Académie de MONTPELLIER
Université de Montpellier-II

Contribution à l'étude de l'évolution des langages à objets et des outils associés

THESE D'HABILITATION A DIRIGER DES RECHERCHES

présentée le 23 février 1998 à
l'Université de Montpellier-II
par

Christophe Dony

devant le jury composé de :

Yves Caseau (Rapporteur) : Directeur Scientifique Bouygues Telecom,
Pierre Cointe (Examinateur) : Professeur Ecole des Mines de Nantes,
Roland Ducournau (Examinateur) : Professeur Université Montpellier-II,
Jacques Ferber (Président) : Professeur Université Montpellier-II,
Amedeo Napoli (Rapporteur) : Chargé de recherche CNRS,
Jean-François Perrot (Rapporteur) : Professeur à l'université Paris VI,
Joël Quinqueton (Examinateur) : Directeur de recherche INRIA.

pour obtenir le titre de

DOCTEUR HABILITE A DIRIGER DES RECHERCHES

Remerciements

Je suis très honoré que Yves Caseau, Pierre Cointe, Roland Ducournau, Jacques Ferber, Amedeo Napoli et Jean-François Perrot aient accepté de faire partie de mon jury et les en remercie. Leurs travaux ont tous, d'un thème à l'autre, très largement contribué à ma formation et influencé mes recherches. Je remercie de plus Yves Caseau, Amedeo Napoli et Jean-François Perrot d'avoir accepté d'être rapporteurs.

Je suis très honoré de la présence de Joël Quinqueton dans ce jury. Je le remercie pour son accueil dans l'équipe LMOA du LIRMM, pour son soutien, ses conseils et aussi pour nos discussions "smalltalkiennes".

Je remercie Michel Habib pour son accueil et son aide lors de mon arrivée à Montpellier ainsi que pour ses conseils.

Je tiens à remercier Marianne Huchard et Thérèse Libourel avec lesquelles j'ai plaisir à collaborer. Qu'elles reçoivent ici l'expression de mon amitié.

Je remercie Jacques Malenfant, avec lequel travailler, de visu puis électroniquement, a également été un plaisir, pour son amitié.

Je remercie également Philippe Reitz pour notre collaboration et ses nombreux coups de mains efficaces.

Merci enfin à tous ceux, au LIRMM et ailleurs, qui m'ont aidé d'une manière ou d'une autre.

Merci à ma famille, à Catherine, Léa et Antoine.

à Antoine

Table des matières

Chapitre 1

Introduction

1.1 Contexte

J'ai réalisé mes recherches aux laboratoires de Marcoussis (centre de recherche Alcatel-Alsthom dans la division informatique, au sein de l'équipe MAIA, pendant mon stage de DEA (1985), puis au sein de l'équipe LORE pendant ma thèse d'université. J'ai ensuite travaillé dans l'équipe mixte RXF-LITP (Rank-Xerox France - Laboratoire d'Informatique Théorique et de Programmation) dirigée par Pierre Cointe. Je suis depuis 1993 membre du LIRMM (Laboratoire d'informatique, de Robotique et de Micro-Electronique de Montpellier) dans l'équipe "Langages et Modèles à base d'Agents et d'Objets" dirigée par Joël Quinqueton.

Mes travaux traitent des environnements de programmation par objets, le mot environnement englobant les notions de langages, de formalisme de représentation, de méthodologie de programmation et d'outils d'aides aux programmeurs. Les langages à objets, nés au début des années 1970 [BDMN73], sont sortis des laboratoires de recherche au milieu des années 1980 et sont depuis le début des années 1990 utilisés à grande échelle dans un très grand nombre de domaines de l'informatique. Ces langages, l'architecture des programmes que l'on écrit avec et les techniques de conception et de réalisation de ces programmes n'ont cessé d'évoluer.

Le succès de ce formalisme de programmation est lié a des possibilités nouvelles qui ont apporté des réponses à un important défi des années passées : produire le plus efficacement possible des logiciels de qualité [Mey90].

- Le formalisme objet offre la possibilité de décrire un système informatique de façon proche de la vision mentale que l'humain en a, c'est-à-dire par une description en principe quasi littérale des entités qui composent ce système.
- Il offre la possibilité de produire des programmes modulaires et aisés à maintenir.
- Il permet ensuite de produire assez aisément des systèmes et des logiciels ouverts ayant des qualités en terme d'aptitude à l'évolution, c'est-à-dire, extensibles et réutilisables[1]. Ces aptitudes sont fondées d'une part sur la mise en œuvre judicieuse de deux formes de polymorphisme [CW85], la surcharge et le polymorphisme d'inclusion et d'autre part sur la possibilité de réalisation de systèmes auto-décrits (ou réflexifs).

Passer de ces possibilités à la pratique dans le cas d'applications de taille importante a

[1] Je comprend l'¡¡extensibilité¿¿ comme la possibilité d'ajouter, simplement et de façon sûre, de nouvelles fonctionnalités ou de nouveaux types de données à un système, par exemple sans avoir à modifier le code existant. La ¡¡réutilisabilité¿¿, quand à elle, désigne la possibilité d'adapter des fonctionnalités ou des données existantes à de nouvelles situations ; par exemple la possibilité qu'un algorithme existant puisse être appliqué sans modification de code, à de nouvelles sortes de données.

nécessité des évolutions ; et ce d'autant plus que le succès du formalisme a contraint les concepteurs de langages à en figer trop tôt certains aspects, avant que certains sujets n'aient été complètement étudiés. L'évolution à porté sur les langages, sur les méthodes de conception, sur les environnements de réalisation et enfin sur l'architecture des programmes. Les langages ont évolué de par l'introduction ou la gestion correcte de fonctionnalités ; citons par exemple l'héritage multiple [DHH+95], les représentations réflexives (voir références au chapitre 7), ou la gestion des exceptions (voir chapitre 4) pour n'en citer que quelques unes. Des variantes du modèle à classes ont été proposées dans la mise en pratique de l'idée de programmation par objets, par exemple celle de programmation par prototypes. Les méthodes de conception par objets ont été développées et approchent d'une première normalisation [RBP+91, Boo93, Cor97]. Des modèles d'architecture logicielle réutilisable et extensible (*framework*) voire une conception réutilisable (*design-pattern* [GHJV95]) ont été proposés.

Bon nombre de ces évolutions utilisent le formalisme objet lui-même, qui est ainsi devenu un moteur pour son propre développement en offrant de nouvelles perspectives de représentation ; citons par exemple l'intégration dans les langages à objets de systèmes de gestion des exceptions, de points de vues, de contraintes, de règles ... à base d'objets.

1.2 Domaines d'étude

Les recherches auxquelles j'ai participé s'inscrivent dans l'étude de ces évolutions pour passer de capacités théoriques à des possibilités plus effectives. Elles concernent donc l'évolution des modèles, des langages, des environnements de programmation et de l'architecture des programmes à objets. Une manière générale de les présenter est d'énoncer quelques questions auxquelles j'ai tenté d'apporter des réponses.

- Comment les langages à objets doivent-ils évoluer pour mieux répondre aux exigences du génie logiciel ?
- Comment les environnements d'aide aux utilisateurs et les outils de génie logiciel peuvent-ils être améliorés par une utilisation plus généralisée du formalisme de représentation par objets et des systèmes réflexifs ?
- Le modèle à classe doit-il être le formalisme unique de structuration des programmes à objets, quelles sont ses limites ? Quels sont ses alternatives ?
- Parmi les alternatives proposées au modèle à classes, qu'est-ce exactement que le modèle à prototypes ? Quelles sont ses caractéristiques ? Quel est son avenir ?
- Une des caractéristiques du modèle à prototypes est d'introduire de nouvelles formes de partage entre objets, sur lesquelles se fondent de nouvelles possibilités de réutilisation. Quelle sont ces formes de partage ? Apportent-elles quelque chose au partage induit par la relation d'héritage entre classes ? Si oui, pour quelles applications ?
- Qu'est qu'une bonne architecture de programme ? Est-il possible de transformer des programmes pour améliorer leur architecture ? Comment ?
- Comment les langages doivent-ils évoluer pour mieux permettre la réalisation de systèmes ouverts, capables d'évoluer ?

J'ai tenté de répondre à ces questions, non pas de façon générale mais par l'étude approfondie, seul ou dans le cadre de diverses collaborations, d'un ensemble de cas précis.

- **Evolutions des langages à objets.**
  - Je me suis intéressé à la fiabilité des programmes via l'étude de systèmes de gestion des exceptions dédiés à la programmation par objets [Don88a, Don89b, Don89a, Don90b,

Don90c, DPW92].

- Toujours dans le cadre de l'évolution des langages, mais vu sous l'angle des modèles de programmation, j'ai étudié, en collaboration avec Jacques Malenfant et Daniel Bardou (dans le cadre de sa thèse), dans un projet initié par Pierre Cointe, la programmation par prototypes, une alternative à la programmation par classes. Le modèle à classe est par certains côtés fort complexe, de plus il contraint fortement les objets et donc la représentation des connaissances au sein des programmes. La programmation par prototypes s'est construite de façon expérimentale au travers de divers prototypes de langages ; nous nous sommes attachés à sa compréhension et à l'étude de ses structures de données (les objets sans classes), de ses mécanismes (le clonage et la délégation) et applications. Plus généralement, nous nous sommes posés le problème de l'intérêt de ce modèle de programmation par objets [DMC92, MCDM92, BD95, BD96, BDM96, DMB97, DMB98].
- Enfin, un dernier point relatif à l'évolution des langages auquel je me suis intéressé est la représentation réflexive, par objets, de leurs structures de données et de leurs mécanismes de calcul [Don89b, Don88b, MCDM92, MDC92, MDC96]. Réaliser des langages auto-décrits ou réflexifs me semble être une évolution fondamentale et irréversible[2] qui a des applications dans tous les autres domaines que j'ai abordés. Elle est par ailleurs fondamentale dans l'optique de la réalisation de systèmes ouverts.

– **Evolution des programmes.**
L'évolution des programmes recouvre deux idées.

- D'une part la structure ou (architecture) des programmes a évolué, Le passage de la théorie à la pratique s'est matérialisé par la découverte de savoir-faire relatifs à la construction de *frameworks*, c'est-à-dire de hiérarchies de classes organisées en vue de futures extensions. Je n'ai pas directement travaillé sur la ¡¡théorie des *framework*¿¿ mais j'ai essayé d'en produire : un pour la gestion des exceptions d'une application[Don89b, Don90b], un autre pour implanter simplement de nouveaux langages à prototypes [DMC92, Don97]. Un troisième dédié aux applications en apprentissage et étudié par J-D Ruvini dans le cadre de sa thèse est en cours de réalisation.
- D'autre part elle sous-tend l'idée de passage des notions de ¡¡programme¿¿ ou de bibliothèque d'objets à celle de composant logiciel. Un composant peut se présenter sous différentes formes (classes - interface, spécification, hiérarchies de classes, code exécutable), en fait n'importe quelle entité logicielle utile. Je pense que la programmation par objets est un support adéquat pour l'évolution vers l'assemblage de composants. Dans ce cadre un problème important est l'assemblage de hiérarchie de classes. Ce problème rejoint d'autres recherches d'actualité qui traitent de la restructuration ou de la ¡¡rétro-conception¿¿ d'applications existantes, initialement mal construites (par exemple sans prise en compte d'extensions futures), ayant mal vieilli (inadaptables à l'évolution des besoins) ou développées dans d'autres formalismes. Dans ce contexte, mes recherches, en collaboration avec Hervé Dicky, Marianne Huchard et Thérèse Libourel, portent sur la construction ou restructuration semi-automatique des bibliothèques de classes [DDHL94a, DDHL94b, DDHL95, DDHL96, DHL97] et plus nouvellement sur la fusion de hiérarchies.

– **Evolution des méthodes et des environnements de programmations**
Je m'intéresse depuis mon stage de DEA aux outils d'assistance au programmeur intégrés dans les environnements. J'ai réalisé un ensemble d'outils d'aide à la mise au point des

[2] Par exemple, tous les langages à objets offrent aujourd'hui à leurs programmeurs, une possibilité de manipuler (au moins en lecture, ce qui est une vision minimaliste de la réflexivité) les classes dans leurs programmes.

programmes. J'ai aussi initié plus récemment une étude relative à la réalisation d'un *framework* pour la réalisation d'application utilisant des outils d'apprentissage. Ce *framework* est appliqué à l'assistance aux utilisateurs d'environnements interactifs, plus précisément à l'apprentissage et à l'automatisation des tâches répétitives des programmeurs. Cette étude est menée au LIRMM en collaboration avec Philippe Reitz et Jean-David Ruvini (dans le cadre de sa thèse).

La plupart de mes études sont étayées par des réalisations logicielles dans lesquelles j'ai tenté d'utiliser au mieux le formalisme objet et les possibilités réflexives offertes par les langages que j'ai utilisés. Inversement, j'ai évidemment essayé de produire moi-même des systèmes paramétrables, extensibles et réutilisables, auto-décrits et ouverts [Don97].

Mes travaux s'inscrivent dans les activités [DCC+94, DCC+96] du groupe ¡¡Evolution des langages à objets¿¿ du GDR Programmation (version 1992-1997), dont j'ai été le responsable. Ce groupe fait partie du pôle ¡¡Programmation par Objets¿¿ de ce GDR dont je suis également coordonnateur. L'objectif du groupe ELO était de réaliser une animation scientifique autour d'un ensemble d'équipes travaillant sur l'exploration des aspects mal définis et en évolution de la programmation par objets que sont, par exemple, la programmation sans classes, la modélisation de l'héritage (multiplicité, exceptions), l'apport de la réflexion, ou la représentation de connaissances complexes et évolutives. Ce groupe est composé de 7 équipes de recherche françaises.

Le mémoire est organisé comme suit. Le chapitre 2 contient mon curriculum vitae détaillé. Le chapitre 3 contient la liste de mes publications et réalisations logicielles. Les chapitres 4, 5, 6 et 7 posent les problèmes, résument les résultats, introduisent la consultation des articles et font un point prospectif pour chacun des domaines que j'ai abordés. Enfin, les chapitres 9, 10, 11 et 12 regroupent les articles joints à ce mémoire.

Chapitre 2

Curriculum Vitae détaillé

2.1 Etat Civil

Né le : 22 septembre 1959,
Adresse professionnelle : LIRMM, 161 rue Ada, 34392 Montpellier Cedex 5,
téléphone : 04 67 41 85 33,
courrier électronique : dony@lirmm.fr,
page : http ://www.lirmm.fr/ dony/.

2.2 Formation

- 1984 : Maîtrise d'informatique, Université Paris VI. Mention bien.
- 1985 : DEA Langages, Algorithmique et Programmation, Université Paris VI. Mention bien.
- 1989 : Doctorat de l'université Paris-VI. Mention très honorable. Titre : Apport des langages à objets au génie logiciel, application à la gestion des exceptions et à l'environnement de mise au point.

2.3 Expérience Professionnelle

Mes activités dans ces différents emplois sont détaillées dans les sections suivantes.

- 1984-1985 Centre Mondial en Informatique et Ressources Humaines : Stagiaire dans le groupe ¡¡didacticiel¿¿ dirigé par Gérard Weindenfeld.
- 1985-1986 Laboratoires de Marcoussis (centre de recherche d'Alcatel-Alsthom) : Stagiaire de DEA à la division informatique, membre du groupe ¡¡langages et méthodes¿¿.
- 1986-1989 Laboratoires de Marcoussis (centre de recherche d'Alcatel-Alsthom) : Ingénieur CIFRE à la division informatique (DIN), membre du groupe ¡¡langage à objets¿¿.
- 1989-1992 Rank-Xerox France & LITP (Université Paris VI). Ingénieur de recherche Rank-Xerox France, membre de l'équipe mixte RXF/LITP.
- 1992-1997 Université Montpellier-II. Maître de Conférences ISIM - Membre du LIRMM département ARC.

2.4 Thèmes de recherches

Etude des langages à objets, des outils pour la programmation par objets, de l'architecture des programmes et de l'utilisation du formalisme objet. Applications à la gestion des exceptions, à la réalisation d'outils d'aide à la programmation, à la programmation par objets sans classes, à la réorganisation semi-automatique de programmes et aux systèmes réflexifs.

2.5 Synthèse chronologique des recherches et réalisations

2.5.1 Laboratoires de Marcoussis (centre de recherche d'Alcatel-Alsthom) - 1985-1989

Stage de DEA sur les environnements de programmation Lisp, applications au projet de la machine Lisp MAIA. Réalisation de ma thèse [Don89b] sur les systèmes de gestions des exceptions et les environnements de mise au point pour les langages à objets. Application au langage LORE conçu par Yves Caseau et largement utilisé à la division informatique.

Gestion des exceptions et programmation par objets

- Conception [Don88a, Don90b, Don89a, Don89b] et implantation [log-4] du système original de gestion des exceptions de LORE, intégré au langage, utilisé à la DIN et globalement repris pour la version commerciale du langage.
- Etat de l'art sur la gestion des exceptions [Don89b].

Environnements

- Réalisation d'une étude sur les environnements de programmation LISP. Implantation d'une version du traceur *zstep*, proposé par Henry Lieberman, pour l'environnement *Common-lisp* de la machine *Lisp Maia* [Don85].
- Conception [Don89b, Don88b, BCCD87] et implantation [log-3] d'outils de mise au point (debugger, traceur, stepper, toplevel) pour LORE V2.2, intégrés au langage.

Divers

- Collaboration à l'implantation en Common-Lisp du noyau de la version 2.2 de LORE [BCCD87].
- Etude croisée sur l'apport des langages à objets au génie logiciel et sur les besoins en outils logiciels pour la programmation par objets [Don89b].

2.5.2 Rank-Xerox France & LITP (Université Paris-VI)- 1989-1992

Participation à l'animation de l'équipe ¡¡objets¿¿ dirigée par Pierre Cointe. Travaux sur les concepts, l'environnement de programmation et l'implantation du langage SMALLTALK et des langages à objets en général. Travaux sur les systèmes réflexifs et sur les langages à prototypes.

Gestion des exceptions (suite)

- Nouvelle conception, adaptation à SMALLTALK et implantation en SMALLTALK [Don90b, Don90c, DPW92] du système de gestion des exceptions précédent, connexion avec le debugger [log-2].
- Etude des problèmes de gestion des exceptions pour les langages d'acteurs (cf. DEA de P.Petit).

Langages à Prototypes

- Première étude et classification des langages à prototypes [DMC92].
- Conception et implantation de Prototalk, une plate-forme SMALLTALK permettant de

simuler les langages objets à prototypes (par opposition aux langages à classes) existants et d'en concevoir de nouveaux [Don97] [log-0]

Systèmes réflexifs

– Etude de la réflexion dans les langages à prototypes. Conception d'un système à base de méta-objets permettant, au sein d'un langage à prototypes, de paramétrer l'envoi de message et l'application des méthodes [MCDM91, MCDM92, MDC92]. Collaboration avec Pierre Cointe et Jacques Malenfant.

Environnements (suite)

– Implantation d'outils de trace et d'un pas-à-pas graphique pour SMALLTALK (cf. DEA de Y.Thomas).
– Premières études sur l'assistance automatique aux utilisateurs de l'environnement SMALLTALK (cf. DEA de J-F.Bernier)

2.5.3 LIRMM Montpellier 1992-1997

Intégration au LIRMM dans l'équipe "Object - Acteurs - Agents" dirigée par Joël Quinqueton. Création du groupe ELO (Evolution des langages à objets) du GDR Programmation. Suite des travaux sur les langages à prototypes et les systèmes réflexifs. mise en œuvre de diverses collaborations au LIRMM sur la réorganisation de hiérarchies sur les applications des techniques d'apprentissages et sur les agents logiciels.

Langages à Prototypes (suite)

– Extension de la plate-forme Prototalk de simulation de langages, opérationnelle et disponible sur internet.
– Etude critique des les langages à prototypes en collaboration avec Daniel Bardou et Jacques Malenfant [DMB97, DMB98].
– Travail sur la sémantique et les utilisations du mécanisme de délégation. Premières études pour la représentation de points de vues par des objets morcelés [BD95, BD96].

Réorganisation des hiérarchies de classes

– Projet sur la réorganisation de hiérarchies en collaboration avec Hervé Dicky, Marianne Huchard et Thérèse Libourel. Première étude de l'insertion automatique de classes dans des hiérarchies d'héritage [DDHL95, DDHL94a, DDHL94b] ; co-encadrements du DEA de Florent Pages.
– Suite de l'étude avec prise en compte du problème de la surcharge [DDHL96, DHL97] ; co-encadrements du DEA de Nicolas Prade). Mise en œuvre de l'algorithme en OBJVLISP par Nicolas Prade et moi-même [log-1].
– Extension de l'étude à la réalisation d'une plate-forme de manipulation de hiérarchies dans le cadre d'une convention avec le CNET. Encadrement de la thèse de Hervé Leblanc qui débute actuellement.

Systèmes réflexifs (suite)

– Participation à l'étude formelle de l'arrêt de la régression infinie dans le système à base de méta-objets précédent [MDC96].
– Première étude de l'adaptation de l'architecture de la tour réflexive à l'interprétation des programmes à objets (cf. stage de DEA de Frédéric Bosch).

Assistance aux utilisateurs

– Application des techniques d'apprentissage à l'assistance automatisée à l'utilisation de logiciels interactifs. Application à l'environnement SMALLTALK. Etude réalisée en collaboration avec Ph.Reitz (LIRMM). Co-encadrement des stages de DEA de J-F Bernier, de F.Jugla et J-D.Ruvini. Co-encadrement avec Philippe Reitz de la thèse de J-D.Ruvini,

actuellement en début de seconde année.
Un assistant est désormais opérationnel au sein de l'environnement SMALLTALK, ses premières versions ont été présentées dans [RD97, JCP97]. Un projet annexe, né de ce travail est la réalisation d'un *framework* pour la réalisation d'applications en apprentissage.

Applications des technologies objet

– Participation, sous forme de consultant à la mise en œuvre et à la présentation du système, au projet de supervision de réseaux par systèmes multi-agents [EDQD96] dirigé par Joël Quinqueton. Ce projet a été réalisé en collaboration avec le CNET, il utilise un système d'acteurs et est mis en œuvre en SMALLTALK. Il a par ailleurs de fortes connexions avec le projet d'assistant précédemment décrit puisqu'il utilise les techniques d'apprentissage pour assister un utilisateur humain.

2.6 Responsabilités de, et participations à des, groupes de recherche

– **Responsable de l'équipe ¡¡ELO¿¿ du GDR Programmation du CNRS**.
Montage du projet ELO (Evolution des langages à objet) en 1992 avec Michel Habib. Le projet regroupe plus de 20 chercheurs dans plusieurs équipes (EMN Nantes, IRIN Nantes, LIFL Lille, LIRMM Montpellier, LGI2P Nimes) et d'une équipe industrielle (Equipe de M.Dao du Cnet) travaillant sur l'utilisation, la compréhension, la sémantique ainsi que diverses évolutions des modèles et des langages de programmation à objets.
– **Responsable du pôle objet du GDR Programmation**.
Le pôle est composé de l'équipe ELO présentée ci-dessus et l'équipe ¡¡Interopérabilité¿¿ dirigé par Pierre Cointe et comprenant une quinzaine de personnes dans trois équipes. La direction du pôle implique la gestion du budget, la représentation au sein du GDR, l'organisation des journées de pôle (cf. organisation de manifestations) et la coordination et synthèse des rapports d'activités des deux équipes [DCC+94, DCC+96].
– Membre du groupe ¡¡objets et classification¿¿ du PRC/IA.
– Le pôle objet est en cours réorganisation dans le cadre de la restructuration des GDR.

2.7 Comités de programmes

– **Comités de programme**
- RPO'93 : Représentation par objets.
- LMO'94, LMO'95, LMO'96 : Langages et modèles à objets.
- Revue TSI (Techniques et Sciences Informatique), numéro spécial ¡¡objets¿¿ , 1996.
- JFLA'98 : Journées Francophones des Langages Applicatifs.

– **Reviewer - revues et conférences**
- Conférence jointe ECOOP/OOPSLA'90.
- ECOOP'91, ECOOP'92, ECOOP'95 : *European conference on object-oriented programming.*
- OOPSLA'91, OOPSLA'92 : *Object-oriented programming systems languages and applications.*
- RPO'92, RPO'93, LMO'94, LMO'95, LMO'96.
- Technique et Science Informatiques.

- Numéro spécial de ¡¡Communication of the ACM¿¿ sur les ¡¡Object-Oriented Frameworks¿¿.
- ACM Computing surveys.

2.8 Organisation de manifestations

– ECOOP Workshop sur ¡¡Gestion d'exceptions et programmation par objets¿¿ Initiateur et organisateur avec Russel Winder (University College London) d'un workshop sur la gestion des exceptions dans le cadre de ECOOP'91. Une synthèse en a été publiée dans ¡¡ACM OOPS Messengers¿¿ [DPW92].
– Journées du GDR Programmation Organisation scientifique, dans le cadre des journées du GDR Programmation des journées du pôle ¡¡programmation par objets¿¿ qui ont eu lieu :
- à Orsay en 1993,
- à Grenoble en 1995,
- à Orléans en 1996.
- à Rennes en 1997
– Membre du comité d'organisation de LMO'97

2.9 Encadrement de thèses

– Encadrement de la thèse de Daniel Bardou (soutenance prévue en janvier 1998) sur la sémantique des objets morcelés et du partage de connaissance dans les langages à prototypes - sous la responsabilité de Joël Quinqueton.
– Encadrement avec Philippe Reitz de la thèse de J-D Ruvini (début en octobre 1996) sur l'assistance automatisée à l'utilisation d'environnements interactifs (cf. section 8 - sous la responsabilité de Joël Quinqueton.
– Encadrement avec M.Huchard et T.Libourel de la thèse de Hervé Leblanc (début décembre 1997) dans le cadre du contrat CNET.
– Participation à l'encadrement de la thèse de Babak Esfandiari sur la supervision de réseaux par des systèmes multi-agents (soutenue en décembre 1996).

