReadStream et WriteStream. En héritage simple, il faut choisir l'une des deux sur-classes et copier la définition de l'autre. En SMALLTALK standard, ReadWriteStream est sous-classe de WriteStream. L'héritage multiple permet évidemment de résoudre ce problème plus simplement en définissant ReadWriteStream sous-classe de ces deux classes, en utilisant l'affinement par combinaison.

Reprenons notre exemple des portes logiques pour montrer les possibilités de factorisation multiple. Nous avons vu une première factorisation des portes d'un point de vue fonctionnel qui a mené à la création de la classe abstraite Porte. Considérons maintenant les portes d'un point de vue graphique, en supposant que nous voulons que l'affichage soit automatique à chaque mise à jour de leur état.

Utilisant l'abstraction, ces caractéristiques sont généralisables à toute porte, nous définissons donc une classe abstraite Porte-Graphique, sous-classe de Porte, en tant que spécialisation de celle-ci. Chaque sous-classe And-Graphique, Or-Graphique devra implémenter sa propre méthode de dessin. Elles sont par ailleurs sous-classes par spécialisation des classes respectives And et Or, en généralisant le raisonnement appliqué pour les And dans le paragraphe précédent. Nous obtenons donc le schéma suivant sur lequel nous avons porté l'interprétation du lien Classe-sous-classe.



(idem pour le OR et le OR graphique)

L'héritage multiple permet donc de combiner abstraction et spécialisation pour une même sous-classe :

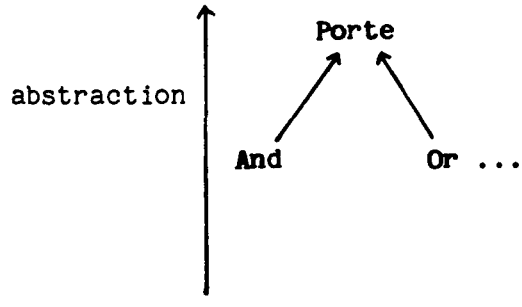
* la classe AndGraphique est un affinement par spécialisation de la classe And et un affinement par concrétisation de la classe Porte Graphique.

* la classe ReadWriteStream présentée initialement est un affinement par spécialisation des deux classes ReadStream et WriteStream.

Nous retrouvons ici l'interprétation de la notion de classe en tant que point de vue sur ses représentants et,en héritage multiple, les sur-classes peuvent être interprétées comme des **points de vue multiples** : Les classes Porte et And correspondent au point de vue fonctionnel sur leurs représentants, notamment les instances de And Graphique, la classe PorteGraphique au point de vue graphique. De la même façon, les représentants de ReadWriteStream sont caractérisées du point de vue accès en lecture par la sur-classe ReadStream, du point de vue accès en écriture par WriteStream.

Une propriété remarquable de l'héritage multiple par rapport à l'héritage simple est de permettre l'abstraction multiple. Les sur-classes abstraites Porte et PorteGraphique sont des abstractions des classes AndGraphique, OrGraphique,... Un lien particulièrement intéressant dans ce treillis est celui qui traduit la spécialisation de Porte en Porte Graphique (*1) . Celle-ci est en fait l'abstraction des spécialisations And/AndGraphique, Or/OrGraphique,... Pour bien constater ceci, reprenons le cheminement qui a mené à la conception de ce treillis d'héritage.

Première phase : description des and, or,... et plus généralement des portes à deux entrées et une sortie semblables à celles-ci (nand, or,...) d'un point de vue fonctionnel. L'analyse des caractéristiques nécessaires pour définir ces objets mène à factoriser les classes And, Or,... en une sur-classe abstraite Porte dont l'existence ne fait pas de doute restrospectivement pour décrire la classification des portes.



(*1) spécialisation abstraite !

Seconde phase : spécialisation des portes pour une version graphique de l'application. Les classes And et Or sont spécialisées respectivement en AndGraphique et OrGraphique. L'analyse des caractéristations graphiques de telles portes montre des similitudes évidentes indépendamment de leurs différences fonctionnelles. Ceci mène à définir une classe abstraite PorteGraphique, abstraction des portes graphiques semblables. Tout comme AndGraphique et OrGraphique sont des spécialisations des classes And et Or, PorteGraphique est naturellement une spécialisation de l'abstraction de ces dernières classes, i.e. de Porte. La classe PorteGraphique vient donc spécialiser la classe Porte d'un point de vue graphique et AndGraphique, OrGraphique qui sont des spécialisations respectivement de And et Or, sont définies comme concrétisations de PorteGraphique :

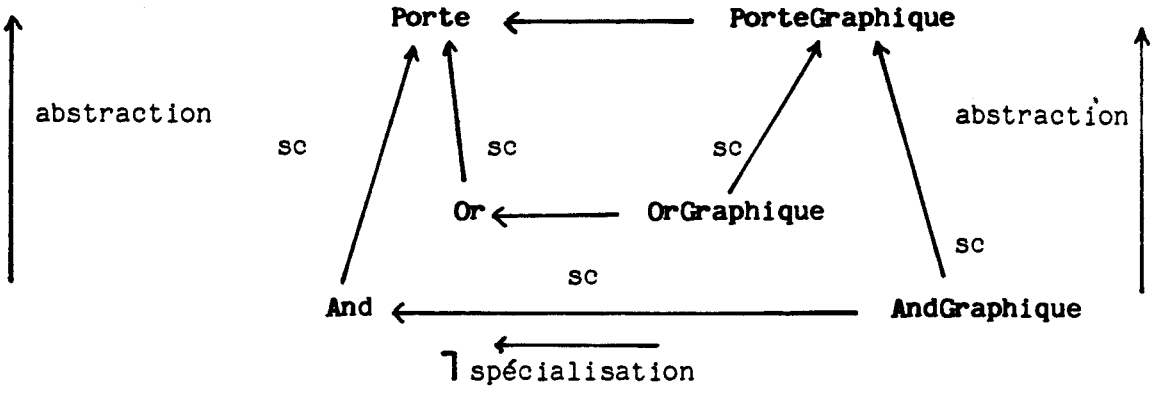


schéma équivalent au treillis précédent, tous les liens traduisant la relation "sur-classe". Il est clair qu'en héritage simple cette seconde abstraction issue de la spécialisation ultérieure d'un point de vue graphique aurait été impossible. Notamment les caractéristiques communes aux classes AndGraphique et OrGraphique de ce point de vue auraient dû être dupliquées dans chacune d'elles, ce qui est une contrainte pratique peu acceptable. L'intérêt de l'héritage multiple est donc autant conceptuel que pratique. Il renforce la puissance de l'héritage en tant que mécanisme implicite garantissant l'abstraction, la spécialisation et, plus généralement, la classification traduites à travers une hiérarchie de classes.

III.3.5.5.- Sous-classe/héritage

L'héritage peut être utilisé de diverses façons dans la construction d'une hiérarchie de classes : (seule la première retiendra par la suite notre attention).
L'approche méthodologique de la construction d'une hiérarchie de classes repose sur le respect du principe d'inclusion que l'on peut énoncer comme suit :

> une classe B peut être sous-classe d'une classe A si l'ensemble de ses représentants est inclus dans celui de A.

En d'autres termes, tout représentant de B est aussi représentant de A , le lien "sur-classe" traduit le lien sémantique IS-A.

Une propriété liée à ce principe a été énoncée au début du paragraphe III.3.5.2 et est caractéristique du polymorphisme d'inclusion propre à la notion de sous-classe (ou de sous-type). Rappelons cette propriété qui est énoncée différemment selon les auteurs :

> "Subtyping is an instance of inclusion polymorphism. The idea of a type being a subtype of another type is useful not only for subranges of ordered types such as integers, but also for more complex structures such as a type representing Toyotas, which is a subtype of a more general type such as Vehicles. Every object of a subtype can be used in a supertype context, in the sense that every Toyota is a vehicle and can be applied on by all operations that are applicable to vehicles".
>
> [CARDELLI-WEGNER.85]

> "An object-oriented language often provides a subtyping mechanism in its type system. One type may be a subtype of another, with the implication that if B is a subtype of A, an object of type B may be used wherever an object of type A may be used. In other words, objects of type B are also of type A".
>
> [HALBERT-O'BRIEN.87]

Ce principe est à la base de l'utilisation de la hiérarchie de classes et implicitement de l'héritage pour la classification, l'abstraction et la spécialisation dans un contexte de représentation des connaissances ou plus généralement de méthodologie orientée objet. Dans cette approche, que nous qualifions de conceptuelle, et qui nous concerne directement, la notion de sous-classe, selon le principe d'inclusion, précède l'héritage qui est considéré comme un mécanisme d'inférence sous-jacent et de second plan.

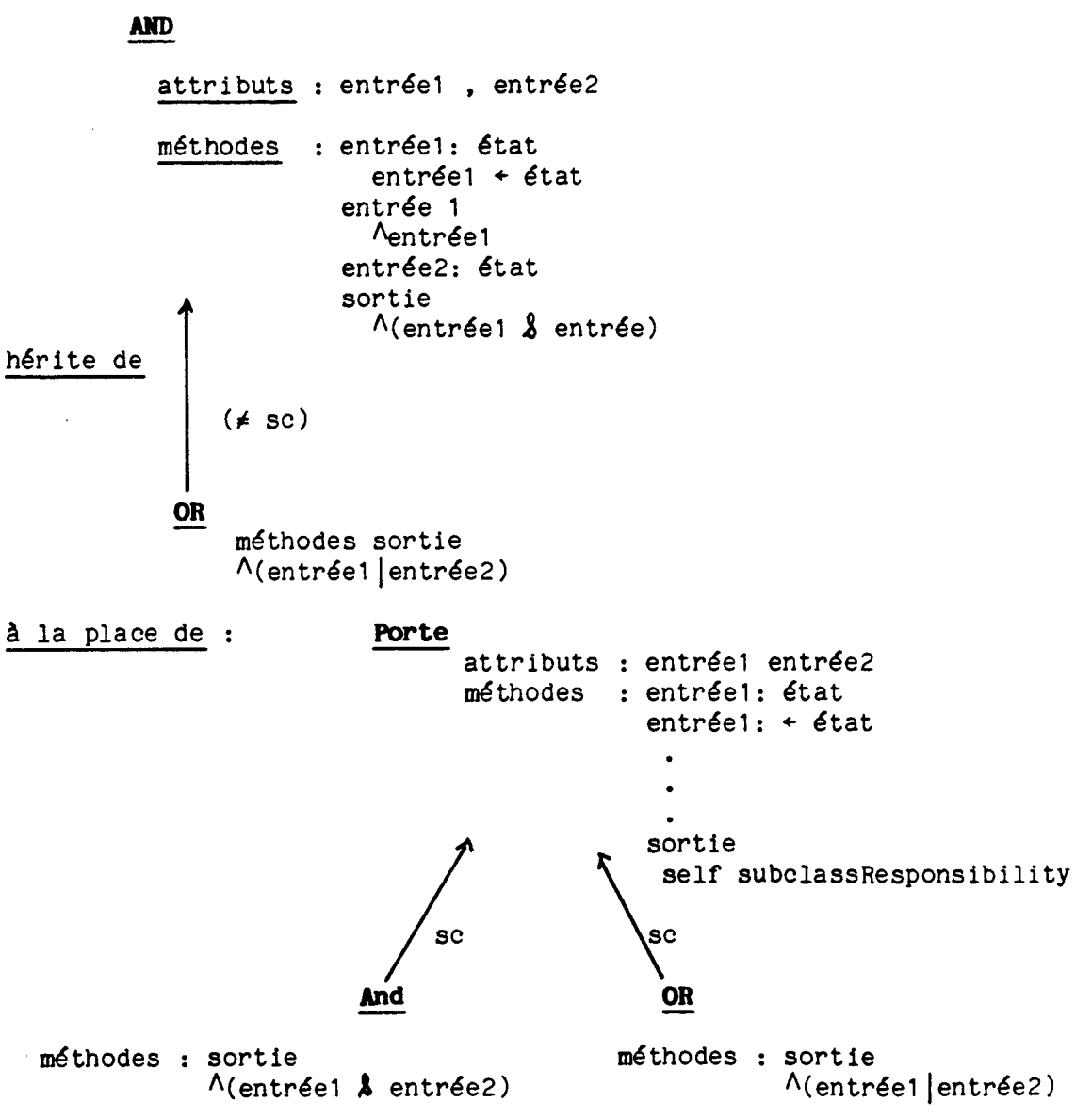
A l'opposé, l'héritage peut être considéré en tant que tel, c'est-à-dire comme un simple mécanisme de partage ou de réutilisation de caractéristiques détaché de la notion de sous-classe telle que précédemment, c'est l'héritage sans sous-classement (inheritance without subclassing). La nécessité de séparer l'héritage de la notion de sous-classe est un résultat de travaux dans le contexte du génie logiciel comme [SNYDER.86] [HORN.87] ou du parallélisme comme [AMERICA.87]. Cette approche de l'héritage que nous qualifions d'utile ne nous concerne pas directement car elle réclame des outils qui s'éloignent de notre modèle conceptuel de base, comme nous allons le voir. L'idée centrale est la suivante :

hériter $\not\Leftrightarrow$ être sous-classe de.

où "être sous-classe de" doit être interprété comme la relation d'inclusion, interprétation sur laquelle repose l'approche conceptuelle de l'héritage. Par exemple, pour les portes logiques, il est assez immédiat de constater que la classe Or hérite de la classe And sans en être sous-classe. Une telle approche nécessite donc un mécanisme d'héritage détaché de la notion de sous-classe qui permet à une classe d'hériter d'une autre classe sans en être sous-classe. Or, dans notre modèle de langages, le seul moyen de spécifier un quelconque héritage est de déclarer une classe sous-classe d'une autre. L'utilisation du

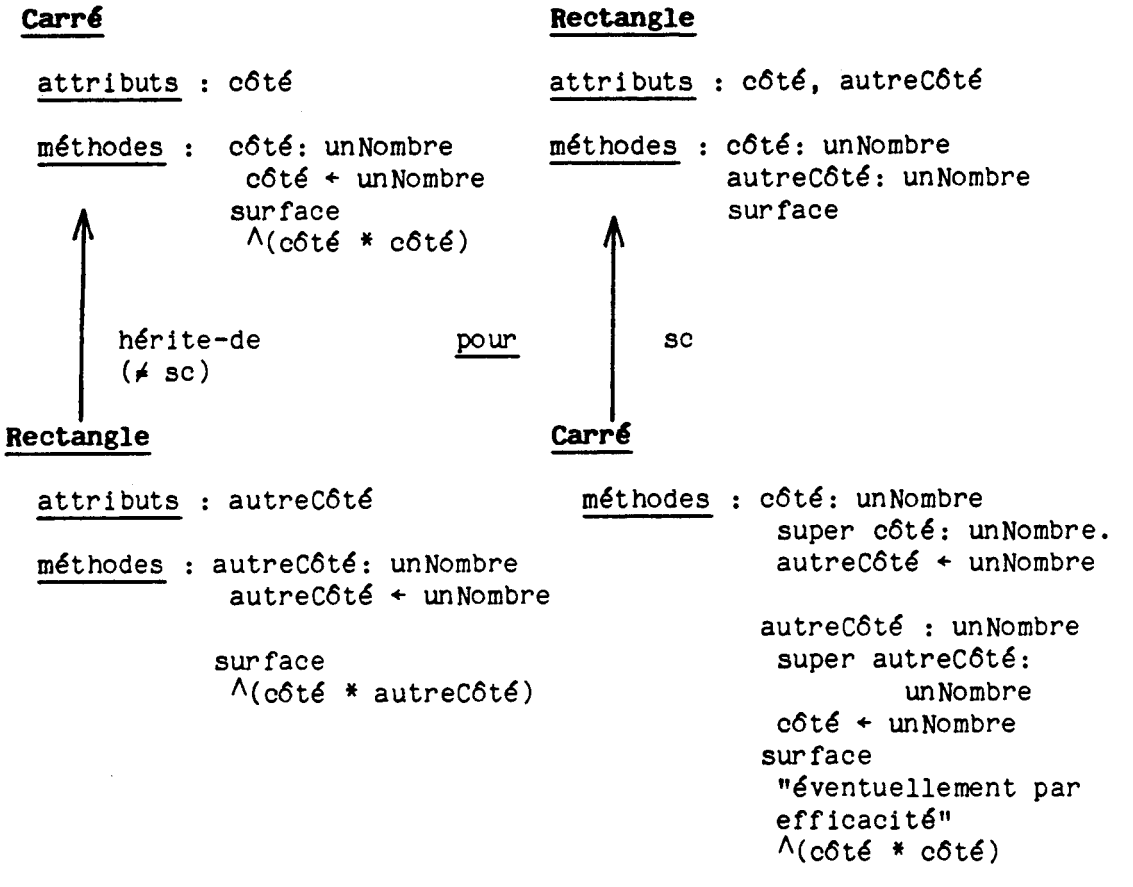
lien sur-classe pourl'héritage en dépit de la notion de sous-classe conduit à rejeter le principe d'inclusion, ce qui est inacceptable dans notre approche. Voyons ceci à travers plusieurs cas d'utilisation de l'héritage sans la notion de sous-classe.

L'héritage pour la différenciation est une alternative à la définition d'une classe abstraite par abstraction de classes semblables dans leur description. C'est le cas des classes And et Or qui partagent l'essentiel des caractéristiques qu'elles définissent, notamment leur protocole entier, mais différent par leurs fonction d'évaluation de la sortie. Dès lors, on peut vouloir exprimer que Or hérite de And et redéfinit par masquage la méthode d'évaluation de la sortie, soit "sortie". Définir Or sous-classe de And est cependant incorrect conceptuellement et méthodologiquement en respect du principe d'inclusion :



Le terme différenciation ("variance" [HALBERT-O'BRIEN.85]) vient du rapprochement entre ce cas et le modèle prototype/délégation (Cf. II.4) où un And serait un prototype et un Or serait défini comme extension du and par redéfinition de sa seule différence qui est la méthode sortie.

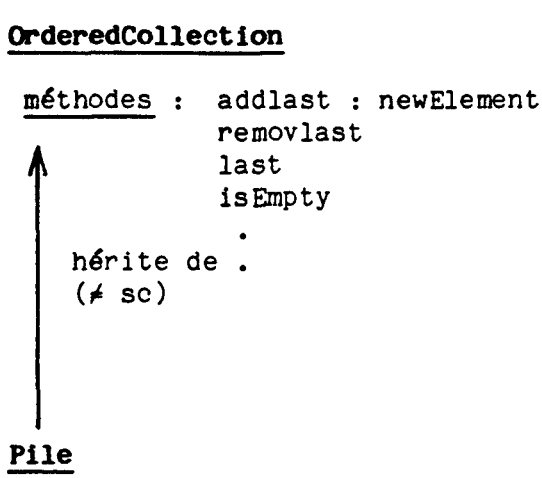
L'héritage pour la généralisation concerne les cas où une classe peut hériter d'une autre alors qu'elle est plus générale que celle-ci et donc ne peut en être sous-classe. Par exemple :



L'héritage pour l'implémentation est certainement le cas le plus intéressant dans cette approche. Il s'agit de classes qui réutilisent certaines des fonctionnalités d'autres classes pour implémenter les leurs. Par exemple, une Pile LIFO peut être implémentée par une OrderedCollection en SMALLTALK avec les correspondances respectives [GOLDBERG-ROBSON.83].

Pile	**OrderedCollection**
push : newElement	addlast : newElement
pop	removelast
top	last
empty	isEmpty

Dans l'approche "utile" de l'héritage, on pourrait considérer que la classe Pile hérite de OrderedCollection de la façon suivante :



Une fois de plus Pile n'est pas sous-classe de OrderedCollection selon le principe d'inclusion, on ne peut utiliser une pile LIFO partout où une OrderedCollection est réclamée puisque toutes les fonctionnalités de OrderedCollection ne sont pas disponibles sur une pile LIFO, par exemple addFirst:, removeFirst, add: before:, add: after:... . Cet exemple est similaire à celui des classes Stack (accessible à une extrêmité) et Deque (accessible en queue et tête) de [SNYDER.86b].

Ce cas montre particulièrement bien la nécessité d'outils particuliers pour l'approche utile de l'héritage, notamment la possibilité d'exclure des caractéristiques. En effet, les techniques d'héritage que nous avons toujours considérées ne conviennent pas ici parce qu'elles sont exhaustives : une sous-classe hérite toujours de toutes les caractéristiques de ses sur-classes (*1).

(*1) moyennant les redéfinitions, les stratégies de gestion de conflits.

Dans l'exemple, même en utilisant le lien sur-classe uniquement pour l'héritage, Pile ne peut exclure les fonctionnalités héritées en trop de Ordered Collection, addFirst, removeFirst,... Il y aurait toujours la possibilité de redéfinir toutes ces méthodes par masquage dans Pile pour qu'elles retournent un message d'erreur (self errror: 'méthode non héritée') ce qui est peu satisfaisant.

> "If inheritance is viewed as an technique implementation then excluding operation is both reasonable and useful".
> [SNYDER.86b]

L'approche utile de l'héritage nécessite donc des techniques d'héritage sélectif comme en CommonObjects [SNYDER.86a]. Dans ce langage, l'héritage est complétement séparé de la notion de sous-classe et est spécifié explicitement par la méthode ":inherit-from" paramétrée par les classes desquelles on souhaite hériter et la liste des caractéristiques à hériter. Dans une telle approche la notion de classe peut elle-même n'avoir aucune interprétation logique dans le sens où elle ne décrirait aucun type d'objet particulier mais simplement un paquet de fonctionnalités héritables. Une telle interprétation de la notion de classe est d'ailleurs présente dans la notion de "**mixin**" [STEFIK-BOBROW.85] [HENDLER.86] [LANG-PEARLMUTTER.86] en héritage multiple :

> "Many object-oriented computing languages allow the user to create multiple-inheritance classe hierarchies. One use of such inheritance schemes is the packaging of sets of operation in classes that can be mixed into other, usually larger, classes. These mixins allow creation of new subclasses which contain added functionnality".
> [HENDLER.86]

De telles classes n'ont de rôle que vis-à-vis de l'héritage pour ajouter des fonctionnalités à d'autres classes, les instancier n'aurait aucun sens (ce sont des classes abstraites).

Ces classes sont souvent des utilitaires et résultent d'une factorisation à l'extrême de caractéristiques partagées par plusieurs classes qui n'ont pas de lien conceptuel entre elles. Par exemple, en LOOPS, les mixins NamedObject et GlobalNamedObject définissent les caractéristiques des objets nommés, DatedObject les caractéristiques des objets datés (date de création, créateur). Avec des techniques d'héritage sélectif, toute classe est potentiellement une mixin puisqu'il suffit de spécifier les caractéristiques à hériter.

Une autre technique d'héritage sélectif est exposée dans [FOX.79]. La solution proposée est d'associer au lien d'héritage un label, "inheritance concept", pour spécifier les caractéristiques qui doivent être héritées (label "pass"), exclues ("exclude"), ajoutées ("add"), modifiées ("substitute précisé par d'autres sous-labels tels que "restrict", "generalize" (!), "refine"). Nous n'insisterons pas davantage sur une telle approche qui a l'avantage de laisser une grande liberté mais s'éloigne du cadre méthodologique assez strict préconisé par l'héritage exhaustif que nous respectons.

Nous terminerons cette brève analyse de l'héritage en donnant la solution standard du problème de la pile dans notre contexte et qui introduit le paragraphe suivant relatif à **l'agrégation**. En effet, l'agrégation pour l'implémentation se substitue avantageusement à l'héritage pour l'implémentation dans de tels problèmes. Dans le cas des piles, l'idée est d'exprimer que tout objet pile référence par un attribut "OrderedCollection" une instance de OrderedCollection (*1) pour s'implémenter, la traduction est alors immédiate :

```
Object subclass: #Stack
        instanceVariableNames: 'orderedCollection '

Stack comment:
'Pile implementee par une OrderedCollection encapsulee.'

Stack methodsFor: 'accessing'

pop
        ^orderedCollection removeLast

push: newElement
        orderedCollection addLast: newElement

top
        ^orderedCollection last


Stack methodsFor: 'testing'

isEmpty
        ^orderedCollection isEmpty


Stack methodsFor: 'at creation'

construct
        " Cree son instance locale de OrderedCollection"

        orderedCollection <- OrderedCollection new

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "

Stack class

Stack class comment:
'Metaclasse de Stack'

Stack class methodsFor: 'instanciation'

new
        ^super new construct
```

(*1) créée à l'instanciation de Stack

L'encapsulation de son attribut OrderedCollection garantit l'utilisation strictement locale de l'objet référencé et donc implicitement la protection des sélecteurs qui ne font pas partie de son propre protocole. La seule propriété d'encapsulation se suffit donc ici pour assurer une simulation de l'héritage sélectif (*1) ou plutôt une réutilisation sélective de fonctionnalités d'autres objets. Il s'agit ici d'un intérêt très pratique de l'agrégation pour l'implémentation. Dans le paragraphe suivant nous nous concentrerons plus sur son intérêt pour la représentation d'une hiérarchie des parties.

(*1) On parle même parfois d' "héritage horizontal" comme on le verra par la suite, inférence qui est assez proche de la délégation.

III.4.- HIERARCHIE STRUCTURELLE : L'AGREGATION

III.4.1.- Fondements

> "A powerful method for dealing with complexity is to describe a system hierarchically".
>
> [BORNING.79] (*1)

La description hiérarchique d'un système complexe, au sens large, considère que sa structure est décomposable en sous-structures, elles-mêmes décomposables en sous-structures,... et récursivement jusqu'à obtenir des structures élémentaires, "atomiques" non décomposables (s'il en est !).

Cette méthode de décomposition aboutit à ce que nous appellerons une hiérarchie structurelle. Des exemples de cette décomposition se trouvent facilement dans la majorité des domaines tels que la mécanique, l'électronique, l'architecture, l'informatique, les sciences naturelles ou encore l'anatomie. Concernant les premiers domaines, elle est à la base des méthodes de conception hiérarchique reprises par les systèmes de C.A.O. et les Systèmes Experts (voir par exemple [NIESSEN.86], [BEGG.84]). En informatique, ce principe est apparu dès les méthodes de programmation structurée bien connues sur lesquelles nous n'insisterons pas :

> "The creative activity of programming... is considered as a sequence of design decisions concerning the decomposition of tasks into substasks and of data into data structures".
>
> [WIRTH.71]

En représentation des connaissances, la hiérarchie structurelle s'est rapidement révélée à travers le lien IS-PART (is-composed-of/is-part-of) des réseaux sémantiques [RICH.83] ou par la notion de subframe de MINSKY [MINSKY.75].

> "IS-PART relationships. The relationships between objects that are made up of a set of components, each of which is made up of a set of components and so forth".
>
> [RICH.83]

(*1) Notons dès maintenant que la hiérarchie de classes a également cette propriété de maîtriser la complexité grâce à l'abstraction et la programmation incrémentale. BORNING énonce d'ailleurs ce principe en introduction aux deux types de hiérarchies traités en thinglab.

- une table composée de 5 parties : le dessus et 4 pieds.

- un immeuble composé d'étages, chaque étage composé d'appartements, chaque appartement étant composé de pièces,...

- une voiture composée de la carosserie, des quatre roues et du moteur, la carosserie composée d'un capot, des 4 portes...

- un micro-ordinateur composé d'une U.C., d'un clavier, d'un écran, l'U.C. composée d'un µp, d'une mémoire et des interfaces d'entrées/sorties...

Dès maintenant, la hiérarchie structurelle peut être définie comme suit :

> elle relie une entité qui représente un tout aux entités représentant ses parties. Elle est bâtie sur le lien sémantique IS-PART.

C'est pourquoi elle est souvent appelée "whole-part hierarchy" dans la littérature anglaise. Cette définition permet de la placer par rapport à la hiérarchie conceptuelle qui, rappelons-le, repose sur le lien IS-A (*) :

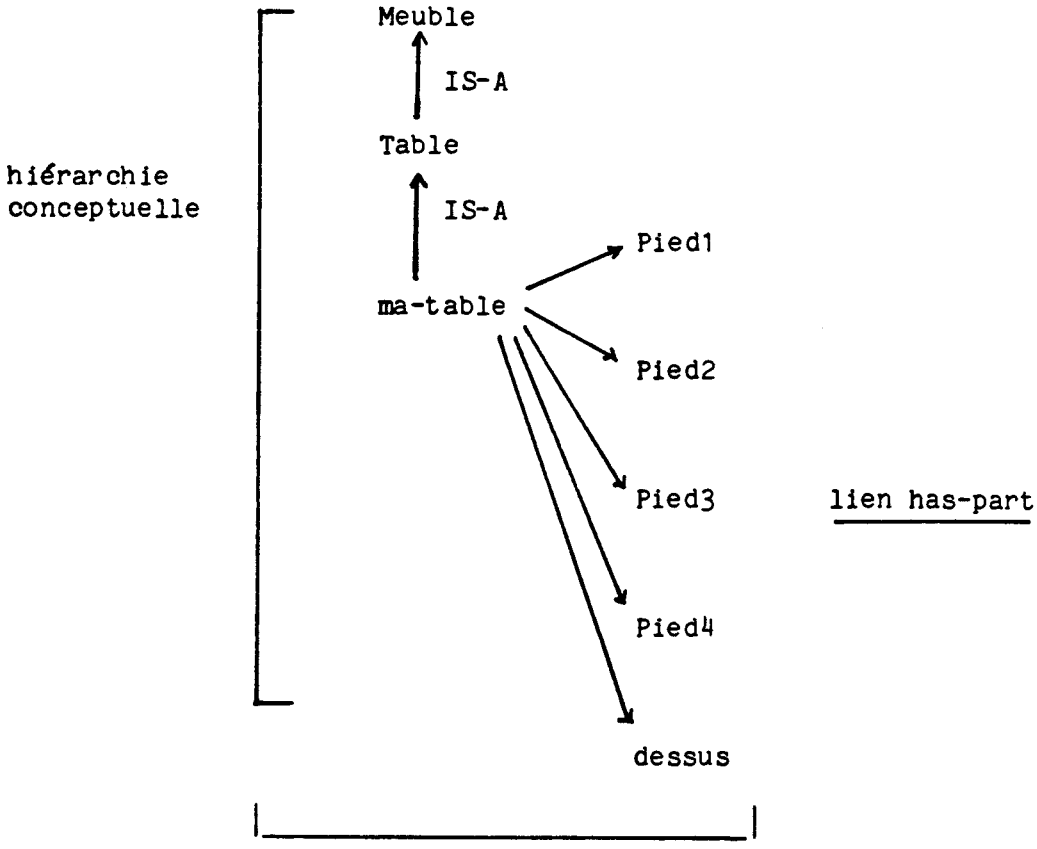
hiérarchie conceptuelle ≡ lien IS-A

hiérarchie structurelle ≡ lien IS-PART.

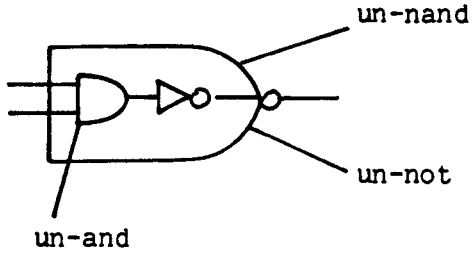
Mettons en évidence ces deux hiérarchies sur deux petits exemples représentés informellement sous la forme de réseaux sémantiques où nous figurerons le lien has-part associé à IS-PART, en tant que son inverse et qui traduit la décomposition. Le premier exemple reprend celui de la table :

(*) Les qualificatifs structurel et conceptuel nous paraissent assez évocateurs. Ils ne sont cependant pas standards. NYGAARD associe même concept et structure et parle de substance à la place de notre terme structure :

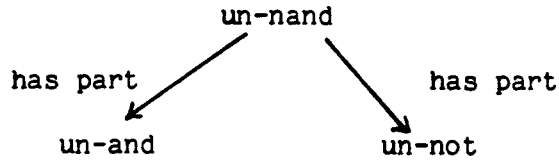
> "Substance hierarchies are organized through nested objetcs (objects within objects")
>
> Structure (concept) hierarchies are organized through patterns/ subpatterns (classes/subclasses)"
>
> [NYGAARD.86]



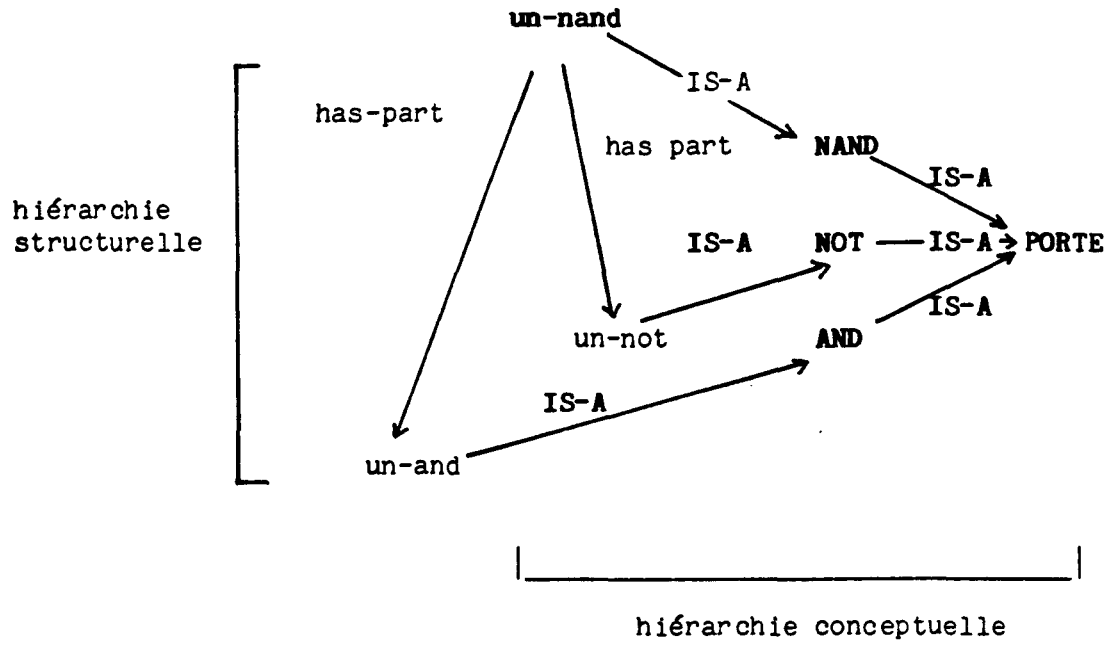
Le second exemple concerne l'implémentation d'une porte NAND selon le schéma :



dont la hiérarchie structurelle correspondante est :



que nous complétons par la hiérarchie conceptuelle bien connue maintenant, ce qui conduit au réseau sémantique suivant :



Le lien entre hiérarchie structurelle et agrégation vient du fait qu'à chaque niveau de cette hiérarchie, une entité est un agrégat (aggregate [MINSKY.75]), un assemblage des entités représentant ses parties, ses composants. La décomposition résulte de l'application récursive de l'agrégation pour chaque entité et mène à une hiérarchie structurée. L'agrégation est donc directement liée au lien IS-PART.

> "Aggregations are also called "a-part-of" relationships and are used to show groupings of objects".
>
> [LOOMIS and al.87]

Cette introduction nous mène tout naturellement à la représentation orientée objet. Dans une telle représentation, rappelons (si besoin était !) que toute entité est modélisée par un objet. Il suffit alors de substituer le terme entité par objet dans la définition précédente de la hiérarchie structurelle pour placer le problème dans notre contexte. Le but de la présentation qui va suivre est avant tout de placer la hiérarchie structurelle par rapport à la hiérarchie conceptuelle et de montrer que cette première reste assez peu considérée dans le paradigme orienté objet contrairement à cette dernière.

Nous allons voir, en effet, que l'agrégation est traitée de façon assez classique par les outils orientés objet et que le **lien IS-PART n'a pas de statut particulier, contrairement au lien IS-A.** Le paragraphe III.4.2. introduit le traitement de base de l'agrégation. Dans le paragraphe III.4.3. sont abordés les problèmes propres à la gestion d'une hiérarchie des parties qui sont peu ou pas pris en compte par la majorité des outils orientés objet. Enfin, le paragraphe III.4.4. montrera l'intérêt de l'agrégation pour définir la notion de frame à partir de celle d'objet.

III.4.2.- Traitement de base de l'agrégation.

Dans une représentation orientée objet, le problème se pose donc en ces termes :

> "In very many situations, we consider objects to be constructed from parts. The parts in their terms may consist of smaller simpler parts. And so forth".
>
> [BLAKE-COOK.87]

où chaque partie est évidemment elle-même un objet. De tels objets sont qualifiés de composites ou composés ("composite objects" ou "compound objects" [STEFIK-BOBROW.85] , [BORNING.79]), ils sont des agrégats, des assemblages de "sous-objets" ("sub objects"), leurs composants qu'ils référencent par leurs attributs selon la représentation standard [BOBROW.86], [MADSEN.86]. (*1)

> "The attributes may describe components that are a fixed part of the surrounding objects, or components which are references to objects".
>
> [MADSEN.86]

Ainsi, dans l'exemple précédent des nand, chaque nand va référencer ses composants, and et not par deux attributs, nommons-les "and" et "not". Chacun de ces attributs contiendra une référence vers une instance respectivement des classes AND et NOT.

Ceci conduit à la définition d'une classe NAND qui va caractériser ces objets composés (*2).

NAND

attributs and, not

(*1) La notion d'agrégation est générale en programmation [BISHOP.86] et se retrouve notamment dans les structures de données à travers les records PASCAL, ADA,... composés de sous-records... Ce rapprochement est évident, en nous rappelant que les attributs représentent les structures de données locales à l'objet.

(*2) Quelques précisions concernant cet exemple. Pour ne pas alourdir la présentation, nous considérerons la classe Nand indépendamment de toute abstraction par factorisation avec les classes And, Or, Not... que nous supposerons existantes. D'autre part, le modèle de représentation des portes ne nécessite pas l'introduction des broches, connections, qui seraient indispensables pour traduire des circuits complexes (voir par exemple [SUSSMAN-STEELE.80]).

Concernant son comportement, considérons qu'un nand doit répondre aux spécifications externes, maintenant classiques, qui sont

entrée1: , entrée , entrée2: , entrée2 , sortie.

Les méthodes associées vont traduire le comportement d'un nand en tant qu'assemblage d'un and et d'un not , en SMALLTALK :

NAND

variables d'instance : and , not

méthodes d'instance :

```
entrée1: état
   and entrée1: état

entrée2: état
   and entrée2: état

entrée1
   ^and entrée1

entrée2
   ^and entrée2

sortie
   not entrée: and sortie.
               "simulation de la connection"
   ^not sortie. (*1)
```

L'instanciation des composants and et not (*2) se fait traditionnellement à l'instanciation de l'objet composé (selon le schéma ^super new init (Cf. II.3.3)), ce qui mène à la version finale :

(*1) qui peut être condensé par cascade en

^not entrée: and sortie; sortie.

(*2) Ici aussi nous adopterons une approche de base, telle qu'en SMALLTALK où les attributs ne sont pas typés.

NAND

variables d'instance and, not

méthodes d'instances

.
.
.

"comme au dessus plus le message construct
de construction de l'agrégation"
construct
  and ← And new.
  not ← Not new

méthode de classe

new
^super new construct

Voici une version complète de Nand en SMALLTALK :

```
Object subclass: #Nand
        instanceVariableNames: 'and not '

Nand comment:
'Nand par decomposition en and et not'

Nand methodsFor: 'at creation'

construct
        " Creation des sous objets, lors de l'instanciation d'un nand (cf new)"

        and <- And new.
        not <- Not new

Nand methodsFor: 'reading'

entree1
        "lecture propagee de l'entree1 du and"

        ^and entree1

entree2
        "lecture propagee de l'entree2 du and"

        ^and entree2

sortie
        " evaluation de la sortie du not par propagation de la sortie
          du and sur l'entree, et retour"

        not entree: and sortie.
        ^not sortie


Nand methodsFor: 'writing'

entree1: etat
        "propagation sur l'entree1 du and"

        and entree1: etat

entree2: etat
        "propagation sur l'entree2 du and"

        and entree2: etat
```

```
Nand class

Nand class comment:
'Metaclasse de Nand'

Nand class methodsFor: 'instanciation'

new
        ^super new construct
```

L'agrégation a été introduite à travers des exemples de décomposition "naturelle" d'entités qui permettent facilement de s'en forger une intuition. Cette technique est cependant très générale et pourrait être définie simplement comme le regroupement d'entités. Dans bien des cas elle traduit une implémentation structurée d'un objet (le tout) en termes de sous-objets (ses parties) correspondant à des aspects particuliers. Par exemple, pour représenter une personne, toutes ses caractéristiques qui concernent son état-civil peuvent être regroupées dans un sous-objet référencé par l'attribut de même nom. Nous verrons (III.4.4) que cette approche permet de définir la notion de frame à partir de celle d'objet.

La phrase de MADSEN résume donc l'approche standard de l'agrégation (*) dont notre exemple du Nand est une application. C'est ce que nous appellerons **le traitement de base de l'agrégation** qui repose uniquement sur les concepts de base, notamment celui d'attribut pour la référence d'objet. La propriété d'encapsulation des attributs permet en particulier de définir des agrégations d'objets strictement internes à l'objet englobant. Cette propriété est essentielle si l'on considère que le tout est maître des parties et qu'ainsi il garantit sa cohérence interne en se réservant l'accès à ses sous-objets, et donc la gestion de leur comportement. C'est le cas quand il s'agit de masquer une implémentation particulière dont le monde des objets extérieurs ne doit pas avoir connaissance.

(*) dans les langages orientés objet et plus généralement dans les langages offrant la structure de record.

Par exemple, concernant la classe Personne, l'utilisateur extérieur n'a pas à savoir que telles caractéristiques ont été regroupées dans tel sous-objet interne (état-civil). Il peut en être de même pour l'exemple des nand si, par souci d'abstraction, son implémentation par décomposition en and et not ne doit pas être connue. Dans ce cas, un tel nand vu de l'extérieur a la même abstraction que toute autre représentation.

Montrons ceci, selon l'exposé du paragraphe III.3.5.2., en nommant Nandd (d pour décomposition) la version précédente et Nande (e pour équation logique) la version suivante :

```
Nande

  attributs :  entrée1 , entrée2

  méthodes :  entrée1: état
                entrée1 ← état

              entrée1
                ^entrée1

              entrée2: état
              entrée2
              sortie
               "équation logique"
               ^(entrée1 et entrée2) not
```

Nandd et Nande peuvent être facilement abstraites par une surclasse abstraite Nandab qui spécifie le protocole des nand (*)

(*) qui est en fait sous-classe de Porte (!). On voit ici le lien avec l'exemple des piles du III.3.5.2. où la classe Pile joue le même rôle que Nand et où la classe d'implémentation des Piles par agrégation d'une OrderedCollection telle que III.3.5.5. joue un rôle semblable à Nandd et serait déclarée sous-classe de Pile.

```
Object subclass: #Nandab
        instanceVariableNames: ''

Nandab comment:
'Classe abstraite des nand'

Nandab methodsFor: 'reading'

entree1
        self subclassResponsibility

entree2
        self subclassResponsibility

sortie
        self subclassResponsibility


Nandab methodsFor: 'writing'

entree1: etat
        self subclassResponsibility

entree2: etat
        self subclassResponsibility
```

```
Nandab subclass: #Nandd
        instanceVariableNames: 'and not '

Nandd comment:
'Implementation des nand par decomposition'

Nandd methodsFor: 'reading'

entree1
        ^and entree1

entree2
        ^and entree2

sortie
        not entree: and sortie.
        ^not sortie


Nandd methodsFor: 'at creation'

construct
        and <- And new.
        not <- Not new

Nandd methodsFor: 'writing'

entree1: etat
        and entree1: etat

entree2: etat
        and entree2: etat

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "

Nandd class

Nandd class comment:
'Metaclasse de Nandd'

Nandd class methodsFor: 'instanciation'

new
        ^super new construct
```

```
Nandab subclass: #Nande
        instanceVariableNames: 'entree1 entree2 '

Nande comment:
'Par equation.'

Nande methodsFor: 'initializing'

init
        entree1 <- false.
        entree2 <- false

Nande methodsFor: 'writing'

entree1: etat
        entree1 <- etat

entree2: etat
        entree2 <- etat

Nande methodsFor: 'reading'

entree1
        ^entree1

entree2
        ^entree2

sortie
        ^(entree1 & entree2) not

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "

Nande class

Nande class comment:
'Metaclasse de Nande'

Nande class methodsFor: 'instanciation'

new
        ^super new init
```

Les instances de Nandd et Nande répondent au même protocole spécifié par Nandab.

A l'inverse, le tout peut autoriser l'accès externe à ses composants, notamment dans le cas où il faut laisser visible la hiérarchie des parties. C'est l'objet du paragraphe suivant où nous allons constater que la solution n'est alors plus immédiate et constitue tout un problème en soi.

III.4.3.- Agrégation et hiérarchie des parties.

Nous avons vu, au paragraphe III.4.2. que l'agrégation était fortement liée à la hiérarchie structurelle, celle-ci étant l'application récursive de la première au niveau de chaque objet considéré comme un tout. Bien que le traitement de base de l'agrégation pose le principe de représentation d'une telle hiérarchie, nous allons montrer quelques problèmes qui ne seront pas détaillés ici, notre but n'étant pas de traiter de la gestion de la hiérarchie des parties qui est tout un sujet en soi. Ces problèmes sont largement discutés dans [BLAKE-COOK.87] et nous les exposerons succintement à travers notre exemple du Nand dans le paragraphe suivant. Puis nous aborderons les problèmes d'inférence propres à la hiérarchie des parties. Ces deux paragraphes vont montrer qu'actuellement la hiérarchie structurelle est peu ou pas gérée par les outils orientés objet, ce qui est une critique couramment faite à ceux-ci.

III.4.3.1. Encapsulation et hiérarchie des parties

La traitement de base de l'agrégation par attribut et référence d'objet laisse toute liberté au tout de gérer la visibilité de ses parties d'après la propriété d'encapsulation par défaut des attributs. Nous avons vu un exemple de masquage des parties par encapsulation. A l'inverse, le tout peut autoriser l'accès à ses composants et c'est le cas en particulier quand il s'agit de laisser apparente la hiérarchie structurelle. En effet, la stricte encapsulation des parties a l'inconvénient d'absorber la hiérarchie qui est alors complétement cachée par le tout :

> "A part belongs to the local state of the whole and the interface is mediated by the owner. If the requirement is strictly interpreted, the existence of the parts should become invisible to the users of the whole... The net result is that the part hierarchy is replaced by a single monolithic whole as far as the external world is concerned".
>
> [BLAKE-COOK.87]

On peut constater aisément ce fait sur la version précédente de Nandd. En simplifiant les méthodes de conception de circuits logiques (*)

(*) Nous n'insisterons pas sur les méthodes de conception hiérarchique ou électronique, le but nétant pas d'en faire un traité. La décomposition qui est présentée est d'ailleurs loin d'être standard, voir [GOTO.86], [SAVARIA.86] pour les méthodes standard de conception VLSI par exemple.

cette version, ainsi que Nande, conviendrait tout à fait à une phase de conception symbolique où le Nand est considéré d'un point de vue simplement fonctionnel sans s'intéresser à son implémentation. Lors de l'étape suivante de la conception, appelée phase d'implémentation, supposons justement que le choix ait été fait de réaliser un Nand à l'aide d'un and et d'un not. A ce niveau, cette décomposition doit être accessible et par là la hiérarchie structurelle induite qui traduit le passage d'une phase à l'autre. Cette version d'un nand doit donc autoriser l'accès à ses composants pour le parcours de la hiérarchie, ce qui est facilement représenté par les sélecteurs "and" et "not" dont les méthodes associées ont pour but de renvoyer les composants de mêmes noms référencés par les attributs de mêmes noms. Ceci mène à une première solution simple définie par la classe Nandh (h comme hiérarchique) qui spécialise la classe Nandd pour permettre l'accès à la décomposition :

Nandh

sur-classe Nandd

méthode and
^and
not
^not

Ainsi, soit UnNandh une instance de Nandh (résultat de "Nandh new") les envois de messages "UnNandh and" et "UnNandh not" renvoient respectivement le and et le not qui peuvent être manipulés :

- UnNandh and entrée1: true , affectation de l'état de l'entrée 1 du And équivalent à UnNandh entrée1: true.

- UnNandh not sortie, lecture de la sortie du not équivalent à UnNandh sortie.

Cette solution simple et opposée à la précédente pose évidemment le problème de maintenance de la cohérence du tout en autorisant un accès complétement libre aux parties. Ici, la cohérence du Nandh serait perdue par le simple envoi de message :

UnNandh not entrée: un_état

si cet état ne correspond pas à l'état de sortie du and , problème qui ne se posait pas dans la version précédente grâce à l'encapsulation. C'est là le centre du problème :

- la stricte encapsulation des parties facilite la maintenance de la cohérence mais masque complétement la hiérarchie,

- la complète accessibilité des parties laisse apparente la hiérarchie mais complique la maintenance de la cohérence.

"When we want to model objects consisting of parts in SMALLTALK and many other object-oriented languages, we are confronted with a dilemna : either sacrifice the data encapsulation properties of the language or utterly flatten the whole-part hierarchy."
[BLAKE-COOK.87]

Une première solution est de descendre au niveau des sous-objets des informations concernant le tout ou même plus généralement de gérer la cohérence par un pointeur inverse d'une partie vers son tout. Pour certaines actions, le sous-objet pourra ainsi demander l'autorisation à son tout, autorisation conditionnée par sa cohérence propre. Par exemple, dans le prolongement de la version précédente, un not intervenant dans l'implémentation d'un Nandh devra avoir un comportement particulier, notamment pour l'affectation de son entrée et ne sera donc pas un simple Not mais un représentant de Not pour Nandh (*) spécialisation de Not comme suit :

```
Not subclass: #NotPourNandh
        instanceVariableNames: 'nand '

NotPourNandh comment:
'Not en temps que partie d''un nand. L''attribut nand contient le pointeur inver
se vers le tout (un Nandh1).'

NotPourNandh methodsFor: 'at creation'

pour: unNand
        " pointeur inverse vers le tout, (cf newPour:)"

        nand <- unNand


NotPourNandh methodsFor: 'writing'

entree: etat
        " si coherence du tout, i-e de son nand englobant"

        (nand coherent: etat)
                ifTrue: [super entree:etat]
                ifFalse: [self error: 'incoherence du tout']
```

(*) toute classe en SMALLTALK est globale, il n'est donc pas possible de définir une classe particulière de Not à l'intérieur de la classe Nandh comme ce serait possible dans les langages typés, voir par exemple [MADSEN.86] [KRISTENSEN and al.83]

```
NotPourNandh class

NotPourNandh class comment:
'Metaclasse de NotPourNandh.'

NotPourNandh class methodsFor: 'instanciation'

newPour: unNand
        ^self new pour: unNand
```

Notre nouvelle version de Nandh est alors :

```
Nandd subclass: #Nandh1
      instanceVariableNames: ''

Nandh1 comment:
'Hierarchie des parties visible. Coherence par pointeur inverse
 (cf NotPourNand).'

Nandh1 methodsFor: 'at creation'

construct
        and <- And new.
        not <- NotPourNandh newPour: self

Nandh1 methodsFor: 'coherence'

coherent: etat
        ^and sortie = etat

Nandh1 methodsFor: 'hierarchical paths'

and
        ^and

not
        ^not
```

Cette solution n'est cependant pas satisfaisante vis à vis des règles "idéales" de conception descendante selon lesquelles un tout fait appel à ses parties qui ont leur comportement propre indépendamment de lui et donc qui n'ont pas à y faire appel :

> "Ideally the whole knows the parts but the parts do not know the whole".
>
> [BLAKE-COOK.87]

Le problème est en fait que l'objet englobant doit laisser visibles ses parties tout en restant maître de gérer leur accès :

> "This is the dilemna. We would like the parts to remain visible, while at the same time access is mediated by the owner".
> [BLAKE-COOK.87]

Pour notre exemple, la solution serait de n'autoriser que l'accès aux sélecteurs de manipulation des entrées du and et de la sortie du not (*). La spécification de telles restrictions n'est en général pas possible, tout au plus peut-on simuler syntaxiquement la visibilité de certains sélecteurs des objets internes à travers le protocole du tout, comme suit :

```
Nandd subclass: #Nandh2
        instanceVariableNames: ''

Nandh2 comment:
'Simulation de l''acces a la hierarchie structurelle par selecteurs, les sous-ob
jets sont strictement encapsules.'

Nandh2 methodsFor: 'pathSelectors'

andEntree1
        ^and entree1

andEntree1: etat
        and entree1: etat

andEntree2
        ^and entree2

andEntree2: etat
        and entree2: etat

notSortie
        ^not sortie
```

(*) L'accès en lecture de la sortie du and et de l'entrée du not pourrait aussi être autorisé, ce qui correspondrait à placer un testeur de l'état interne du Nand.

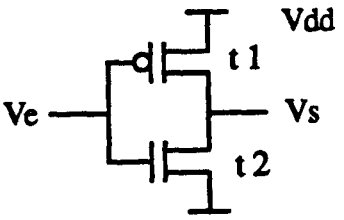
Les composants restent alors encapsulés et leur accès est simulé (*0). En appelant unNandh1 une instance de la version précédente de Nandh et unNandh2 une instance de cette dernière, nous aurons les correspondances suivantes :

unNandh1 and entrée1: unétat - unNandh2 andEntrée1: un_état

unNandh1 not sortie - unNandh2 notSortie

et cette fois-ci le problème de l'accès à l'entrée du not ne se pose plus puisque le message unNandh1 not entrée: un_état n'a pas de "pseudo" correspondance dans le protocole de unNandh2 (*1). Cette solution artificielle a cependant l'avantage de préserver l'encapsulation mais on imagine facilement la complexité croissante du protocole du tout avec le profondeur de la hiérarchie (*).

Par exemple, considérons l'implémentation d'un not par deux transistors MOS respectivement N et P selon le schéma :



L'accès au sélecteur sortie de T1 à travers le Nand puis le not entraînerait les définitions suivantes :

Nand	**Noth**	**TransistorP**
notT1Sortie	attributs t1 t2	attributs sortie...
^not t1Sortie	méthodes t1Sortie	méthodes sortie
	^t1Sortie	^sortie

où t1 et t2 référencent des instances de Transistor P et N respectivement.

Ces problèmes montrent la nécessité d'outils plus puissants que le traitement de base de l'agrégation pour gérer la hiérarchie des parties que très peu de langages offrent. Dans [BLAKE-COOK.87] est proposée une extension de SMALLTALK pour permettre l'envoi de message "aux parties" à travers (*2) le tout par la notion de "compound selector" (qui n'a rien à voir avec ceux présentés au III.3.3). Un

---

(*0) Nandh2 étant sous-classe de Nandd, and entrée1: état pourrait se définir par self entrée1: état (héritée de Nandd) et de la même façon pour les autres méthodes.

(*1) Un_Nandh2 notEntrée: un_état renverrait évidemment "message not understood". L'accès en lecture de la sortie du and et de l'entrée du not serait simulé de la même façon par :

```
andSortie
    ^and sortie
notEntrée
    ^not entrée
```

(*2) "the need to see structure of an object is the similar to the need for "grey-boxes" rather than "black-boxes" in engeneering".

sélecteur composé est formé d'un chemin d'accès à une partie (suite de noms de parties imbriquées séparés par '.') et d'un sélecteur de son protocole. La sémantique de l'envoi de message est étendue au "message forwarding" inspirée de la délégation (Cf. II.3.5).

Si le sélecteur composé fait partie du protocole d'un objet composite, la méthode est appliquée. Ainsi, le tout peu interdire l'accès à certains sélecteurs de ses parties. Par exemple :

```
Nandh

   sur-classe  Nandd

   méthode     not.entrée: un_état
                 self error: "inaccessible"
          "interdit l'affectation de l'état d'entrée
           du not".
```

Sinon le mécanisme du "message forwarding" implicite est lancé (*1). Montrons son principe sur l'exemple suivant, l'envoi du message "unNandh and.entrée1: unétat" provoque l'acheminement automatique (délégation) du message entrée:1 unétat au and de unNandh de manière équivalente à "un_Nandh and entrée1: un_état" sur notre version précédente de Nandh (*2). La notion de chemin a été introduite dès Thinglab [BORNING.79], qui résulte de travaux bien plus anciens à partir de SMALLTALK 76, pour faciliter le parcours d'une hiérarchie structurelle. Dans ce langage, les objets sont formés de parties et la cohérence est exprimée en termes de contraintes gérées automatiquement (*3). Enfin, citons Loops où la métaclasse Template décrit le comportement commun de toutes les classes d'objets composites. Une telle classe décrit les classes d'instanciation des sous-objets et leurs interconnections. En particulier la méthode d'instanciation de base est redéfinie pour instancier automatiquement les sous-objets à la création d'un objet composite (*4).

(*1) par trapping de l'erreur "message non understood"

(*2) la délégation implicite dans leur outil correspond à la délégation explicite qui apparaît dans la définition de la méthode :

```
and Entrée1: un_état
    and entrée1: un_état "délégation explicite"
```

(*3) Toute classe a une liste de "Parts descriptions", chacun d'elles spécifie les caractéristiques d'une partie par son nom (name), une liste de contraintes (constraints), sa classe d'instanciation (class) et un ensemble de fusions (merges). Les fusions permettent d'établir des contraintes de correspondances entre les parties du genre : 2 segments adjacents d'un triangle ont une extrêmité en commun.

(*4) On voit ici l'intérêt des métaclasses pour définir des classes particulières à partir des concepts de base.

En conclusion de ce paragraphe, disons que peu d'outils sont proposés pour gérer la hiérarchie des parties :

> "Part whole relations are not often catered for in object-oriented languages".
>
> [BLAKE-COOK.87]

Cette critique va même plus loin comme nous allons le voir dans ce qui suit.

III.4.3.2.- Inférence et hiérarchie des parties

Nous avons vu que la hiérarchie des parties posait des problèmes et que le traitement de base de l'agrégation pouvait paraître insuffisant.

Certains auteurs vont même jusqu'à dire que l'agrégation est en général ignorée par les langages orientés objet :

> "Aggregation is ignored by the popular object-oriented programming languages".
>
> [LOOMIS and al.87]

Cette critique vient principalement du fait qu'il y aurait une inférence propre à l'agrégation qui n'est pas gérée implicitement par ces langages comme peut l'être l'héritage (*1) . Nous n'entrerons pas dans les détails de ce débat mais exposerons succintement le fond du problème qui fait l'objet de travaux de recherche tels que [CHOURAQUI-DUGERDIL.86], [LOOMIS and al.87]. Le principe est de dire qu'une partie peut dans certains cas "hériter" de caractéristiques de son tout. Prenons un exemple cité dans ce dernier article où une voiture a des composants portières et capot, et est décrite par les attributs couleur, poids et fournisseur. On peut alors considérer que les attributs couleur et fournisseur soit propagés du tout aux parties contrairement au poids. Plus précisément, c'est l'attribut avec sa valeur qui est inféré à moins que cette valeur ne soit redéfinie en

---

(*1) "There are three types of relationships, each of which has its own distinct semantics : generalization, aggregation and association. It is this area of relationships that an Object Modeling Technique differs most from the conventional object-oriented paradigm of SMALLTALK-80, which directly supports only the relationship notions of generalization" [LOOMIS and al. 87].

propre dans une des parties :une portière peut avoir une couleur différente de la couleur globale de la voiture (*1). Leur modèle offre donc la possibilité d'établir un lien d'agrégation entre une partie et un tout sur lequel est défini un mécanisme de propagation de certains attributs déclarés propageables et qualifiés de transitifs par opposition aux attributs intransitifs qui ne peuvent être propagés. Notons que l'agrégation et plus généralement les relations sont dissociées ici de la notion d'attribut (*2).

Le même type d'approche est utilisé dans [CHOURAQUI-DUGERDIL.86] [DUGERDIL.85]. L'agrégation est représentée par le champ (attribut) PARTIE-DE auquel on peut associer des champs qui seront propagés du tout à la partie. Le mécanisme de propagation est appelé ici "héritage sélectif" par opposition à l' "héritage vertical" standard des LOO Ces auteurs prennent l'exemple du MUR, caractérisé par les champs FORME, HAUTEUR, LARGEUR, EPAISSEUR, FACE,... Le champ FACE contiendra les 2 faces d'un mur, instances de la classe FACE-TYPE1 et FACE-TYPE2. FACE-TYPE1 définit les champs COULEUR et PARTIE-DE auquel est associée la liste des champs héritables par une face en tant que partie d'un mur et qui sont (HAUTEUR, LARGEUR) mais pas FORME, EPAISSEUR, FACE... (!)

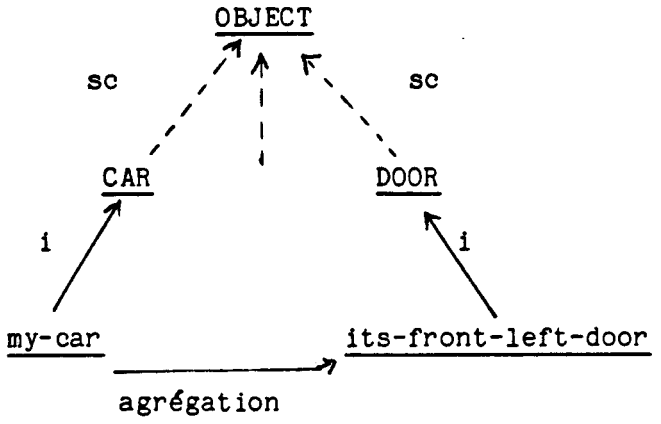
(*1) Quelle est alors la couleur de la voiture ? Notamment si toutes ses parties ont une couleur différente. D'autre part, si toutes les parties ont la même couleur, celle globale de la voiture doit être changée ou cette fois-ci c'est le tout qui "hérite" de la couleur de ses parties (!) Ceci pose le problème des liens inverses et de la gestion de la cohérence du tout en général.

(*2) Dans notre modèle de langages, les relations n'ont pas de statut particulier et sont représentées par des attributs qui ont pour valeur des références d'objets. Rien n'empêche cependant de considérer qu'une relation binaire entre deux objets A et B soit "un objet" (!) avec deux attributs référençant respectivement A et B et éventuellement d'autres attributs comme la raison de cette relation ou un label quelconque et ses méthodes. Voir par exemple la classe Association en SMALLTALK. D'autre part, tout étant objet en SMALLTALK, un objet peut toujours être considéré comme l'agrégation des objets référencés par ses variables d'instances, ce qui ne fait pas de différence entre l'attribut "couleur" et l'attribut "ma-portière-avant-gauche" puisque toutes deux référencent des objets. La question "to be or not to be an attribute" semble ne pas avoir de réponse immédiate.

(*3) On par le même d'héritage horizontal [FERBER.83] qui évoque bien les caractéristiques de ce type d'héritage tel que nous les résumerons par la suite.

Insistons sur les caractéristiques de cette inférence, ou "propagation" ou "héritage sélectif". Il s'agit d'une inférence entre objets (au sens large, notamment entre instances terminales) et non entre classes comme pour l'héritage.

C'est pourquoi elle est parfois appelée "héritage horizontal" [FERBER.83] ou "transversal" par opposition à l'héritage "vertical" classique sur la hiérarchie de classes. En effet, elle opère sur des liens entre objets instances de classes généralement non sous-classes l'une de l'autre (*1). Par exemple :



Il s'agit non seulement d'une inférence d'attribut mais aussi de valeurs d'attributs (valeurs d'instance) et plus précisément par défaut : la portière avant gauche de ma voiture a par défaut la couleur de celle-ci mais peut avoir une couleur différente en affectant une valeur différente à l'attribut couleur (*2)

(*1) Ce n'est pas toujours le cas et même à la limite ces classes peuvent être identiques. Par exemple un circuit (ou un bloc fonctionnel) est composé de circuits eux-mêmes composés de circuits... ce qui établit des liens d'agrégation entre instances d'une même classe, soit CIRCUIT. (de la même façon une liste composée de listes composées de listes...)