2.10 Encadrement de stages de DEA

Encadrement (ou co-encadrement) de 14 stages :

– Stages Réalisés dans le cadre du DEA ITCP de l'université Paris-VI. (89-92)
- 1990, Pascal Petit : Spécification d'un système de gestion des exceptions pour la plateforme Actalk de simulation des langages d'acteurs,(en collaboration avec J-P. Briot).
- 1990, Philippe Fontenoy : Etude et comparaison des ateliers de conception et de spécification par objets, Stage réalisé aux laboratoires de Marcoussis.
- 1991, Yves Thomas : Spécification, implantation et intégration à l'environnement SMALLTALK d'outils de trace et d'un pas-à-pas graphique.
- 1991, Jean-François Bernier : Utilisation du moteur d'inférence NeOpus pour l'écriture de règles d'expertise sur les programmes SMALLTALK (en collaboration avec François Pachet du LAFORIA).
- 1992, Christian Santellani : Introduction à la description des objets dans le cadre de l' ¡¡Open Systems Interconnection¿¿, Stage réalisé aux laboratoires de Marcoussis.

- 1992, Mohamed Tahar Loucif : Intégration de Act1 et de OBJECT-LISP dans la plate-forme Prototalk de simulation des langages à prototypes.

– Stages réalisés dans le DEA d'informatique de l'université Montpellier-II (92-96)
-1993, F.Jugla : Etude et premier prototype d'un assistant à l'utilisation de l'environnement SMALLTALK.
-1994, Florent Pages : Etude de méthodes de réorganisation des hiérarchies de classes (en collaboration avec Marianne Huchard et Thérèse Libourel du LIRMM).
- 1994, Cyril Bourdin : Points de vues et représentation multiples dans les langages à objets (en collaboration avec Bernard Carré du LIFL).
- 1994 : Eric Gelin : Contribution au typage statique d'un langage à objets - étude et applicabilité au langages à objets du typage dans les langages fonctionnels. (en collaboration avec Philippe Reitz du LIRMM).
-1995, Jean-David Ruvini : Suite du stage de Florent Pages sur l'assistance à l'utilisation des environnements de programmation (en collaboration avec Philippe Reitz du LIRMM).
-1995, Nicolas Prade : Suite du stage sur les réorganisation de hiérarchies, prise en compte de la surcharge et du masquage (en collaboration avec Marianne Huchard et Thérèse Libourel du LIRMM).
- 1996, Frédéric Bosch : Tours réflexives et interprétation des langages à objets.
- 1996, Thibaud Brunel : Etude des ¡¡Design Patterns¿¿. Application : production de patterns à partir d'un logiciel de gestion.

2.11 Responsabilité de contrats

- Responsable d'un contrat IA2 état/région en collaboration avec la société CIMM à Béziers portant sur l'étude des normes pour l'interopérabilité des applications : CORBA et OLE/ACTIVE-X.

- Co-responsable avec M.Huchard et T.Libourel d'un contrat avec le CNET d'une durée de trois ans, portant sur l'étude des réorganisations de hiérarchie (cf. section 6).

2.12 Enseignement et responsabilités pédagogiques liés à la recherche

– Cours dans le DEA ITCP (Université Paris-VI 1989-92)
DEA d'¡¡Informatique Théorique, Calcul et Programmation¿¿ de l'Université Paris-VI. Interventions dans le cours d'option de programmation par objets en 89-90. Co-responsable du cours de tronc commun et du cours d'option de programmation par objets en 90-91 et 91-92.

– DEA d'Informatique (Université Montpellier-II)
- 1992 et 1993 : 10 H de cours dans le module ¡¡Programmation avancée¿¿ sur les langages à prototypes et divers aspects de recherche en programmation par objets.
- 1994 : Responsable du module de 50H ¡¡Objets, acteurs, agents¿¿, au sein duquel j'ai donné 12H de cours.
- 1995 : Responsable du cours de tronc commun de ¡¡Programmation par objets¿¿(16H) et co-responsable avec Joël Quinqueton du module de (40H) ¡¡objets-acteurs-agents¿¿ dans lesquels j'ai donné 16H de cours.
- 1996 : Responsable du cours de tronc commun de ¡¡Programmation par objets¿¿(16H) -

10H de cours.
- 1997 : Responsable avec Roland Ducournau du module (Tronc commun et option 35H) ¡¡Langages et Modèles à Objets¿¿.

– Ecole Doctorale SPI-Montpellier(95, 96, 97)
- Co-responsable avec Philippe Reitz d'un cours de 20H d'initiation à C++.

Chapitre 3

Liste de publications et réalisations logicielles

Notes
- Les publications majeures de cette liste sont, par ordre chronologique : [Don88a, Don90b, DMC92, MDC96, BD96, DDHL96, DMB97]
- Les publications jointes à ce document sont : [Don90b, DMC92, BD96, Don97, DHL97, MDC96], leur clé est soulignée dans la liste suivante.

— **Chapitres de livres**

[DMB97] C. Dony, J. Malenfant, and D. Bardou. Les langages à prototypes. In R.Ducournau, J.Euzenat, and A.Napoli, editors, *Langages et Modèles d'Objets*. INRIA - Collection Didactique, 1998. A paraître.

[DMB98] C. Dony, J. Malenfant, and D. Bardou. Classification of object-centered languages. In Ivan Moore, James Noble, and Antero Taivalsaari, editors, *Prototype-Based Object-Oriented Programming*. Springer-Verlag, 1998. A paraître.

— **Revues avec comité de lecture et sélection**

[MDC96] Jacques Malenfant, Christophe Dony, and Pierre Cointe. A semantic of introspection in a prototype-based language. *Kluwert International Journal on Lisp And Symbolic Computation*, 9(2/3) :153–179, 1996.

[EDQD96] Babak Esfandiari, Gilles Deflandre, Joel Quinqueton, and Christophe Dony. Agent-oriented techniques for network supervision. *Annals of Telecommunications*, 51(9-10) :521–529, 1996.

— **Revues d'information**

[DPW92] C. Dony, J. Purchase, and R. Winder. Report on the ecoop'91 workshop on exception handling and object-oriented programming. *ACM OOPS Messenger*, 3(2) :17–30, April 1992.

— **Conférences internationales avec sélection**

[JCP97] J.D.Ruvini, C.Dony, and P.Reitz. The apprentice and the assistant : two interface agents for smalltalk. In H.S.Nwana and D.T.Nduwu, editors, *Actes de PAAM'98 : The Third International Conference on The Practical Application of Intelligent Agents and Multi-Agents*, pages 237–252, London, UK, 1998.

[DDHL96] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. *ACM Sigplan Notice - Proceedings of ACM OOPSLA'96, Object-Oriented Programming Languages, Systems and Applications*, 31(10) :251–267, October 1996. Egalement en Rapport de Recherche LIRMM No 95054, Février 1996.

[BD96] Daniel Bardou and Christophe Dony. Split objects : a disciplined use of delegation within objects. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'96*, 31(10) :122–137, October 1996.

[MDC92] J. Malenfant, C. Dony, and P.Cointe. Behavioral reflection in a prototype-based language. In Akinori Yonezawa and Brian. C. Smith, editors, *Proceedings of International Workshop on ¡¡ New Models for Software Architecture : Reflection and Meta-Level Architectures ¿¿*, pages 143–153, Tokyo, Japan, November 1992. ACM Sigplan, JSSST, IPJS.

[DMC92] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages : From a new taxonomy to constructive proposals and their validation. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'92*, 27(10) :201–217, October 1992.

[Don90c] Christophe Dony. Improving exception handling with object-oriented design. In *Proceedings of IEEE COMPSAC'90, Fourteenth Computer Software and Applications Conference*, pages 36–42, Chicago, USA, November 1990.

[Don90b] Christophe Dony. Exception handling and object-oriented programming : Towards a synthesis. *ACM SIGPLAN Notices - Proceedings of the joint conference ECOOP-*

OOPSLA'90, 25(10) :322–330, October 1990. Egalement disponible en Rapport de Recherche RXF-LITP No 90-101.

[Don88a] Christophe Dony. An object-oriented exception handling system for an object-oriented language. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP '88*, LNCS 322, pages 146–161, Oslo, August 15-17 1988. Springer-Verlag.

— **Conférence nationales avec sélection**

[DDHL95] H. Dicky, C. Dony, M. Huchard, and T. Libourel. Ares, adding a class and restructuring inheritance hierarchies. In *Actes des 11ièmes Journées Bases de Données Avancées, BDA'95*, pages 25–42, Nancy, 1995.

[BD95] D. Bardou and C. Dony. Propositions pour un nouveau modèle d'objets dans les langages à prototypes. In *Actes de LMO'95, Langages et Modèles à Objets, Nancy*, pages 93–109, October 1995.

[DDHL94a] H. Dicky, C. Dony, M. Huchard, and T. Libourel. Ares, un algorithme d'ajout avec restructuration dans les hiérarchies de classes. In *Actes de LMO'94, Langages et Modèles à Objets, Grenoble*, pages 125–136, October 1994.

[MCDM92] J.Malenfant, P.Cointe, C. Dony, and P. Mulet. Etude de la réflexion de comportement dans le langage self. In *Actes de RPO'92, Représentation par objets : le point sur la recherche et les applications, La Grande-Motte*, page ? ?, June 1992.

[Don89a] Christophe Dony. Apports du formalisme objet aux problèmes de la gestion des exceptions. In EC2, editor, *Actes des Deuxièmes Journées Internationales : Le Génie Logiciel et ses Applications*, pages 401–416, Toulouse, France, December 1989.

— **Workshop internationaux (avec sélection) dans le cadre de conférences**

[MCDM91] J.Malenfant, P. Cointe, C. Dony, and P. Mulet. Reflection in prototype-based object-oriented programming languages. In *Proceedings of OOPSLA'91 workshop on reflection and meta-level architectures*, October 1991.

— **Journées diverses**

[RD97] Jean-David Ruvini and Christophe Dony. Ébauche de deux agents interfaces pour l'environnement smalltalk-80. Actes des *Cinquièmes journées francophones sur l'intelligence artificielle distribuée et les systèmes multi-agents*, Poster Session, La colle-sur-Loup, April 1997.

[BDM96] Daniel Bardou, Christophe Dony, and Jacques Malenfant. Comprendre et interpréter la délégation, une application aux objets morcelés. In *Actes des Journées du GDR Programmation*, Orléans, November 1996.

— **Rapports de recherche - Articles nouveaux**

[DHL97] C. Dony, M. Huchard, and T. Libourel. Automatic hierarchies reorganization, an algorithm and case studies with overloading. Technical report 97279, LIRMM, 1997. Version étendue de [DDHL96].

[Don97] C. Dony. Prototalk : A framework for the design and the operational evaluation of prototype-based languages. Technical Report 97254, LIRMM, 1997. Description du framework utilisé dans [DMC92].

— **Rapports de recherche**

[DDHL94b] H. Dicky, C. Dony, M. Huchard, and T. Libourel. Ares, un algorithme d'ajout

avec restructuration dans les hiérarchies de classes. Technical Report 94039, LIRMM, May 1994. version étendue de l'article LMO'94.

[Don90a] C. Dony. Etude critique du système de gestion des exceptions de smalltalk - objectworks v2.5. Technical report, Rapport de recherche de l'equipe RXF-LITP, June 1990.

[Don89b] C. Dony. *Langages à objets et génie logiciel. Applications à la gestion des exceptions et à l'environnement de mise au point.* PhD thesis, Université de Paris VI, June 1989.

[Don88b] Christophe Dony. Une implantation réflexive du debugger de lore, utilisant le système de gestion des exceptions. Rapport de recherche No L9.0, Laboratoires de Marcoussis, CRCGE, June 1988.

[BCCD87] Christophe Benoit, Françoise Carré, Yves Caseau, and Christophe Dony. *Lore 2.2 : manuel de référence.* Laboratoires de Marcoussis, CRCGE, March 1987.

[Don85] C. Dony. Etude des environnements de programmation lisp - implantation d'un outil graphique d'évaluation en pas-à-pas pour la machine maia. Rapport de dea, LITP - Laboratoires de Marcoussis, CRCGE, September 1985.

— **Rapports d'activité**

[DCC+94] Christophe Dony, Bernard Carré, Pierre Cointe, Roland Ducournau, Michel Habib, Marianne Huchard, Amedeo Napoli, Mourad Oussalah Roger Rousseau et Jean-Claude Royer. Rapports d'activités 1994 du pôle objet du gdr programmation. Rapport d'activité, 1994. rédacteur principal pour les activités du groupe ELO et coordonnateur du rapport du pôle.

[DCC+96] Christophe Dony, Bernard Carré, Pierre Cointe, Roland Ducournau, Michel Habib, Marianne Huchard, Amedeo Napoli, Mourad Oussalah Roger Rousseau et Jean-Claude Royer. Rapports d'activités 1996 du pôle objet du gdr programmation. Edité par le CNRS, 1996. Coordonnateur pour les activités du groupe ELO et pour le rapport du pôle.

— **Liste des logiciels réalisés, utilisés et pour certains diffusés et accessibles sur le réseau**.

[log-0] Christophe Dony. Logiciel : Plate-forme de conception et de simulation de langages à prototypes, Version 2. Disponible sur le site http ://www.lirmm.fr/ dony/research.html. Implanté en SMALLTALK (Parcplace-Objectworks v4.0).

[log-1] Nicolas Prades and Christophe Dony. Logiciel : Ares - version 0.1 experimentale, sans la surcharge, 1996. Disponible sur le site http ://www.lirmm.fr/ dony/research.html. Implanté en OBJVLISP.

[log-2] Christophe Dony. Logiciel : Système de gestion des exceptions pour smaltalk, 1990. Disponible sur le site http ://www.lirmm.fr/ dony/research.html, Implanté en SMALLTALK (Parcplace-Objectworks v4.0).

[log-3] Christophe Dony. Logiciel : Outils de mise au point pour lore2.2 (debugger symbolique, traceur, stepper), 1988. Implantés en Lucid Common-Lisp et en Lore.

[log-4] Christophe Dony. Logiciel : Système de gestion des exceptions de lore2.2, 1988. Système original intégré au langage, utilisé aux laboratoires de Marcoussis et repris pour la version commerciale du langage, Implanté en Lucid-Common-Lisp et en Lore.

[log-5] Christophe Dony. Logiciel : Pas-à-pas interactif et graphique pour maia, 1986. Implémenté en Common-Lisp.

Chapitre 4

Evolutions liées au génie logiciel : le cas de la gestion des exceptions

- Article joint (cf. chapitre 9) relatif à cette section : [Don90b]
- Publications relatives à cette section : [Don90b, DPW92, Don88a, Don90c, Don89b, Don89a, BCCD87]

4.1 Introduction

Dans le cadre de l'évolution des langages à objets pour la production de logiciels de qualité s'est posé, au milieu des années 80, le problème de la réalisation de programmes résistant aux erreurs. Les langages et les applications devenant de plus en plus complexes, la possibilité de réaliser des programmes fiables est devenue à cette époque un argument important pour le succès d'un langage. Doter un langage d'un système de gestion des exceptions permet de mettre en œuvre les notions de fiabilité, d'encapsulation, de modularité et de réutilisabilité.

- Fiabilité : l'exécution d'un programme doit laisser un système dans un état cohérent même si elle se termine de façon anormale.
- Modularité : l'interface d'un module doit spécifier toutes les exceptions que l'un de ses services peut signaler et la mise en œuvre du module respecter ces spécifications.
- Encapsulation : Une encapsulations résistant aux exceptions peut être définie comme un objet logiciel capable de cacher l'implémentation des services qu'il implémente, i.e. capable d'intercepter les exceptions de bas niveaux levées par l'activation de ces services et de retourner, même en cas d'exception, des réponses claires et conformes à ses spécifications.
- Réutilisation : la possibilité qui en résulte de pouvoir spécifier par avance des réponses appropriées aux situations exceptionnelles améliore l'aptitude des programmes à être réutilisés.
- Enfin, maîtriser la propagation des exceptions est un prérequis indispensable pour bâtir des environnements de programmation et plus particulièrement des environnements de mise au point.

A l'époque, les systèmes de gestion des exceptions dans les langages à objets étaient insatisfaisants : certains ne permettaient pas la réalisation de véritables encapsulations[1], d'autres avaient un pouvoir d'expression trop réduit, d'autres enfin ne prenaient pas en compte certaines caractéristiques essentielles du formalisme objet ; ils étaient en fait simplement dérivés

[1] Un signalement d'exception peut révéler la représentation interne d'un type abstrait.

de ceux existant dans les langages procéduraux comme PL/I, Ada [IBH+79], Clu[LS79] ou Mesa [MMS81]. Parallèlement au mien, de nombreux système ont été développés à la fin des années 1980 ; citons parmi les principaux ceux de C++ [KS90], de Clos [Ste90], d'Eiffel [Mey92], d'Objectworks-Smalltalk [Par90].

J'ai spécifié, implanté et intégré à SMALLTALK un système de gestion des exceptions dédié au formalisme objet[2]. Ce système était une refonte et une adaptation à SMALLTALK de celui développé pour le langage LORE [Cas87] durant ma thèse d'université [Don88a, Don89a, Don89b]. Le cahier des charges, la spécification et la mise en œuvre de ce système sont décrites dans [Don90b]. Nous nous proposons de résumer ici les points fondamentaux qui sous-tendent ses spécifications et de faire le point sur les recherches dans le domaine. Le lecteur se reportera au chapitre 9 ou l'article joint présente, plus en détail et avec de nombreux exemples, le système.

4.2 Rappel de Terminologie

Une exception dénote une situation qui empêche la continuation usuelle du calcul, une erreur dénote une exception pour laquelle le calcul ne pourra pas reprendre au point ou il a été interrompu. On peut distinguer [Goo75] les exceptions de domaine (non vérification des assertions d'entrée d'une opération), de portée (non vérification des assertions de sortie) et enfin les exceptions programmées (permettant de réaliser explicitement des branchements non locaux).

La détection d'une situation exceptionnelle conduit à l'interruption du calcul. Le signalement consiste à rechercher une continuation exceptionnelle ou traitant (*handler* en anglais), correspondant à la fois à l'exception signalée et au point d'activation du signalement, puis à l'invoquer en lui fournissant si possible des informations sur les causes et le contexte de l'interruption. Le traitant permet de ¡¡rattraper¿¿ des occurrences d'exceptions.

Un traitant est associé à, ou protège, ou est défini pour, une entité (un programme, une fonction, un bloc, une expression, une variable, une classe, etc). Un traitant a une portée et une durée de vie et peut référencer (rattraper) plusieurs exceptions. Il est possible d'avoir une pile de traitants, un handler associé à une instruction masque celui associé à une instruction englobante. Un traitant est exécuté si l'exception qu'il référence est levée durant l'exécution de la section de code, ou de l'entité, qu'il protège. Son rôle est de remettre le système dans un état tel que le calcul standard puisse reprendre. Usuellement, les traitants peuvent choisir, en fonction du contexte et des informations dont ils disposent, entre (1) transférer le contrôle au point suivant l'instruction de signalement (reprise), (2) détruire le contexte d'exécution situé entre l'instruction de signalement et l'instruction à laquelle le handler est associé, le contrôle reprenant à l'instruction qui suit (terminaison) ou (3) re-signaler une exception (propagation). Pour plus de détails, le lecteur se reportera par exemple à [Goo75, LS79, Don89b].

J'utiliserai pour les exemples une syntaxe inspirée de ce que l'on peut écrire dans les langages procéduraux :

[2]Ce travail a été réalisé au sein de l'équipe mixte RXF-LITP

```
{protected-expression1 ... protected-expressionN};instructions protégées
  {when exception-name1;exceptions à rattraper
    (param1 ... paramN) : ;paramètres du traitant
    handling-instruction1 ... handling-instructionN}};corps du traitant

    signal(exception-name, arg2, ..., argn);signale une exception
    exit(value); au sein du traitant, provoque la terminaison
    resume(value); au sein du traitant, provoque la reprise
```

4.3 Les clés du système.

La gestion des exceptions classique pose un ensemble de problèmes assez complexes, fort bien résumés dans [Goo75] et qu'il serait trop long de reprendre ici. Les trois remarques fondamentales qui sous-tendent mon système sont les suivantes : (1) la modularité nécessite des traitants à portée dynamique, (2) la conception par objets et la réutilisation bénéficient de la possibilité d'associer des traitants aux classes, (3) l'ensemble du système bénéficie considérablement d'une utilisation généralisée du formalisme objet.

4.3.1 Importance des traitants à portée dynamique

Les travaux sur la gestion des exceptions dans les langages procéduraux tels que PL/I, Clu [LS79], Ada [IBH+79] ou Mesa [MMS81] ont montré que la mise en œuvre de la modularité et de la notion d'¡¡encapsulation résistant aux erreurs¿¿ reposait sur une recherche des traitants dans la pile d'exécution ou, autrement dit, sur les traitants associés à des expressions exécutables et ayant une portée dynamique[3].

Implanter des encapsulation résistant aux exceptions nous semble être la conjonction des deux possibilités suivantes :

- ne pas propager les exceptions internes à un module, relevant des détails d'implémentation ;
- propager aux appelants les exceptions ayant le niveau conceptuel d'un résultat (exemple : l'erreur `pile-vide` pour le type `pile`).

Les handlers à portée dynamique associés aux expressions permettent de respecter ces deux propriétés. Ils sont actifs durant toute l'exécution de l'expression à laquelle ils sont associés et sont visibles partout durant ce laps de temps. Ils sont recherchés, en descendant dans la pile d'exécution, à partir du point d'activation de l'instruction de signalement. L'exemple suivant illustre cette idée, la procédure `process-yield` encapsule correctement son utilisation privée d'une liste (des processus actifs) en rattrapant les exceptions liées à la manipulation de listes et en propageant une exception de plus haut niveau conceptuel (`process-was-not-active`).

```
procedure process-yeld (a-process)
  {remove(Active-process-list, a-process)
        when : empty-list, item-not-found :
              {signal (process-was-not-active)}}
```

L'autre avantage important de ces traitants est qu'ils sont exécutés dans leur environnement lexical de définition et peuvent ainsi traiter les exceptions de façon dépendante du contexte en accédant à des variables locales ou à des paramètres, comme dans cette version de la fonction `pgcd`.

```
function pgcd (int a, b)
  {pgcd(b, divide(a, b))
        when : division-by-zero : {exit (b)}}
```

[3] Durée de vie dynamique et portée indéfinie [Ste90].

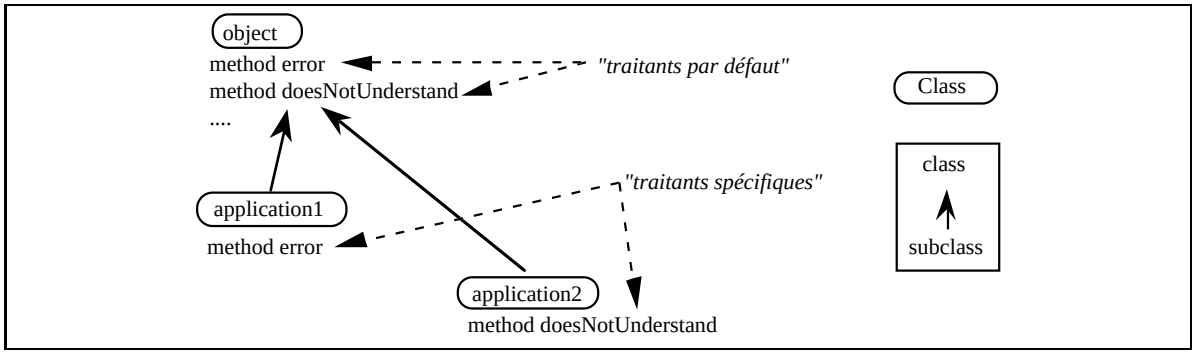


FIG. 4.1 – Traitants à portée lexicale, associés aux classes.

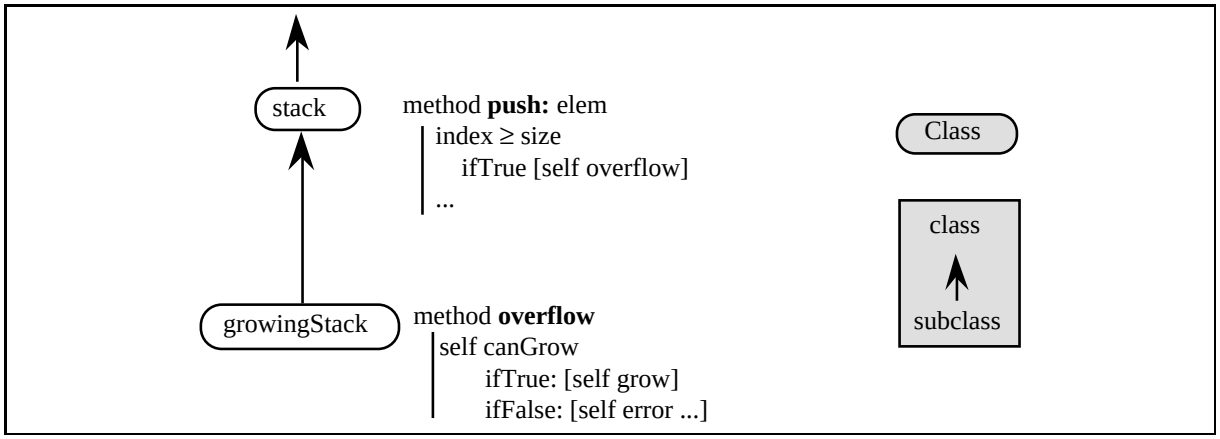


FIG. 4.2 – Traitants associés aux classes et réutilisation.

4.3.2 Importance des traitants associés aux classes

La spécification initiale du langage Smalltalk-80 [GR83] a montré l'intérêt d'associer des traitants aux classes, ce qui revient à associer automatiquement un handler au corps de toutes les méthodes définies sur la classe. On pourrait croire qu'il ne s'agit là que d'une facilité syntaxique assez puissante. Il s'agit en fait d'une possibilité importante en terme de conception par objets.