(*2) La signification de cette inférence n'est d'ailleurs pas très claire. Pour reprendre l'exemple des voitures, la propagation de la couleur signifie-t-elle que la portière a la couleur bleue parce que la voiture a la couleur bleue, ce qui est traduit par l'inférence de l'attribut et de sa valeur ou que la portière a une couleur parce que la voiture a une couleur. Dans ce cas, pourquoi l'attribut poids ne peut-il être propagé(sans sa valeur). En fait, seule la valeur devrait être propagée par défaut et relève de l'agrégation. Le partage du nom de l'attribut lui-même relève de l'héritage classique et serait traduit par exemple par une classe Objet-coloré sur-classe (du genre mixin) de CAR, DOOR, CAPOT qui spécifierait l'attribut couleur et de façon similaire pour Objet-pesant.

Cette inférence de valeurs entre instances n'est évidemment pas supportée par l'héritage entre classes. Insistons ici sur un point, certains langages comme LOOPS ou les langages de frames proposent un mécanisme de valeurs par défaut associées aux attributs au niveau de la classe. Pour une instance, la valeur d'un attribut, s'il n'est pas instancié, est alors recherchée dans la classe (*1). Cette notion est cependant différente de celle fournie par la propagation puisqu'il s'agit ici d'une valeur par défaut partagée par toutes les instances de la classe (générale au concept) alors que précédemment il s'agit d'une valeur par défaut des instances particulières d'une classe que sont les parties du tout, instance particulière d'une autre classe.

Enfin, cette inférence est sélective (*2) contrairement à l'héritage exhaustif sur la hiérarchie des classes.

Il semble donc, finalement, qu'il y ait une certaine inférence propre à l'agrégation et ainsi nécessité de la traiter à part entière. Notons que la propagation peut tout à fait être programmée par envoi de messages entre les parties et le tout, comme pour l'exemple des voitures donné ci-dessous, l'avantage des outils précédents étant, bien entendu, d'offrir un moyen déclaratif de la spécifier :

```
CAR

 sur-classe : OBJECT

 attributs :  color

 méthodes :  color: unecouleur
               color ← unecouleur

             color
               ^color

DOOR

 sur-classe : OBJECT

 attributs  : color, car

 méthodes :  color: unecouleur
               color ← unecouleur

             color
              (color isNil) ifTrue: [^car color]
                            ifFalse: [^color]
```

(*1) par le mécanisme d'héritage et donc de manière ascendante, ce qui permet à une sous-classe de redéfinir une valeur par défaut par masquage.

(*2) et même souvent très sélective.

Une fois de plus (Cf. III.4.3.1.) cette solution ressemble fortement à la délégation, si l'on demande à une instance de DOOR sa couleur et qu'elle n'en a pas de particulière, elle redirige le message vers son tout ((color isNil) ifTrue: [^car color]).

Ce rapprochement entre la délégation et l'agrégation est naturelle puisque la relation IS-PART s'établit entre objets et que les langages bâtis sur le modèle prototype/délégation ne considèrent qu'un seul type d'entité, l'objet. On rencontrera encore souvent ce parallèle dans ce chapitre sur l'agrégation.

Terminons ce bref exposé sur les problèmes liés au traitement de la hiérarchie des parties en constatant que ce n'est pas le point fort des langages orientés objet. En effet, contrairement à la spécification d'une hiérarchie conceptuelle dont l'inférence propre est gérée implicitement par héritage, l'agrégation n'est pas traitée comme une notion à part mais est une application particulière des concepts de base et ses aspects particuliers, tels que l'accès aux parties, la gestion de la cohérence et son inférence propre, doivent en général être programmés explicitement.

III.4.4.- Le pont vers les frames

La notion de frame a été placée au § II.1 dans le courant de pensée orienté objet comme une des révélations du concept général d'objet, du moins rétrospectivement. Nous avons vu que, conceptuellement, ces deux notions procédaient d'une même vision des "choses" du monde réel. En effet, il n'y a pas de différence fondamentale entre elles et le concept de frame peut se définir en fonction des concepts de base comme nous allons le voir au § III.4.4.2. après une synthèse succincte des langages de frames.

III.4.4.1. Les frames

De nombreux langages de frames ont été développés suite aux travaux initiaux de MINSKY [MINSKY.75] et sont repertoriés dans [BONNET.85]. Tout comme les objets , les frames sont des entités caractérisées par des attributs, définis par leurs classes, ces classes étant organisées en arbre ou en treillis sur lequel opère un mécanisme d'héritage. Trois différences apparentes les séparent cependant :

1) aux attributs des frames (appelés aussi slots) sont attachés des facettes

2) tout attribut est en général accessible directement de l'extérieur du frame

3) les frames n'ont pas toujours de méthodes propres.

1) Les facettes des attributs sont nombreuses et dépendent du langage considéré. Elles sont de deux types que nous appellerons facettes passives et facettes actives. Les principales **facettes passives** sont :

- VALEUR qui contient la valeur de l'attribut,

- DEFAUT valeur par défaut de l'attribut s'il n'a pas de valeur propre dans sa facette VALEUR.

- DOMAINE domaine de valeurs possibles de l'attribut.

Il y a beaucoup de variantes sur ces facettes, notamment sur DOMAINE. Sous celle-ci, nous regroupons en fait d'autres facettes qui permettent de spécifier des contraintes sur les valeurs possibles.

Par exemple, les facettes ValueClass en KEE [FIKES-KEHLER.72] , RANGE en LORE [C.G.E.86] et SRL [WRIGHT-FOX.83], &dom en MERING, TYPE en KOOL [ALBERT.86] , &un et &liste-de en SHIRKA [RECHENMANN.86]... permettent de spécifier soit une classe d'appartenance pour la valeur de l'attribut; par exemple :

FRAME AUTO

attributs : propriétaire DOMAIN Personne "la classe Personne"

soit une classe standard INTEGER, STRING, BOOLEAN...

En KEE (et en SRL) on peut spécifier une combinaison booléenne de classes par les mots clés évidents INTERSECTION, UNION, NOT.ONE.OF., en MERING une liste de classes (appelées types) permises. Les facettes &domaine en SHIRKA et &valeurs en MERING peuvent être suivies de valeurs possibles en extension comme :

FRAME AUTO

attributs : marque &domaine "Citroën" "Renault" "Peugeot".....

Les facettes &intervalle en SHIRKA et ValueClass en KEE peuvent être suivies d'un intervalle d'entiers par exemple. La distinction entre attribut monovalué et multivalué par les facettes &un et &listede de SHIRKA est spécifiée en MERING par la facette &mode dont les valeurs possibles sont &uni et &multi. Les facettes CardinalityMin et CardinalityMax de KEE équivalentes à &card-min &card-max limitent le nombre de valeurs d'un attribut multivalué, etc...

VALUE, DEFAUT et DOMAINE ne sont donc pas les seules facettes et, à part des variantes, on trouve également beaucoup d'autres facettes plus ou moins générales ou propres aux fonctionnalités de chaque langage.

Les **facettes actives**, ou attachements procéduraux, permettent d'associer à un attribut des procédures (fonctions Lisp par exemple)) qui seront déclenchées lorsqu'un événement se produit notamment l'affectation, la lecture d'un attribut, l'ajout et la suppression d'une valeur pour les attributs multivalués. Ces procédures s'appellent réflexes en MERING, valeurs actives en KEE (et on le verra en LOOPS bien que ce ne soit pas à proprement parler un langage de frames), réactions sur événement en SHIRKA. Leur ancêtre est la notion de "démon", introduite par KRL [BOBROW-WINOGRAD.77]. Il s'agissait de procédures LISP lancées sur écriture de l'attribut puis a été introduite la notion de "servants" dans ce langage pour indiquer comment déterminer la valeur de l'attribut quand elle manquait. Nous adopterons le terme **réflexe** assez évocateur de MERING. Les réflexes possibles diffèrent en fonction des langages. SHIRKA en propose une panoplie très détaillée dont :

- &à-vérifier contrainte à vérifier pour toute valeur affectée à l'attribut (*), par exemple :

  FRAME enfant

  attributs : âge &un entier

  &à-vérifier (compris-entre 0 15)

- &sib-exec ("si besoin exécuter la procédure qui suit") correspondant aux servants de KRL, IF-NEEDED de FRL [ROBERTS. GOLDSTEIN.77] ou UFL [YOUNG.PROCTOR.86], &si-req de MERING :

(*) les exemples donnés de manière informelle pour ne pas entrer dans les détails de chaque langage.

```
FRAME      rectangle

 attributs :        largeur
                       .
                       .
                       .
                    longueur
                       .
                       .
                       .
                    surface  $un réel
                             $sib-exec
                                 (produit largeur longueur)
```

- $avant-ajout, $après-ajout, $avant-sup, $après-sup, $avant-lec, $après-lec associées respectivement à l'ajout (correspondant à la simple affectation pour un attribut monovalué) à la suppression et à la lecture d'une valeur. Les facettes $avant-xx permettent de faire des contrôles. Les facettes $après-xx sont intéressantes pour propager automatiquement d'autres événements comme l'affichage ou l'affectation d'autres attributs... Par exemple pour qu'après chaque affectation de l'état d'une entrée d'un and sa sortie soit automatiquement mise à jour, il suffit de définir le frame And comme suit :

```
FRAME  And

 attributs :  entrée1  $un booléen
                       $après-ajout (calcule-sortie...)
                  .
                  .
                  .
```

La facette $après-ajout correspond aux démons de KRL, à la facette IF-ADDED de FRL ou UFL, $si-ajout de MERING ou encore WhenFilled de KOOL. $après-sup correspond à $si-suppr de MERING ou WhenRemoved de KOOL ...

De façon synthétique, nous pouvons classifier ces réflexes en deux groupes associés respectivement à l'action de :

- lecture d'un attribut : AVANT-LECTURE, APRES-LECTURE ET SI-BESOIN (servant,IF-NEEDED, $sib-exec, $si-req)

- écriture d'un attribut : AVANT-ECRITURE, APRES-ECRITURE.

Les facettes se retrouvent sous des formes différentes dans d'autres outils orientés objet qui ne sont pas à proprement parler des langages de frames. Il est courant qu'une valeur par défaut puisse être associée à un attribut défini par une classe comme dans LRO2 [GOETZ.86], [GOETZ-ROCHE.86] (*1) ou LOOPS (*). Plus encore, LRO2 permet les attachements procéduraux en lecture ou écriture (LECTURE, DEBECRITURE et FINECRITURE) aux attributs (appelés propriétés).

Plus généralement, facettes passives et actives sont semblables respectivement aux annotations de propriété ("property annotation") et aux valeurs actives ("active values", le terme a d'ailleurs été repris par KEE) de LOOPS [BOBROW-STEFIK.81] [STEFIK and al.86]. Les annotations de propriété sont des attachements de propriétés aux valeurs des variables d'instance ou aux propriétés elles-mêmes (et récursivement). A la différence des facettes, LOOPS ne gère pas de propriétés avec une sémantique particulière mais permet simplement de les attacher. Leur interprétation est réalisée explicitement par programme. Les valeurs actives permettent de définir des procédures d'accès particulières aux valeurs des variables d'instances ou des propriétés qui seront représentants de toute sous-classe de la classe ActiveValue. ActiveValue définit le protocole standard de lecture et écriture d'une valeur, respectivement par les méthodes GetWrappedValue et PutWrappedValue. Il suffit alors de redéfinir ces méthodes pour effectuer des réflexes avant ou après lecture ou écriture. En conclusion du point **1)** on pressent déjà la manière de passer des objets simples aux frames qui se traduira par le fait qu'un frame est un objet dont les attributs sont eux-mêmes des objets avec leurs attributs, les facettes, et leurs méthodes, la lecture, l'écriture et les réflexes. Mais précisons tout d'abord les points **2)** et **3)**.

(*1) Déjà dans la version initiale appelée LRO de EAQUE-LRO [ROCHE.84], à chaque classe est associée une instance type dont les valeurs d'instances sont les valeurs par défaut "l'héritage" se faisant entre instances. LRO2 qui fait suite à LRO, fait une différence nette entre classe et instance et la valeur par défaut est directement associée à l'attribut au niveau de la classe. LRO2 est un langage orienté objet utilisable comme tel ou intégré dans l'outil de développement de systèmes experts MP-LRO2, héritier de EAQUE-LRO dont il forme la couche orientée objet pour la représentation des connaissances.

(*2) D'autres langages gèrent les valeurs par défaut. En Thinglab par exemple, à toute classe peut être associée une instance type appelée Prototype qui contient des valeurs par défaut des variables d'instances.

2) Les attributs sont en général accessibles directement par des primitives de lecture et écriture comme val? et aj-val de SHIRKA ou selon une syntaxe particulière comme en KEE (THE tel-slot OF tel-objet...). De la même façon en LRO2 ou en LOOPS, toute lecture/écriture d'un attribut se fait par application d'une primitive (GetValue, PutValue en LOOPS)(*1) . Ces primitives ou plus généralement le système gère alors automatiquement :

- les contrôles avant écriture, vérifications des contraintes imposées par toutes les facettes de type DOMAINE ou &à-vérifier, etc...

- les réflexes avant ou après écriture ou lecture

- en lecture, la recherche de la valeur par défaut ou l'application de la procédure SI-BESOIN (selon un ordre généralement paramétrable précisé par la facette &ordre-lect en MERING par exemple) si l'attribut n'a pas de valeur particulière.

En LOOPS s'il s'agit d'une valeur active, des procédures particulières GerWrappedValue et PutWrappedValue sont lancées respectivement en lecture ou en écriture de l'attribut. Cette approche est duale de celle du type SMALLTALK (*2) :

(*1) beaucoup d'autres langages orientés objet permettent l'accès direct aux attributs comme Flavors, CommonObjects, Ceyx,... par primitive standard ou en compilant automatiquement deux méthodes dont les noms sont construits syntaxiquement sur le nom de l'attribut. Comme nous l'avons déjà précisé "accessing instance variable by operations does not violate encapsulation" si ces opérations peuvent être redéfinies par l'utilisateur. La compilation automatique de méthodes d'accès (qui peut être contrôlée par des déclarations comme : Getable en CommonObjects) correspond à l'accessibilité par défaut des attributs qui peut être inhibée par la suite tandis que l'approche de type SMALLTALK est la non accessibilité par défaut (stricte encapsulation), qui peut être inhibée en définissant les méthodes d'accès.

(*2) que nous avons adoptée dans notre modèle de base des langages orientés objet. Dans [STEFIK and al.86] est proposé le paradigme d'Access oriented programming dual de object oriented programming comme c'est bien résumé dans la phrase suivante : "In access-oriented programming, fetching or storing data can cause procedures to be involved in terms of actions and side effects, this is dual to object-oriented programming. In object oriented programming, when one object sends a message to another, the receiving object may change its data as a side effect. In access-oriented programming, when one object changes its data, a message may be sent as a side effect".

- l'approche de type frame considère l'accessibilité par défaut des attributs, qui peut être contrôlée et même inhibée par les facettes et réflexes (par exemple par un réflexe avant écriture qui l'interdit).

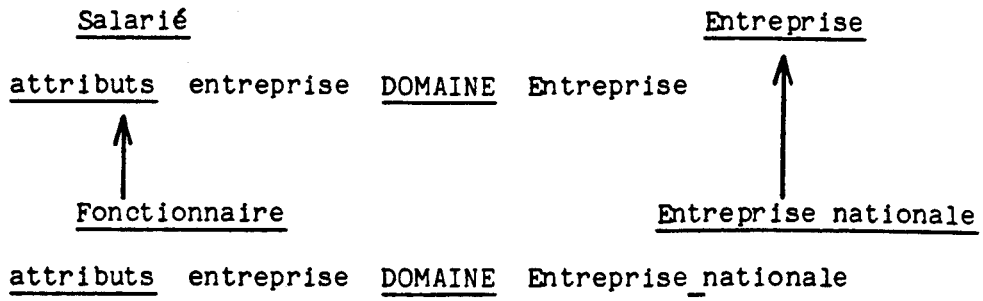
- l'approche de type SMALLTALK considère la non-accessibilité par défaut des attributs (stricte encapsulation) qui peut être inhibée par la définition d'une méthode d'accès contrôlé.

3) Contrairement aux objets des L.O.O. certains langages de frames (KRL, FRL, SHIRKA, KOOL,...) n'attachent pas de méthodes aux frames eux-mêmes, qui n'ont alors pas d'activité propre. Dans ces langages, les frames apparaissent comme des structures assez passives manipulées par application des primitives extérieures d'accès à leurs attributs qui propage l'activation des réflexes, comme cela a été résumé au 2). Cette différence entre objet et frame tend à disparaître avec des outils comme KEE, MERING ou UFL où les frames disposent d'un comportement propre défini par leurs méthodes activables par envoi de message (ou appel procédural pour UFL).

Enfin, quelques précisions concernant l'héritage. Nous avons vu au paragraphe III.3.2.1. que l'affinement par substitution n'a de sens que si les caractéristiques portent une définition propre au niveau de la classe. Ainsi, dans le modèle orienté objet de base, celui-ci ne portait que sur les méthodes. Pour les frames celui-ci a maintenant un sens pour les attributs puisqu'il leur est attaché des informations, les facettes. Ainsi et de la même façon que pour les méthodes, l'affinement par substitution d'un attribut se spécifie par masquage du nom hérité et par l'association de nouvelles facettes ou de facettes redéfinies (par masquage !). Il est ainsi possible de redéfinir une valeur par défaut (*), des contraintes de domaine de valeurs plus fortes, des réflexes. La sémantique de l'affinement des attributs dépend fortement de chaque langage :

- dans la majorité des langages, la facette DOMAINE doit être compatible avec celle héritée (ou celles en héritage multiple). S'il s'agit d'une classe, elle doit être sous-classe de la classe (ou des classes) associée à la même facette dans la sur-classe, comme dans l'exemple suivant :

(*) dans les langages orientés objet où une valeur par défaut peut être associée à un attribut (dans la méthode facette) comme en LOOPS, LRO2, la redéfinition d'une valeur par défaut se fait également par masquage de l'attribut et association d'une nouvelle valeur.



un intervalle doit être inclus dans les intervalles hérités (facette &Intervalle de SHIRKA par exemple).

- la valeur par défaut écrase toujours l'ancienne.

- les réflexes peuvent être ajoutés, écrasés par masquage, combinés (appliquer tous les réflexes après écriture par exemple).

A la base, les stratégies d'héritage sont cependant les mêmes et les techniques présentées au III.3.3. ne font pas la différence entre langages orientés objet ou langage de frames (par exemple la profondeur d'abord pour SHIRKA, la largeur d'abord pour MERING).

En conclusion, on voit apparaître beaucoup de similitudes entre frames et objets. Nous allons voir, en fait, comment la notion de frame peut être définie en fonction du concept de base d'objet.

III.4.4.2.- Frame = objet composite

Les similitudes entre frame et objet sont immédiates à partir de la constatation suivante : les facettes sont les caractéristiques des attributs des frames, les attributs sont donc eux-mêmes des objets. Les facettes passives (VALEUR, DEFAUT, DOMAINE) sont les attributs des objets attributs et les facettes actives (AVANT-LECTURE, APRES-LECTURE, SI-BESOIN, AVANT-ECRITURE, APRES-ECRITURE) sont les méthodes des objets attributs. Comme tout objet, un objet attribut est responsable de la manipulation de ses attributs et notamment de son attribut VALEUR. Ainsi la lecture et l'écriture de cet attribut sont des méthodes propres à l'objet attribut combinaison des réflexes associés (en lecture et en écriture). Synthétiquement :

> **les frames sont des objets dont les attributs sont eux-mêmes des objets avec leurs propres attributs (qui, idéalement, sont des objets et récursivement) et leurs propres méthodes.**

Cette définition est à la base de la représentation des frames (*) en termes d'objets en MERING, LORE, UFL , SHIRKA, KOOL et d'autres langages. Plus précisément :

(*) et des valeurs actives de LOOPS [STEFIK and al.86]

> Un frame est l'agrégation de ses objets attributs. C'est un objet composite formant un tout dont les parties sont ses objets attributs.

Tout naturellement, ces objets attributs sont caractérisés par des classes comme tout objet :

> "les attributs sont des entités à part entière. En particulier le type ATTRIBUT définit un moule général dont tous les sous-types d'attributs sont issus".
>
> [FERBER.84]

Comme toute classe, les classes d'attributs sont arrangées en arbre ou en treillis d'héritage dont le sommet, soit la classe ATTRIBUT (*1), est sous-classe de OBJECT et définit les caractéristiques minimum et tout attribut. LORE, par exemple, propose un treillis prédéfini de classes d'attributs assez complet : attribut monovalué, multivalué, accessible en lecture, en écriture, avec ou sans tel réflexe... L'utilisateur peut alors définir ses propres classes d'attributs en les plaçant dans ce sous-treillis.

Nous constatons donc toute la puissance des concepts de base du modèle orienté objet pour définir des notions plus riches telles que les frames. On pourra trouver dans [BRIOT.85] la définition d'un micro-MERING en termes d'objets. Voyons ici comment simuler simplement les frames en SMALLTALK, sachant que cette approche peut être généralisée à tout autre L.O.O. proche.

Un frame peut être représenté par un objet SMALLTALK, agrégation de ses objets attributs référencés par ses variables d'instances.

Tout objet attribut est caractérisé par la classe Attribut qui définit en particulier son attribut valeur et ses méthodes propres de lecture et écriture de la valeur, respectivement v et v: x. Les réflexes correspondent à des méthodes propres aux objets attributs. Leur application est gérée par les méthodes v et v: x qui y font référence par self messages. Tout accès à la valeur d'un attribut d'un frame se fait donc par envoi du message v , en lecture, v: x en écriture à l'objet attribut du frame (*2). C'est ici tout le problème de l'accès aux sous-objets, les objets attributs d'un tout, le frame, qui se pose comme pour toute agrégation (Cf. III.4.3.2.). Nous simulerons ici la solution élégante de MERING qui repose sur la notion originale de A-transmission. Dans ce langage il y a deux types d'envois de message, ou transmissions, appelés M-transmission et A-transmission.

(*1) ou de façon similaire la classe ActiveValue en LOOPS [STEFIK and al.86].

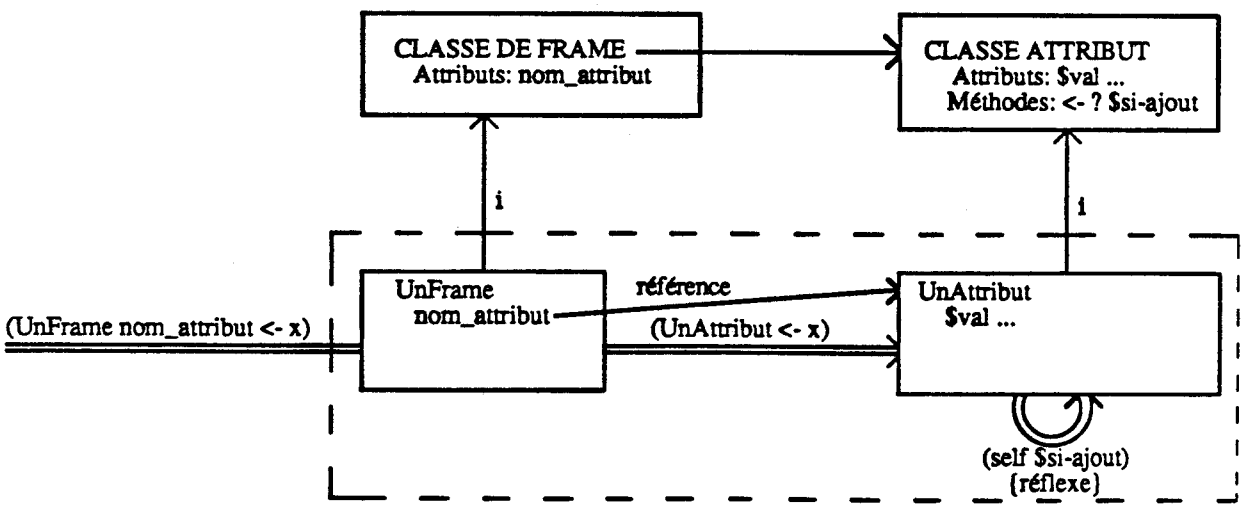
(*2) de façon similaire en LOOPS ActiveValue définit les méthodes GetWrappedValue et PutWrappedValue respectivement de lecture et écriture de la valeur. La définition de réflexes avant et après se fait par redéfinition de ces méthodes dans toute sous-classe de ActiveValue [STEFIK and al.86]

Une M-transmission (m comme méthode) est un envoi de message classique. Une A-transmission (A comme applicateur) est l'envoi d'un message à un objet attribut "à travers" l'objet frame, de façon similaire à la notion de messages composés (Cf. III.4.3.1.). La syntaxe d'une A-transmission est :

(objet-destinataire nom-attribut sélecteur arguments)

objet destinataire à la réception de ce message redirige (délègue) le message "sélecteur arguments" à son objet attribut local référencé par nom-attribut.

En particulier, pour le sélecteur d'écriture ← en MERING, correspondant à notre v: x , la transmission (Un_Frame nom-attribut ← x) provoquera la transmission locale (objet-attribut ← x) selon le schéma suivant (en représentant par => l'envoi de message) :



Nous simulerons cette solution par enchaînement de message en SMALL-TALK en supposant que toute classe de frame définit pour chaque attribut une méthode de même nom qui renvoie l'objet attribut référencé, de façon similaire aux méthodes d'accès aux parties du paragraphe III.4.3.1. (1ère version de Nandh) :

```
Classe de frame

  attributs : nomattribut...

  méthodes :  nomattribut
                "renvoyer le contenu de la variable
                 d'instance nomattribut"

              ^nomattribut
```

ainsi (unFrame nomattribut ← x) en MERING, s'écrira ici :

```
unFrame  nomattribut  v: x
```

où l'évaluation du message unFrame nomattribut renvoie l'objet attribut référencé par nomattribut auquel le message v: x est envoyé (enchaînement).

Le problème de gestion de la cohérence du tout, le frame, se pose évidemment ici et la solution sera du type pointeur inverse (Cf. III.4.3.1.), chaque objet attribut référençant son frame par sa variable d'instance "frame". Ce pointeur inverse permettra également de propager des événements sur les autres attributs du frame comme nous le verrons par la suite. En MERING, lors d'une A-transmission, les symboles ‖ et % représentent respectivement le frame destinataire et l'objet attribut, celui-ci peut ainsi faire référence au frame dans ses propres méthodes. En SHIRKA, % (ou notre variable d'instance "frame") correspond à l'attribut "lui-même" que l'on peut référencer dans un attachement procédural (*) ou auquel on peut attacher des facettes et des réflexes pour exprimer notamment des contraintes globales aux attributs. Par exemple la classe Carré définie comme sous-classe de Rectangle, hérite des attributs largeur et longueur sur lesquelles elle impose une contrainte globale d'égalité :

```
(Carré
     sorte de = (Rectangle)
     lui-même &à-vérifier (égalité largeur longueur))
```

(*) ou un filtre dont nous n'avons pas parlé.
En SHIRKA, à tout attribut correspond une variable de même nom, une variable d'un nom donné fait référence à l'attribut de même nom. La variable "lui-même" correspond donc à l'attribut "lui-même". Par ailleurs, SHIRKA gère un mécanisme de renommage (facette &var-nom, utilisée avec d'autres comme &var → &var← ...) pour gérer les conflits qui peuvent apparaître notamment dans la gestion des filtres imbriqués, quand plusieurs frames sont concernés.

Pour notre petite simulation, nous utilisons donc le traitement de base de l'agrégation avec pointeur inverse. La création des objets attributs se fait lors de l'instanciation d'un objet frame par la méthode new redéfinie en ^super new creattribut (à la manière de ^super new init), dans la classe abstraite Frame.

La méthode creattribut, à définir pour chaque frame, instancie les objets attributs qu'elle affecte aux variables d'instance correspondantes.

```
Object subclass: #Frame

Frame comment:
'Surclasse abstraite des classe de frames.'

Frame methodsFor: 'attributes creation'

creattributs
        self subclassResponsibility

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

Frame class

Frame class comment:
'Metaclasse de Frame.'

Frame class methodsFor: 'instanciation'

new
        ^super new creattributs
```

Les objets attributs sont instances de la classe Attribut (ou de toute sous-classe) selon le modèle simplifié suivant :

* les variables d'instance "valeur, défaut et domaine" représentant les facettes de même nom. Défaut et domaine devront être affectées lors de l'instanciation (newAttPour:) (*1) de l'attribut par redéfinition de la méthode pour:. Domaine contiendra une classe d'appartenance de la valeur (*2). La variable "frame" référencera le frame propriétaire (le tout) et sera affectée lors de l'instanciation d'un attribut (par pour:)

* l'écriture et la lecture de la valeur sont implémenentées par les méthodes v et v:x . Les méthodes avantEcriture:x, aprèsEcriture et siBesoin représentent les réflexes correspondants. Les méthodes v:x et v gèrent leur application par self messages (*3). Remarquons que avantEcriture:x permet de réaliser à la fois les contrôles sur la valeur (appartenance au domaine) et d'autres réflexes avant écriture que l'on pourrait ajouter en la redéfinissant dans une sous-classe d'attributs.

(*1) Remarquons qu'une amélioration propre à SMALLTALK serait de définit défaut et domaine en variables de classe puisqu'elles sont partagées par toute instance d'une même classe d'atrributs.

(*2) Nous nous restreindrons aux attributs monovalués. Il serait facile de définir une sous-classe d'attributs multivalués pour lesquels valeur référencerait une collection et en ajoutant les méthodes de suppression, ajout,... et les réflexes correspondants.

(*3) Remarquons que les réflexes avant et après seraient programmés très facilement en Flavors par l'attachement de méthodes before et after aux méthodes v et v:x (Cf. § III.3.4.3). Les méthodes before et after de Flavors apparaissent rétrospectivement comme une généralisation de l'attachement de réflexes à toute méthode.

```
Object subclass: #Attribut
        instanceVariableNames: 'valeur defaut domaine frame '
Attribut comment:
'Surclasse de toutes les classes d''attributs.'

Attribut methodsFor: 'writing'
v: x
        " methode d'ecriture de la facette valeur, gere les reflexes
        associes "
        (self avantEcriture: x)
                ifTrue:
                        [valeur <- x.
                        self apresEcriture]
                ifFalse: [^ ' contraintes avant ecriture !']
Attribut methodsFor: 'at creation'
pour: unFrame
        "Par defaut, pointeur inverse vers son frame englobant,
        unFrame.Elle sera redefinie pour affecter les facettes
        defaut et domaine dans chaque classe d'attribut."
        frame <- unFrame
Attribut methodsFor: 'reflexes'
apresEcriture
        "Par defaut rien, donc self"
        ^self
avantEcriture: x
        "au minimum verification de la valeur a ecrire:
        indeterminee ou appartenance au domaine."
        ^ x isNil or: [x isKindOf: domaine]
siBesoin
        "Par defaut, la valeur elle-meme (eventuellement
        indeterminee). "
        ^valeur
Attribut methodsFor: 'reading'
v
        " La variable valcalculee est locale a la methode. Elle
        contiendra le resultat de l'application de siBesoin, si
        besoin !. Nil = indeterminee."
        | valcalculee |
        valeur isNil
                ifTrue:
                        [valcalculee <- self siBesoin.
                        valcalculee isNil
                                ifTrue: [^defaut]
                                ifFalse: [^valcalculee]]
                ifFalse: [^valeur]
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
Attribut class

Attribut class comment:
'Metaclasse de Attribut.'

Attribut class methodsFor: 'instanciation'
newAttPour: unFrame
        ^self new pour: unFrame
```

Soit la classe de frames Andf définie informellement comme suit :

Andf

attributs : entrée1

domaine : booléen

défaut : false

aprésEcriture : évaluer et renvoyer la valeur de l'état de sortie

entrée2

idem

sortie

domaine : booléen

défaut : false

avantEcriture : interdite

siBesoin : entrée1 and entrée2

entrée1 et entrée2 seront donc représentants des classes Attentrée, sortie représentant de Attsortie, ces deux classes étant des spécialisations de Attétat comme suit :

```
Attribut subclass: #AttEtat
        instanceVariableNames: ''

AttEtat methodsFor: 'at creation'
pour: unFrame
        super pour: unFrame.
        defaut <- false.
        domaine <- Boolean


AttEtat subclass: #AttSortie

AttSortie methodsFor: 'reflexes'
avantEcriture
        " interdiction d'ecriture sur la sortie"
        ^false
avantEcriture: x
        " interdiction d'ecriture sur la sortie"
        ^false
siBesoin
        "La sortie est evaluee par reflexe si besoin en lecture"
        ^frame entree1 v & frame entree2 v
```

```
AttEtat subclass: #AttEntree

AttEntree methodsFor: 'reflexes'
apresEcriture
        "Apres ecriture d'une entree, renvoyer la sortie"

        ^frame sortie v

Frame subclass: #Andf
        instanceVariableNames: 'entree1 entree2 sortie '

Andf methodsFor: 'attributes accessing'
entree1
        ^entree1
entree2
        ^entree2
sortie
        ^sortie

Andf methodsFor: 'at creation'
creattributs
        entree1 <- AttEntree newAttPour: self.
        entree2 <- AttEntree newAttPour: self.
        sortie <- AttSortie newAttPour: self
```

DIALOGUE AVEC UN FRAME

```
Creation d'un frame:
   Andf1 <- Andf new

Valeurs par defaut:
   Andf1 entree1 v
   false

   Andf1  sortie v
   false

Contraintes avant ecriture
   1 au lieu de true
    Andf1 entree2 v: 1
    ' contraintes avant ecriture !'

   Ecriture interdite sur la sortie
     Andf1 sortie v: true
     ' contraintes avant ecriture !'

Apres ecriture d'une entree, evaluation et retour de la sortie:
  Andf1 entree1 v: true
  false

La sortie va passer a true:
  Andf1 entree2 v: true
  true

Sachant qu'elle est evaluee par reflexe siBesoin:
   Andf1 sortie v
   true
```

III.5.- HERITAGE/AGREGATION

Nous avons vu que héritage et agrégation étaient deux mécanismes fondamentaux de structuration de la connaissance, nous allons essayer ici de les comparer.

Rappelons synthétiquement que l'héritage permet de traduire une hiérarchie conceptuelle de classes bâtie sur le lien IS-A interprété comme une relation d'abstraction/concrétisation, généralisation/spécialisation, et plus généralement d'affinement, de classification ou de taxonomie. L'agrégation d'objets permet de traduire une hiérarchie structurelle bâtie sur le lien IS-PART de décomposition d'un tout (objet composite, agrégat) en parties (sous-objets, composants). Elle ne bénéficie pas, en général, d'un traitement particulier et doit être explicitement programmée par attributs, référence d'objet et envoi de message pour son inférence propre.

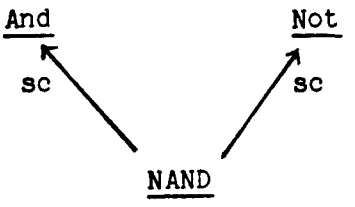
Bien que l'opposition IS-A/IS-PART semble conférer à l'héritage et à l'agrégation des rôles bien distincts, l'appréhension de ces deux notions ne semble pas toujours immédiate. Cette synthèse a pour but de marquer les différences entre elles et constitue avant tout une mise en garde conceptuelle et méthodologique.

Au III.5.1., nous allons montrer que l'héritage entre classes ne peut se substituer à l'agrégation.

Au III.5.2., nous verrons que dans une certaine mesure, l'agrégation peut se substituer à l'héritage au prix d'une programmation assez lourde qui n'est pas satisfaisante dans un contexte de représentation des connaissances.

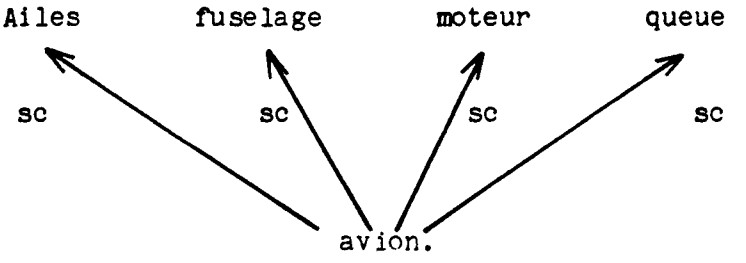
III.5.1.- Agrégation par héritage ?

Nous avons vu, dans le paragraphe III.3.5.4. l'intérêt de l'héritage multiple pour créer des classes par combinaison de plusieurs sur-classes. D'autre part, l'agrégation permet de définir des classes d'objets, assemblages d'objets d'autres classes. En rapprochant combinaison et assemblage, une tentation est alors de représenter l'agrégation par héritage multiple. Pour notre exemple du Nand, ceci conduirait à définir cette classe, sous-classe des classes And et Not de ses parties :



Cette approche de l'agrégation par héritage pose non seulement des problèmes conceptuels et méthodologiques mais également techniques.

L'opposition conceptuelle et méthodologique que nous partageons considère qu'il s'agit d'une utilisation impropre de la hiérarchie de classes. Rappelons que celle-ci traduit fondamentalement une hiérarchie conceptuelle bâtie sur le lien IS-A et non structurelle bâtie sur le lien IS-PART. Ici, il y aurait confusion entre les deux hiérarchies, un nand n'est pas un and ni un not mais est composé d'un and et d'un not. Prenons un second exemple tiré de [HALBERT-O'BRIEN.87] :



De la même façon :

> "A conceptual error is being made here. An airplane is composed of these parts : it is not the sum of the behavior of these parts. Multiple supertypes allow us to view an instance of a subtype as being an instance (*1) of any of its several supertypes. But we are not trying to view an airplace in such a way : we don't really want to think of an airplane as a more sophisticated way of viewing an engine or a fuselage (*2). We should not use subtyping for accomplish this kind of aggregation".
> [HALBERT-O'BRIEN.87]

En effet, l'argument inverse serait de dire qu'un nand est un and avec les caractéristiques d'un not ou inversement un not avec les caractéristiques d'un and (!). Utiliser la hiérarchie de classes pour l'agrégation, c'est, d'autre part, aller à l'encontre du principe d'inclusion (III.3.5.5.) par lequel le lien "sur-classe" traduit l'inclusion de l'ensemble des objets représentants de la sous-classe dans celui associé à chacune de ses sur-classes. On ne peut considérer que l'ensemble des nand est inclus dans celui des and et celui des not, ni même que l'ensemble des avions est inclus dans l'ensemble des ailes, l'ensemble des moteurs... Par ailleurs, rappelons que, d'après ce principe, tout représentant d'une sous-classe peut se substituer à un représentant d'une sur-classe, ce n'est pas le cas ici.

(*1) ou plutôt un représentant d'après nos définitions.

(*2) or wings or a tail !

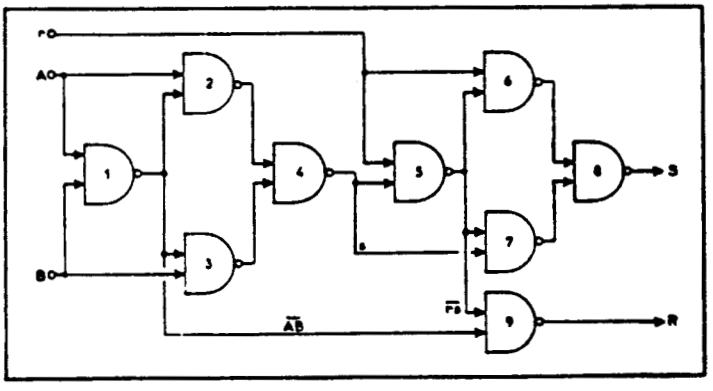
Un avion ne peut être utilisé là où on s'attend à considérer des ailes... ni même qu'un nand ne peut remplacer un and ou un not dans un circuit.

Ce point de vue, qui peut paraître théorique, est conforté par l'opposition technique qui suit :

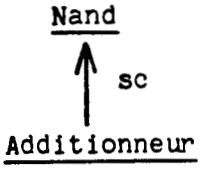
opposition technique : hériter ce n'est pas instancier.

"If our airplane has more than one engine, or if we wanted wings to be broken down into two instances of wing, we could not use subtyping for aggregation".
[HALBERT-O'BRIEN.87]

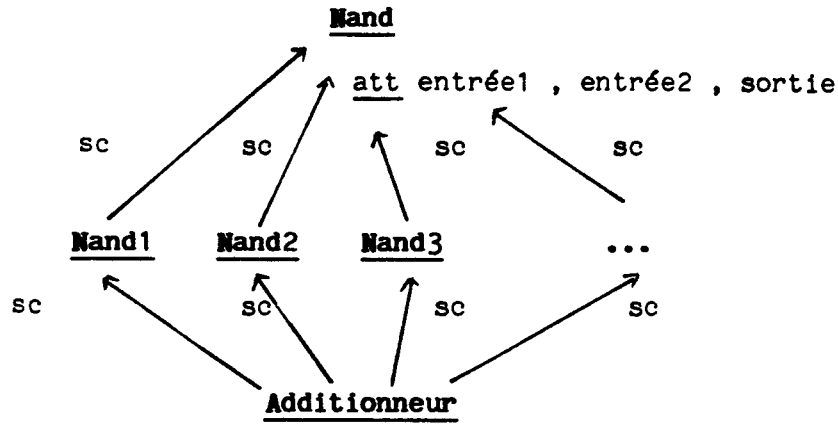
Or, l'agrégation repose essentiellement sur l'instanciation, elle traduit des relations entre instances comme nous l'avons vu au III.4.3.2. Chaque instance de Nand est composée de deux objets, instances respectives de And et Not et l'instanciation de Nand doit provoquer l'instanciation de And et de Not, ce qui n'est pas traduit par l'héritage de la classe Nand précédente. Tout au plus l'héritage permettrait de considérer que les caractéristiques d'un Nand sont la somme des caractéristiques d'un And et d'un Not, moyennant les problèmes de conflit d'héritage. Cette opposition est d'autant plus forte s'il s'agit d'un agrégat de plusieurs instances d'une même classe comme dans l'exemple de l'additionneur suivant :



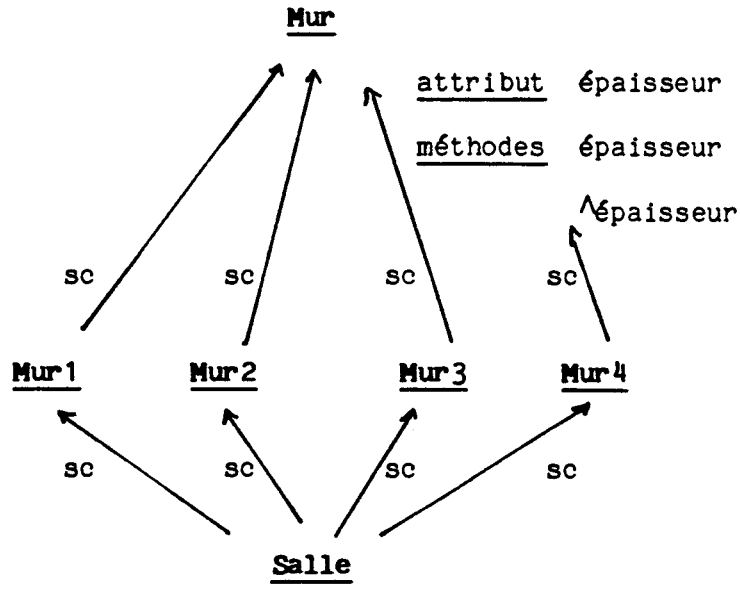
qui ne peut se traduire par la hiérarchie de classe :



ni même par l'artifice qui consisterait à établir plusieurs chemins d'héritage entre la sous-classe et la surclasse :

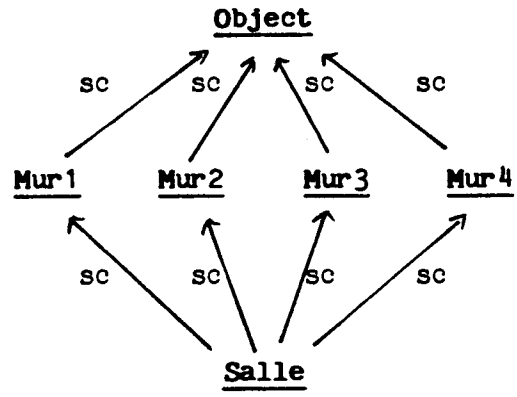


En effet, l'approche générale [SNYDER.86b] est qu'une classe n'hérite qu'une seule fois des attributs définis dans une sur-classe accessible par plusieurs chemins d'héritage que ce soit par stratégie linéaire (ce qui est évident) ou par stratégie graphique. Ainsi Additionneur héritera d'un seul jeu d'attributs entrée1 , entrée2 , sortie et non de neuf, comme l'utilisateur aurait pu le souhaiter. Plus précisément, la sous-classe définissant un tout hériterait des caractéristiques des sur-classes définissant les parties, ce qui mène à des incohérences d'interprétation. Un additionneur n'est pas caractérisé par les attributs entrée1 , entrée2 , sortie qui ne lui sont pas propres mais qui caractérisent ses parties. De la même façon, sur l'exemple suivant :



Par héritage, une salle aurait la caractéristique épaisseur, ce qui n'est évidemment par l'interprétation souhaitée.

Un utilisateur acharné à représenter ce type d'agrégation de parties par héritage pourrait toujours s'en sortir pour hériter autant de fois des caractéristiques d'une sur-classe qu'il y a de parties. Le résultat final sera toujours une simulation de l'agrégation car les objets représentant les parties n'auront pas toujours d'existence propre (pas d'instanciation) et leurs caractéristiques seront justaposées dans l'espace des caractéristiques du tout. Ainsi, les attributs du tout seront l'amalgame des attributs des parties, ce qui pose des problèmes de conflits d'une façon particulièrement cruciale si les parties sont issues d'une même classe. Le subterfuge est alors plus ou moins facile à réaliser selon la stratégie d'héritage implémentée. Le principe est de supprimer d'abord la sur-classe de factorisation, soit Mur, puisque l'on ne peut en hériter qu'une fois.



Dans le cas de stratégies linéaires (*), où il ne peut y avoir de conflits de noms, il faut alors nommer les attributs communs différemment, par exemple par préfixage du nom de la classe, soit pour l'attribut épaisseur, les attributs mur1épaisseur, mur2 épaisseur,... définis dans les classes respectives Mur1 , Mur2,... Dans le cas d'une stratégie graphique, ce préfixage est automatique. La caractérisation développée de Salle sera alors :

Salle : attributs

mur1épaisseur
mur2épaisseur
mur3épaisseur
mur4épaisseur

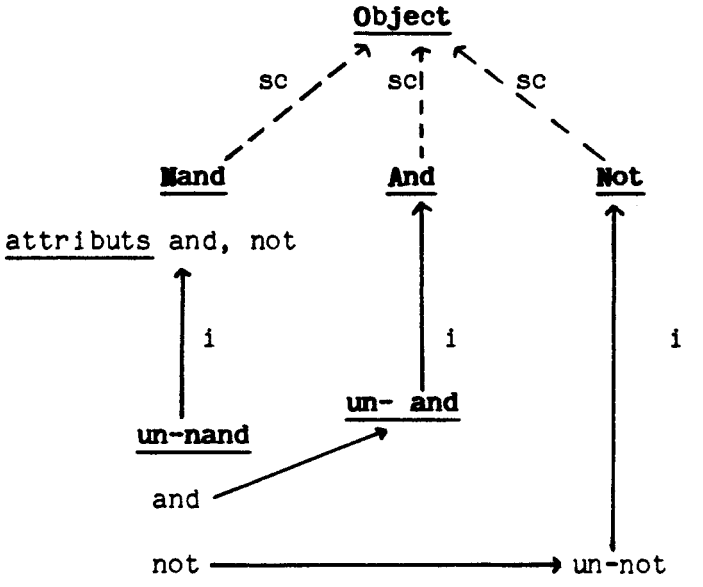
ce qui n'est pas satisfaisant dans une approche modulaire du type orienté objet. L'héritage entre classes ne peut donc généralement se substituer à l'agrégation entre objets. C'est pourquoi le traitement de base de l'agrégation par attribut et référence d'objet est le moyen le plus satisfaisant de la représenter dans la majorité des outils orientés objet.

(*) où les stratégies graphiques avec fusion des attributs de même nom comme [BORNING-INGALLS.82].

Les problèmes précédents ne se posent pas alors :

> "We already have an excellent mechanism : object fields. In our example, Airplane should have fields that contain the wings, fuselage, engine and tail objects. Achieving aggregation through fields instead of through inheritance also allows more than one instance of a particular type of object, since a subtype cannot inherit a supertype more than once".
>
> [HALBERT-O'BRIEN.87].

Cette représentation a été présentée au III.4.2. comme le traitement de base de l'agrégation. Un Nand est alors caractérisé par deux attributs and et not qui référencent respectivement une instance de And et de Not.



De la même façon, la classe Additionneur pourra définir un attribut pour chacune des portes Nand ou un seul attribut contenant une collection à neuf cases qui contiendra pour chaque additionneur, neuf objets nand; une salle sera l'assemblage de quatre murs, avec leur propre épaisseur définie localement.

Nous préconisons donc cette approche satisfaisante à tous points de vue, notamment conceptuel puisqu'elle ne fait pas de confusion entre hiérarchie structurelle et hiérarchie conceptuelle. L'interprétation de la hiérarchie de classes est alors homogène, le lien sur-classe ne traduisant que la relation IS-A et l'inclusion ensembliste.

Même si cette approche pose des problèmes pour le traitement de la hiérarchie des parties comme nous l'avons vu au paragraphe III.4.3., elle est à la base de la représentation d'une telle hiérarchie qui ne peut être confondue avec la hiérarchie de classe

> "Class hierarchies do not provide part hierarchies".
>
> [BLAKE-COOK.87]

III.5.2.- Héritage par agrégation ?

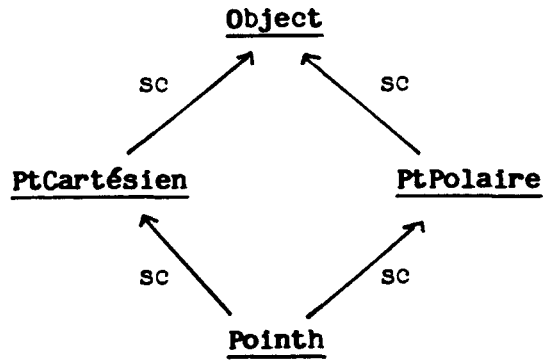
Le paragraphe précédent est parti de la propriété commune à l'héritage et l'agrégation pour décrire des combinaisons ou assemblages. Nous avons vu que les combinaisons réalisées par l'héritage ne pouvaient pas (ou difficilement) se substituer aux assemblages réalisés par l'agrégation. La question inverse est posée ici, peut-on réaliserl'héritage par l'agrégation ?

A priori, la réponse est non puisque :

- l'héritage traduit une relation entre classes alors que l'agrégation relie des instances de ces classes.

- il n'y a pas de mécanisme d'inférence de caractéristiques entre instances.

L'héritage peut cependant être simulé par l'agrégation comme suit. Nous avons vu, au paragraphe III.4.3.2. que l'inférence entre instances pouvait être programmée explicitement par envoi de message. Il suffit alors de lever le premier obstacle en remplaçant la relation d'héritage entre classes par des relations d'agrégation entre leurs instances respectives. Prenons l'exemple suivant où chaque sur-classe correspond à un point de vue cartésien ou polaire sur les représentants de la sous-classe Pointh des points à la fois caractérisée en coordonnées polaires et cartésiennes. Pointh redéfinit le protocole d'écriture selon un point de vue pour assurer la conversion selon l'autre point de vue :

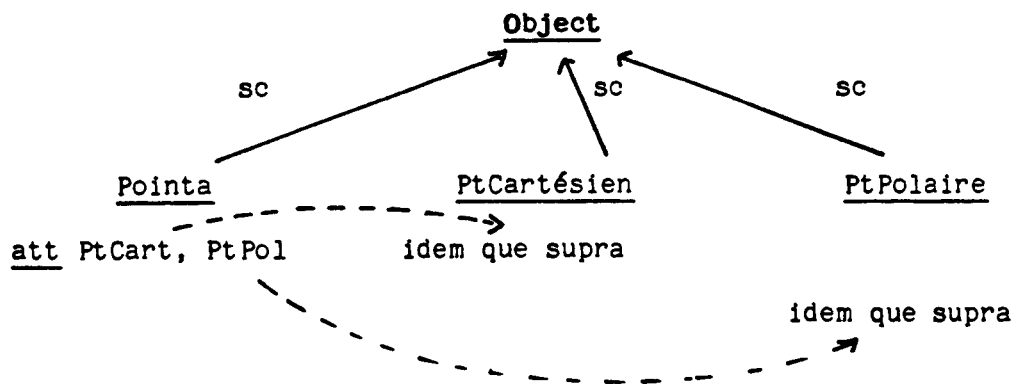


```
Object subclass: #PtCartesien
        instanceVariableNames: 'x y '

PtCartesien comment:
'Point cartesien.'

PtCartesien methodsFor: 'reading'

x
        ^x

y
        ^y


PtCartesien methodsFor: 'writing'

x: x0 y: y0
        x <- x0.
        y <- y0


Object subclass: #PtPolaire
        instanceVariableNames: 'r angle '

PtPolaire comment:
'Point polaire.'

PtPolaire methodsFor: 'reading'

angle
        ^angle

r
        ^r

PtPolaire methodsFor: 'writing'

r: r0 angle: a0
        r <- r0.
        angle <- a0
```

```
Class named: #Pointh
        superclasses: 'PtCartesien PtPolaire '
        instanceVariableNames: ''

Pointh comment:
'Par heritage multiple.'

Pointh methodsFor: 'writing'

r: r0 angle: a0
        "calcul des coordonnees cartesiennes equivalentes"

        | x0 y0 |
        x0 <- r0 * a0 cos.
        y0 <- r0 * a0 sin.

        "affectation des attributs dans les deux representations"

        self super.r: r0 angle: a0.
        self super.x: x0 y: y0

x: x0 y: y0
        "calcul des coordonnees polaires equivalentes "

     |r0 a0|
        r0 <- (x0 squared + y0 squared) sqrt.
        x0 ~= 0
                ifTrue: [a0 <- (y0 / x0) arcTan]
                ifFalse: [y0 = 0
                                ifTrue: [a0 <- 0.0]
                                ifFalse: [y0 < 0
                                                ifTrue: [a0 <- Pi / 2 negated]
                                                ifFalse: [a0 <- Pi / 2]]].

      "affectations des attributs dans les deux representations"

       self super.x:x0 y:y0.
       self super.r:r0 angle:a0
```

```
Dialogue avec un Pointh.

Creation d'un Pointh:

  Pth1 <- Pointh new

Soient ses coordonnees cartesiennes:

  Pth1 x:10.0 y:10.0

Alors:

  Pth1 x
   10.0
  Pth1 y
   10.0
  Pth1 r
   14.1421
  Pth1 angle
   0.785398

Puis, soient ses coordonnees polaires:

   Pth1 r: 10.0 angle: 0.0

Alors:

  Pth1 r
   10.0
  Pth1 angle
   0.0
  Pth1 x
   9.99999
  Pth1 y
   0.0
```

Une simulation par agrégation serait :



```
Object subclass: #Pointa
        instanceVariableNames: 'ptCart ptPol '

Pointa comment:
'Par aggregation.'

Pointa methodsFor: 'at creation'

construct
        ptCart <- PtCartesien new.
        ptPol <- PtPolaire new

Pointa methodsFor: 'writing'

r: r0 angle: a0
        "calcul des coordonnees cartesiennes equivalentes"

        | x0 y0 |
        x0 <- r0 * a0 cos.
        y0 <- r0 * a0 sin.

        "affectation des attributs dans les deux representations.
         Noter le remplacement de self par le sous objet correspondant"

        ptPol r: r0 angle: a0.
        ptCart x: x0 y: y0

x: x0 y: y0
        "calcul des coordonnees polaires equivalentes "

     |r0 a0|
        r0 <- (x0 square + y0 square) sqrt.
        x0 ~= 0
                ifTrue: [a0 <- (y0 / x0) arcTan]
                ifFalse: [y0 = 0
                                ifTrue: [a0 <- 0.0]
                                ifFalse: [y0 < 0
                                                ifTrue: [a0 <- Pi / 2 negated]
                                                ifFalse: [a0 <- Pi / 2]]].

      "Affectations des attributs dans les deux representations.
       Noter le remplacement de self par le sous objet correspondant"

       ptCart x:x0 y:y0.
       ptPol r:r0 angle:a0


Pointa methodsFor: 'reading'

angle
        ^ptPol angle

r
        ^ptPol r

x
        ^ptCart x

y
        ^ptCart y
```

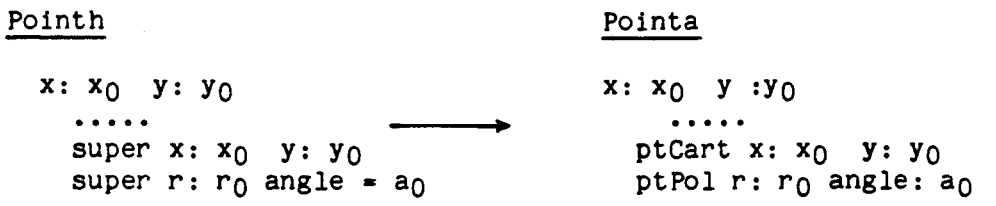
```
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "
Pointa class

Pointa class comment: 'Metaclasse de Pointa'

Pointa class methodsFor: 'instanciation'
new
        ^super new construct


Dialogue avec un Pointa.

Creation d'un Pointa:

  Pta1 <- Pointa new

Soient ses coordonnees cartesiennes:

  Pta1 x:10.0 y:10.0

Alors:

  Pta1 x
   10.0
  Pta1 y
   10.0
  Pta1 r
   14.1421
  Pta1 angle
   0.785398

 ...
```

Les attributs ptCart et ptPol référenceront respectivement une instance de PtCartésien et PtPolaire à l'instanciation d'un point par redéfinition classique de la méthode de classe new dans Point (*1). Nous appellons Pointh (h comme héritage) la première version de Point et Pointa (a comme agrégation) la seconde.

(*1)
```
new
    ^super new construct
        avec construct, méthode d'instance :
construct
    ptPol ← PtPolaire new
    ptCart ← PtCartésien new.
```

Concernant les spécifications externes, Pointh et Pointa sont bien équivalentes car elles définissent exactement le même protocole pour leurs instances. Les liens d'agrégation (représentés par les attributs ptCart et ptPol) de chaque instance de Pointa avec les instances respectives de PtCartésien et PtPolaire se substituent aux liens sous-classe entre Pointh et PtCartésien, PtPolaire. L'héritage de PtCartésien et PtPolaire pour Pointh est simulé par la redirection de message d'une instance de Pointa vers ses composants respectifs (*1) Une différence essentielle à ce niveau de la présentation est que l'objet unique dans une solution de type héritage est scindé en plusieurs objets dans une simulation par agrégation. Ici, un Pointh est remplacé par un Pointa et 2 objets respectivement instance de PtCartésien et PtPolaire. La structuration réalisée par les sur-classes est remplacée par la structuration en sous-objets. Cette différence est d'autant plus marquée dans les méthodes définies par la sous-classe qui sont des combinaisons de méthodes héritées par self messages ou super messages (qui cachent, rappelons-le, des self messages, Cf. III.3.4.2.). Ces messages doivent être répartis sur les sous-objets en remplaçant super(self !) par le composant correspondant, par ex :

(*1) Remarquons une fois de plus le rapprochement entre l'agrégation et le modèle prototype/délégation. Les instances de PtCartésien et PtPolaire apparaissent comme des prototypes d'une instance de Point (Pointa), inversement extension de ceux-ci. La redirection de messages s'assimile à la délégation entre l'extension et ses prototypes. Ce n'est cependant qu'une similitude. En effet, la délégation réalise automatiquement la redirection des messages, qui est au contraire programmée explicitement ici. Cette différence n'est que d'ordre technique; dans ce cas précis, il est possible d'implémenter un mécanisme qui réalise cette propagation comme les sélecteurs composée (Cf. III.4.3.1.) ou une généralisation des applicateurs à la MERING (Cf. III.4.4.2.) ou encore la solution de Pie qui sera présentée dans ce paragraphe. Ces mécanismes ne peuvent cependant s'étendre à la "vraie délégation" à cause d'un problème technique appelé "The SELF problem" par LIEBERMAN [LIEBERMAN.86b]. Ce problème est largement discuté dans ce dernier article et ne sera pas présenté ici. Donnons simplement le résultat de LIEBERMAN : "True delegation cannot be implemented in these system", en faisant référence aux systèmes bâtis sur le modèle classe/instance/héritage.

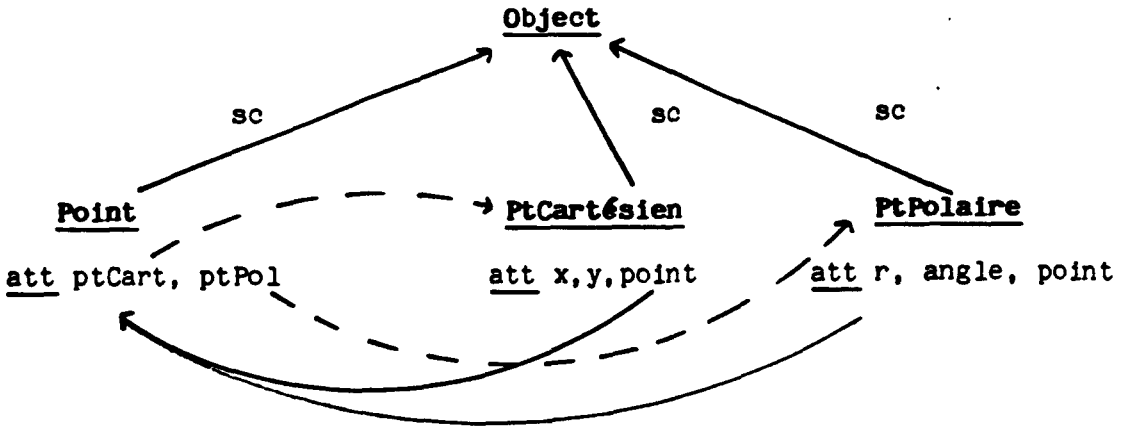


Cet exemple pose le principe de la simulation de l'héritage par l'agrégation, qui, fondamentalement, repose sur la transformation des liens IS-A en liens IS-PART. Cette simulation est même parfois assimilée à l'héritage multiple comme dans THINGLAB [BORNING.79] et PIE [GOLDSTEIN.BOBROW.80]. Or, conceptuellement, ces deux approches sont radicalement opposées comme le sont les liens IS-A et IS-PART, les hiérarchies conceptuelle et structurelle. Traiter l'héritage par l'agrégation mène à une confusion entre ces concepts qu'il faut s'interdire, ne serait-ce que méthodologiquement.

> "It is this sense that multiple inheritance was implemented by means of parts in Thinglab. This creates a rather blurred distinction between a part hierarchy and a class hierarchy".
> [BLAKE-COOK.87]

De plus, l'approche de type agrégation a l'inconvénient d' "aplatir" complétement la hiérarchie de classes et de perdre ainsi l'avantage de l'aspect déclaratif de l'héritage. En effet, celui-ci doit être remplacé par une programmation explicite de l'inférence par envoi de messages, comme nous l'avons vu dans l'exemple type précédent. Notamment, l'héritage du protocole doit être réalisé "à la main", comme c'est le cas pour Pointa qui spécifie ainsi le même protocole que Pointh. Il est évident qu'une solution de type pointeur inverse de l'agrégation permettrait de ne pas avoir à spécifier tout ce protocole au niveau de Pointa. Plus généralement,l'idée d'un traitement de type agrégation étant posée, le problème peut alors être considéré comme un problème d'agrégation à part entière avec tout ce que cela suppose (Cf. III.4.3.1.). Considérons par exemple la solution suivante où chaque partie référence le tout par pointeur inverse, en l'occurrence par l'attribut "point". Un point (instance de Pointaprim) permet alors par les méthodes pointCartésien et pointPolaire l'accès direct à ses parties et la cohérence globale est maintenue par la méthode cohérence. Le problème de circularité est résolu en paramètrant cette méthode par le sous-objet qui a été modifié et qui est donc pris comme référence pour propager la mise à jour de l'autre pour la cohérence.

(*) Ce problème classique de cohérence globale avec circularité se pose dans bien des cas, notamment quand les sous-objets ont des rôles de même niveau. D'autres solutions sont possibles et dépendent du cas précis traité (par exemple ici le paramètre pourrait être un booléen). En Thinglab, ce problème serait résolu par déclaration de contraintes globales dans le tout.




```
Object subclass: #Pointaprim
        instanceVariableNames: 'ptCart ptPol '

Pointaprim comment:
'Par aggregation, les sous objets sont visibles.'

Pointaprim methodsFor: 'coherence'

coherent: source
        | x0 y0 r0 a0 |
        (source isKindOf: PtCartesienPrim)
                ifTrue:
                        [x0 <- source x.
                        y0 <- source y.
                        r0 <- (x0 square + y0 square) sqrt.
                        x0 ~= 0
                          ifTrue: [a0 <- (y0 / x0) arcTan]
                          ifFalse: [y0 = 0
                                      ifTrue: [a0 <- 0.0]
                                      ifFalse: [y0 < 0
                                                  ifTrue: [a0 <- Pi / 2 negated]
                                                  ifFalse: [a0 <- Pi / 2]]].
                        ptPol r: r0 angle: a0]
                ifFalse:
                        [r0 <- source r.
                        a0 <- source angle.
                        ptCart x: r0 * a0 cos y: r0 * a0 sin]


Pointaprim methodsFor: 'subobjects accessing'

pointCartesien
        ^ptCart

pointPolaire
        ^ptPol


Pointaprim methodsFor: 'at creation'

construct
        ptCart <- PtCartesienPrim newPour: self.
        ptPol <- PtPolairePrim newPour: self
```

```
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "

Pointaprim class

Pointaprim class comment:
'Metaclasse de Pointaprim.'

Pointaprim class methodsFor: 'instanciation'

new
        ^super new construct


Object subclass: #PtCartesienPrim
        instanceVariableNames: 'x y point '

PtCartesienPrim comment:
'Point cartesien, en temps que partie d''un point (pour Pointaprim).'

PtCartesienPrim methodsFor: 'at creation'

pour: unPoint
        point <- unPoint

PtCartesienPrim methodsFor: 'writing'

x: x0 y: y0
        x <- x0.
        y <- y0.
        point coherent: self

PtCartesienPrim methodsFor: 'reading'

x
        ^x

y
        ^y
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "
PtCartesienPrim class

PtCartesienPrim class comment: 'Metaclasse de PtCartesienPrim.'

PtCartesienPrim class methodsFor: 'instanciation'

newPour: unPoint
        ^super new pour: unPoint
```

```
Object subclass: #PtPolairePrim
        instanceVariableNames: 'r angle point '

PtPolairePrim comment:
'Point polaire, en temps que partie d''un point (pour Pointaprim).'

PtPolairePrim methodsFor: 'at creation'

pour: unPoint
        point <- unPoint

PtPolairePrim methodsFor: 'writing'

r: r0 angle: a0
        r <- r0.
        angle <- a0

PtPolairePrim methodsFor: 'reading'

angle
        ^angle
r
        ^r
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "
PtPolairePrim class

PtPolairePrim class comment: 'Metaclasse de PtPolairePrim.'

PtPolairePrim class methodsFor: 'instanciation'

newPour: unPoint
        ^super new pour: unPoint


Dialogue avec un PointaPrim.

Creation d'un PointaPrim:

  Ptap1 <- Pointaprim new

Soient ses coordonnees cartesiennes:

  Ptap1 pointCartesien x:10.0 y:10.0

Alors:

  Ptap1 pointCartesien x
   10.0
  Ptap1 pointCartesien y
   10.0
  Ptap1 pointPolaire r
   14.1421
  Ptap1 pointPolaire angle
   0.785398

...
```

Cette solution s'éloigne alors de Pointh puisqu'un Point ne répond plus au même protocole, qui était hérité. Soit Pointa' cette dernière version de Point et un_point, un-pointa un-pointa' des instances respectives de Point, Pointa, Pointa'. Les correspondances sont alors :

un-point x	un-pointa x	un-pointa' pointCartésien x
un-point x: x_0	un-pointa x: x_0	un-pointa' pointCartésien x: x_0
y: y_0	y: y_0	y: y_0

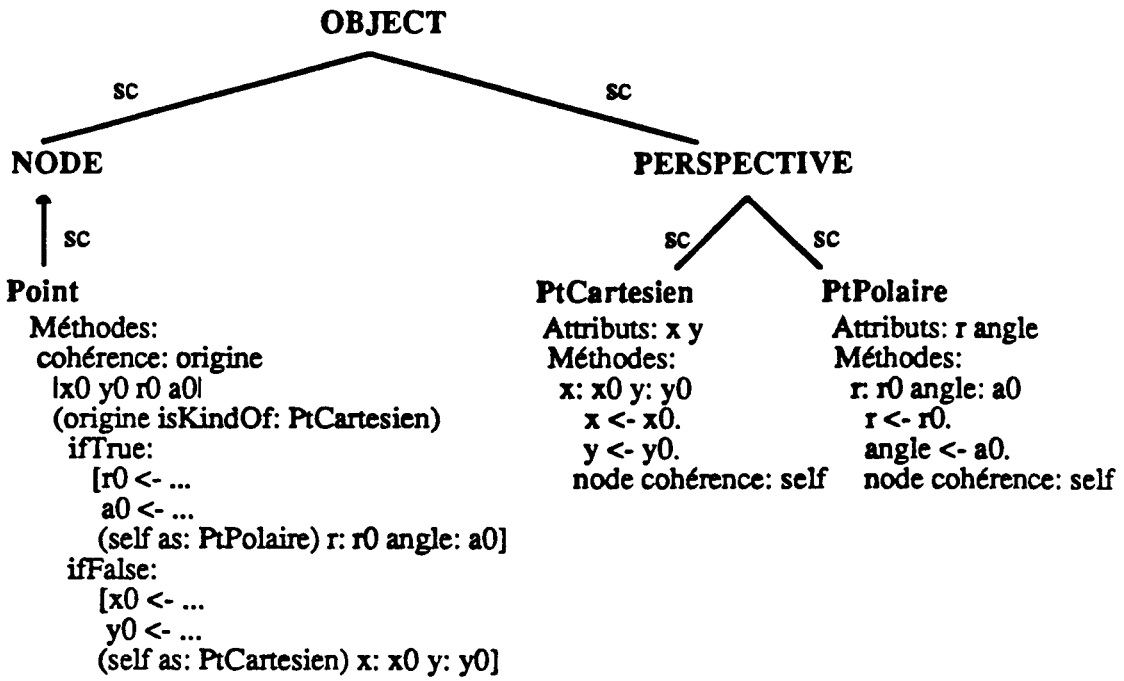
. . . .

Cette approche a été adoptée pour une première expérimentation de l'héritage multiple en SMALLTALK dans l'environnement **PIE** [GOLDSTEIN-.BOBROW.80]. L'outil proposé, repose sur la notion de **perspective** qui peut être associée à un objet. Un tel objet est représentant de la classe Node. Celle-ci définit essentiellement la variable d'instance "perspectives" qui contient la liste des perspectives de l'objet, elles-mêmes représentants de la classe Perspective.

> "Multiple inheritance is achieved by assigning perspectives to nodes. A perspective is an instance of a class that represents the node from a particular point of view. For example a node representing a part of a displayed circuit design might have a CircuitElement perspective and a DisplayObject perspective" (*).

La classe Perspective définit une variable d'instance "node" qui contiendra le pointeur inverse vers le noeud (appelons ainsi un représentant de Node). Tout noeud sait répondre au message "as: nom-perspective" par lequel il renvoie l'objet perspective correspondant à qui on peut alors envoyer des messages. Ainsi, Point serait déclarée sous-classe de Node, PtCartésien et PtPolaire sous-classes de Perspective. Point hérite de la variable "perspectives" qui contiendra une liste formée d'une instance de PtCartésien et une instance de PtPolaire. Ces classes de perspectives héritent de la variable "node" qui pointera vers une instance de Point et se substitue à notre attribut "point" (pointeur inverse). La gestion de la cohérence doit être programmée explicitement à la manière de notre solution. Ainsi :

(*) L'application à l'héritage multiple est en fait une restriction des notions qui suivent comme nous le verrons dans le chapitre IV. En fait, nous verrons à la section IV.3.3.3. que cette approche dépasse la simulation de l'héritage multiple et permet rétrospectivement de réaliser la représentation multiple et évolutive d'objet par agrégation.



La seule méthode définie dans Point (soit Pointa" cette dernière version) est cohérence puisque l'accès aux sous-objets, les perspectives, est assuré par la méthode générale as: .

Soit un-pointa" une instance de Pointa", la correspondance est alors :

un-pointa' pointcartésien x, un-pointa' point cartésien x: x_0 y: y_0

(un-pointa" as:PtCartésien) x,	(un-pointa" as:PtCartésien) x:x_0 y:y_0 ...

Cette technique apparaît donc comme une généralisation de l'approche précédente (Pointa') qu'elle facilite en offrant des outils adaptés. Bien qu'élégante, elle reste une simulation de l'héritage multiple par agrégation dont elle conserve les inconvénients principaux :

- elle met à plat de la hiérarchie de classes, perdant ainsi le bénéfice de l'inférence implicite garantie par l'héritage.

- l'héritage multiple permet de voir un même objet sous des points de vue différents alors que l'agrégation le scinde en sous-objets transformant ainsi la hiérarchie conceptuelle en hiérarchie structurelle.

- l'objet n'a alors plus qu'un rôle de gestionnaire de ses parties et un effort de programmation plus grand doit être fourni pour les contrôler, notamment pour maintenir la cohérence globale. On imagine facilement la complexité croissante de ces contrôles quand le nombre de points de vue augmente.

L'agrégation peut donc difficilement se substituer à l'héritage. Remarquons cependant qu'une telle approche peut être (et est souvent) empruntée pour pallier l'absence d'héritage multiple dans les langages à héritage simple.

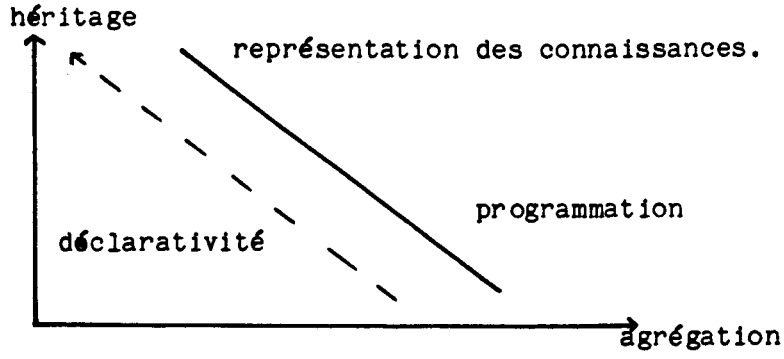
III.5.3.- Conclusion

Héritage entre classes et agrégation reposent fondamentalement sur les notions opposées de hiérarchie conceptuelle de classes bâtie sur le lien IS-A et hiérarchie structurelle d'objets bâtie sur le lien IS-PART.

Nous avons vu que l'héritage entre classes ne pouvait se substituer à l'agrégation. Inversement, l'agrégation peut se substituer à l'héritage au prix d'un effort de programmation assez poussé. Même si dans certains cas un même problème peut être vu sous les deux angles, les qualités déclaratives de l'héritage dûes à son caractère implicite privilégient son utilisation notamment dans un contexte de représentation des connaissances. Inversement, le caractère explicite du traitement de l'agrégation relève plus de la programmation.

Nous avons vu, en particulier, au paragraphe III.3.5.5. que l' "agrégation pour l'implémentation" se substituait avantageusement à l' "héritage pour l'implémentation" dans notre modèle de langages où des techniques d'héritage sélectif, indépendant de la notion de sous-classe, n'existent pas. Synthétiquement l'intérêt porté à l'héritage et l'agrégation par les domaines d'application de l'approche orientée objet peut être traduit par le schéma suivant (*).

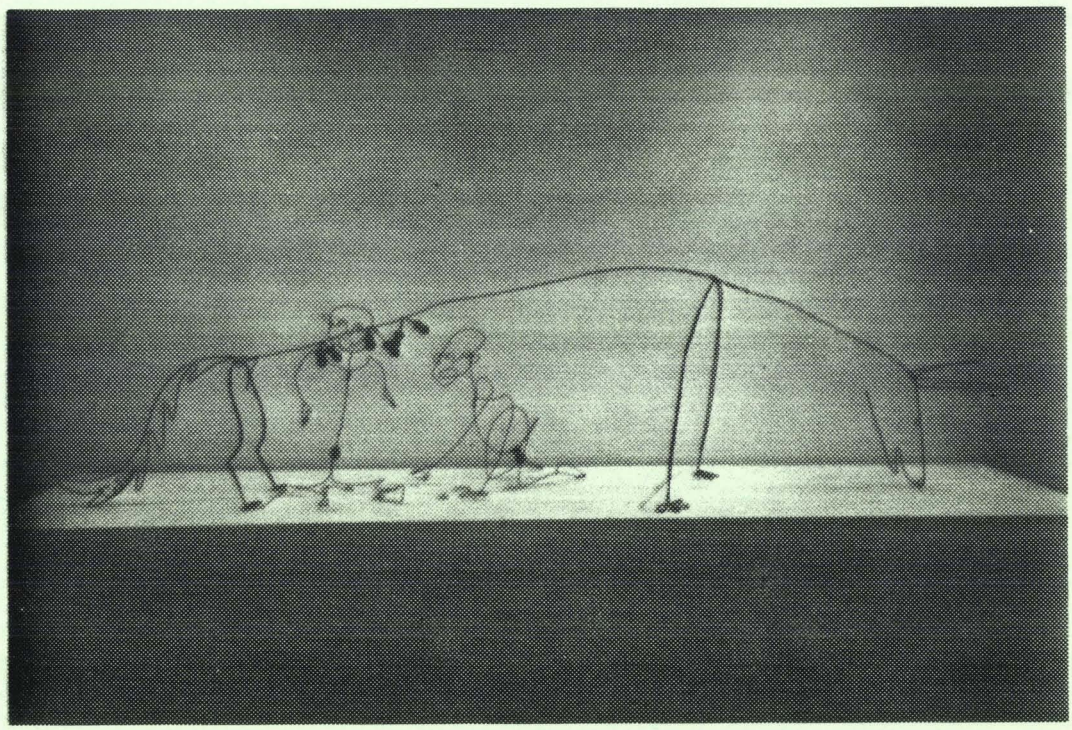
(*1) inspiré d'une présentation de J. BEZIVIN aux Journées "I.A. et Sciences Cognitives", GRENOBLE, 18-20 Février 1987.



Cependant, bien que la hiérarchie de classes et l'héritage participent fortement à l'intérêt du paradigme orienté objet pour la représentation des connaissances, nous allons constater dans le chapitre suivant que leur pleine utilisation dans ce contexte est limitée par l'instanciation.

R.O.M.E

REPRESENTATION
D'**O**BJET
MULTIPLE ET
EVOLUTIVE



"Romus et Romulus"
A. Calder
(fil de fer et bois)

ou

la naissance de R.O.M.E. chez les L.O.O.

CHAPITRE IV

*****

L E L A N G A G E R O M E

IV.1.- INTRODUCTION

Nous présentons le langage **ROME, Représentation d'Objet Multiple et Evolutive,** d'expérimentation de concepts orientés objet.

ROME est défini à la base par un noyau métacirculaire. L'originalité de ce noyau repose sur différents points :

* la séparation entre les notions de méthodes, encapsulées dans l'objet, i.e. applicables uniquement par celui-ci, et de sélecteurs réservés à la communication entre objets par envoi de message.

* l'introcution dans le modèle métacirculaire de base de la notion de classe abstraite par opposition aux classes d'instanciations.

* une stratégie graphique d'héritage multiple sans conflits, fondée sur la notion originale de **point de vue**. L'activation et l'activité d'un objet sont repensées autour de cette notion, ce qui mène à la réalisation d'un mécanisme d'inférence de points de vue.

Ces différents points sont détaillés et analysés par rapport au modèle orienté objet conventionnel étudié dans les chapitres précédents. Ceci fait l'objet de la section IV.2 de ce chapitre.

Dans la section suivante, la notion originale de **Représentation Multiple et Evolutive** est présentée. Il s'agit d'une extension du modèle orienté objet de base étudiée dans le cadre de la représentation des connaissances. Elle est fondée essentiellement sur la critique du mécanisme d'instanciation classique, comme seul moyen d'exploiter la hiérarchie de classes pour les objets, qui induit une **Contrainte de Représentation Unique Figée**. Par cette contrainte, l'objet ne peut être directement représentant que d'une seule classe, sa classe d'instanciation, qui fige ses caractéristiques à sa création. Nous montrons que cette contrainte du modèle classique :

* complique artificiellement le treillis des classes par rapport à sa spécification initiale, notammant dans le cas de classifications multiples par points de vue (représentation multiple).

* interdit l'utilisation de l'interprétation de ce treillis comme hiérarchie conceptuelle de généralisation/affinement pour les objets eux-mêmes (représentation évolutive).

La Représentation Multiple et Evolutive de Rome est alors proposée comme solution, par la remise en cause de cette contrainte pour une classe d'objets R-OBJECT. Sa sémantique est analysée algébriquement. Puis nous montrons que l'héritage multiple par points de vue présenté initialement reste identique et trouve ici son intérêt amplifié, notamment parce que la représentation multiple privilégie et encourage la définition de classifications multiples par points de vue. Enfin nous comparons cette notion à d'autres solutions, tant du point de vue de l'implémentation que des concepts.

IV.2.- LE NOYAU METACIRCULAIRE DE BASE

IV.2.1.- Présentation générale

IV.2.1.1.- L'objet

Tout objet dispose d'une caractérisation, ensemble de caractéristiques de 3 types :

Les attributs : Ou variables d'instance, caractéristiques factuelles locales à l'objet. Le premier attribut de tout objet est "iclass" qui le lie à sa classe d'instanciation et correspond à l'attribut "est-un" classique (II.2.2.).

Les méthodes : Caractéristiques comportementales locales à l'objet. Une méthode est définie par un couple :

(nom_de_méthode λ-expression lisp)

Tout objet sait appliquer ses méthodes par la primitive **applymeth** :

(applymeth nom_de_méthode arguments)

De façon similaire au couple apply/funcall de lisp, il existe une autre primitive d'application de méthode, **callmeth** qui est à applymeth ce que funcall est à apply :

(callmeth nom_de_méthode argt1 ... argtn)

= **{nom_de_méthode argt1 ... argtn}**

<=> (applymeth nom_de_méthode (list argt1 ... argtn))

Les premières méthodes dont dispose tout objet sont les méthodes d'accès à ses attributs :

en lecture : {? nom_attribut}

en écriture : {← nom_attribut valeur}

Les sélecteurs : Caractéristiques de communication de l'objet, interface ou protocole de communication de l'extérieur. A chaque sélecteur est associé un nom de méthode de l'objet :

définition de sélecteur = (nom_de_sélecteur nom_de_méthode)

L'envoi de message par la primitive **send** est le moyen de communication entre objets à travers leur protocole :

(send objet_destinataire nom_de_sélecteur argt1 ... argtn)

= **[objet_destinataire nom_de_sélecteur argt1 ... argtn]**

|_____|
message

A la réception d'un message, l'objet réagit en appliquant ("applymeth...ant"). Sa méthode associée au sélecteur si celui-ci appartient à son protocole, sinon il y a erreur (du type "message not understood"). La transformation MR (Message Reaction) traduit ce mécanisme et établit le lien entre send et applymeth :

Transformation MR :

message : [O 'sélecteur argt1 ... argtn]

=> réaction : {méthode argt1 ... argtn}

avec (sélecteur méthode) la définition de sélecteur du protocole de O inférée par héritage.

IV.2.1.2.- Les classes

IV.2.1.2.a.- Les classes de représentation, r-classes

Une classe de représentation est un objet qui sait caractériser d'autres objets, ses représentants, c'est-à-dire définir un modèle d'objet tel que précédemment. Pour cela elle est elle-même caractérisée par les 3 attributs suivants :

attributs contient la liste des attributs

méthodes contient la liste des méthodes

sélecteurs contient la liste des sélecteurs.

Par ailleurs, toute classe de représentation est sous-classe d'autres classes, au minimum la classe la plus générale OBJECT (Cf. infra). La liste de ses surclasses est contenue dans son attribut **surcl**. Cette relation classe/surclasse induit un treillis des classes (par abus de langage) de sommet OBJECT sur lequel opère un mécanisme d'inférences des caractéristiques, l'héritage multiple.

IV.2.1.2.b.- Les classes d'instanciation, i-classes

Une i-classe est une r-classe qui sait instancier (créer, générer) des objets (ses instances) selon le modèle qu'elle définit (en tant que r-classe). Pour cela toute i-classe dispose de la méthode d'instanciation **basicnew** :

(basicnew (lambda (nom_obj . linit) ...))

où nom_obj est le nom de l'objet à créer et linit est une liste d'initialisation des attributs de la nouvelle instance.

A la méthode basicnew est associé le sélecteur **new** du protocole de toute i-classe. Ainsi l'instanciation est réalisée par l'envoi du message new à une i-classe.

Tout objet est instance d'une i-classe.

IV.2.1.3.- OBJECT/R-CLASS/I-CLASS

Selon l'inspiration d' OBJVLISP, il existe 3 classes initiales qui définissent métacirculairement les objets vus précédemment :

OBJECT est la classe la plus générale, sommet du treillis des classes, dont tout objet est représentant. Elle définit donc le modèle commun à tous les objets, notamment :

* l'attribut **iclass** qui lie l'objet à sa classe d'instanciation, lors de sa création (tout objet est instance d'une i-classe).

* les méthodes ? et ← d'accès à ses attributs.

R-CLASS sous-classe de OBJECT, est la (méta)classe de toutes les classes de représentation dont elle définit le modèle notamment les attributs **attributs méthodes sélecteurs** et **surcl**.

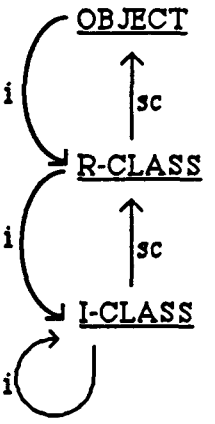
I-CLASS sous-classe de R-CLASS est la (méta)classe de toutes les classes d'instanciation. C'est elle qui définit la méthode d'instanciation **basicnew** et le sélecteur **new** associé.

Ces trois classes initiales ne peuvent se définir l'une sans l'autre. OBJECT qui est la classe de représentation la plus générale est instance de R-CLASS et sous-classe d'aucune autre classe.

R-CLASS définit les classes de représentation qui sont des objets particuliers, elle est donc sous-classe de OBJECT. Par ailleurs, elle est génératrice de ces classes, donc doit disposer de la fonctionnalité d'instanciation définie par I-CLASS, elle en est donc instance.

I-CLASS définit les classes d'instanciation, qui sont des classes de représentation particulières, elle est donc sous-classe de R-CLASS. Par ailleurs, elle est génératrice des i-classes donc, de façon similaire à R-CLASS, est instance de I-CLASS, c'est-à-dire d'elle-même.

Synthétiquement, ces trois classes sont situées sur le schéma méta-circulaire suivant :



Dans les paragraphes suivants, nous précisons les aspects originaux de ce modèle, et le comparons au modèle classique du chapitre II.

IV.2.2.- Les classes : abstraction vs génération [CARRE et COMYN.87a]

Conventionnellement, la notion de classe des langages orientés objet associe deux fonctionnalités fondamentales (II.3.2) :

* l'abstraction par laquelle une classe décrit un modèle d'objet (attributs, méthodes), qui peut être dérivé d'autres modèles (relation classe/sous-classe).

* la génération ou instanciation par laquelle une classe est un objet générateur d'autres objets, ses instances (new).

Nous avons vu (III.3.5.2) l'importance de la notion de classe abstraite dans la construction d'une hiérarchie de classes, dont une révélation est en particulier celle de mixin (III.3.5.5).

De telles classes ne sont pas instanciables car ne décrivent pas de modèles complets d'objet ("viables") mais n'ont que le rôle de décrire des modèles partiels notamment issus du processus de factorisation de caractérisations communes. En se rapprochant des deux fonctionnalités précédentes, ceci peut se formuler comme suit :

> Une classe abstraite n'a que la fonctionnalité d'abstraction et non celle de génération.

Or, dans le modèle classique, toute classe a la fonctionnalité de génération. En effet, la classe CLASS de toutes les classes définit à la fois la fonctionnalité d'abstraction et celle d'instanciation de telle sorte qu'il n'est pas possible dans ce modèle de créer des classes abstraites, qui n'ont dès lors pas de statut réel.

En ROME, les classes ont avant tout la fonctionnalité d'abstraction d'objets définie par la classe R-CLASS (attributs, méthodes, sélecteurs, surcl). Certaines ont en plus la fonctionnalité de génération définie par I-CLASS (méthode basicnew, sélecteur new).

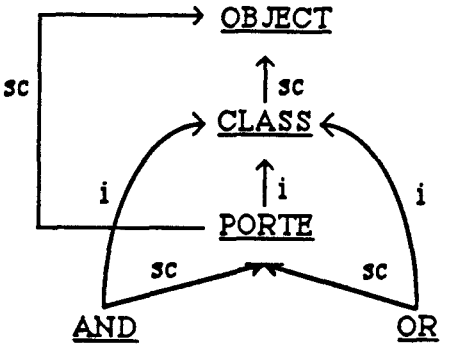
I-CLASS, sous-classe de R-CLASS, hérite de celle-ci la fonctionnalité d'abstraction. Les i-classes ont donc les deux fonctionnalités et correspondent aux classes des autres langages.

R-CLASS, qui ne définit que la fonctionnalité d'abstraction, est en particulier la classe des classes abstraites :

> Une classe abstraite est représentant de R-CLASS sans être représentant de I-CLASS.

L'inexistence, dans le modèle classique, de véritables classes abstraites vient rétrospectivement de la fusion des deux fonctionnalités au sein du même modèle de classe défini par CLASS, i.e. la fusion de R-CLASS et I-CLASS en CLASS.

Reprenons l'exemple des portes logiques, dans le modèle classique, nous obtenons le schéma (II.3.3) :



Nous avons vu (III.3.5.2) que PORTE est une classe abstraite. Il suffit, pour s'en persuader, d'observer la méthode calcule-sortie qui ne peut être définie à ce niveau (self subclassResponsability en SMALLTALK).

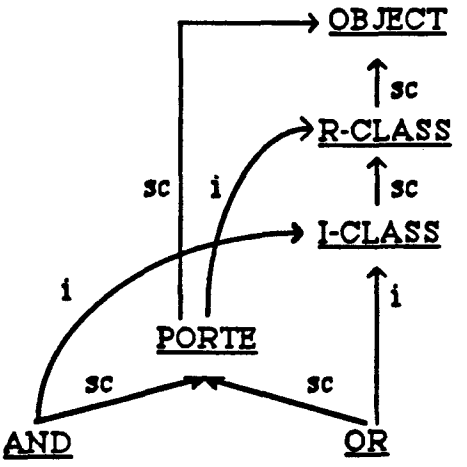
PORTE n'est cependant abstraite qu'en intention, puisque en tant qu'instance de CLASS, elle dispose de la fonctionnalité de génération (new) (*)

(*) en reprenant l'implémentation de l'exemple en OBJVLISP, il suffit de faire (send PORTE 'new) pour se créer une instance de PORTE, objet non viable.

En ROME, PORTE, classe abstraite, est instance de R-CLASS, contrairement à AND et OR, instances de I-CLASS car classes d'instanciation respectivement de portes and et or.

```
[R-CLASS 'new 'PORTE
 'surcl '(OBJECT)
 'attributs '(entree1 entree2 sortie)
 'methodes
  '(e1= (lambda (etat) {<- 'entree1 etat} {calcule_sortie})
    e2= (lambda (etat) {<- 'entree2 etat} {calcule_sortie})
    calcule_sortie (lambda () {erreur "indeterminee"}))
 'selecteurs
   '(entree1= e1=
     entree2= e2=)]

[I-CLASS 'new 'AND
 'surcl '(PORTE)
 'methodes
  '(calcule_sortie
    (lambda () {<- 'sortie (and {? 'entree1} {? 'entree2})}))]

[I-CLASS 'new 'OR
 'surcl '(PORTE)
 'methodes
  '(calcule_sortie
    (lambda () {<- 'sortie (or {? 'entree1} {? 'entree2})}))]
```

menant au schéma suivant :



PORTE ne peut être instanciée :

[PORTE 'new 'aPorte] engendre une erreur car new ne fait pas partie du protocole de PORTE (n'est pas définie par sa classe, R-CLASS).

Contrairement à AND et OR.

> [AND 'new 'aAnd]

= aAnd.

Il en serait de même pour les classes abstraites Number, Boolean ... de SMALLTALK (III.3.5.2) ou les mixins NamedObject DatedObject de LOOPS (III.3.5.5.).

Notons que la classe OBJECT est une classe abstraite puisqu'elle ne définit aucun modèle d'objet particulier mais au contraire le modèle abstrait d'objet le plus général.

IV.2.3.- Objets : r-classes, i-classes, instances terminales

Reprenons tout d'abord les définitions suivantes (II.3.1) :

Définitions

* Tout objet est représentant direct (instance) de sa classe d'instanciation.

  Soit IC la classe d'instanciation de O ([IC 'new 'O])

  [O 'repd? IC] = true (*)

  (O isMemberOf: IC = true en SMALLTALK).

* Tout objet est représentant d'une classe, soit C

  - s'il est représentant direct de C
  - ou si sa classe d'instanciation est sous-classe de C.

  [O 'repd? C] = true (*)

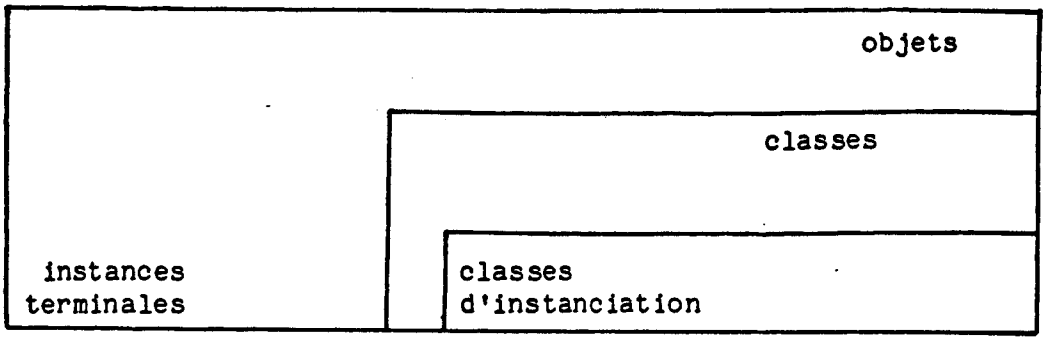
  (O isKindOf: C = true en SMALLTALK).

Ainsi :

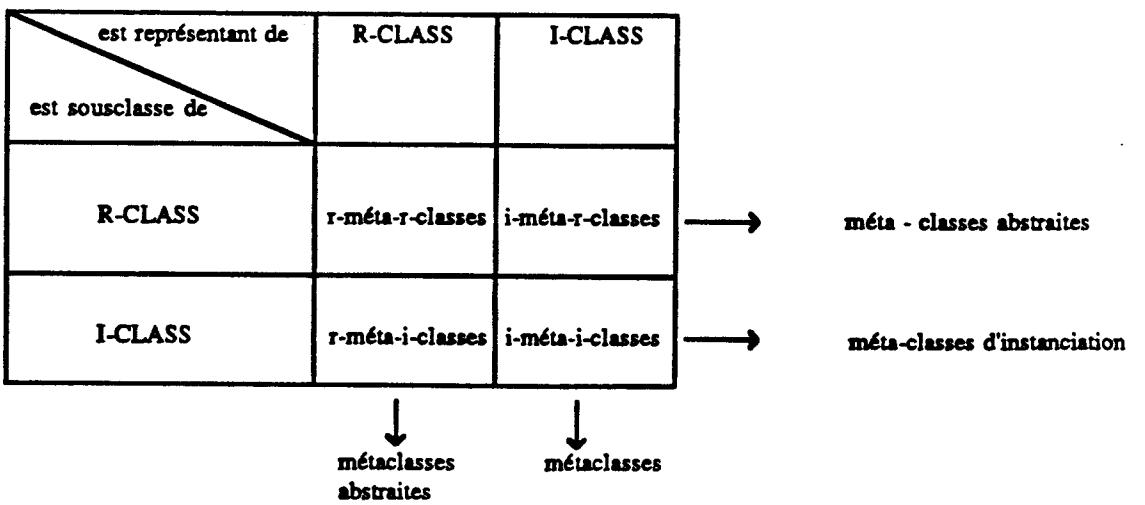
* tout objet est représentant de OBJECT
* tout objet représentant de R-CLASS est une r-classe, ou plus simplement une classe.
* toute r-classe représentant de I-CLASS est une i-classe, ou classe d'instanciation
* toute objet non représentant de R-CLASS est une instance terminale.

(*) repd? et repg? sont des sélecteurs définis par OBJECT.

Schématiquement, nous avons le diagramme suivant :



Comme dans le modèle classique issu de OBJVLISP (Cf. II.3.2), les métaclasses de ROME sont des classes, donc représentant de R-CLASS ou I-CLASS, sous-classes d'une des métaclasses initiales, R-CLASS ou I-CLASS. La seule différence par rapport à ce modèle vient une fois de plus de la distinction R-CLASS/I-CLASS.



Les X-méta-r-classes (1ère ligne) décrivent des modèles de r-classes, puisqu'elles sont sous-classes de R-CLASS.

Les X-méta-i-classes (2ème ligne) décrivent des modèles de i-classes, puisque sous-classes de I-CLASS.

Les r-méta-X-classes (1ère colonne) sont des classes de représentation de classes, notamment des métaclasses abstraites.

Les i-méta-X-classes (2ème colonne) sont des classes d'instanciation de classes.

Notamment :

* R-CLASS est une i-méta-r-classe, méta-r-classe initiale,

* I-CLASS est une i-méta-i-classe, méta-i-classe initiale.

IV.2.4.- Encapsulation des méthodes vs interface

IV.2.4.1.- Applymeth vs send

Nous avons vu au paragraphe II.3.1. que , classiquement, les notions de méthodes et de sélecteurs ne font qu'une et que le seul moyen d'appliquer une méthode est l'envoi de message, c'est le postulat P2 de OBJVLISP. Ce postulat s'applique également à l'objet lui-même, ce qui conduit à l'utilisation des variables SELF et SUPER dans les langages respectant ce postulat, aspects étudiés en II.3.4 :

> L'objet pour appliquer une de ses méthodes doit envoyer à lui-même (self) le message correspondant.

Il y a ainsi complète assimilation entre méthodes et protocole : toute méthode fait partie du protocole de l'objet et est potentiellement applicable par envoi de message.

En ROME, le postulat P2 n'est pas vérifié. Méthodes et sélecteurs sont dissociés dans la caractérisation d'objet (IV.2.1.1.). Les méthodes sont des opérations internes à l'objet, de telle sorte que seul lui-même (SELF) peut les appliquer par la primitive **applymeth**. L'envoi de message, par la primitive **send**, est réservé à la communication entre objets à travers leur protocole. Le protocole est une liste d'associations sélecteur-méthode qui spécifie pour un sélecteur donné la méthode à appliquer (par applymeth) lors de la réception d'un message.

Sur l'exemple de la classe PORTE (II.2.2.1), les sélecteurs entrée1= et entrée2= sont liés respectivement aux méthodes e1= et e2=. Soit unAnd une instance de AND,

[AND 'new 'unAnd]

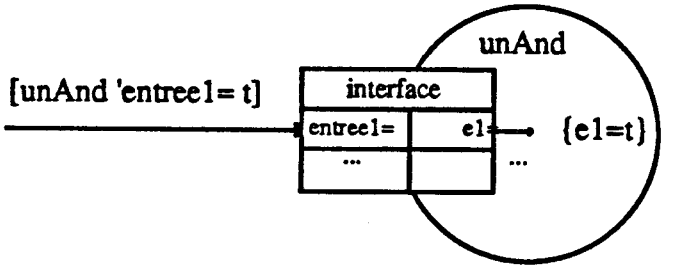
à la réception du message :

[unAnd 'entrée1= t]

unAnd répondra en appliquant sa méthode e1=

{e1= t}

d'après la transformation MR.



Cette approche permet d'assurer une nette séparation entre implémentation du comportement (méthodes) et interface externe de l'objet et va ainsi dans le sens d'une plus grande encapsulation, principe fondamental de l'approche Orientée Objet (III.2). Ceci est à rapprocher de la distinction public/privé de certains langages comme Treillis/Owl ou C_{++}. Dans ces langages il est possible de définir des opérations privées, réservées à l'objet (self) et publiques, c'est-à-dire applicables par les autres objets, les clients. La spécification public/privé est un peu différente en ROME où toute méthode est avant tout privée, seules certaines, liées à des sélecteurs, peuvent faire l'objet de requêtes externes.

Notons en particulier que l'encapsulation des attributs d'objet est garantie par celle des méthodes d'accès en lecture, ? , et surtout en écriture, ← , définies par OBJECT.

Le paragraphe suivant expose les conséquences de cette approche sur la programmation des méthodes.

IV.2.4.2.- Conséquences sur la programmation des méthodes

La primitive applymeth de ROME est aux méthodes ce que la primitive apply de LISP est aux fonctions. La programmation des méthodes en ROME se fait donc dans un style "à la LISP" dans l'environnement fonctionnel propre de l'objet, défini par l'ensemble des méthodes dont il dispose. L'implémentation des méthodes reste strictement local à cet environnement sans passer par le protocole externe contrairement à la technique classique des self messages (III.3.4.1). Celle-ci est cependant directement transposable en ROME moyennant la transformation

(send SELF nom_de_méthode arguments)

en l'application de méthode de ROME :

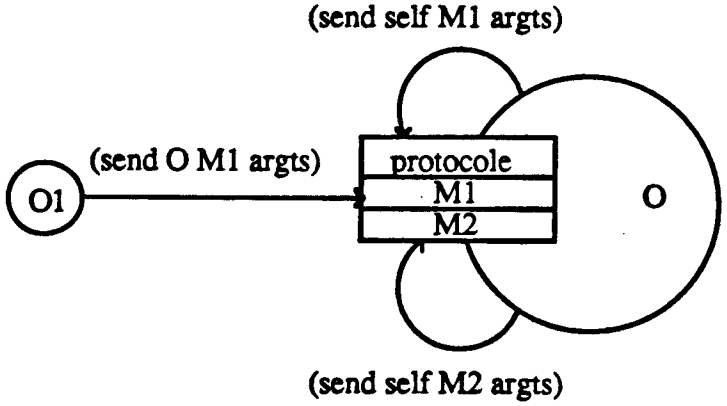
(applymeth nom_de_méthode arguments)

puisque classiquement toute méthode est applicable par self-message et qu'en ROME toute méthode est applicable (par "self") par applymeth.

La différence essentielle des deux approches vient du fait que classiquement toute méthode est applicable non seulement par self mais également par tout autre objet, puisqu'elle fait implicitement partie du protocole de l'objet. Ainsi l'écriture modulaire des méthodes grâce à self (III.3.4.1) en les décomposant en sous-méthodes et récursivement rend implicitement celles-ci publiques, même si elles ne doivent pas l'être.

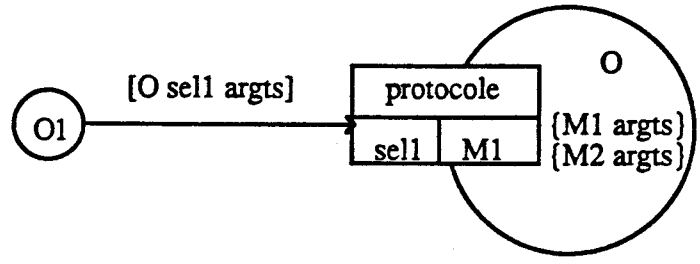
Soit O un objet caractérisé par les méthodes M1 et M2 et O1 un autre objet.

Classiquement



O pour appliquer ses méthodes M1 et M2 doit s'envoyer les messages correspondants, tout comme tout autre objet, tel O1. Ceci n'est particulièrement pas satisfaisant si M2 est privée, i.e. réservée à self (ici O) comme sur la figure. En effet, rien n'empêche O1 d'envoyer le message "M2 argts".

En ROME



O est caractérisé par les méthodes M1 et M2 qu'il peut appliquer localement (par applymeth {...}) et le sélecteur sel1 lié à la méthode M1 , qui seule peut faire l'objet d'une requête externe.

Des exemples de méthodes privées sont "calcule-sortie" de PORTE (III.3.4.1 et IV.2.2.1) qui ne doit pas être applicable directement de l'extérieur mais seulement sur écriture des entrées (méthodes entrée1= , entrée2=) "délai" de NotTps ou "valideAge" de personne (III.3.4.1) que nous reprenons ici en ROME :

```
[I-CLASS 'new 'Personne
 'surcl '(OBJECT)
 'attributs '(age)
 'methodes
  '(age= (lambda (unNombre)
           {valideAge unNombre}
           {<- 'age unNombre})
    valideAge (lambda (x)
                ;entre 1 et 100
                (when (or (< x 1) (> x 100))
                      {erreur "age entre 1 et 100"})))
 'selecteurs '(tonAge= age=)]
```

Rappelons que cette décomposition de age= en valideAge permet d'affiner la contrainte sur l'âge par redéfinition de la méthode valideAge dans des sous-classes, telles que ENFANT, ADULTE,...

Contrairement à la solution classique (en SMALLTALK au III.3.4.1), la méthode valideAge est strictement encapsulée dans l'objet, il n'y a en effet aucun sens de l'inclure dans son protocole.

Ce dernier exemple peut être généralisé à un type de décomposition de méthodes, proposé notamment par FLAVORS (III.3.4.3). En FLAVORS il est possible de décomposer une méthode en 3 parties, ou 3 sous-méthodes, **before**, **main** et **after**; les méthodes before et after étant respectivement appliquées avant et après la main-méthode. Cette décomposition est strictement réservée à l'implémentation de la méthode principale qui seule fait partie du protocole de l'objet. De ce point de vue la méthode valideAge précédente peut être considérée comme méthode before de age= , de même que la méthode calcule-sortie, after-méthod de e1= et e2=.

Cette décomposition à la FLAVORS est en fait une généralisation des réflexes avant/après , écriture/lecture des frames (III.4.4.2) qui sont les parties before et after respectives des méthodes d'écriture et de lecture d'attribut de frames.

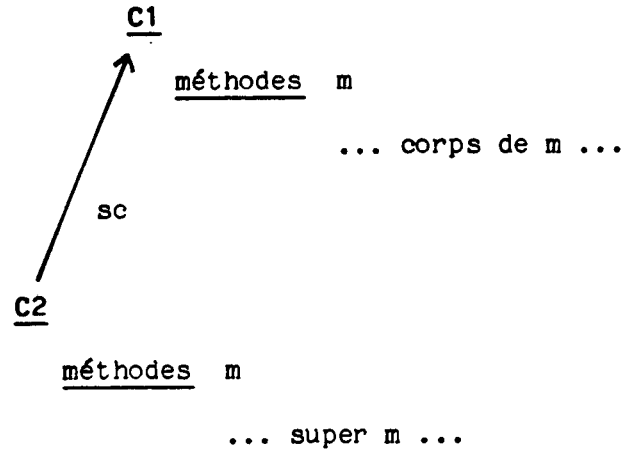
De façon similaire, ces méthodes sont réservées à l'implémentation des méthodes d'écriture/lecture qui seules font partie du protocole d'accès aux attributs. Ce fait peut être facilement constaté en, reprenant la définition des frames en temps qu'objets composites vus au III.4.4.2 et leur implémentation en SMALLTALK. Les méthodes (réflexes) "aprèsEcriture", "avantEcriture" et "siBesoin", définies par la classe ATTRIBUT, sont propres à self et n'interviennent que comme sous-méthode de "v:" (écriture) pour les deux premières et "v" (lecture) pour la dernière. Seules "v:" et "v" doivent faire partie du protocole d'un représentant de ATTRIBUT. D'où en ROME :

```
[R-CLASS 'new  'ATTRIBUT
 'surcl '(OBJECT)
 'attributs '(valeur defaut domaine frame)
 'methodes
  '(;writing
    v=
     (lambda (x)
      {avantEcriture x}
      {<- 'valeur x}
      {apresEcriture})
    ;reflexe avant ecriture
    avantEcriture
     (lambda (x)
      ;au minimum verification de l'appartenance au domaine
      (when (not (member x {? 'domaine}))
            {erreur x " n'appartient pas au domaine" {? 'domaine}}))
    ;reflexe apres ecriture, par defaut rien
    apresEcriture (lambda ())

    ;reading
    v
     (lambda ()
      (if (nequal {? 'valeur} '??) ;'?? = indetermine
          {? 'valeur}
          (let ((valcalculee {siBesoin}))
           (if (nequal valcalculee '??)
               valcalculee
               {? 'defaut}))))
    ;reflexe si besoin en lecture, par defaut l'indetermine
    siBesoin (lambda () '??))
 'selecteurs
  '(;seules les methodes d'acces peuvent faire l'objet de messages
    valeur= v=
    valeur? v)]
```

Enfin cette approche de la séparation méthodes encapsulées/sélecteurs externes permet de résoudre l'ambiguïté du SUPER que nous avons vu au III.3.4.2. Rappelons synthétiquement que SUPER est nécessaire (en substitution à SELF) quand il y a à la fois :

* redéfinition d'une méthode dans une sous-classe par masquage de celle héritée.

* référence pourtant à celle-ci pour la redéfinition.

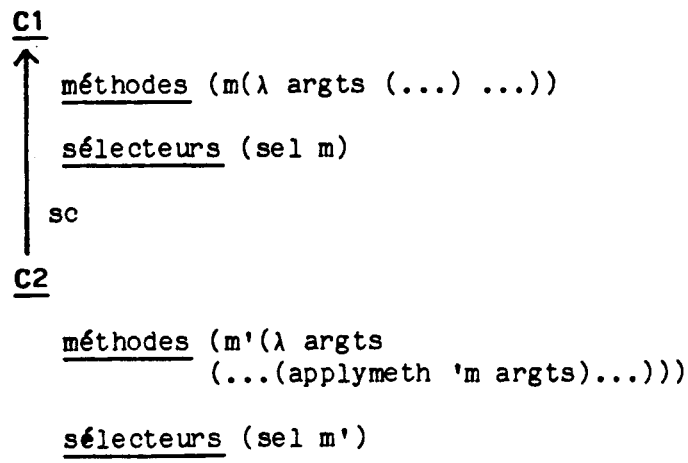


La nécessité du Super dans ce cas vient de l'association systématique méthode/sélecteur. La méthode héritée doit être à la fois :

* masquée pour l'extérieur (dans le protocole)

* non masquée pour l'objet lui-même, pour l'implémentation de la redéfinition.

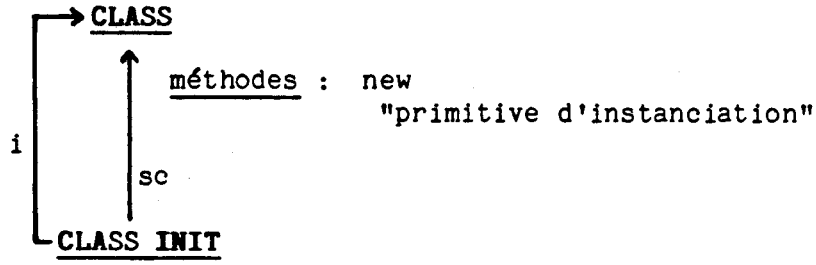
ROME qui, au contraire, permet la dissociation méthode/sélecteur propose la solution suivante :



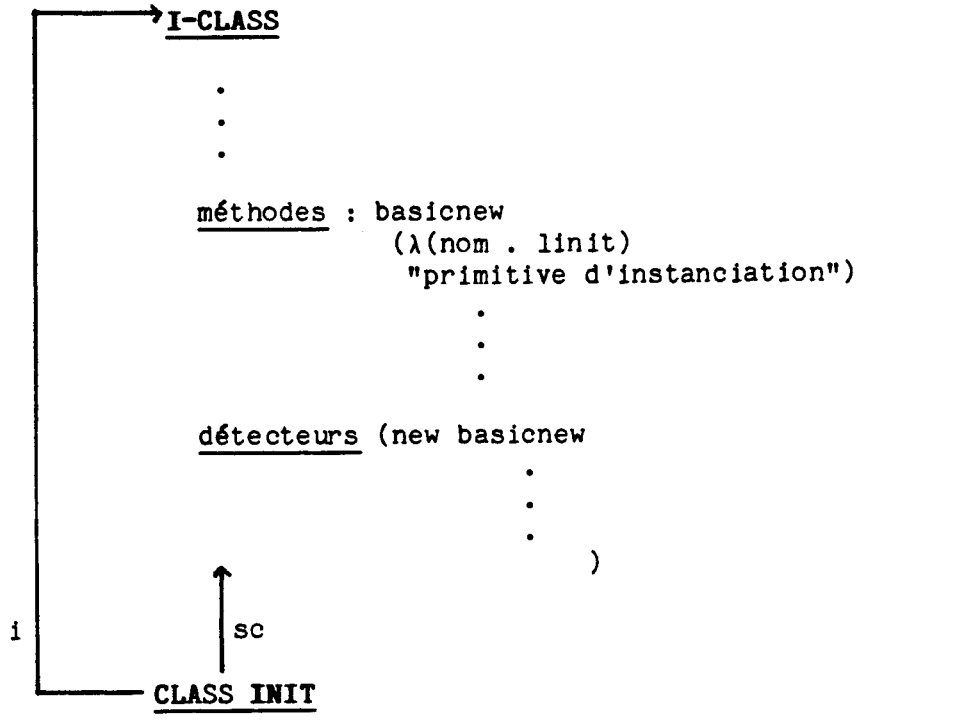
* l'encapsulation des méthodes garantit que m (qui classiquement devrait être masquée) ne peut être appliquée que par l'objet, notamment dans la définition de m' , sans l'artifice du SUPER.

* puisque le masquage n'intervient que pour l'extérieur, seul le sélecteur sel, dans le protocole, reçoit une nouvelle définition, en lui réassociant la méthode m'.

Prenons l'exemple classique de l'utilisation de SUPER pour la définition de nouvelles méthodes d'instanciation par redéfinition du new (III.3.4.2). Soit la métaclasse CLASS_INIT des classes qui initialisent leurs instances, par redéfinition du new primitif comme suit :



En ROME, puisque la méthode primitive new (basicnew de I-CLASS) est nécessaire pour l'implémentation de la nouvelle, elle n'est pas masquée du point de vue interne mais seulement du point de vue externe. On définit simplement dans CLASS_INIT une méthode d'instanciation avec initialisation qui fait appel à la primitive basicnew.




```
[I-CLASS 'new 'CLASS_INIT

  'surcl '(I_CLASS)
  'méthodes '(newinit(lambda ℓ [(applymeth 'basicnew ℓ)'init]))
  'sélecteur '(new newinit)]
```

IV.2.5.- Héritage multiple sans conflits

Par le fait qu'une classe peut-être sous-classe de plusieurs autres classes (l'attribut "surcl" de toute classe contient la liste de ses sur-classes, Cf. IV.2.1.2.), ROME permet l'héritage multiple de caractéristiques.

Précisons, tout d'abord, que l'affinement par substitution (masquage, III.3.2.1) ne porte que sur les caractéristiques de type méthode et sélecteur, pour lesquelles l'héritage de ROME assure la règle du plus affiné :

* méthode : redéfinition d'une méthode.

* sélecteur : association d'une nouvelle méthode à un sélecteur.

Exemple du 1er type :

- [re]définition de la méthode calcule-sortie de PORTE dans AND et OR (IV.2.2.1)

- redéfinition de valideAge de PERSONNE dans ENFANT, ADULTE ... (IV.2.2.3.2)

Exemple du 2ème type :

- (new basicnew) affiné en (new newinit) (IV.2.2.3.2)

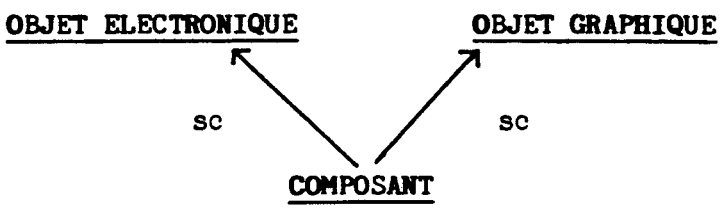
Il ne concerne pas cependant les attributs qui sont spécifiés simplement par leur nom, sans information associée (comme pour le modèle de base III.3.2.1).

Au paragraphe III.3.3. ont été présentées les 2 stratégies principales d'héritage multiple, linéaires ou graphiques, et leur comportement par rapport au problème de conflit. Nous allons montrer la technique d'héritage multiple en ROME et ses originalités par rapport aux techniques classiques.

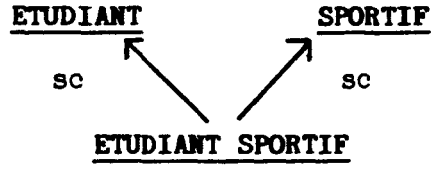
IV.2.5.1.- Point de vue : principe d'indépendance

Nous avons vu au paragraphe III.3.5 qu'en associant à la notion de classe, le concept de point de vue sur les objets qu'elle définit, l'héritage multiple permet de décrire un objet sous des points de vue différents, respectivement associés aux sur-classes de la classe de cet objet.

Ex (III.3.5.1)



Les surclasses OBJET_ELECTRONIQUE et OBJET_GRAPHIQUE correspondent respectivement aux descriptions des points de vue électronique et graphique d'un composant.



de façon similaire pour les classes Etudiant et Sportif par rapport à étudiant_sportif.

Tout comme les stratégies graphiques, l'héritage de ROME part du principe qu'il ne peut y avoir de conflits d'héritage multiple entre surclasses différentes, i.e. entre points de vue différents.

Ex : Supposons que les classes objet_électronique et Objet_graphique définissent toutes deux un jeu de coordonnées, de même nom x,y, respectivement représentant l'emplacement du composant sur le schéma et à l'affichage, ces deux jeux de caractéristiques étant distinctes, ils doivent être tous deux accessibles pour un composant, indépendamment du fait qu'elles aient les mêmes noms.

De façon similaire, un étudiant peut être caractérisé par un attribut numéro_de_carte, un sportif également, un étudiant sportif par les deux attributs.

Les stratégies graphiques offrent donc (plus ou moins) des mécanismes de sélection des caractéristiques (paramétrage de l'héritage par une classe) pour lever les conflits (*). Ces mécanismes sont importants pour permettre la spécification de points de vue différents de manière indépendante sans se soucier de conflits qui surviendraient ultérieurement lors de la combinaison (II.3.5.4.) des classes correspondantes en sous-classes. C'est pourquoi la solution syntaxique simple (quelle que soit la stratégie) qui consiste à nommer différemment les caractéristiques ne convient pas puisqu'elle suppose que lors du développement d'une classe, celle-ci sera susceptible d'entrer en conflit avec d'autres classes utilisées conjointement comme surclasses par combinaison.

(*) Nous ne reviendrons pas sur les solutions linéaires qui ne permettent d'hériter que d'une seule caractéristique selon un algorithme arbitraire.

Ceci est notamment insatisfaisant méthodologiquement vis à vis de la **conception incrémentale des classes** (III.3.5.2) selon laquelle chaque classe n'a connaissance que d'elle-même et de ses surclasses et décrit à son niveau un modèle complet (éventuellement partiellement abstrait) d'objet. Par exemple objet_graphique ne doit faire aucune supposition sur sa sous-classe composant et encore moins sur la classe objet_électronique.

Ceci introduit le principe d'indépendance sur lequel repose l'héritage multiple de ROME.

Définition : Deux classes sont indépendantes si elles ne sont pas sous-classes l'une de l'autre (Il n'existe pas de chemin d'héritage de l'une à l'autre).

Principe d'indépendance : Il n'y a pas de conflit d'héritage entre classes indépendantes.

Plus simplement, le but du principe d'indépendance est d'éliminer complétement les conflits d'héritage, puisque si deux classes ne sont pas indépendantes, par définition elles sont donc en relation d'affinement (il existe un chemin d'héritage entre elles) le "conflit" correspond alors à un affinement par substitution qui est "résolu" par la règle du plus affiné. Le principe d'indépendance est assuré en ROME grâce aux expressions de point de vue ou as-expressions.

IV.2.5.2.- Expressions de points de vue ou as-expressions.

Le principe d'indépendance vaut pour les trois types de caractéristiques, attributs, méthodes, sélecteurs. Le mécanisme de sélection s'exprime syntaxiquement sous la forme d'une **expression de point de vue**, ou **as-expression**

(nom_de_caractéristique as une_classe)

Selon le type de caractéristique, nous parlerons de as-expression d'attribut, de méthode ou de sélecteur. Intuitivement, une as-expression exprime sous quel point de vue associé à une_classe (as une_classe) il faut considérer la caractéristique de nom nom_de_caractéristique.

Précisons formellement la notion de point de vue associé à une classe.

Définition :

* Treillis d'héritage

Soit **C** l'ensemble des classes (l'ensemble des représentants de R-CLASS) et $\leq$ la relation d'ordre partiel classe/surclasse, **C** est partiellement ordonné avec un élément maximum, OBJECT. Par abus de langage, nous appellerons treillis d'héritage (*) la structure $(C, \leq)$.

* Dépendance d'une classe

Soit $C \in \mathbf{C}$

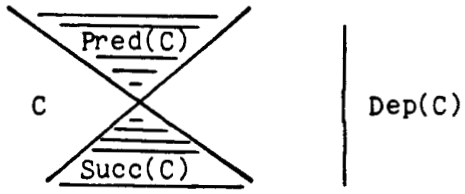
$Pred(C) = \{C' \in \mathbf{C} \mid C' \leq C\}$

$Succ(C) = \{C' \in \mathbf{C} \mid C' \geq C\}$

alors la dépendance de C est :

$Dep : \mathbf{C} \rightarrow \mathbf{P}(C)$

$Dep(C) = Pred(C) \cup Succ(C)$.

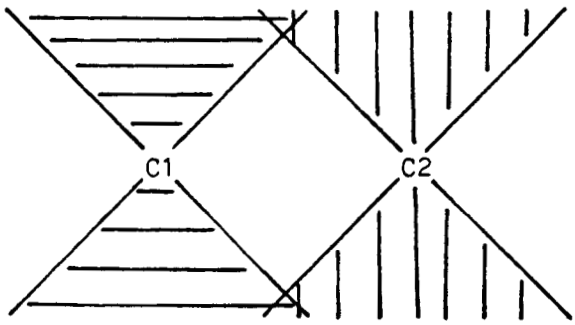


* Indépendance

Soient C1 et C2 $\in \mathbf{C}$

C1 est indépendance de C2

ssi C1 (resp. C2) $\notin$ Dep(C2) (resp. Dep(C1))



(*) Cf. III.3.1.

* Classes de représentation d'un objet

Soit **O** l'ensemble des objets (l'ensemble des représentants de OBJECT), soit l'application suivante :

$$i : \mathbf{O} \rightarrow \mathbf{C}$$

$i(O)$ = classe d'instanciation de O

alors l'ensemble des classes de représentation de O est caractérisé par l'application :

$$Rep : \mathbf{O} \rightarrow \mathbf{P}(\mathbf{C})$$

$$Rep(O) = \{C \in \mathbf{C} \mid C \geq i(O)\}$$

* Treillis de représentation d'un objet
(ou, par abus de langage, treillis d'héritage d'un objet)

Restriction du treillis d'héritage à $Rep(O)$

= $(Rep(O) , \leq)$

* Dépendance d'une classe par rapport à un objet

Soit $C \in Rep(O)$

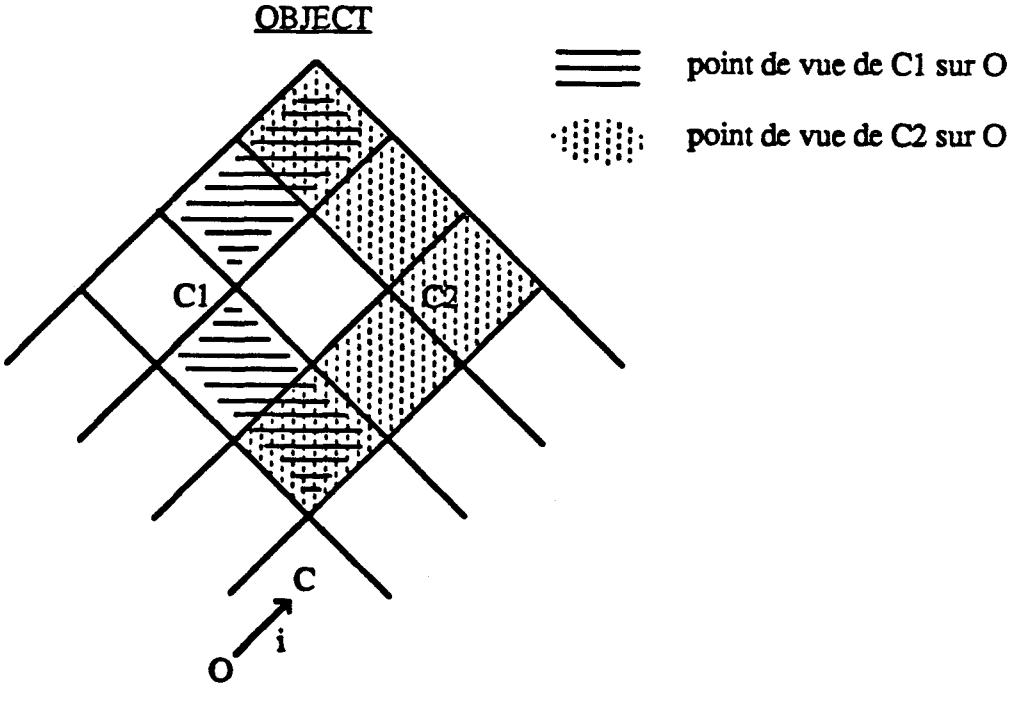
$Dep(C,O) = Dep(C) \cap Rep(O)$

* **Point de vue d'une classe sur un objet**

Soit $C \in Rep(O)$

c'est la restriction du treillis de représentation à $Dep(C,O)$

= $(Dep(C,O) , \leq)$ = pdv (C,O)



* Points de vue indépendants

Soient C1 , C2 ∈ Rep(O) , les points de vue de C1 et C2 sont indépendants si C1 et C2 sont indépendantes.

C'est le cas sur la figure précédente.
Remarquons qu'il est immédiat que :

$$Rep(O) = \bigcup \{Dep(C,O) \mid C \geq i(O)\}$$

ce qui s'interprète par le fait qu'un objet est connu sous tous ses points de vue.