- Elle permet de définir très simplement des traitants communs à toutes les instances d'une même classe.
- Elle permet de spécifier au niveau de la classe quelles sont les exceptions qui pourront être signalées à ses clients. Elle permet par exemple à un programmeur de spécifier simplement que les exceptions `pile-vide` et `pile-pleine` sont les deux seules qui pourront être propagées en dehors d'une méthode définie sur la classe `Pile`. Il suffit pour cela de définir au niveau de la classe un traitant rattrapant toutes les autres exceptions.
- Elle permet d'anticiper l'ajout de méthodes à une classe. Le traitant sera actif même pour des méthodes non encore écrites ou moment ou il est défini.
- Elle permet bien sûr aux traitants d'être hérités ou redéfinis dans des sous-classes. Il est possible de définir au niveau de chaque classe abstraite un ensemble de traitants par défauts, comme le suggère la figure 4.1.
- Ceci permet enfin de créer de nouveaux types de données en étendant simplement le domaine de définition ou les fonctionnalités des opérations signalant des exceptions. L'exemple de la figure 4.2 illustre cette dernière possibilité : une classe `GrowingStack` (les piles capables de changer de taille dynamiquement est implantée par simple redéfinition d'un handler[4].

[4]Il serait bien sûr possible de faire autrement, l'important, sur cet exemple très simple, est de montrer l'idée.

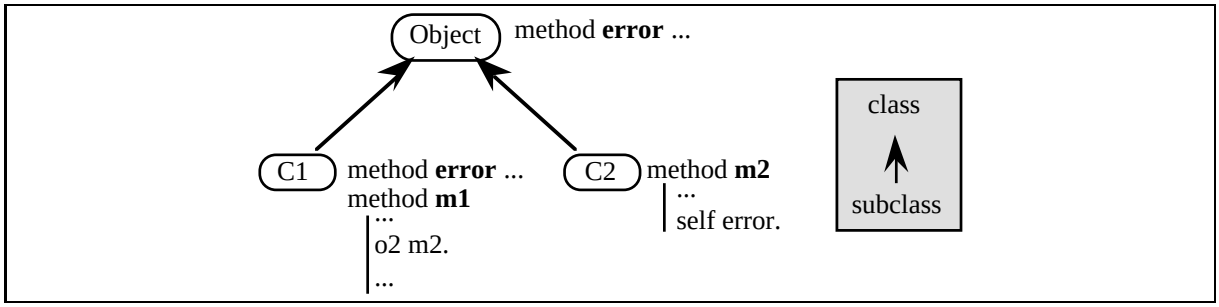


FIG. 4.3 – Problème des traitants à portée lexicale.

Malheureusement ces handlers à la SMALLTALK ont un gros défaut, leur portée est lexicale. La raison en est que la première spécification de ce langage ne proposait pas véritablement de gestion des exceptions et que celle-ci était réalisée simplement avec les construction de base : méthodes et envoi de messages (par exemple le message `error`). C'était évidemment très simple à réaliser mais aussi assez limitatif. Portée lexicale signifie que les traitants ne sont invoqués que si le point d'activation de l'instruction de signalement est textuellement inclus dans une des méthodes définies sur la classe qu'ils protègent. Voyons le problème avec un exemple (figure 4.3) dans lequel `o2` est une instance de la classe `C2`. Le problème est que la méthode `M1` (ou la classe `C1`) ne possèdent simplement aucun moyen de rattraper l'exception que lève l'exécution de la méthode `M1`, exception réellement signalée au sein de la méthode `M2`. L'encapsulation est incorrecte, `M1` ne peut cacher, en cas d'exception, son utilisation d'une instance de la classe `C2`.

L'ensemble de ces analyses suggérait donc un système de cohabitation entre différentes sortes de traitants, associés aux classes et aux expressions, mais ayant tous une portée dynamique. C'est ce qui a été implanté.

4.3.3 Représentation des exceptions et de leurs fonctions

Les systèmes classiques représentaient les exceptions par des chaînes de caractères, des symboles, voire au mieux des variables du type `¡¡exception¿¿`. La représentation par objets des exceptions a été proposée assez tôt dans divers système à objets [Nix83, WM81, Bor86a, Pit88, BDG+88]. Parmi les diverses possibilités, il est vite apparu que la bonne idée était de représenter chaque exception par une classe et chaque occurrence d'une situation exceptionnelle comme une instance de la classe correspondante. Dans cette solution, le signalement d'une exception se traduit par la création d'une instance (nommons la `exc`) de la classe correspondant à l'exception signalée. Les champs de `exc` sont affectés aux valeurs caractéristiques de l'événement et chaque traitant invoqué suite à ce signalement reçoit `exc` comme argument. Les avantages induits par ce choix de représentation de connaissances sont extrêmement nombreux, voici les principaux :

- Une exception (un concept) aussi bien qu'une occurrence dynamique d'exception est une entité de première classe, pouvant être inspectée, modifiée, passée en argument.
- Les exceptions peuvent être organisées en une hiérarchie basée sur des fonctionnalités communes et partagées. `Division-par-zéro` devient naturellement une sous-classe de `erreur-arithmétique`. Cette dernière peut être vue comme une exception abstraite représentant l'ensemble des exceptions arithmétiques, en référence à la notion de classe abstraite, encore qu'ici, il ne soit pas exclu de la signaler, alors qu'on n'instancie pas les classes abstraites.
- L'organisation des exceptions profite des avantages inhérents au formalisme objet : modularité, généricité, partage de code.
- Les traitements par défaut les plus généraux, s'implémentent simplement comme des propriétés définies sur la classe. L'accès à ces comportements est possible pour tout handler par simple envoi de message.
- Une nouvelle exception s'intègre conceptuellement très simplement au reste du système, ce qui permet à une application de développer simplement sa propre gestion en profitant de manière

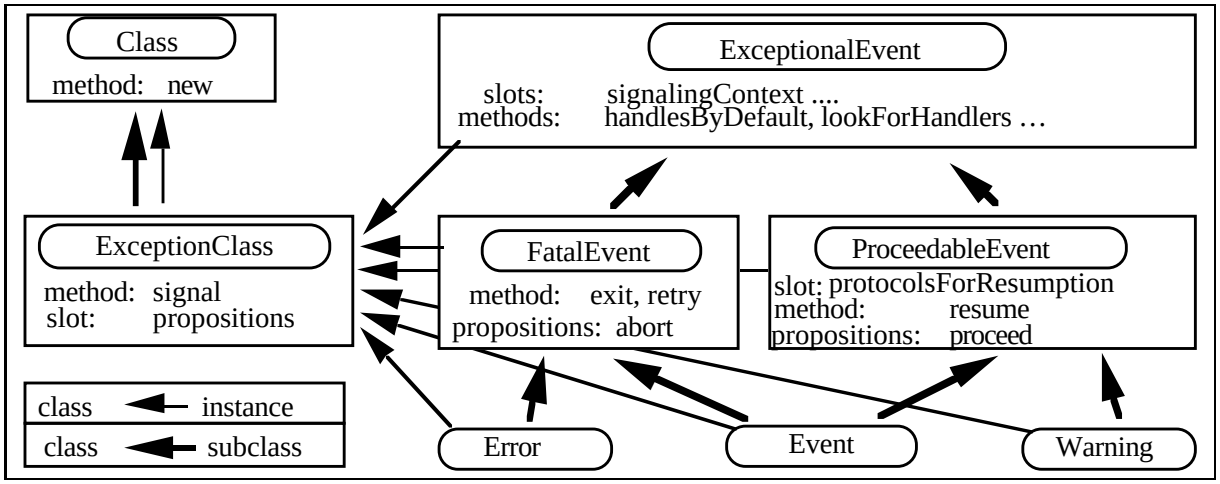


FIG. 4.4 – Représentation des exceptions et de leurs protocoles.

sélective des traitements par défaut déjà en place.

– Toute exception représente le sous-arbre d'héritage dont elle est la racine. On peut ainsi se protéger soit globalement contre les exceptions arithmétiques ou les exceptions d'entrée-sortie, soit plus sélectivement contre la division par zéro. Le pouvoir d'expression obtenu est beaucoup plus important qu'avec les classiques clauses *others* à la ADA.

Le reproche que l'on pouvait faire à tous ces systèmes est qu'ils avaient réifié les données mais ne s'étaient pas intéressés aux comportements. C'est ce que j'ai entrepris de faire.

4.4 Le système

J'ai donc développé un système pour le langage SMALLTALK qui met en œuvre les idées précédentes, ainsi que quelques autres.

– Sur la plan des fonctionnalités, les traitants peuvent être associés à n'importe quelle expression ou aux classes. Un traitant défini sur une classe `C` sera invoqué si une exception est signalée durant l'exécution (et non pas par une instruction incluse dans) d'une méthode définie sur `C`, ou une une de ses sous-classes. Tous les traitants de ce système ont une portée dynamique.

– Il existe 3 sortes de traitants, des plus spécifiques aux plus généraux. Les plus généraux sont associés aux exceptions elle-mêmes, ils proposent des réponses indépendantes de toutes application. Le plus général est celui associé à la racine de la hiérarchie des exceptions qui rattrape tout, remet le système dans un état cohérent et provoque un retour au ¡¡toplevel¿¿ de l'interprète. Les plus spécifiques sont associés aux expressions, comme dans l'exemple de la fonction `pgcd` ou de la fonction `process-yeld`. Les intermédiaires sont associés aux classes des applications et servent à la définition des types abstraits. Les diverses utilisations des divers sortes de traitants sont présentées au chapitre 9, sections 6 et 7.

– Un nouvel algorithme original permet de rechercher les traitants en respectant l'historique du calcul (représenté par la pile d'exécution) et en prenant en compte, à chaque niveau dans la pile, les deux sortes de traitants. Pour chaque bloc de pile, on regarde si un traitant y est défini et sinon, si un traitant est défini sur la classe du receveur courant[5], sinon on passe au bloc précédent dans la pile.

– Sur le plan de la représentation des données. J'ai repris et systématisé [Don90c] la représentation par objets proposée dans les systèmes précédemment cités.
Les différentes sortes d'exceptions sont donc représentées par une hiérarchie de classe mais l'ensemble des protocoles pour traiter les événements exceptionnels également, ce qui n'était pas le cas auparavant (cf. figure 4.4). Tous les traitements sont réalisés par envoi de messages de manière

[5]Le receveur courant pour l'invocation de la méthode correspondant à ce bloc de pile

générique ; une exception de type ¡¡error¿¿ ne comprend pas le message `resume`. Toutes les exceptions sont donc signalées avec la même primitive. L'ensemble des avantages de cette nouvelle représentation d'un strict point de vue génie logiciel sont présentés dans [Don90c] et d'un point de vue plus objet dans [Don90b] (cf. chapitre 9, section 5).

La spécification complète fait intervenir de nombreux autres choix que nous ne développons pas ici, le lecteur de reportera à l'article joint à ce document, mais ils sont moins spécifiques de la programmation par objets. Un de ses choix est celui du modèle avec ¡¡reprise¿¿ qui est cher et rend l'implantation plus complexe. Il nécessite dans notre implantation la création d'une fermeture lexicale à chaque définition de traitant. Il nécessite également des primitives de restauration inconditionnelles de type *unwind-protect*. La reprise me semble importante pour un langage orienté vers le prototypage et disposant d'un environnement de programmation interactif comme SMALLTALK. Elle n'est pas présente en C++ ou JAVA.

4.5 Implantation réflexive

Il est important de dire un mot de l'implantation. Elle est réalisée entièrement en Parcplace-SMALLTALK [6], c'est-à-dire entièrement en termes du langage qu'elle enrichit. Ceci résulte des possibilités réflexives étendues de ce langages dans lequel la plupart des structures de données, classes, méthodes, mais aussi et surtout les blocs de la pile d'exécution[7] sont représentées par des objets et sont donc manipulable en SMALLTALK. La première conséquence est que cette implantation est entièrement accessible au programmeur. La seconde est que l'implantation est concise, lisible, extensible et réutilisable. La troisième conséquence est malheureusement le coût de la chose. Chaque recherche de handler peut nécessiter la réification d'un nombre important de blocs de piles.

Le langage ParcPlace-SMALLTALK a lui-même été doté (dans sa version 2.5) d'un système de gestion des exceptions, développé en même temps que le mien, dont l'implantation initiale utilisait la possibilité réflexive mais a été adaptée afin que les blocs de pile dans lesquels des handlers ont été définis puissent être reconnus à moindre coût par la machine virtuelle.

4.6 Applications du système

J'ai utilisé intensivement ce système pour implanter des outils de mise au point (de recherche des exceptions non traitées !) que j'ai intégrés aux langages LORE puis SMALLTALK. Mon travail a porté sur la boucle d'interaction de l'interprète (autrement dit le *toplevel*), sur les continuations d'exception (handlers) par défaut, et sur l'inspecteur de pile d'exécution (usuellement dénommé *debugger*). Ces outils ont un code lisible et portable parce qu'ils sont écrits intégralement en utilisant les constructions du langage, et en particulier, en ce qui concerne le debugger, en utilisant le système de gestion des exceptions[8] [Don88b]. Par portable, j'entends que si la version du langage LORE sur laquelle j'ai travaillé avait été portée, il n'y aurait pas eu besoin de réécrire le ¡¡debugger¿¿ alors qu'il aurait fallu réécrire le système de gestion des exceptions qui utilisait le langage d'implantation (*Common-Lisp*). Dans le cas de l'implantation en SMALLTALK, ni le système de gestion des exceptions ni le debugger n'auraient eu à être réécrits en cas de portage.

4.7 Conclusion et perspectives

Globalement, au vus des systèmes existants, on peut considérer que le problème de la gestion des exceptions en programmation par objets standard (modèle à classes, programmation séquentielle) a été correctement résolu. Le problème était d'importance : permettre la réalisation de programmes fiables, se

[6] ©Parc-Place systems

[7] Dans le cas de la pile, la réification d'un bloc n'est effectuée qu'à la demande, demande matérialisée par un accès, au sein d'une méthode, à la pseudo-variable `thisContect`

[8] Et donc sans aucun accès direct à la pile d'exécution.

terminant correctement dans tous les cas de figure. J'ai pour ma part contribué à montrer quelles sont les spécificités de la programmation par objets en la matière et en quoi une représentation par objets permet d'améliorer de façon marquante leur simplicité d'utilisation et leur puissance d'expression. Le système que j'ai proposé améliorait les systèmes existants à l'époque par une systématisation de la représentation par objets et par la combinaison originale de divers types de traitants.

Aux alentours de 1992, la plupart des systèmes étaient stabilisés et ils ont peu évolués depuis. Les systèmes de gestion des exceptions que l'on trouve dans de nouveaux langages, comme par exemple celui de JAVA, très proche de celui de celui C++, ne proposent pas de fonctionnalités nouvelles. Ils sont même en retrait, toujours dans le cas de JAVA, de ce que l'on sait faire, ce qui marque dans ce cas le choix d'un compromis entre simplicité et puissance d'expression. Relativement peu de systèmes, en dehors du mien et de ceux présentés dans [Mey92, KMMPN87b, PW90] proposent la définition de traitants au niveau des classes. Nous n'en voyons pas bien la raison car cela ne complique pas fondamentalement l'implantation et apporte beaucoup en terme de conception par objets. Similairement, tous les systèmes ont repris l'idée de représenter les exceptions par des classes mais peu d'entre eux ont été au bout de l'idée pour offrir un traitement générique des situations exceptionnelles.

La comparaison précise de tous ces systèmes est fastidieuse car les variations sont nombreuses et subtiles. Le lecteur intéressé trouvera de nombreux éléments de comparaison dans [Don88a, Don89b] et dans [DPW92]. Le concepteur d'un nouveau langage à objets a en fait aujourd'hui le choix entre diverses politiques de gestion dont les tenants et aboutissants sont connus. Citons deux exemples de politiques en la matière.

Certains systèmes, comme par exemple celui d'Eiffel, sont plus orientés vers la fiabilité des programmes et restreignent volontairement le pouvoir d'expression du programmeur. La gestion d'exceptions en Eiffel, introduite en 1988, est conçue exclusivement pour la gestion des situations exceptionnelles et pas comme une nouvelle structure de contrôle. Son originalité est d'être parfaitement adaptée à la philosophie de la programmation par contrats et fondée sur les constructions propres à ce langage permettant d'écrire des assertions. En Eiffel les exceptions sont signalées quand un contrat n'est pas ou risque de ne pas être respecté, c'est-à-dire essentiellement lorsqu'une méthode échoue, par exemple à cause d'une exception système ou lorsqu'une assertion (pré ou post condition) est violée. Les handlers peuvent être associés uniquement aux corps des méthodes ou aux classes. Ils sont définis dans des clauses spéciales (*rescue clauses*), le corps des handlers est donc séparé du corps de la méthode et ce de façon délibérée[9]. Dans un traitant, le programmeur a le choix entre la ré-exécution de la méthode (*retry*) ou la propagation d'une exceptions après remise des objets concernés dans un état cohérent. D'autres systèmes sont plus orientés vers la puissance d'expression comme celui de Parc-Place SMALLTALK qui est philosophiquement proche du mien, les traitants associés aux classes en moins. Les exceptions y sont représentées par des objets mais pas par des classes, ceci pour ne pas multiplier leur nombre et parce que les classes sont des objets chers. Sa puissance d'expression s'en trouve légèrement réduite [Don90a].

Il reste par contre de nombreuses recherches à effectuer en liaison avec la gestion des exceptions si l'on s'intéresse à des domaines connexes. La gestion d'exceptions pose par exemple des problèmes en preuve de programmes ou en inférence de type : comment intégrer une possibilité de résultat exceptionnel dans le calcul du type rendu par une fonction ? La gestion des exceptions en milieu concurrent et éventuellement réparti, dans les systèmes d'acteurs ou dans les systèmes multi-agent en est un autre exemple. J'ai abordé ce sujet dans le cadre d'une étude en collaboration avec Jean-Pierre Briot sur la spécification d'un système pour Actalk [Bri94] qui s'est matérialisé par un stage de DEA[10] en 1992 mais qui a été interrompue par mon départ de Paris. Ce problème a bien sûr été développé depuis et il se repose aujourd'hui dans les systèmes multi-agents ou dans le contexte de travaux sur la programmation par réutilisation et assemblage de composants logiciels répartis.

[9] La question de cette séparation entre code standard et code exceptionnel revient fréquemment dans les discussions relatives à la gestion des exceptions.

[10] DEA - Pascal Petit : Spécification d'un système de gestion des exceptions pour la plate-forme Actalk de simulation des langages d'acteurs,(en collaboration avec J-P. Briot).

Chapitre 5

Evolution des modèles de programmation : Programmation sans classes

– Articles joints (cf. chapitre 10) relatifs à cette section : [DMC92, BD96, Don97]
– Publications relatives à cette section : [DMB98, DMB97, BD96, Don97, BD95, DMC92, MCDM92]

5.1 Introduction

Une part importante de mon travail est relative aux évolutions des langages (sémantique, implantation, pouvoir d'expression) ainsi qu'à l'évolution des variations sur l'idée de programmation par objets. J'ai participé [BCCD87] à l'implantation de la version Common-Lisp du langage LORE. J'ai développé des extensions et des outils pour LORE et SMALLTALK (voire sections précédentes). Enfin j'ai travaillé sur les langages à prototypes et sur la réflexivité. J'ai évoqué le langage LORE précédemment à propos de gestion des exceptions et je reviendrai sur l'étude des systèmes réflexifs au chapitre 7.

Les langages à prototypes quand à eux sont apparus à la fin des années 80 ; ils proposent une approche de la programmation par objets reposant sur la notion de représentants typiques (prototypes) plutôt que sur celle de descriptions en compréhension (classes) des concepts. Ils se voulaient une alternative au modèle à classe, jugé trop complexe, visant à offrir une conception simplifiée des programmes en même temps que de nouvelles possibilités d'expression des connaissances. Ils ont été développés en assez grand nombre depuis une décennie ; citons quelques noms tels SELF [US87, ABC$^+$95, CUCH91, SU95], KEVO [Tai91, Tai93], NAS [Cod88], EXEMPLARS [LTP86, Lal89], AGORA [Ste94], GARNET [MGD$^+$90, MGdZ92], MOOSTRAP [MC93, Mul95], CECIL [Cha93], OMEGA [Bla94], NEWTON-SCRIPT [Smi94] . D'autres langages, tels OBJECT-LISP [All89] ou YAFOOL [Duc91] ne se réclamant pas de l'approche par prototypes offrent néanmoins des mécanismes proches.

Nous nous sommes intéressés à ces langages pour trois raisons. La première relevait de leur intérêt potentiel pour la simplification de la conception des programmes et pour le prototypage d'applications. La seconde correspondait à un projet d'étude sur la réflexion de comportement, initié par Pierre Cointe en 1991 dans l'équipe RXF-LITP et mené en collaboration avec Jacques Malenfant. Nous avions besoin de langages plus simples que les langages à classes comme support à notre étude. La dernière raison est que nos premières études ont montré leur intérêt potentiel, notamment en terme de pouvoir d'expression, et ont aussi révélé un ensemble de problèmes intéressants pour la compréhension globale de la programmation par objets. Nous nous sommes donc attachés à les comprendre et à les classifier, à analyser les variations dans la sémantique de leurs mécanismes spécifiques, à étudier les problèmes qu'ils posent.

Nous avons en premier lieu proposé dans [DMC92] (cf. chapitre 10, article 1) un recensement des constructions et mécanismes primitifs de la programmation par prototypes (représentation des objets,

création et évolution des objets, expression du partage, mécanismes de calcul), puis une étude de leur sémantique opérationnelle sur laquelle se fonde une première taxinomie. Afin de comprendre la sémantique des mécanismes recensés dans les différents langages, nous les avons implantés dans une plate-forme de simulation nommée *Prototalk*, écrite en SMALLTALK. Cette plate-forme est un framework, conçu pour l'implantation rapide, via une réutilisation maximale, d'interprètes de langages à objets ; elle est décrite dans [Don97] (cf. chapitre 10, article 3). Indépendamment de son intérêt pour la présente étude, elle propose un bon exemple de programmation par objets et de réutilisation. C'est un *framework* ou les interprète sont entièrement réifiés et dans lequel il est possible d'implanter très simplement un évaluateur des expressions d'un langage en exprimant uniquement ses différences avec les évaluateurs déjà implantés. Enfin, après mon arrivée au LIRMM, j'ai travaillé plus spécifiquement sur la sémantique et les applications du mécanisme de délégation en collaboration avec Daniel Bardou dans le cadre de sa thèse et avec Jacques Malenfant. J'ai rédigé ce chapitre[1] comme une synthèse pédagogique des résultats obtenus avec Jacques puis avec Daniel[2].

La caractérisation très générale et informelle des langages à prototypes est relativement aisée : ce sont des langages dans lesquels on trouve en principe une seule sorte d'objets dotés d'attributs [3] et de méthodes, trois primitives de création d'objets : création *ex-nihilo*, *clonage* et *extension* (ou *description différentielle*), un mécanisme de calcul : l'envoi de message intégrant un mécanisme de *délégation*. Ceci étant posé, leur caractérisation, leur utilisation et leur compréhension précise pose en fait un certain nombre de problèmes.

- Leur genèse est moins simple qu'il n'y parait. Beaucoup d'idées proches ont été développées parallèlement en représentation de connaissance, en programmation parallèle et en programmation par objets.
- Il existe diverses interprétations de ce qu'est un prototype, objet concret ou représentant moyen d'un concept.
- La sémantique des mécanismes de base (clonage, copie différentielle, délégation) n'est pas unifiée et autorise différentes interprétations [DMC92, Mal95, BD96, BDM96, Mal96].
- La description différentielle, sur laquelle repose le mécanisme de délégation, rend les objets interdépendants, ce qui pose de nouveaux problèmes et autorise diverses interprétations quant au statut des objets [DMC92, Mal96, BD96, BDM96].
- La délégation n'est pas aux langages à prototypes ce que l'héritage entre classes est au langages à classes. Plus exactement, cette affirmation n'explique rien.
- En même temps que les classes, a été supprimée par exemple, la possibilité d'exprimer que deux concepts partagent certaines caractéristiques. Cette seconde possibilité est si importante en terme d'organisation des programmes que de nombreux langages à prototypes ont cherché à la réintroduire, ce qui a été fait de façon plus ou moins appropriée. Nous parlerons d'un ensemble de problèmes d'¡¡organisation des programmes¿¿ (cf. section 5.9).

Au travers de l'explicitation de ces points, nous posons les questions suivantes :

- Quel est la genèse de langages à prototypes ? Pourquoi ont-ils été développés et d'où sont-ils issus ?
- Pourquoi posent-ils des problèmes et ces problèmes ont-ils des solutions ?
- Qu'est-ce que la délégation ? Quelles sont ses formes ? Quels problèmes pose-t-elle ? Quelles sont ses utilisations valides ?
- Finalement, nous nous demandons si les langages à prototypes proposent une alternative valable aux langage à classes et si oui pour quelles applications ?

5.2 Notion de prototype

On trouve en sciences cognitives l'idée de représenter un concept, ou une famille d'entités, par un représentant distingué ainsi que l'idée de copie différentielle. Dans ce contexte, différents modèles de la notion de concept ont été proposés [CM84, Kle91]. La ¡¡ théorie des prototypes ¿¿ est une extension

[1] Je tiens à remercier Michel Dao, Roland Ducournau, Jérome Euzenat et Amedeo Napoli pour leur relectures avisées de cette partie.

[2] La version complète de cette synthèse est à paraître dans [DMB97]

[3] Nous utilisons ce terme pour désigner une caractéristique non comportementale d'un objet ou d'un *frame*, nous aurions pu utiliser les équivalents que sont ¡¡ champ ¿¿ ou ¡¡ slot ¿¿.

d'un de ces modèles dans laquelle les concepts ne sont décrits ni en compréhension ni en extension mais indirectement au travers de prototypes du concept, c'est-à-dire d'exemples. Cette théorie découle du principe selon lequel l'humain se représente mentalement un concept, identifie une famille d'objets et mène des raisonnements sur ses membres en faisant référence, au moins dans un premier temps, à un objet précis, typique de la famille. Ma ¡¡ 2CV ¿¿ est, par exemple, un prototype du concept de ¡¡ voiture ¿¿, comme ¡¡ netscape ¿¿ l'est pour le concept de ¡¡ navigateur internet ¿¿. On trouve aussi dans la théorie des prototypes la notion de description différentielle qui désigne la possibilité de décrire 'un nouveau représentant du concept via l'expression de ses différences par rapport à un représentant existant.

Afin de mieux expliquer comment ces notions ont été utilisées, il nous apparaît nécessaire de distinguer deux sortes de prototypes que nous rencontrerons dans nos langages : le représentant concret et le représentant moyen d'un concept. Il est préalablement nécessaire d'établir une distinction terminologique entre les objets du monde dont nous souhaitons réaliser une description informatique (que nous appellerons le ¡¡ domaine ¿¿) et les objets de nos langages. Nous utiliserons le terme ¡¡ entité ¿¿ pour désigner les premiers.

– **Représentant concret et instance prototypique.** Le représentant concret, dont ma ¡¡ 2CV ¿¿ est un exemple pour le concept ¡¡ voiture ¿¿, correspond à une entité concrète. Nous reprenons le terme d' ¡¡ instance prototypique ¿¿ pour désigner un représentant concret utilisé comme référence pour décrire ou créer d'autres objets. L'instance prototypique d'un concept est ainsi souvent le premier objet d'une famille.

– **Représentant moyen.** Un représentant moyen représente une entité qui peut être abstraite ou incomplète. Le représentant moyen ne représente aucune entité concrète. Il peut ne posséder que les attributs les plus courants avec les valeurs les plus courantes pour la catégorie d'entités qu'il représente. La ¡¡ ménagère de moins de 50 ans ¿¿ est un exemple célèbre de représentant moyen du concept ¡¡ téléspectateur ¿¿ ; un objet possédant quatre roues et un moteur est un représentant moyen du concept ¡¡ voiture ¿¿. Ses attributs peuvent contenir des valeurs moyennes, par exemple, la femme française typique a 1,8 enfants.

5.3 Utilisations informatiques de la notion de prototype antérieures aux langages à prototypes

5.3.1 Les prototypes en représentation des connaissances

Les langages à prototypes existaient avant que l'appellation n'apparaisse. On trouve ainsi les notions de prototype et de copie différentielle dans la théorie des frames de Minsky [Min75] et dans certains systèmes inspirés de cette théorie comme les langages de frames tels KRL [BW77] ou FRL [RG77] .

> ¡¡ *Les frames sont un formalisme de représentation créé pour prendre en compte des connaissances qui se décrivent mal ... [dans d'autres formalismes] ... comme la typicalité, les valeurs par défauts, les exceptions, les informations incomplètes ou redondantes. La structure d'un* frame *... doit pouvoir évoluer à tout moment, par modification, adjonction ou modification de propriétés.* ¿¿ [MNC+89]

Nous donnons ici une vision simplifiée à l'extrême de ce que sont les frames, sans illustrer leur richesse et leur diversité.

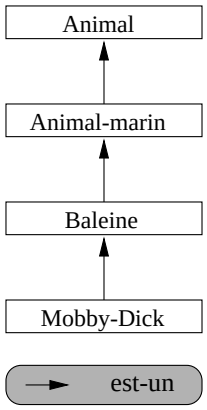
• **Structure d'un frame.** Un *frame* est un ensemble d'attributs. Chaque attribut permet de représenter une des caractéristiques du *frame* et se présente sous la forme d'un couple ¡¡ nom d'attribut – ensemble de facettes ¿¿. La facette la plus courante étant la valeur de l'attribut, nous ne considérerons que celle-ci dans nos exemples. La figure 5.1 propose un exemple de définition d'un *frame* dotée de 4 attributs représentant de façon minimale une baleine.

```
Frame
  nom : "baleine"
  catégorie : mammifère
  milieu : marin
  ennemi : homme
  poids : 10000
  couleur : bleu
```

FIG. 5.1 – Exemple de *Frame*

```
Frame
  nom : "Moby-Dick"
  est-un : baleine
  couleur : blanche
  ennemi : Cpt-Haccab
```

FIG. 5.2 – Description différentielle




```
Frame
  nom : "animal"
  milieu : terrestre
  nourriture : nourriture
  nbPattes : 4
  poids : 10

Frame
  nom : "animal-marin"
  est-un : animal
  milieu : marin
  nourriture : n-marine
  nbPattes : 0
  nbNageoires : 4
```

```
Frame
  nom : "baleine"
  est-un : animal-marin
  nourriture : plancton
  couleur : bleu
  poids : 10000
  ennemi : homme

Frame
  nom : "Moby-Dick"
  est-un : baleine
  couleur : blanc
  ennemi : Cpt-Haccab
```

FIG. 5.3 – Exemple de hiérarchie de *frames*.

• **Description différentielle.** La description (ou création) différentielle permet de décrire un nouveau *frame* en exprimant ses différences par rapport à un *frame* existant[4]. Elle met en relation le nouveau *frame* avec celui sur lequel sa description différentielle s'appuie et qui est appelé son prototype ou son *parent*. Cette relation est matérialisée par un lien généralement appelé *est-un*. Nous avons représenté ce lien dans nos exemples par l'intermédiaire d'un attribut supplémentaire[5] également nommé *est-un*. La figure 5.2 montre la définition d'un *frame* représentant `Moby-Dick` qui est comme la baleine précédente à ceci près qu'elle est blanche et que son ennemi est défini plus précisément.

• **Héritage et Hiérarchies de frames.** La relation *est-un* est une relation d'ordre définissant des hiérarchie de frames [Bra83]. Un *frame* hérite de son parent un ensemble d'attributs et on trouve dans les systèmes de frames des hiérarchies d'héritage, comme celle de la figure 5.3, très similaires aux hiérarchies de classes[6], à ceci près que les nœuds de cette hiérarchie représentent des exemples plutôt que des description de concepts. Au sommet de la hiérarchie se trouvent généralement des représentants moyens de concepts (par exemple `Animal`) et dans le bas de la hiérarchie des représentants concrets (par exemple `Moby-Dick`). On trouve des hiérarchies similaires dans les programmes réalisés avec les langages à prototypes.

[4] ¡¡ *The object being used as a basis for comparison (which we call the prototype) provides a perspective from which to view the object being described. (...) It is quite possible (and we believe natural) for an object to be represented in a knowledge system only through a set of such comparisons.* ¿¿ [BW77]

[5] Ce lien est en premier lieu utilisé par le système et n'est pas nécessairement accessible au programmeur via un attribut. Nous avons donc, pour des raisons de simplicité dans notre exposé, introduit, une forme de réflexivité qui pose le problème de la modification éventuelle de l'attribut *est-un*.

[6] Notons que la définition d'une sous-classe est également une description différentielle.

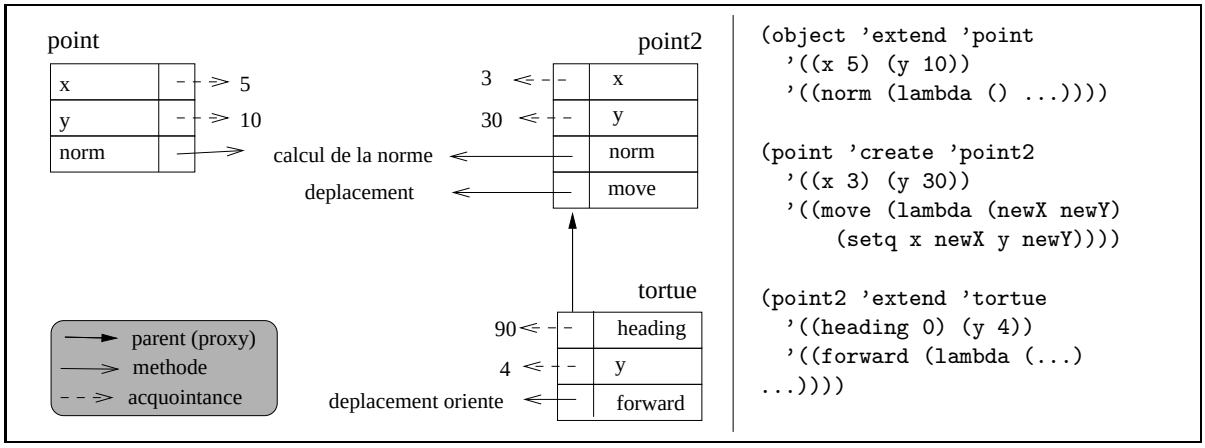


FIG. 5.4 – Clonage et extension en ACT1.

5.3.2 Langages d'acteurs

L'idée de représenter des entités du monde par des objets sans classes a également été appliquée dans le langage ACT1 [Lie81], bien qu'il ne soit fait mention, dans les articles relatifs à ce langage, ni de la notion de prototype ni de l'utilisation qui a pu en être faite dans les langages de frames qui lui sont antérieurs[7], tel que KRL. On trouve cependant dans ACT1 des idées et des mécanismes assez similaires à ceux évoqués précédemment ainsi qu'une part des caractéristiques essentielles des langages à prototypes actuels.

• **Structure d'un acteur.** Les objets dans ACT1 sont appelés ¡¡ acteurs ¿¿, ils possèdent des attributs (appelés *accointances*) et sont dotés de comportements (nous utilisons les termes classiques de ¡¡ méthode ¿¿ pour désigner un comportement et celui de ¡¡ propriété ¿¿ pour désigner indifféremment un attribut ou une méthode). Les méthodes peuvent être invoquées en envoyant des messages aux acteurs[8]. La figure 5.4 montre un acteur[9] appelé `point` possédant deux attributs `x` et `y` et une méthode `norm` calculant la distance de ce point a l'origine.

• **Clonage.** Bien que des primitives de copie d'objets aient existé dans les langages à classes (par exemple en SMALLTALK) antérieurement à ACT1, ce langage a introduit la copie superficielle [10] (primitive `create`), ou *clonage*, comme moyen primitif de création d'objets. Trois primitives `create`, `extend` et `c-extend` permettent de créer de nouveaux acteurs [Bri84]. Ce sont ces trois primitives et la mise en œuvre de la copie différentielle qui nous intéressent ici. Nous en discutons au travers des exemples proposés dans [Bri84]. La primitive `create` permet ainsi de créer `point2` par copie de `point` (Fig. 5.4), de spécifier de nouvelles valeurs de propriétés, par exemple `x` et `y`, et d'en définir de nouvelles, par exemple la méthode `move`.

• **Extensions.** La création par description différentielle en ACT1 est conceptuellement similaire à celle des frames. Elle s'effectue en envoyant à un acteur existant le message `extend`, qui crée un nouvel acteur, que nous appellerons donc ¡¡ extension ¿¿ du premier, lui même appelé en ACT1 le *mandataire* (*proxy* en anglais) du nouvel acteur ; c'est l'équivalent du prototype ou du parent des frames. La figure 5.4 montre la définition de l'acteur nommé `tortue` représentant une tortue *Logo*[11] comme une extension de

[7] Ceci peut être dû en partie au fait qu'il n'est pas évident d'isoler clairement l'utilisation faite des prototypes dans KRL. D'autre part, l'objectif de ACT1, mettre en œuvre un outil de programmation parallèle à base d'objets, est notablement éloigné de celui de KRL.

[8] Cette vision est simplificatrice mais nous suffit ici ; en fait, les comportements d'un acteur sont regroupés au sein d'un *script* et l'invocation peut faire intervenir un mécanisme de filtrage.

[9] Créé par extension d'un acteur pré-défini.

[10] A l'inverse de la copie profonde, la copie superficielle ne copie pas les objets composant l'objet copié.

[11] C'est à dire représentant un robot se déplaçant dans le plan tout en traçant un trait sur son passage.

l'acteur précédent `point2` qui devient son *mandataire*.

• **Héritage et première forme de délégation**. Le lien reliant une extension à son mandataire est tout à fait similaire au lien *est-un* des frames. L'extension peut hériter des propriétés de son parent. L'héritage est mis en œuvre lorsqu'un acteur ne sait pas répondre à un message, auquel cas le système demande à son mandataire de répondre à sa place. Ce passage du contrôle au mandataire est appelé ¡¡ délégation ¿¿[12]. Les articles décrivant ACT1 laissent dans l'ombre un point important de la problématique des langages à prototypes (cf. paragraphe 5.5.1) en ne précisant pas le contexte d'exécution une méthode après qu'il y ait eu une délégation. On trouve quoi qu'il en soit dans ACT1 une première forme de ce qui deviendra la délégation dans les langages à prototype.

• **Copie-extension.** Il est apparu que le lien que le lien de délégation établissait une dépendance forte entre un acteur et ses extensions. Pour permettre la création différentielle d'un nouvel acteur indépendant de son géniteur, une troisième primitive de ACT1, nommée `c-extend`, compose un clonage[13] et une extension du clone.

5.4 Motivations et intérêts de la programmation par prototypes

On trouve dans les systèmes que nous venons de décrire l'essence de ce qui a été appelé programmation par prototypes. Les études relatives à l'introduction d'objets sans classes dans les langages de programmation par objets ont été réalisées au milieu des années 80. Elles visaient à proposer des alternatives au style usuel de programmation par classes utilisé avec SIMULA, SMALLTALK, C++ ou les FLAVORS. Ces études portaient en premier lieu sur la complexité du monde des classes [Bor86c] et les limitations que celles-ci imposent en terme de représentation.

Expérimenter un modèle de programmation par objets s'appuyant sur la théorie des prototypes est apparu comme une possibilité de réduire cette complexité (langages plus simples) et de relâcher les contraintes portant sur les objets. Ce paragraphe illustre les problèmes que posent les classes et montre en quoi les prototypes sont une solution potentielle à ces problèmes.

5.4.1 Description simplifiée des objets

Le processus de raisonnement humain fait souvent passer l'exemple avant l'abstraction [Lie86], l'accumulation d'exemples menant à terme à une généralisation. Or le modèle à classes oblige le programmeur à formaliser un concept, une abstraction, avant de permettre la manipulation de représentants (d'exemples, d'instances) de ce concept. Les langages à prototypes proposent un modèle de programmation plus proche de la démarche cognitive, s'appuyant sur les exemples, attribuée à l'humain face à un problème complexe. Un langage à prototypes permet la description et la manipulation d'objets sans l'obligation préalable d'avoir à décrire leur modèle abstrait.

5.4.2 Modèle de programmation plus simple

Le modèle à classes est complexe [Bor86c, LTP86, Lal89, SLU89], parce que les classes y jouent différents rôles qu'il est parfois difficile de dissocier et qui peuvent rendre leur conception, leur mise en œuvre et leur maintenance difficile. Notons parmi ces rôles : descripteur de la structure des instances, bibliothèque de comportements pour les instances, support de l'encapsulation et support à l'implantation de types abstraits, à l'organisation des programmes, à la modularité, au partage entre descriptions de

[12] ¡¡ *Whenever an actor receives a message he cannot answer immediately on the basis of his own local knowledge (...), he delegates the message to another actor, called his proxy* ¿¿. [Lie81]

[13] Notons que ce clonage peut être complexe : si l'on souhaite par exemple créer une tortue colorée (traçant des traits colorés) comme une ¡¡ c-extension ¿¿ de l'acteur tortue, il faut d'abord cloner `point2` puis cloner `tortue`, lier `clone-tortue` à `clone-point2` par un lien *est-un* et enfin créer le nouvel objet par extension de `clone-tortue`. Cet exemple introduit le problème du clonage d'un objet placé dans une hiérarchie d'objets (cf. paragraphe 5.8.3).

concepts, à la réutilisation. De plus, le même lien entre classes est utilisé pour modéliser différentes relations, entre concepts ou entre types abstraits (suivant la vision que l'on a d'une classe à un instant donné), subtilement différentes les unes des autres [Lal89] : héritage de spécifications, héritage d'implantations, sous-typage ou ressemblance[14]. Enfin, dans un système intégrant des méta-classes (comme SMALLTALK ou CLOS), la classe se voit de plus dotée du rôle d'instance, pouvant recevoir des messages et mener, si l'on peut dire, sa propre vie. La ¡¡ sur-utilisation ¿¿ du même support (la classe) tend à rendre complexe la programmation par objets, et plus encore la réutilisation de programmes existants.

De ce constat est issue l'idée de rechercher d'autres formes d'organisation pour les programmes et d'autres manières de représenter les objets. L'idée initiale de la programmation par prototypes est de réaliser des programmes en ne manipulant qu'une seule sorte d'objets privés du rôle de descripteur. Cette idée suppose que parmi les différents rôles joués par les classes, certains sont soit non indispensables soit modélisables autrement.

5.4.3 Expressivité

Les prototypes ont été utilisés dans de nombreux langages de représentation de connaissances, par exemple dans KRL, pour la souplesse de représentation qu'ils autorisent. L'absence de classes permet de relâcher certaines des contraintes qui pèsent sur leurs instances. Les prototypes autorisent notamment la définition de caractéristiques distinctes pour différents objets d'une même famille conceptuelle, l'expression du partage de valeurs d'attributs entre objets, l'évolution aisée de leur structure ainsi que l'expression de connaissances par défaut, incomplètes ou exceptionnelles. La description de certaines connaissances pose, dans un langage à classes, des problèmes que la programmation par prototypes permet de résoudre. En voici une liste non exhaustive.

Instances différenciées ou exceptionnelles.

Considérons en premier lieu le cas des instances exceptionnelles [DE83, Bor86b, Don89b] ayant des caractéristiques propres que les autres objets de la même famille n'ont pas. La manière standard de représenter une instance exceptionnelle, par exemple `Jumbo` l'éléphant qui sait voler, ou la liste vide dont les méthodes *car* et *cdr* s'implantent différemment de celles des listes non vides, est de créer une nouvelle classe pour les représenter. On trouve, en SMALLTALK par exemple, des classes n'ayant qu'une seule instance : `True`, `False` ou `UndefinedObject`. Même si des alternatives existent pour définir des propriétés au niveau d'un objet, elles sont restées marginales [15].

La représentation d'objets exceptionnels pose évidemment moins de problèmes dans un langage basé sur la description d'individus. On peut par exemple y définir l'éléphant sachant voler par clonage d'un autre éléphant et ajout de la propriété. La représentation de booléens évoquée ci-dessus est tout à fait naturelle dans un langage à prototypes.

Partage de propriétés entre objets. Représentation de points de vues.

Le modèle à classes ne permet pas à différents objets de partager des attributs ou plus précisément d'avoir des attributs communs. Posons le problème de la mise en œuvre de la représentation de diverses facettes d'une même personne personne, par exemple la personne en tant qu'employé ¡¡ employé ¿¿ et la personne en tant que ¡¡ sportif ¿¿. On souhaite évidemment que les facettes soient représentées de façon

[14] Par exemple en SMALLTALK, la classe des ensembles (*Set*) est une sous-classe de la classe des collections non ordonnées quelconques (`Bag`) : ¡¡ *A set is like a bag except that duplicates are not allowed* ¿¿ [Lal89].

[15] L'expression ¡¡ poser problème ¿¿ que nous avons employée ne dénote en effet pas nécessairement une impossibilité de représentation : il est souvent possible d'adapter, dans une implantation donnée, le modèle à classes pour lui faire faire ce que l'on souhaite. Certaines de ces adaptations ne dénaturent pas le modèle mais affectent l'efficacité de la recherche de méthode comme par exemple le qualifieur ¡¡ `eql` ¿¿ de CLOS ; d'autres nécessitent des constructions spéciales et le rendent plus complexe comme les *patterns* d'objets du langage BETA [KMMPN87a] ; d'autres enfin sont intrinsèquement contradictoires avec le modèle (de vraies instances différenciées [SLU89] par exemple) et font que le résultat de l'adaptation engendre d'autres problèmes sémantiques.

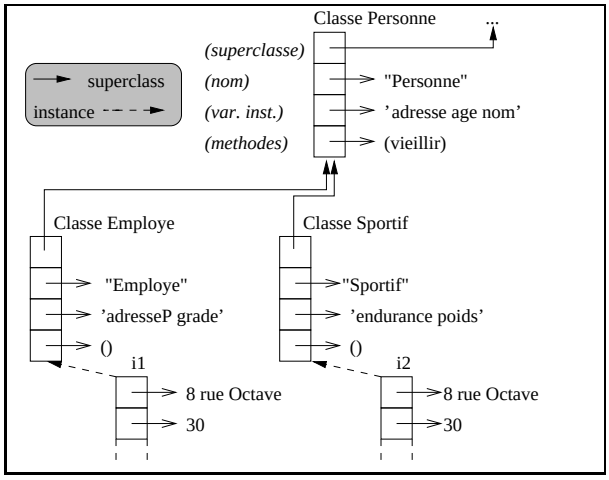


FIG. 5.5 – Deux objets ayant chacun un champs `âge`.

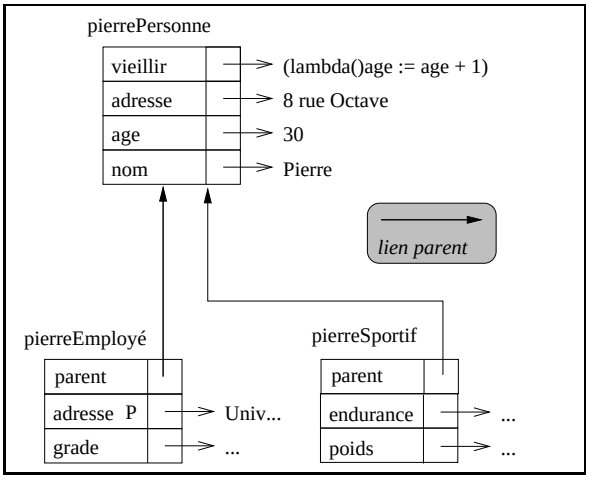


FIG. 5.6 – Deux objets partageant un champs `âge`.

distincte ; il n'y a aucune raison pour que le salaire de l'employé soit visible lorsque l'on s'adresse au sportif. Il n'existe pas de solution standard dans un langage à classes pour exprimer le fait que ces deux objets doivent partager l'attribut âge. Le schéma classique de spécialisation de la classe `Personne` définissant l'attribut `âge`, par deux sous-classes `Employé` et `Sportif` (Fig. 5.5), ne répond pas à la question car deux instances respectives de ces deux sous-classes représentent des entités différentes indépendantes en terme d'état ; si l'âge de l'une change, l'autre n'est pas affecté. D'autres solutions standard à ce problème (utilisant des constructions présentes dans la majorité des langages à classes) sont insatisfaisantes ;

- la composition (redéfinir les classes `Sportif` et `Employé` comme possédant une variable d'instance de type `Personne`) reporte le problème sur l'accès aux attributs et aux méthodes de la personne : comment demander à un sportif son âge ? Il est nécessaire de redéfinir toutes les méthodes de `Personne` sur `Employé` et `Sportif`.
- Une solution consiste à créer une nouvelle classe `EmployéSportif`, sous-classe de `Employé` et de `Sportif`. Le défaut ici est que la hiérarchie résultante peut devenir rapidement inexploitable si l'on souhaite créer de multiple extensions de la classe `Personne` et les combiner.
- Les variables de classes à la SMALLTALK ou leurs équivalents permettent à des instances de partager des valeurs mais la portée de ces variables est trop large ; elle concerne toutes les instances d'une classe.

C'est parce qu'il n'y a pas de bonne solution standard à ce problème que des mécanismes spécifiques de représentations de facettes d'une même entité (on parle de points de vues) ont été développés ; qu'il s'agisse de la ¡¡ multi-instanciation ¿¿ de ROME [Car89], ou du mécanisme de multi-hiérarchies (une par point de vue) avec passerelles de communication développé dans TROPES [MRU90] ou encore de la notion du maintien par par des démons de contraintes entre objets co-référents [FV88].

Dans un monde de prototypes, l'héritage entre objets induit un partage d'attributs et permet de répondre simplement à la question posée (Fig. 5.6). Deux objets représentant la personne vue comme un ¡¡ employé ¿¿ la personne vue comme un sportif ¡¡ sportif ¿¿ peuvent être définis comme des extensions d'un troisième objet détenant l'attribut `âge` ; cet attribut et donc sa valeur étant alors partagés par les trois objets. Notons que dans cette solution nous avons représenté deux points de vue sur une entité (un employé et un sportif) mais nous n'avons nulle part représenté l'entité. Nous revenons sur ce problème dans les sections 5.8.3 et 5.8.4.

Objets incomplets.

La possibilité pour un objet d'hériter les valeurs des attributs d'un autre objet est utilisée intensivement dans les langages de frames pour représenter des objets incomplets *i.e.* des objets dont certaines valeurs d'attributs ne sont pas connues mais dont des valeurs par défaut peuvent être trouvées dans les

```
(setq point (kindof))                  ;Création d'un objet ex nihilo.
(ask point (have 'x 3))                ;Création d'un attribut pour point.
(ask point (have 'y 10))
(defobfun (norm point) ()              ;Une méthode norm pour l'objet point.
  (sqrt (+ (* x x) (* y y))))          ;Les variables sont celles du receveur.
(defobfun (move point) (newx newy)     ;Une méthode avec paramètres,
  (ask self (have 'x (+ x newx))       ;pour additionner deux points.
  (ask self (have 'y (+ y newy)))      ;Modification des valeurs des attributs.
(defobfun (plus apoint) (p)            ;Une méthode d'addition de deux points.
  (let ((newx (+ x (ask p x)))
        (newy (+ y (ask p y)))
        (newp (kindof apoint)))        ;création d'une extension de l'objet
    (ask newp (have 'x newx))          ;passé en argument.
    (ask newp (have 'y newy))
    newp))

(setq point2 (kindof point))           ;point2 est une extension de point,
(ask point2 (have 'y 4)                ;avec un nouvel attribut y.
(setq tortue (kindof point2))          ;Une extension de point2,
(ask tortue (have 'cap 90))            ;avec un nouvel attribut cap,
(defobfun (forward tortue) (dist)      ;et une méthode forward.
  (ask self (move (* dist (cos cap))
           (* dist (sin cap))))
```

OBJECT-LISP *est une extension de Lisp vers les objets autorisant l'envoi de messages ainsi que l'appel fonctionnel classique. L'envoi de message est réalisé par la fonction* `ask` *; son premier argument est le receveur et le second un appel fonctionnel ou le nom de fonction fait figure de sélecteur du message (il doit exister une méthode correspondant à ce nom) ; les arguments de l'appel fonctionnel sont les arguments du message. Les primitives suivantes sont utilisées dans l'exemple :*
– *création d'objets ¡¡ ex nihilo ¿¿ : fonction* `kindof` *sans arguments,*
– *création d'extensions : fonctions* `kindof` *(avec un argument qui est l'objet étendu),*
– *définition de méthodes : fonction* `defobfun`*,*
– *définition d'attributs : méthode* `have`*.*

FIG. 5.7 – Un exemple de langage à prototypes – OBJECT-LISP.

objets dont ils héritent. Par exemple si on cherche ce que mange `Moby-Dick`, on trouvera une valeur dans le *frame* `baleine` dont `Mobby-dick` hérite (cf. Fig. 5.3). Les valeurs héritées peuvent changer au gré de l'évolution des parents d'un objet.