* Point de vue plus affiné / plus général

Soient C1 , C2 ∈ Rep(O) , C1 ≥ C2 , le point de vue de C1 (resp. C2) sur O est alors dit plus général (resp. plus affiné) que celui de C2 (resp. C1). Ceci selon l'interprétation de la relation classe/sur-classe (≤) comme une relation de généralisation/affinement (III.3.1).

* Point de vue plus restreint / plus global

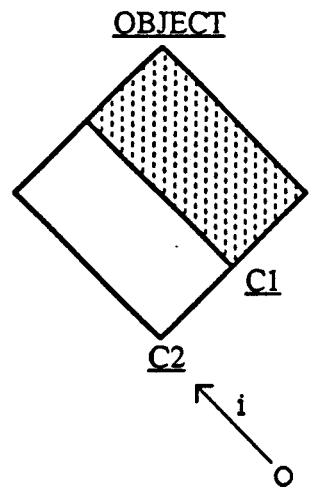
Soient C1 , C2 ∈ Rep(O) , le point de vue de C1 (resp. C2) sur O est plus restreint (resp. plus global) que celui de C 2 (resp. C1).

ssi $Dep(C1,O) \subseteq Dep(C2,O)$

Remarquons que les relations plus affiné/plus général et plus restreint/plus global sont dissociées comme sur la figure suivante. En particulier :

C2 < C1 => pdv (C2,O) plus affiné que pdv (C1,O)

⇏ pdv (C2,O) plus restreint que pdv (C1,O)



Deux points de vue ne sont pas forcément comparables par rapport à ces deux relations.

Intersection de points de vue (ou point de vue composé).

Soit C1, C2 ∈ Rep(O)

pdv (C1 . C2 , O)

= (Dep(C1,O) ∩ Dep(C2,O) , ≤)

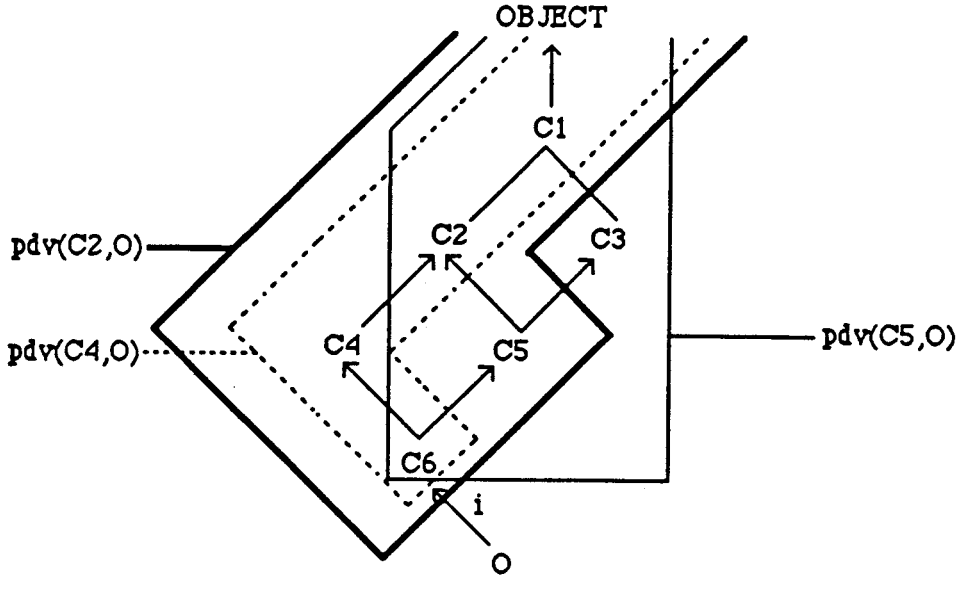
Remarquons qu'une intersection de points de vue est toujours plus restreinte que chacun de ces points de vue.

Propriété : Soient C1,C2 ∈ Rep(O)
C1 < C2
=> pdv(C1.C2,O) = (Succ(C1) ∪ [C1 C2] ∪ Pred(C2))
∩ Rep(O)

d'après les définitions :

pdv(C1.C2,O) = Dep(C1,O) ∩ Dep(C2,O)
= (Succ(C1) ∪ Pred(C1)) ∩ (Succ(C2) ∪ Pred(C2)) ∩ Rep(O)
= (Succ(C1) ∩ Succ(C2)
∪ Succ(C1) ∩ Pred(C2)
∪ Pred(C1) ∩ Succ(C2)
∪ Pred(C1) ∩ Pred(C2)) ∩ Rep(O)
C1 < C2 =>
= (Succ(C1) ∪ ∅ ∪ [C1 C2] ∪ Pred(C2)) ∩ Rep(O) **c.q.f.d.**

Illustrons ces définitions sur l'exemple simple et informel suivant :

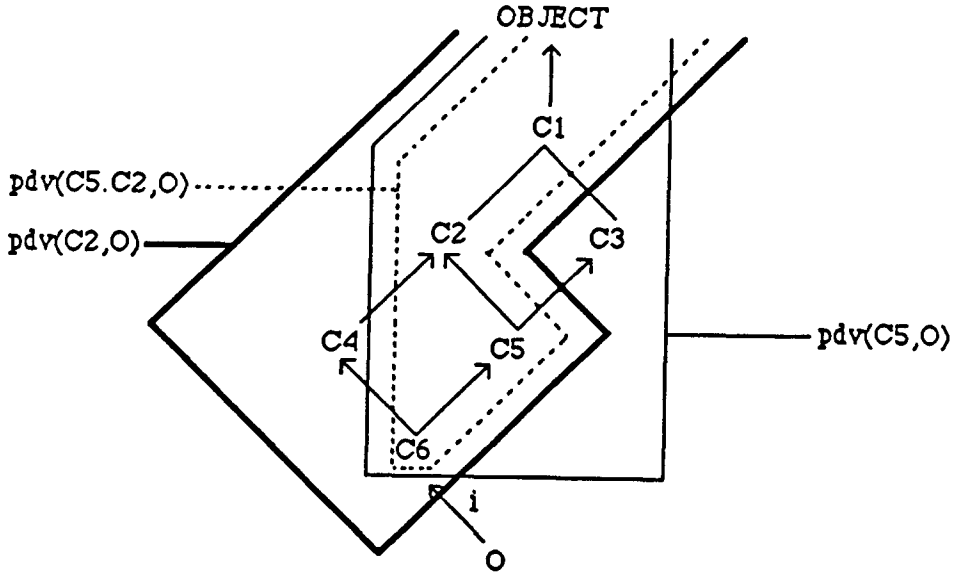


C2 et C4 sont indépendantes de C3

C4 est indépendante de C5.

Soit O une instance de C6, nous avons figuré les points de vue de C4 sur O et de C5 sur O , qui sont indépendants. Le point de vue de C2 sur O est plus général et plus global que celui de C4 sur O. Par contre, pdv(C2,O) est plus général mais incomparable à pdv(C5,O).

Par intersection, cette fois le point de vue de C2 et C5 sur O est plus restreint que ceux de C2 et de C5 :



Les points de vue vont permettre de restreindre le treillis d'héritage d'un objet. Soit un objet O représentant d'une classe C ($C \in Rep(O)$), l'expression de point de vue :

(nom_caractéristique as C)

permet de considérer la caractéristique de O de nom nom_caractéristique définie sous le point de vue de C sur O. L'héritage de ROME est dynamique (III.3.3) sur le treillis, les expressions de points de vue contraignent l'algorithme de recherche de base, appelé lookup, à se confiner sous un point de vue, restriction du treillis. A la base la stratégie du lookup est linéaire en profondeur d'abord avec prise en compte en priorité de l'affinement (règle du plus affiné, III.3.3)

L'expression de points de vue permet donc implicitement de pallier les défauts de cette stratégie en ce qui concerne les conflits entre classes indépendantes, en restreignant la partie du treillis observée.

Les as-expressioons ne permettent actuellement que l'expression d'un point de vue simple, celui d'une classe, notamment pas de point de vue composé explicitement (*). Le point de vue considéré n'est cependant pas systématiquement simple au sens précédent, du fait d'un mécanisme d'inférence implicite de points de vue basé sur leur composition. Nous pouvons dire que les expressions de points de vue sont syntaxiquement simples, dans leur écriture, mais dynamiquement composées, dans leur évaluation.

Précisons leur sémantique selon le type de caractéristique.

IV.2.5.3.- As-expressions de sélecteurs

Nous avons vu, au paragraphe IV.2.1. qu'une définition de sélecteur par une classe associe à celui-ci une méthode :

définition de sélecteur par une classe C : (sélecteur méthode)

Il convient ici de compléter ceci comme suit :

Une définition de sélecteur associe à un sélecteur une spécification de méthode sous la forme :

(*) Ceci pourra faire l'objet de travaux ultérieurs.

C: (sélecteur spécification_de_méthode)

où spécification_de_méthode peut être simplement un nom de méthode :

forme implicite C: (sélecteur méthode)

ou une as_expression de méthode :

forme explicite : C:(sélecteur (méthode as C')) C' ≥ C

C' devant être une sur-classe de C , en respect de la conception incrémentale des classes selon laquelle C ne connaît que ses sur-classes et ne peut donc faire référence qu'à celles-ci dans les as-expressions.

La deuxième forme va permettre de spécifier explicitement le point de vue (C') sous lequel il faut associer méthode à sélecteur pour les représentants de C .

Une définition de sélecteur de la première forme (sélecteur méthode) par une classe C est implicitement transformée en définition de seconde forme comme suit par le mécanisme d'inférence de point de vue, dont c'est le premier aspect :

forme implicite C: (sélecteur méthode)

transformé automatiquement en

forme explicite C: (sélecteur(méthode as C))

ce qui peut s'interpréter par le fait que le point de vue par défaut est celui de la classe de définition si celle-ci ne spécifie pas explicitement de point de vue de référence.

Nour dirons qu'il s'agit d'une transformation par as implicite.

Les as-expressions de sélecteur interviennent dans l'expression de points de vue de l'extérieur sur un objet à travers l'envoi de message

[O '(sélecteur as C) argts]

où C doit être une classe de représentation de O . La définition du sélecteur est alors recherchée sous le point de vue de C sur O et fournit à l'objet la spécification de méthode associée à appliquer (applymeth). Au paragraphe IV.2.1.1. a été montré le lien entre l'envoi de message et l'application de méthode d'après la transformation MR qui doit être complétée ici :

transformation MR

message : [O '(sélecteur as C) argts]

=> réaction : {as_méthode argts}

où as-méthode est la as-expression de méthode associée à sélecteur dans la classe de représentation de O , soit C', où est trouvée par lookup la définition du sélecteur sous le point de vue de C.

Dans le cas de as implicite, la définition de sélecteur inférée est donc :

C' : (sélecteur (méthode as C'))

c'est-à-dire, pour MR :

as-méthode = (méthode as C').

ainsi la transformation MR se réduit à :

transformation MR dans le cas de as implicite

message : [O '(sélecteur as C) argts]

=> réaction : {(méthode as C') argts}

où C' est la classe de représentation de O où est trouvée par lookup la définition du sélecteur, (sélecteur méthode), sous le point de vue de C.

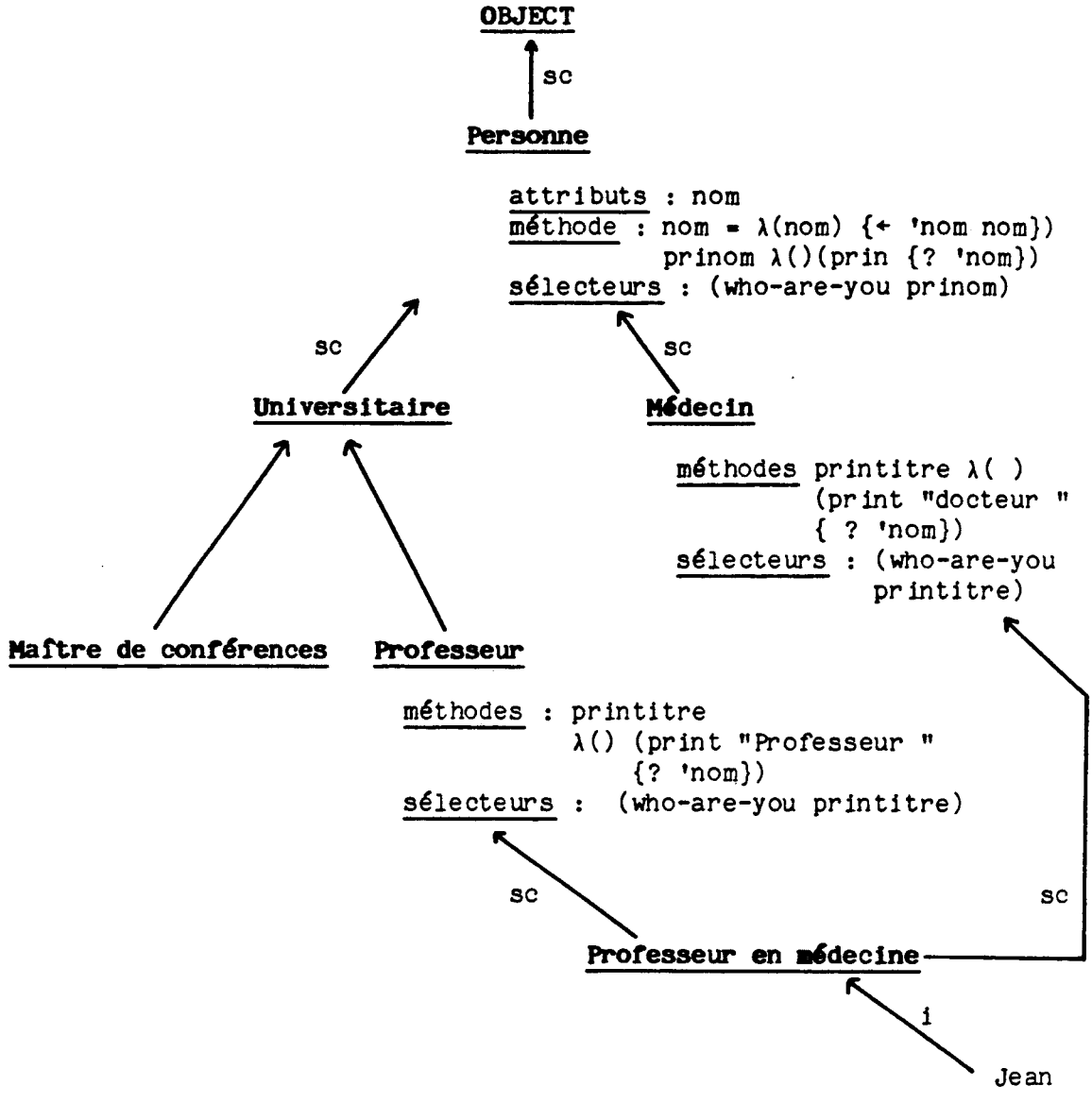
Nous détaillerons au paragraphe suivant le mécanisme général d'évaluation des as-expressions de méthode . A ce niveau, précisons simplement que l'énoncé par O :

{(méthode as C') argts}

provoque l'application de la méthode trouvée par lookup sous le point de vue de C' sur O , soit C" la classe où est trouvée cette méthode, on la notera :

C" : méthode

Exemple : Considérons le treillis d'héritage partiel suivant :


OBJECT
sc
Personne
attributs : nom
méthode : nom = λ(nom) {← 'nom nom})
prinom λ()(prin {? 'nom})
sélecteurs : (who-are-you prinom)
sc
sc
Universitaire
Médecin
méthodes printitre λ( )
(print "docteur "
{ ? 'nom})
sélecteurs : (who-are-you
printitre)
Maître de conférences
Professeur
méthodes : printitre
λ() (print "Professeur "
{? 'nom})
sélecteurs : (who-are-you printitre)
sc
sc
Professeur en médecine
i
Jean

traduit en ROME comme suit :

```
[I-CLASS 'new 'Personne
 'surcl '(OBJECT)
 'attributs '(nom)
 'methodes
  '(nom= (lambda (nom) [<- 'nom nom])
    prinom (lambda () (prin [? 'nom])))
 'selecteurs '(who-are-you prinom)]

[R-CLASS 'new 'Universitaire 'surcl '(Personne)]

[I-CLASS 'new 'Professeur
 'surcl '(Universitaire)
 'methodes '(printitre (lambda () (print "Professeur " [? 'nom])));**
 'selecteurs '(who-are-you printitre)]

[I-CLASS 'new 'Medecin
 'surcl '(Personne)
 'methodes '(printitre (lambda () (print "Docteur " [? 'nom])));**
 'selecteurs '(who-are-you printitre)]

[I-CLASS 'new 'Professeur_en_medecine
 'surcl '(Professeur Medecin)]

;Jean Martin est professeur en medecine:

[Professeur_en_medecine 'new 'jean 'nom "Jean Martin"]
```

Jean Martin est Professeur en médecine :

```
[Professeur_en_medecine 'new 'jean 'nom "Jean Martin"] ; (*)
```

Nous observons qu'il existe un conflit d'héritage (volontaire) sur le sélecteur who-are-you pour les représentants de Professeur_en_medecine, tel que l'objet jean.

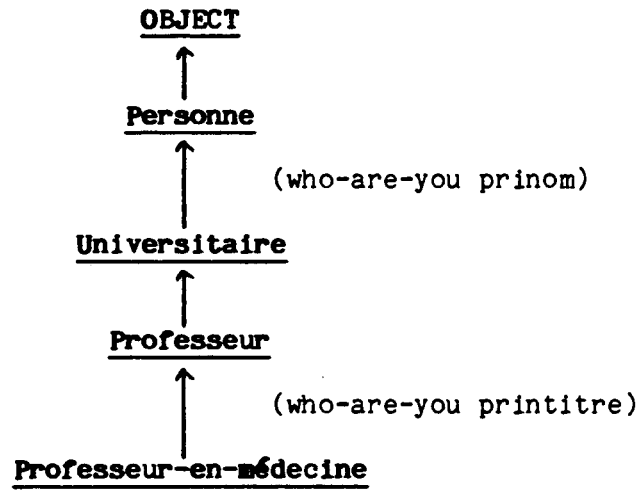
(*) l'affectation d'un attribut à l'instanciation (initialisation de l'objet) est possible si une méthode de même nom suffixée par "=" est définie, c'est le cas pour nom de Personne, de façon similaire à 'surcl , 'attributs, 'méthodes, 'sélecteurs de R_CLASS pour une classe.

(**) équivalent à :
printitre (λ()(prin "Professeur ") {prinom} (terpri))
et de façon similaire pour Médecin.

Une as-expression du sélecteur who-are-you permet de demander à ces objets de se comporter en tant que Professeur ou en tant que Médecin :

```
> [Jean '(who-are-you as Professeur)]

  Professeur Jean Martin

> [Jean '(who-are-you as Médecin)]

  Docteur Jean Martin
```

Considérons le premier exemple, le point de vue de Professeur sur les représentants de Professeur-en-Médecine est :



C'est alors who-are-you tel que défini par Professeur qui est pris en compte selon la règle du plus affiné. Ceci provoque l'application de la méthode (printitre as Professeur) par as implicite et la transformation MR.

```
[Jean '(who-are-you as Professeur)]

MR

{(printitre as Professeur)}

lookup

Professeur : printitre.
```

Il en est de même si le point de vue spécifié est celui de la classe universitaire :

```
> [Jean '(who-are-you as Universitaire)]

  Professeur Jean Martin
```

qui permet de spécifier le point de vue Universitaire par opposition à celui de Médecin.

Le point de vue de Universitaire sur les représentants de Professeur-en-Médecine est le même que celui de professeur, ce qui provoque le même effet.

Sous le point de vue Universitaire la définition de who-are-you la plus affinée est celle de Professeur. Nous verrons au paragraphe IV.2.5.5. que ceci conduit à une propriété des as-expressions de ROME qui est la conservation de l'affinement par les points de vue et qui est l'une des différences essentielles de la sémantique du as par rapport aux mécanismes de sélection des autres stratégies graphiques, tels les sélecteurs composés de extended SMALLTALK (III.3.3.3 et III.3.4.3).

Insistons sur l'importance du as-implicite assuré par le mécanisme d'inférence des points de vue pour la cohérence de cette notion.

Montrons pour cela qu'il ne suffit pas, dans MR, d'appliquer la spécification de méthode associée au sélecteur dans C', ce qui dans le cas de as-implicite ménerait à la transformation MR1 :

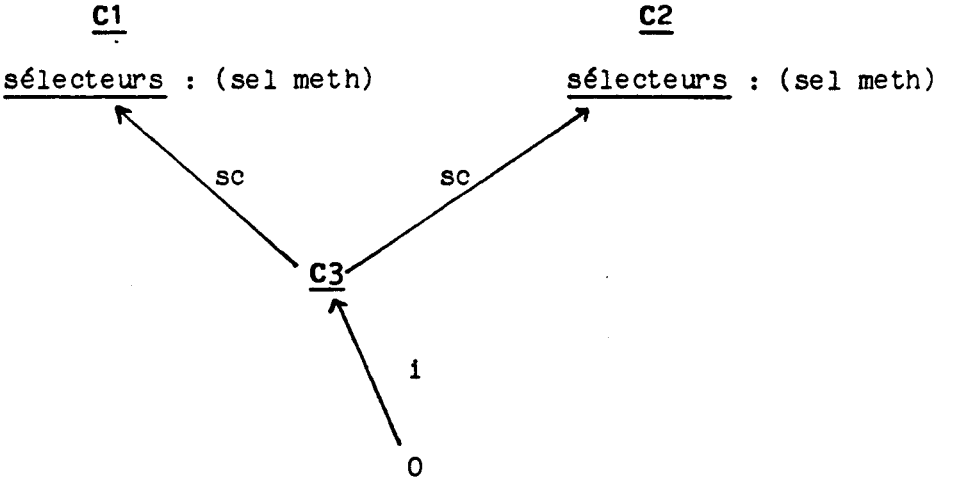
MR1

message : [O '(sélecteur as C) argts]

=> réaction : {méthode argts}

où (sélecteur méthode) est trouvée dans C' sous le point de vue de C.

Cette transformation ne convient pas du fait du lookup de base :



Le message suivant provoquerait le bon effet :

$$[O\ '(sel\ as\ C1)] \Rightarrow \{meth\} \overset{lookup}{\Rightarrow} \underline{C1 : meth}$$

contrairement à :

$$[O\ '(sel\ as\ C2)] \Rightarrow \{meth\} \overset{lookup}{\Rightarrow} \underline{C1 : meth}$$

alors que les points de vue de C1 et C2 sont indépendants. Ce problème apparaîtrait par exemple pour le message :

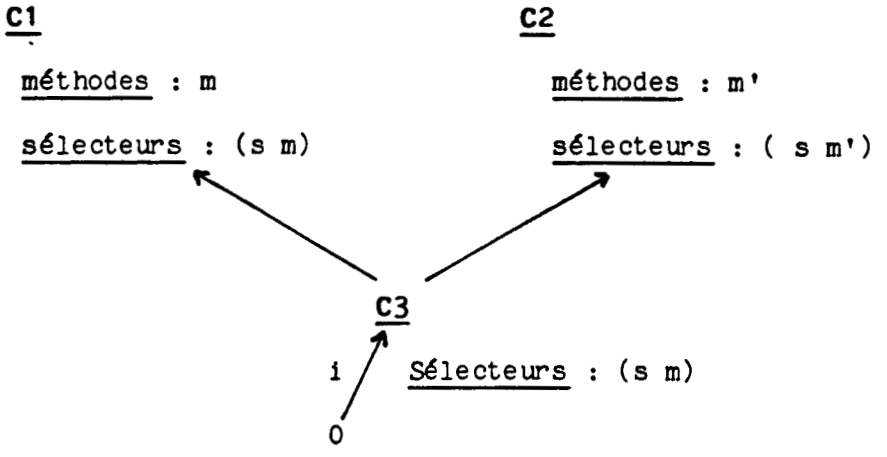
$$[Jean\ '(who\text{-}are\text{-}you\ as\ Médecin)] \overset{MR1}{\Rightarrow} \{printitre\} \overset{lookup}{\Rightarrow} \underline{Professeur:printitre}$$

Dans les deux cas, le lookup trouve meth dans C1 (c'est tout le problème des stratégies linéaires). C'est pourquoi il faut inférer par as implicite le point de vue de la classe où est trouvée la définition de sélecteur par le lookup de la méthode associée.

Une autre idée serait d'inférer simplement le point de vue associé au sélecteur dans le message pour la méthode, conduisant à la transformation MR2 , sans as implicite :

MR2 :
message : [O '(sélecteur as C) atgts]
=> réaction : {(méthode as C) argts}
où (sélecteur méthode) est trouvée dans C' sous le point de vue de C.

Le problème ici est directement lié à l'affinement par combinaison (III.5.4.). Bien que des classes soient indépendantes entre elles (comme Professeur et Médecin), i.e. qu'elles ne "voient" pas leurs caractéristiques mutuelles, une sous-classe commune par combinaison (comme Professeur_en_medecine) "voit" quant à elle toutes leurs caractéristiques et peut donc y faire référence par héritage sous son point de vue :



C3 hérite de m et m' , la définition de sélecteur (s m) dans C3 signifie que dans tous les cas, pour ses représentants, c'est la méthode m de C1 qu'il faut appliquer en réponse à un message de sélecteur s .

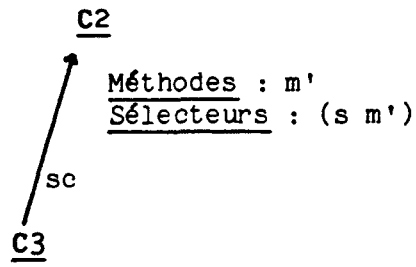
Or, dans ce cas, la transformation MR2 ne permettrait pas cet effet

$$[0\ '(s\ as\ C1] \xrightarrow{MR2} \{(m\ as\ C1)\} \xrightarrow{lookup} \underline{C1:m}$$

ce qui est satisfaisant mais :

$$[0\ '(s\ as\ C2)] \xrightarrow{MR2} \{(m\ as\ C2)\} \xrightarrow{lookup} \emptyset$$

Le lookup ne trouve pas de méthode m sous le point de vue de C2 qui est :



contrairement à MR :

$$[0\ '(s\ as\ C2)] \xrightarrow{MR} \{(m\ as\ C3)\} \xrightarrow{lookup} C1 : m$$

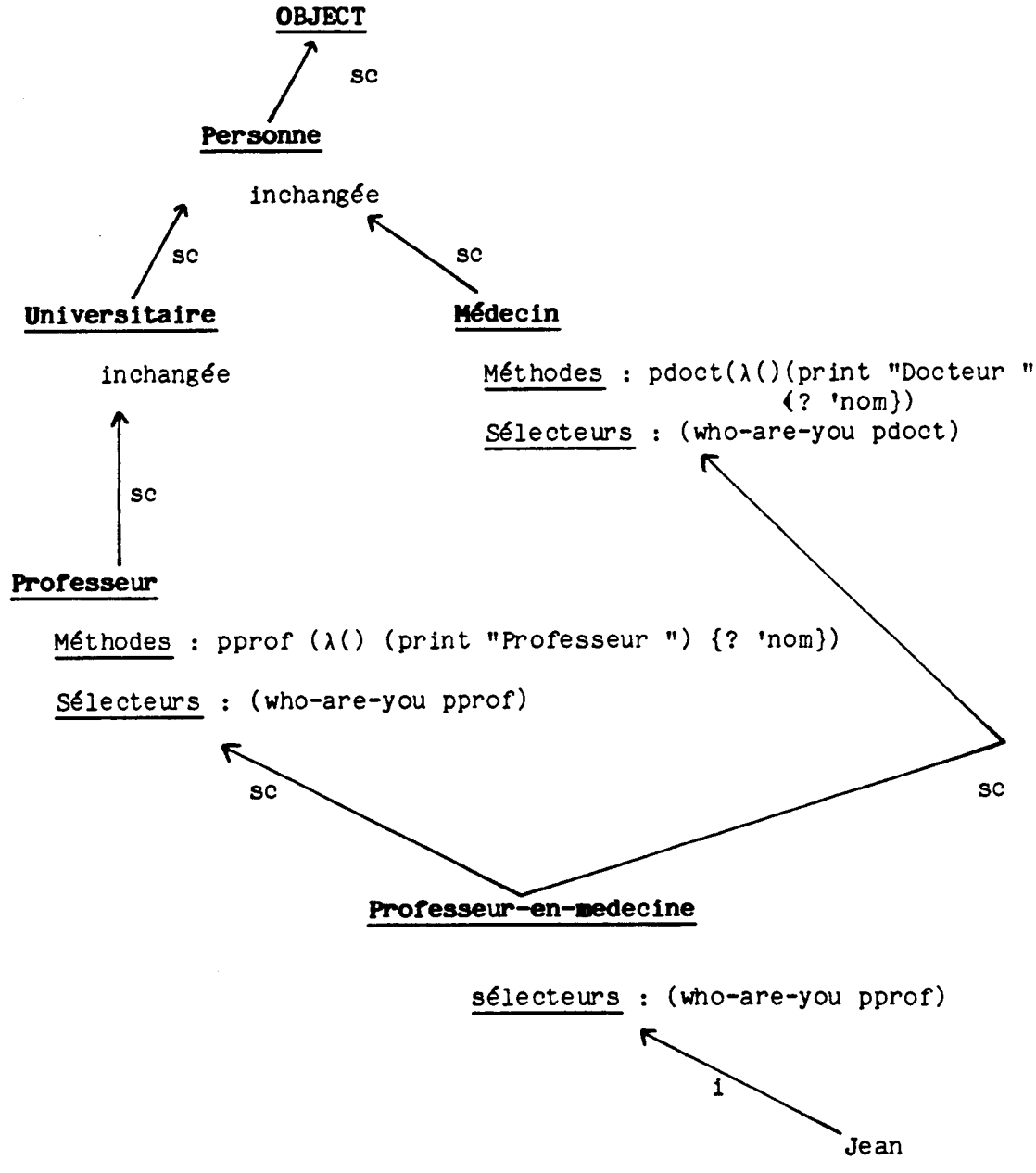
puisque le point de vue de C3 est le treillis initial.

Cette technique trouve son interprétation assez simple dans le fait que sous le point de vue de C2 (pdv initial spécifié dans le message) le point de vue définissant le sélecteur s le plus affiné est celui de C3. C'est donc celui-ci qu'il faut considérer pour la réponse sachant qu'il est en fait plus global à cause de la combinaison. On dira qu'il y a ici **relâchement de point de vue** :

> Dans la transformation MR, il y a relâchement de point de vue si le point de vue de C' est plus global que celui de C.

Par inférence de C3 à partir de C2 dans MR; on agrandit le treillis de représentation de O sur lequel il faut inférer ses caractéristiques. Nous verrons que le relâchement de point de vue intervient plus généralement pour l'exécution des as-expressions de méthodes.

Donnons ici un exemple de relâchement en considérant à nouveau l'exemple précédent développé différemment selon le modèle exact de C1, C2, C3 précédent (en fait sans conflit sur la méthode printitre).



Le titre de docteur est inhibé par celui de professeur pour les professeur-en-médecine (!), Ainsi :

```
[Jean '(who-are-you as Médecin)]

MR
=> {(pprof as Professeur-en-medecine)}

lookup
=>      Professeur : pprof
```

ce qui est l'effet souhaité.

Précisons ici l'intérêt du as explicite. Remarquons que dans le problème précédent les méthodes associées à s sous les deux points de vue de C1 et C2 n'ont pas le même nom, respectivement m et m'. Ceci permet à C3 de redéfinir s en lui associant m sans ambiguité. En fait il s'agit plus généralement pour C3 de faire référence à une caractéristique, ici une méthode, définie sous le point de vue de C1 du genre :

C3 : (sélecteur (méthode as C1))

c'est la possibilité offerte par la forme explicite de définition de sélecteur. Nous pouvons ainsi reprendre l'exemple à partir de son écriture initiale, c'est-à-dire avec conflit sur printitre et la modification de Professeur-en-medecine suivante, relative au dernier problème :

Professeur-en-medecine

sélecteur : (who-are-you(printitre as Professeur))

Ainsi, comme précédemment :

```
[Jean ((who-are-you as Médecin)]

  MR
  => {(printitre as Professeur)]

 lookup
  =>      Professeur : printitre
```

ce qui est l'effet souhaité. Ceci est un exemple de la transformation MR avec as explicite qui permet dans une sous-classe de faire référence explicitement à un point de vue d'une sur-classe.

Notons ici que sur cet exemple particulier le as explicite semble superflu si l'on se repose sur le lookup de base, qui trouvera de toute manière printitre dans Professeur :

Professeur-en-medecine

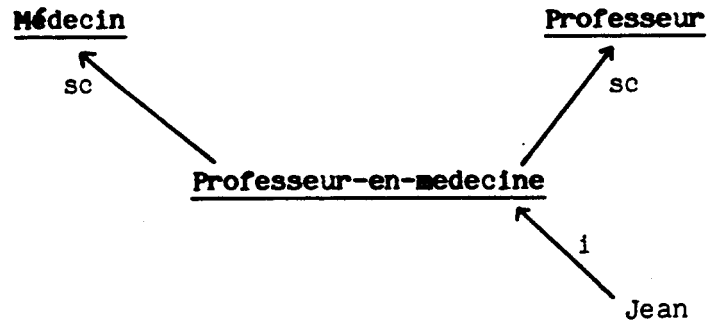
sélecteurs : (who-are-you printitre) (*)

[Jean '(who-are-you as Médecin)]

MR
=> {(printitre as Professeur-en-medecine)}
(as implicite et relâchement)

lookup
=> Professeur : printitre

Le problème est ici que le lookup (linéaire) n'est pas symétrique par rapport à l'ordre des sur-classes (III.3.3). Il suffit pour s'en convaincre d'inverser l'ordre des sur-classes, dont les définitions restent identiques :



Dans ce cas :

[Jean '(who-are-you as Médecin)]

MR
=> {(printitre as Professeur-en-medecine)}

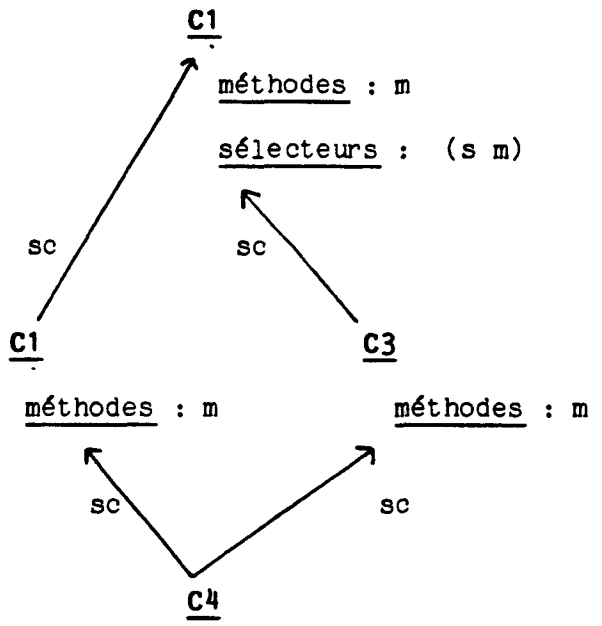
lookup
=> Médecin : printitre, ce qui n'est pas l'effet souhaité.

(*) Cette redéfinition qui peut paraître redondante par rapport à Professeur et Médecin est cependant obligatoire pour signifier que l'on fait référence à (printitre as Professeur-en-medecine) par as implicite, ce qui provoque le relâchement par MR.

Le as explicite est ici requis. De manière générale quand il y a conflit d'héritage il est conseillé de lever l'ambiguïté par as expression, c'est son rôle, sans s'appuyer sur le lookup de base. La définition d'une classe en est d'autant plus claire et symétrique par rapport aux points de vue indépendants de ses sur-classes.

Revenons maintenant sur le relâchement pour montrer que celui-ci ne peut être systématique, ce qui introduit la technique opposée de **contrainte de point de vue**.

Dans la transformation MR, le relâchement systématique revient à toujours considérer le point de vue de C' sur l'objet. Montrons que ceci n'est pas cohérent, sur le treillis suivant :



C2 et C3 sont indépendantes et redéfinissent chacune la méthode m de leur surclasse commune C1. les redéfinitions respectives sont cependant indépendantes (C2 et C3 ne se "connaissent" pas) et pour les représentants de C4 , tels que O , elles sont toutes deux les méthodes m les plus affinées selon les deux points de vue de C2 et C3. Ainsi, les messages suivants :

[O '(s as C2)] et [O '(s as C3)]

doivent mener respectivement à l'application de

C2 : m et C3 : m

puisque ce sont les méthodes m les plus affinées sous les points de vue respectifs.

Ce n'est cependant pas le cas pour le deuxième message, si l'on considère MR stricto sensu, i.e. avec relâchement systématique :

$$[O\ '(s\ \text{as}\ C3)] \xrightarrow{MR} \{(m\ \text{as}\ C1)\} \xrightarrow{lookup} \underline{C2 : m}$$

Le point de vue de C1 considéré pour m étant le treillis initial (relâché par rapport à celui de C3, et plus global) le lookup de base (qui ne tient pas compte des indépendances) trouve la définition de m dans C2.

Il faut donc contraindre le lookup à rester sous le point de vue le plus affiné précédent, c'est-à-dire initial ici (C3 spécifié dans l'envoi de message). La solution adoptée, de façon homogène avec l'inférence des points de vue pour l'évaluation de as-expressions de méthode, est de contraindre le lookup à l'intersection des points de vue de C , initial, et C' où est trouvée la définition de sélecteur (sous le point de vue de C) et non pas du point de vue simple de C' , par relâchement systématique :

MR
message : [O '(sélecteur as C) argts]

- as implicite :
  - {(méthode as C') argts} si C' ≤ C , relâchement
  - {(méthode as C.C') argts} , si C' > C contrainte
- as explicite : {(méthode as C") argts}, C" ≥ C'

où C' est la classe de représentation de O où est trouvée la définition de sélecteur (sélecteur as-méthode) la plus affinée sous le point de vue de C.

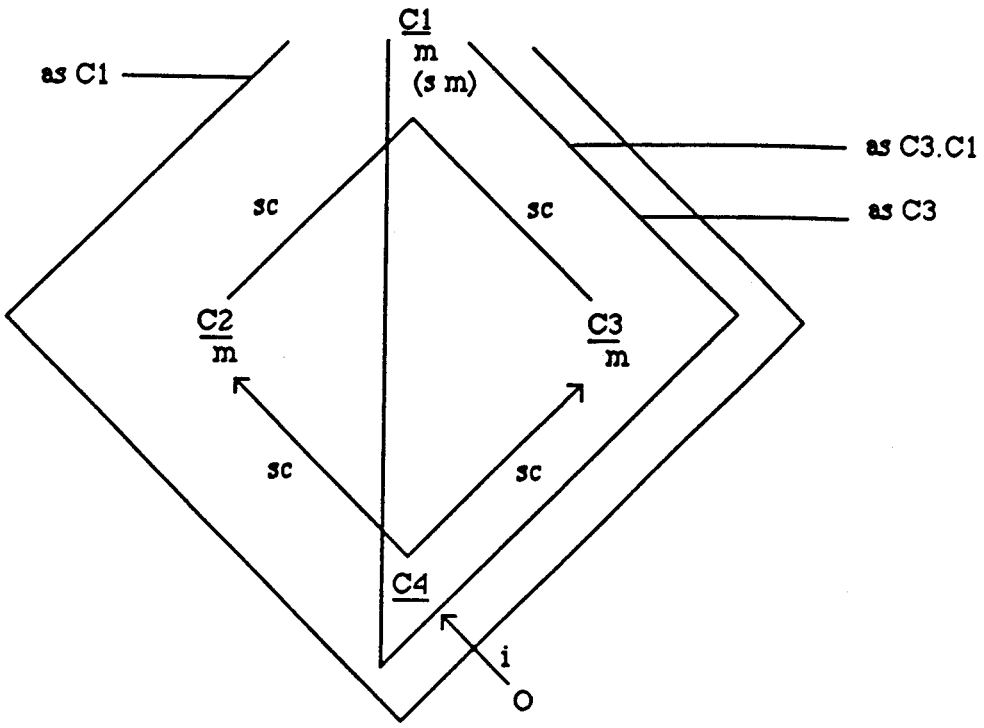
(méthode as C.C') signifiant de considérer l'intersection des points de vue de C et C' pour le lookup de méthode.

Nous obtenons alors :

$$[O\ '(s\ \text{as}\ C2)] \xrightarrow{MR} \{(m\ \text{as}\ C2.C1)\ \text{argts}\} \xrightarrow{lookup} C2 : M$$

$$[O\ '(s\ \text{as}\ C3)] \xrightarrow{MR} \{(m\ \text{as}\ C3.C1)\ \text{argts}\} \xrightarrow{lookup} C3 : M$$

ce qui est l'effet souhaité. Détaillons la deuxième transformation (la première est analogue) :



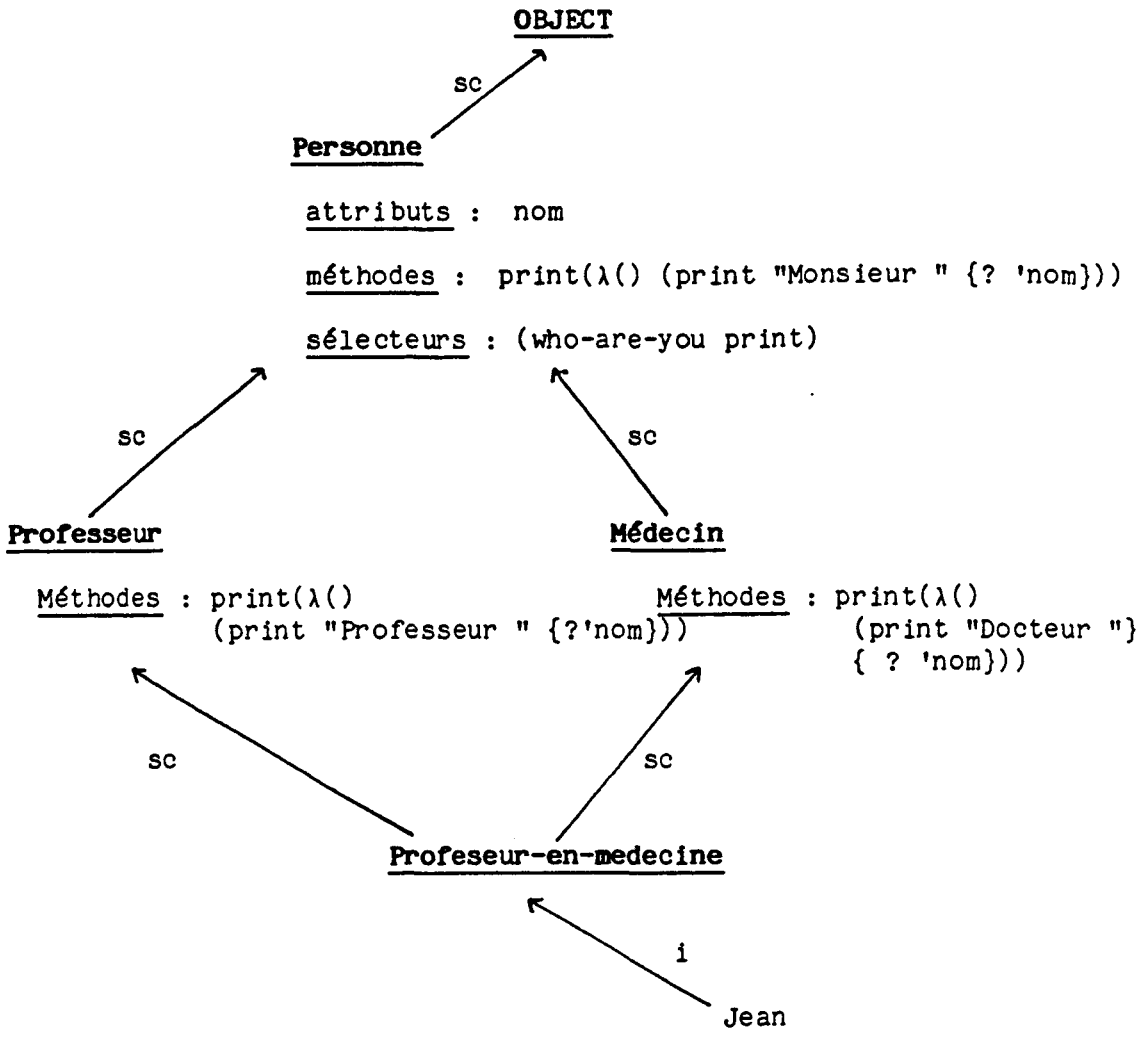
[O '(s as C3)] , s est trouvée dans C1 qui est sur-classe de C3, d'où

{(m as C3.C1)} , m est trouvée dans C3.

Donnons un exemple en reconsidérant une fois de plus le précédent avec un développement selon ce dernier modèle (*)

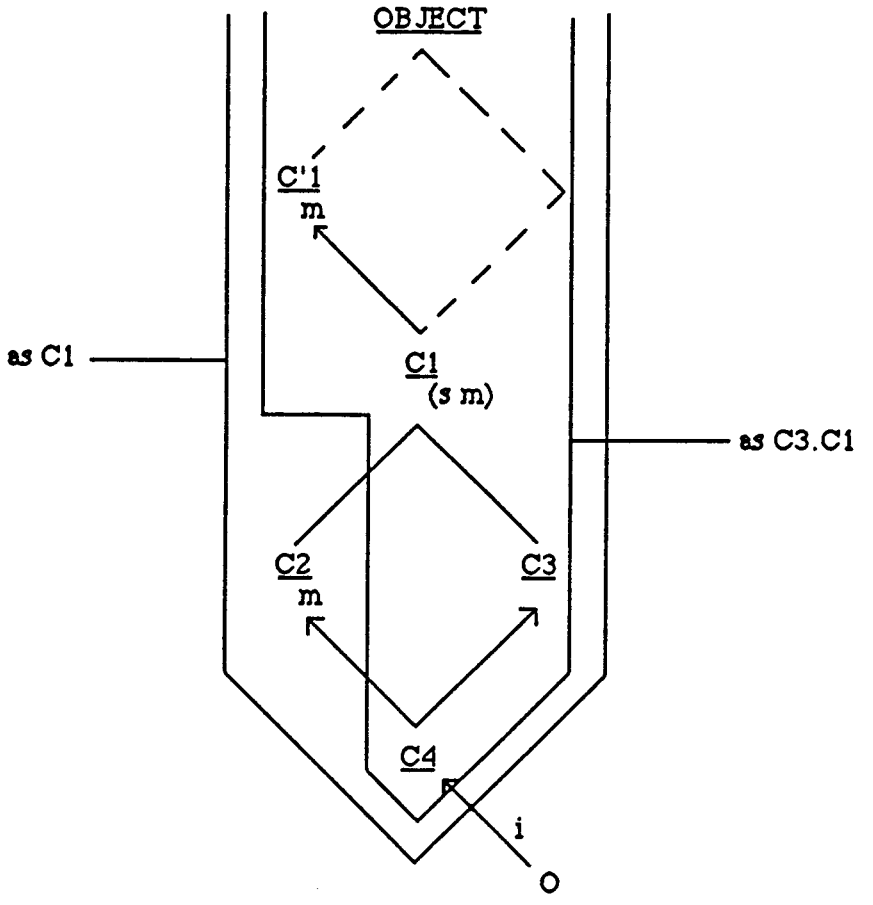
(*) Remarquons que ce cas d'utilisation de l'héritage englobe les cas de
- masquage pour adapter : m de C1 est l'implémentation par défaut redéfinie dans C2 et C3

- masquage pour implémenter : m de C1 est spécifiée mais indéterminée et implémentée dans C2 et C3 (III.3.5.2).




```
[Jean '(who-are-you as Médecin)]

 MR
 => {(print as Médecin.Personne)}

lookup
 => Médecin : print
```

Notons que si la méthode m n'est pas redéfinie dans C3, c'est alors celle de C1 qui est inférée sous le point de vue de C3 et C1 (m as C3.C1). Cette méthode est en effet la plus affinée (et la seule connue) sous le point de vue de C3, ce n'est notamment pas celle de C2 , classe indépendante de C3. Il en serait tout autrement sous le point de vue de C4, qui, elle, connaît C2 et donc peut faire référence à ses caractéristiques, en particulier la méthode m . Plus généralement m peut ne pas être définie par C1 mais simplement héritée par celle-ci, c'est alors cette méthode héritée qui est inférée car les sur-classes de C1 appartiennent à l'intersection des points de vue de C2 et C1 (IV.2.5.2.)



D'où :

$$[O\ '(s\ \text{as}\ C3)] \xrightarrow{MR} \{(m\ \text{as}\ C3.C1)\} \xrightarrow{lookup} C'_1 : m$$

Montrons que pour ce problème, la transformation MR2 qui semblerait, a priori, convenir ici, n'est en fait pas satisfaisante. Modifions tout d'abord MR2 par rapport au relâchement, qui doit être pris en compte par cette transformation comme nous l'avons montré :

MR2

message : [O '(sélecteur as C) argts]

as non spécifié dans C' :	{(méthode as C') argts} , C' ≤ C, relâchement {(méthode as C) argts}, C' > C, conservation
as explicite : {(méthode as C") argts} , C" ≥ C'	

avec C' de même définition que dans MR . Rappelons que le but de MR2 est de s'abstraire du as implicite (C') par conservation du point de vue initial, si le point de vue inféré est plus général (C' > C). Cette solution semble convenir pour le problème de contrainte initiale (p. IV-42). En effet, pour les deux messages :

[o '(s as C2)] et [o '(s as C3)]

la définition du sélecteur s trouvée dans C1 , sous les deux points de vue, ne spécifie pas de as explicite, c'est donc les points de vue initiaux, i.e. de C2 et C3 qui sont conservés, d'où respectivement :

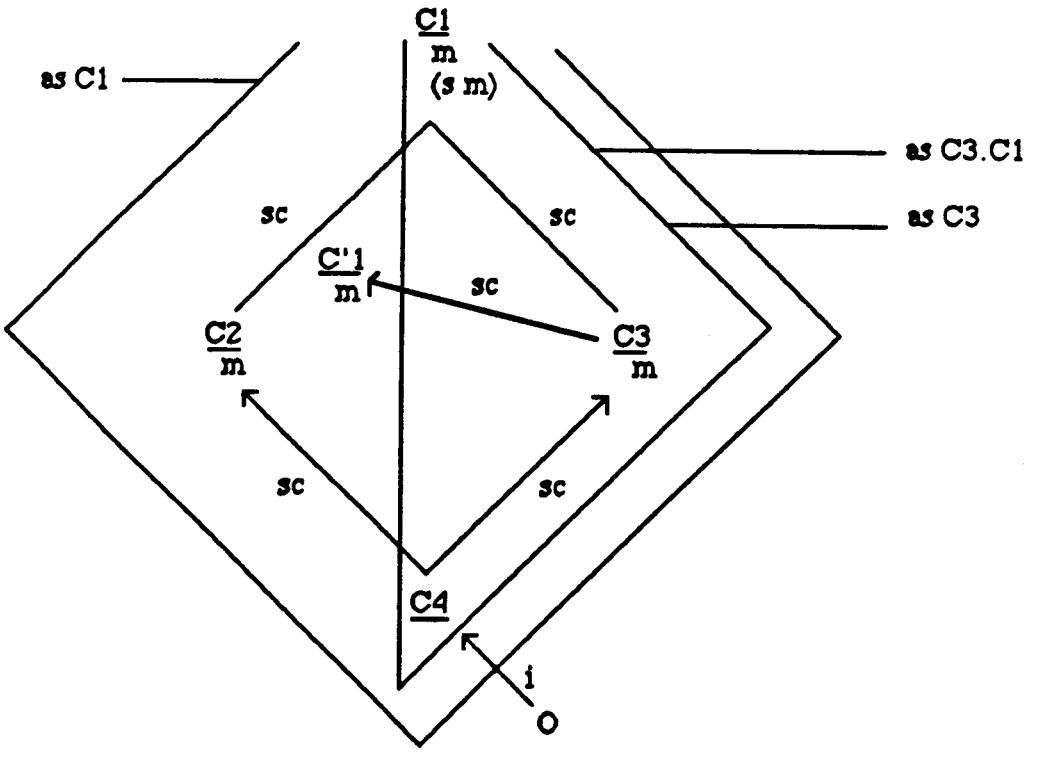
{(m as C2)} et {(m as C3)}

qui mènent bien aux applications :

C2 : m et C3 : m

En fait, MR2 convient dans ce cas particulier parce qu'il n'y a pas d'ambiguité sur la méthode m , qui est ici simplement trouvée dans la classe du point de vue initial, C3.

Considérons le cas contraire, où m n'est pas définie par C3. C'est donc m de C1 (qui définit l'association (s m)) qui doit être inférée d'après ce que nous avons présenté précédemment. Contrarions maintenant ce fait par un conflit d'héritage de m issue également d'une sur-classe C'1 de C3 indépendante de C1 :



par MR2 :

$$[\ '(s\ as\ C3)] \underset{C1>C3}{\overset{MR2}{\Rightarrow}} \{(m\ as\ C3)\} \overset{lookup}{\Rightarrow} \underline{C'1 : m}$$

alors que l'association (s m) est définie dans C1 et n'a de sens que sous le point de vue de C1 , indépendante de C'1.

C'est l'intérêt du as implicite de C1 d'assurer le fait que la définition (s m) dans cette dernière n'a de sens que sous son point de vue .

Par MR :

$$[O\ '(s\ as\ C3)] \underset{C1>C3}{\overset{MR}{\Rightarrow}} \{(m\ as\ C3.C1)\} \overset{lookup}{\Rightarrow} \underline{C1 : m}$$

contrainte

Le principe de conservation de point de vue de MR2 ne peut donc se substituer à l'inférence du as implicite et à la contrainte de point de vue de MR.

Enfin, terminons sur le cas du as explicite dans la transformation MR , pour montrer que le problème de contrainte se pose également ici, ce qui ménera à la version finale de MR. Considérons MR réduite au cas du as explicite :

[O '(sélecteur as C) argts]

=> réaction : as explicite {(méthode as C") argts} , C" ≥ C'

C' étant la classe où est trouvée la définition de sélecteur (sélecteur (méthode as C")).

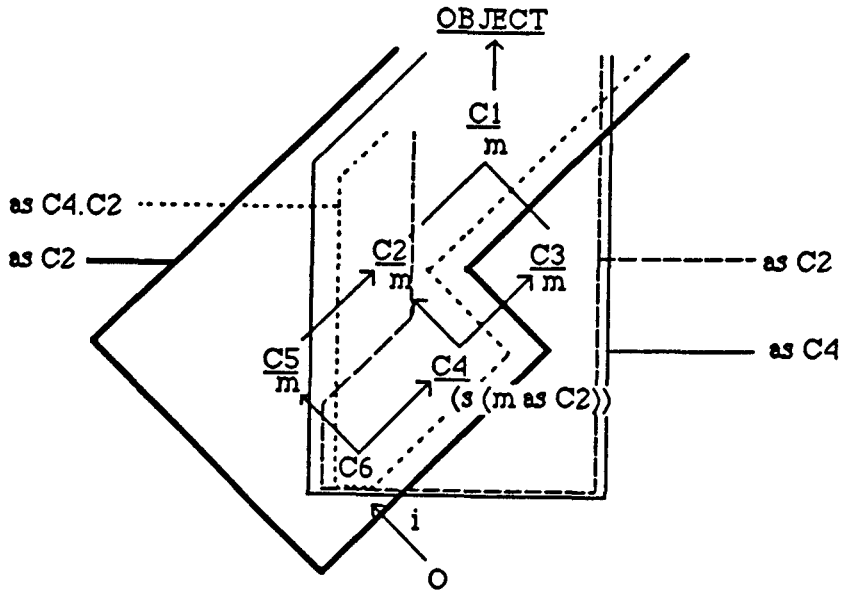
Cette formulation de MR entraîne donc un relâchement systématique du point de vue de C à celui de C" . Ce n'est pas tenir compte des points de vue précédents de C' et de C qui ont mené à considérer celui de C" et qui peuvent être plus affinés, donc le contraindre. Les deux cas suivants peuvent apparaître :

Sachant que l'on a toujours C" ≥ C' , d'après la conception incrémentale des classes, on peut avoir :

$$C" \leq C \quad ou \quad C' > C$$

Examinons le premier cas :

C', dans laquelle est trouvée la définition de sélecteur, est sous-classe de C , nous avons vu que, dans ce cas, pour le as implicite, ceci conduit à un relâchement de point de vue, étant donné que la visibilité de C' peut être plus globale que celle de C :



pour le message :

[O '(s as C3)]

C = C3 , C' = C4 car s est trouvé dans C4 sous le point de vue de C3 , C" = C2.

C4 ≤ C3 , il y a donc relâchement puisque le point de vue de C4 est plus global que celui de C3. Notamment, C4 peut faire référence à (m as C2).

Le relâchement systématique conduirait à considérer simplement (m as C2) , c'est-à-dire C5 :m alors que C5 est indépendante de C4 , ce qui n'est pas l'effet souhaité puisque les seules méthodes m "visibles" sous le point de vue de C4 sont celles de C2 et C3. C4 associant à s la méthode (m as C2) , ce doit être C2 : m qui est inférée.

Le mécanisme de contrainte s'applique donc également ici, et de façon homogène avec ce qui a été présenté précédemment. Le point de vue de C4 ("d'où l'on vient") étant plus affiné que celui de C2 , il impose une contrainte sur celui-ci, c'est donc le point de vue composé qui est inféré :

$$[O\ '(s\ as\ C3)] \xrightarrow{MR} \{(m\ as\ C4.C2)\} \xrightarrow{lookup} C2 : m$$

Plus généralement :

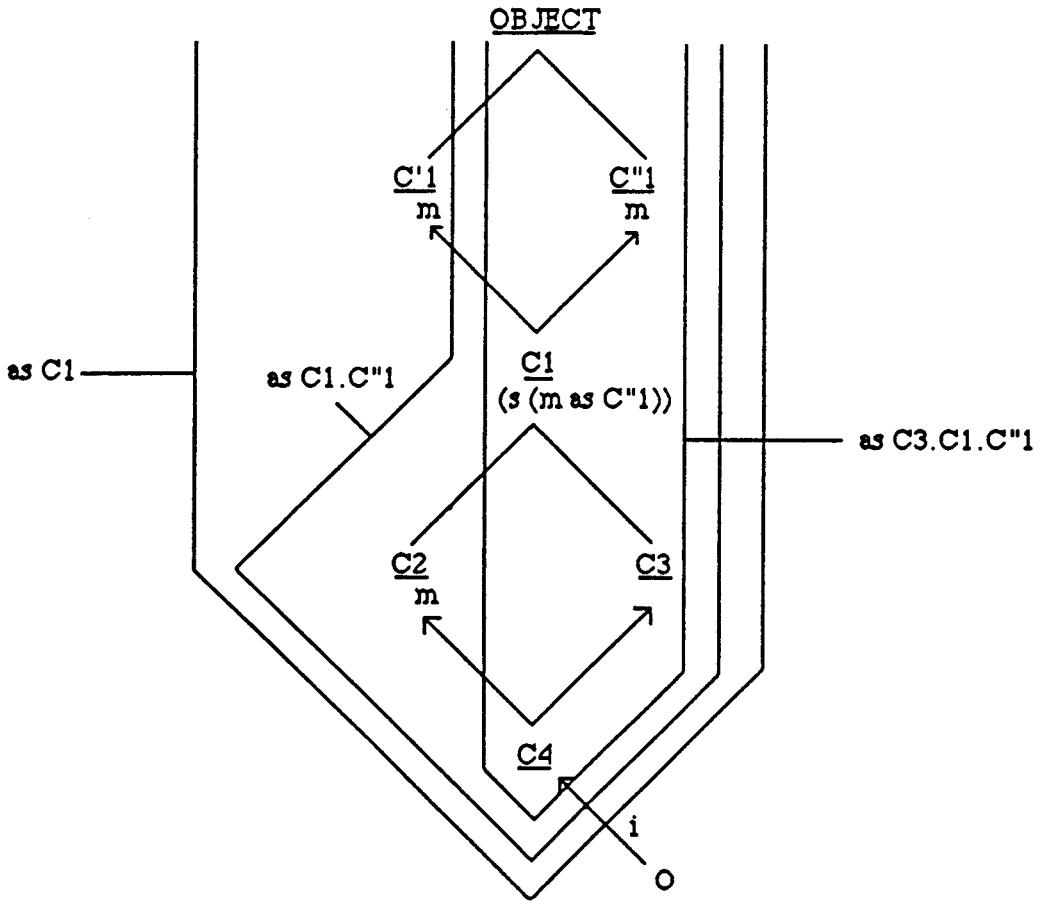
$$[o\ '(s\ \text{as}\ C)] \underset{\substack{C'\leq C\\ C''\leq C''}}{\overset{MR}{\Rightarrow}} \{(m\ \text{as}\ C'.C'')\}$$

Examinons maintenant la deuxième cas : $C' < C$, avec toujours $C'' \geq C$.

De la même façon, le point de vue de C étant plus affiné que celui de C' , lui-même plus affiné que celui de C" , la contrainte est triple, et le point de vue considéré pour la recherche de la caractéristique est l'interaction des points de vue respectifs de C , C' , C".

$$[o\ '(s\ \text{as}\ C)] \underset{C''\geq C'>C}{\overset{MR}{\Rightarrow}} \{(m\ \text{as}\ C.C'.C'')\}.$$

Reprenons l'exemple de la page IV-45 où cette fois l'association de m au selecteur s dans C1 doit se faire par as explicite, car il y a conflit entre les points de vue indépendants des sur-classes de C1, soient C'1 , C"1 :



[O '(s as C3)] devrait conduire à l'application de C"1 : m , et non à celle de C2 : m puisque l'expression (as C3) spécifie un point de vue indépendant de celui de C2. Ce n'est cependant pas le cas avec MR si l'on considère le relâchement systématique de C en C" c'est-à-dire ici de C3 en C"1 , dont le point de vue, plus global, conduit à inférer C2 : m .

C'est pourquoi la formulation précédente s'applique ici :

$$[O\ '(s\ as\ C3)] \underset{C"1 \geq C1 > C}{\overset{MR}{\Rightarrow}} \{(m\ as\ C3.C1.C"1)\} \overset{lookup}{\Rightarrow} \underline{C"1 : m}$$

ce qui est l'effet souhaité.