Cette possibilité de manipulation d'objets incomplets ne peut être comparée à la possibilité de spécifier, dans la définition d'une classe, des valeurs initiales pour les différents attributs de ses futures instances. Ces valeurs sont utilisées à l'instanciation, il n'existe ensuite plus aucune relation entre un objet et sa classe pour ce qui concerne les valeurs des attributs. Ne pas avoir d'indirections dans l'accès aux attributs est le gage d'une compilation efficace ; il ne cache pas une impossibilité de représentation liée au modèle à classes[16].

5.5 Premières propositions de langages à prototypes

5.5.1 Langages fondés sur les instances prototypiques et la délégation

Henry Lieberman a repris dans [Lie86, Lie90] certaines idées de ACT1 pour les appliquer à la programmation par objets. Il a proposé un modèle de programmation basé sur les instances prototypiques et

[16] Il est évidemment possible d'implanter une telle relation, ce qui a été fait par exemple dans certains langages de représentation [Rec88]

l'a mis en œuvre ultérieurement dans le langage OBJECT-LISP [All89]. La figure 5.7 propose une version OBJECT-LISP de l'exemple ¡¡ point-tortue ¿¿ présenté au paragraphe 5.3.2. Le premier exemple concret d'un concept (`point`) sert de modèle pour définir les suivants. Les nouveaux objets sont créés par extension avec un équivalent de la primitive `extend` de ACT1. Une extension possède un lien *est-un* vers son prototype. Les objets possèdent des attributs et des méthodes ; ils communiquent en s'envoyant des messages (primitive `ask`). Il n'existe pas de mécanisme d'encapsulation : les variables sont accessibles par envoi de message (`(ask tortue x)`) ou directement dans le corps des méthodes. Le mécanisme de délégation est mis en œuvre aussi bien lors de l'accès à la valeur d'une variable que pour l'activation d'une méthode : si la propriété n'est pas trouvée chez le receveur alors la demande est déléguée à son parent.

Le point véritablement nouveau par rapport à ACT1 est l'explicitation du mécanisme de liaison dynamique. Le mécanisme est conceptuellement parfaitement similaire à celui des langages à classes[17] . Dans notre exemple, l'envoi du message `norm` à `tortue` rend ainsi la valeur `5`, la méthode `norm` est trouvée dans le parent du receveur (`point2`) mais l'accès aux variables `x` et `y` est interprété dans le contexte du receveur initial (`tortue`), ce qui donne, via deux nouvelles délégations, `3` pour `x` et `4` pour `y`.

Un certain nombre de langages sont issus de ce modèle et ont, à la base, les mêmes caractéristiques : citons par exemple SELF, GARNET, NEWTON-SCRIPT ou MOOSTRAP. SELF est certainement le plus connu ; il a donné lieu au plus grand nombre de publications, a bénéficié d'un gros effort de développement et d'une large diffusion.

5.5.2 Langages fondés sur les instances prototypiques et le clonage

A la même époque, [Bor86c] a proposé une description informelle d'un monde d'objets sans classes organisé autour du clonage. Un prototype y représente un exemple standard d'instance et les nouveaux objets sont produits par copies et modifications de prototypes. Une fois la copie effectuée, aucune relation n'est maintenue entre le prototype copié et son clone. Conscient toutefois de la pauvreté du modèle ainsi obtenu, Borning proposait de l'étendre en instaurant une certaine forme d'héritage à base de contraintes [Bor81]. Ce modèle n'a pas été développé par son auteur mais a inspiré les langages à prototypes basés sur le clonage comme KEVO [Tai93], OMEGA [Bla94] ou OBLIQ [Car95].

5.5.3 Langages intégrant hiérarchies de classes et hiérarchies d'objets.

Le problème a également été abordé plus directement sous l'angle de l'organisation des programmes : pour contourner les limitations que nous avons évoquées, a été imaginé [LTP86, Lal89], à la même époque et parallèlement à celui de Lieberman, un modèle de programmation intégrant des classes et des instances (appelés cette fois *exemplars*, ce que l'on peut traduire par ¡¡ exemplaires ¿¿) dotées d'une certaine autonomie. La principale raison d'être de cette proposition était d'expérimenter un découplage entre une hiérarchie de sous-typage, composée de classes et une hiérarchie de réutilisation d'implantations, composée d'instances.

Cette proposition pose un certain nombre de problèmes qu'il serait trop long de décrire ici ; les auteurs l'ont d'ailleurs abandonnée, n'ayant pas réussi à faire la synthèse entre le rôle des classes et l'héritage entre instances. Cette étude reste cependant intéressante. D'une part, l'héritage entre instances est tout à fait similaire à celui proposé par Lieberman et a donc inspiré au même titre les langages à prototypes ultérieurs. D'autre part, elle proposait une séparation des interfaces et des implantations qui n'est pas sans rappeler celle de JAVA. Enfin, c'était une première tentative d'utilisation de la délégation dans un monde de classes, idée que nous suggérons de reprendre aujourd'hui sous d'autres formes (cf. conclusion).

[17] Il permet au code d'une méthode d'être interprété dans le contexte des attributs et des méthodes du receveur initial du message, et ce, quel que soit l'endroit où la méthode a été trouvée. Nous supposons ce mécanisme, ainsi que ses applications à l'écriture de méthodes polymorphes, connu du lecteur.

5.5.4 Langages de frames pour la programmation

Certains langages, dits hybrides [MNC+89], autorisent la définition de méthodes dans un monde dédié à la représentation et fondé sur les frames comme YAFOOL [Duc91]. On peut donc les assimiler aux langages à prototypes. L'originalité de ces langages par rapport à ceux qui se voulaient uniquement basés sur les instances prototypiques (comme SELF ou OBJECT-LISP) est qu'ils permettent de définir des hiérarchies intégrant des représentants moyens (comme `Animal`) comme celle de la figure 5.3. Nous verrons que les concepteurs de la plupart des langages à prototypes ont dû réintroduire cette possibilité.

5.6 Récapitulatif des concepts et mécanismes primitifs

Voici une synthèse des propositions précédentes qui permet d'isoler les concepts fondamentaux des langages à prototypes.

• **Objets sans classes**. La caractéristique commune à tous les objets des langages à prototypes est de ne pas être liés à une classe, de ne pas avoir de descripteur. L'objectif de ne manipuler que des objets concrets, affiché par SELF par exemple, n'est pas généralisé, nous avons vu qu'il existait des langages permettant de manipuler des représentants moyens. Certains auteurs parlent également d'objets autonomes mais cette caractéristique n'est pas générale en programmation par prototypes ; les objets ne sont pas non plus véritablement autonomes dès lors qu'ils sont créés comme des extensions d'autres objets.

• **État et comportements**. Les objets sont définis par un ensemble de propriétés. Une propriété est à la base un couple ¡¡ nom[18]– valeur[19] ¿¿. Les propriétés sont soit des attributs, auquel cas la valeur est un objet quelconque, soit des méthodes, la valeur est alors une fonction.

• **Envoi de messages**. Les objets communiquent par envoi de messages ; ils sont capables de répondre aux messages en appliquant un de leurs comportements (ou éventuellement en rendant la valeur d'un de leurs attributs). Ils peuvent être vus comme des frames sans facettes dotés de procédures et répondant à des messages comme les acteurs ACT1.

• **Trois formes primitives de création d'objets**. Les objets peuvent être créés soit *ex nihilo*, soit en copiant un objet existant (clonage), et dans certains langages en étendant un objet existant (extension ou création différentielle).

• **Héritage**. Dans le cas de la création différentielle, un nouvel objet est créé comme une extension d'un objet existant, qui devient son parent. La relation ¡¡ est-extension-de ¿¿ lie le nouvel objet et son parent. Il s'agit d'une relation d'ordre qui définit des hiérarchies d'objets. Dans une hiérarchie, une extension hérite les propriétés de son parent qui n'ont pas été redéfinies à son niveau. Si une propriété héritée est un attribut alors l'extension possède cet attribut, si c'est une méthode alors elle lui est applicable. La relation est matérialisée par un lien appelé lien ¡¡ parent ¿¿ ou ¡¡ lien de délégation ¿¿. Ce lien est parfois accessible au programmeur ; en SELF par exemple, chaque objet possède au moins un attribut nommé `parent` contenant l'adresse de son parent.

• **Délégation**. La délégation est le nom donné au mécanisme qui met en œuvre l'héritage, c'est-à-dire qui cherche et active une propriété héritée. La délégation est généralement implicite[20] [SLU89, DMC92]. Dans ce cas, le terme ¡¡ déléguer ¿¿ est une image ; dans la pratique, le système, lorsqu'il ne trouve pas de

[18] Une propriété est unique dans le système mais plusieurs propriétés peuvent avoir le même nom (c'est ce que nous appelons ¡¡ surcharge ¿¿).

[19] Une propriété peut également posséder un type, un domaine, une signature, des facettes, etc.

[20] Dans l'autre alternative, la délégation explicite, l'objet dispose d'un module de réception des messages et choisit lui-même la propriété à activer ou délègue lui-même le message à un autre objet. La délégation explicite est citée dans certains articles [SLU89] et est utilisée dans ACT1 mais nous ne connaissons pas de langages à prototypes qui l'utilisent.

propriété dans le receveur du message, la recherche dans ses parents successifs et s'il la trouve, l'active[21]

• **Liaison dynamique**. Les langages à prototypes sont des langages où les schémas usuels de réutilisation des langages à objets s'appliquent. Ainsi, l'activation d'une propriété s'effectue toujours dans le contexte du receveur initial du message. Dans toute méthode, la variable `self` (ou un équivalent) désigne l'objet qui a effectivement reçu le message et non celui à qui on l'a délégué (*i.e.* celui où la méthode à été trouvée). L'accès à un attribut ou l'envoi d'un message à `self` nécessitent donc un mécanisme de liaison dynamique.

Nous avons à ce point de l'exposé une idée générale de ce que sont les langages à prototypes et les possibilités nouvelles qu'ils offrent. Les langages existants proposent néanmoins un ensemble de variations subtiles autour des concepts que nous avons présenté. Ces variations résultent d'une part d'interprétations différentes données aux concepts précédents (par exemple à celui d'extension) et d'autre part à la nécessité, pour les concepteurs, de résoudre des problèmes non envisagés initialement (comme celui de la gestion de propriétés communes à des ensembles d'objets).

5.7 Caractérisation et interprétation des mécanismes

Les points, sources de confusions, qui réclament plus particulièrement des précisions sont : la caractérisation de la différence entre clonage et création différentielle, la caractérisation de la différence entre l'héritage dans les langages à classes et dans les langages à prototypes, et enfin la compréhension des diverses utilisations possibles du concept d' ¡¡ extension ¿¿. Nos caractérisations sont fondées sur l'héritage et le partage, ce qui est partagé ou hérité est relatif aux propriétés. On distingue trois formes de partage[22].

- Il y a partage de noms entre deux objets lorsque qu'ils ont chacun une propriété de même nom.
- Il y a partage de valeurs lorsque deux objets `o1` et `o2` ont chacun une propriété de nom x et que la valeur de la propriété `x` de o_1 et de o_2 est la même (égalité physique).
- Il y a partage de propriétés lorsque deux objets possèdent la même propriété (même adresse et donc même nom et même valeur). Il peut y avoir partage de valeurs sans qu'il y ait partage de propriétés.

5.7.1 Distinction entre clonage et extension

La distinction entre clonage et extension, montrant que ces deux mécanismes ne sont pas redondants, provient de ce que le clonage et la création différentielle induisent deux formes de partage distinctes [DMC92].

• **Une caractérisation du clonage**. Tout objet cloné, partage avec son clone, au moment où celui-ci est créé, les noms et les valeurs de ses propriétés. Le clone et son original évoluent indépendamment, la modification d'un attribut du premier n'est pas répercutée sur le second : le partage est ponctuel.

• **Caractérisation du mécanisme d'extension**. Un objet dans une hiérarchie d'objets hérite de ses parents un ensemble d'attributs et de méthodes. Tout parent partage avec ses extensions les propriétés [23] qu'il définit et que ces derniers n'ont pas redéfinies. Les propriétés du parent non redéfinies dans une extension sont aussi des propriétés de l'extension. L'extension est dépendante de son parent[24]. Cette dépendance et ce partage sont persistants, ils durent aussi longtemps qu'existe le lien entre les deux

[21] Le lecteur trouvera dans [Mal95] une description formelle de la sémantique du mécanisme de délégation pour un langage à la Lieberman et dans la plate-forme *Prototalk* [Don97] une mise en œuvre d'évaluateurs correspondants qu'il pourra étudier, étendre et modifier à sa guise.

[22] Une description plus précise en est donnée dans [BD96] (cf. chapitre 10, article 2)

[23] Ce qui est hérité étant identique pour les attributs et les méthodes, il est possible d'unifier les deux sortes de propriétés ; c'est ce qu'ont fait les concepteur de SELF. On ne trouve en SELF que des *slots* dont les valeurs peuvent éventuellement être des fonctions exécutables, on y accède dans tous les cas par envoi de message ; lorsque la valeur d'un slot est une fonction alors elle est exécutée.

[24] La réciproque, à savoir l'indépendance du parent vis-à-vis de ses extensions sera discutée au paragraphe 5.7.3.

objets.

Les deux mécanismes induisent donc des partages distincts avec des durées de vie différentes (ponctuel pour le clonage, persistant pour l'extension). Leurs applications sont distinctes.

5.7.2 Caractérisation de la différence entre hiérarchies d'objets et hiérarchies de classes

L'étude comparative de l'héritage dans les hiérarchies de classes et dans les hiérarchies d'objets (hiérarchies de délégation) a fait l'objet de plusieurs travaux. D'après Lieberman, la délégation est un mécanisme plus général que l'héritage de classes, elle permet de le simuler. Dans [Ste87], il est montré que la simulation inverse est possible à condition d'utiliser les classes comme des objets et de se servir des variables de classe à la SMALLTALK pour représenter les propriétés, ce qui est un cas très particulier. Cet article ne montre par ailleurs pas du tout ce que son titre laisse supposer, à savoir que la délégation est la même chose que l'héritage dans les hiérarchies de classes.

En fait les deux formes d'héritages sont distinctes car l'héritage dans les langages à classes n'induit aucun partage d'attributs entre instances[25]. Un objet, instance d'une classe `C` possède, via sa classe : d'une part un ensemble de méthodes (déclarées et définies dans les super-classes de `C`) et d'autre part un ensemble de noms d'attribut (déclarés dans les super-classes de `C`) dont il détient en propre la valeur. Deux objets dont les classes sont liés par un lien ¡¡ sous-classe-de ¿¿ partagent donc des méthodes mais uniquement des noms d'attribut. Ces objets sont indépendants en terme d'états.

Les deux formes d'héritage ne sont donc pas équivalentes. L'héritage d'attributs entre objets est caractéristique des hiérarchies d'objets ; il est à l'origine des possibilités nouvelles de représentation offertes par les langages à prototypes (cf. paragraphe 5.4), mais aussi de problèmes nouveaux.

5.7.3 Deux interprétations du lien de délégation

L'héritage d'attributs induit un partage qui rend une extension dépendante de son parent, mais qu'en est-il de la réciproque ? Un parent est-il dépendant de ses extensions ou en d'autres termes, une extension peut-elle, en se modifiant, modifier aussi son parent ? Le problème se pose lorsque l'on demande à un objet (par envoi de message ou par un autre moyen) de modifier la valeur d'un attribut qu'il hérite. Déléguer ou ne pas déléguer les accès en écriture aux attributs, telle est la question. Considérons par exemple l'envoi à `tortue` du message `move`, qui provoque l'activation de la méthode `move` détenue par `point2`, laquelle accède en écriture aux attributs `x` et `y` du receveur initial (liaison dynamique), ce dernier (`tortue`) ne détenant en propre que l'attribut `y`. Comment interpréter l'affectation ¡¡ `x := newX` ¿¿ ? La réponse à cette question dépend de l'interprétation que l'on a de la notion d'extension et des possibilités que l'on veut offrir. Les langages à prototypes divergent sur ce point.

• **Interprétation No 1**. Des langages tels que YAFOOL ou GARNET ne délèguent pas l'accès en écriture aux attributs. L'affectation est alors réalisée dans le contexte strict du receveur initial : lorsque celui-ci ne possède pas l'attribut considéré, cet attribut doit être créé avant la réalisation de l'affectation proprement dite. Dans notre exemple, l'affectation de la variable `x` sera précédée par une redéfinition automatique de la propriété `x` sur `tortue`.

Cette solution limite le partage d'attributs entre objets à du partage de valeurs : le parent partage avec ses extensions uniquement le nom et la valeur de ses propriétés non redéfinies. Elle rend le parent complètement indépendant de ses extensions[26].

Dans ce contexte, une extension représente systématiquement une entité différente de celle représentée par le parent, duquel elle hérite néanmoins certaines caractéristiques. Cette interprétation est ainsi adaptée à la mise en œuvre de l'exemple ¡¡ point-tortue ¿¿ dans lequel deux entités différentes, représentées par les objets `point2` et `tortue` ont la même abscisse. Mais en restreignant le partage d'attributs, cette

[25] Sauf pour le cas très particulier des variables de classe.

[26] On n'obtient pas pour autant un équivalent du clonage car l'état de l'extension est toujours dépendant du parent.

solution interdit également certaines utilisations de la délégation telle que celle utilisée dans l'exemple ¡¡ personne, employé, sportif ¿¿[27] (cf. Fig. 5.6).

• **Interprétation No 2**. Des langages tels que SELF ou OBJECT-LISP délèguent l'accès en écriture aux attributs. L'affectation est alors toujours réalisée, quel que soit le receveur initial, au niveau de l'objet détenant l'attribut, dans notre exemple il s'agit de l'objet `point2`. Dans ce contexte, un parent est dépendant de ses extensions. Une extension représente la même entité que son parent, elle en décrit une partie spécifique. Cette interprétation permet ainsi de représenter l'exemple ¡¡ personne-employé-sportif ¿¿, dans lequel les extensions représentent des parties d'un tout, ce tout étant la représentation d'une personne.

On pourrait croire en première analyse qu'il est néanmoins possible de se ramener à la première interprétation à condition de redéfinir sur une extension tous les attributs définis par ses parents. En fait, utiliser cette seconde interprétation pose, si l'on souhaite par exemple représenter le point et la tortue, un ensemble de problèmes que nous nous proposons d'aborder maintenant.

5.8 Discussion des problèmes relatifs à l'identité des objets.

Les premiers problèmes que pose la programmation par prototypes sont relatifs à l'identité des objets. Nous utiliserons ici le terme ¡¡ identité ¿¿ pour désigner l'entité (ou les entités) du domaine qu'un objet représente. Avec le modèle classe-instance, un objet a une identité unique, il représente une et une seule entité du domaine. L'objet y est par ailleurs une unité d'encapsulation autonome détenant l'ensemble des valeurs des attributs ; toute modification de la valeur d'un de ces attributs est sous son contrôle. Avec l'héritage entre objets, cette bijection entre entités décrites et objets peut ne plus exister. En effet, dans une hiérarchie, un même objet peut représenter à la fois une entité, plusieurs entités ou des parties de plusieurs autres entités. Par exemple, l'objet `point2` de la figure 5.4 représente un point, mais également une partie d'une tortue puisqu'il détient son abscisse. Les représentations des entités point et tortue partagent les propriétés définies dans l'objet `point2`. Dans l'autre exemple, l'entité personne est représentée par les trois objets `personne`, `employé` et `sportif`.

Dès lors qu'un objet définit des propriétés représentant plusieurs entités, se pose le problème de la modification accidentelle d'une entité suite à la modification d'une propriété partagée.

5.8.1 Problèmes potentiels d'intégrité

Le premier problème potentiel est la modification d'une extension par l'intermédiaire de son parent. Il se pose avec les deux interprétations de la notion d'extension. Par exemple, l'envoi à l'objet `point2` du message `move` provoque la modifications d'attributs de `point2` et subséquemment des entités point et tortue, car l'attribut `x` est partagé[28]. Ce résultat peut néanmoins être considéré comme une conséquence naturelle de l'utilisation de la description différentielle. Si le programmeur de `tortue` ne souhaite pas que cet objet dépende de `point2`, il peut utiliser le clonage. Il y a bien modification indirecte d'une entité mais elle correspond à l'intention du programmeur.

Le second problème potentiel est la modification accidentelle (non prévue par le programmeur) d'un parent via une de ses extensions. Ce problème ne se pose qu'avec la seconde interprétation de la notion d'extension utilisée pour représenter des entités différentes. L'exemple en est l'envoi du message `move` à `tortue` que nous avons décrit et qui modifie l'objet `point2`, ce qui évidemment ne correspond pas nécessairement à l'intention du programmeur. Dans cette configuration, le lien de délégation octroie, à `tortue` un accès en lecture et en écriture aux propriétés définies dans `point2`. Demander à la tortue de se déplacer entraîne également le déplacement du point.

[27] En effet, l'envoi d'un message à `Sportif` pour modifier son adresse, résulterait alors en une redéfinition de `adresse` dans `Employé` et non en une modification de `adresse` au niveau de `Personne`.

[28] La distinction entre objet et entité apparaît clairement ici, la tortue a bien été modifiée alors que l'objet `tortue` ne l'a pas été.

Il est ainsi possible de modifier plusieurs entités en pensant n'en modifier qu'une. Plus généralement, il est impossible de placer une frontière nette entre des entités représentées par des objets appartenant à une même composante connexe d'une hiérarchie. Ces composantes devenant en pratique les réelles unités d'encapsulation des langages à prototypes ([CUCH91] emploie le terme d'*encapsulation de modules*). Une affectation peut entraîner la modification d'un très grand nombre d'entités sans qu'il soit aisé de prédire lesquelles ou même leur nombre. Briser l'encapsulation dans un tel contexte devient extrêmement simple. Il suffit, pour accéder en lecture et en écriture aux attributs d'un objet `O`, d'en créer une extension `E`, d'y définir une méthode réalisant un accès en écriture aux attributs définis sur `O` et d'envoyer le messages correspondant à `E`. Les variables d'un objet deviennent de fait des variables semi-globales, modifiables par n'importe lequel de ses descendants.

5.8.2 Solutions aux problèmes d'intégrité

Toutes les solutions proposées au problème précédent passent par de la limitation du partage d'attributs.

• **Responsabilité du programmeur.** Des langages comme SELF ne proposent aucune solution à ce problème. La solution standard pour le programmeur est de créer des extensions en redéfinissant systématiquement tous les attributs de ses parents et en n'héritant que les méthodes. Même avec cette précaution, il est impossible, comme nous l'avons montré, d'assurer l'encapsulation.

• **Restriction du partage d'attributs.** En restreignant le partage d'attributs à du partage de valeurs (cf. paragraphe 5.7.3), les langages comme YAFOOL ou GARNET solutionnent le problème au détriment du pouvoir d'expression du langage.

• **Distinction entre liens de délégation.** Offrir en standard les deux possibilités est tentant, une solution mixte a ainsi été implantée dans le langage *NewtonScript* [Smi95] où le programmeur a le choix entre deux sortes de liens de délégation. A l'un de ces liens (lien `_proto`) est associé du partage de valeurs et une sémantique de valeurs par défaut, tandis que du partage d'attributs est associé à l'autre (lien `_parent`). L'existence de deux types de liens de délégation complique cependant considérablement le modèle de programmation, la lisibilité des programmes et la recherche de sélecteurs. En effet, bien que le lien `_proto` soit prioritaire au lien `_parent`, il est difficile de prévoir ce qui peut se passer lorsque le sélecteur recherché est accessible en suivant deux chemins différents (incluant éventuellement des liens des deux types).

• **Contrôle des extensions.** Le langage AGORA [Ste94] permet à chaque objet de contrôler la création de ses futures extensions. Il est impossible d'étendre un objet, s'il ne possède pas de méthodes particulières, appelées ¡¡ *mixin-methods* ¿¿ permettant de spécifier de façon précise les droits d'accès en lecture et en écriture aux propriétés qui seront octroyés à ses descendants. Cette solution rend au programmeur le contrôle total des accès aux propriétés d'un parent, son principal inconvénient réside dans le fait que celui-ci doit systématiquement prévoir toutes les possibilités d'extension, ce qui exclut toute réutilisation non anticipée ; le problème est similaire à celui du choix de la ¡¡ virtualité ¿¿ des méthodes en C++.

• **Cas des langages excluant la création d'extensions.** Tous les langages à prototypes n'incluent pas le mécanisme de délégation. Les problèmes évoqués étant directement liés au partage de propriétés qu'il induit, il va de soi que ces problèmes ne se posent pas dans ces langages (comme KEVO), qui ont en contrepartie un pouvoir d'expression plus limité.