La version finale de la transformation MR est donc la suivante :

<u>MR</u>

message : [O '(sélecteur as C) argts]

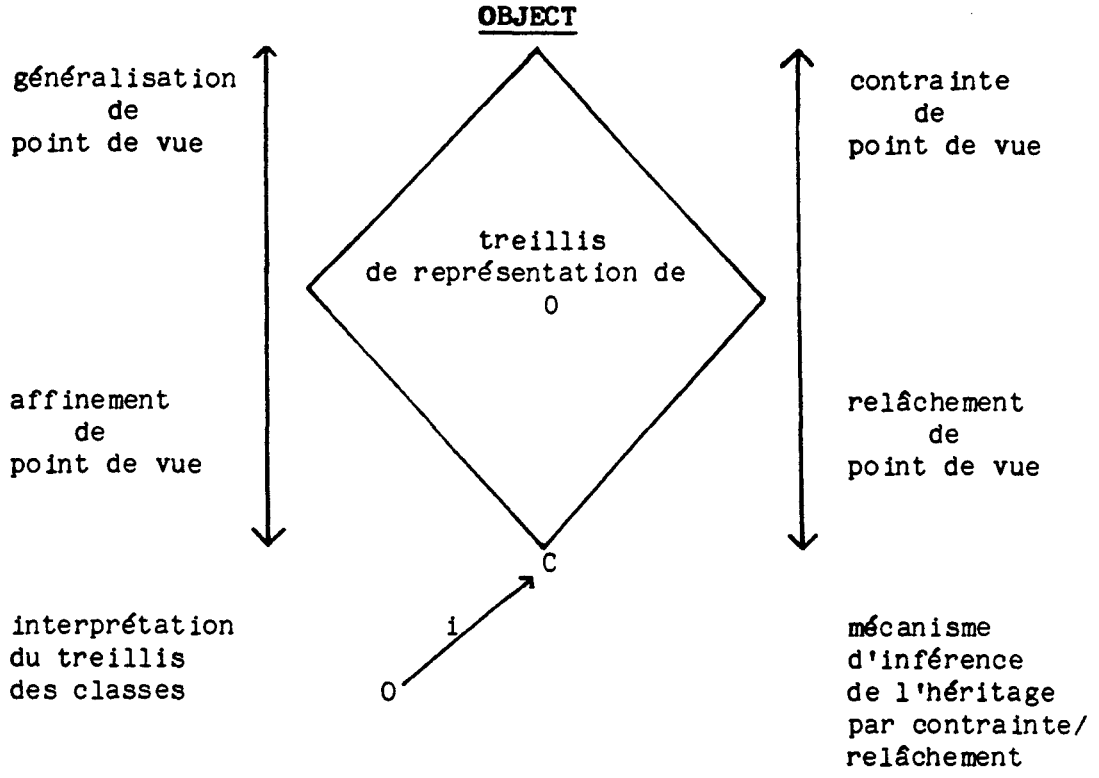
=> réaction :

- <u>as implicite</u> dans C' :
  - {(méthode as C') argts)} <u>si</u> C' ≤ C, <u>relâchement</u>
  - {(méthode as C.C') argts <u>si</u> C' > C <u>contrainte</u>
- <u>as explicite</u> C" dans C' (C" ≥ C') :
  - {(méthode as C'.C")} <u>si</u> C' ≤ C
  - {(méthode as C.C'.C") argts)} <u>si</u> C' > C

où C' est la classe de représentation de O où est trouvée la définition du sélecteur (sélecteur as-méthode) la plus affinée sous le point de vue initial de C .

De façon générale, si le point de vue précédent (C par rapport à C', C' par rapport à C") est plus général, il y a relâchement, inversement, s'il est plus affiné, il y a contrainte.

En conclusion, la transformation MR et le mécanisme d'inférence des points de vue du message à la réaction peuvent se traduire par le schéma suivant :



avec la justification simplifiée du mécanisme :

contrainte : le point de vue considéré à l'étape précédente était plus affiné, il faut donc en tenir compte quand on monte vers le plus général.

relâchement : le point de vue considéré à l'étape précédente était plus général, quand on descend vers un plus affiné, c'est ce dernier qu'il faut prendre en compte même s'il conduit à un relâchement du fait de l'affinement par combinaison.

Ce mécanisme est généralisé de façon récurrente à l'évaluation des as-expressions de méthodes comme nous allons le présenter dans le paragraphe suivant.

IV 2.5.4 As-expressions d'attributs

Nous avons vu que les as-expressions de sélecteurs permettent de spécifier un point de vue de l'extérieur, à travers l'envoi de message, sous lequel l'objet destinataire doit s'activer (transformation Message-Réaction). La réaction de celui-ci consiste à appliquer (applymeth) sa méthode qu'il associe au sélecteur dans son environnement propre (encapsulé). Cet environnement propre de l'objet est constitué de ses caractéristiques d'implémentation, attributs et méthodes, définies par sur treillis de représentation et inférées par héritage selon la règle du plus affiné et le principe d'indépendance. Le mécanisme de sélection des points de vue assure ce dernier principe. Le paragraphe précédent nous a permis d'introduire la technique de contrainte/relâchement sur laquelle est fondé le mécanisme d'inférence des points de vue. Cette technique est la même pour tout type de caractéristique, en particulier ici pour les attributs.

Rappelons que les attributs sont encapsulés dans l'objet et accessibles seulement par les méthodes héritées de OBJECT:

{? nom_attribut} en lecture
{<- nom_attribut valeur} en écriture.

Comme toute méthode, celles-ci ne sont applicables que par l'objet lui-même et donc de telles expressions ne peuvent apparaitre que dans le corps d'autres méthodes de celui-ci.*

L'évaluation des as-expressions d'attributs est donc subordonnée a celle des as-expressions de méthodes. Nous présentons cependant dès maintenant les premières pour terminer ensuite par les méthodes, aboutissement du mécanisme d'évaluation.

Une as-expression d'attribut peut prendre place partout où il y a une référence à l'attribut, notamment dans les méthodes d'accès précédentes:

{? '(nom_attribut as C)} en lecture
{<- '(nom_attribut as C) valeur} en écriture

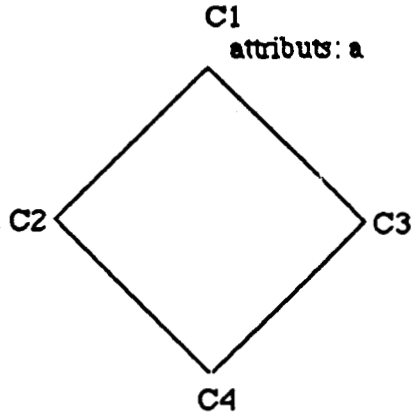
C devant être une classe de représentation de l'objet concerné. L'attribut sélectionné est alors celui défini sous le point de vue de C. Rappelons que la règle du plus affiné ne concerne pas les attributs puisqu'aucune information ne leur est associée, ce sont de simples variables d'instance au sens du modèle de base des L.O.O **. L'héritage des attributs suit les règles suivantes:

---

* Remarquons également que comme toute méthode celles-ci peuvent être liées à des sélecteurs, les rendant de fait applicables de l'extérieur, ce qui laisse peu de garantie à l'encapsulation. La classe suivante par exemple pourrait être la mixin des objets dont les attributs sont accessibles en lecture/écriture à travers les sélecteurs ? et <- associés respectivement aux méthodes de même nom:
[R-CLASS 'new 'RW-OBJECT 'surcl '(OBJECT) 'selecteurs '(? ? <- <-)].

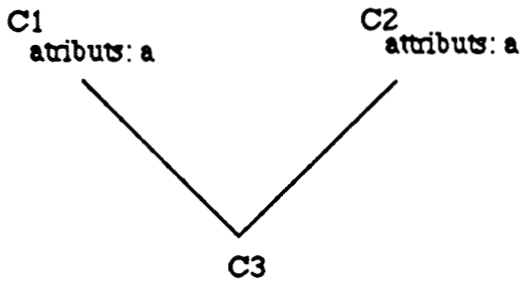
** L'extension aux frames de ROME, en cours d'étude, généralisera l'affinement par substitution aux attributs dans le sens introduit au paragraphe III.4.4.

règle1: un attribut défini par une classe accessible par plusieurs chemins d'héritage n'est hérité qu'une fois



les représentants de C4 ne disposent que d'un seul attribut de nom a, bien qu'il soit hérité par C2 et C3

règle 2: il n'y a pas de conflit d'héritage sur les d'attributs de même nom définis par des classes indépendantes (principe d'indépendance)



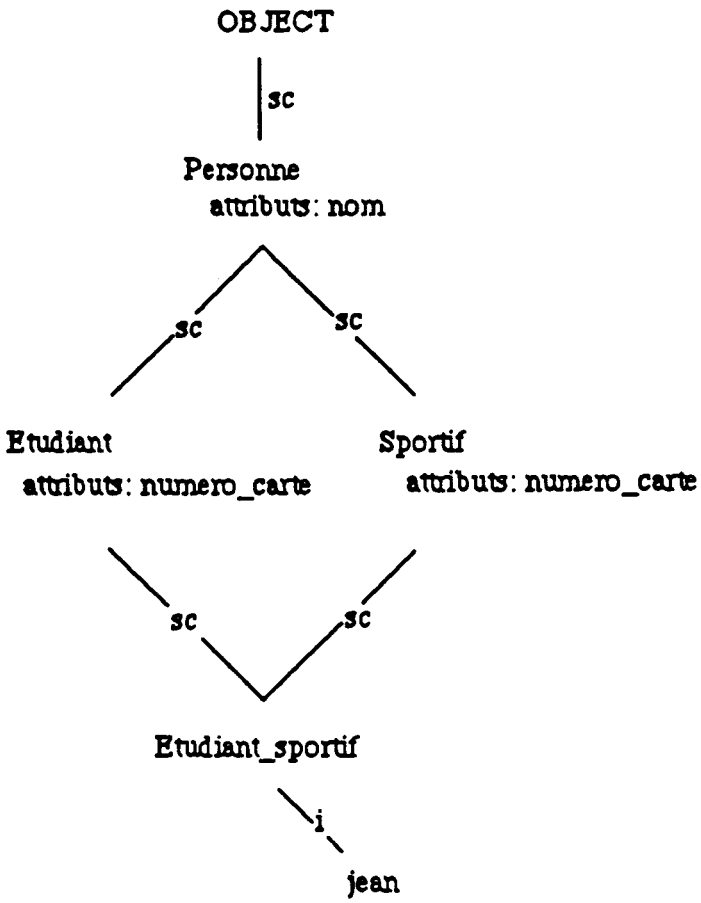
les représentants de C3 disposent de deux attributs de nom a, (a as C1) et (a as C2).

Nous verrons au paragraphe IV.2.5.6 que ces règles ne sont pas vérifiées uniformément par toutes les langages à stratégie graphique, ce qui représente tout un problème en soi.

Etant donné que nous réservons pour la suite la présentation des as-expressions de méthodes qui seules peuvent faire référence à des attributs, utilisons ici l'inspecteur d'objets de ROME pour montrer les as-expressions d'attributs. L'inspecteur permet à l'utilisateur de se mettre à la place d'un objet, sous le point de vue le plus global, celui de sa classe d'instanciation, et donc d'avoir accès à tout son environnement propre, i-e essentiellement d'appliquer ses méthodes et d'accéder à ses attributs sans restriction. Cet outil est utilisable sur tout objet par l'envoi du message "inspect" défini par OBJECT:

```
> [o 'inspect]
inspect> ; l'objet est entr'ouvert, nous y entrons
inspect> ...
inspect>{fin}; nous sortons de l'objet
```

Ainsi par ce biais montrons le respect des deux règles anoncées sur l'exemple suivant:



programmé en ROME comme suit:

```
[I-CLASS 'new 'Personne
 'surcl '(OBJECT)
 'attributs '(nom)]

[I-CLASS 'new 'Etudiant
 'surcl '(Personne)
 'attributs '(numero_carte)]

[I-CLASS 'new 'Sportif
 'surcl '(Personne)
 'attributs '(numero_carte)]

[I-CLASS 'new 'Etudiant_Sportif
 'surcl '(Etudiant Sportif)]

[Etudiant_Sportif 'new 'jean]
```

Mettons nous à la place de jean par l'inspecteur:

```
> [jean 'inspect]
inspect>
```

l'attribut nom de Personne vérifie la règle 1, il n'y a donc pas d'ambiguité:

```
inspect> {<- 'nom "Jean Martin"}
= Jean Martin
inspect> {? 'nom}
= Jean Martin
```

et ceci quelquesoit le point de vue

```
inspect> {? '(nom as Etudiant)}
= Jean Martin
inspect> {? '(nom as Sportif)}
= Jean Martin
```

Observons maintenant le respect de la règle 2 sur l'attribut numero_carte hérité concuremment de Etudiant et Sportif. Ces classes étant indépendantes, d'après le principe d'indépendance il n'y a pas de conflit sur leurs attributs de même nom accessibles par as-expression:

```
inspect> {<- '(numero_carte as Etudiant) "E12"}
= E12
inspect> {<- '(numero_carte as Sportif) "S12"}
= S12
inspect> {? '(numero_carte as Etudiant)}
= E12
inspect> {? '(numero_carte as Sportif)}
= S12
```

l'inspection est concluante, sortons de l'objet jean par:

```
inspect> {fin}
= jean
```

Nous n'insisterons pas d'avantage sur les as-expressions d'attributs que nous reverrons dans le paragraphe suivant et critiquerons au paragraphe IV.2.5.6.1.

IV.2.5.5 As-expressions de méthodes

IV2.5.5.1 Présentation

Nous avons vu aux paragraphes IV.2.1 et IV.2.4 que les méthodes sont des caractéristiques comportementales propres à l'objet, qu'il exécute par les primitives d'application callmeth et applymeth:

{nom_methode argts}

Comme nous l'avons introduit en IV.2.5.3, la référence à nom_methode dans cette primitive peut être substituée par une as-expression de méthode explicite, spécifiant le point de vue sous lequel l'objet doit considérer sa méthode de nom nom_methode:

{(nom_methode as C) argts}

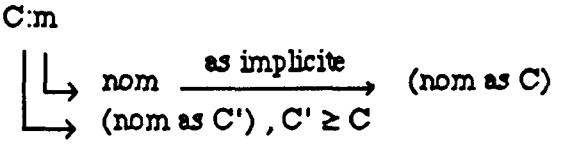
La référence à une méthode par un objet se fait soit à travers les définitions de sélecteurs soit dans le corps d'autres de ses méthodes. Précisons ce second cas. Notons C:m la méthode m définie par la classe C et:

la référence syntaxique à une caractéristique d'implémentation "caractéristique", méthode ou attribut, dans le corps de C:m. D'aprés la conception incrémentale des classes, les caractéristiques auxquelles C peut faire référence dans C:m sont celles qu'elle définit ou qu'elle hérite de ses surclasses, appelons environnement syntaxique cet ensemble de caractéristiques "visibles" de C. Dès lors une référence de caractéristique dans C:m peut prendre les deux formes suivantes:

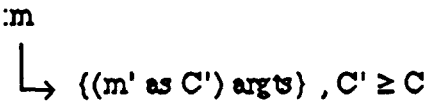
- un nom de caractéristique, soit nom; cette référence est alors automatiquement transformée par as implicite en (nom as C)

- une as-expression de caractéristique (as explicite) du type: (nom as C') avec C' ≥ C

soit, schématiquement:



nom appartenant à l'environnement syntaxique de C. En particulier C:m fait des références syntaxiques à d'autres méthodes appartenant à l'environnement syntaxique de C, c'est le principe de la composition de méthodes (IV.2.4):



Ainsi l'application effective d'une méthode définie par une classe peut être due:

-1- soit à l'application d'une autre méthode, telle C:m qui fait appel à (m' as C')
-2- soit à un envoi de message, si cette méthode tient lieu de réaction à celui-ci.

Le deuxième cas (-2-) a été amené au paragraphe IV.2.5.3 dans lequel a été montré comment un envoi de message se transforme par M.R en application de méthode par l'objet destinataire:

message [O '(s as C) argts]
=> réaction {(m as pdva) argts}

où pdva, que nous appellerons ici le point de vue appelant de m peut prendre les formes suivantes:

C' étant la classe de représentation de O où est trouvée la définition du sélecteur s sous le point de vue de C

- cette définition associe m à s par as implicite
  - C' ≤ C, relâchement: pdva = C'
  - C' > C, contrainte: pdva = C.C'

- cette définition associe m à s par as explicite, soit (s (m as C")), C" ≥ C', il y a toujours contrainte de C" par rapport à C', pdva est au moins égal à C'.C"
  - C' ≤ C, relâchement de C' par rapport à C: pdva = C'.C"
  - C' > C, contrainte de C' par rapport à C: pdva = C.C'.C"

L'évaluation de la réaction, i-e de {(m as pdva) argts} se déroule alors en deux temps:

- recherche par lookup de m sous le point de vue appelant, pdva, soit C" la classe où est trouvée sa définition
- application de la méthode C"':m.

C"':m fait appel à d'autres méthodes ce qui nous amène au cas **-1-**. Le problème est alors de savoir sous quel point de vue sont appelées ces méthodes, sachant que la méthode C"':m appelante est elle-même considérée sous un point de vue, pdva ?

Pour cela à l'application effective de la méthode C"':m est associé un **point de vue d'exécution**, pdve, inféré à partir de son point de vue appelant et de C"' par le **mécanisme d'inférence des points de vue, MIP**, selon la technique de contrainte/relâchement. Ce pdve est le point de vue sous lequel sont résolues toutes les références syntaxiques de caractéristiques intervenant dans le corps de cette méthode. En particulier pour un appel de méthode, soit m':

{(m' as Cl) argts} , Cl ≥ C' (as explicite ou rétabli par as implicite, Cl=C"')

il lui est associé un point de vue appelant inféré à partir de pdve et de Cl par le même mécanisme MIP. Ceci mène à une définition récurrente du mécanisme d'évaluation des méthodes puisque dans tous les cas, **-1-** ou **-2-**, un point de vue appelant est associé à une application effective de méthode et détermine les points de vue appelants de chacune des méthodes appelées.

Précisons maintenant le mécanisme d'inférence des points de vue, soit MIP qui, à partir d'un point de vue composé et d'une classe, en déduit le nouveau point de vue à considérer pour l'objet O:

Soit $pdv = \prod_{i=1}^{n} C_i$, le point de vue composé des classes C_i sur O ($C_1.C_2. \ldots .C_n$)

$\forall i, C_i \in Rep(O) . \forall\ 1 \le i < j \le n, C_i < C_j$

Soit $C \in Rep(O)$ alors

MIP(pdv , C)

$= C$, si $C < C_1$

$= (\prod_{i=1}^{k} C_i).C$, si $\exists\ k, 1 \le k < n / C_k < C < C_{k+1}$

$= pdv.C$, si $C > C_n$

indéfini, sinon.

Notons

pdva / C:m

l'application effective de C:m sous le point de vue appelant pdva. Le point de vue d'exécution de pdva / C:m est alors:

pdve = MIP (pdva , C)

Soit (m' as C'), C' ≥ C, une référence syntaxique à une méthode m' (appartenant à l'environnement syntaxique de C) dans le corps de C:m par:

{(m' as C') argts}

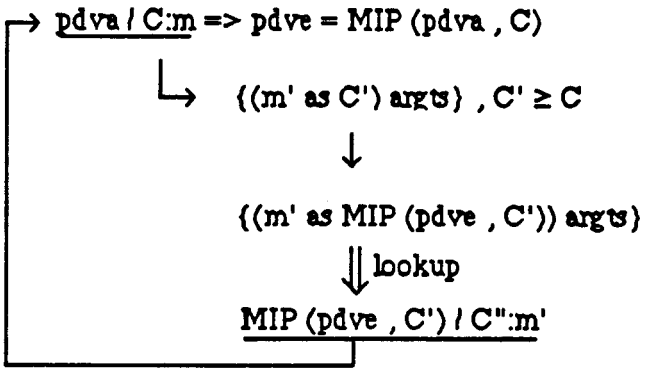
à l'exécution de pdva / C:m , cette application est transformée en:

{(m' as MIP (pdve , C')) argts}
soit {(m' as pdva') argts} avec pdva' = MIP (pdve, C') le point de vue appelant de m'

Il y a alors exécution en deux temps de cette expression:

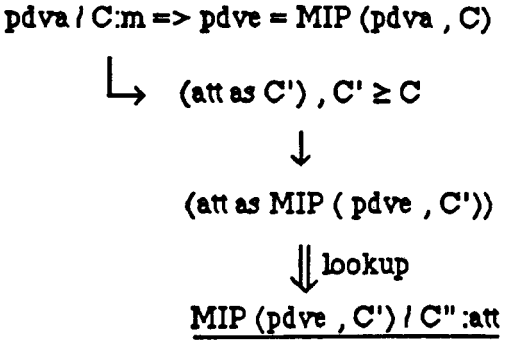
- recherche de m' sous pdva', soit C" la classe où elle est trouvée
- application de la méthode C":m' sous pdva':
  pdva' / C":m', ce qui appelle le même mécanisme d'évaluation pour m'.

Nous pouvons schématiser ce procédé comme suit:



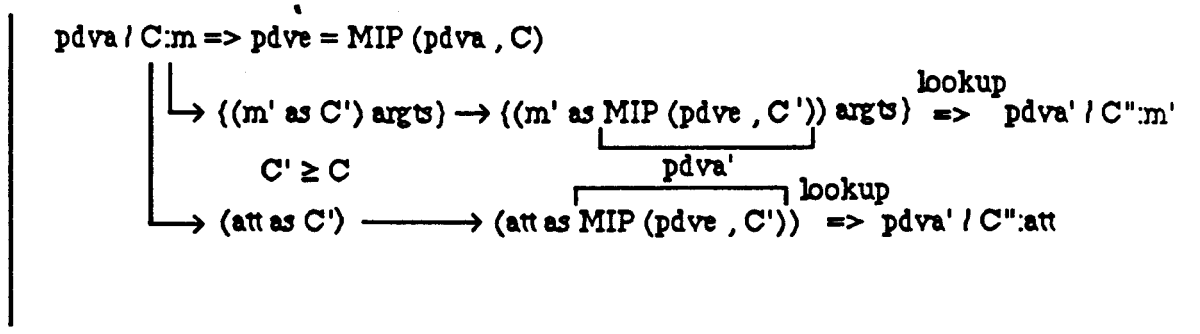
Il en est de même pour les références d'attributs à l'intérieur de C:m, ce que nous schèmatiserons par:

soit att un attribut appartenant à l'environnement syntaxique de C et référencé dans C:m:



C'' étant la classe de représentation de O qui spécifie l'attribut att sous le point de vue appelant, MIP(pdve , C').

En résumé le **mécanisme d'évaluation des as-expressions de caractéristiques d'implémentation** se schématise comme suit:



C'' étant dans chaque cas la classe où est trouvée la caractéristique (méthode m ou attribut att) sous leur point de vue appelant pdva' respectif.

L'activité d'un objet peut se réduire au processus d'application de ses méthodes dans son environnement à partir de son activation par envoi de message. Un objet en cours d'activité se comporte toujours selon un point de vue courant initialisé par MIP lors de son activation par envoi de message puis inféré au cours de ses applications de méthodes:

Le point de vue courant
- initial est le point de vue appelant de la méthode en réaction à l'envoi de message selon la transformation M-R.
- puis évolue de façon récurrente selon le mécanisme d'évaluation des as-expressions de méthodes.

Il décrit une suite de points de vue appelant, point de vue d'exécution, point de vue appelant ...
Montrons par récurence que ce point de vue courant, soit pdvc, est toujours de la forme annoncée pour MIP:

Soit P la propriété suivante:

$$pdvc = \prod_{i=1}^{n} C_i \ , \ \forall i, C_i \in Rep(O) \ . \ \forall\, 1 \le i < j \le n, \ C_i < C_j$$

- Initialement, en rappelant les classes intervenant dans M-R:
  C spécifiée dans le message
  C' classe où est trouvée la définition du sélecteur du message sous pdv(C,O)
  C'' , C'' ≥ C' classe spécifiée dans la définition du sélecteur par C', dans le cas de as explicite,

nous avons vu que le point de vue appelant issu de M-R peut prendre les formes suivantes:

$$C' \text{ si } C' \leq C \Rightarrow pdvc = pdva = \prod_{i=1}^{1} C_i \text{ avec } C_1 = C$$

$$C.C' \text{ si } C' > C \Rightarrow pdvc = \prod_{i=1}^{2} C_i \text{ avec } C = C_1 < C_2 = C'$$

$$C'.C'' \text{ avec } C'' > C' \text{ si } C' \leq C \Rightarrow pdvc = \prod_{i=1}^{2} C_i \text{ avec } C' = C_1 < C_2 = C''$$

$$C.C'.C'' \text{ avec } C'' > C' \text{ si } C' > C \Rightarrow pdvc = \prod_{i=1}^{3} C_i \text{ avec } C = C_1 < C' = C_2 < C_3 = C''.$$

Le point de vue initial a donc dans tous les cas la forme annoncée, P est vérifiée initialement.

- L'étape suivante est l'exécution de la méthode m associée au sélecteur dans la classe C''' retrouvée par lookup sous ce point de vue initial, soit l'exécution de:

$$pdvc = pdva \,/\, C''':m \text{ , avec } C''' \in Dep(pdvc, O).$$

De façon générale montrons donc que si P est vérifiée avant l'application d'une méthode, sur son point de vue appelant, elle l'est pour chacune des références internes de cette méthode:

soit P(pdva) avant application de C:m

pdva / C:m => pdve = MIP (pdva , C)

↳ (caracteristique as C') ⟶ pdva' / C'':caracteristique
C' ≥ C avec pdva' = MIP (pdve, C')

P(pdva') ?

prouvons la vérification intermédiaire de P sur pdve:

P(pdva), C ∈ Dep(pdva,O) => P(pdve)

en effet:

$$pdva = \prod_{i=1}^{n} Ci\ (P(pdva))$$

d'après la propriété de l'intersection de points de vue montrée au IV.2.5.2 et étendue ici à n classes:

$$Dep(pdva,O) = Dep(\prod_{i=1}^{n} Ci, O) = (Succ(C1) \cup [C1\ C2] \cup \ldots \cup [Cn\text{-}1\ Cn] \cup Pred(Cn)) \cap Rep(O)$$

avec $Succ(C1) \cap [C1\ C2] = C1$ car $C1 < C2$

$[Ci\text{-}1\ Ci] \cap [Ci\ Ci+1] = Ci$ car $Ci\text{-}1 < Ci < Ci+1$

$[Cn\text{-}1\ Cn] \cap Pred(Cn) = Cn$ car $Cn\text{-}1 < Cn$

$$\Rightarrow$$

$$Dep(\prod_{i=1}^{n} Ci, O) = (Succ(C1) \cup]C1\ C2] \cup \ldots \cup]Cn\text{-}1\ Cn] \cup Pred(Cn)\text{-}\{Cn\}) \cap Rep(O)$$

Or $C \in Dep(pdva,O)$, d'où trois cas:

$C \in Succ(C1) \Rightarrow C \leq C1 \Rightarrow pdve = MIP(pdva,C) = C$

$$\exists\ k \in [1\ n\text{-}1] / C \in]Ck\ Ck+1] \Rightarrow Ck < C \leq Ck+1 \Rightarrow pdve = MIP(pdva,C) = (\prod_{i=1}^{k} Ci).C$$

$$C \in Pred(Cn)\text{-}\{Cn\} \Rightarrow C > Cn \Rightarrow pdve = MIP(pdva,C) = (\prod_{i=1}^{n} Ci).C$$

P est donc vérifiée dans tous les cas par pdve, et MIP est toujours défini pour pdve.

Montrons enfin que pdva' vérifie P, sachant P(pdve):

pdva' = MIP (pdve,C')

$pdve = \prod_{i=1}^{n} Ci$ (P(pdve)), avec toujours Cn = C (cf supra)

or $C' \geq C$, d'où deux cas pour C':

soit C' = C = Cn (notamment dans le cas de as implicite)

$$\Rightarrow Cn\text{-}1 < Cn = C = C',\ pdva' = MIP(pdve,C) = (\prod_{i=1}^{n-1} Ci).C = \prod_{i=1}^{n} Ci \text{ (inchangé)}$$

soit C'> C = Cn

$$\Rightarrow pdva' = (\prod_{i=1}^{n} Ci).C$$

la contrainte ne fait qu'ajouter au point de vue des surclasses de la dernière imposée (Cn)

cqfd

Nous pouvons également montrer qu'une caractéristique référencée syntaxiquement dans une méthode définie par une classe est toujours définie sous son point de vue courant d'appel (compatibilté entre l'environnement syntaxique et l'environnement dynamique d'exécution).

Soit une référence à une caractéristique telle que précédemment définie:

pdva / C:m => pdve = MIP (pdva , C)

(caracteristique as C') ⟶ **pdva' / C":caracteristique**

$C' \geq C$ **avec pdva' = MIP (pdve, C')**

(caractéristique as C') appartient à l'environnement syntaxique de C. Il suffit de montrer que C' appartient au point de vue courant d'appel à caractéristique, soit pdva', sous lequel celle-ci est recherchée:

pdva' = MIP(pdve,C')

or nous avons vu que MIP(pdve,C') est toujours de la forme

$$\prod_{i=1}^{n} Ci \text{ , avec } Cn = C'$$

C' appartient donc toujours au point de vue d'appel de la caractéristique.

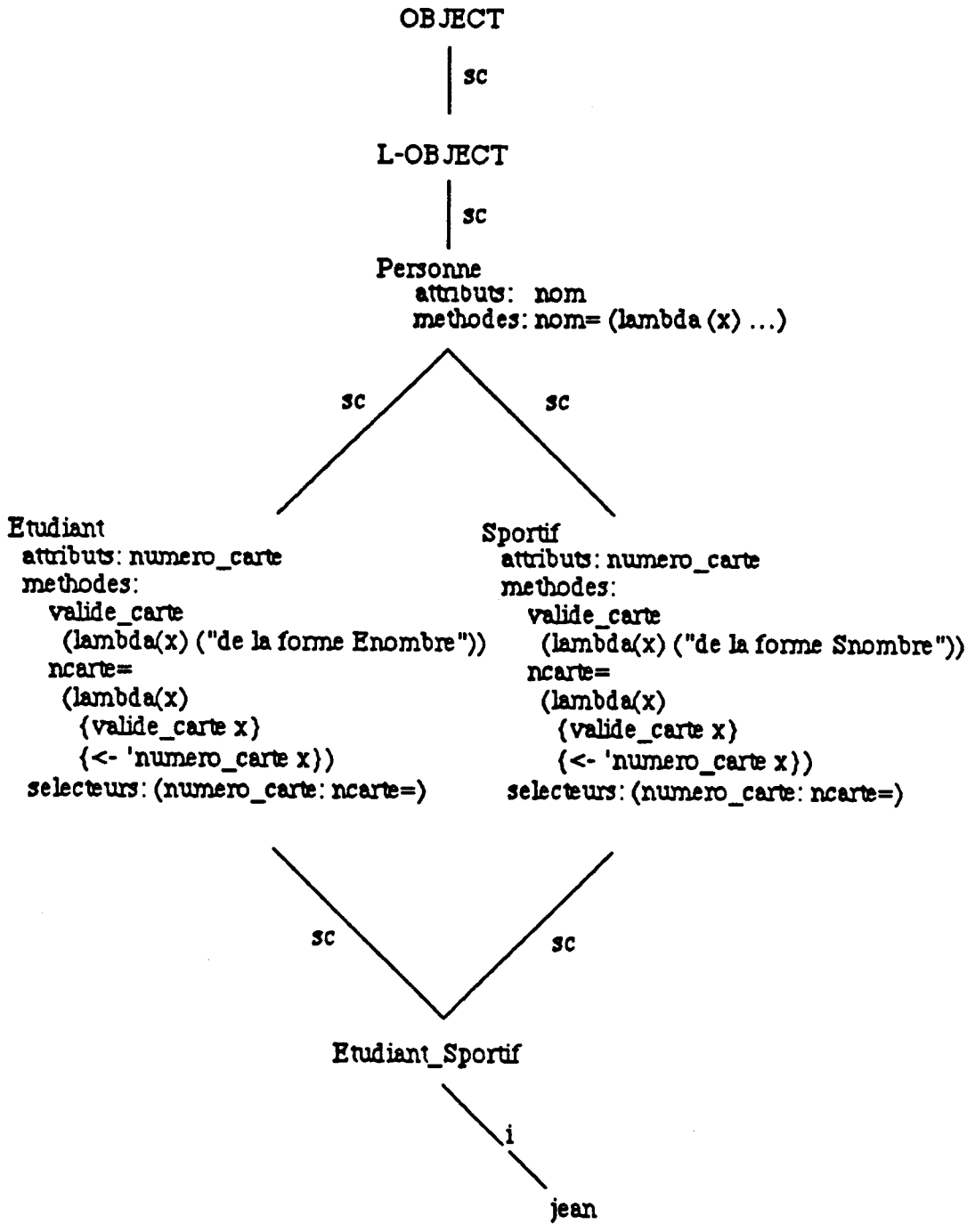
Présentons maintenant ce mécanisme sur des exemples.

IV.2.5.5.2 Exemples

Reprenons l'exemple du IV.2.5.4 programmé cette fois avec des méthodes d'accès en écriture aux attributs. Concernant l'accès en lecture nous simplifions en définissant la mixin L-OBJECT qui permet cet accès par le message ? associé à la méthode de même nom. Ainsi la classe Personne, sommet de notre exemple sera sous-classe de celle-ci.

```
[R-CLASS 'new 'L-OBJECT
 'surcl '(OBJECT)
 'selecteurs '(? ?)]
```

L'exemple se présente alors comme suit:



En ROME:

```
[I-CLASS 'new 'Personne
 'surcl '(L-OBJECT)
 'attributs '(nom)
 'methodes '(nom= (lambda (x) (when (stringp x) {<- 'nom x}))))]
```

```
[I-CLASS 'new 'Etudiant
 'surcl '(Personne)
 'attributs '(numero_carte)
 'methodes
  '(ncarte= (lambda (x) {valide_carte x} {<- 'nom x})
    valide_carte (lambda (x)
                   (if (and (equal (sref x 1) "E") (ndigitp (substring x 2 (1- (slen x)))))
                       t
                       {error "numero_carte de la forme Enombre"})))
 'selecteurs '(numero_carte: ncarte=)]

[I-CLASS 'new 'Sportif
 'surcl '(Personne)
 'attributs '(numero_carte)
 'methodes
  '(ncarte= (lambda (x) {valide_carte x} {<- 'nom x})
    valide_carte (lambda (x)
                   (if (and (equal (sref x 1) "S") (ndigitp (substring x 2 (1- (slen x)))))
                       t
                       {error "numero_carte de la forme Snombre"})))
 'selecteurs '(numero_carte: ncarte=)]

[I-CLASS 'new 'Etudiant_Sportif
 'surcl '(Etudiant Sportif)]

[Etudiant_Sportif 'new 'jean 'nom "Jean Martin"] *
```

L'exemple est construit de telle façon que pour les représentants de Etudiant_Sportif, tel jean, il y ait conflit sur:

- l'attribut numero_carte
- les méthodes valide_carte et ncarte=
- les sélecteurs numero_carte

caractéristiques définies indépendamment dans les surclasses Etudiant et Sportif.
Testons le mécanisme des points de vue comme suit:

les numeros de cartes de jean en tant que Etudiant et Sportif:

```
> [jean '(numero_carte: as Etudiant) "E12"] (1)
> [jean '(numero_carte: as Sportif) "S15"] (2)
```

ces messages ont provoqué l'effet souhaité:

```
> [jean '? '(numero_carte as Etudiant)]
= E12
> [jean '? '(numero_carte as Sportif)]
= S15
```

---

* L'affectation du nom à l'instanciation est autorisée du fait de l'existence de la méthode nom=.

détaillons l'exécution du message (2)

[jean '(numero_carte: as Sportif) "S15"]

M-R
=> {(ncarte= as Sportif) "S15"}
car en reprenant les notations de M-R:
Sportif = C' ≤ C = Sportif, d'où pdva = Sportif

lookup
=> Sportif / Sportif:ncarte= "S15" => pdve = MIP (Sportif,Sportif) = Sportif

{valide_carte "S15"}
as implicite
⟶ {(valide_carte as Sportif) "S15"}
MIP
=> {(valide_carte as Sportif) "S15"}
car le point de vue appelant de valide_carte est
MIP (pdve,Sportif) = MIP (Sportif,Sportif) = Sportif
lookup
=> Sportif / Sportif:valide_carte "S15"
cqfd

{<- 'numero_carte "S15"}
as implicite
⟶ {<- '(numero_carte as Sportif) "S15"}
MIP
=> {<- '(numero_carte as Sportif) "S15"}
de façon similaire à valide_carte
lookup
=> (<- Sportif:numero_carte "S15")
cqfd

Cet exemple montre donc la cohérence de la gestion des points de vue au cours de l'activation de l'objet, ici notamment la conservation du point de vue appelant initial spécifié dans le message.

Modifions un peu l'exemple pour montrer la contrainte de point de vue dans l'application des méthodes.Pour cela ajoutons la méthode *pcarte* d'impression du numero_carte respectivement dans Etudiant et Sportif, et donc conflictuellement pour Etudiant_Sportif dans lequel nous définissons la méthode *pcartes*, qui fait appel à pcarte selon les deux points de vue et est associée au sélecteurs *cartes?*. Ces modifications peuvent se faire dynamiquement en ROME comme suit *:

---

* les messages *addmethod* et *addselecteur* sont définis par R-CLASS et permettent d'ajouter dynamiquement des méthodes et des sélecteurs à la classe receptrice.

```
[Etudiant 'addmethod
 'pcarte '(lambda () (print {? 'numero_carte}))]

[Sportif 'addmethod
 'pcarte '(lambda () (print {? 'numero_carte}))]

[Etudiant_Sportif 'addmethod
 'pcartes '(lambda () (prin "AS ETUDIANT ") {(pcarte as Etudiant)}
                      (prin "AS SPORTIF") {(pcarte as Sportif)})]

[Etudiant_Sportif 'addselecteur 'cartes? 'pcartes]
```

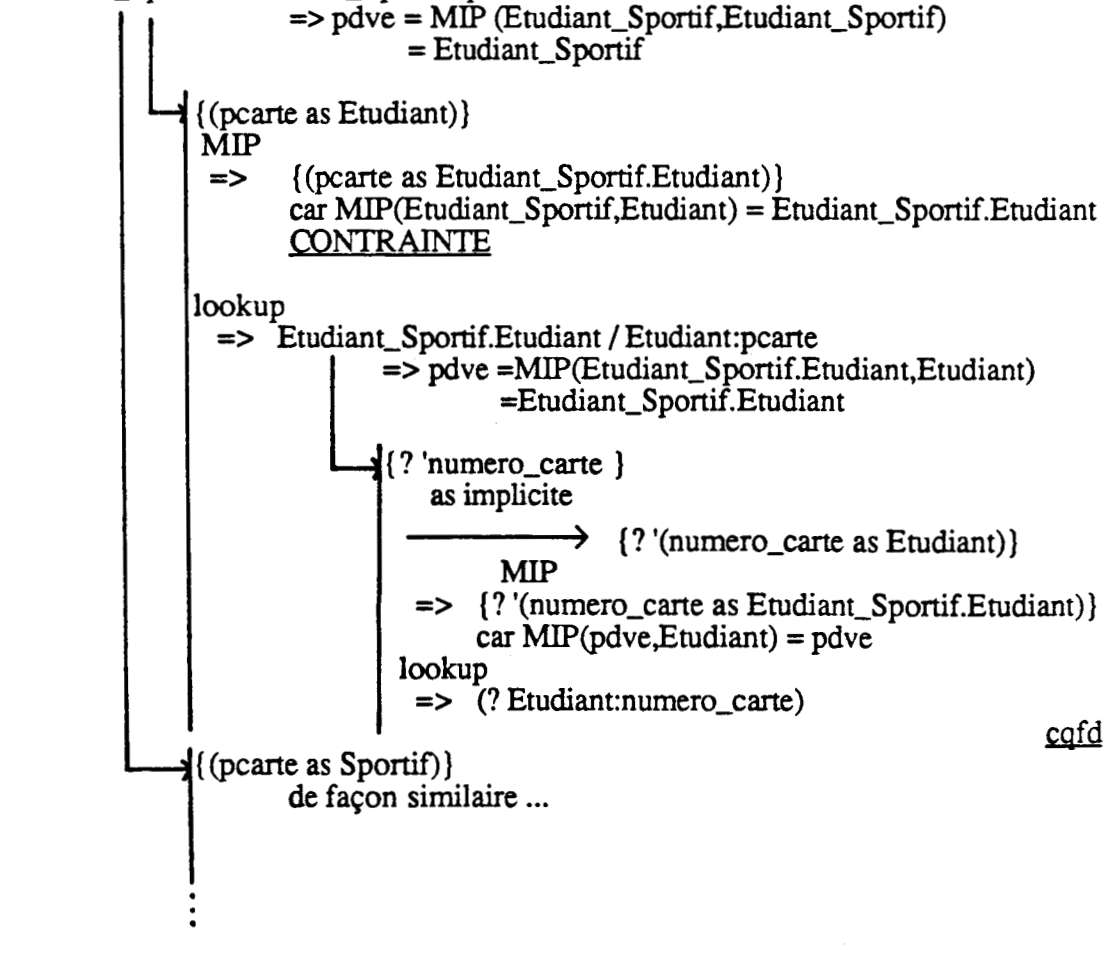
Dès lors:

```
> [jean 'cartes?]
AS ETUDIANT E12
AS SPORTIF S15
```

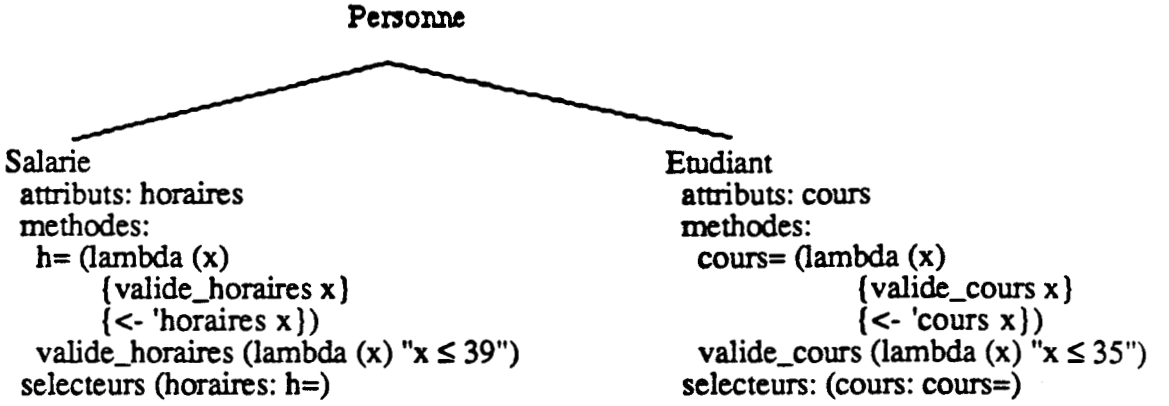
en effet:

[jean 'cartes?]

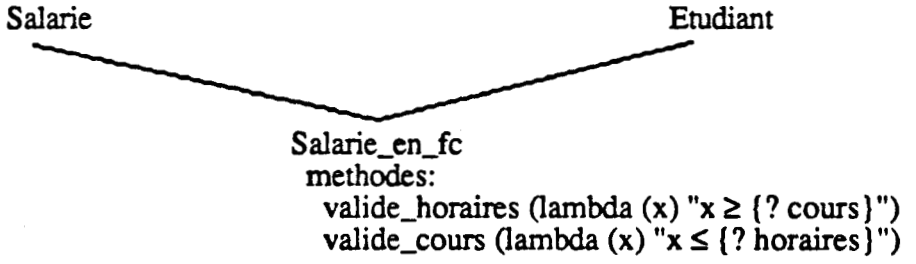
ce message ne spécifie pas de as explicite, c'est alors le point de vue le plus général sur l'objet qui est implicitement pris en compte, i-e celui de OBJECT (ou de façon équivalente à ce niveau, celui de sa classe d'instanciation). OBJECT ne peut engendrer que des relâchements, puisqu'elle est le sommet de tout treillis de représentation d'objet. La définition sans ambiguité de cartes? est alors retrouvée dans Etudiant_Sportif sous le point de vue de OBJECT, d'où implicitement par M-R:



Donnons maintenant un exemple de relâchement de point de vue dans l'application de méthodes. Considérons tout d'abord la classe *Salarie* et une nouvelle version de la classe *Etudiant:*



Les méthodes internes valide_horaires et valide_cours imposent leur contrainte respectivement sur les atttributs horaires et cours. Définissons maintenant la classe *Salarie_en_fc* des salariés en formation continue, sousclasse de Salarie et Etudiant, pour lesquels la contrainte (arbitraire) sur les horaires et les cours est simplement: *cours ≤ horaires*. Pour cela il suffit de redéfinir dans cette classe les méthodes valide_horaires et valide_cours comme suit:



soit en ROME:

```
[I-CLASS 'new 'Salarie
 'surcl '(Personne)
 'attributs '(horaires)
 'methodes
  '(h= (lambda (x) {valide_horaires x} {<- 'horaires x})
    valide_horaires (lambda (x)
                      (if (< x 40)
                      t
                      {error "horaires < 40"})))
 'selecteurs '(horaires: h=)]
```

```
[I-CLASS 'new 'Etudiant
 'surcl '(Personne)
 'attributs '(cours)
 'methodes
  '(cours= (lambda (x) {valide_cours x} {<- 'cours x})
    valide_cours (lambda (x)
                         (if (< x 36)
                         t
                         {error "horaires < 36"})))
 'selecteurs '(cours: cours=)]

[I-CLASS 'new 'Salarie_en_fc
 'surcl '(Salarie Etudiant)
 'methodes
  '(valide_horaires
    (lambda (x)
      (let ((cours {? 'cours}))
       (if (or (equal cours '??) ;indéterminé initialisé à l'instanciation
               (>= x cours))
           t
           {error "cours <= horaires "})))
   valide_cours
    (lambda (x)
      (let ((horaires {? 'horaires}))
       (if (or (equal horaires '??) ;indéterminé
               (<= x horaires))
           t
           {error "cours <= horaires "})))))]
```

Jean est salarié en formation continue:

```
> [Salarie_en_fc 'new 'jean]
```

ses attributs cours et horaires sont pour l'instant indéterminés:

```
> [jean '? 'cours]
= ??
> [jean '? 'horaires]
= ??
```

fixons lui ses horaires:

```
> [jean 'horaires: 30]
```

vérifions

```
> [jean '? 'horaires]
= 30
```

ses cours maintenant (tels que cours ≤ horaires), soit le message (M):

```
> [jean 'cours: 20]
```

vérifions:

```
> [jean '? 'cours]
```

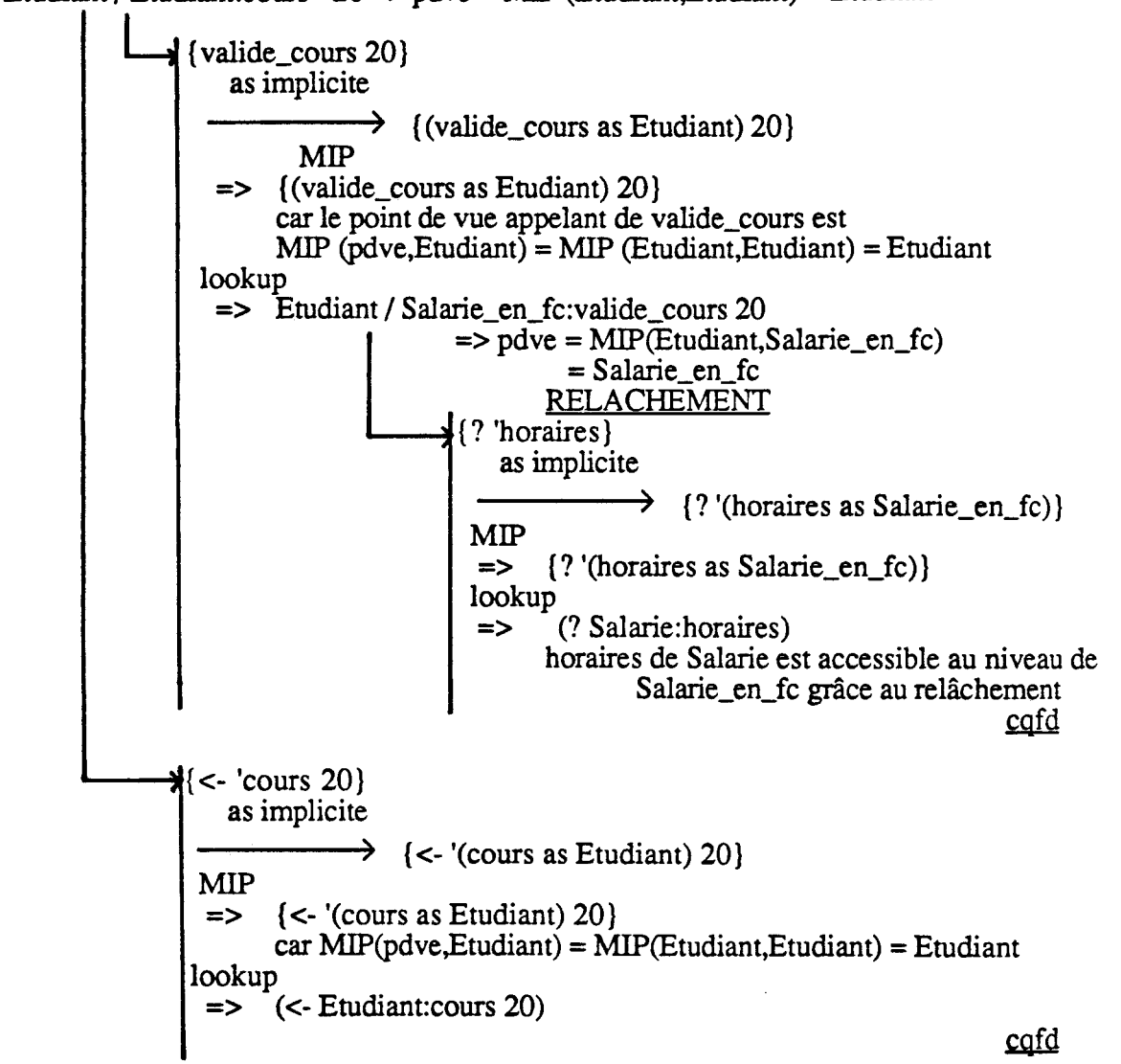
et toujours:

> [jean '? 'horaires]
= 30

Détaillons l'exécution du message (M):

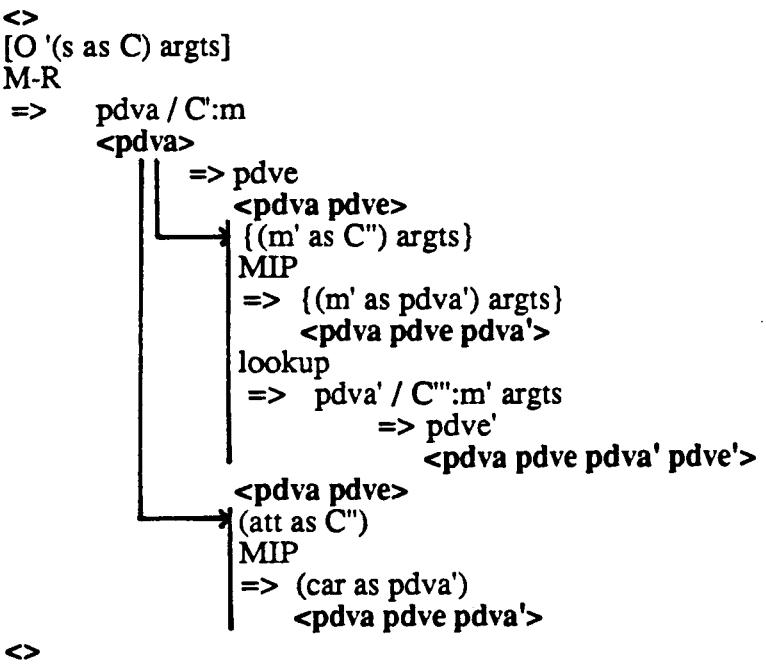
(M) [jean 'cours: 20]

Par as implicite sous M-R, i-e sous le point de vue le plus général sur jean, OBJECT, cours: est trouvé dans Etudiant, d'où:



Insistons sur le fait que le point de vue d'exécution de Etudiant / Etudiant:cours=, en l'occurence Etudiant, est conservé au retour de l'appel de la méthode Etudiant / Salarie_en_fc:valide_cours pour toutes les références suivantes, telles que l'accès à l'attribut cours. **Le point de vue courant est géré en pile.** A chaque inférence de point de vue, MIP considère le point de vue courant qui est en sommet de pile et empile le point de vue courant résultat. Au retour d'une référence effective, application de méthode ou accès à un attribut, pour une méthode il y a dépilement de son point de vue d'exécution et dans les deux cas dépilement du point de vue d'appel de telle sorte qu'en sommet de pile le point de vue courant soit bien le point de vue d'exécution de la méthode appelante. Il en est de même pour le point de vue initial:

Notons <pdvc1 pdvc2 ...> la pile des points de vue courants, et <> la pile vide:



Ces exemples nous semblent être les exemples minimaux montrant tous les aspects du mécanisme d'inférence des points de vue, on pourra en trouver d'autres dans [Carré & Comyn 87c] et [Mathieu 87].

Dans le paragraphe suivant nous comparons la stratégie graphique de ROME à d'autres stratégies graphiques, notamment celle de Extended Smalltalk.

IV.2.5.6 Comparaison aux autres stratégies graphiques

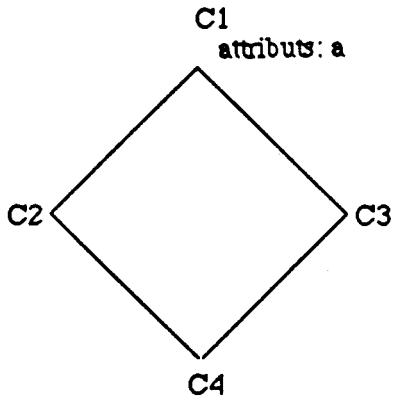
IV.2.5.6.1 Héritage des attributs

Le premier point de comparaison concerne l'héritage des attributs. Nous avons vu (IV.2.5.4) en ROME les deux règles observées:

règle1: un attribut défini par une classe accessible par plusieurs chemins d'héritage n'est hérité qu'une fois

règle 2: il n'y a pas de conflit d'héritage sur les d'attributs de même nom définis par des classes indépendantes (principe d'indépendance).

La règle 1 est vérifiée par la majorité des langages à stratégie graphique, notamment en Extended Smalltalk et Treillis/owl. Elle ne l'est cependant pas en CommonObjects où l'héritage est essentiellement fondé sur un mécanisme de copie (avec effacement pour les méthodes si redéfinition) des caractéristiques héritées des surclasses. Ainsi en reprenant l'exemple du IV.2.5.4:



il y a duplication de a dans C2 et C3 et pour C4 héritage de deux attributs de nom a.

La règle 2 n'est pas vérifiée par la majorité des langages, notamment Extended Smalltalk et Treillis/owl. Par contre de façon homogène avec le cas précédent, CommonObjects observe cette règle.

En résumé:

- Extended Smalltalk et Treillis/owl ne font hériter qu'une seule fois dans tous les cas, notamment sans tenir compte du principe d'indépendance (règle2)

- CommonObjects fait hériter dans tous les cas autant de fois qu'il y a de chemins d'héritage de la classe considérée à la classe définissant l'attribut.

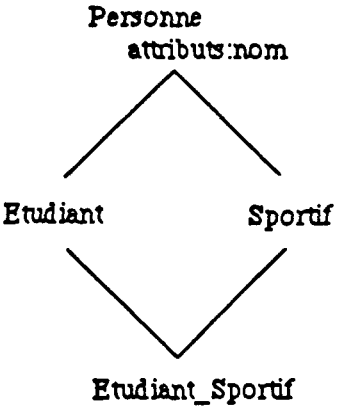
- Rome fait hériter une seule fois si la classe de définition est accessible par plusieurs chemins et n fois si n classes indépendantes le définissent.

Nous arguons que le respect de la règle 2 et par conséquent du principe d'indépendance est fondamental vis à vis de la conception de classes indépendantes. Le mécanisme des points de vue de Rome satisfait ce principe. L'approche adoptée par Extended Smalltalk et Treillis/owl ne sont pas satisfaisantes ici contrairement à celle de CommonObjects.

Concernant la règle 1, elle pose tout le problème de la sémantique de l'héritage et du lien classe/sousclasse et réclame tout une étude en soi, nous ne présenterons que quelques idées.

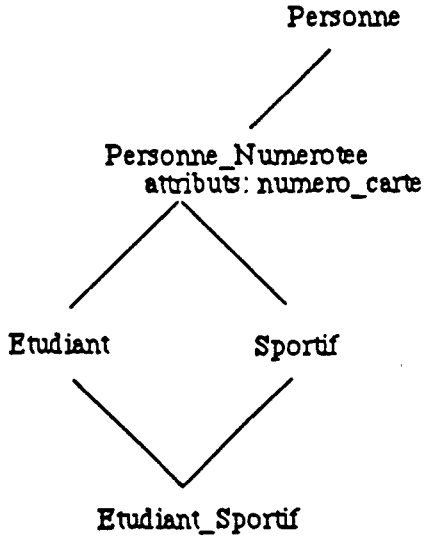
Les stratégies extrêmes de Rome et de CommonObjects posent toutes deux des problèmes ici. Opérons par contre-exemple.

- La stratégie de CommonObjects ne convient pas pour l'exemple de l'attribut nom de Personne:



Contrairement à Rome, en CommonObjects un Etudiant_Sportif dispose de deux attributs nom, ce qui n'est pas l'effet souhaité.

- Inversement la stratégie de Rome ne convient dans le cas suivant:
en reprenant l'attribut numero_carte défini par Etudiant et Sportif, puisque toutes deux la définissent, on peut le factoriser dans une mixin , soit Personne_Numerotee.
Rétrospectivement c'est un cas typique de réutilisabilité de mixins.



Contrairement à CommonObjects, en Rome un Etudiant_Sportif dispose d'un seul attribut numero_carte, ce qui n'est pas l'effet souhaité.

Dans ces deux stratégies extrêmes tout n'est pas possible de manière équivalente et il faut concevoir les classes en fonction du mécanisme sous-jacent.

Il nous semble qu'une approche de la solution est à rechercher dans les deux sémantiques essentielles de la relation classe/surclasse qui sont abstraction/concrétisation et généralisation/spécialisation (III.3.5). La première dont ce dernier exemple est une application, semble réclamer une approche de type CommonObjects, i-e par copie: Personne_Numerotee est une abstraction (partielle) de Etudiant , Sportif et éventuellement d'autres classes (Chômeur, Prisonnier ...). La seconde dont le premier exemple est une application semble par contre réclamer une approche à la Rome, Etudiant, Sportif et Etudiant_sportif sont des spécialisations de la classe Personne. Ainsi nous pouvons énoncer que, concernant l'inférence des attributs, Rome privilégie la spécialisation tandis que CommonObjects privilégie l'abstraction.

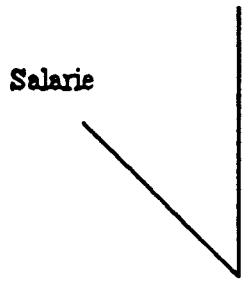
Un compromis pourrait passer par l'étiquetage du lien classe/surclasse comme lien d'abstraction ou de spécialisation auxquels seraient associées des sémantiques différentes. Rappelons que nous avons montré au paragraphe III.3.5 que ces deux interprétations du même lien pouvaient intervenir dans un même treillis de classes. Peut être d'après ce que nous constatons ici, doivent-elles être dissociables?

IV.2.5.6.2 Héritage des sélecteurs et méthodes

Intéressons nous ici à comparer la sémantique des points de vue sur le comportement d'un objet en Rome aux autres stratégies graphiques. Pour cela nous allons la comparer au mécanisme de sélection d'Extended Smalltalk par sélecteurs composés présenté au III.3.4.3 sans que ceci restreigne l'étendue de la comparaison aux autres langages qui adoptent une approche semblable. Nous allons constater que ce mécanisme et celui des points de vue de Rome ont peu de propriétés communes et des sémantiques complètement différentes.

La différence essentielle porte sur l'affinement. Nous avons vu au IV.3.4.3 que les stratégies graphiques classiques posent le problème de l'inhibition de l'affinement puisque toute méthode définie par des surclasses de niveaux supérieurs (plus généraux) est potentiellement accessible indépendamment de ses redéfinitions éventuelles. Pour cela il suffit de spécifier un sélecteur composé préfixé par une classe à partir de laquelle le lookup retrouvera cette méthode. Prenons l'exemple Salarie_en_fc programmé en Smalltalk, du moins partiellement comme suit, en considérant que Personne existe:

```
Etudiant
  superclasses: Personne
  instance variables: cours
  methods:
    cours: x
      x <= 35
        ifTrue: [cours <- x]
        ifFalse: [self error: 'cours <= 35']
```




```
Salarie_en_fc
  superclasses: Salarie Etudiant
  methods:
    cours: x
      (horaires isNil) or (x <= horaires)
        ifTrue: [cours <- x]
        ifFalse: [self error: 'cours <= horaires']
```

et

```
Jean <- Salarie_en_fc new
```

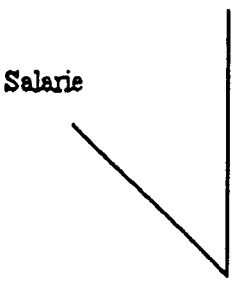
Il suffit pour outre-passer l'affinement de cours: de Etudiant dans Salarie_en_fc d'envoyer le message suivant:

Jean Etudiant.cours: 30

qui provoque simplement l'application de cours: définie par Etudiant, indépendamment de sa redéfinition dans Etudiant_Salarie. Ceci laisse peu de sémantique au masquage.

Nous avons vu que contrairement à cette approche, en Rome, c'est toujours la méthode la plus affinée qui est appliquée sous le point de vue courant, soit sur la solution suivante, en Rome cette fois:

```
Etudiant
 surcl: Personne
 attributs: cours
 methodes:
  cours=
   (lambda (x)
    (if (<= x 35)
        {<- 'cours x}
        {error " cours <= 35"}))
  selecteurs (cours: cours=)
```




```
Salarie_en_fc
 surcl: (Salarie Etudiant)
 methodes:
  cours=
   (lambda (x)
    (let ((horaires {? 'horaires}))
     (if (or (equal horaires '??)
             (<= x horaires))
         {<- 'cours x}
         {error "cours <= horaires "})))
```

et

[Salarie_en_fc 'new 'jean]

le message

[jean '(cours: as Etudiant) 30]

et même [jean 'cours: 30] puisqu'il n'y a pas d'ambiguité sur ce sélecteur provoque l'effet suivant:

MIP
=> {(cours= as Etudiant) 30}

lookup
=> Salarie_en_fc:cours= 30

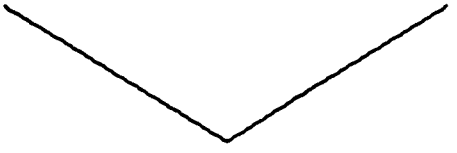
méthode cours= la plus affinée sous le point de vue spécifié, Etudiant.
La notion de point de vue en Rome n'inhibe pas l'affinement mais au contraire a la propriété de conservation de l'affinement.

Une autre différence essentielle vient plus particulièrement du mécanisme d'inférence des points de vue de Rome qui n'a pas d'équivalent ici. Notamment il n'y a pas de notion équivalente au as implicite, ce qui engendre les problèmes évoqués pour la transformation M-R1 du paragraphe IV.2.5.3.

Montrons ceci sur l'exemple Etudiant/Sportif développé ici en Smalltalk, en rappelant tout d'abord qu'il ne peut y avoir de variables d'instance de même nom, nous nommerons donc numero_carte différemment dans chaque classe:

```
Etudiant
 superclasses: Personne
 instance variables: numetudiant
 methods:
  numero_carte: x
   self valide_carte: x.
   numetudiant <- x
  valide_carte: x
   "de la forme Enombre"
  numero_carte
   ^ numetudiant
```

```
Sportif
 superclasses: Personne
 instance variables: numsportif
 methods:
  numero_carte: x
          self valide_carte: x.
    numsportif <- x
  valide_carte: x
   "de la forme Snombre"
  numero_carte
   ^ numsportif
```



```
Etudiant_Sportif
  superclasses: Etudiant Sportif
```

et

Jean <- Etudiant_Sportif new

Les méthodes numero_carte et numero_carte: sont les méthodes d'accès respectivement en lecture et en écriture aux attributs numetudiant et numsportif.