5.8.3 Problèmes de gestion des entités morcelées

Le partage d'attributs crée un autre problème, connexe aux précédents. Lorsqu'une entité est représentée par plusieurs objets, nous parlons alors d' ¡¡ entité morcelée ¿¿ [DMC92, Mal96, BD96] ; c'est par exemple le cas de l'entité tortue, dont la représentation utilise les objets `point2` et *tortue*. Le problème est qu'il n'existe aucun objet du langage représentant l'entité tortue dans sa globalité. On pourrait considérer que

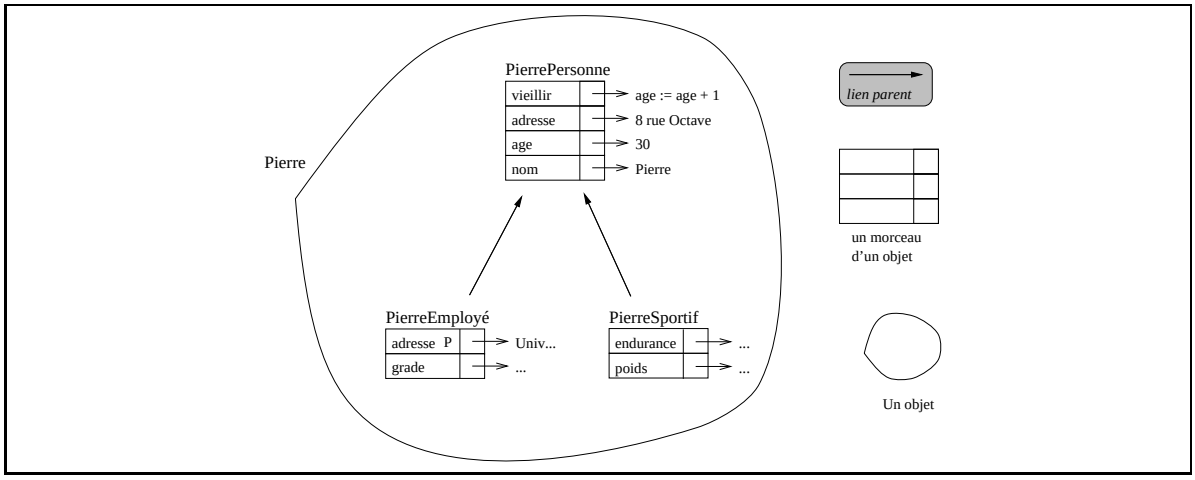


FIG. 5.8 – Objets morcelés.

l'objet `tortue` joue ce rôle mais ceci lui confère un double statut, celui de représentant de tortue et en même temps celui de représentant d'une de ses parties. Ce double statut se manifeste si l'on demande à l'objet `tortue` de se cloner, demande-t-on un clonage de l'objet ou un clonage de l'entité ? Dans le premier cas, seul l'objet doit être copié. Dans le second cas, il est nécessaire de réaliser une copie de toutes les parties de la tortue, c'est-à-dire une copie des objets `tortue` et `point2`, similaire à celle réalisée par la primitive `c-extent` de ACT1. Le double statut se manifeste dans cet exemple par l'existence de deux primitives de clonage, ayant des noms différents, entre lesquelles le programmeur doit choisir et qui sont sources de confusion. Le problème se révèle encore mieux sur un exemple plus complexe, par exemple celui de la personne ¡¡ pierre ¿¿ (cf. Fig. 5.6) représentée par les objets `pierrePersonne`, `pierreEmployé` et `pierreSportif`. Il n'existe plus dans ce cas aucun objet susceptible de représenter l'entité ¡¡ pierre ¿¿ dans sa globalité. Si on souhaite la cloner, aucune primitive du langage n'est capable de réaliser cette opération puisque cette entité n'est pas réifiée ; un tel clonage doit être réalisé de façon *ad hoc* par le programmeur. On ne trouve aucune solution à ce problème dans les langages existants.

5.8.4 Utilisation sémantiquement fondées de la délégation

Nous pouvons à cet instant faire un point sur les utilisations sémantiquement fondées du mécanisme de délégation. Tous les problèmes que nous venons d'évoquer viennent du fait que des entités du domaine sont représentées par plusieurs objets partageant des attributs. Les utilisations cohérentes de la délégation sont celles ou une bijection est rétablie entre objets du langages et entités représentées.

Une première solution à ce problème a été décrite, elles consiste à restreindre le partage de propriétés à du partage de valeur et correspond à notre interprétation No 1 de la délégation (cf. section 5.7.3.

Une seconde solution consiste à utiliser la délégation pour réaliser un partage de propriétés, non plus entre objets mais entre morceaux formant les différentes parties d'un objet, l'objet lui-même représentant une entité du domaine. On peut On peut ainsi imaginer l'objet `Pierre` représentant une personne, formé de 3 ¡¡morceaux¿¿ comme dans la figure 5.8.

Une application évidente des objets morcelés est la représentation de points de vue [Car89, Fer89a] de l'entité correspondante ; mais il peut y en avoir d'autres. Des études ont été menées pour intégrer une représentation explicite d'entités morcelés d'abord dans un langage à prototypes [BD95] puis dans un langage à classes. Une première vision de cette idée est présentés dans [BD96], l'idée est développée dans la thèse de Daniel Bardou. Pour plus de précisions, nous invitons le lecteur à consulter le chapitre 10, article 2.

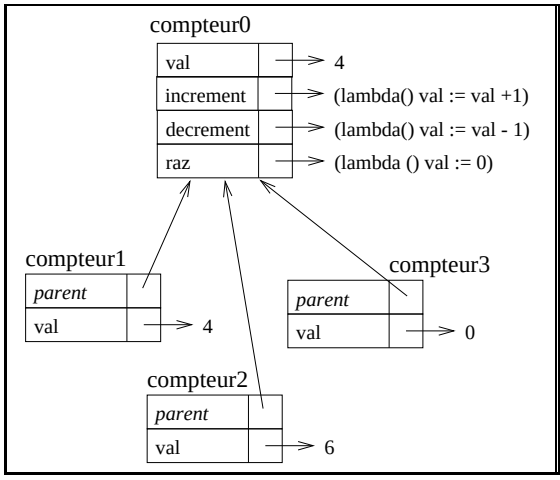


FIG. 5.9 – Solution des instances prototypiques.

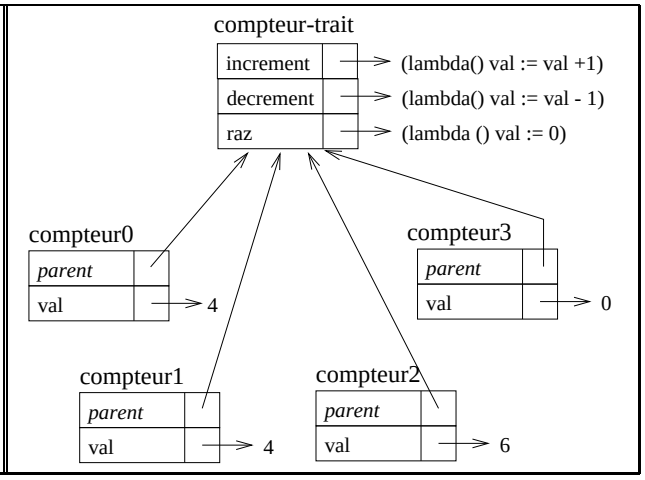


FIG. 5.10 – Solution des `traits`.

5.9 Discussion des problèmes liés à l'organisation des programmes

La seconde série de problèmes de la programmation par prototypes est liée à la disparition de la représentation en compréhension des concepts. Confrontés à la réalisation de programmes complexes, les utilisateurs de langages à prototypes ont vite été limités par l'absence d'équivalents aux possibilités de partage offertes par les classes.

5.9.1 Problème du partage entre membres d'une famille de clones

Nous employons le terme de ¡¡ famille de clones ¿¿ pour désigner l'ensemble des objets mis en relation par la fermeture transitive de la relation ¡¡ est-un-clone-de ¿¿ qui lie conceptuellement un clone et son modèle. On peut assimiler une famille de clones à l'ensemble de tous les objets appartenant conceptuellement à un même type de données ou à l'ensemble des instances d'une classe.

Le premier problème est celui du partage des méthodes communes à tous les objets d'une famille de clones. Considérons par exemple un ensemble de clones de l'objet `point` de la figure 5.4, ces objets sont tous indépendants et il n'existe aucun objet représentant la famille. Considérons maintenant le problème suivant : comment doter tous les membres de la famille d'une nouvelle méthode (par exemple `moveToOrigin`). Diverses solutions ont été proposées pour introduire cette possibilité dans les langages à prototypes. Les premières sont de la responsabilité du programmeur et utilisent les constructions primitives existantes ; il s'agit des approches par instances prototypiques, représentants moyens ou *traits*. Les secondes déplacent le problème au niveau de l'implantation et proposent une gestion automatique du partage entre objets d'une même famille.

Gestion des familles de clones utilisant des prototypes

La méthode dite des ¡¡ instances prototypiques ¿¿, initialement proposée dans [Lie86], consiste à élever un des membres au statut de représentant de la famille. Dans la pratique l'instance prototypique devient le parent de tous les autres membres. La figure 5.9 montre un exemple d'instance prototypique (`compteur0`) pour une famille de ¡¡ compteurs ¿¿. Doter tous les membres de la famille d'une nouvelle propriété s'effectue alors simplement en la définissant sur cet objet. Cette solution, bien qu'utilisable dans la pratique, est peu satisfaisante car l'instance prototypique peut être considérée tantôt comme un individu, tantôt comme l'ensemble des individus qu'elle représente. Le statut particulier de représentant de la famille n'est en rien matérialisé ; rien ne distingue l'instance prototypique des autres objets. Pourtant les autres membres de la famille hériteront de toutes ses évolutions. L'évolution personnelle de l'instance

prototypique peut devenir contradictoire avec son statut de représentant si cette évolution fait qu'il n'est plus représentatif. Imaginons par exemple un ministre représentant de ses congénères ayant des démêlés avec la justice, caractéristique nouvelle que les autres ne souhaiterons pas hériter.

Pour la plupart des langages, on a abandonné l'idée de ne manipuler que des objets représentant des entités concrètes. La méthode des ¡¡ représentants moyens ¿¿ repose sur le même principe que celle des instances prototypiques mais un représentant moyen (cf. paragraphe 5.2) est choisi comme représentant de la famille, il est éventuellement doté d'attributs et éventuellement incomplet.

La méthode des `traits`, proposée par SELF [UCCH91], proche de la précédente consiste à créer des objets, appelés `traits`, ne contenant que les méthodes partagées par les objets de la famille (la figure 5.10 en montre un exemple). La méthodologie des `traits` suggère de diviser la représentation d'une nouvelle sorte d'entités en deux parties : un objet `trait` qui contient les méthodes factorisées et un prototype contenant les attributs et ayant le `trait` pour parent. Obtenir un nouvel objet de la famille de clones consiste alors à cloner le prototype mais pas le `trait`.

Les représentants moyens et les `traits` ont le même double statut que les instances prototypiques. Le `trait` est une bibliothèque de propriétés, mais il a aussi le statut d'objet standard, car rien dans le langage ne le distingue des autres objets. En particulier, il est possible d'envoyer des messages directement au `trait` afin d'invoquer les propriétés qu'il détient, et cela pose problème lorsque celles-ci contiennent des références à des variables. Par exemple, si on envoie le message `incr` à `compteur-trait`, l'accès à la variable `val` lèvera une exception puisque `compteur-trait` ne possède pas cette propriété. Un `trait` détient des propriétés qui lui sont applicables (on peut les activer par envoi de message puiqu'il les détient) mais pratiquement inapplicables (elles sont prévues pour être appliquées à ses extensions). Avec les représentants moyens, dans une moindre mesure, il peut se poser le même problème qu'avec les `traits` : un représentant moyen peut très bien être incomplet, c'est-à-dire détenir des méthodes faisant référence à des variables qu'ils ne possèdent pas, ou dont la valeur n'est pas définie (parce qu'il n'y a pas de valeur moyenne pour cette variable, par exemple).

Gestion automatique des familles de clones.

Une seconde approche est l'automatisation. Dans le langage KEVO, tout objet appartient à une famille de clones automatiquement gérée par le système. Si une méthode est ajoutée ou retirée à un objet, celui-ci change de famille. SELF propose une construction appelée `map`, elle aussi gérée automatiquement et invisible au programmeur. Un `map` détient les méthodes d'un objet ainsi que pour chaque attribut, son nom et l'indice auquel la valeur est rangée dans l'objet. Les `maps` permettent d'obtenir une implantation mémoire des objets identique à celle des instances d'une classe ; ils réduisent l'espace mémoire occupé par chaque objet dans lesquels seules les valeurs des attributs sont stockées (Fig. 5.11, partie droite). Un nouveau `map` est créé à chaque création *ex nihilo* d'un nouvel objet et tous les membres d'une famille de clone dont la structure n'a pas été modifiée partagent le même `map`.

Ni les familles de clones automatisées de KEVO, ni les *maps* de SELF ne présentent l'inconvénient de conférer un double statut à des objets ; des constructions spécifiques sont utilisées pour assurer la factorisation. Toutefois, le programmeur n'a aucun contrôle sur ces constructions. Il ne peut spécifier quels objets doivent appartenir à la même famille, ni déterminer quels objets sont effectivement considérés par le système comme lui appartenant. Avec KEVO, lorsque l'on désire ajouter une propriété à toute une famille de clones, il suffit de choisir un objet membre de cette famille et d'invoquer une primitive d'ajout collectif, sans savoir avec précision quels objets vont être modifiés. En SELF, aucune primitive ne permet d'ajouter une propriété à tous les objets rattachés au même `map`.

Conclusion

La gestion des familles d'objets par des prototypes pose des problèmes conceptuels. Les trois solutions ont à la base le même défaut qui est d'utiliser des prototypes pour représenter des abstractions (les familles de clones), donc de réintroduire des formes d'abstraction sans support adéquat. La gestion automatique des familles de clones est insatisfaisante, elle ne fait que masquer le problème fondamental de l'absence d'objets du langage capables de représenter sans ambiguïtés des collections d'objets ou d'autres

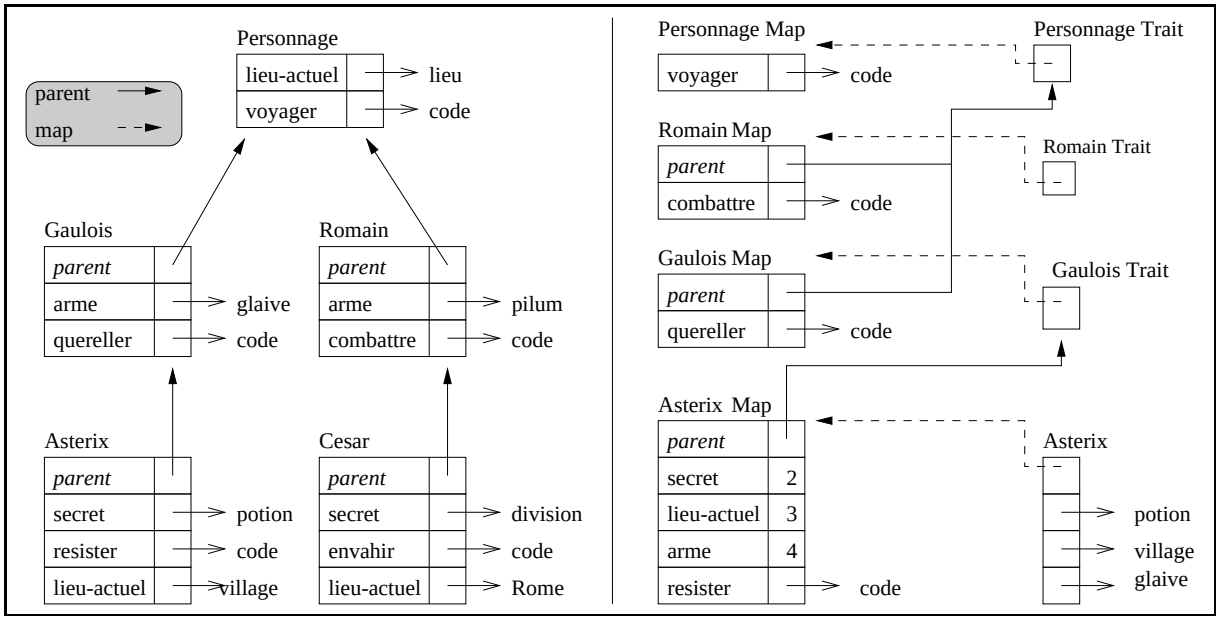


FIG. 5.11 – Hiérarchies réalisées avec des représentants moyens d'une part et des `traits` et des `maps` à la SELF d'autre part.

abstractions.

5.9.2 Problème du partage entre familles de clones

Une dernière forme de partage essentielle pour l'organisation des programmes a dû être introduite dans les langages à prototypes : il s'agit du partage entre différentes familles de clones. Si on prend l'exemple d'un programme représentant des ¡¡ gaulois ¿¿ et des ¡¡ romains ¿¿, le problème est de savoir sur quel objet définir les propriétés communes à ces deux familles d'entités comme par exemple la méthode `voyager`. Dans le monde des classes, ce type de partage est réalisé par des super-classes, souvent abstraites, communes à plusieurs classes. Dans notre exemple la méthode `voyager` serait définie sur une classe `Personnage` super-classe de `Romain` et `Gaulois`. Dans les langages à prototypes, il a été mis en œuvre à l'aide des constructions que nous venons de décrire, instances prototypiques, représentants moyens, `traits` ou `maps`.

La partie gauche de la figure 5.11 [29] montre une hiérarchie de représentants moyens permettant de répondre au problème posé. Dans cet exemple, un représentant moyen du concept personnage détient les propriétés `lieu-actuel` et `voyager`. Cette utilisation des représentants moyens ne pose a priori pas de problèmes nouveaux. La partie droite de la figure 5.11 montre un équivalent en SELF, les représentants moyens sont remplacés par des `traits` ne contenant que des méthodes. De plus chaque objet possède un `map`. Le lien `parent` d'un objet est stocké dans son `map`. La gestion automatique du partage entre familles pose de nouveaux problèmes conceptuellement assez similaires à ceux posés par le partage entre les membres d'une même famille et que nous ne détaillons pas plus ici.

Globalement, les méthodes des instances prototypiques et la gestion automatisée des familles supportent donc mal la réalisation de véritables hiérarchies d'héritage, incluant des niveaux abstraits capables de factoriser des propriétés communes à diverses familles d'objets.

[29] Extrait d'un exemple YAFOOL réalisé par J. Quinqueton

5.10 Conclusion et perspectives

Nous avons expliqué la genèse des langages à prototype et les problèmes qu'ils posent. Nous avons proposé une classification des langages à prototypes, via l'étude de la sémantique opérationnelle de leurs mécanismes ; cette classification a été mise en œuvre dans une plate-forme de simulation. Nous avons expliqué les diverses interprétations possibles du lien de délégation et montré ce qui le distingue de celui d'héritage dans les langages à classe. Nous avons enfin fait une étude spécifique des application de la délégation à la représentation d'objets morcelés ayant par exemple des applications pour la gestion de vues, de facettes ou d'aspects d'entités [BD95, BD96].

Les langages à prototypes ont permis de découvrir ou de redécouvrir des mécanismes peu utilisés comme le clonage ou l'héritage entre objets. La conception fondée sur les prototypes offre effectivement une alternative intéressante à la conception fondée sur la représentation de concepts en terme de prototypage d'applications. Les objets sans classes, moins contraints, permettent ou simplifient effectivement la représentation de certaines connaissances.

Une première grande série de problèmes de la programmation par prototypes est liée à la disparition de la possibilité de description en compréhension des concepts. Cette disparition se manifeste dès que l'on souhaite partager des propriétés communes à des familles d'objets. Tous les langages à prototypes ont réintroduits des objets ayant en plus du statut de membre d'une famille, celui de représentant d'une famille d'objets (les instances prototypique des OBJECT-LISP, les représentants moyens de YAFOOL ou les `traits` de SELF), ou celui de descripteur d'objets (les `maps` de SELF ou les familles de clone de KEVO), ou encore celui de bibliothèque de comportements (les `traits` ou les représentants moyens) ou plusieurs de ces statuts à la fois.

Une seconde série de problèmes est relative à des utilisations non fondées du mécanisme de délégation. La relation support au mécanisme est une relation entre objets, que l'on peut nommer ¡¡est-une-extension-de¿¿ ; un objet est en relation avec un autre, son modèle, lorsqu'il est créé par une copie différentielle, c'est-à-dire en exprimant ses différences par rapport au modèle. La délégation est un mécanisme d'héritage qui induit un partage des attributs ou des valeurs de ces attributs entre objets. Ces formes de partage, absentes en programmation par classes, sont à la base de possibilités originales de représentation. Elles sont aussi la cause de problèmes d'inter-dépendance entre objets, ou de gestion d'entités représentés par des ensembles d'objets.

Ces derniers surviennent lorsque la relation ¡¡est-une-extension-de¿¿ est utilisé pour partager des attributs entre objets représentant des entités différentes, ce qui fait voler en éclat la notion d'encapsulation. Il existe aux moins deux interprétations fondées de cette relation. La première interprétation consiste à dire qu'une extension $o2$ d'un objet $o1$ représentant une entité $E1$, représente une entité $E2$ distincte de $E1$ mais ayant par défaut des caractéristique communes. Au niveau de la représentation, ces caractéristiques sont rangées dans $o1$ et héritées donc partagées par $o2$. Nous parlons de sémantique de partage de valeur. Cette interprétation est utilisée dans divers langages comme YAFOOL ou GARNET. La seconde interprétation consiste à utiliser la description différentielle pour exprimer un partage de propriétés entre les différentes parties d'une entité. Une application en est est la représentation, par des ¡¡objets morcelés¿¿, de divers points de vue d'une même entité avec partage possible entre morceaux représentant ces points de vues (`Jean` le sportif et le même `Jean`, professeur, ont le même age) (cf. section 5.4.3).

L'héritage entre objets (fondé sur la description différentielle des objets) et l'héritage entre classes (fondé sur la description différentielle de concepts) sont complémentaires. La délégation n'est pas un mécanisme spécifique de la programmation par prototypes. Elle doit donc être considérée indépendamment des langages dans lesquels elle est utilisée. C'est un mécanisme ayant ses spécificités et ses applications propres ; il peut être mis en œuvre dans un monde ou il y a des classes. Nous avons ainsi abordé le problème de la représentation d'entités avec points de vues d'abord dans le cadre d'un langage à prototype[30] [BD95], puis indépendamment du cadre [BD96]. L'application à un langage à classe est étudiée dans la thèse de Daniel Bardou. Autres exemples : la délégation a été utilisée par Gille Vanvormhoudt comme technique d'implantation du système de points de vue de *Rome* en SMALLTALK [Van94] ou par

[30] Voir aussi le stage de DEA de Cyril Bourdin, encadré en collaboration avec Bernard Carré du LIFL.

ailleurs pour implanter à nouveau la notion de vue dans les bases de données à objets [NM95].

Pour tirer un constat plus général de cette étude, reconsidérons les motivations initiales ayant amené le développement des langages à prototypes. En ce qui concerne la description simplifiée des objets et les possibilités de représentation, le constat est favorable. Si des langages sont développés et utilisés, c'est parce qu'ils répondent à des besoins. Le constat est moins favorable en ce qui concerne la simplicité du modèle de programmation. Sur ce point, l'argumentation en faveur des langages à prototypes était fondée sur un modèle de programmation minimaliste [31]. La réduction du nombre et la simplicité des concepts de base ne s'est pas avérée être un facteur de plus grande simplicité de programmation. Le développement d'applications importantes requiert manifestement la possibilité de décrire des abstractions. A l'heure actuelle aucun des langages à prototypes existants ne le permet de façon réellement satisfaisante.

Les problèmes relatifs à l'organisation des programmes condamnent-ils la programmation par prototypes ? Si l'on réduit ce courant à des langages manipulant uniquement des ¡¡ objets concrets ¿¿ la réponse est certainement positive. Je pense que cette réduction serait une erreur. Les langages à prototypes existants, avec ou sans les solutions apportées aux problèmes d'intégrité et de partage, ont permis l'implantation de logiciels importants, on peut penser à l'environnement SELF [ABC+95], à la construction d'interfaces graphiques [MGdZ92] en GARNET, à l'architecture logicielle du *Newton* d'*Apple* réalisée en *NewtonScript* [Smi95]. Ils ont également été utilisés comme couche basse (ou assembleurs de haut niveau) d'autres langages à objets : les langages YAFOOL ou OBJECT-LISP sont à la base des langages à prototypes mais disposent d'une couche logicielle permettant au programmeur de penser en termes de classes et d'instances[32]. Des applications importantes ont été réalisées en YAFOOL (par exemple *RESYN*). L'utilisateur final de GARNET peut très certainement ignorer qu'il utilise les prototypes.

En résumé, si les langages à prototypes présentent des défauts, les motivations qui ont conduit à leur développement sont toujours d'actualité et les recherches les concernant restent finalement assez peu nombreuses. Les défauts relatifs à l'identité des objets sont graves mais nous avons identifié deux interprétations correctes de la délégation qui les suppriment ou les limitent. Par ailleurs, les problèmes liés à d'organisation condamnent les langages à prototypes purs à ne pas être utilisés pour le développement de logiciels importants. Les pistes de recherche suivante nous semblent intéressantes :

- Développer l'idée d'utilisation des langages à prototypes purs comme couches de bases, comme assembleurs, d'autre langages à objets plus élaborés.
- Nous savons aujourd'hui beaucoup plus de choses qu'il y a dix ans sur la programmation par prototypes. Il serait intéressant de proposer un langage faisant la synthèse des connaissances actuelles. On pourrait y limiter le partage à du partage de valeur entre objets et y autoriser le partage de propriétés entre points de vues sur les objets.
- La piste de recherche que nous avons suivie consiste à intégrer le partage de propriétés ou de valeurs entre objets dans des langages à classes, via l'intermédiaire d'objets morcelés. Nous avons mis en avant l'application de tels objets pour la représentation de points de vues ; il y en a probablement d'autres. Nous voyons également des applications des points de vues et de la délégation dans le cadre de nouvelles recherches sur les ¡¡aspects¿¿. Nous allons en reparler en conclusion générale de ce mémoire.

Remerciements

Je remercie Michel Dao, Roland Ducournau, Jérome Euzenat et Amédéo Napoli pour leurs commentaires avisés sur cette section ainsi que Bernard Carré pour nos fructueuses discussions.

[31] En parlant de la conception du langage SELF : *¡¡ We employed a minimalist strategy, striving to distill an essence of object and message ¿¿* [SU95].

[32] Jusqu'a une certaine limite ; dans les deux cas, ces couches logicielles ne transforment pas ces langages en véritables langages à classes, le partage de valeur d'attributs et ses conséquences étant toujours présent.