Dès lors le message suivant a bien l'effet souhaité:

Jean Etudiant.numero_carte: 'E12'

vérifions

Jean Etudiant.numero_carte
E12

contrairement au message:

Jean Sportif.numero_carte: 'S15'
... error "de la forme Enombre" ...

en effet lors de l'application de la méthode numero_carte: de Sportif, ce point de vue est oublié pour toutes les méthodes auxquelles elle fait appel, notamment *self valide_carte: x.*(pas de point de vue d'exécution). Les méthodes sont toujours recherchées à partir de la classe d'instanciation (implicitement comme M-R1), le point de vue est toujours celui de cette classe, i-e le plus général sur l'objet. Ainsi *valide_carte* recherchée sur tout le treillis de représentation de Jean est trouvée dans Etudiant (le lookup de base est linéaire en profondeur d'abord). Bien que les deux classes Etudiant et Sportif soient indépendamment correctes, il faut les modifier du fait de l'existence ultérieure de Etudiant_Sportif, de manière à spécifier explicitement le sélecteur composé pour toute méthode qui devient conflictuelle dans cette sousclasse:

Etudiant

```
...
numero_carte: x
  self Etudiant.valide_carte: x.
  numetudiant <- x
...
```

Sportif

```
...
numero_carte: x
  self Sportif.valide_carte: x.
  numsportif <- x
...
```

Ce qui est évidemment peu satisfaisant vis à vis de la conception incrémentale des classes et du principe d'indépendance.Par ailleurs cette référence explicite de méthode a pour consequence d'inhiber tout affinement ultérieur de celle-ci, conformément au problème déjà évoqué au IV.3.4.3.

En conclusion les mécanismes de sélection des stratégies graphiques classiques sont plus des outils de nommage explicite de méthode que des techniques de restriction de point de vue sur l'objet telles qu'en Rome. Cette "explicitation" engendre une résolution figée des références, ce qui entraine notamment une inhibition de l'affinement (cf citation de Bobrow à la fin du paragraphe III.4.3). Ces mécanismes sont donc peu comparables à celui de Rome bien que les intentions initiales semblent se rapprocher, en particulier la réalisation du principe d'indépendance.

IV.3 Représentation Multiple et Evolutive

Nous présentons ici une contribution originale de Rome pour la représentation orientée objet des connaissances, sous la notion de **Représentation Multiple et Evolutive**, R.M.E, d'objet. La représentation multiple permet de rattacher l'objet à plusieurs classes. La représentation évolutive permet de faire évoluer ce rattachement. Nous montrons tout d'abord que l'instanciation, qui induit une Contrainte de Représentation Unique Figée, C.R.U.F, va à l'encontre de la pleine utilisation de l'aspect déclaratif des langages orienté objet résidant dans leur capacité d'expression de classifications. Le paragraphe IV.3.2 présente ensuite la classe R-OBJECT des objets à R.M.E. Puis nous comparons la R.M.E de Rome à d'autres solutions telles que l'agrégation classique utilisée notamment dans PIE et LOOPS. Dans la conclusion générale (chapitre V) nous introduirons les développements possibles à partir de la R.M.E pour la représentation orientée objet des connaissances.

IV.3.1 Problématique

IV.3.1.1 Instanciation: Contrainte de Représentation Unique Fiqée, C.R.U.F

Nous avons vu que le postulat P3 du modèle de base des L.O.O (II.3.1) fixe le lien entre un objet et le treillis des classes par l'instanciation:

> Tout objet est représentant direct d'une et une seule classe, sa classe d'instanciation.
>
> "Each object is associated with a single class"
> [Goldstein & Bobrow 80]

C'est ce que nous nommons la **Contrainte de Représentation Unique Fiqée, C.R.U.F.** L'instanciation est classiquement le seul moyen de déclarer l'appartenance d'un objet à une classe, ceci en se référant à l'interprétation ensembliste du modèle (III.3.1). Précisons que d'après le principe d'inclusion (III.3.5.5), l'objet peut être considéré comme représentant de sa classe d'instanciation mais également de toutes les surclasses de celle-ci:

> par l'instanciation, l'objet est:
> - instance, représentant direct de sa classe d'instanciation
> - représentant des surclasses de celle-ci

La contrainte de Représentation Unique se référe à l'unicité du lien de représentation directe, ou lien d'instanciation qui interdit à l'objet d'être représentant direct de plusieurs classes. D'autre part ce lien de représentation est figé à l'instanciation, l'objet ne peut donc évoluer en tant que représentant d'autres classes. De plus, concernant l'interprétation conceptuelle des classes (i-e leur fonctionnalité d'abstraction), la caractérisation de l'objet est ainsi figée à sa création et ne peut évoluer du même fait que précédemment. Nous allons montrer dans les deux paragraphes suivants les conséquences de la contrainte C.R.U.F notamment vis à vis de la classification multiple par points de vue et l'évolution. La notion de Représentation Multiple et Evolutive présentée ensuite se fonde essentiellement sur la remise en cause de cette contrainte.

IV.3.1.2 Classification multiple par points de vue

Nous avons vu au paragraphe III.3.5.1 que la capacité d'expression de classifications, de taxonomies, simples ou multiples est l'un des aspects les plus déclaratifs du modèle orienté objet, atout essentiel pour la représentation des connaissances. La contrainte de Représentation Unique limite cependant la pleine utilisation de cet atout notamment dans le cas de classification multiple selon des points de vue différents. De façon générale la classification d'entités d'un domaine d'expertise est rarement unique, ne serait-ce qu'à

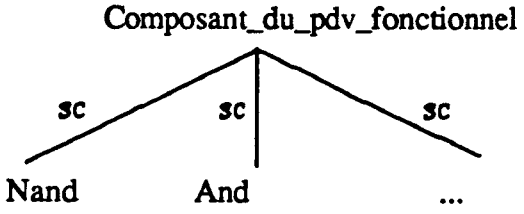
cause des points de vue distincts de plusieurs experts sur les mêmes entités (multi-expertise).

En CAO électronique par exemple [Carré & Comyn 87ab], nous pouvons dissocier trois points de vue différents sur les composants logiques:

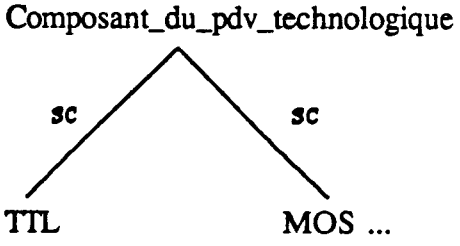
- le point de vue fonctionnel concernant la conception au niveau logique
- le point de vue technologique qui s'intéresse aux contraintes électriques: délais, temps de réponse, dissipation ...
- le point de vue de l'encombrement spatial vis à vis du placement - routage de la carte concerne les types de boîtiers, l'écartement des broches ...

Sous chaque point de vue, l'expertise mène à une classification propre des composants:

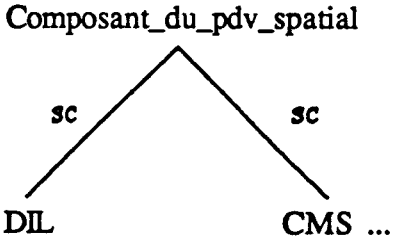
- du point de vue fonctionnel:



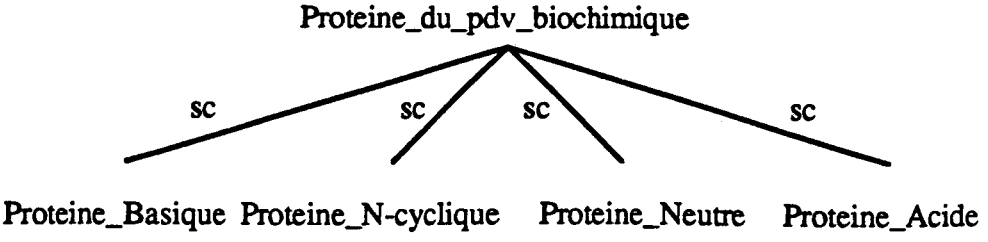
- du point de vue technologique:



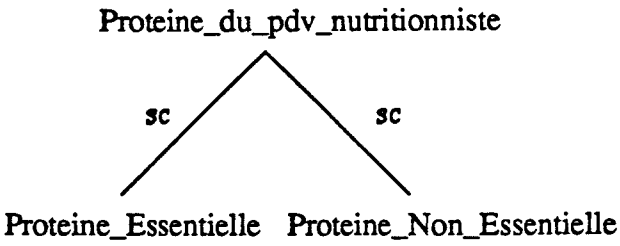
-du point de vue encombrement spatial:



Prenons un second exemple dans le domaine de la nutrition [Carré & Comyn 88]. Les protéines peuvent être classées du point de vue du biochimiste comme suit:

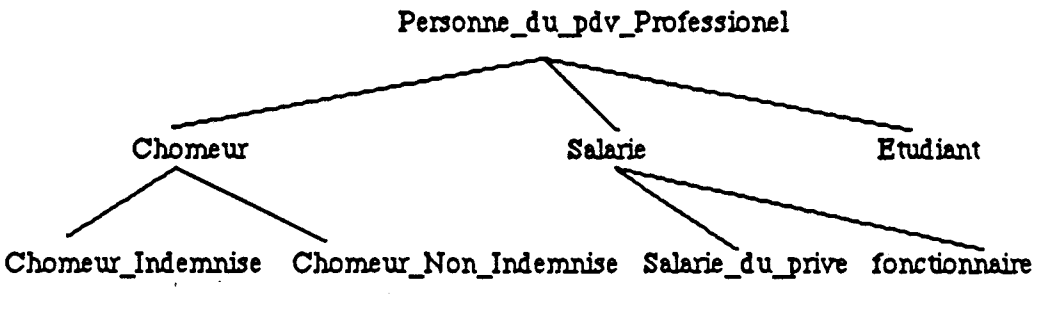


et du point de vue du nutritionniste en:

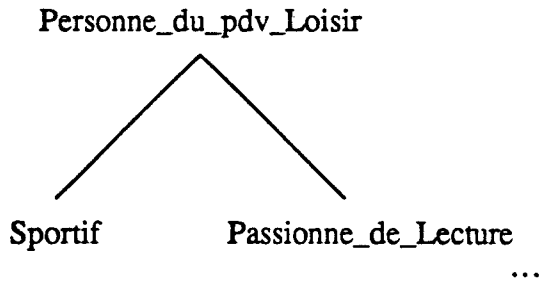


Considérons enfin l'exemple simple suivant concernant la classification multiple des personnes:

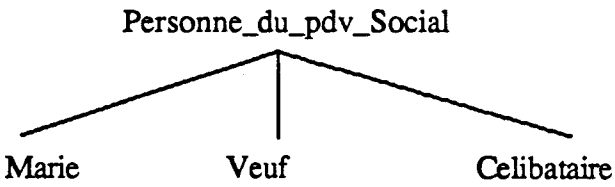
- du point de vue professionnel:



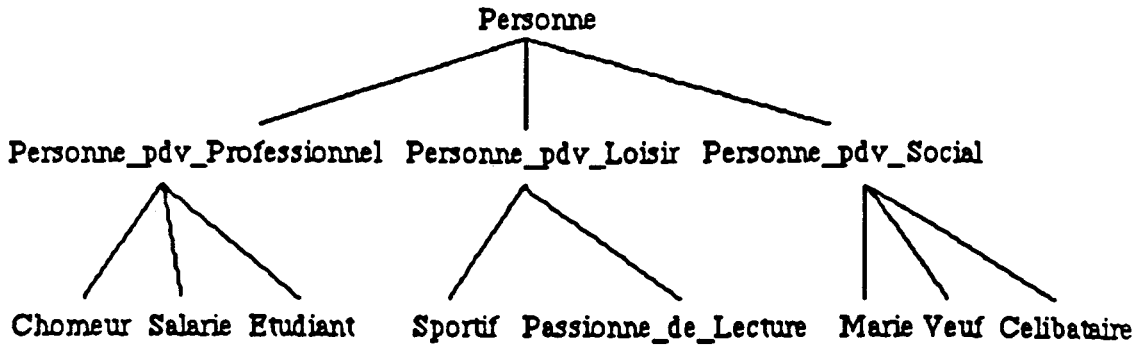
- du point de vue loisir:



et du point de vue social:



conduisant naturellement à la hierarchie de classes suivante (en simplifiant à outrance!):



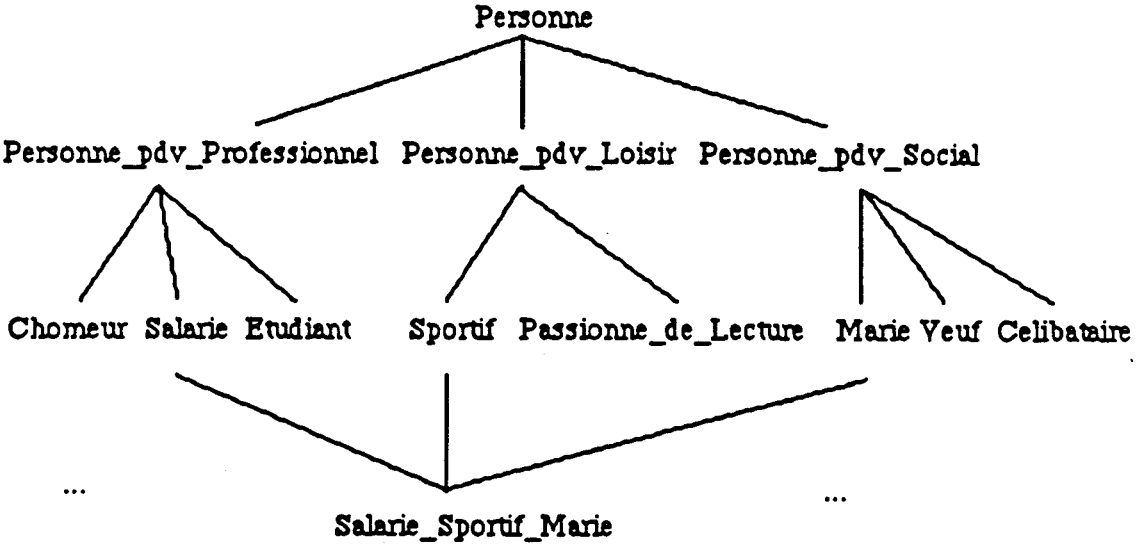
et de façon similaire pour les autres exemples.

Le problème est alors:

> Comment peut on déclarer qu'une personne, soit Jean, est salariée, sportive et mariée, soit dans la représentation par objet, que l'objet jean est représentant des classes Salarie, Sportif et Marie?

A cause de la contrainte de représentation unique l'approche classique oblige à créer une *sousclasse produit* de ces trois classes dont jean sera alors instance.On utilise par là *l'héritage multiple pour la représentation multiple.* Classiquement en effet le seul moyen de déclarer un objet représentant de plusieurs classes est de passer par l'héritage multiple au dessus de l'unique classe de représentation directe possible de l'objet, sa classe d'instanciation (III.3.1).

De manière générale ceci mène à créer toutes les combinaisons de classes possibles de la classification multiple ce qui induit un treillis des classes produit des hiérarchies, utilisant l'héritage multiple:



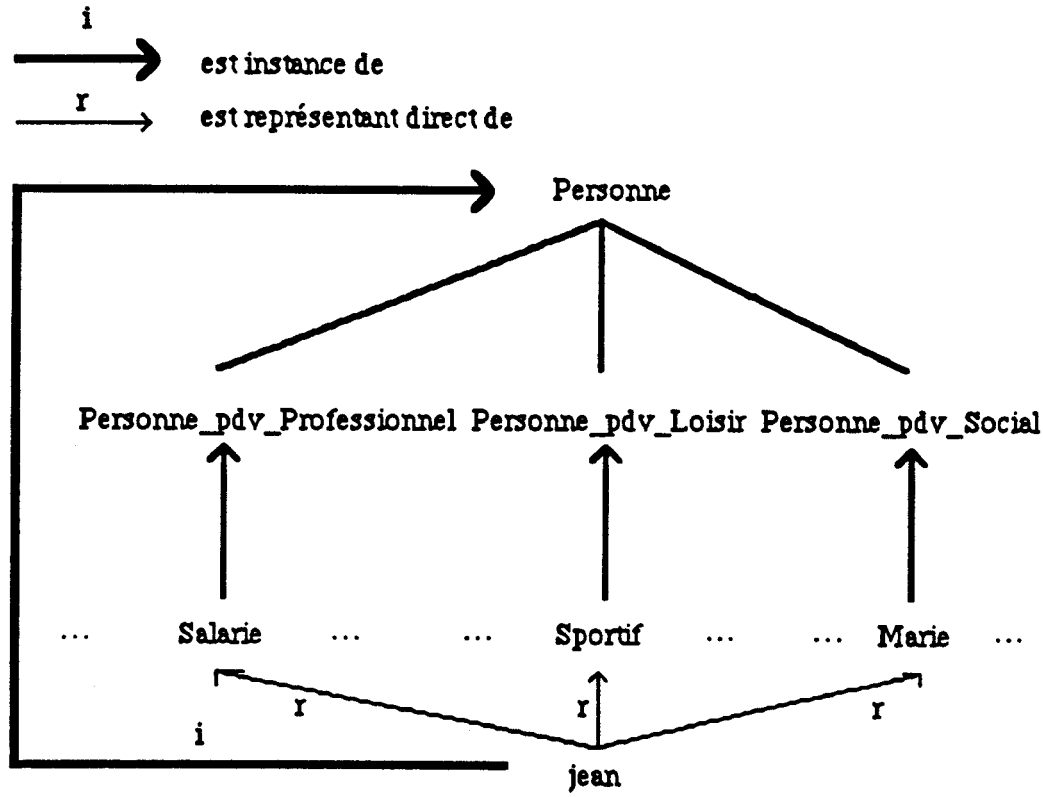
Il est facile de constater comment le treillis devient artificiellement complexe quand le nombre de classes augmente:

> "Above a certain level of complexity, finding a type with certain known characteristics can become difficult"
>
> [Lang & Pearlmutter 86]

et la gestion de cette "pléthore de combinaisons possibles de classes devient un problème épineux" (selon leurs propres termes).

Cette solution classique est d'autant moins satisfaisante si les classes à combiner sont complétement indépendantes et que leurs sousclasses produits n'ont de sens que vis à vis de l'instanciation. Notamment, dans la plupart des cas, de telles classes ne décrivent pas de modèle d'objet, i-e de caractérisations, et ne sont donc pas des classes de représentation au sens du IV.2.2: elles n'ont pas de sens vis à vis de la fonctionnalité minimum des classes qui est l'abstraction d'objets.

Du point de vue de la représentation des connaissances, ces classes n'ont pas de réalité pour l'expert. Celui-ci ne les fait pas apparaitre dans ses classifications, considérant implicitement pouvoir déclarer un objet représentant de plusieurs classes. Dans l'exemple précédent tout objet représentant de la classe Personne doit pouvoir être considéré comme représentant direct de plusieurs sousclasses de celle-ci. C'est le sens de la ***Représentation Multiple*** en Rome qui remet en cause la contrainte de représentation unique. Par cette notion, un objet instance d'une classe minimum, sa classe d'instanciation, peut être représentant direct de plusieurs sousclasses de représentation de celle-ci. Soit pour l'exemple:



Nous détaillerons la réalisation de cette notion en Rome au paragraphe IV.3.2, résolue de manière uniforme avec la notion d'évolution d'objet.

IV.3.1.3 Evolution d'objet

Le problème posé ici est celui de l'évolution de l'appartenance d'un objet à ses classes de représentation et implicitement de sa caractérisation. Supposons pour l'exemple des personnes que Jean qui était salarié devienne chômeur. Il s'agit ici de pouvoir déclarer que l'objet jean le modélisant n'est plus représentant de la classe Salarie mais devient représentant de la classe Chomeur et de modifier en conséquence sa caractérisation. Cette déclaration est impossible directement dans le modèle classique du fait de la Contrainte de Représentation Unique Figée qui fige le lien de représentation direct de l'objet à sa

création, c'est le lien d'instanciation. De façon induite, *l'ensemble de ses classes de représentation (son treillis de représentation) est figé et par conséquent sa caractérisation est complétement fixée à sa création.*

Nous pensons que la hiérarchie des classes représentant la connaissance d'un domaine est de ce fait sous utitisée notamment pour les objets eux-mêmes. En effet l'interprétation conceptuelle de celle-ci et sa sémantique de généralisation/ affinement de concepts ne peuvent être exploitées en tant que telles dans quelque *processus d'évolution* d'objet comme:

- l'évolution des objets au cours de leur vie (objets de "longue vie") comme ce pourrait être le cas sur l'exemple des personnes. Cette évolution peut s'effectuer à travers le treillis des classes courant. Celui-ci peut lui-même évoluer par l'ajout de nouvelles sousclasses selon un processus de conception incrémentale propre au style de programmation orienté objet (cf le chapitre sur la spécialisation, III.3.5.3). Cet affinement incrémental des classes doit pouvoir être utilisé pour les objets eux-mêmes, notamment ceux existant déjà dans l'environnement du problème.

- l'identification d'objets à travers la hiérarchie de classes par affinement de la connaissance que l'on peut disposer sur eux. C'est le cas par exemple dans tout problème de diagnostic de situations [Fikes & Kehler 85] sur une typologie de situations connues; ou dans un schéma de conception (CAO, génie logiciel), par exemple en électronique [Carré & Comyn 87a] où les composants du problèmes évoluent du niveau abstrait de conception fonctionnelle jusqu'à l'implantation physique selon des critères de choix propres à chaque niveau (optimisation fonctionnelle, techonologique ...)

- plus généralement dans tout processus de décision qui fait évoluer des objets de la connaissance.

C'est pourquoi, de façon à augmenter l'intérêt de la hiérarchie de classes comme modèle de généralisation/affinement des objets eux-mêmes, Rome propose la notion de ***Représentation Evolutive*** d'objet. Cette notion est fondée sur la remise en cause du caractère figé du lien de représentation direct et permet de rattacher dynamiquement un objet à des classes de représentation sousclasses plus affinées de sa classe minimum, sa classe d'instanciation. Ceci est résolu de manière uniforme avec la Représentation Multiple, ce qui conduit à la notion de ***Représentation Multiple et Evolutive.***

IV.3.2 Représentation Multiple et Evolutive, R.M.E

IV.3.2.1 Remise en cause de la Contrainte de Représentation Unique Figée

Le traitement de la R.M.E en Rome est fondé sur la remise en cause de la contrainte C.R.U.F pour une classe d'objets, les r-objets:

- la *Représentation Unique* par laquelle un objet n'est représentant direct que de sa classe d'instanciation est remplacée par la *Représentation Multiple* par laquelle un objet peut être représentant direct de plusieurs classes.

- la *Représentation Figée* par laquelle l'objet est complétement figé par sa classe d'instanciation à sa création est remplacée par la *Représentation Evolutive* par laquelle sa représentation multiple précédente peut évoluer à travers la hiérarchie des classes après sa naissance.

---

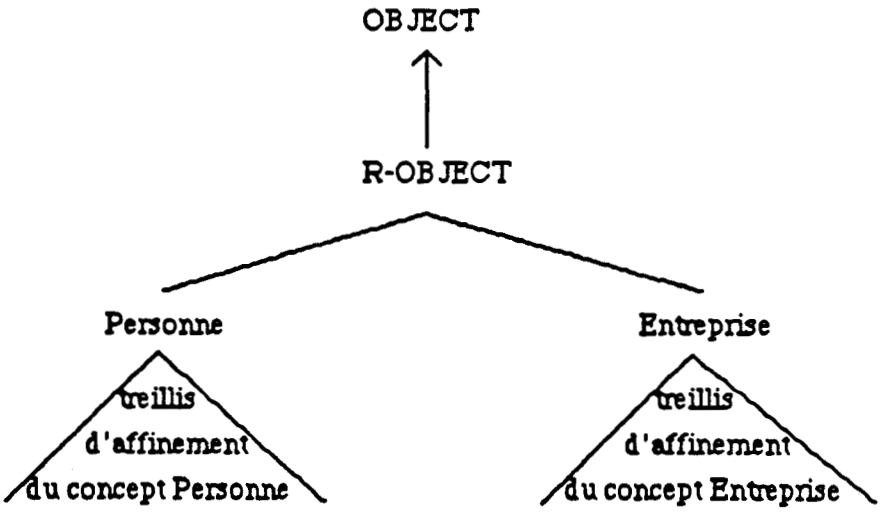
* selon une vision assez aristotélicienne des choses et même inéiste, une entité réelle ne pouvant jamais être un "seau vide" [Piatelli-Palmarini 79].

La R.M.E est définie par une classe R-OBJECT d'objets, les r-objets, sousclasse de OBJECT. Un r-objet peut dynamiquement appartenir à des sousclassses de sa classe d'instanciation. Vis à vis de la caractérisation de r-objets, celle-ci joue le rôle de *classe de représentation minimale et irrévocable* en dehors de laquelle l'objet n'a plus d'existence. Elle fixe la caractérisation minimale de l'objet *, notamment les caractéristiques nécessaires à la R.M.E qu'elle hérite de R-OBJECT.
L'objet est donc toujours au minimum représentant de sa classe d'instanciation et donc de ses surclasses, représentation minimale qu'il ne peut remettre en cause, notamment au cours de son évolution. Après sa création l'objet peut être déclaré, et ceci dynamiquement, représentant de sousclasses de sa i-classe qui affinent la caractérisation définie par celle-ci, éventuellement selon des points de vue différents. Ces *sousclasses de représentation potentielles* d'une classe d'instanciation n'ont d'intérêt que vis à vis de la fonctionnalité d'abstraction ou de généralisation de caractérisations possibles des instances de celle-ci, elles ne sont donc généralement que des r-classes (instances de R-CLASS).

La classe d'instanciation d'un r-objet limite en fait son treillis de représentation potentiel à l'intérieur duquel celui-ci peut évoluer; il ne peut s'affiner qu'en dessous de sa classe d'instanciation. Nous formulons ceci par le principe de représentation minimale:

> Principe de représentation minimale:
> Toute classe de représentation potentielle d'un r-objet est sous-classe de sa classe d'instanciation.

Ce principe est une garantie minimale de la cohérence de la R.M.E, qui contraint un objet à être toujours caractérisé à l'intérieur du domaine d'affinement du concept général correspondant à sa classe d'instanciation dont il est une incarnation. Montrons ceci sur l'exemple suivant:



Personne et Entreprise sont deux classes d'instanciation des concepts de personne et d'entreprise. Les treillis d'affinement dont elles sont respectivement sommets constituent les domaines de représentation potentielle de leurs instances. D'après le principe de représentation minimale, toute personne (respectivement entreprise) est toujours représentant au minimum de la classe Personne et ne peut être représentant de concepts indépendants, tels que Entreprise (respectivement Personne !).

Détaillons maintenant la classe R-OBJECT et la R.M.E en Rome.

IV.3.2.2 R-OBJECT: des objets pour la R.M.E

La classe R-OBJECT définit un modèle d'objets à R.M.E selon les idées présentées précédemment:

Représentation Multiple:
un r-objet peut être représentant direct de plusieurs sousclasses de sa classe d'instanciation auxquelles il est lié par son attribut *rclasses*, multivalué, qui en contient la liste.

Représentation Evolutive:
R-OBJECT définit deux méthodes pour la R.M.E:
- basicr+(C), C étant une classe de représentation potentielle du r-objet, il devient représentant de cette classe.
- basicr- (C), C étant une classe de représentation du r-objet vérifiant le principe de représentation minimale, il n'est plus représentant de C.
ces méthodes sont associées respectivement aux sélecteurs **r+** et **r-**. Elles manipulent l'attribut *rclasses* et par là réalisent la R.M.E.

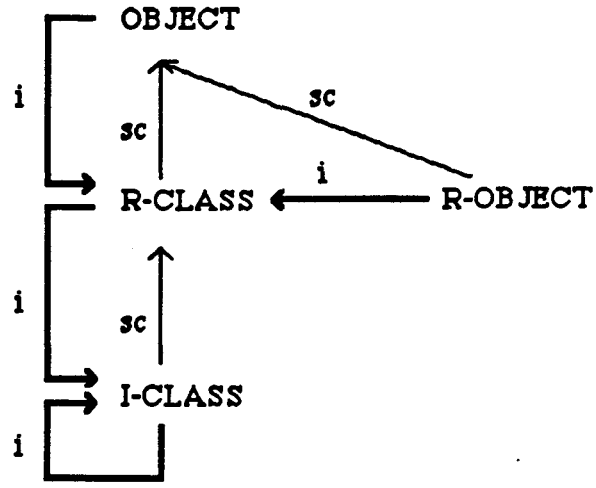
Précisons que l'attribut *rclasses* qui contient la liste des classes de représentation directe d'un objet est défini par OBJECT et donc hérité par R-OBJECT.

La classe **R-OBJECT** est donc définie comme suit:

```
[R-CLASS 'new
  'R-OBJECT
  'surcl '(OBJECT)
  'methodes
   '(basicr+ (lambda (C) (...))
     basicr- (lambda (C) (...)))
  'selecteurs
   '(r+ basicr+
     r- basicr-)]
```

L'association des méthodes de R.M.E à des sélecteurs permet de les appliquer par envoi de message à tout r-objet.

Insistons sur le fait que R-OBJECT est une sousclasse de OBJECT et non confondue avec cette dernière. Ceci a pour conséquence que seules les sousclasses de R-OBJECT sont des classes de r-objets, les autres sont des classes d'objets "classiques". Un r-objet se distingue d'un objet classique par le fait qu'il dispose des méthodes de R.M.E pour faire évoluer ses classes de représentation directe (attribut rclasses). Notons en particulier que, de ce fait, les classes elles-mêmes ne sont pas des objets à représentation multiple et évolutive. Leur classe (R-CLASS) n'est pas sous classe de R-OBJECT.



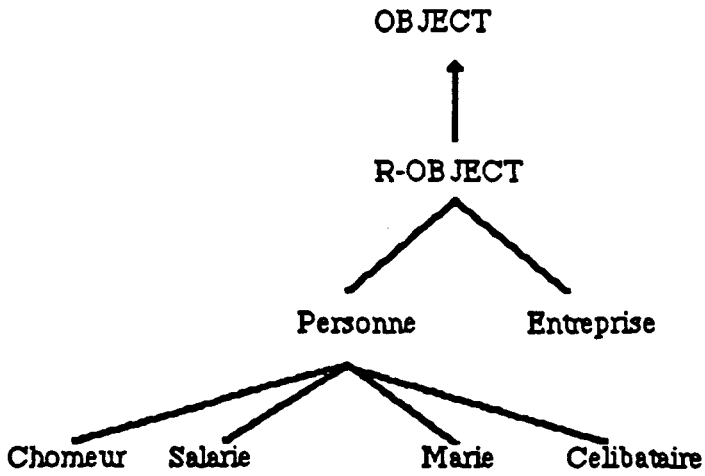
Les définitions de représentant et en particulier de représentant direct que nous avons adoptées (II.3.1) sont étendues ici à la R.M.E:

- tout objet est *représentant direct* de ses classes de représentation directe contenues dans son attribut rclasses. Le message suivant permet de tester ceci:
  [O 'repd? C] = true ssi O est représentant direct de C.

- tout objet est *représentant* d'une classe:
  - si il est représentant direct de celle-ci
  - ou si cette classe est surclasse d'une de ses classes de représentation directe.

le message correspondant est ici:
  [O 'repg? C] = true.

Présentons dès maintenant un petit exemple de R.M.E. Soit la classification:



Les classes Chomeur, Salarie, Marie, Celibataire sont les classes de représentation potentielles des instances de Personne. Soit donc la solution en Rome:

[I-CLASS 'new 'Personne 'surcl '(R-OBJECT)]

[R-CLASS 'new 'Chomeur 'surcl '(Personne)]

[R-CLASS 'new 'Salarie 'surcl '(Personne)]

[R-CLASS 'new 'Marie 'surcl '(Personne)]

[R-CLASS 'new 'Celibataire 'surcl '(Personne)]

[I-CLASS 'new 'Entreprise 'surcl '(R-OBJECT)]

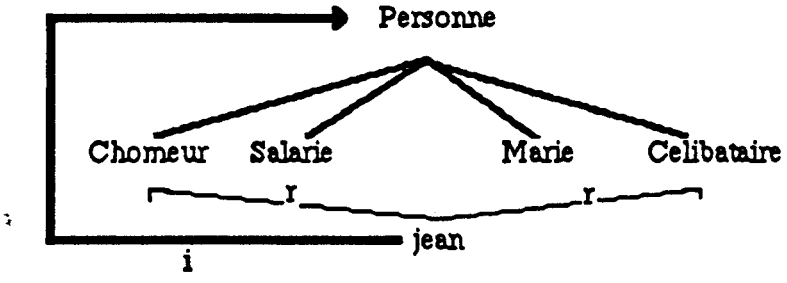
jean est une personne:

> [Personne 'new 'jean]
= jean

il est chômeur et célibataire:

> [jean 'r+ 'Chomeur]
= jean
> [jean 'r+ 'Celibataire]
= jean

l'état de représentation de jean est alors:



Inspectons les attributs iclass et rclasses de jean:

> [jean 'inspect]
inspect> {? 'iclass}
= Personne
inspect> {? 'rclasses}
= (Celibataire Chomeur)
inspect> {fin}
= jean

jean se marie et trouve du travail, il n'est donc plus représentant de Celibataire:

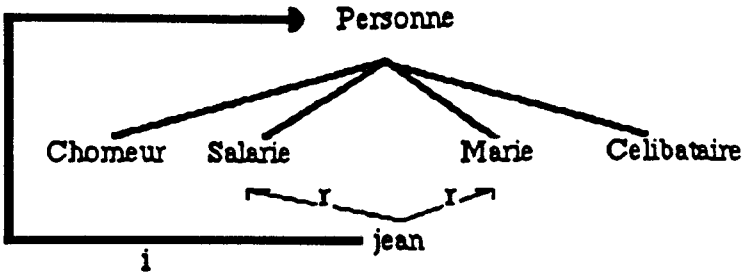
> [jean 'r- Celibataire]

mais de Marie:

> [jean 'r+ Marie]

d'autre part il n'est plus représentant de Chomeur mais de Salarie:

> [jean 'r- Chomeur]
> [jean 'r+ Salarie]



Testons la représentation de jean:

> [jean 'repg? Chomeur]
= ()
> [jean 'repg? Salarie] ;et même plus précisément repd?
= t
et toujours:
> [jean 'repg? Personne]
= t

La classe Entreprise ne vérifie pas le principe de représentation minimale (P.R.M) pour les instances de Personne, elle ne fait pas partie de leur domaine de R.M.E. L'objet jean ne peut donc en être représentant:

> [jean 'r+ Entreprise]
ERREUR: ... Entreprise ne vérifie pas P.R.M ...

Personne est la classe de représentation minimale et irrévocable de ses instances d'où:

> [jean 'r- Personne]
ERREUR: ... Personne ne vérifie pas P.R.M ...

Maintenant, affinons incrémentalement le treillis de représentation par l'ajout de sousclasses d'un point de vue loisirs, telles que Passionne_de_lecture, Sportif et par l'affinement de Chomeur en Chomeur_indemnise et Chomeur_non_indemnise:

[R-CLASS 'new 'Passionne_de_lecture 'surcl '(Personne)]

[R-CLASS 'new 'Sportif 'surcl '(Personne)]

[R-CLASS 'new 'Chomeur_indemnise 'surcl '(Chomeur)]

[R-CLASS 'new 'Chomeur_non_indemnise 'surcl '(Chomeur)]

Ces sousclasses deviennent des nouvelles classes de représentation potentielles pour les instances de Personne, notamment celles existant déjà, telles que jean, qu'il est facile d'affiner sous le nouveau point de vue et d'affiner sa représentation en tant que chômeur en chômeur indemnisé par exemple:

> [jean 'r+ Sportif]
> [jean 'r+ 'Chomeur_indemnise]

il est resté représentant de ses anciennes classes, avec en plus ces nouvelles:

> [jean 'repg? Chomeur]
= t
> [jean 'inspect]
inspect> {? 'rclasses}

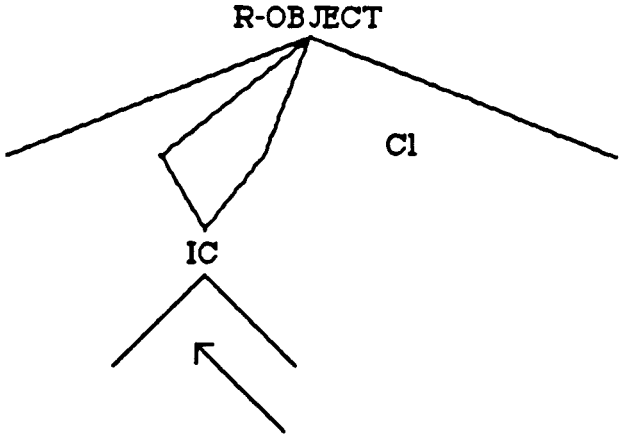
= (Sportif Celibataire Chomeur_indemnise)
inspect> {fin}
= jean

IV.3.2.2.1 Sémantique des méthodes de R.M.E

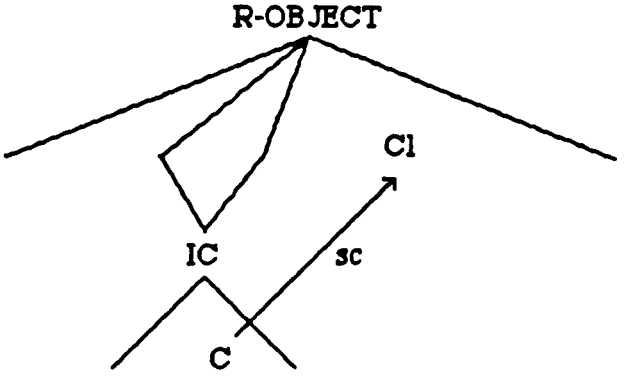
Les méthodes de R.M.E basicr+ et basicr- de R-OBJECT associées aux sélecteurs r+ et r- vérifient une sémantique bien précise que nous allons dans un premier temps introduire intuitivement puis formaliser.

D'après le principe de représentation minimale, le domaine de R.M.E des r-objets instances d'une i-classe est réduit aux sousclasses de celle-ci, que nous appellerons leurs *classes de représentation potentielles*. Les opérations r+ et r- pour de tels r-objets ne sont définies que sur ce domaine. Ceci est avant tout dû à un souci de cohérence de la R.M.E pour limiter le domaine d'affinement d'un concept général (modélisé par la classe d'instanciation) aux concepts plus fins qui en dérivent selon un certain processus de classification. La réalisation du *principe de représentation minimale* a donc pour but d'empêcher une entité de sortir de son domaine de cohérence, i-e pour un objet de devenir représentant de classes indépendantes (au sens exact du IV.2.5.2) de sa classe de représentation minimale.

Soit IC une classe d'instanciation de r-objets et Cl une classe indépendante de IC:



Une instance de IC ne peut devenir représentant de Cl, ni représentant direct, ni représentant au sens large. En particulier ceci est un garde-fou pour empêcher des constructions utilisant l'héritage multiple pour sortir du domaine de cohérence du concept:



Bien que C soit sousclasse de IC, les instances de celle-ci ne peuvent être représentants de C puisq'elles seraient de ce fait représentants de toutes ses surclasses, en particulier de classes telles que Cl (indépendante de IC) qui sortent du domaine d'affinement de IC. Nous pouvons ainsi préciser le principe de représentation minimale comme suit:

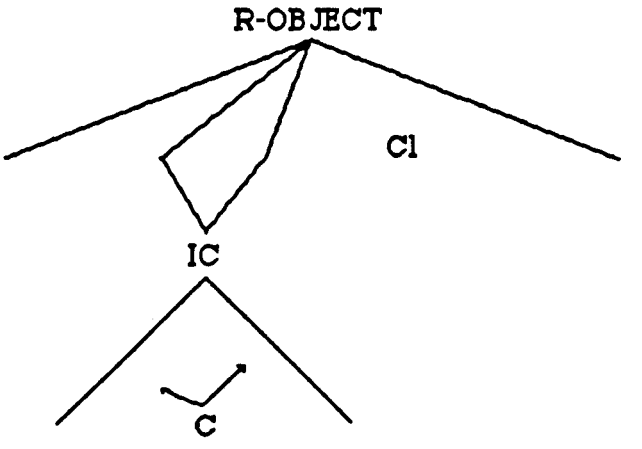
> **Principe de représentation minimale:**
> **Toute classe de représentation potentielle d'un r-objet est sousclasse stricte de sa classe d'instanciation.**

avec la définition suivante:

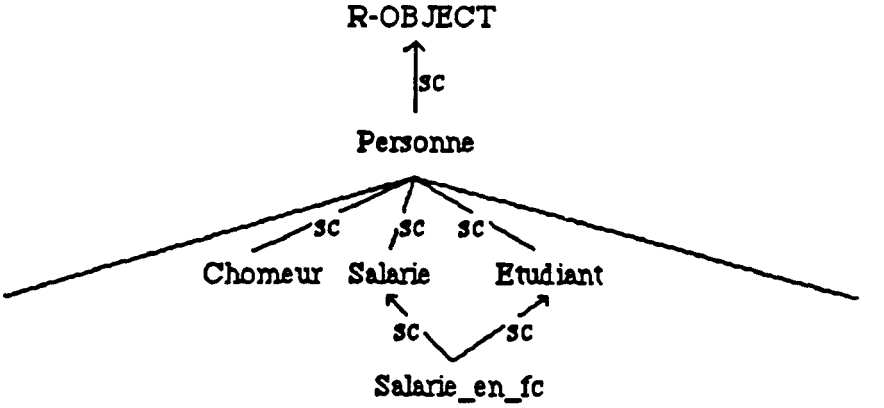
> C1 est sousclasse stricte de C2
> si aucune de ses surclasses n'est indépendante de C2.

Ce principe n'empêche cependant pas l'héritage multiple à l'intérieur du domaine de représentation, i-e entre sousclasses strictes:

> une classe de représentation potentielle peut être sousclasse de plusieurs autres classes de représentation potentielles.



ex: la classe Salarie_en_fc, affinement de Salarie et Etudiant à l'intérieur du domaine de représentation potentielle de Personne



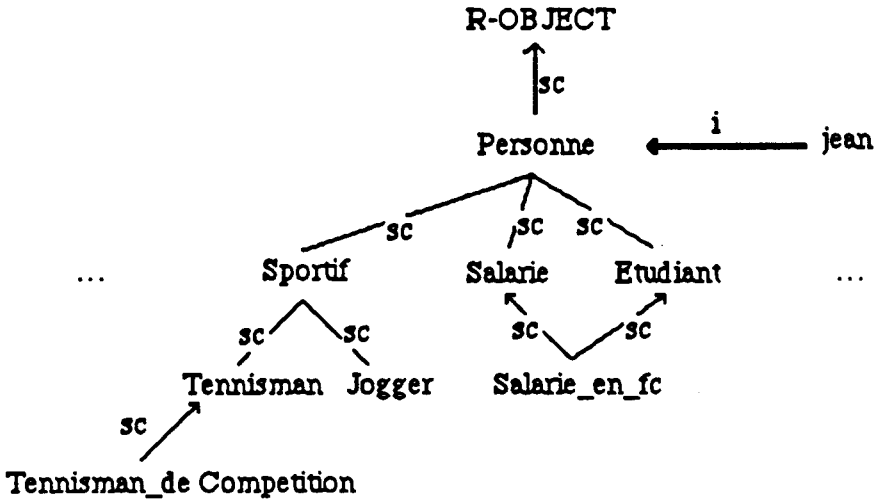
La *sémantique intuitive* des opérations r+ et r- est la suivante:

Soit IC une classe d'instanciation de r-objets et soit O une instance de IC

[O 'r+ C]
La classe C doit être une classe de représentation potentielle de O, i-e une sousclasse stricte de IC. Si O est déjà représentant de C, cette opération laisse invariant son treillis de représentaion. Sinon O devient représentant direct de C et par transitivité représentant de toutes les surclasses de C.

[O 'r- C]
La classe C doit être sousclasse stricte de IC. Si O n'est pas représentant de C, ce message laisse invariant son treillis de représentation. Sinon O n'est plus représentant de C ni par, transitivité, de toutes ses sousclasses. Il reste représentant des surclasses de C et des autres classes dont il était représentant, indépendantes de C.

Montrons ceci sur l'exemple que nous reprenons:



jean est salarie en formation continue, fait du tennis de compétition et du jogging:

> [jean 'r+ Salarie_en_fc]
> [jean 'r+ Tennisman_de Competition]
> [jean 'r+ Jogger]

jean est donc représentant direct de ces trois classes, ce que l'on peut tester en demandant l'état de son attribut rclasses par le message de même nom (défini par OBJECT):

> [jean 'rclasses]
= (Jogger Tennisman_de_Competition Salarie_en_fc)

il est donc représentant de toutes leurs surclasses, ce que nous testons en lui demandant tout son treillis de représentation par le message rep-o (défini par OBJECT):

> [jean 'rep-o]
= (Jogger Tennisman_de_Competition Tennisman Sportif Salarie_en_fc Salarie Etudiant Personne R-OBJECT OBJECT)

jean a arrêté le tennis de compétition:

> [jean 'r- Tennisman_de_Competition]

il reste représentant des surclasses de celle-ci, soient Tennisman et ses surclasses, et des autres classes indépendantes dont il était représentant, soit Salarie_en_fc, Jogger et leur surclasses:

> [jean 'rclasses]
= (Tennisman Jogger Salarie_en_fc)
> [jean 'rep-o]
= (Tennisman Jogger Sportif Salarie_en_fc Salarie Etudiant Personne R-OBJECT OBJECT)

jean a terminé sa formation, il n'est plus étudiant:

> [jean 'r- Etudiant]

il n'est donc plus représentant ni de Etudiant ni de ses sousclasses, soit Salarie_en_fc:

> [jean 'repg Salarie_en_fc]
= ()

mais reste Salarie, qui est indépendant de Etudiant, ainsi que tennisman et jogger:

> [jean 'rclasses]
= (Salarie Tennisman Jogger)
> [jean 'rep-o]
= (Salarie Tennisman Jogger Sportif Personne R-OBJECT OBJECT)

Les opérations r+ et r- effectuent donc des mises à jour du treillis de représentation du r-objet par retrait ou ajout de classes de représentation selon une sémantique que nous formalisons ici.

Les définitions du IV.2.5.2 sont étendues à la R.M.E dans le sens où la représentation directe unique classique due à la contrainte C.R.U.F est remplacée par la représentation directe multiple. Ainsi soit O l'ensemble des objets, C l'ensemble des classes

Classes de représentation d'un objet

Soit l'application suivante :

repd: $O \rightarrow 2^C$
repd(O) = l'ensemble de ses classes de représentation directe

c'est à dire la valeur de son attribut rclasses.

Pour un objet non r-objet, cet ensemble est réduit à sa classe d'instanciation

$$i(O) \notin \text{R-OBJECT} \Rightarrow \text{repd}(O) = \{i(O)\}$$

l'ensemble des classes de représentation de O est alors:

$$\text{Rep}(O) = \{C' \in O \,/\, \exists\, C \in \text{repd}(O)\,,\, C' \geq C\}$$

Les autres définitions restent dès lors inchangées.

Formalisons le principe de représentation minimale:

définition:

soient C1, C2 $\in$ C, C2 est sousclasse stricte de C1, noté C2 $<_S$ C1
<=> C2 < C1 et Dep(C2) $\subset$ Dep(C1)

Montrons que cette définition est équivalente à la définition initiale:

Dep(C1) = Pred(C1) $\cup$ Succ(C1)
Dep(C2) = Pred(C2) $\cup$ Succ(C2)

C2 < C1 => Succ(C2)$\subset$ Succ(C1)$\subset$ Dep(C1)

et par la définition initiale:
C2 sousclasse stricte de C1
=> $\forall$ C $\in$ Pred(C2), C ne peut être indépendante de C1 d'où C $\in$ Dep(C1)
=> Pred(C2) $\subset$ Dep(C1)

cqfd

propriété

C2 $<_S$ C1 => Dep(C2) = Succ(C2) $\cup$ [C2 C1] $\cup$ Pred(C1)

- Dep(C2) $\supset$ Succ(C2) $\cup$ [C2 C1] $\cup$ Pred(C1)

  • Succ(C2) $\subset$ Dep(C2)
  • C2 < C1 => [C2 C1] $\subset$ Pred(C2) $\subset$ Dep(C2)
  • C2 < C1 => Pred(C1) $\subset$ Pred(C2) $\subset$ Dep(C2)

- Dep(C2) $\subset$ Succ(C2) $\cup$ [C2 C1] $\cup$ Pred(C1)

Dep(C2) = Pred(C2) $\cup$ Succ(C2)
il suffit donc de montrer que:

Pred(C2) $\subset$ Succ(C2) $\cup$ [C2 C1] $\cup$ Pred(C1)

$\forall$ C $\in$ Pred(C2) telle que C $\neq$ C2 , C $\notin$ Succ(C2)
reste donc à prouver que: C $\in$ [C2 C1] $\cup$ Pred(C1)

C $\in$ Pred(C2) $\subset$ Dep(C2) $\subset$ Dep(C1) = Succ(C1) $\cup$ Pred(C1)
soit C $\in$ Pred(C1)
soit C $\in$ Succ(C1) et C $\in$ Pred(C2) => C $\in$ Succ(C1) $\cap$ Pred(C2) = [C2 C1]

cqfd

Principe de représentation minimale

Soit O un r-objet, C $\in$ C est une classe de représentation potentielle de O
ssi C $<_S$ i(O).

propriété

Soit O un r-objet, $Rep(O) = Pred(i(O)) \cup \bigcup_{C \in repd(O)} [C\ i(O)]$

en effet:

$$Rep(O) = \{C' \in C \ / \ \exists C \in repd(O), C' \geq C\} = \bigcup_{C \in repd(O)} Pred(C)$$

or $\forall\ C \in repd(O), C <_s i(O) \Rightarrow Pred(C) = [C\ i(O)] \cup Pred(i(O))$
d'où le résultat.

Classes de R.M.E d'un r-objet:

Soit O un r-objet (i(O) < R-OBJECT), c'est l'ensemble des sousclasses strictes de sa classe d'instanciation, défini comme suit:

$$RME(O) = \{C \in C / C <_s i(O)\}$$

Treillis de représentation potentielle d'un r-objet:

Soit O un r-objet (i(O) < R-OBJECT), ce treillis (par abus de langage) est défini ainsi:

$$TRP(O) = (RME(O) \cup \{i(O)\}, \leq)$$

$\leq$ étant la relation d'ordre classe/surclasse.Une conséquence du principe de représentation minimale est que i(O) est l'élément maximal de TRP(O).

Famille de R.M.E d'un objet

$F \subset C$ est une famille de R.M.E d'un r-objet O si

1) $F \subset TRP(O)$
2) $\forall\ C, C' \in C$
   $i(O) \geq C \geq C'$ et $C' \in F \Rightarrow C \in F$ (propriété de filtre)

La condition 2) est équivalente à: $\forall\ C \in F, [C\ i(O)] \subset F$

Famille de R.M.E courante d'un objet

C'est la partie évolutive du treillis de représentation de l'objet. D'après une propriété montrée précédemment, celui-ci est égal à:

$$Rep(O) = \underbrace{\bigcup_{C \in repd(O)} [C\ i(O)]}_{\text{famille de R.M.E courante de O}} \cup \underbrace{Pred(i(O))}_{\text{partie figée du treillis de représentation de O}}$$

Sémantique algébrique de r+ et r-

Les opérations r+ et r- traduisent la transformation de la famille de R.M.E courante de l'objet respectivement par ajout et retrait d'une classe de représentation ($\in$ R.M.E(O)). Elles sont définies par les opérations algébriques de même nom suivantes:

Soit F l'ensemble des familles de R.M.E de l'objet:

r+ : $F \times RME(O) \rightarrow F$

$r+(F,C) = \bigcap \{F' / F' \in F \text{ et } F' \supset F + \{C\}\}$

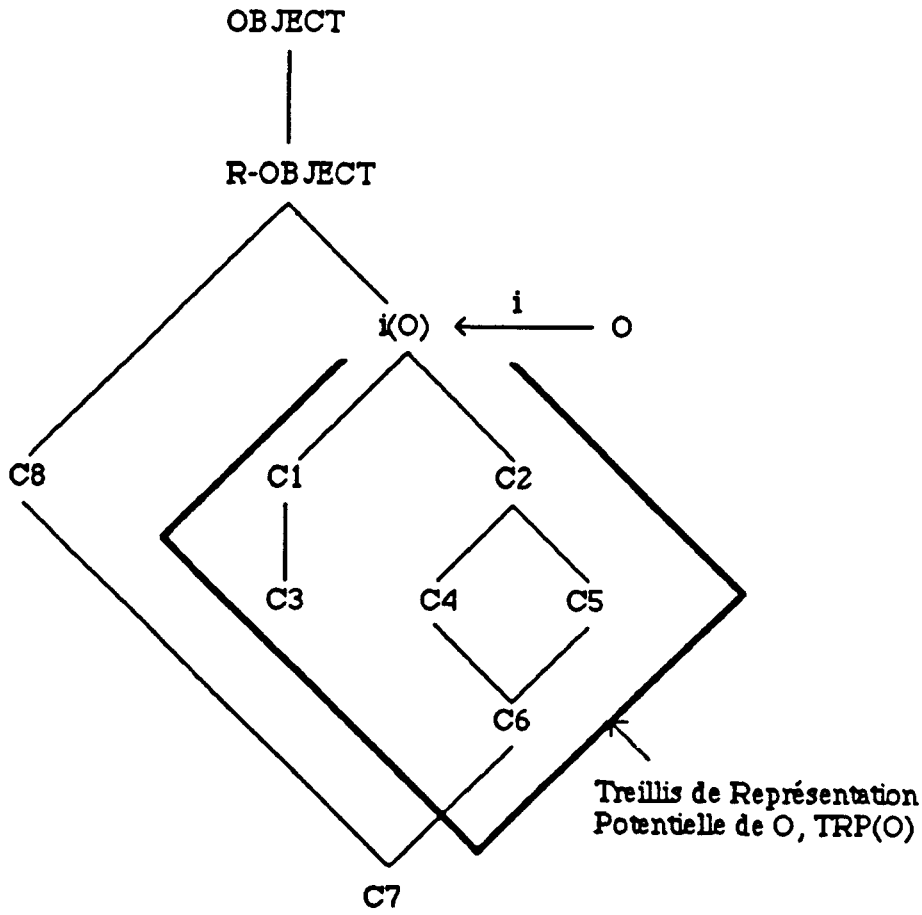
r- : $F \times RME(O) \rightarrow F$

$r-(F,C) = \bigcup \{F' / F' \in F \text{ et } F' \subset F - \{C\}\}$

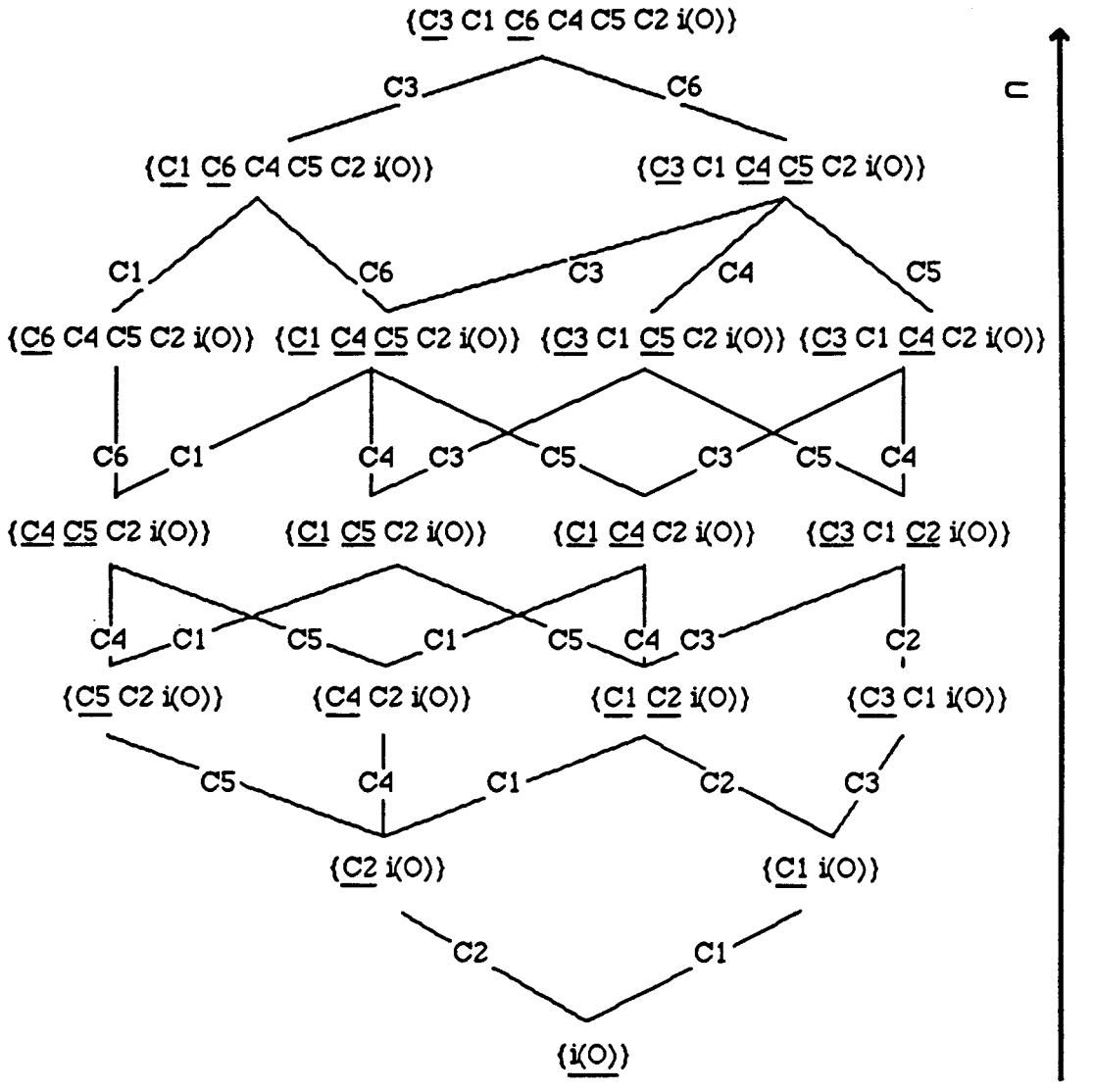
Intuitivement, F étant la famille de R.M.E courante de O, C une classe de R.M.E de O:

- par r+, la nouvelle famille de R.M.E de O est la plus petite famille contenant l'ancienne famille F à laquelle est ajoutée C
- par r-, la nouvelle famille de R.M.E de O est la plus grande famille contenue dans l'ancienne famille F privée de C.

Montrons ceci sur l'exemple suivant:

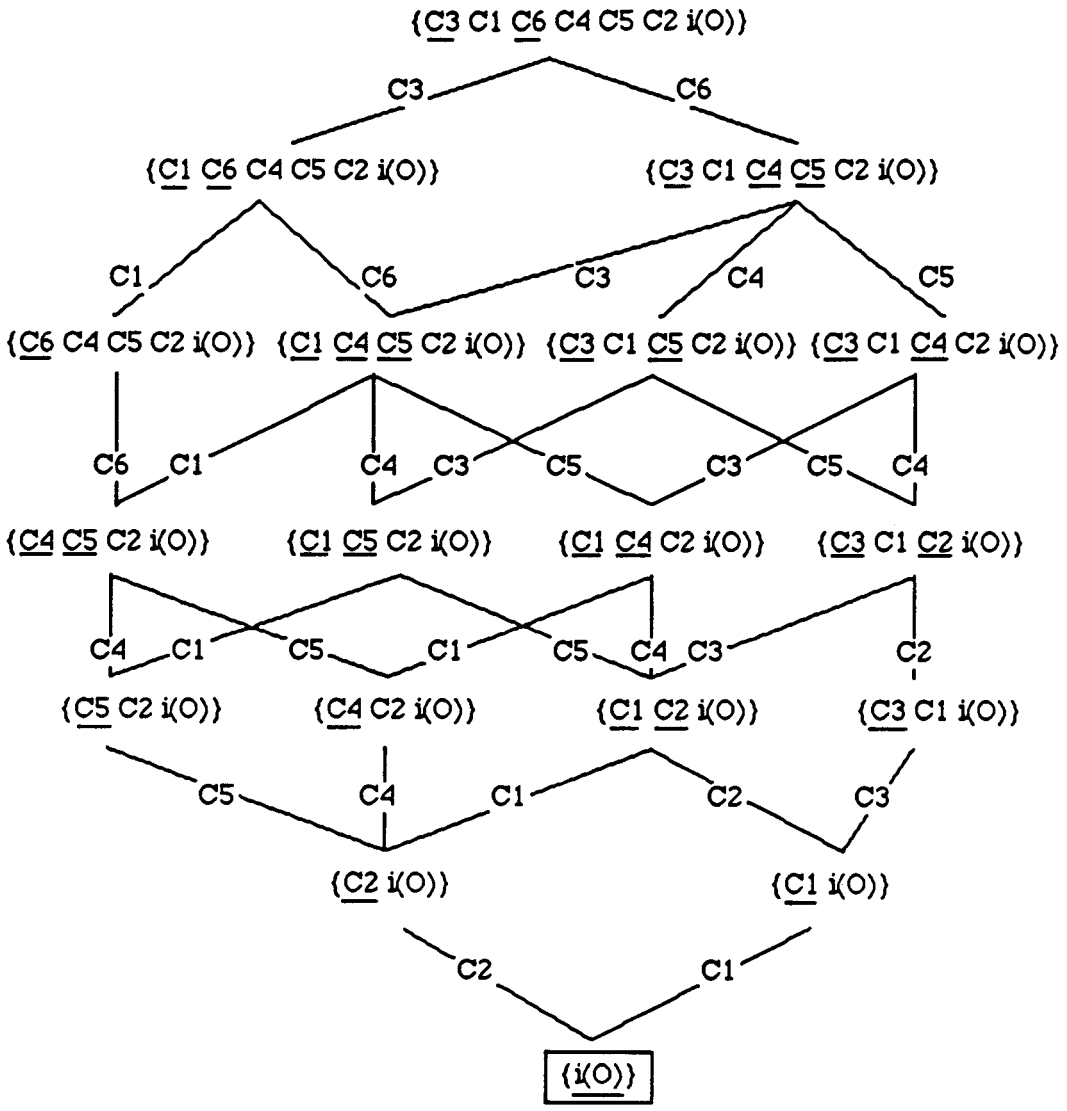


Le Treillis de Représentation Potentielle de O est figuré, il ne contient pas C7, qui n'est pas une classe de R.M.E de O, elle n'est pas sousclasse stricte de O. La structure par inclusion des familles de R.M.E de O, soit l'ensemble F, est la suivante:

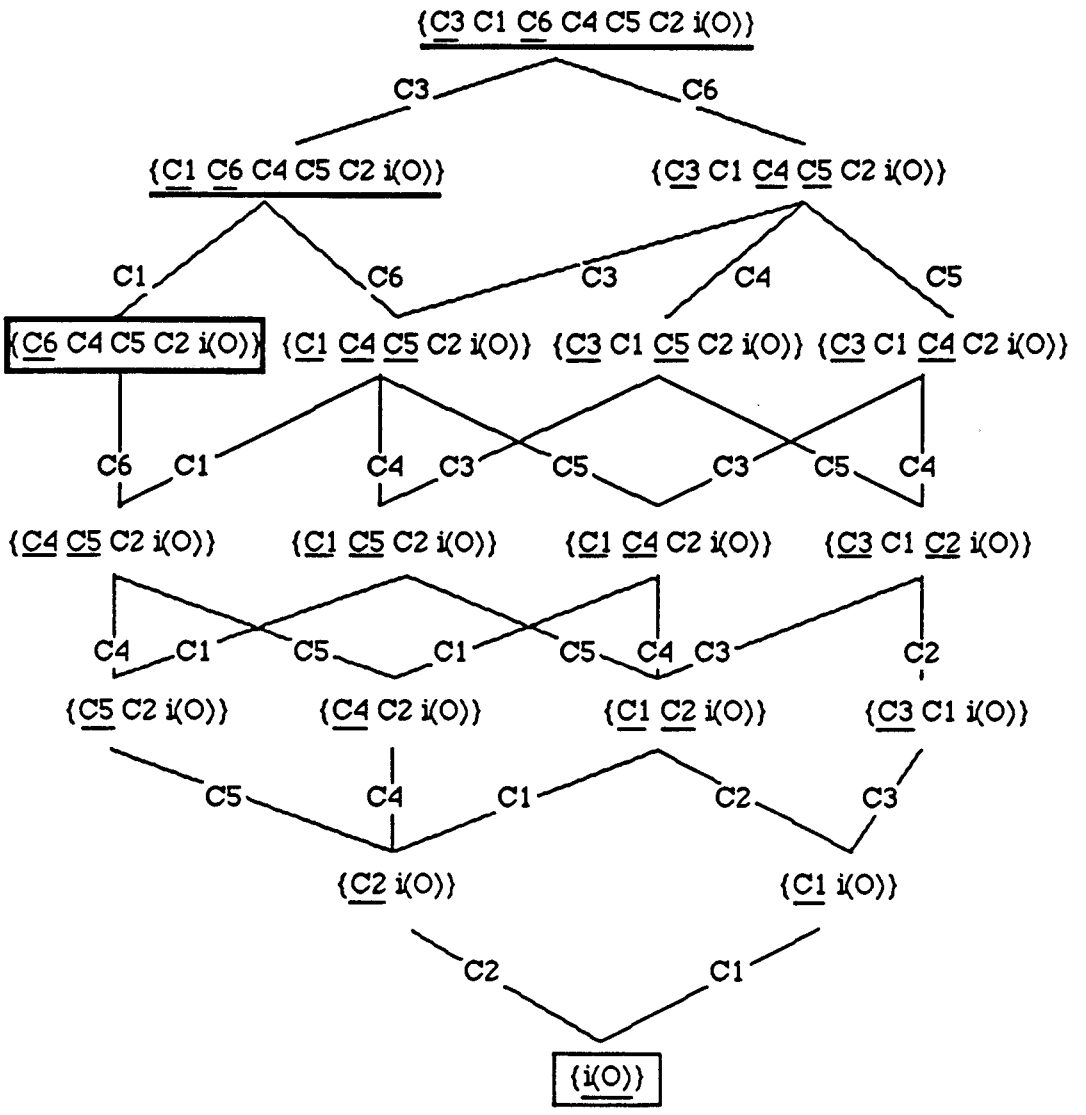


En sommet la famille {C3 C1 C6 C4 C5 C2 i(O)} est la famille de R.M.E maximale de O. La famille de R.M.E minimale, au niveau le plus bas, est celle réduite à sa classe d'instanciation. Les classes soulignées sont les classes de représentation directe de la famille, i-e les éléments de repd(O). Cette structure se construit facilement en enlevant (r-) à chaque famille d'un niveau, chacune de ses classes de représentation directe, ce qui produit le niveau inférieur, jusqu'à la classe de représentation directe minimale, i(O). Ce procédé est figuré par l'étiquetage de l'arc d'inclusion par la classe de représentation directe enlevée.