Chapitre 6

Evolution des programmes et de leur construction : manipulations de hiérarchies

- Article joint (cf. chapitre 10) relatif à cette section : [DHL97] (article inédit, version longue de [DDHL96]).
- Publications relatives à cette section : [DHL97, DDHL96, DDHL95, DDHL94a, DDHL94b]

6.1 Introduction

Le projet suivant, initié par Marianne Huchard en 1993 et mené depuis 1994 en collaboration avec Hervé Dicky, Marianne Huchard et Thérèse Libourel au LIRMM nous fait entrer dans le domaine de l'étude de l'architecture des applications objet. L'enjeu de cette étude est la bonne structuration des applications, la maîtrise de l'utilisation des moyens mis à la disposition des concepteurs et des programmeurs, sous-typage ou composition par exemple.

Les recherches dans ce domaine, pris au sens large, sont nombreuses et importantes ; elles traitent de la conception de nouvelles applications bien structurées (méthodes de conception [Cor97]), de la conception de classes réutilisables [JF88, Boo93], d'outils d'aide à l'analyse et à la compréhension de bibliothèques de classes [Bor95], de l'¡¡adaptation¿¿ ou restructuration de l'architecture d'applications existantes[1] en vue de l'amélioration de leur structure logicielle voire enfin de la ¡¡rétro-conception¿¿ d'applications existantes [Bez94]. La rétro-conception désigne la construction d'une spécification d'un programme à partir de son code source. L'adaptation désigne la restructuration du code source d'une application. L'adaptation peut bénéficier d'une phase de rétro-conception.

Le projet que nous menons est relatif à la manipulation des hiérarchies d'héritage. et plus précisément à l'insertion semi-automatique de classes dans une hiérarchie. Ces hiérarchies constituent la pierre angulaire de l'architecture des applications ou des bases de données à objets. L'insertion automatique d'une classe a donc des applications assez nombreuses dans les domaines précédents.

- Aide à la conception : elle permet d'évaluer rapidement les conséquences de l'insertion d'une nouvelle classe, voire de constater qu'elle existe déjà sous un autre nom (histoire vécue).
- Aide à la réalisation de frameworks : elle permet de mettre en évidence des classes de factorisation.
- Aide à la restructuration d'applications : elle permet la réorganisation complète de hiérarchies, par remise à plat et insertions successives des classes qui les composent.
- Aide au travail collaboratif : elle peut avoir des extensions à la fusion de hiérarchie réalisées par différentes équipes.

[1] Restructuration de systèmes mal conçus, ou dont les spécifications on disparues, ou devenus difficiles à gérer parce que trop gros ou développés par de nombreuses personnes

Les travaux relatifs à l'analyse et à la réorganisation des hiérarchies sont nombreux [MS89, LBSL90, Ber91, LBSL91, Cas92, Run92, GM93, Cas94, DDHL94a, Cas95, DDHL95, Moo95, CL96, DDHL96, Moo96], le problème ayant des applications industrielles, mais cachent de multiples variations. Notre approche consiste à insérer une classe au meilleur endroit possible dans la hiérarchie, en se basant sur les propriétés (attributs ou méthodes) qu'elle possède et en modifiant la hiérarchie initiale. Un exemple d'approche différente est la ¡¡décomposition¿¿ [Cas94] qui cherche à détecter et à séparer dans une hiérarchie les différents points de vues implantés. Par exemple, dans une hiérarchie représentant des molécules en chimie organique, on peut trouver un ensemble de méthodes pour dessiner les molécules et un autre spécialisées pour la synthèse. Autre exemple, il est intéressant de pouvoir séparer structurellement au sein d'une classe représentant un document (un livre) les aspects ¡¡composition¿¿ et les aspects ¡¡affichage¿¿ [VBL97]. Cette approche est proche d'un nouveau thème de recherches identifié sous l'appellation ¡¡aspect-oriented programming¿¿ [Kic96], visant justement à réaliser cette séparation au moment la conception et de la programmation. Cette approche est intéressante et peut être combinée avec la notre. Une autre approche est nommée ¡¡mise en facteur¿¿[2] ; elle consiste à factoriser, de façon syntaxique, toute instruction commune à deux méthodes dans la hiérarchie [Moo96]. Nous ne voyons pas bien, a priori, comment cette approche pourrait permettre d'obtenir des résultats ayant un sens mais suivons néanmoins ses évolutions.

Pour notre problème d'insertion de classes, nous avons cherché à améliorer les algorithmes existants sur les points suivants : réalisation d'un algorithme incrémental, capacité à préserver un certain nombre de propriétés, si elle en possède, de la hiérarchie fournie en entrée (factorisation maximale des propriétés, chemins d'héritage, classes importantes [3]), et gestion de hiérarchies incluant de la surcharge et du masquage. Le premier problème qui nous est posé est celui des critères selon lesquels on juge de la qualité d'une hiérarchie et qui permettent de réaliser une insertion puis de caractériser les résultats obtenus. Il est clair que bon nombre de ces critères sont relativement subjectifs et informels, c'est pourquoi nous parlons toujours d'une aide à l'insertion de classes car il est clair que l'utilisateur aura toujours son mot à dire. Nous avons jusqu'à présent utilisé le critère de la factorisation maximale des propriétés. Le problème pratique principal est ensuite celui de la gestion de hiérarchies réelles, incluant toutes des propriétés dont les noms sont surchargés ; cette caractéristique rend le problème de l'insertion automatique indécidable en toute généralité car sa gestion nécessite de pouvoir décider de l'équivalence de deux codes.

Nous avons réalisé et publié plusieurs versions d'un algorithmes d'insertion semi-automatique d'une classe dans une hiérarchie. L'article présenté au chapitre 11 [DHL97] est un nouvel article qui fait la synthèse complète de nos connaissances et de nos résultats actuels. Nous les résumons dans les sections suivantes. Ce projet est en cours et offre de nombreuses perspectives dont nous reparlerons en conclusion de ce chapitre.

6.2 Représentation des hiérarchies et treillis de Galois

Il existe en programmation par objets différents critères pour décider si une classe doit être ou ne pas être une sous-classe d'une autre[4]. Un de ces critères est celui de la factorisation maximale des propriétés des classes ; nous appelons propriété un attribut ou une méthode définis sur la classe. Ce n'est évidemment pas le seul critère et il faut bien admettre qu'il est plus facile à modéliser que d'autres. L'expérience montre néanmoins qu'il correspond assez bien à ce que l'on obtient lorsqu'on construit des hiérarchies de classes reflétant une classification de concepts[5].

Le point de départ de notre étude fut l'utilisation, d'après une étude antérieure [GM93], d'une représentation, la sous-hiérarchie de Galois, issue du treillis de Galois de la relation ¡¡a pour propriété¿¿, pour représenter les hiérarchies d'héritages dans le cadre du problème de l'insertion d'une classe[6]. Rappe-

[2] Refactoring

[3] L'algorithme étant susceptible de faire disparaître certaines classes, qui suite à la réorganisation ne définissent plus aucune propriété, il est possible d'indiquer que certaines classes ne doivent pas disparaître.

[4] C'est d'ailleurs un des rôles des méthodes de conception que d'aider les programmeurs à s'y retrouver.

[5] A condition de ne considérer que les propriétés constituant l'interface des classes.

[6] Les treillis sont par ailleurs utilisés comme support à la représentation et au codage efficace des hiérarchies d'héritage [Cas87, Cas93, HNR97] ; l'utilisation de la notion de treillis est ici tout à fait différente.

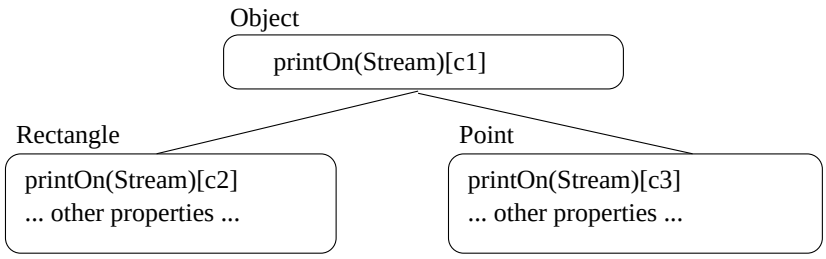


FIG. 6.1 – Surcharge et masquage.

lons l'idée. Comme l'explique la section 7 du chapitre 11, si l'on considère un ensemble de classes {1 2 3 4} et un ensemble de propriétés {*a b c d e*} (figure 16.a, chapitre 11), le treillis de Galois de la relation ¡¡définit la propriété¿¿ (figure 16.b, chapitre 11) peut être transformé simplement en treillis d'héritage de Galois (figure 16.c, chapitre 11), structure que l'on peut assimiler sans aucun souci à une hiérarchie d'héritage dans laquelle chaque noeud non vide représente une classe dotée des propriétés qu'elle définit localement, les noeuds supérieurs représentant les classes dont elle hérite. La propriété du treillis de Galois qui nous intéresse ici est qu'il garantit d'une part une factorisation maximale des propriétés et d'autres part la factorisation la plus compacte en nombre de classes (figure 3, chapitre 11). La représentation optimale d'une hiérarchie étant obtenue en supprimant les noeuds vides dans une structure appelée ¡¡sous-hiérarchie de Galois¿¿ (cf. figure 16.d, chapitre 11).

6.3 Algorithme Ares

L'algorithme ARES (Ajout et REStructuration) est un algorithme d'insertion d'une classe dans une hiérarchie. C'est un algorithme incrémental, capable de préserver l'éventuelle structure de sous-hiérarchie de Galois de la hiérarchie donnée en entrée. Ceci signifie que si la hiérarchie fournie en entrée est factorisée maximalement alors la hiérarchie fournie en sortie le sera également[7]. Ceci signifie également que si l'on reconstruit de toute pièce une hiérarchie à partir d'un ensemble de classes, la structure de sous-hiérarchie de Galois utilisée par l'algorithme assure que la hiérarchie produite sera maximalement factorisée et compacte ; la hiérarchie finale étant indépendante de l'ordre d'insertion. Les premières versions de l'algorithme, réalisées par H.Dicky, T.Libourel et M.Huchard, traitaient des hiérarchies sans surcharge telles que celles décrites dans la section 3.1, chapitre 11 ou des hiérarchies avec surcharge en supposant connu un ordre partiel entre les différentes occurrences des propriétés de même nom.

Dés le début de mon travail sur le projet, je me suis particulièrement intéressé au problème du traitement pratique de la surcharge. La surcharge des noms de propriétés désigne la possibilité d'utiliser le même nom pour désigner des propriétés différentes [CW85]. Le masquage est un cas particulier de surcharge, ayant un sens en présence d'héritage (il y a de la surcharge en ADA mais pas de masquage) ; il désigne le cas ou deux propriétés de même nom sont définies sur des classes dont l'une est une sous-classe au sens large de l'autre. La propriété définie sur la sous-classe est une spécialisation de la propriété de même nom de la sur-classe. La figure 6.1 montre un exemple de hiérarchie (fortement inspiré par SMALLTALK) dans laquelle trois propriétés de même nom sont définies ; elles ont des code différents (entre crochets). La méthode `printOn` de la classe `Point` (nommons la `p2`) est une spécialisation de celle (`p1`) définie sur `Object`. La surcharge et le masquage, présents dans toutes les hiérarchies réelles compliquent notre problème d'insertion. En effet, à chaque fois que deux propriétés de même nom se trouvent respectivement sur la classe à insérer et quelque-part dans la hiérarchie, se pose le problème de la comparaison de ces propriétés : sont-elles différentes et si oui, l'une d'elles est-elle une spécialisation de l'autre.

Une première version du traitement de la surcharge est décrite dans [DDHL95]. Si l'on appelle propriété générique (à la CLOS), l'ensemble des propriétés de même nom, l'insertion automatique en présence

[7] ARES accepte également en entrée des hiérarchies quelconques.

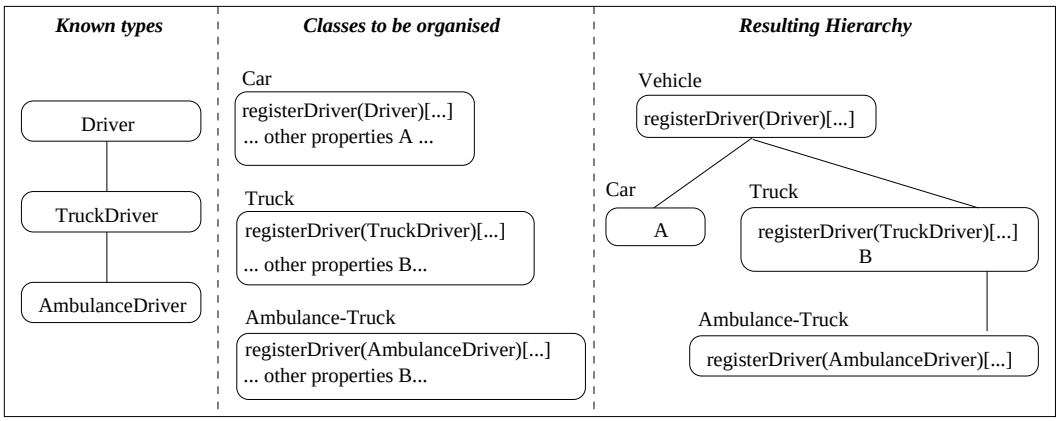


FIG. 6.2 – *LCG* : utilisation des signatures des propriétés

de surcharge peut s'effectuer en supposant connues, pour chaque propriété générique, les relations de spécialisation entre les différentes propriétés qui la composent. Par exemple la propriété générique `printOn` serait constituée dans notre exemple des propriétés `p1`, `p2` et `p3`. Si l'on considère une hiérarchie constituée des classes `Object` et `Rectangle`, Ares y insère correctement la classe `Point` à condition de savoir que `p2` est une spécialisation de `p1` et est incomparable avec `p3`. Un exemple plus complet est donné au chapitre 11, section 3.2. Evidemment, un modèle sans surcharge aurait considéré que les trois propriétés étaient identiques, puisque de même nom, et aurait factorisé la propriété sur la classe *Object*.

L'étape suivante dans la gestion de la surcharge, proposée dans la troisième version de Ares et publiée dans [DDHL96] consistait donc à faire en sorte que l'algorithme sache comparer les propriétés, en utilisant leurs définitions (signatures et codes). Le problème de la comparaison des propriétés de même nom rend l'insertion d'une classe indécidable en toute généralité car celle-ci peut nécessiter la connaissance de l'équivalence des codes. Il existe néanmoins un grand nombre de situations où il est possible d'automatiser cette comparaison. L'algorithme doit également prendre en compte les règles de masquage de propriétés propres à chaque langage[8]. La section suivante propose deux exemples d'insertion dans lesquels se pose un problème de surcharge.

6.4 Deux exemples d'insertion avec surcharge

Soient p_i et p_j deux propriétés de même nom `p`. Nous appelons ¡¡lowest common generalizations¿¿ et notons $LCG(p_i, p_j)$ l'ensemble des plus ¡¡plus petites¿¿ généralisations de deux propriétés de même nom. Dans la plupart des cas, $LCG(p_i, p_j)$ est un singleton que nous assimilerons à l'ensemble. Voici deux exemples de calcul de *LCG* et d'exploitation du résultat par ARES.

- Les signatures des propriétés donnent la *LCG*

Le premier exemple (*cf.* Figure 6.2) vient de EIFFEL et est extrait de [Mey92]. La hiérarchie est constituée d'une seule classe Car et on souhaite y insérer la classe $Truck$. Il est nécessaire de comparer les propriétés $p_1 = registerDriver(TruckDriver)$ et $p2 = registerDriver(Driver)$. Sachant que $TruckDriver$ est un sous type de $Driver$ et que Eiffel autorise les redéfinitions covariantes, il est aisé de déduire que $p_1 < p_2$ et donc que $LCG(p1, p2) = p2$. En conséquence, *Ares* définit p_2 dans une nouvelle classe de factorisation (que NOUS, pas *Ares*, nommons $Vehicule$). Les mêmes critères sont utilisés pour l'insertion de la classe $AmbulanceTruck$ qui produit la hiérarchie finale. *Ares* est capable dans un tel cas de produire une hiérarchie avec plusieurs niveaux de masquage.

[8]Par exemple, en C++ ou en JAVA, la signature d'une méthode redéfinie doit être la même que celle de la méthode qu'elle redéfinie (invariance) alors qu'Eiffel autorise les redéfinitions covariantes pour tous les paramètres ainsi que pour le type de retour.

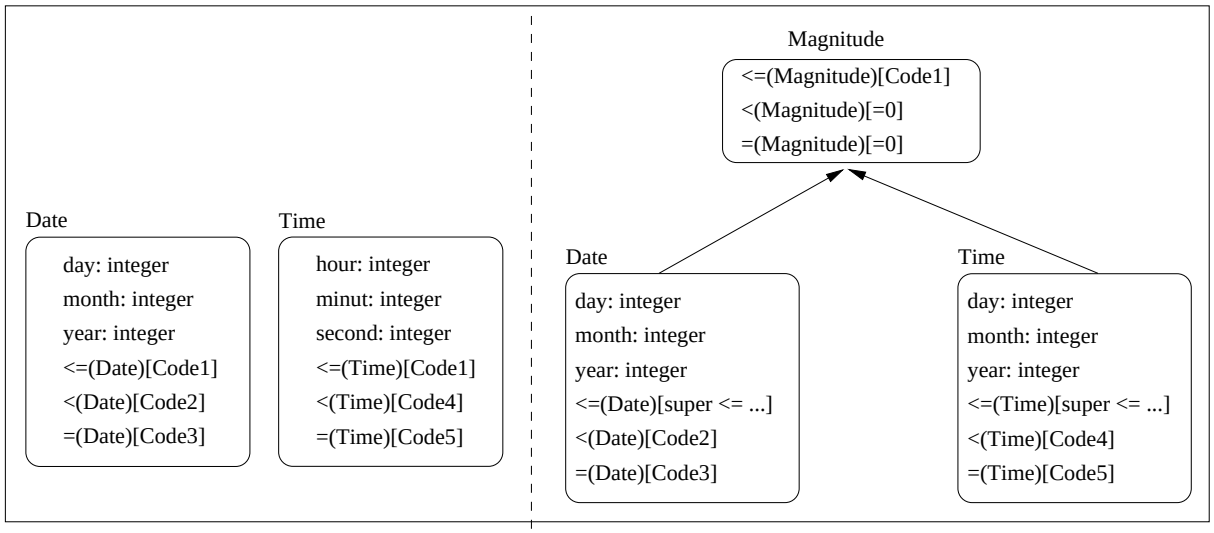


FIG. 6.3 – LCG : Utilisation des signatures et du corps des propriétés

• Les codes et les signatures des propriétés donnent la LCG

Le second exemple, (*cf.* Figure 6.3) est inspiré de SMALLTALK [GR83] et transposé dans un univers ou les identificateurs sont typés. Cet exemple pourrait être également judicieusement traité par la généricité (types paramétrés) mais nous avons travaillé à produire un résultat équivalent à la hiérarchie SMALLTALK. Etant donné une hiérarchie composée de la classe $Date$, l'insertion de la classe $Time$ doit produire une classe que nous nommons $(Magnitude)$[9] factorisant la méthode $<=$ partagée par $Date$ et $Time$ et détenant deux versions ¡¡déferrées¿¿[10] des méthodes $<$ and $=$. Les propriétés $<=, <, =$ des deux classes doivent pour cela être comparées et les trois LCG calculées.

Cet exemple contient une difficulté supplémentaire, les paramètres des méthodes de la classe à insérer ($Time$), par exemple ceux de la méthode $<$, sont de type $Time$, c'est-à-dire de type ¡¡like current¿¿, soit encore d'un type non connu dans la hiérarchie mais dont nous devrions connaître la position pour calculer la LCG. Nos tables de calcul de LCG incluent donc des entrées correspondant au cas ou les propriétés à comparer sont potentiellement égales ou potentiellement comparables. Des signatures sont par exemple potentiellement égales si elles ont respectivement à la même position, soit le même type soit ¡¡like current¿¿.

- Les deux méthodes $<$ (de $Date$ et $Time$) ont donc des signatures potentiellement égales et des corps différents. Ceci suffit à Ares pour déterminer que les deux méthodes sont incomparables et que leur LCG est une méthode virtuelle pure qui doit être définie dans leur super-classe commune ; le paramètre de cette méthode ayant pour type cette super-classe commune.
- Les deux méthodes $<=$ (de $Date$ et $Time$) ont des signatures potentiellement égales et le même code $code1$. Ceci permet à Ares de déterminer que leur LCG, est une méthode de code $code1$ qui doit être définie sur leur super-classe commune C avec un paramètre de type C. Aussitôt que cette super-classe sera déterminée, l'information sera mise à jour. Ares détermine également qu'il est important de laisser dans $Date$ et $Time$ des versions de la méthode $<=$ afin de faire un contrôle de type, leur code étant un appel à la super méthode.

Il est important de noter que la hiérarchie résultante produite dans cet exemple, qui peut être discutée, n'est pas la seule que nous puissions produire. En fait *Ares* la première partie de l'algorithme calcule pour chaque couples de propriétés, les LGC et les $remainder$[11]. Ces résultants pouvant ensuite être combinés de différentes façons. Le fonctionnement détaillé du calcul de la LCG, de son utilisation par

[9] L'ensemble des classes dont les éléments peuvent être comparés deux à deux.
[10] ou virtuelles pures.
[11] Propriétés restantes pouvant masquer la LCG sur les sous-classes.

Ares et de l'algorithme proprement dit sont présentés au chapitre 11, sections 4 et 5. L'ensemble des cas de surcharge que nous savons actuellement traiter est détaillé en annexe. La section 6 propose enfin deux études de cas plus complètes dans lesquelles nous avons par exemple appliqué l'algorithme à la hiérarchie complète des Magnitude de SMALLTALK. Cette étude de cas met en évidence, d'une part des défauts de la hiérarchies SMALLTALK découverts par Ares et d'autre part certaines limites actuelles de l'algorithme liées à l'utilisation du principe factorisation maximale.

6.5 Conclusion et perspectives

La version de Ares intégrant la réalisation automatique de certains cas de comparaisons de propriétés a été présentée dans [DDHL96] ; une version plus récente, correspondant à l'état actuel de l'algorithme est proposée par [DHL97] et est jointe à ce mémoire au chapitre 11. La section 8 de cette article compare Ares à ses pairs ; la comparaison porte sur un ensemble de critères assez pointus que j'invite lecteur intéressé à consulter. Elle montre notamment que nous sommes les premiers à avoir traité de façon réaliste du problème la surcharge et à traiter de manière automatique un certain nombre de cas de comparaison de propriétés. Les perspectives de recherche liées à cette étude sont importantes, relativement au développement de l'algorithme d'une part et en élargissant le problème d'autre part.

L'algorithme offre de larges potentialités d'extensions et d'améliorations.

- Poursuite du traitement de la surcharge qui n'en est en fait qu'à ses débuts. Les comparaisons de code peuvent être améliorées pour détecter des cas d'inclusion de code et la génération automatique d'appels à la ¡¡super-méthode¿¿. Nous pourrons utiliser localement à cet effet des techniques de ¡¡refactoring¿¿ [Moo96].
- Paramétrage de l'algorithme en fonction des possibilités du langage cible. Pour le moment nous traitons de la redéfinition covariante des propriétés.
- Intégration d'autres possibilités de représentation ; dans l'exemple de *Date* et *Time*, il serait intéressant de générer un type paramétré si le langage cible le permet.
- Etude de l'abandon du principe de factorisation maximale, et donc de la représentation par sous-hiérarchies de Galois, qui ne correspond pas toujours à la pratique des utilisateurs comme le montre l'étude de cas de la section 6.2 (chapitre 11), et qui produit souvent de trop nombreuses classes de factorisations ; cette remise en question est une modification majeure de l'algorithme.
- Application de l'algorithme à des hiérarchies de grande taille, ce qui n'a pas encore été fait.
- Mise en œuvre des dernières versions en JAVA, comme il se doit.
- Plus extrapolatoire, utiliser des techniques d'inférence de type pour la réorganisation des hiérarchies dans les langages typés dynamiquement.

L'élargissement du problème se conçoit à deux niveaux.

- Le projet se poursuit actuellement dans le cadre d'une convention avec le CNET avec l'équipe de Michel Dao. Nous nous proposons cette fois de réaliser un environnement de manipulation de hiérarchies offrant une palette d'outils, des plus simples (comme l'ajout d'une propriété à une classe) aux plus complexes (comme la fusion de hiérarchies[12]). Hervé Leblanc vient de débuter un travail de thèse dans le cadre de cette convention. Cet environnement doit pouvoir être utilisé comme support à des méthodes de conception ou de composition de programmes.
  Notre plan de travail dans ce cadre prévoit (1) l'étude d'un nouveau modèle de représentation proche d'UML pour manipuler les hiérarchies, indépendamment des particularités des différents langages et de leurs syntaxes puis (2) une étude des critères, formels et informels, sur lesquels on juge de la qualité des hiérarchies, et (3) la réalisation de l'ensemble d'algorithmes évoqués ci-avant.
- Un objectif encore plus général est de réfléchir à l'intégration de ces outils dans un atelier d'assemblage de composants logiciels. Nous en reparlons dans la conclusion générale.

[12] Il peut par exemple être utile de fusionner deux hiérarchies développées par des équipes différentes mais traitant d'un même thème.

Chapitre 7

Evolution vers les systèmes ouverts : systèmes réflexifs

– Article joint (cf. chapitre 10) relatif à cette section : [MDC96].
– Publications relatives à cette section : [MDC96, MDC92, MCDM92, MCDM91, Don88b]

Les systèmes réflexifs jouent évidemment un rôle fondamental pour l'évolution des systèmes et des formalismes puisqu'ils permettent à tout utilisateur, de tester de nouvelles idées, de comprendre ou d'adapter une application, sans pour autant être un spécialiste de la compilation ou de l'interprétation. Combien compte-t-on d'extensions de SMALLTALK ou de CLOS ? Combien de prototypes ont-ils été réalisés avec ces langages ? On définit un système réflexif comme un système capable de fournir à ses utilisateurs une représentation de tout ou partie de lui-même en connexion causale[1] avec son implantation effective en machine [Mae87]. La réflexivité peut être limitée à l'introspection, qui donne un droit en lecture mais pas en écriture aux structures et mécanismes d'un système. L'introspection est bien sûr plus facile à mettre en œuvre que la connexion causale.