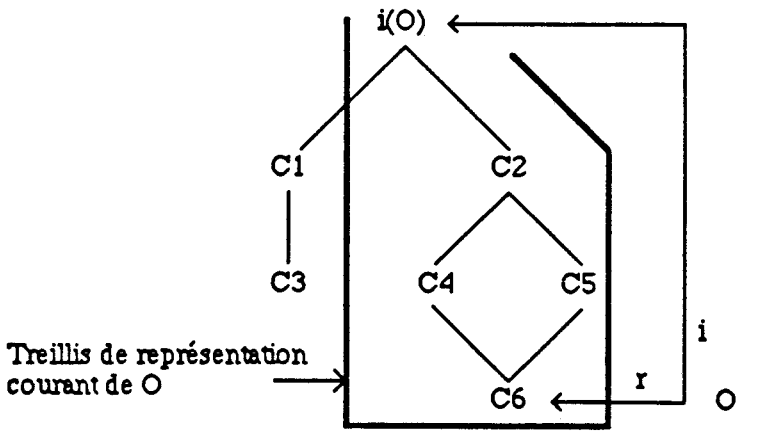
A l'instanciation de l'objet, sa famille de R.M.E est sa famille minimale. La famille de R.M.E courante de O est encadrée:



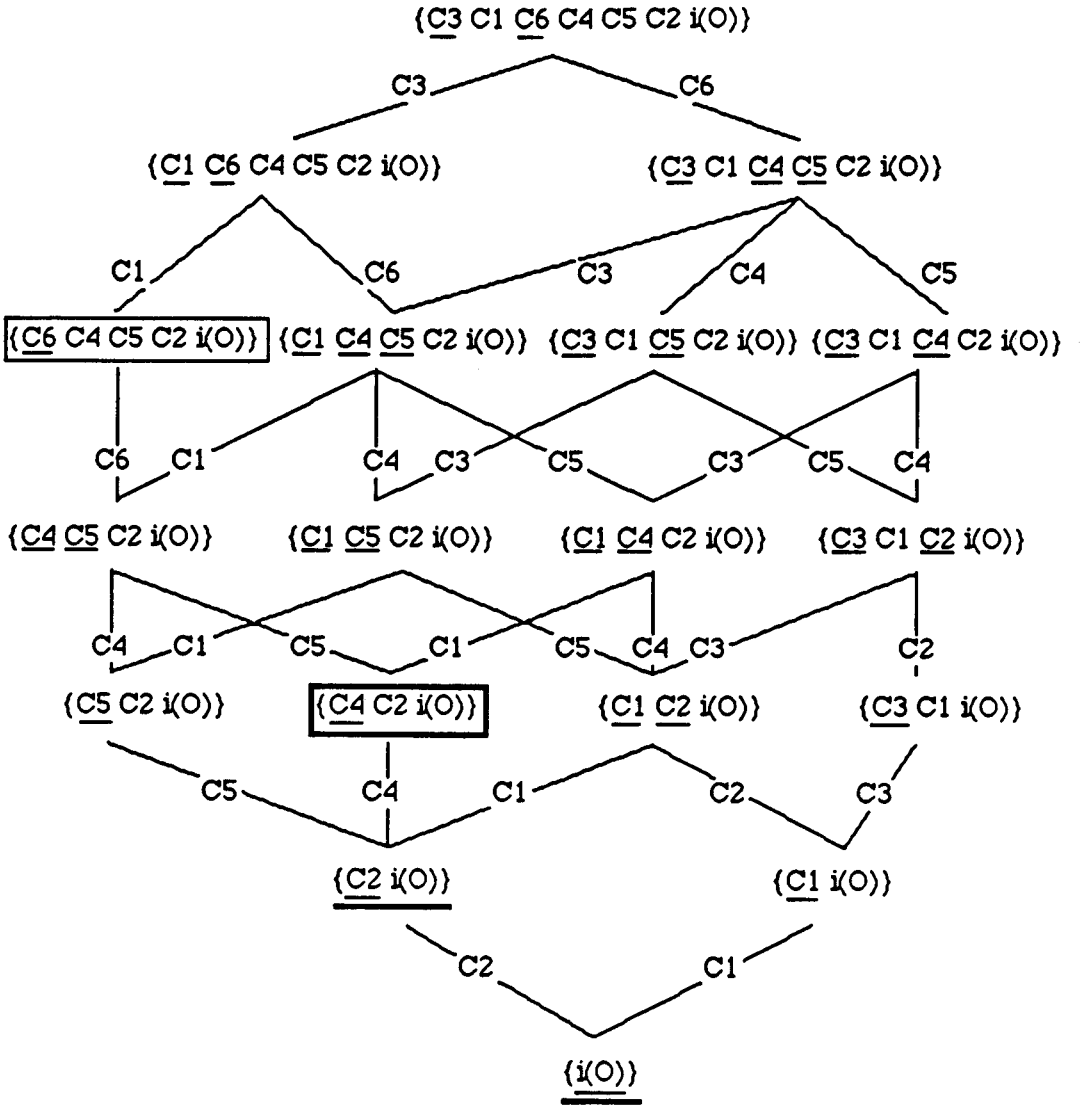
Appliquons r+({i(O)} , C6), les familles contenant {C6,i(O)} sont soulignées en gras et la plus petite encadrée en gras, elle devient la famille de R.M.E courante de O:



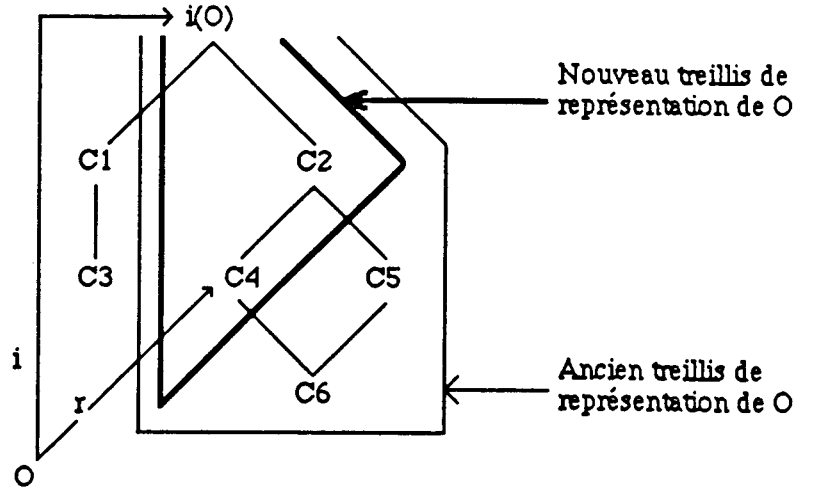
Ce qui est bien l'effet souhaité:



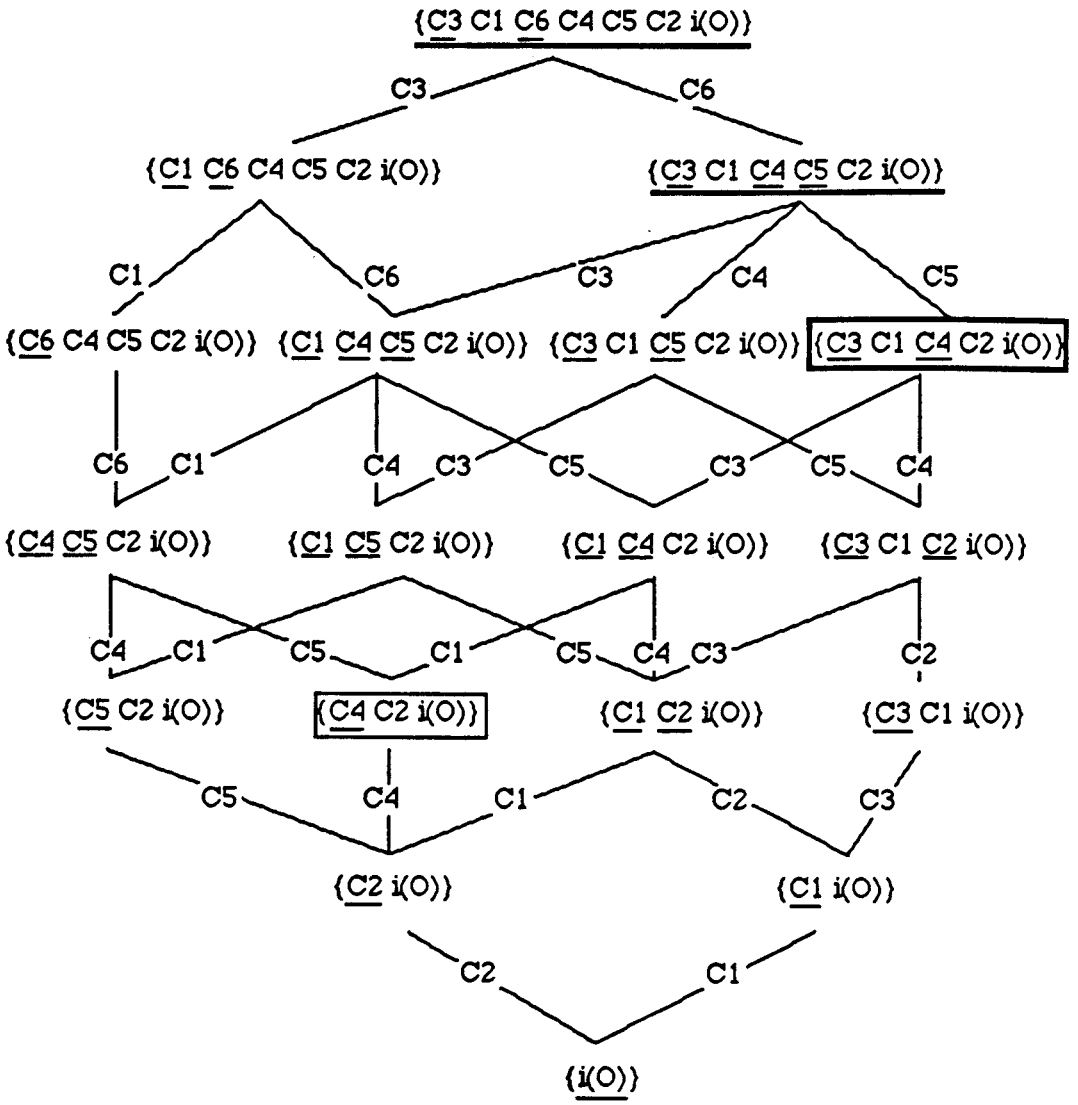
Appliquons maintenant r-({C6 C4 C5 C2 i(O)} , C5), les familles contenues dans {C6 C4 C2 i(O)} sont soulignées en gras et la plus grandre encadrée en gras elle devient la nouvelle famille de R.M.E de O:



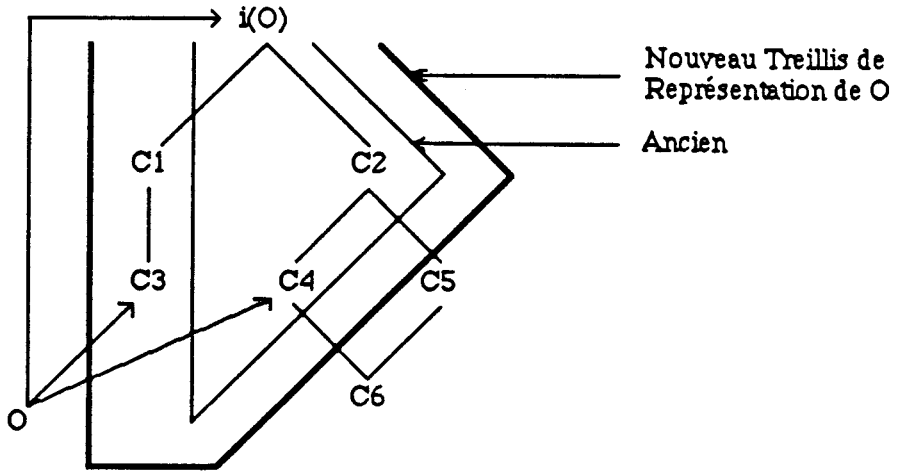
Ce qui correspond bien à:



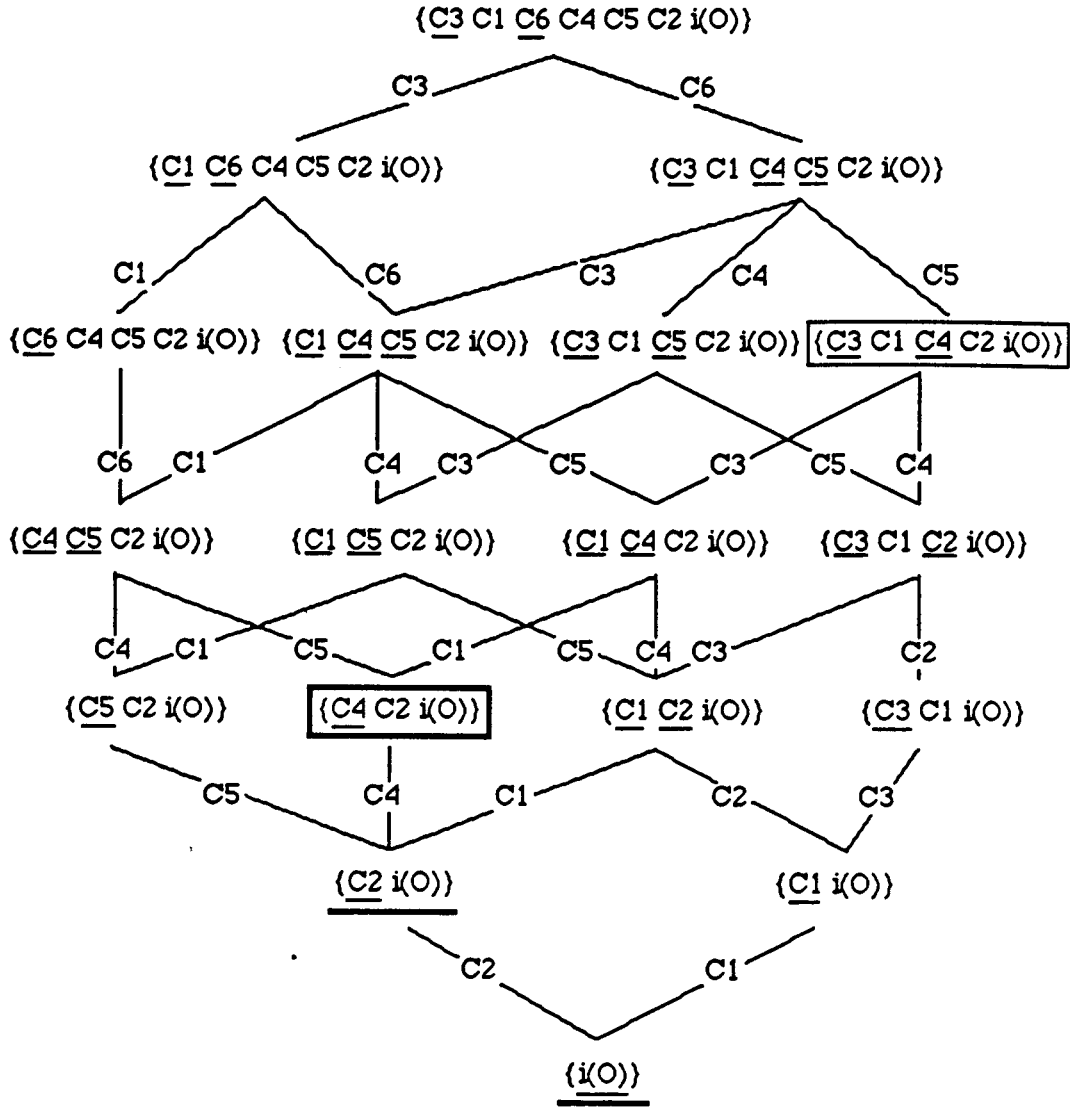
Appliqons r+({C4 C2 i(O)} , C3):



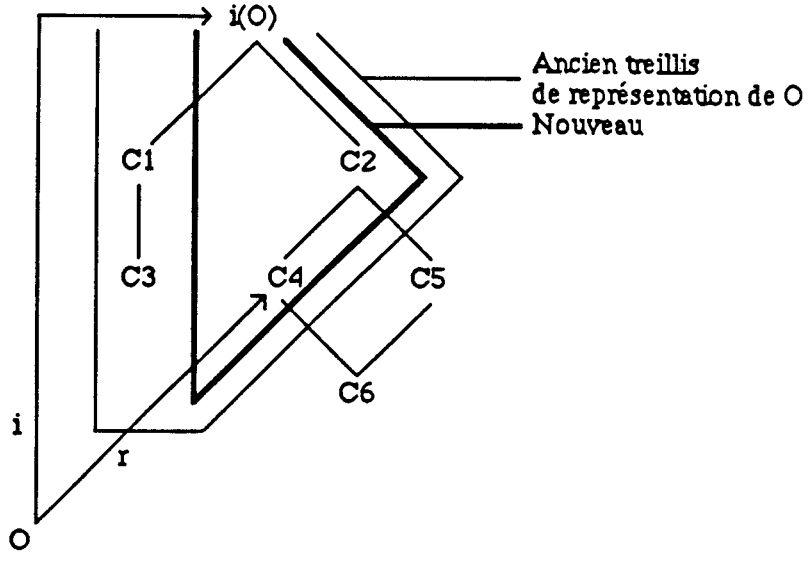
La plus petite famille contenant {C3 C4 C2 i(O)} est {C3 C1 C4 C2 i(O)}:



Enfin effectuons r- ({C3 C1 C4 C2 i(O)} , C1):



La plus grande famille contenue dans {C3 C4 C2 i(O)} est {C4 C2 i(O)}:



IV.3.2.3 R.M.E et héritage:

L'héritage, en tant que mécanisme d'inférence de la caractérisation d'un objet sur son treillis de représentation, présenté au paragraphe IV.2.5, est le même ici moyennant la définition de ce treillis, donnée au paragraphe précédent, qui prend en compte la représentation multiple et évolutive pour les r-objets:

$$Rep(O) = \{ C' \in C / \exists\ C \in repd(O), C' \geq C \}$$

Rétrospectivement le mécanisme initial correspond au cas particulier où:

$$repd(O) = \{i(O)\}$$

ce qui est toujours le cas pour les objets "classiques" de Rome (objets non r-objets), pour lesquels il a été défini initialement, la représentation directe étant alors réduite et figée au seul lien d'instanciation.

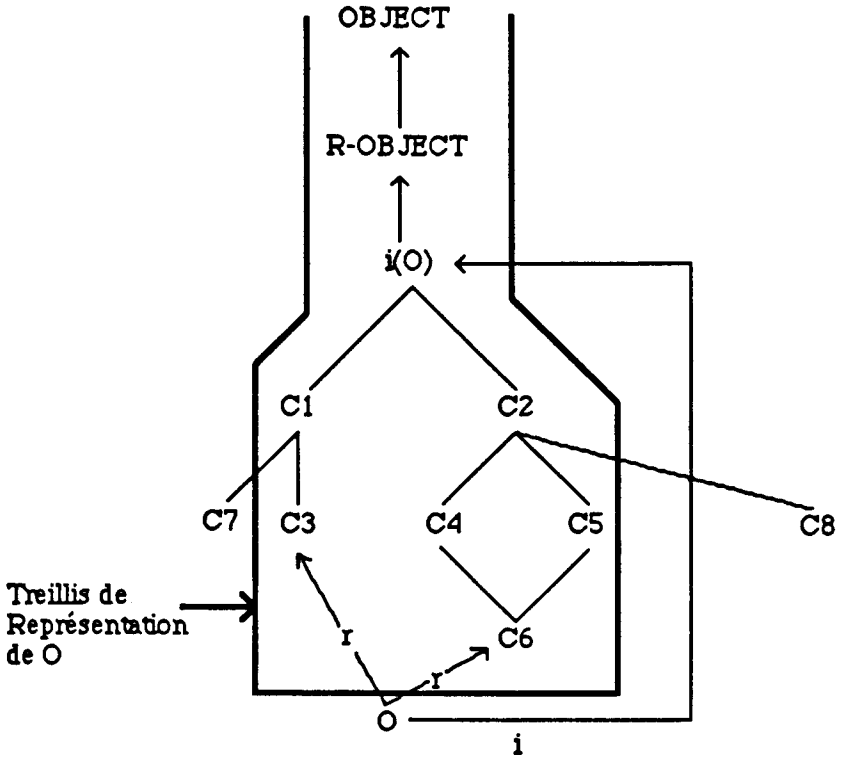
Pour les r-objets l'héritage tient donc toujours compte de leur treillis de représentation courant:

> un r-objet dispose de toutes les caractéristiques inférées sur son treillis de représentation courant selon le même mécanisme d'héritage par point de vue que les objets "classiques", avec le respect notamment de la règle du plus affiné et du principe d'indépendance.

Les modifications du treillis de représentation dues à l'application des méthodes de R.M.E induisent implicitement une mise à jour de la caractérisation de l'objet qui se comporte toujours selon sa représentation courante:

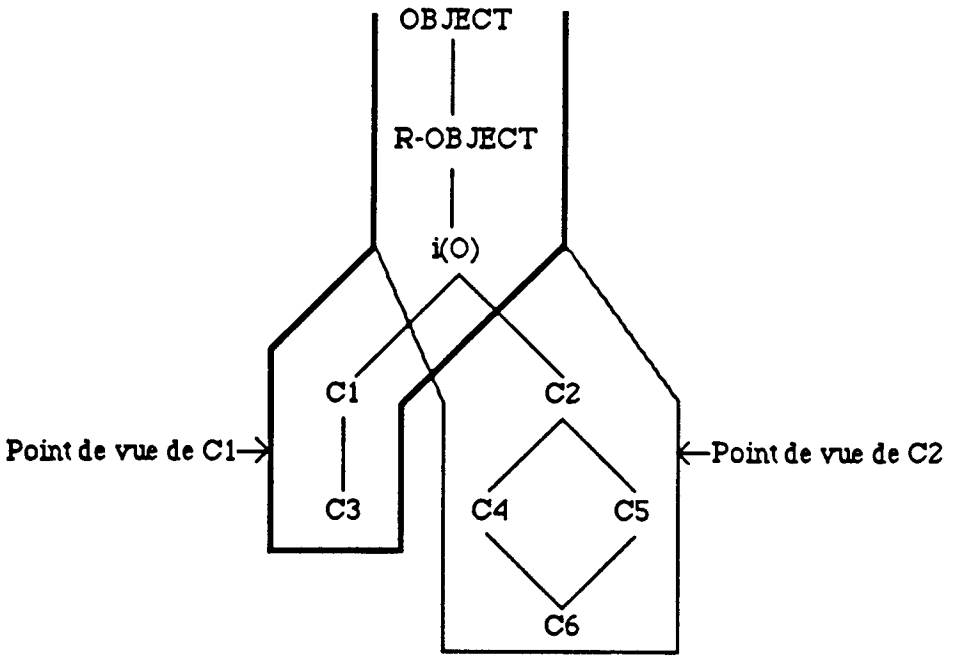
> l'application des oprérations r+ et r- induisent une évolution de la caractérisation du r-objet

Le respect du principe d'indépendance est d'autant plus crucial ici que la R.M.E favorise la classification multiple, sous des points de vue indépendants, d'un même objet. La sélection de point de vue conserve la même sémantique, cette fois sur le treillis de représentation multiple et évolutive. La définition d'un point de vue reste en effet inchangée moyennant l'extension de Rep(O) précisée au début de ce paragraphe. Montrons ceci sur l'exemple informel suivant:



Rep(O)= (C3 C1 C6 C4 C5 C2 i(O) R-OBJECT OBJECT).

Montrons maintenant les points de vue de C1 et C2 sur O en ne représentant que son treillis de représentation:



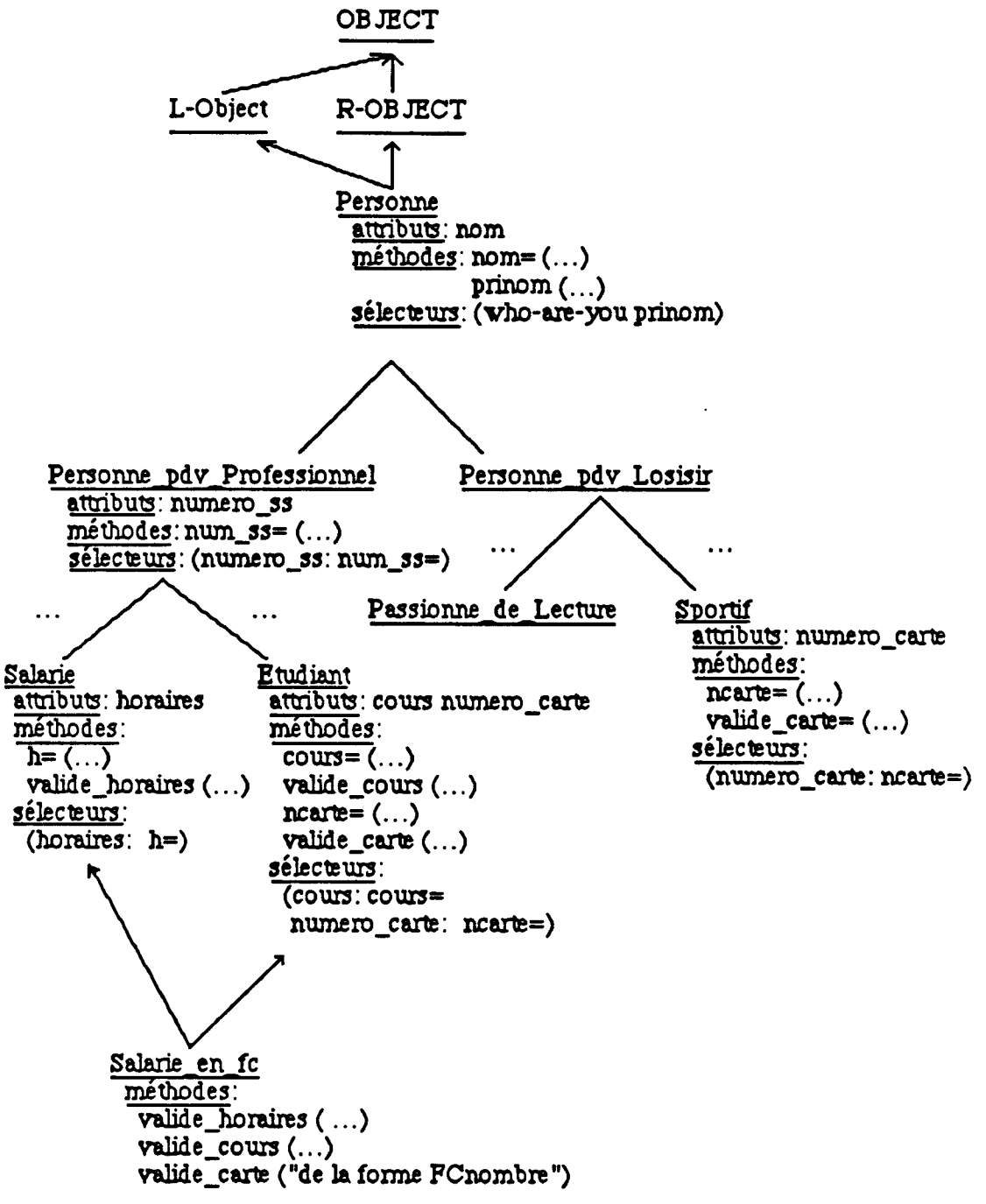
en effet:

Dep (C1) = (C7 C3 C1 i(O) R-OBJECT OBJECT)
=> pdv (C1, O) = Dep(C1) ∩ Rep (O) = (C3 C1 i(O) R-OBJECT OBJECT)

Dep(C2) = (C6 C4 C5 C8 C2 i(O) R-OBJECT OBJECT)
=> pdv (C2, O) = Dep(C2) ∩ Rep(O) = (C6 C4 C5 C2 i(O) R-OBJECT OBJECT)

Les points de vue présentent ici un intérêt tout particulier pour sélectionner une classification spécifique parmi les classifications multiples de l'objet. Sur l'exemple précédent C1 et C2 peuvent être considérées comme les sommets respectifs de deux classifications des instances de la classe i(O). Leur point de vue respectif permet de considérer l'une ou l'autre des classifications et le mécanisme d'inférence des points de vue assure la cohérence de la sélection au cours de l'activité de l'objet sous l'un ou l'autre des points de vue.

Le mécanisme d'héritage par point de vue ayant été présenté dans ses détails au paragraphe IV.2.5 , nous allons simplement le confronter ici à la représentation évolutive puis à la représentation multiple à travers un exemple, qui reprend ceux du IV.2.5.5.2:



Les instances de Personne sont classées sous les points de vue professionnel, correspondant à la classe Personne_pdv_Professionnel, et loisir, correspondant à la classe Personne_pdv_Loisir. La mixin L-Object déjà vue au paragraphe IV.2.5.5.2 nous permet l'accès aux attributs en lecture. La programmation en Rome de cet exemple est:

```
[I-CLASS 'new 'Personne
 'surcl '(L-Object R-OBJECT)
 'attributs '(nom)
 'methodes
  '(nom= (lambda (nom) (when (stringp nom) {<- 'nom nom}))
    prinom (lambda () (print "Mr " {? 'nom})))
 'selecteurs '(who-are-you prinom)]

;du point de vue professionnel

[R-CLASS 'new 'Personne_pdv_Professionnel
 'surcl '(Personne)
 'attributs '(numero_ss)
 'methodes '(num_ss= (lambda (n) (when (numberp n) {<- 'numero_ss n})))
 'selecteurs '(numero_ss: num_ss=)]

[R-CLASS 'new 'Salarie
 'surcl '(Personne_pdv_Professionnel)
 'attributs '(horaires)
 'methodes
  '(h= (lambda (n) {valide_horaires n} {<- 'horaires n})
    valide_horaires (lambda (n) (if (< n 40) t {erreur "horaires < 40"})))
  'selecteurs '(horaires: h=)]

[R-CLASS 'new 'Etudiant
 'surcl '(Personne_pdv_Professionnel)
 'attributs '(cours numero_carte)
 'methodes
  '(cours= (lambda (n) {valide_cours n} {<- 'cours n})
    valide_cours (lambda (n) (if (<= n 35) t {erreur "cours <= 35"}))
    ncarte= (lambda (n) {valide_carte n} {<- 'numero_carte n})
    valide_carte (lambda (n)
                   (if (and (equal (sref x 1) "E") (ndigitp (substring x 2 (1- (slen x)))))
                       t
                       {erreur "numero_carte de la forme Enombre"})))
  'selecteurs '(horaires: h= numero_carte: ncarte=)]
```

```
[R-CLASS 'new 'Salarie_en_fc
 'surcl '(Salarie Etudiant)
 'methodes
  '(valide_horaires
    (lambda (x)
     (let ((cours {? 'cours}))
      (if (or (equal cours '??) ;indetermine initialisé à l'instanciation
              (>= x cours))
          t
          {erreur "cours <= horaires "})))
   valide_cours
    (lambda (x)
     (let ((horaires {? 'horaires}))
      (if (or (equal horaires '??) ;indetermine
              (<= x horaires))
          t
          {erreur "cours <= horaires "})))
  valide_carte (lambda (n)
                 (if (and (equal (substring x 1 2) "FC")
                          (ndigitp (substring x 2 (1- (slen x)))))
                     t
                     {erreur "numero_carte de la forme FCnombre"}))]

;du point de vue loisirs

[R-CLASS 'new 'Personne_pdv_Loisirs 'surcl '(Personne)

[R-CLASS 'new 'Passionne_de_Lecture 'surcl '(Personne_pdv_Loisirs)]

[R-CLASS 'new 'Sportif
 'surcl '(Personne_pdv_Loisirs)
 'attributs '(numero_carte)
 'methodes
 '(ncarte= (lambda (n) {valide_carte n} {<- 'numero_carte n})
  valide_carte (lambda (n)
                 (if (and (equal (sref x 1) "S") (ndigitp (substring x 2 (1- (slen x)))))
                     t
                     {erreur "numero_carte de la forme Snombre"})))
 'selecteurs '(numero_carte: ncarte=)]
```

IV.3.2.3.1 Représentation Evolutive et héritage:

Soit l'objet jean instance de Personne:

> [Personne 'new 'jean 'nom "Jean Martin"]

Sa caractérisation est réduite pour l'instant à celle inférée à partir de sa seule classe de représentation directe, i-e sa classe d'instanciation Personne:

> [jean 'rclasses]
= (Personne)
> [jean 'who-are-you]
Mr Jean Martin
> [jean '? 'nom]
Jean Martin

Nous savons que jean est salarié en formation continue:

> [jean 'r+ Salarie_en_fc]

son treillis de représentation est alors:

> [jean 'rep-o]
= (Salarie_en_fc Salarie Etudiant Personne_pdv_Professionnel Personne L-Object R-OBJECT OBJECT)
> [jean 'rclasses]
= (Salarie_en_fc)

Sa caractérisation a évolué en conséquence, il dispose notamment des attributs numero_ss de Personne_pdv_professionnel, horaires de Salarie, cours et numero_carte de Etudiant. D'aprés la redéfinition des méthodes *valide_carte valide_horaires* et *valide_cours* de sa classe de représentation Salarie_en_fc, son numéro de carte d'étudiant doit commencer par "FC" et ses horaires et cours doivent être tels que "cours ≤ horaires":

> [jean 'numero_ss: 1621062193067]
= 1621062193067
> [jean 'numero_carte: "E12"]
... ERREUR de la forme FCnombre ...
> [jean 'numero_carte: "FC12"]
= FC12
> [jean 'horaires: 30]
= 30
> [jean 'cours: 20]
= 20

Jean a terminé sa formation, il n'est plus étudiant:

> [jean 'r- Etudiant]

son treillis de représentation est maintenant:

> [jean 'rep-o]
= (Salarie Personne_pdv_Professionnel Personne L-Object R-OBJECT OBJECT)

et ses caractéristiques ont évolué de fait; il est resté Salarie et a donc conservé ses attributs numero_ss et horaires:

```
> [jean '? 'numero_ss]
= 1621062193067
> [jean '? 'horaires]
= 30
```

en tant que salarié (uniquement et non plus salarié en formation continue) ses horaires peuvent aller jusqu'à 39 heures:

```
> [jean 'horaires: 39]
= 39
> [jean '? 'horaires]
= 39
```

il n'est plus représentant de Etudiant et n'a donc plus de raison de disposer d'une carte:

```
> [jean '? 'numero_carte]
... ERREUR attribut numero_carte non herite ...
```

IV.3.2.3.2 Représentation Multiple et héritage multiple par points de vue:

Nous allons montrer l'*intérêt typique du mécanisme des points de vue pour la sélection de classifications* indépendantes des personnes, sous les aspects professionnel et loisir. Nous reprenons pour cela le petit exemple précédent en insistant cette fois sur l'indépendance de ces deux points de vue telle qu'elle a été montrée au paragraphe IV.2.5.5.2. La nuance porte ici sur l'absence de la classe produit "Etudiant_Sportif" qui devient superflue du fait de la représentation multiple. L'héritage garantit ici les mêmes propriétés, les deux classes de représentation potentielles Etudiant et Sportif étant indépendantes, ce qui parait ici encore plus flagrant.

Soit paul, instance de Personne et représentant de Salarie_en_fc du point de vue professionnel et Sportif du point de vue loisir:

```
> [Personne 'new 'paul 'nom "paul ochon"]
= paul
> [paul 'r+ 'Salarie_en_fc]
= paul
> [paul 'r+ 'Sportif]
= paul
```

son treillis de représentation est alors:

```
> [paul 'rclasses]
= (Sportif Salarie_en_fc)
> [paul 'rep-o]
= (Salarie_en_fc Salarie Etudiant Personne_pdv_Professionnel Sportif
Personne_pdv_Loisir Personne L-Object R-OBJECT OBJECT)
```

les points de vue professionnel, par la classe Etudiant, et loisir, par Sportif, définissent tous deux un attribut *numero_carte*, les méthodes associées *ncarte=,valide_carte*, et le sélecteur *numero_carte*, de façon quasiment identique (volontairement !) mais indépendantes. La sélection de point de vue sur paul assure cette indépendance:

```
> [paul '(numero_carte: as Etudiant) "FC13"]
= FC13
```

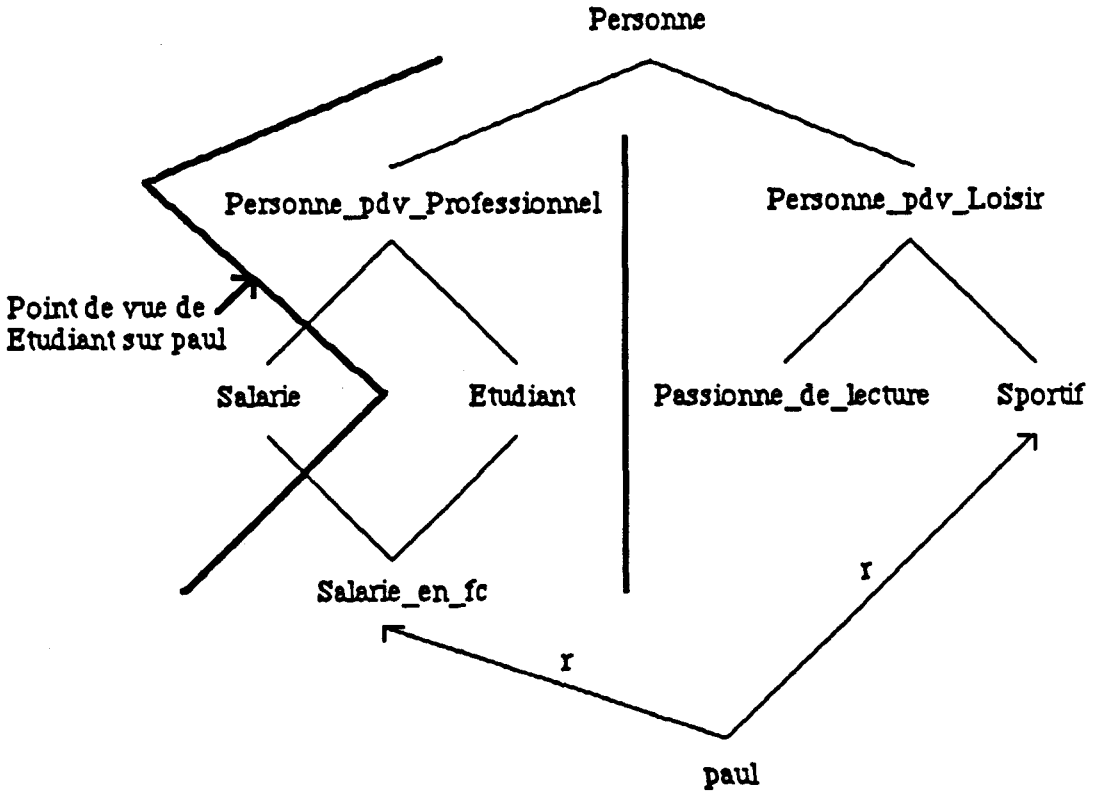
la forme de numero_carte est en effet imposée par Salarie_en_fc, qui définit la méthode valide_carte la plus affinée sous ce point de vue sur paul

> [paul '(numero_carte: as Sportif) "S13"]
= S13

Expliquons le premier message. Le point de vue Etudiant sur paul est (les "..." représentent la suite de la partie figée du treillis de représentation des instances de Personne, c'est à dire ses surclasses R-OBJECT L-Object OBJECT):

Dep (Etudiant) = (Salarie_en_fc Etudiant Personne_pdv_Professionel Personne ...)
Rep (paul) = (Salarie_en_fc Etudiant Personne_pdv_Professionel Sportif
Personne_pdv_Loisir Personne ...)

pdv (Etudiant , paul) = Dep (Etudiant) ∩ Rep (paul)
= (Salarie_en_fc Etudiant Personne_pdv_Professionnel
Personne ...)



la définition du sélecteur *numero_carte:* sous ce point de vue est trouvée dans la classe Etudiant:

Etudiant
...
sélecteurs:
(... numero_carte: ncarte=)

par as implicite la définition de méthode associée est "(ncarte= as Etudiant)" , d'où, d'après la transformation M-R, la méthode appliquée (applymeth) en réaction au message par paul est "ncarte=" sous le point de vue appelant "Etudiant".

message: [paul '(numero_carte: as Etudiant) "FC13"]
M-R
=> {(ncarte= as Etudiant) "FC13"}

la méthode trouvée par lookup sous le point de vue Etudiant est définie par celle-ci, d'où l'application:

Etudiant / Etudiant: ncarte= "FC13"

Le point de vue d'exécution est identique au point de vue appelant (MIP (Etudiant , Etudiant) = Etudiant). Cette méthode appelle "(valide_carte as Etudiant)" par as implicite, dont le point de vue appelant associée est identique au point de vue d'exécution (appel de MIP avec les mêmes arguments que précédemment). Sous ce point de vue la méthode valide_carte est trouvée dans Salarie_en_fc (relâchement):

Etudiant / Salarie_en_fc: valide_carte "FC13"

qui impose la contrainte "numero_carte de la forme FCnombre", vérifiée sur l'argument "FC13".On poursuit donc l'exécution de Etudiant / Etudiant: ncarte= "FC13", avec l'affectation de l'attribut "(numero_carte as Etudiant)" par as implicite. Le point de vue d'appel de cette référence reste identique au point de vue d'exécution (appel identique de MIP). Sous ce point de vue, c'est bien l'attribut "numero_carte" de Etudiant qui est considéré:

Etudiant / (<- Etudiant:numero_carte "FC13")

Ce qui est l'effet souhaité. Le comportement induit par l'autre message, spécifiant le point de vue de Sportif, serait justifié de façon symétrique.

Le fait important à retenir est que *le mécanisme est identique sur tout objet en considérant la représentation multiple en général, pour les r-objets, unique en particulier, pour les objets "classiques"*: l'application repd, à la base des définitions de treillis de représentation d'un objet et (de façon induite) de point de vue d'une classe sur celui-ci, étant multivaluée dans le premier cas, monovaluée dans le second.

L'intérêt tout particulier du mécanisme de sélection de points de vue pour la représentation multiple réside dans le fait qu'il permet, comme nous l'avons vu sur l'exemple précédent, de considérer une des classifications exclusives de l'objet. Le mécanisme d'inférence des points de vue assure ensuite que l'activation de l'objet se déroulera exlusivement sous cette classification, indépendamment des autres.

La sémantique de la notion de point de vue induit une facilité d'expression de la sélection en permettant une certaine imprécision sur la classe spécifiée. Ce fait est lié à la propriété de conservation de l'affinement que nous avons montrée au paragraphe IV.2.5.6.2, par laquelle, pour une caractéristique référencée sous un point de vue (appelant), le point de vue considéré effectivement est toujours le plus affiné, i-e celui qui la définit de façon la plus affinée. Cette propriété est importante pour la réalisation de la notion de point de vue, étant donné son intérêt pour sélectionner explicitement des classifications indépendantes (principe d'indépendance), laissant implicite la prise en compte de l'affinement à l'intérieur de celles-ci.

Nous pouvons constater ceci sur l'exemple précédent pour la méthode "valide_carte" appelée sous le point de Etudiant et considérée effectivement sous le point de plus affiné de Salarie_en_fc. Ainsi, bien que le point de vue le plus affiné soit celui de cette dernière classe, le mécanisme de sélection permet de spécifier un point de vue moins précis (Etudiant) mais suffisant pour exprimer simplement qu'il s'agit de celui-ci par opposition au point de vue indépendant de Sportif. De façon encore plus générale ici la spécification

du point de vue de Personne_pdv_Professionnel par opposition à celui de Personne_pdv_Loisir aurait suffi:

```
> [paul '(numero_carte: as Personne_pdv_Professionnel) "FC15"]
= FC15
> [paul '(numero_carte: as Personne_pdv_Loisir) "S15"]
= S15
```

ce qui correspond simplement à considérer paul respectivement des points de vue professionnel et loisir, deux classifications indépendantes des instances de Personne. Montrons en effet que le premier message est équivalent à la précédente activation de paul sous le point de vue plus précis de Etudiant ([paul '(numero_carte: as Etudiant) "FC13"]):

le point de vue de Personne_pdv_Professionnel sur paul est:

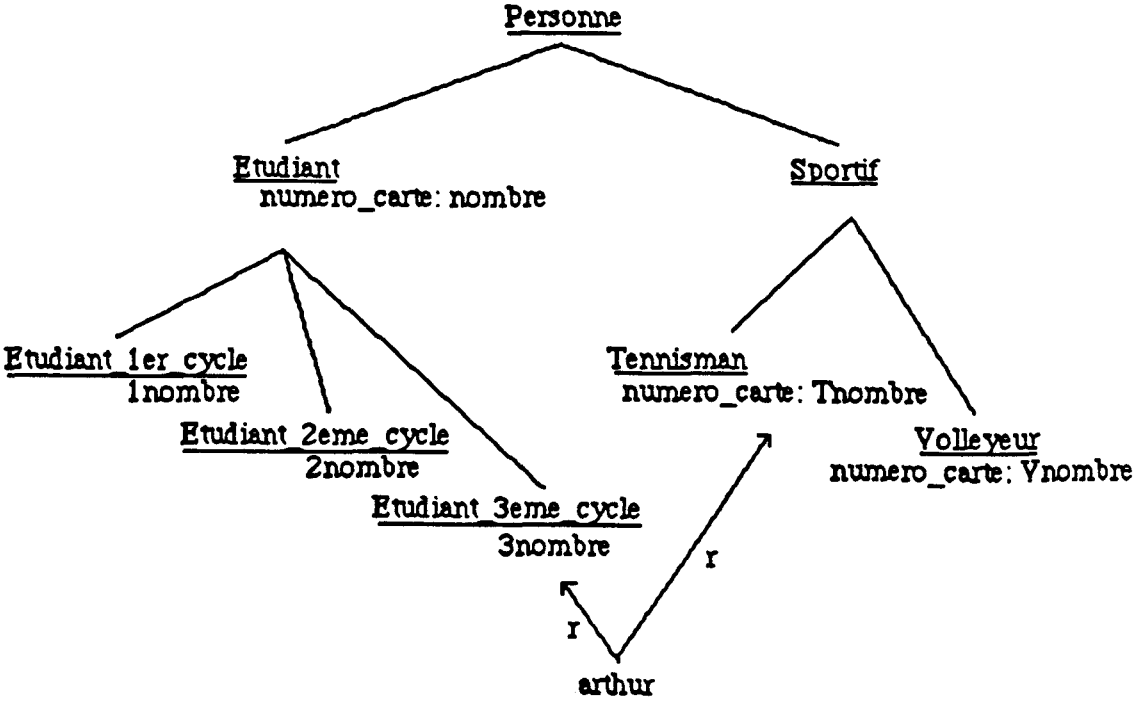
(Salarie_en_fc Salarie Etudiant Personne_pdv_Professionnel Personne ...)

sous lequel le lookup trouve la définition du sélecteur dans la classe Etudiant (relâchement), ce qui provoque par la transformation M-R de façon analogue à l'exemple précédent:

```
message: [paul '(numero_carte: as Personne_pdv_Professionnel) "FC15"]
M-R
 =>  {(ncarte= as Etudiant) "FC15"}
```

c'est à dire l'application de la même méthode et donc la même réaction.

Cette facilité peut être montrée de façon encore plus nette sur l'exemple simple suivant. Tout étudiant a un attribut numero_carte dont le format est défini par chaque sousclasse d'appartenance possible; ceci de façon (volontairement) similaire pour une classification distincte (et partielle) des instances de Personne, du point de vue loisirs, en ne représentant que les sportifs. Arthur est étudiant de troisième cycle et tennisman:



soit en Rome:

```
[I-CLASS 'new 'Personne 'surcl '(L-Object R-OBJECT)]

[R-CLASS 'new 'Etudiant
 'surcl '(Personne)
 'attributs '(numero_carte)
 'methodes
  '(ncarte= (lambda (n) {valide_carte n} {<- 'numero_carte n})
    valide_carte (lambda (n)
                   (if (ndigitp n) t {erreur "numero_carte doit être un nombre"})))
  'selecteurs '(horaires: h= numero_carte: ncarte=)]

[R-CLASS 'new 'Etudiant_1er_cycle
 'surcl '(Etudiant)
 'methodes
  '(valide_carte (lambda (n)
                   (if (and (equal (sref x 1) "1") (ndigitp (substring x 2 (1- (slen x)))))
                       t
                       {erreur "numero_carte de la forme 1nombre"}))))]

[R-CLASS 'new 'Etudiant_2eme_cycle
 'surcl '(Etudiant)
 'methodes
  '(valide_carte (lambda (n)
                   (if (and (equal (sref x 1) "2") (ndigitp (substring x 2 (1- (slen x)))))
                       t
                       {erreur "numero_carte de la forme 2nombre"}))))]

[R-CLASS 'new 'Etudiant_3eme_cycle
 'surcl '(Etudiant)
 'methodes
  '(valide_carte (lambda (n)
                   (if (and (equal (sref x 1) "3") (ndigitp (substring x 2 (1- (slen x)))))
                       t
                       {erreur "numero_carte de la forme 3nombre"}))))]

;de façon similaire la classification sous Sportif

[R-CLASS 'new 'Sportif 'surcl '(Personne)]

[R-CLASS 'new 'Tennisman
 'surcl '(Sportif)
 'attributs '(numero_carte)
 'methodes
  '(ncarte= (lambda (n) {valide_carte n} {<- 'numero_carte n})
    valide_carte (lambda (n)
                   (if (and (equal (sref x 1) "T") (ndigitp (substring x 2 (1- (slen x)))))
                       t
                       {erreur "numero_carte de la forme Tnombre"})))
  'selecteurs '(numero_carte: ncarte=)]
```

```
[R-CLASS 'new 'Volleyeur
 'surcl '(Sportif)
 'attributs '(numero_carte)
 'methodes
  '(ncarte= (lambda (n) (valide_carte n) (<- 'numero_carte n))
    valide_carte (lambda (n)
                   (if (and (equal (sref x 1) "V") (ndigitp (substring x 2 (1- (slen x)))))
                       t
                       (erreur "numero_carte de la forme Vnombre"))))
  'selecteurs '(numero_carte: ncarte=)]
```

La définition de la notion de point de vue nous permet de dialoguer avec arthur sous les points de vue indépendants et relativement imprécis de Etudiant et Sportif, sans se soucier explicitement de sa représentation plus affinée sous ces deux points de vue, en tant qu'étudiant de 3ème cycle et joueur de tennis.

```
> [arthur '(numero_carte: as Etudiant) "315"]
= 315
> [arthur '(numero_carte: as Sportif) "T15"]
= T15
```

Sa caractérisation la plus affinée est prise en compte implicitement par le mécanisme d'inférence des points de vue, notamment ici pour ses méthodes "valide_carte" appelées des méthodes "ncarte=". Pour le premier message, succintement, la méthode associée est celle de Etudiant, son point de vue d'exécution est Etudiant sous lequel le point de vue d'appel de valide_carte est identique (par as implicite). Sous ce point de vue appelant, qui est égal à (Etudiant_3eme_cycle Etudiant Personne ...), la méthode "valide_carte" est trouvée dans Etudiant_3eme_cycle, d'où le comportement annoncé. Ceci n'empêche évidemment pas d'être plus précis dans la sélection, comme dans les vérifications suivantes:

```
> [arthur '? '(numero_carte as Etudiant_3eme_cycle)]
= 315
> [arthur '? '(numero_carte as Tennisman)]
= T15
```

Comme nous l'avons déjà précisé nous pensons que cette propriété des points de vue de Rome, due à celle de conservation de l'affinement, est assez primordiale pour ce concept. Elle permet une relative facilité d'expression, induisant une amplification de la déclarativité de la représentation, du fait de la prise en charge implicite de certains problèmes, tels l'affinement. Cette facilité est cependant limitée par l'existence possible d'ambigüités sous un point de vue, auquel cas le point de vue spécifié doit être plus précis pour pallier ces problèmes. C'est le cas sur notre exemple si l'objet arthur sous le point de vue sportif, est représentant à la fois de Tennisman et Volleyeur (un bon sportif ne fait rarement qu'un seul sport!):

```
> [arthur 'r+ Volleyeur]
= arthur
```

Concernant les caractéristiques associées à numero_carte, la spécification du point de vue Sportif, toujours par opposition au point de vue indépendant Etudiant, ne suffit plus car sous ce premier point de vue, il reste une ambigüité. En effet celui-ci englobe les classes Tennisman et Volleyeur (et éventuellement d'autres) qui définissent toutes deux indépendamment un attribut numero_carte:

Dep(Sportif) = (Tennisman Volleyeur Sportif Personne ...)
Rep(arthur) = (Etudiant_3eme_cycle Etudiant Volleyeur Tennisman Sportif Personne ...)
pdv(Sportif , arthur) = Dep(Sportif) ∩ Rep(arthur)
= (Volleyeur Tennisman Sportif Personne ...).

Sous le point de vue sportif il faut donc cette fois préciser un point de vue plus précis, Tennisman ou Volleyeur. Si non, l'approche actuelle de Rome est de prendre en compte le point de vue par défaut sur la caractéristique calculé par le lookup de base, i-e en profondeur d'abord en vérifiant toujours la règle du plus affiné (remarquons qu'il s'agit ici du cas où la définition de la caractéristique la plus affinée n'est pas unique). Une solution plus satisfaisante mais non implantée actuellement en Rome serait de reconnaître s'il y a ambigüité sous un point de vue et de générer une erreur dans ce cas. Observons ceci sur l'exemple:

```
> [arthur ? '(numero_carte as Sportif)]
= ??
```

ce qui correspond au numéro de carte en tant que volleyeur, qui a la valeur indéterminée (??) initialisée lors du dernier r+ ([arthur 'r+ Volleyeur]):

```
> [arthur '? (numero_carte as Volleyeur)]
= ??
> ;et toujours
> [arthur '? '(numero_carte as Tennisman)]
= T15
```

En effet sous le point de vue Sportif calculé précédemment (qui est ordonné selon le parcours du lookup)*, le point de vue trouvé par défaut pour l'attribut numero_carte est celui de Volleyeur. La réaction au premier message s'effectue donc sous ce point de vue. De façon semblable, un message similaire au premier message à arthur sous le point de vue Sportif ([arthur '(numero_carte: as Sportif) "T15"]), espérant également invoquer implicitement le point de vue Tennisman, n'est plus possible ici:

```
> [arthur '(numero_carte: as Sportif) "T16"]
... ERREUR de la forme Vnombre ...
;contrairement à:
> [arthur '(numero_carte: as Sportif) "V15"]*
= V15
```

---

* Remarquons que Volleyeur précéde Tennisman. C'est en effet l'ordre courant de l'attribut rclasses de l'objet qui détermine le parcours du lookup. A chaque r+ la nouvelle classe de représentation directe vient en tête de liste, ici le dernier r+ portait sur Volleyeur. Ceci est arbitraire mais voit cependant une petite utilité dans le fait que si le point de vue par défaut doit être considéré dans tout message succédant à [O 'r+ C], c'est justement cette dernière classe qui le détermine (selon la chronologie de la représentation).

Il est évidemment conseillé, dans la mesure du possible, de toujours spécifier un point de vue non ambigü, la gestion du message d'erreur pour ambigüité évoquée précédemment serait un élément de l'aide possible à apporter. Les points de vue non ambigüs ici pour les caractéristiques concernant "numero_carte" sont Etudiant par opposition à Tennisman et Volleyeur sous celui de Sportif:

```
> [arthur '? (numero_carte as Etudiant)]
= 315
> [arthur '? (numero_carte as Tennisman)]
= T15
> [arthur '? (numero_carte as Volleyeur)]
= V15
```

IV.3.3 Comparaison à d'autres approches

Nous allons essayer de comparer la Représentation multiple et évolutive de Rome à d'autres approches possibles [Carré & Comyn 88]. Le premier point examiné critique la réalisation de cette notion par la remise en cause de la contrainte de Représentation Unique Figée du modèle classique des langages orientés objet. Puis nous aborderons succintement le point de vue du modèle prototype/délégation sur ce problème. Enfin nous rappellerons, suite à la comparaison héritage/agrégation de la section III.5, que l'agrégation classique peut dans une certaine mesure se substituer à l'héritage, et apporter de fait une conception différente de la R.M.E. [Carré & Comyn 87c].

IV.3.3.1 C.R.U.F vs R.M.E

Nous avons vu que la remise en cause de la contrainte C.R.U.F est à la base du développement de la R.M.E en Rome. Dans un premier temps, le remplacement de la représentation unique par la représentation multiple d'objet permet de supprimer les classes produits (IV.3.1.2) nécessaires à la représentation unique par l'instanciation.

Une autre implantation possible, sans sortir du modèle classique, serait de conserver ces classes internes au système, car indispensables au mécanisme d'instanciation, mais cachées de l'utilisateur. Celles-ci seraient gérées automatiquement en proposant à l'utilisateur des primitives d'accés, de type associatif, telles que "instancier telle classe produit (sousclasse) de telles classes, si elle existe sinon la créer". C'est le sens de la solution apportée par Oaklisp sous la forme de son "mixin manager" pour gérer la complexité des mixins quand le treillis des classes croît. Cette approche permet la représentation multiple en passant classiquement par l'héritage multiple au dessus de la classe d'instanciation gérée automatiquement. Nous pouvons parler ici de représentation multiple "virtuelle" dans le sens où cette classe produit est cachée de l'utilisateur, donnant l'illusion que l'objet est représentant direct de ses surclasses, seules connues de l'utilisateur. Cette subtilité relève uniquement de l'implantation et permet de réaliser la représentation multiple par une technique différente de celle adoptée par Rome. Elle ne résout cependant pas le problème de la représentation évolutive.

Une solution à la représentation évolutive dans la continuité de l'approche précédente, i-e sans remettre en cause techniquement la contrainte C.R.U.F, pourrait passer par un mécanisme de *copie/destruction* de l'objet. Ce mécanisme pourrait se dérouler en quatre phases. Soit un objet O instance d'une classe C1 et à faire évoluer comme instance de la classe C2. Insistons sur le fait que C1 (respectivement C2) est (respectivement sera) l'unique classe de représentation directe de l'objet, éventuellement calculée automatiquement par un mécanisme de représentation multiple virtuelle tel que le précédent. Le cheminement serait alors:

- instanciation de C2, soit un nouvel objet que l'on appellera par abus de langage la nouvelle version de O
- copie des caractéristiques de O restant valides pour sa nouvelle version
- substitution de toutes les références à O dans l'environnement par des références à la nouvelle version
- destruction de O.

La solution consiste donc ici également en une évolution "virtuelle" de l'objet dans le sens où, contrairement à Rome, la nouvelle version de l'objet est en fait une autre instance: on fait "renaître" l'objet comme instance d'une nouvelle classe [Comyn 87], cette réinstanciation pouvant être cachée de l'utilisateur à travers des primitives spécifiques (du type des techniques classiques de coercion), subtilités relevant de l'implantation.

En conclusion ce point de comparaison relève donc plus de problèmes d'implantation que sémantiques.

IV.3.3.2 Prototype/délégation

Une courte intrusion dans le monde du paradigme prototype/délégation (II.4) qui sort de notre modèle de L.O.O étudié, pour préciser tout d'abord que la représentation multiple est implicitement possible si l'objet peut être l'extension de plusieurs prototypes. Ce parallèle entre représentation multiple sur un treillis de classes et multiplicité du lien de prototypage entre objets doit être modulé par les nuances entre ces deux modèles exposées au paragraphe II.4.

Concernant la représentation évolutive, nous pouvons constater que le mécanisme décrit au paragraphe précédent trouve sa réalisation implicite ici:

- la nouvelle version de l'objet serait créée ici comme extension de l'ancienne qui deviendrait dès lors son prototype
- la copie des caractéristiques toujours valides du prototype dans l'extension est réalisée implicitement par le mécanisme de la délégation.

Sur ce dernier point, la délégation réalise en fait une copie virtuelle de la caractérisation par accès dynamique (et par nécessité) à l'ancienne version. C'est d'ailleurs le sens d'une approche adoptée pour la gestion des versions d'objets en LOOPS exposée dans [Mittal & al. 86]: "Virtual copies, at the boundary between classes and instances" ou plutôt entre classes et prototypes (!). Ici, il est possible de créer une version d'un objet par extension d'une autre version à laquelle il fait référence par un lien de protoypage. La nouvelle version a accès aux caractéristiques de cette ancienne par un mécanisme de copie virtuelle fondé sur un accès dynamique à travers le lien de prototypage, proche de la délégation. La nouvelle version partage ainsi implicitement les caractéristiques qu'elle souhaite disposer de l'ancienne.

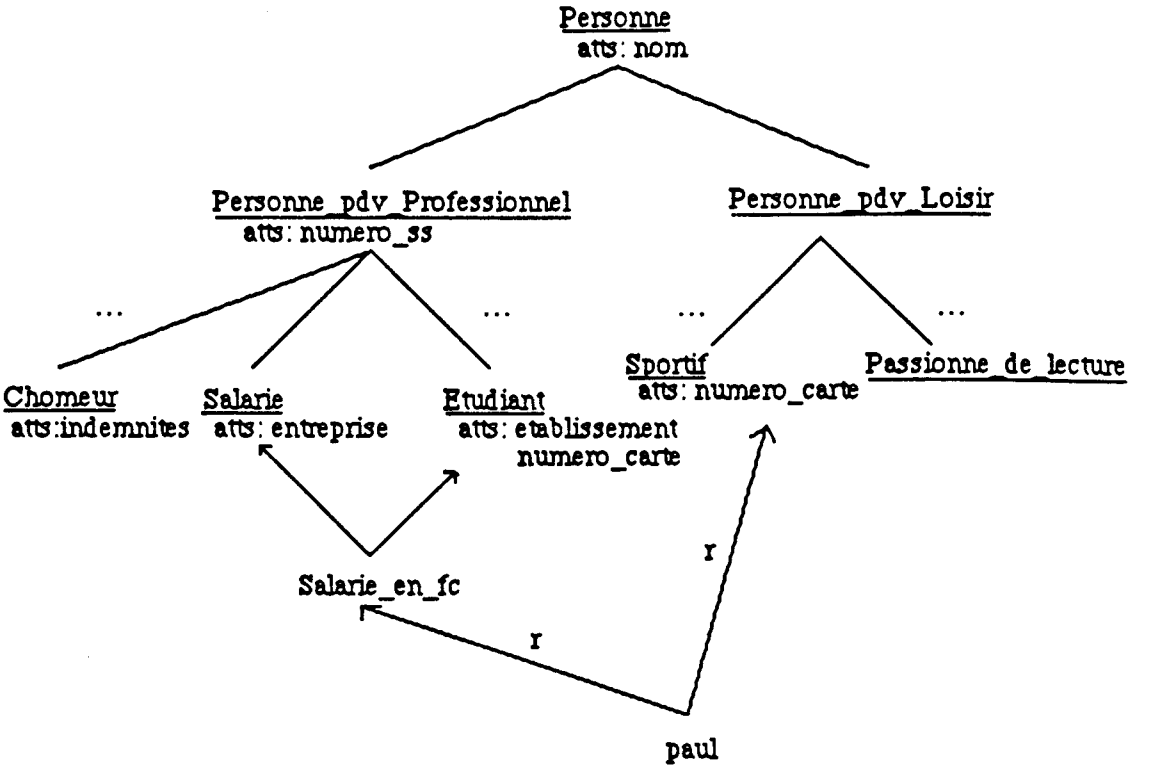
IV.3.3.3 Héritage vs agrégation

Nous avons vu dans la section III.5 que l'agrégation pouvait se substituer à l'héritage, notamment à l'héritage multiple. La transformation consiste à remplacer la *hiérarchie conceptuelle de classes* par une *hiérarchie structurelle entre instances* moyennant un effort de programmation pour gérer les agrégats, notamment la répartition (la distribution) du comportement du tout sur ses parties, ainsi que sa cohérence. Nous avons vu que pour ces raisons de programmation explicite plus importante et aussi parce qu'elle met à plat la hiérarchie conceptuelle initiale, l'approche orientée agrégation est moins déclarative et donc moins satisfaisante pour la représentation des connaissances que l'approche de type héritage.

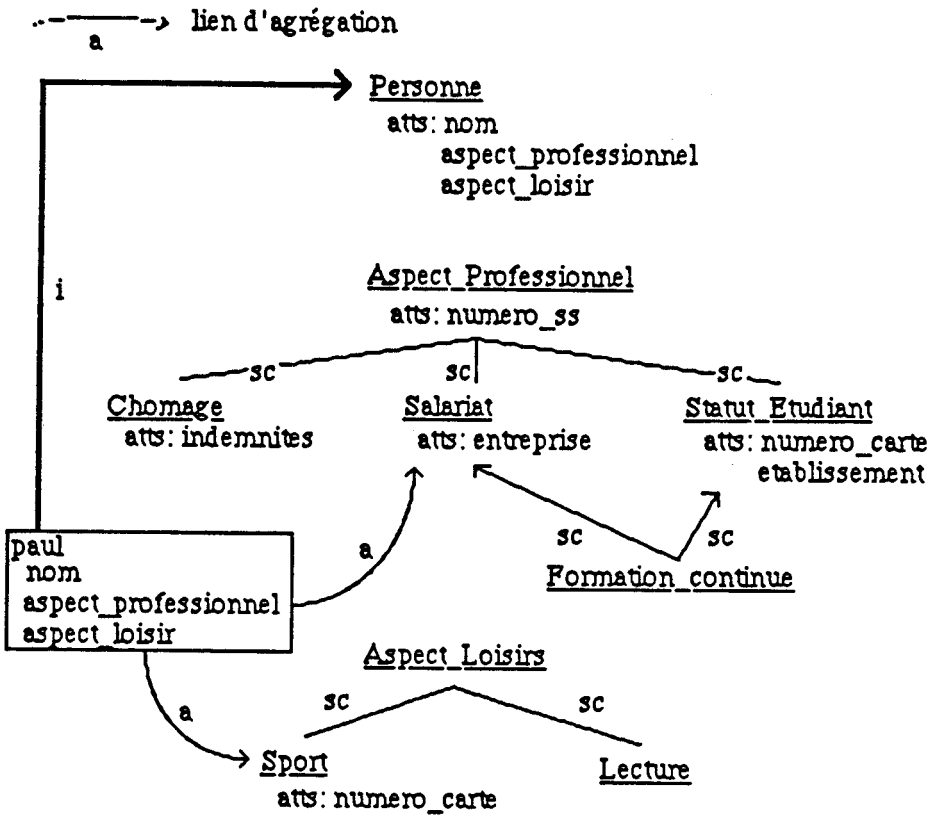
Du point de vue de la représentation multiple et évolutive ici, une approche semblable peut être adoptée avec les mêmes inconvénients, mais en conservant les concepts de base des L.O.O.[Carré & Comyn 87c]. Le traitement consiste à remplacer les liens de représentation multiple vers les classes de représentation par des liens d'agrégation (IS-A par HAS-PART) vers des objets représentant les différents aspects, à travers des attributs prévus à cet effet dans la classe d'instanciation. La hiérarchie conceptuelle issue de la classification est alors réduite à une unique classe d'instanciation, qui correspond à notre classe minimum du treillis de représentation potentielle. Cette classe est cependant de définition beaucoup plus complexe car elle doit prévoir tous les attributs correspondants aux différents aspects et la gestion de ceux-ci.

ex:

Soit la classification multiple suivante où nous ne montrons que quelques attributs définis par les classes. Une instance de Personne, paul, est représentant de Salarie_en_fc et Sportif:



Une solution de type agrégation de ce problème consiste alors à définir dans Personne les attributs "aspect_professionnel aspect_loisir" qui référenceront par agrégation des objets représentants des classes respectives Statut_Professionnel Activite_de_loisir, soit schématiquement:



Soit en Smalltalk, par exemple:

```
Object subclass: #Personne
        instanceVariableNames: 'nom aspectProfessionnel aspectLoisir '
!Personne methodsFor: 'acces aux aspects'!
aspectLoisir: aspect
   ^aspectLoisir at: (aspect name)
aspectProfessionnel
   ^aspectProfessionnel
!Personne methodsFor: 'modif aspects'!
aspectLoisirs: aspect
 (aspect inheritsFrom: AspectLoisir)
  ifTrue: [aspectLoisir at: (aspect name) put: (aspect new)]
  ifFalse: [self error: 'aspect loisir requis']
aspectProfessionnel: aspect
 (aspect inheritsFrom: AspectProfessionnel)
  ifTrue: [aspectProfessionnel <- (aspect new)]
  ifFalse: [self error: 'aspect professionnel requis']
!Personne methodsFor: 'atCreation'!
construct
 aspectLoisir <- Dictionary new

Personne class
        instanceVariableNames: ''
!Personne class methodsFor: 'creation'!
new
 "associer a chaque instance un dictionnaire de loisirs"
  ^super new construct
```

```
Object subclass: #AspectProfessionnel
        instanceVariableNames: 'numeroSs '
!AspectProfessionnel methodsFor: 'acces'!
numeroSs
  ^ numeroSs
numeroSs: n
 numeroSs _ n

AspectProfessionnel subclass: #Chomage
        instanceVariableNames: 'indemnites '
!Chomage methodsFor: 'acces'
indemnites
  ^ indemnites
indemnites: n
 indemnites  <- n

AspectProfessionnel subclass: #Salariat
        instanceVariableNames: 'entreprise '
!Salariat methodsFor: 'acces'!
entreprise
  ^ entreprise
entreprise: e
 (e isKindOf: Entreprise)
  ifTrue: [entreprise <- e]

AspectProfessionnel subclass: #StatutEtudiant
        instanceVariableNames: 'numeroCarte etablissement '
!StatutEtudiant methodsFor: 'acces'!
numeroCarte
  ^ numeroCarte
numeroCarte: n
 ((n at: 1) = $E)
  ifTrue: [numeroCarte <- n]

Class named: #FormationContinue
        superclasses: 'Salariat StatutEtudiant '
        instanceVariableNames: ''

Object subclass: #AspectLoisir
        instanceVariableNames: ''

AspectLoisir subclass: #Sport
        instanceVariableNames: 'numeroCarte '
!Sport methodsFor: 'acces'!
numeroCarte
  ^ numeroCarteObject subclass: #AspectLoisir
        instanceVariableNames: ''
numeroCarte: n
 ((n at: 1) = $S)
  ifTrue: [numeroCarte _ n]

AspectLoisir subclass: #Lecture
        instanceVariableNames: ''
```

Représentation Multiple:

La représentation multiple trouve son pendant ici dans la multiplicité des attributs d'aspects définis par la classe d'instanciation. Si sous un aspect l'objet peut avoir plusieurs sous aspects, il est possible de les rassembler dans une structure du genre Collection. C'est le cas par exemple ici pour l'aspect loisirs que nous avons choisi de représenter par un dictionnaire Smalltalk (semblable à une A-liste lisp), créé à l'instanciation de Personne par redéfinition du new et qui associe à chaque loisir un nom qui est celui de sa classe (sport lecture ...).

L'ajout des aspects s'effectue par des méthodes de gestion des attributs correspondants qui doivent être définies par la classe d'instanciation. Ici Personne définit les méthodes aspectProfessionnel: c et aspectLoisirs: c où c est la classe d'instanciation de l'aspect, qui mettent à jour les attributs respectifs, en instanciant la classe c:

paul aspectProfessionnel: Salariat

paul aspectLoisirs: Sport

l'état résultant de paul est représenté sur la figure précédente.

Représentation Evolutive

L'évolution d'objet passe ici par l'évolution classique de l'état de ses attributs, en particulier ici de ses attributs d'aspect, qui doit être gérée par des méthodes associées. Ici ce sont les méthodes aspectProfessionnel: et aspectLoisirs: de Personne qui ont la charge de gérer l'évolution de ses instances.

L'exemple précédent montre l'évolution de paul à partir de son statut de simple instance de personne par ajout d'aspects professionnel et loisirs. La mise à jour, qui pouvait paraître simple jusque là, se complique ici si l'on veut affiner l'un des aspects. Supposons par exemple que son aspect professionnel doive être affiné (r+) de Salariat à FormationContinue (ou encore son aspect loisir Sport doive s'affiner en Tennis, que nous n'avons pas représenté ...), l'ancien état de son attribut aspectProfessionnel ne peut être simplement remplacé par affectation par une instance de FormationContinue au risque de perdre les caractérisitques qu'il conserve en tant que salarié. Il en est de même inversement si son aspect professionnel doit évoluer (r-) de FormationContinue à Salariat, ses caractérisitques en tant que salarié doivent être conservées. Les méthodes de gestion des attributs d'aspect sont donc loin d'être aussi simples que les versions rudimentaires que nous avons annoncées dans le programme Smalltalk.

C'est en fait ici tout le problème de la représentation multiple et évolutive qui se pose à nouveau cette fois pour les objets représentant les aspects, i-e sur les attributs, et, de façon générale, récursivement sur toute la hiérarchie d'agrégation.

Un autre problème d'évolution se pose ici et concerne l'affinement incrémental de la classification elle-même par simple ajout de sousclasses de représentation. Cette possibilité facilitée par l'héritage (III.3.5.3 Spécialisation) et reprise en tant que telle dans le contexte de la représentation multiple et évolutive (IV.3.2.2) est plus difficilement utilisable ici. Etudions les deux cas typiques d'affinement possibles.

Le premier cas concerne l'affinement de la classification courante sous un point de vue particulier. Il s'agit ici d'ajouter des sousclasses de représentation sous ce point de vue par affinement de celles existant déjà. Par exemple supposons que l'on veuille affiner, sous le point de vue loisirs, la classe Sportif en Tennisman Jogger ... et donc d'affiner les sportifs existant dans l'environnement, tels que paul, selon les sports qu'ils pratiquent. Ce problème serait résolu en Rome par l'ajout dans un premier temps de ces

sousclasses de représentation sous Sportif, selon un certain processus de conception incrémentale classique, puis par exemple par le message de R.M.E:

[paul 'r+ Jogger]

La solution du problème dans une approche de type agrégation pose en fait les mêmes problèmes que précédemment. Même si l'on peut utiliser la propriété d'affinement incrémental due à l'héritage pour la définition des sousclasses correspondantes ici, soient Tennis Jogging ... comme sousclasses de Sport, l'affinement des objets eux-mêmes, tels paul, réclame comme précédemment une solution à l'affinement de ses aspects, soit l'affinement de son sous-objet Sport en sous-objet Tennis.

Le second cas concerne l'affinement de la classification par ajout d'un nouveau point de vue. La solution en Rome serait alors semblable à celle présentée pour le problème précédent. Pour l'approche de type agrégation, il s'agit ici de compléter la hiérarchie structurelle par ajout de nouveaux aspects et donc de modifier la définition des classes elles-mêmes et par conséquent la structure de leurs instances.

Supposons par exemple que l'on veuille ajouter l'aspect social aux instances de Personne, il faut dès lors:

- dans un premier temps modifier la classe Personne pour ajouter l'attribut correspondant "aspectSocial"
- puis modifier la structure de toutes les instances existant pour leur ajouter ce nouvel aspect.

Le second point est généralement tout un problème en soi si une modification de la structure de la classe ne se répercute pas automatiquement sur ses instances, ce qui est le cas pour la plupart des langages (ce n'est cependant pas le cas en Smalltalk 80). Un mécanisme additionnel est alors nécessaire (telles que les méthodes très particulières d'ajout dynamique de variables d'instance de LOOPS).

Le problème est en fait que tout affinement d'agrégats réclame des modifications en profondeur alors que l'héritage permet l'affinement incrémental par la spécialisation des classes en sousclasses, sans modifier la défintion des premières. Cet affinement incrémental de la hiérarchie conceptuelle est rendu utilisable pour les instances elles-mêmes grâce à la R.M.E. L'approche orientée agrégation conduit donc à des solutions à priori moins évolutives à moins de développer des outils additionnels équivalents, tels que l'ajout ou le retrait dynamique d'aspects comme sur les perspectives de LOOPS que nous verrons plus bas (et que nous avons déjà introduites au paragraphe III.5.2).

Héritage et points de vue

Le dialogue avec l'objet sous un point de vue donné correspond ici au cheminement à travers l'agrégat d'objets grâce aux méthodes aspectProfessionnel et aspectLoisirs: nomLoisir qui retourne respectivement son sous-objet référencé par aspectProfessionnel et celui associé à nomLoisir dans le dictionnaire de ses loisirs (selon un schéma de traitement de l'agrégation exposé au II.4.3.1):

en supposant que paul ait un aspectProfessionnel de type FormationContinue,

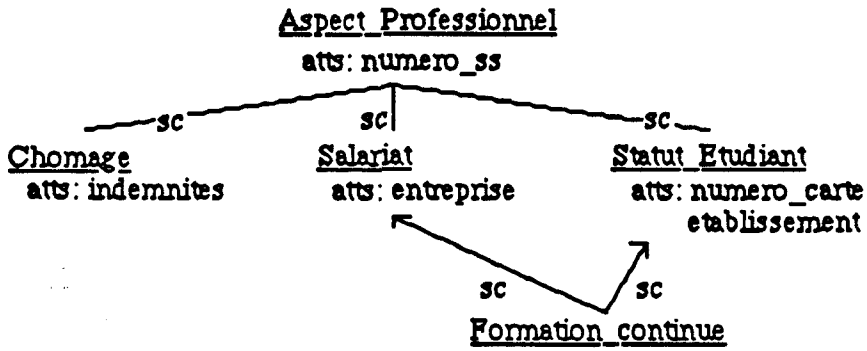
Paul aspectProfessionnel numeroCarte: 'E12'

Ibeme <- Entreprise new
Paul aspectProfessionnel entreprise: Ibeme

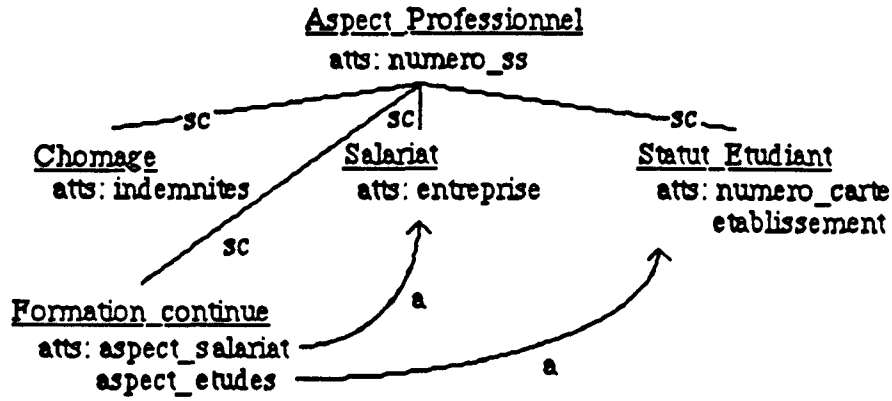
(Paul aspectLoisirs: Sport) numeroCarte: 'S12'

Comme nous l'avons déjà précisé au paragraphe III.5.2 l'héritage implicite des caractéristiques est donc remplacé ici par leur distribution sur les sous-objets représentant les aspects. *Un avantage de l'agrégation ici est sa gestion implicite, grâce à l'encapsulation, des conflits de nom de caractéristiques* puisque des aspects indépendants sont représentés par des objets distincts, par exemple l'objet Sport et l'objet FormationContinue disposent tout deux des caractéristiques de même nom associées à leur attribut numeroCarte. Le problème des conflits peut cependant se poser de façon classique s'il y a héritage multiple entre sousclasses d'une même classe d'aspect, par exemple Salariat et Etudes pour FormationContinue sous l'aspect professionnel.

Le problème se pose donc une fois de plus de façon récursive sur les sous-aspects, il est possible notamment d'utiliser l'agrégation de façon uniforme sur les sous-aspects également. Sur notre petit problème il s'agirait de remplacer la hiérarchie d'héritage suivante:



par la hiérarchie d'agrégation:



L'aspect Formation_continue définit lui-même un agrégat de sous-aspects, Salariat et Statut_Etudiant par ses attributs respectifs aspect_Salariat et aspect_etudes qui seraient gérés de façon maintenant classique par des méthodes de mise à jour, aspect_salariat: c et aspect_etudes: c. Insistons sur le fait que la classe c définissant le sous-aspect n'est pas forcément Salariat et Statut_Etudiant respectivement, si l'on désire conserver la possibilité de classification plus fine telle que Fonctionnariat... sous Salariat et Etudes1erCycle ... sous Statut_Etudiant, ce qui poserait à nouveau le dilemne agrégation/héritage récursivement. En nous limitant à la solution précédente, l'accès aux sous-aspects est alors récursive à travers la hiérarchie d'agrégation. Le message précédent:

paul aspectProfessionnel numeroCarte: 'E12'

deviendrait ici:

paul aspectProfessionnel aspectEtudiant numeroCarte: 'E12'

posant tout le problème du traitement de l'accès aux sous-objets d'un agrégat que nous avons étudié au paragraphe II.4.3.1. Ce type de solution par sous-aspects (et récursivement) permet donc de résoudre implicitement les conflits éventuels d'héritage qui pourraient intervenir entre les surclasses de FormationContinue dans la solution initiale.

De plus elle a l'avantage de permettre de sélectionner un sous-aspect plus fin (donc plus interne ici, au sens de la hiérarchie des parties), possibilité qui est requise par la notion de point de vue comme nous l'avons précisé au IV.3.2.3.2. C'est le cas pour le message précédent qui fait apparaître le sous-aspect étudiant de l'aspect professionnel, possibilité absente en tant que telle de la solution intiale, puisque le sous-aspect (le sous-objet) étudiant n'existe pas, seul l'aspect instance de FormationContinue a une existence propre dans cette solution. Remarquons cependant que nous aurions pu le faire apparaître en utilisant cette fois l'héritage par sélecteur composé, cette solution étant en fait un compromis entre héritage et agrégation:

paul aspectProfessionnel StatutEtudiant.numeroCarte: 'E12'

Reprenons notre dernier type de solution, selon une approche purement orientée agrégation, qui présente les avantages de gérer implicitement les conflits et de donner une existence propre aux sous-aspects. Elle pose cependant un problème supplémentaire d'agrégation que nous avons étudié dans les sections III.4.3 et II.5.2, qui est la gestion de la cohérence globale des sous-objets, ici des sous-aspects. Ce problème se pose classiquement pour toute transformation de combinaison par héritage multiple en assemblage par agrégation. Montrons ceci sur l'exemple typique de la transformation précédente que nous complétons en ajoutant les caractéristiques concernant les horaires et cours, avec leurs contraintes associées dans chaque classe selon l'exemple traité aux paragraphes IV.2.5.5.2 et IV.3.2.3, soit à la Smalltalk:

```
Salarie
 superclasses: Personne
 instance variables: horaires
 methods:
  horaires: x
   self valideHoraires: x.
   horaires <- x.
  valideHoraires: x
   x < 40
    ifFalse: [self error: "horaires < 40"]
```

```
Etudiant
 superclasses: Personne
 instance variables: cours
 methods:
   cours: x
    self valideCours: x.
    cours <- x
   valideCours: x
    x <= 35
     ifFalse: [self error: "cours <= 35"]
```

```
Salarie_en_fc
 superclasses: Salarie Etudiant
 methods:
  valideCours: x
    (horaires isNil) or (x <= horaires)
     ifFalse: [self error: "cours <= horaires"]
  valideHoraires: x
    (cours isNil) or (x >= cours)
     ifFalse: [self error: "cours <= horaires"]
```

Comment représenter sur la solution distribuée du type agrégation la contrainte globale "cours ≤ horaires" définie ici par combinaison dans Salarie_en_fc et possible parce que son domaine de visibité couvre toutes ses surclasses? Un problème similaire a été étudié au paragraphe III.5.2 (points en coordonnées polaires et cartésiennes), où différents types de solutions ont été proposées. Nous donnerons ici la solution suivante:

une instance de FormationContinue, appelée le tout, doit gérer elle-même la contrainte globale à ses sous-objets, aspectProfessionnel et aspectEtudiant , i-e disposer elle-même des méthodes valideHoraires: et valideCours: globales car nécessitant des informations de chacun de ces deux sous-objets:

```
FormationContinue
...
methods:
  valideCours: x
    "horaires est une variable locale"
   | horaires |
   horaires <- aspectSalariat horaires.
   (horaires isNil) or (x <= horaires)
    ifFalse: [sef error: "cours <= horaires"]
  valideHoraires: x
   | cours |
   cours <- aspectEtudiant cours.
   (cours isNil) or (x >= cours)
    ifFalse: [self error: "cours <= horaires"]
  ...
```

Les sous-objets doivent informer le tout à chaque modification de leur variable d'instance respective cours et horaires. Pour cela leurs méthodes valideCours et valideHoraires respectivement doivent être redéfinies pour rediriger le contrôle sur le tout, qu'il référence

par une variable d'instance "tout" (initialisée à leur création par une méthode du genre newPour: du III.5.2)*. Ceci conduit aux sousclasses respectives (en supposant que les classes StatutEtudiant et Salariat aient été modifiés pour ajouter les caractéristiques concernant cours et horaires):

```
StatutEtudiantEnFC
  superclasses: StatutEtudiant
  instance variables: tout
  methods:
    valideCours: x
      tout valideCours: x

SalariatEnFc
  superclasses: Salariat
  instance variables: tout
  methods:
    valideHoraires: x
      tout valideHoraires: x
```

Nous constatons pour toutes ces raisons que le choix (ou l'obligation !) d'adopter une solution orientée agrégation réclame un effort de programmation beaucoup plus conséquent et complique la définition des classes.

Certaines aides peuvent être apportées pour gérer les agrégats d'aspects tels que les *perspectives* de PIE [Borning & Ingalls 82], extension de Smalltalk 76, que nous avons présentées au III.5.2. Ces outils reposent en fait sur une généralisation de l'approche précédente et ont été étudiés comme première version de l'héritage multiple (ou plutôt de sa simulation) en Smalltalk, puis reprises par LOOPS. Rappelons que PIE définit pour cela la classe *Node* d'objets pouvant disposer d'une liste de perspectives, instance de sous classes de la classe *Perspective* et dont la liste est contenue dans leur attribut *perspectives* défini par Node. La classe Perspective définit quant à elle l'attribut *node* implantant le lien inverse d'une perspective vers son noeud, i-e d'un aspect vers son tout (l'attribut node correspond en fait à notre attribut "tout"). La classe Node définit les méthodes de gestion de la liste de perspectives:

- addPersp pour ajouter une perspective
- deletePersp pour en supprimer

Le cheminement dans l'objet à travers les perspectives est assuré par une méthode générale *as:* définie par Node, qui est paramétrée par un nom de perspective, et par laquelle le noeud renvoie son objet perspective associé au paramètre, avec lequel on peut alors dialoguer (de façon semblable à notre exemple en Smalltalk).

---

* comme nous l'avons vu au III.5.2, il est possible de factoriser certaines caractéristiques communes à tout ce genre de sous-objets dans une mixin qui définirait notamment la variable d'instance tout et la méthode newPour. C'est d'ailleurs la solution adoptée dans PIE avec la mixin Perspective que nous reverrons plus bas.

Le fait que les méthodes addPersp et deletePersp permettent d'ajouter et retirer des perspectives dynamiquement rapproche plus, rétrospectivement, cette technique de la représentation multiple et évolutive que de l'héritage multiple pour lequel elle a été étudiée initialement. Ces méthodes apparaissent à priori comme des méthodes de R.M.E dans une approche orientée agrégation, telles que nos méthodes r+ et r- pour notre solution de type héritage. Elles prennent en compte notamment, du moins partiellement, l'affinement incrémental de la hiérarchie des classes de perspectives et donc de réaliser en partie ce problème pour les instances. Elles permettent, de façon assez évidente, de résoudre le second cas d'affinement incrémental énoncé plus haut, qui est l'ajout de nouveaux aspects. Cependant le problème de l'affinement sous un aspect existant reste entier.

En effet, ces méthodes restent très primitives et ne résolvent pas les problèmes récursifs d'évolution des aspects par affinement/généralisation évoqués plus haut. L'effort de programmation reste donc conséquent d'autant que tous les problèmes énoncés précédemment, tels que ce dernier et la gestion de la cohérence globale, doivent toujours être résolus explicitement.

En conclusion la R.M.E de Rome implantée par agrégation d'aspects, par extension par exemple des perspectives précédentes, pourrait faire l'objet de toute une étude en soi. Il faut cependant conserver à l'esprit l'intérêt pour la représentation des connaissances de rester proche de la nature hiérarchique de celles-ci, soit conceptuelle soit structurelle. Le choix d'une approche orientée héritage ou agrégation dépend donc avant tout de cette nature.

CHAPITRE V

*****

C O N C L U S I O N

Comme nous l'avons précisé dès la section II.1, le courant de pensée orienté objet est la révélation d'un conglomérat de problèmes étudiés dans différentes disciplines de l'informatique. Les chapitres II et III de cette thèse établissent un bilan conceptuel, méthodologique et technique de certaines notions de base du paradigme orienté objet "à la Smalltalk". Nous avons montré toute la richesse conceptuelle de l'approche bâtie sur ces fondements et donc la contribution qu'elle peut apporter à la représentation des connaissances.

Malgré les apports essentiels de ce paradigme à ce vaste problème, cette approche fait toujours l'objet de nombreuses recherches et constitue un sujet en pleine évolution. Nous espérons que ROME apporte sa pierre (!) à la fois en offrant ses solutions à certains problèmes du modèle classique introduits dans les chapitres précédents et en augmentant la richesse conceptuelle de celui-ci.

Sa contribution au modèle classique se situe dans:
- la séparation dans la définition du concept d'objet des notions de méthodes et de sélecteurs qui permet d'assurer une plus grande encapsulation, propriété essentielle de cette notion
- l'intégration dans la définition métacirculaire du modèle de la notion de classe abstraite
- une stratégie d'héritage multiple bâtie sur la notion de point de vue associée à celle de classe. Elle apporte sa solution aux problèmes de conflits qui apparaissent lors de la construction d'un treillis de classes par combinaison. Elle est caractérisée par son respect du principe d'indépendance et sa propriété de conservation de l'affinement.

Nous insistons particulièrement sur la comparaison qui a été faite à la section IV.2.5.6 entre ce dernier mécanisme et les autres stratégies graphiques. Il a été précisé que ROME tout autant que les langages bâtis sur ces dernières n'apporte toujours pas de solution universelle à tous les problèmes d'héritage. La recherche de cette solution réclame certainement une étude toujours approfondie de l'héritage et de la notion de classe tant d'un point de vue sémantique, comme dans le cadre de la théories des graphes [DUCOURNEAU.HABIB.87], du lambda calcul [CARDELLI.WEGNER.85] ou encore de l'algébre des types [REICHEL.87], que méthodologique et conceptuel. Ce dernier problème est sans doute dû à la grande diversité d'interprétation et d'utilisation du même lien classe/sousclasse notamment à la fois pour la généralisation, l'abstraction, la spécialisation, la combinaison et la réutilisation. La question posée est alors: tous ces processus peuvent-ils se traduire par un seul et même lien classe/sousclasse avec une sémantique uniforme? Si non quelles sont leur sémantique propre et comment les dissocier?

La contribution de ROME à l'enrichissement du modèle orienté objet de la représentation des connaissances se situe dans la notion de Représentation Multiple et Evolutive. Nous avons montré d'une part que ce concept facilite l'expression de hiérarchies conceptuelles de classes, notamment par points de vue multiples et indépendants sur les mêmes entités. D'autre part il permet d'exploiter dans une plus large mesure ces classifications pour les objets eux-mêmes en profitant de leur interprétation en tant que modèle de généralisation/affinement dynamique de concepts. Nous avons montré que l'intégration de cette notion est possible par la remise en cause du mécanisme d'instanciation classique comme seul moyen d'associer, de manière unique et figée, un objet à une classe.

Son développement en ROME se traduit par la définition d'une classe d'objets à représentation multiple et évolutive disposant de méthodes de manipulation des liens de représentation multiple qui les lient au treillis des classes. Nous avons analysé la solution adoptée, sa sémantique, son homogénéité par rapport aux autres notions, notamment l'héritage multiple par points de vue, et enfin comment elle se place par rapport à d'autres solutions.

Cette solution constitue un premier pas vers l'étude approfondie du concept de Représentation Multiple et Evolutive d'objet. D'un point de vue technique, la réalisation actuelle est assez rudimentaire et réclame de nombreuses extensions. La pleine utilisation du concept passe tout d'abord par le développement de ROME, simple langage orienté objet expérimental aujourd'hui, comme véritable outil de représentation des connaissances. Pour cela l'extension de ROME aux frames (FROME) est actuellement en cours d'étude et de réalisation. La notion de frame est définie en ROME à partir de celle plus primitive d'objet selon la conception par agrégation d'attributs introduite par Ferber [FERBER. 83], étudiée dans la section .III.4 et couramment utilisée dans de nombreux outils [RECHENMANN.86] [ALBERT.86] [DUCOURNEAU.QUINQUETON.86] [BENOIT.al 86] [CARRE.88]... Son application à ROME pose le problème supplémentaire du passage de la R.M.E d'objet à la R.M.E de frame.

La notion de frame nous permet d'envisager des moyens d'expression déclaratifs de la R.M.E en définissant à partir des opérations primitives (et impératives) r+ et r- des facettes d'attributs appropriées. Nous pouvons prévoir notamment la possibilité d'exprimer déclarativement des contraintes d'exclusivité entre classes de représentation multiples. Par exemple en reprenant la classification des personnes d'un point de vue professionnel, il s'agirait de pouvoir exprimer qu'une personne représentant de la classe Chomeur ne peut être à la fois représentant de la classe Salarie. D'autres facettes devraient permettre de traduire l'évolution. Par exemple nous pourrions définir pour la classe Chomeur un attribut "cause", représentant la cause de la perte d'emploi, sur l'écriture duquel des réflexes de classification automatique se déclencheraient; ceci de telle sorte que si la cause est le licenciement l'objet s'affine selon la sousclasse Chomeur_indemnise, s'il s'agit d'une démission l'objet devient représentant de Chomeur_non_indemnise.

Plus généralement il s'agirait d'exprimer la dualité qui semble naturelle entre l'état d'un objet, au sens de l'état de ses attributs, et l'état de sa représentation, au sens de ses liens d'appartenance aux classes. Nous pouvons établir de manière assez évidente une dualité entre l'attribut "age" d'une personne et son appartenance à une sousclasse d'âge. Il en est de même sur l'exemple précédent. Comment représenter cette dualité? Cette question évoque le thème de l'intégration (et non la juxtaposition !) des formalismes objet et relationnel (notamment prolog) qui est l'objectif de nombreux travaux actuellement [COMYN.87] [BRESSAN.87] [ALBERT.85] [VOYER.87] [AIT-KACI.NASR.86] [RECHENMANN.VIGNARD.85] [DIXNEUF.al.88]. La nécessité de cette intégration est bien montrée par cette phrase de Voyer:

> "Il est difficile, voire imposible, de représenter toute la connaissance d'un domaine dans un formalisme unique. En effet il existe des connaissances touchant aux principes de cheminement du raisonnement, aux stratégies de résolution, aux démarches diagnostiques de l'expert. Elles sont exprimées par des règles heuristiques, des métarègles... D'autre part, il y a des connaissances conceptuelles du style: " une voiture est un objet qui possède des roues, un moteur, un volant etc..". Celles-ci sont généralement très structurées, et bien représentées par un réseau sémantique, un réseau de schémas ..."

Il nous semble que la R.M.E de ROME constitue un pas du modèle orienté objet vers le modèle orienté règles, en ce sens qu'elle pose les bases d'un mécanisme d'inférence dynamique de la caractérisation des objets.

Ce prolème posé, nous pouvons envisager d'intégrer la notion de règle au modèle pour traduire et exprimer déclarativement l'évolution par un certain processus déductif, l'héritage étant chargé quant à lui de l'inférence inductive des caractéristiques des entités. Ceci peut se montrer sur le petit exemple intuitif suivant [CARRE.COMYN.87], soit la règle:

" si une entreprise tombe en faillite alors elle licencie tous ses salariés".

Cette règle d'évolution des entreprises et de ses salariés étant appliquée, l'héritage est alors en charge d'inférer la nouvelle caractérisation de ces entités à partir de leurs nouvelles classes de représentation (respectivement par exemple Entreprise_en_faillite et Chomeur).

Parallèlement à ces travaux d'un caractère assez fondamental, il serait intéressant d'étudier les nouvelles possibilités offertes par les concepts de ROME dans différents contextes.

La notion d'héritage multiple par points de vue, en tant qu'outil favorisant la conception indépendante de classes, devrait trouver son intérêt en génie logiciel, notamment pour la "réutilisabilité" de classes indépendantes, et en conception de bases d'entités considérées sous des points de vue différents.

Comme nous l'avons introduit dans la section IV.3.1 la notion de représentation évolutive devrait trouver son application dans tout problème qui peut être vu comme celui de l'évolution d'entités. Nous pensons notamment à son utilisation dans les chaînes de conception telles qu'en C.A.O (qui a forgé notre intuition [CARRE.COMYN.87ab]) ou de façon analogue en génie logiciel. L'idée est de voir un processus de conception d'une entité ("de longue vie", composant mécanique, électronique, logiciel) à travers l'évolution de celle-ci des niveaux les plus abstraits (niveau fonctionnel) jusqu'aux niveaux les plus concrets d'implantation physique selon l'interprétation de la hiérarchie de classes comme modèle d'abstraction/concrétisation d'objets. L'intégration des règles trouverait ici son intérêt pour représenter les choix de conception à chaque niveau et donc de traduire le processus d'évolution.

L'autre thème que nous considérons fait référence à tout problème d'"identification" d'entités sur une typologie de concepts telle qu'une hiérarchie de classes peut la représenter. Il s'agit ici des problèmes de classification tels que l'on peut les rencontrer en apprentissage ou en diagnostic de situations.

Il semble que le thème de l'évolution d'objet soit un champ d'investigation aussi vaste que peut l'être le concept motivant d'objet.

REFERENCES BIBLIOGRAPHIQUES

******

[AGHA.86] G. AGHA
"AN OVERVIEW OF ACTOR LANGUAGES"
ACM SIGPLAN Notices, Vol. 21, n° 10, 1986.

[AIT-KACI.NASR.86] H. HAIT-KACI and R. NASR
"LOGIN : A LOGIC PROGRAMMING LANGUAGE WITH BUILT-IN INHERITANCE"
The Journal of Logic Programming, pp. 185-215 1986.

[ALBERT.85] P. ALBERT
"PROLOG ET LES OBJETS"
5èmes Journées Les Systèmes Experts et leurs applications. Avignon, 1985.

[ALBERT.86] P. ALBERT
"KOOL"
Présentation aux 6èmes Journées Les Systèmes Experts et leurs applications. Avignon. 1986.

[AMERICA.87] P. AMERICA
"INHERITANCE AND SUBTYPING IN A PARALLEL OBJECT ORIENTED LANGUAGE"
BIGRE n° 54, ECOOP'87, PARIS, 1987.

[BEGG.84] V. BEGG
"SYSTEMES EXPERTS ET CAO"
Editions Hermès, 1984.

[BENOIT et AL.86] C. BENOIT, Y. CASSEAU et C. PHERIVONG
"LORE : UN LANGAGE OBJET RELATIONNEL ET ENSEMBLISTE"
BIGRE + GLOBULE 48. Journées L.O.O., PARIS, 1986.

[BEZIVIN.84] J. BEZIVIN
"SIMULATION ET LANGAGES ORIENTES OBJET"
BIGRE + GLOBULE 41. Journées L.O.O., BREST, 1984.

[BISHOP.86] J. BISHOP
"DATA ABSTRACTION IN PROGRAMMING LANGUAGES"
ADDISON WESLEY. International Computer Science series, 1986.

[BLAKE.COOK.87] E. BLAKE AND S. COOK
"ON INCLUDINC PART HIERARCHIES IN OBJECT ORIENTED LANGUAGES, WITH AN IMPLEMENTATION IN SMALLTALK"
BIGRE n° 54, ECOOP'87. PARIS, 1987.

[BOBROW and AL.86] D.G. BOBROW, K. KAHN, G. KICZALES, L. MASINTER, M. STEFIK and F. ZYBEL
"COMMONLOOPS, MERGING LISP AND OBJECT-ORIENTED PROGRAMMING"
Proceedings of the OOPSLA'86 Conference.
PORTLAND, 1986.

[BOBROW.STEFIK.87] D.G. BOBROW and M.S. STEFIK
"THE LOOPS MANUAL"
Mémo KB-VLSI-81-13. XEROX PALO ALTO RESEARCH CENTER. 1981.

[BOBROW.WINOGRAD.77] D.G. BOBROW and T. WINOGRAD
"AN OVERVIEW OPF KRL A KNOWLEDGE REPRESENTATION LANGUAGE"
Vol. 1, Cognitive Science, 1977.

[BONNET.85] A. BONNET
"ETUDE CRITIQUE ET SYNTHESE SUR LES LANGAGES ORIENTES OBJET POUR L'INTELLIGENCE ARTIFICIELLE"
Génie Logiciel, n° 2, 1985.

[BORNING.79] A. BORNING
"THINGLAB - A CONSTRAINT ORIENTED SIMULATION LABORATORY"
Research report SSL-79-3. XEROX PALO ALTO RESEARCH CENTER. 1979.

[BORNING.INGALLS.82] A.H. BORNING and D.H. INGALLS
"MULTIPLE INHERITANCE IN SMALLTALK.80"
Proceedings of the AAAI'32. PITTSBURGH, 1982.

[BRACHMAN.83] R.J. BRACHMAN
"WHAT IS-A IS AND ISN'T : AN ANALYSIS OF TAXONOMIC LINKS IN SEMANTIC NETWORKS"
IEEE Computer, vol. 16, n° 10,1983.

[BRESSAN.88] S. BRESSAN
"PROGRAMMATION LOGIQUE ET PROGRAMMATION ORIENTEE OBJET"
Mémoire de D.E.A. LIFL, LILLE. 1988.

[BRIOT.83] J.P. BRIOT
"INSTANCIATION ET HERITAGE DANS LES LANGAGES OBJETS"
Thèse, LITP, PARIS, 1983.

[BRIOT.COINTE.86] J.P. BRIOT and P. COINTE
"THE OBJVLISP MODEL : DEFINITION OF A UNIFORM, REFLEXIVE AND EXTENSIBLE OBJECT ORIENTED LANGUAGE"
Rapport n° 1, CMI, PARIS,1986.

[BRUCE.WEGNER.86] K.B. BRUCE and P. WEGNER
"AN ALGEBRAIC MODEL OF SUBSTYPES IN OBJECT-ORIENTED LANGUAGES"
(DRAFT). ACM SIGPLAN Notice, vol. 21, n° 10. 1986.

[C.G.E.86] "LORE, AN OBJECT ORIENTED PROGRAMMING ENVIRONMENT"
Laboratoires de MARCOUSSIS, C.G.E. RESEARCH Center. MARCOUSSIS, 1986.

[CARDELLI.WEGNER.85] L. CARDELLI and P. WEGNER
"ON UNDERSTANDING TYPES, DATA ABSTRACTION, AND POLYMORPHISM"
ACM Computing Survey, vol. 17, n° 4, 1985.

[CARRE.88] B. CARRE
"FOVL, DES OBJETS AUX FRAMES EN OBJVLISP"
ETCA, PARIS, LIFL, LILLE
RAPPORT DE RECHERCHE EN COURS.

[CARRE.COMYN.87a] B. CARRE et G. COMYN
"INSTANCIATION ET REPRESENTATION"
Rapport IT n° 94, LIFL, LILLE, 1987.

[CARRE.COMYN.87 b] B. CARRE et G. COMYN
"CONTRAINTES LIEES A UNE REPRESENTATION PAR OBJETS"
Rapport IT n° 95, LIFL, LILLE, 1987.

[CARRE.COMYN.87c] B. CARRE et G. COMYN
"ETUDE ET SPECIFICATION D'UN NOYAU ORIENTE OBJET POUR LA REALISATION D'UN OUTIL DE DEVELOPPEMENT DE SYSTEMES EXPERTS"
Rapport IT n° 96, LIFL, LILLE, 1987.

[CARRE.COMYN.88] B. CARRE et G. COMYN
"ON MULTIPLE CLASSIFICATION, POINTS OF VIEW AND OBJECT EVOLUTION"
Artificial Intellligence and Cognitive Sciences.
Edited by J. DEMONGEOT, T. HERVE, V. RIALLE and C. ROCHE.
Manchester University Press, 1988

[CERCONE.McCALLA.87] N. CERCONE and G. McCALLA Editors
"THE KNOWLEDGE FRONTIER. ESSAYS IN THE REPRESENTATION OF KNOWLEDGE"
Springer Verlag, 1987.

[CHOURAQUI.DUGERDIL.86] E. CHROURAQUI et P. DUGERDIL
"APPLICATIONS DES LANGAGES ORIENTES OBJET à LA C.A.O. DE L'ARCHITECTURE"
BIGRE + GLOBULE.48, Journées L.O.O., PARIS, 1986.

[COINTE.84] P. COINTE
"UNE EXTENSION DE VLISP VERS LES OBJETS"
Science of Computer Programming 4".
NORTH-HOLLAND, 1984.

[COINTE.86] P. COINTE
"THE OBJVLISP KERNEL : A REFLEXIVE LISP ARCHITECTURE TO DEFINE A UNIFORM OBJECT-ORIENTED SYSTEM".
Workshop on Meta-Level Architecture and Reflection. ALGHERO. Octobre 1986.

[COMYN.87] G. COMYN
"REPRESENTATION DES CONNAISSANCES, PROBLEMES POSES PAR LES REPRESENTATIONS MIXTES"
Exposé invité. E.P.F.L., LAUSANNE, 1987.

[DAHL-NYGAARD.65] O.J DAHL and K. NYGAARD
"BASIC CONCEPTS OF SIMULA, AN ALGOL BASED SIMULATION LANGUAGE"
Digital simulation in operational research.
S.H. HILLINGDALE ed. 1965.

[DIXNEUF.al.88] P. DIXNEUF, A. MELLER, M. PORECHERON
"ELOISE : A RULE ORIENTED PROGRAMMING ENVIRONMENT ON LORE"
C.G.E., 1988.

[DREYFUS.79] H. DREYFUS, 1979
"INTELLIGENCE ARTIFICIELLE : MYTHES ET LIMITES"
Ed. FLAMMARION 1984.

[DUCOURNEAU.HABIB.87] R. DUCOURNEAU et M. HABIB
"LA MULTIPLICITE DE L'HERITAGE DANS LES LANGAGES ORIENTES OBJET"
Rapport LIB n° 1/87. BREST, 1987.

[DUCOURNEAU.QUINQUETON.86] R. DUCOURNEAU et J. QUINQUETON
"YAFOOL : ENCORE UN LANGAGE OBJET A BASE DE FRAMES !"
Rapport INRIA n° 72, 1986.

[DUGERDIL.85] P. DUGERDIL
"UNE METHODOLOGIE ORIENTEE OBJET POUR LA REPRESENTATION DES CONNAISSANCES EN C.A.O. ARCHITECTURE"
Rapport Technique GRTC/JUIN 85/66.
GRTC, MARSEILLE, 1985.

[FAHLMAN.79] S.E. FAHLMAN
"NETL : A SYSTEM FOR REPRESENTING AND USING REAL WORLD KNOWLEDGE"
MIT Press. CAMBRIDGE USA.1979.

[FERBER.83] J. FERBER
"MERING : UN LANGAGE D'ACTEURS POUR LA REPRESENTATION ET LA MANIPULATION DES CONNAISSANCES"
Thèse, PARIS-VI, 1983.

[FERBER.84] J. FERBER
"QUELQUES ASPECTS DU CARACTERE SELF REFLEXIF DU LANGAGE MERING"
BIGRE + GLOBULE 41. Journées L.O.O., 1984.

[FIKES.KEHLER.85] R. FIKES and T. KEHLER
"THE ROLE OF FRAME-BASED REPRESENTATION IN REASONING"
Communication fo the CAM, Vol. 28, n° 9, 1985

[FOX.79] M.S. FOX
"ON INHERITANCE IN KNOWLEDGE REPRESENTATION"
Proceedings of IJCAI'79, 1979.

[GALAMBOS. al.86] J.A. GALAMBOS, R.P. ABELSON, J.B. BLACK editors
"KNOWLED STRUCTURES". Editions IEA, 1986.

[GOETZ.86] J.M. GOETZ
"LRO2 : LANGAGE DE PROGRAMMATION ET DE REPRESENTATION DES CONNAISSANCES PAR OBJET"
Manuel d'utilisation. CRIL. PARIS, 1986.

[GOETZ.ROCHE.86] J.M. GOETZ et C. ROCHE
"IMPLEMENTATION DU LANGAGE DE REPRESENTATION D'OBJETS : LRO2"
BIGRE + GLOBULE 48. Journées L.O.O. PARIS, 1986.

[GOLDBERG-ROBSON.83] A. GOLDBERG and D. ROBSON
"SMALLTALK-80, THE LANGUAGE AND ITS IMPLEMENTATION"
ADDISON-WESLEY, 1983.

[GOLDSTEIN.BOBROW.80] J.P. GOLDSTEIN and D.G. BOBROW
"EXTENDING OBJECT ORIENTED PROGRAMMING IN SMALLTALK"
Conference Record of the 1980 LISP Conference.
STANFORD. 1980.

[GOTO.86] S. GOTO
"DESIGN METHODOLOGIES"
Advances in, CAD for VLSI, vol. 6.
S. GOTO Editor, NORTH-HOLLAND,1986.

[HALBERT.O'BRIEN.87] D.C. HALBERT and P.D. O'BRIEN
"USING TYPES AND INHERITANCE IN OBJECT ORIENTED LANGUAGES"
BIGRE N° 54, ECOOP'87, PARIS, 1987.

[HENDLER.86] J. HENDLER
"ENHANCEMENT FOR MULTIPLE INHERITANCE"
SIGPLAN Notices, vol. 21, n° 10, 1986.

[HEWITT.BISHOP.STEIGER.73] C. HEWITT, P. BISHOP and R. STEIGER
"A UNIVERSAL MODULAR ACTOR FORMALISM FOR ARTIFICIAL INTELLIGENCE"
IJCAI'73. STANFORD. 1973.

[HEWITT.SMITH.75] C. HEWITT and B. SMITH
"A PLASMA PRIMER"
MIT ARTIFICIAL INTELLIGENCE LABORATORY
DRAFT, 1975.

[HOFSTADTER.79] D. HOFSTADTER
"GODEL ESCHER BACH, LES BRINS D'UNE GUIRLANDE ETERNELLE". 1979.
Interéditions, 1985.

[INGALLS.78] D.H. INGALLS
"THE SMALLTALK-76 PROGRAMMING SYSTEM, DESIGN AND IMPLEMENTATION"
Conference record of the fifth annual ACM Symposium on principles of programming languages. 1978.

[KRASNER.83] G. KRASNER
"SMALLTALK-80 BITS OF HISTORY, WORDS OF ADVICE"
ADDISON WESLEY, 1983

[KRISTENSEN and AL.83] B.B. KRISTENSEN, O.L. MADSEN, B. MOLLER-PEDERSEN and K. NYGAARD
"ABSTRACTION MECHANISM IN THE BETA PROGRAMMING LANGUAGE"
ACM Conference record of the tenth annual Symposium on principles of programming languages. 1983.

[KROGDHAL.85] S. KROGDHAL
"MULTIPLE INHERITANCE IN SIMULA-LIKE LANGUAGES"
BIT N° 25, 1985.

[KROGDHAL.OLSEN.86] S. KROGDHAL and K.A. OLSEN
"ADA, AS SEEN FROM SIMULA"
Software-Practice and experience, vol. 16, N° 8, 1986.

[LALONDE.PUGH.85] W.R. LALONDE and J.R. PUGH
"SPECIALIZATION, GENERALIZATION AND INHERITANCE, TEACHING OBJECTIVES BEYOND DATA STRUCTURES AND DATA TYPES"
ACM SIGPLAN Notices, vol. 20, n° 8, 1985.

[LANG.PEARLMUTTER.86] K.S. LANG and B.A. PEARLMUTTER
"OAKLISP : AN OBJECT-ORIENTED SCHEME WITH FIRST CLASS TYPES"
Proceedings of the OOPSLA'86 Conference.
PORTLAND. 1986.

[LAURIERE·86] J.L. LAURIERE
"INTELLIGENCE ARTIFICIELLE. RESOLUTION DE PROBLEMES PAR L'HOMME ET LA MACHINE"
EYROLLES, 1986.

[LE VERRAND.82] D. LE VERRAND
"LE LANGAGE ADA : MANUEL D'EVALUATION"
DUNOD, 1982.

[LIEBERMAN.81] H. LIEBERMAN
" A PREVIEW OF FACT 1"
MIT. A.I. MEMO N° 625. 1981.

[LIEBERMAN.86.a] H. LIEBERMAN
"DELEGATION AND INHERITANCE : TWO MECHANISMS FOR SHARING KNOWLEDGE IN OBJECT ORIENTED SYSTEMS"
BIGRE + GLOBULE 48. Journées L.O.O., PARIS, 1986.

[LIEBERMAN.86.b] H. LIEBERMAN
"USING PROTOTYPICAL OBJECTS TO IMPLEMENT SHARED BEHAVIOUR IN OBJECT ORIENTED SYSTEMS"
Proceedings of the OOPSLA'86 Conference.
PORTLAND. 1986.

[LISKOV and AL.77] B. LISKOV, A.SNYDER, R. ATKINSON and C. SCHAFFERT
"ABSTRACTION MECHANISMS IN CLU".
Communications of the ACM. Vol. 20, n° 8, 1977.

[LOOMIS and AL.87] M.E.S. LOOMIS, A.V. SHAH and J.E. RUMBAUGH
"AN OBJECT MODELING TECHNIQUE FOR CONCEPTUAL DESIGN"
BIGRE N° 54. ECOOP'87. PARIS, 1987.

[MADSEN.86] O.L. MADSEN
"BLOCK STRUCTURE AND OBJECT ORIENTED LANGUAGES"
ACM SIGPLAN Notices. Vol. 21, n° 10, 1986.

[MATHIEU.87] P. MATHIEU
"ROME : REALISATION"
Mémoire de D.E.A.. LIFL, LILLE, 1987

[MEYER.86] B. MEYER
"GENERICITY VERSUS INHERITANCE"
Proceedings of the OOPSLA'86 Conference.
PORTLAND, 1986.

[MINSKY.75] M. MINSKY
"A FRAMEWORK FOR REPRESENTING KNOWLEDGE"
The psychology of computer vision.
Mc GRAW HILL, Computer Sciences series.
P. H. WINSTON Ed., 1975.

[MITTAL and AL.86] S. MITTAL, D.G. BOBROW and K.M. KAHN
"VIRTUAL COPIES : AT THE BOUNDARY BETWEEN CLASSES AND INSTANCES"
Proceedings of the OOPSLA'86 Conference.
PORTLAND. 1986.

[MOON.86] D.A. MOON
"OBJECT ORIENTED PROGRAMMING WITH FLAVORS"
Proceedings of the OOPSSLA'86 Conference.
PORTLAND. 1986.

[NIESSEN.86] C. NIESSEN
"ABSTRACTION REQUIREMENTS IN HIERARCHICAL DESIGN METHODS"
In advances in CAD for VLSI, vol. 6,
"DESIGN METHODOLOGIES", S. GOTO Ed , NORTH HOLLAND 1986.

[NILSSON.82] N.J. NILSSON
"PRINCIPLES OF A.I."
SPRINGER VERLAG, 1982.

[NYGAARD.86] K. NYGAARD
"BASIC CONCEPTS IN OBJECT ORIENTED PROGRAMMING"
ACM SIGPLAN Notices, vol. 12, n° 10, 1986.

[NYGAARD.DAHL.78] K. NYGAARD and O.J. DAHL.
"THE DEVELOPMENT OF THE SIMULA LANGUAGES"
ACM SIGPLAN Notices, vol.13, n° 8, 1978.

[PFEIFER.86] R. PFEIFER
"ON THE USE OF EXPERIENCE IN EXPERT SYSTEMS"
6èmes Journées Internationales Les Systèmes Experts et leurs applications.
Avignon, 1986.

[PIATELLI-PALMARINI.79] M. PIATELLI-PALMARINI
"THEORIES DU LANGAGE. THOERIES DE L'APPRENTISSAGE. LE DEBAT ENTRE Jean PIAGET et Noam CHOMSKY"
Centre Royaumont pour une science de l'homme.
Editions du SEUIL. 1979.

[PITETTE.86] G. PITETTE
"PROGRAMMATION ORIENTEE OBJET AVEC ADA : POSSIBILITES, DIFFICULTES, EXPERIENCE D'ENSEIGNEMENT"
BIGRE + GLOBULE 49. Journées ADA, PARIS, 1986.

[PINSON.81] S. PINSON
"REPRESENTATION DES CONNAISSANCES DANS LES SYSTEMES EXPERTS"
R.A.I.R.O. Informatique, vol. 15, n° 4. 1981.

[PUGH.84] J.R. PUGH
"ACTORS, THE STAGE IS SET"
ACM SIGPLAN Notices, vol. 19, n° 3, 1984.

[RECHENMANN.86] F. RECHENMANN
"SHIRKA : SYSTEME DE GESTION DE BASES DE CONNAISSANCES CENTREES OBJET"
Manuel d'utilisation. INRIA. 1986.

[RECHENMANN.VIGNARD.85] F. RECHENMANN, P. VIGNARD
"CRIKA : QUAND LES REGLES RENCONTRENT LES SCHEMAS"
Rapport de recherche INRIA n° 64, 1985.

[REICHEL.87] H. REICHEL
"INITIAL COMPUTABILITY. ALGEBRAIC SPECIFICATIONS AND PARTIAL ALGEBRAS"
Clarendon Press OXFORD. 1987.

[RICH.83] E. RICH
"ARTIFICIAL INTELLIGENCE"
MC GRAW HILL. 1983.

[ROBERTS.GOLDSTEIN.77] R.B. ROBERTS and I.P. GOLDSTEIN
"THE FRL PRIMER"
Report AIM-408. MIT, CAMBRIGDE. 1977.

[ROCHE.84] C. ROCHE
"EAQUE-LRO, GENERATION DE S.E., APPLICATION A DES PROBLEMES D'ORDONNANCEMENT"
Thèse 3ème Cycle - INPG, 1984.

[SANDBERG.86] D. SANDBERG
"AN ALTERNATIVE TO SUBCLASSING"
Proceedings of the OOPSLA'86 Conférence.
PORTLAND. 1986.

[SAVARIA.86] Y. SAVARIA
"CONCEPTION DES CIRCUITS VLSI"
Notes de cours.
16ème session de l' Ecole Internationale
d'Informatique A.F.C.E.T. - MONTREAL 1986.

[SNYDER.86a] A. SNYDER
"COMMONOBJECTS : AN OVERVIEW"
A.C.M. SIGPLAN Notices, vol. 21, n° 10, 1986.

[SNYDER.86b] A. SNYDER
"ENCAPSULATION AND INHERITANCE IN OBJECT ORIENTED PROGRAMMING LANGUAGES"
Proceedings of the OOPSLA'86 Conference.
PORTLAND. 1986.

[STEFIK and AL.86] M.J. STEFIK, D.G. BOBROW and K.M. KAHN
"INTEGRATED ACCESS-ORIENTED PROGRAMMING INTO A MULTIPARADIGM ENVIRONMENT"
IEEE SOFTWARE., January 1986.

[STEFIK.BOBROW.85] M.J. STEFIK and D.G. BOBROW
"OBJECT-ORIENTED PROGRAMMING : THEMES AND VARIATIONS".
A.I. MAGAZINE, WINTER 1985.

[STROUSTRUP.86] B. STROUSTRUP
"AN OVERVIEW OF C++"
ACM SIGPLAN Notices, Vol 21, n° 10, 1986.

[STROUSTRUP.86] B. STROUSTRUP
"THE C++ PROGRAMMING LANGUAGE"
ADDISON WESLEY. 1986.

[STROUSTRUP.87] B. STROUSTRUP
"WHAT IS OBJECT-ORIENTED PROGRAMMING"
BIGRE n° 54, ECOOP'87. PARIS, 1987.

[SUSSMAN-STEELE.80] G.J. SUSSMAN and G.L.STEELE
"CONSTRAINTS : A LANGUAGE FOR EXPRESSING ALMOST-HIERARCHICAL DESCRIPTIONS"
Artificial Intelligence, vol. 14, 1980.

[VOYER.87] R. VOYER
"MOTEURS DE SYSTEMES EXPERTS"
EYROLLES, 1987.

[WEGNER.86] P. WEGNER
"CLASSIFICATION IN OBJECT ORIENTED SYSTEMS"
ACM SIGPLAN Notice, Vol. 21, n° 10, 1986.

[WEINREB.MOON.81] D. WEINREB and D. MOON
"LISP MACHINE MANUAL"
Symbolics Inc. 1981.

[WIRTH.71] N. WIRTH
"PROGRAM DEVELOPMENT BY STEPWISE REFINMENT"
Communications of the ACM, vol. 14, n° 4. 1971.

[WRIGHT.FOX.83] J.M. WRIGHT and M.S. FOX
"SRL/1.5 USER MANUAL".
Draft Intelligence Systems Laboratory, the Robotics Institute. CARNEGIE-MELLON UNIVERSITY, PITTSBURGH, 1983.

[YONEZAWA and AL.86] A. YONEZAWA, J.P. BRIOT, E. SHIBAYAWA
"OBJECT ORIENTED COPNCURENT PROGRAMMING IN ABCL/1"
Proceedings of the OOPSLA'86 Conference.
PORTLANG 1986.

[YOUNG.PROCTOR.86] S.J. YOUNG and C. PROCTOR
"UFL : AN EXPERIMENTAL FRAME LANGUAGE BASED ON ABSTRACT DATA TYPES".
THE COMPUTER JOURNAL, VOL. 29, N° 4, 1986.

Résumé:

Cette thèse se propose dans un premier temps d'étudier les fondements du paradigme orienté objet classe/instance/héritage à la Smalltalk pour la représentation des connaissances. L'approche orientée objet est analysée des points de vues conceptuel, méthodologique et technique.

Nous insistons sur l'opposition entre hiérarchie conceptuelle de classes (lien sémantique IS-A) et hiérarchie structurelle d'objets (lien sémantique IS-PART). Nous montrons que la première, à laquelle est attaché le mécanisme d'inférence propre au modèle qui est l'héritage (multiple), constitue l'un des aspects les plus déclaratifs par sa capacité d'expression de classifications (multiples) d'objets.

Puis nous présentons ROME (Représentation d'Objet Multiple et Evolutive), langage orienté objet d'expérimentation de nouveaux concepts issus de notre analyse précédente.

ROME est défini à la base par un noyau métacirculaire, inspiré de OBJVLISP, dans la lignée du modèle précédent mais original par l'apport de notions nouvelles. En particulier le concept de point de vue, associé à celui de classe, permet de réaliser une stratégie d'héritage multiple paramétrée et sans conflits, fondée sur un principe d'indépendance entre classes.

Enfin la notion originale de Représentation Multiple et Evolutive est proposée et intégrée au modèle orienté objet. Il s'agit d'amplifier l'interprétation de la hiérarchie (multiple) de classes comme modèle de généralisation/affinement dynamique d'objets.

Cette notion facilite l'expression de classifications multiples d'objets sous des points de vue indépendants en offrant la possibilité de les rattacher à plusieurs classes.

Elle permet par ailleurs de faire évoluer ces objets, en rendant dynamique le rattachement multiple précédent, selon une sémantique algébrique précise. Elle offre un cadre d'étude au problème très vaste de l'évolution d'entités comme on peut la concevoir en systèmes experts, en CAO, en génie logiciel...

Nous montrons que son intégration dans les concepts orientés objet de base est possible par la remise en cause du mécanisme d'instanciation classique comme seul moyen d'exploiter la hiérarchie de classespour les objets eux-mêmes, ce qui induit une contrainte de représentation d'objet unique et figée.

L'apport de cette dernière notion devrait contribuer à l'intérêt du modèle orienté objet pour la représentation des connaissances.

Mots clés:

Représentation des connaissances, langages orientés objet, langages de frames, méthodologie, objet, classe, métaclasse, métacircularité, mixin, héritage multiple, instanciation, SMALLTALK, OBJVLISP, encapsulation, abstraction, généralisation, affinement, spécialisation, agrégation, IS-A, IS-PART, classifications, points de vue, représentation multiple, évolution d'objet.