La réflexivité est souvent décriée par les tenants de systèmes sûrs et efficaces. Ils ont raison en ce sens que la connexion causale permet, si elle est mal utilisée, de casser (plus ou moins) un système[2]. Ils ont encore raison en ce sens que la connexion causale est difficile à implanter et peut ralentir l'exécution des programmes. Ils ont tort car la réflexivité est une notion fondamentale en génie logiciel, comme outil de manipulation et donc de structuration potentielle des programmes [LC96] et comme outil de réalisation de systèmes ouverts, c'est-à-dire capables de s'adapter (eux-même) ou d'être adaptés par leurs utilisateurs. Les études chiffrés sur les problèmes du logiciel montrent bien qu'un des tout premier problèmes des acheteurs est la vitesse à les systèmes qu'ils achètent deviennent obsolètes[3] parce que le cahier des charges initial était insuffisamment visionnaire. Les mentalités évoluent néanmoins, les langages à objets offrent aujourd'hui (au moins en lecture) aux programmeurs, une représentation des classes à l'exécution[4]. Pour confirmer cette évolution, rendre les systèmes réflexifs plus sûrs et plus efficaces est toujours un sujet de recherche d'avenir.

En ce qui concerne la recherche dans le domaine des langages réflexifs, les résultats actuellement les plus aboutis, même s'il reste des problèmes [Mul95], sont relatifs à la réflexion de structure, c'est-à-dire à la représentation réflexive des structures de données des langages. Ainsi, les systèmes (CLOS, LORE, OBJVLISP, SMALLTALK, etc[5]) dans lesquels des méta-objets [KdB91] représentent les structures de données telles que classes [Coi87, BC89] , relations [Cas87, Cas89, Fer89a], attributs, méthodes, voire exceptions ou pile d'exécution, sont assez bien maîtrisés aujourd'hui. Toute une série de problèmes difficiles restant

[1] Si l'une est modifiée, l'autre doit l'être en conséquence
[2] Exemple : `(defun eval (l))`
[3] Ce n'est peut-être pas un problème pour les fournisseurs de système, mais ceci est un autre débat.
[4] Avec la classe `java.lang.class`, nous sommes certes encore loin d'un ¡¡ méta-object protocol ¿¿ complet en JAVA.
[5] Liste n'ayant ici aucune prétention à l'exhaustivité, faite des principaux systèmes réflexifs que j'ai utilisés.

à résoudre sont relatifs à la représentation réflexive des mécanismes de calcul (on parle de réflexion de comportement[Mae87, Fer89b]) et à la réalisation d'implantations efficaces associées. Dans le cas d'un langage à objets, on souhaite par exemple implanter une description réflexive, donc particularisable pour des objets ou des groupes d'objets, de la recherche et de l'application des méthodes.

Les systèmes réflexifs, qu'ils s'agisse de les utiliser ou d'en concevoir de nouveaux, sont importants et récurrents dans mes recherches sans en constituer le thème véritablement central. En ce qui concerne leur utilisation, la plupart des systèmes que j'ai implantés utilisent intensivement les possibilités réflexives des systèmes sous-jacent et sont eux-mêmes réflexifs. Citons l'exemple du système de gestion des exceptions LORE ou celui de SMALLTALK intégralement implanté en SMALLTALK grâce à la possibilité de réification des blocs de la pile d'exécution. Ce système est lui-même réflexif : les exceptions qu'il signale le sont par exemple grâce à ses propres primitives. Il est donc possible de les rattraper et de modifier ainsi son fonctionnement.

J'ai par ailleurs collaboré sur ce sujet à des études menées par Pierre Cointe et Jacques Malenfant. L'article présenté au chapitre 12 fait la synthèse des études auxquelles j'ai participé. Comme je l'ai expliqué, un de nos intérêts pour la programmation par prototype était motivé par l'étude de la réflexion de comportement [Mae87]. L'étude de cette dernière dans un langage à classe [Fer89a, Fer89b] est plus ardue car les classes sont déjà des entités complexes et surchargées de rôles [Bor86c] (descripteur de la structure et des comportements des instances, modules, outil pour la réutilisation, etc). Leur ajouter celui de descripteur des mécanismes de calcul ne fait qu'aggraver le constat [MCDM91].

Nous avons donc proposé un modèle à prototypes réflexif dans lequel a été implantée une description réflexive du couple ¡¡recherche de méthode (`lookup`) - application de méthode (`apply`)¿¿ [MCDM92, MDC92] permettant de paramétrer pour tout objet, la recherche de méthodes et pour toute méthode la façon dont elle est appliquée. Ce modèle est basé sur l'existence de méta-objets associés à chaque objet (cf. chapitre 12, section 3). Les premiers problèmes de ce modèle, comme toujours avec les représentations réflexives, sont ceux du démarrage (bootstrap) de l'arrêt des régressions infinies qui ne manquent pas d'apparaître lors de la recherche et de l'applications des méthodes. Les régressions se situent a différents points clés de l'envoi de message : pour rechercher la méthode m d'un objet o, on doit rechercher son méta-objet (première régression si ceci est effectué par envoi de message), puis rechercher sur son méta-objet mo la méthode `lookup` grâce à laquelle la recherche de m pour o va s'effectuer. Mais il faut pour cela chercher sur le méta-objet de mo la méthode `lookup` grâce à laquelle la recherche de *lookup* pour mo va s'effectuer, etc (seconde régression, voir le chapitre 12, section 3.1 pour plus détails). Un phénomène similaire se produisant au moment d'appliquer une méthode. Ces régressions sont réglées (cf. chapitre 12, section 4) de façon classique, comme dans le modèle de métaclasses à la Objvlisp, par l'existence de méta-objets qui sont leurs propres méta-objets et par une recherche cablée (non-réflexive) des méta-objets.

La validation (la preuve de convergence du processus d'envoi de message dans tous les cas de figures, que certaines méthodes aient été redéfinies ou non) expérimentale de ce modèle a été réalisée dans une la version réflexive du langage Self [MCDM92] implantée par Philippe Mulet, il a par la suite été ré-implanté par Marco Jacques dans *proto-reflex* [JM95] ainsi que dans MOOSTRAP [Mul95]. Une première validation formelle du modèle est proposée dans [MDC92]. Elle est reprise et complétée dans [MDC96] avec la représentation explicite des continuations et en en utilisant un modèle théorique basé sur les règles de réécriture.

J'ai du arrêter, suite à mon départ de Paris, faute de temps et pour avoir fait d'autres choix, de travailler sur ce sujet précis pour lequel les recherches se sont focalisées sur le problème fondamental et difficile [Fer89a] de l'exécution efficace des programmes dans un tel contexte.

Chapitre 8

Conclusion et perspectives globales

Les résultats proposés dans ce mémoire, produits au travers d'un ensemble de collaborations et via l'encadrement de stages de DEA et de thèses en cours, forment une contribution à l'évolution des environnements de conception et de programmation dédiés à la réalisation, avec des langages à objets, de programmes fiables, bien conçus, réutilisables et ouverts. Cette contribution porte sur différents aspects :

- fiabilité des programmes : utilisation du système de gestion des exceptions ;
- représentation des connaissances en programmation par objets : compréhension des utilisations correcte du mécanisme de délégation ;
- réutilisabilité des programmes : réalisation de modules résistant aux exceptions et paramétrables par des traitements d'exceptions spécifiques ; assistance à la réorganisation de hiérarchies ;
- connaissance des langages : compréhension de la programmation sans classes ; framework pour la réalisation rapide d'interprètes ;
- ouverture : études sur la réflexion, utilisation des systèmes réflexifs.
- application des technologies objets : réalisation de frameworks.

J'ai acquis en réalisant ces travaux, une connaissance multi points de vue du formalisme de programmation par objets, de sa mise en œuvre et de son utilisation. Inversement, cette vue d'ensemble a été bénéfique à mes divers travaux. La connaissance des systèmes réflexifs m'a par exemple été fort utile pour la réalisation du système de gestion des exceptions ; de même que la connaissance des langages et de la structure des applications pour la réorganisation de hiérarchies. D'un point de vue opérationnel, la cohérence de l'ensemble des travaux peut être matérialisée. Il serait conceptuellement assez simple d'étendre un environnement de programmation existant par les outils et modèles que j'ai réalisés ou spécifiés. Cet environnement serait clairement architecturé autour d'un langage à classes, si possible doté de méta-classes. Il pourrait alors être doté d'un système performant de gestion des exceptions ; il serait possible d'y utiliser des objets morcelés pour gérer des points de vues en utilisant a bon escient le mécanisme de délégation ; il intégrerait un module d'aide à la construction et à la réorganisation de hiérarchies ; il intégrerait également un assistant qui automatiserait les tâches répétitives (voir ci-dessous la section 8) et serait doté d'un ensemble d'outils de mise au points. La seule chose que je n'y intégrerait pas est le modèle à base de méta-objets pour la réflexion de comportement ; non pas qu'il soit inintéressant mais d'une part il est conçu pour un monde sans classes et d'autre part, si l'on s'en tenait aux résultats auxquels j'ai participé, il ralentirait trop l'exécution des programmes.

Inversement, chacune des études spécifiques décrites dans ce mémoire propose un tout cohérent, forme une contribution à un domaine spécifique et peut être évaluée individuellement. Les conclusions de chaque chapitre, font un bilan détaillé[1] de mes résultats, des recherche en cours et des perspectives propres à chacun des thèmes abordés. Il ne me semble pas utile de les répéter ici dans le détail, j'invite le lecteur à s'y référer, voici un condensé des résultats.

- Les système de gestion des exceptions pour la programmation par objets sont aujourd'hui correctement formalisés. L'implanteur d'un nouveau langage à le choix entre des solutions différentes mais éprouvées.

[1] Eventuellement en renvoyant à un bilan réalisé dans un des articles joints à ce document.

- Les langages à prototypes purs peuvent être de bons outils de prototypage. Ils peuvent être utilisés comme assembleurs de haut niveau pour la programmation par objets. Ils ne conviennent pas pour la réalisation de logiciel de grande taille à cause de leur manque d'outil de modélisation d'abstractions. Le mécanisme de délégation n'est pas spécifique aux langages à prototypes, il peut être mis en œuvre dans un monde de classe qu'il enrichira de possibilité nouvelles d'expression en autorisant le partage de propriétés entre objets.
- Les programmes à objets sont architecturés en hiérarchies de classes. Il est possible d'aider les concepteur et les mainteneurs de ces hiérarchies avec des algorithmes spécialisés de manipulation. Nous avons proposé un algorithme d'insertion d'une classe dans une hiérarchie qui garantit une factorisation maximale des propriétés et sait gérer un certain nombre de cas de surcharge.
- Nous avons montré par de nouveaux exemples l'intérêt de la représentation réflexive des structures de données des langages. Nous avons contribué au développement des systèmes réflexifs en prouvant qu'il est possible de représenter réflexivement le processus de recherche et d'invocation de méthodes à l'aide de méta-objets dans un langage à prototypes.

Qu'en est-il des futures évolutions des langages à objets. La question doit s'envisager relativement aux nouveaux thèmes qui mettent directement en jeu les langages à objets. L'existance du thème "évolution des langages à objets" est une conséquence de la nécessité de collaboration entre les nombreux chercheurs, de différents domaines (architectures réparties, génie logiciel, bases de données, présentation de connaissances), qui continuent à adapter le formalisme objet à leurs besoins. Des nouveautés ont par exemple été apportées récemment par le langage JAVA ; elles sont en grande partie relatives [Fla96] premièrement aux problèmes de sécurité que pose l'exécution de morceaux de code (*applets*) compilés en un point du réseau et exécutés en un autre point au sein d'un navigateur, et secondement à la production de composants interopérables (*JavaBeans*)[2].

Un de ces défis nouveaux proposés au formalisme objet est lié au développement du réseau qui suggère une nouvelle étape dans l'industrie du logiciel : le développement de programmes par recherche et assemblage de composants logiciels spécialisés, répartis sur le réseau et éventuellement écrits dans différents langages.

- Le défi consiste d'une part à réaliser des programmes en assemblant (idée du légo ou différents composants adaptables s'assemblent pour former des structures) ou en faisant collaborer des composants exécutables, répartis sur le réseau. L'exécution d'objets concurrents et répartis est étudiée depuis assez longtemps[BGY96]. Aux problème connus de collaboration entre objets répartis s'ajoutent ici ceux de la recherche et de la documentation des composants et ceux de l'assemblage et de l'interopérabilité ; comment faire coopérer ou combiner des codes exécutables produits à partir de langages différents[3].
- D'autre part, le défi est de réaliser des applications par la composition [NT95] et la réutilisation des composants dont on possède les sources. Cette dernière idée n'est pas nouvelle :

  > ¡¡ The main activity of programming is not the origination of new independant programs, but in the integration, modification, and explanation of existing ones ¿¿ T.Winograd, 1979 [Win79].

  Sa mise en pratique prend aujourd'hui tout son sens, de par l'évolution du réseau et des connaissances sur la réutilisation et la décomposition modulaire des programmes. Elle peut éventuellement laisser l'industriel méfiant : qui va développer les composants ? combien vont-ils coûter ? qui garantira leur qualité ? Ces problèmes qui se posent déjà aujourd'hui continueront sûrement à se poser. Mais elle est en fait déjà commencée. Des normes de communication pour les applications réparties ont été proposées *COM-OLE* de *Microsoft* ou *CORBA* de *l'OMG*[4]. Des modèles de composants (service logiciel que l'on peut intégrer à une application) ont été produits (contrôles *Active/X*

[2] La conception de JAVA fait par ailleurs une très large part à la synthèse de traits connus (machine virtuelle à la Smalltalk pour la portabilité, gestion automatique de la mémoire et variables de type référence à la Lisp, syntaxe et typage à la C++, processus concurrents, paquetages, exceptions, etc [DB96]).

[3] Même dans l'optique simplificatrice ou tous les composants seraient compilés en terme des instructions d'une unique machine virtuelle, par exemple celle de JAVA, comment par exemple compiler en instructions de cette machine virtuelle un programme écrit en utilisant l'héritage multiple.

[4] "Object Management Group", consortium d'industries dont SUN, HEWLETT-PACKARD ou DIGITAL.

(technologie Microsoft) ou *Java-Beans*. Cette idée pose par ailleurs aux chercheur une foule de problèmes nouveaux et intéressants.

Ces problèmes ont été répertoriés lors des premières discussions dans le cadre de la restructuration du Gdr programmation qui devrait s'opérer, pour ce qui concerne les équipes objets et programmation répartie, autour de cette problématique d'objets répartis et d'assemblage. Voici, dans ce cadre, des précisions sur quelques axes de recherche qui m'intéressent et dont certaines sont déjà en cours.

- Gestion des composants.
  Recherche des composants sur le réseau, gestion de leur évolution (gestion de leurs versions), maintien de la cohérence des programmes les utilisant. Ce sujet offre des connexion fortes avec certaines recherches en bases de données à objet.
- Assemblage de composants.
  L'assemblage de hiérarchies est un point de vue sur l'assemblage de composants. L'environnement de manipulation de hiérarchies (construction, réorganisation, fusion, etc) évoqué en conclusion du chapitre 6, section 6.1, s'intègre bien sûr parfaitement au projet d'assemblage de composants. Il sera intéressant de le coupler avec en environnement de gestion des composants tel qu'évoqué ci-avant. Une collaboration sur ce thème est en cours de constitution avec des équipes de l'université de Nice.
- Applications des objets morcelés à la séparation des aspects d'une application pour de nouvelles formes de réalisation modulaire des composants.
  Une nouvelle direction de recherche est relative à la mise en œuvre de nouvelles formes de modularité permettant de séparer structurellement les différents aspect d'une réalisation logicielle. Rappelons l'exemple de la classe `Document` [VBL97] vue sous les aspects ¡¡analyse¿¿ ou ¡¡affichage¿¿. On parle de ¡¡programmation par aspects¿¿ [Kic96] ou encore de ¡¡conception par sujets¿¿ [OKH+95] (un sujet est une spécification d'un point de vue sur une application ou un type de données) et nous avons évoqué la détection automatique des aspects dans une réalisation existante [Cas95]. La séparation des aspects facilitera évidemment considérablement l'assemblage hiérarchies et plus généralement de composants. Les objets morcelés utilisant la délégation semblent pouvoir s'appliquer très directement à la représentation de classes avec aspects. Cette idée est issue de discussions avec Jacques Malenfant et Bernard Carré.
- Mise en œuvre de l'adaptabilité des composants.
  L'aptitude des composants à être adaptés conditionnera évidemment les capacités d'assemblage. Une part du problème consiste à permettre aux utilisateurs d'adapter les composants à leur problème et à leur cadre de travail. Une seconde question concerne la capacité des composants à s'adapter dynamiquement aux événements survenant sur le réseau ou dans leur environnement. Dans les deux cas, il s'agira d'une application des techniques de représentation réflexives par objets.
- Gestion des exceptions en milieu réparti.
  La question est simple : que se passe-t-il si l'exécution d'un composant sur une machine distante échoue ? Il y a une connexion à établir avec les études similaires pour les systèmes multi-agents.
- Environnements de programmation spécialisés et assistance aux utilisateurs.
  Les différences entre un environnement classique et un environnement spécialisé pour les composants concerneront l'aspect réparti des outils de mise au point (de nombreux environnements existent déjà), l'existence d'outils spécifiques à l'assemblage tels que la détection des incompatibilité, l'existence d'outils de recherche et de gestion des composants.

Autre projet en cours

A propos des environnements de programmation, je souhaite dire un mot d'un projet récent relatif à l'assistance aux utilisateurs des environnements objet. Ces derniers ont également beaucoup évolué depuis *Interlisp* et SMALLTALK [5] mais j'ai été frappé par le fait qu'ils ne prennent rarement correctement en charge l'utilisateur dans la réalisation de tâches répétitives ou fastidieuses. J'ai initié un projet plus

[5] Encore qu'une bonne part de l'évolution ait consisté à refaire pour des langages statiquement typés ce qui existait en 1978 avec *Interlisp*.

récent qui traite de la conception d'assistants à l'utilisation de logiciels interactifs. Par logiciel interactif, j'entend en premier lieu environnement de programmation interactif tel celui de SMALLTALK, mais le terme pourrait s'appliquer à l'environnement d'utilisation d'*Excel* ou de *Word*. Cet étude pourra avoir des applications pour l'assistance à la navigation sur le réseau.

Ce projet a débuté au travers de divers stages (cf. stages de Jean-François Bernier, Florent Jugla et Jean-David Ruvini, section 2.10) et est devenu le travail de thèse de Jean-David Ruvini depuis octobre 1996 ; il fait l'objet d'une collaboration avec Philippe Reitz au sein de l'équipe ¡¡Objet - Acteurs - Agents¿¿ du département ARC. Il traite plus précisément de l'assistance à un utilisateur humain ou à un programme informatique en train d'exécuter une tâche de type réactif ou interactif, i.e. une tâche où l'opérateur exécute des suites d'actions en réponse à une situation ou à des informations qu'il reçoit. L'assistance est organisée autour de deux agents logiciels (cf. travaux de Patti Maes) : l'apprenti et l'assistant. L'apprenti observe l'opérateur et enregistre ses réactions face aux situations qu'il rencontre. L'apprenti travaille sur une représentation objet de l'environnement de travail du programmeur, il utilise les techniques de détection de répétition et de détection de similarités issues de recherches en apprentissage. Philippe Reitz est plus particulièrement responsable de ces points. L'assistant travaille (entre autres) à partir des règles retenues par l'apprenti, il tente d'aider l'opérateur à chaque fois qu'une situation reconnue comme typique par l'apprenti se présente. Les premiers résultats ont été présentés dans diverses journées dont [RD97]. Aujourd'hui une version d'un apprenti et d'un assistant ont été implantées au sein de l'environnement SMALLTALK, ils sont opérationnels et peuvent faire l'objet d'une démonstration. Ils sont en cours de publication.

Les perspectives sur ce projet concernent la réalisation d'un framework pour l'apprentissage par détection de similarités et l'extension notre méthodologie à l'apprentissage des habitudes de travail communes à des groupes d'utilisateurs travaillant sur un même projet.

Bibliographie

[ABC+95] O. Agesen, L. Bak, C. Chambers, B.W. Chang, U. Hølzle, J. Maloney, R.B. Smith, D. Ungar, and M. Wolczko. *The SELF 4.0 Programmer's Reference Manual*. Sun Microsystems Inc. and Stanford University, 1995.

[All89] Apple Computer, Inc. *Macintosh Allegro Common Lisp Reference Manual, Version 1.3*, 1989.

[BC89] J.-P. Briot and P. Cointe. Programming with Explicit Metaclasses in Smalltalk-80. In *Proceedings of OOPSLA'89, New Orleans, Louisiana. Special Issue of ACM SIGPLAN Notices (24)10*, pages 419–431, 1989.

[BCCD87] Christophe Benoit, Françoise Carré, Yves Caseau, and Christophe Dony. *Lore 2.2 : manuel de référence*. Laboratoires de Marcoussis, CRCGE, March 1987.

[BD95] D. Bardou and C. Dony. Propositions pour un nouveau modèle d'objets dans les langages à prototypes. In *Actes de LMO'95, Langages et Modèles à Objets, Nancy*, pages 93–109, October 1995.

[BD96] Daniel Bardou and Christophe Dony. Split objects : a disciplined use of delegation within objects. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'96*, 31(10) :122–137, October 1996.

[BDG+88] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon. Special Issue, Common Lisp Object System Specification, X3J13 Document 88-002R. *ACM SIGPLAN Notices*, 23, 1988.

[BDM96] Daniel Bardou, Christophe Dony, and Jacques Malenfant. Comprendre et interpréter la délégation, une application aux objets morcelés. In *Actes des Journées du GDR Programmation*, Orléans, November 1996.

[BDMN73] J. Birtwistle, O.J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA begin*. Petrocelli Charter, New York, 1973.

[Ber91] P. Bergstein. Object Preserving Class Transformations. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'91*, 26(11) :299–313, 1991.

[Bez94] Jean Bezivin. Un réseau sémantique au coeur d'un AGL. In *Actes du Colloque Langages et Modèles à Objets, Grenoble*, pages 3–17, 1994.

[BGY96] J.-P. Briot, J.M. Geib, and A. Yonezawa, editors. *Object-Based Parallel and Distributed Computation*. Lecture Notes in Computer Science 1107. Springer-Verlag, Berlin, 1996.

[Bla94] G. Blaschek. *Object-Oriented Programming With Prototypes*. Springer-Verlag, Berlin, 1994.

[Boo93] G. Booch. *Object-Oriented Analysis and Design with Applications, Second Edition*. Benjamin/Cummings, 1993.

[Bor81] A.H. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4) :353–387, 1981.

[Bor86a] A. Borgida. Exceptions in Object-Oriented Languages. *ACM Sigplan Notices*, 21(10) :107–119, 1986.

[Bor86b] A. Borgida. Exceptions in Object-Oriented Languages. *ACM SIGPLAN Notices*, 21(10) :107–119, 1986.

[Bor86c] A.H. Borning. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM-IEEE Fall Joint Computer Conference, Montvale, New Jersey*, pages 36–39, 1986.

[Bor95] Isabelle Borne. *Environnement de programmation par objets : une approche pédagogique.* Thèse d'habilitation à diriger des recherches, Université de Paris-5, 1995.

[Bra83] R.J. Brachman. What IS-A Is and Isn't : An Analysis of Taxonomic Links in Semantic Networks. *Computer*, 16(10) :30–37, 1983.

[Bri84] J.-P. Briot. *Instanciation et héritage dans les langages objets.* Thèse de 3ième cycle, Université de Paris 6, 1984. Rapport LITP 85-21.

[Bri94] J.-P. Briot. Modélisation et classification de langages de programmation concurrente à objets : l'expérience Actalk. In *Actes du Colloque Langages et Modèles à Objets, Grenoble*, pages 153–165, 1994.

[BW77] D.G. Bobrow and T. Winograd. An Overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1(1) :3–46, 1977.

[Car89] B. Carré. *Méthodologie orientée objet pour la représentation des connaissances. Concepts de points de vue, de représentation multiple et évolutive d'objet.* Thèse d'université, Université des Sciences et Technologies de Lille, 1989.

[Car95] L. Cardelli. A Language With Distributed Scope. *Computing Systems*, 8(1) :27–59, 1995.

[Cas87] Y. Caseau. *Etude et réalisation d'un langage objet : LORE.* Thèse d'université, Université de Paris-Sud, 1987.

[Cas89] Yves Caseau. A model for a reflective object-oriented language. *ACM SIGPLAN Notices*, 24(4) :22–24, April 1989.

[Cas92] E. Casais. An incremental class reorganization approach. *ECOOP'92 Proceedings*, 1992.

[Cas93] Y. Caseau. Efficient handling of multiple inheritance hierarchies. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'93*, 28(10), 1993.

[Cas94] E. Casais. Automatic reorganization of object-oriented hierarchies. *Object-Oriented Systems*, 1(2) :95–115, 1994.

[Cas95] E. Casais. Managing class evolution in object-oriented systems. In O.Nierstrasz and D.Tsichritzis, editors, *Object-Oriented Software Composition*, pages 201–244. Prentice Hall, 1995.

[Cha93] C. Chambers. Predicate Classes. In O. Nierstrasz, editor, *Proceedings of Ecoop'93, Kaiserslautern, Germany. Lecture Notes in Computer Science 707*, pages 268–296, Berlin, 1993. Springer-Verlag.

[CL96] J.-B. Chen and S. C. Lee. Generation and reorganization of subtype hierarchies. *Journal of Object Oriented Programming*, 8(8), 1996.

[CM84] B. Cohen and G.L. Murphy. Models of Concepts. *Cognitive Science*, 8(1) :27–58, 1984.

[Cod88] J.J. Codani. *Microprogrammation, Architectures, Langages à Objets : NAS.* PhD thesis, Thèse de l'Université de Paris Pierre et Marie Curie, 1988.

[Coi87] P. Cointe. Metaclasses Are First Class : The ObjVlisp Model. In *Proceedings of OOPSLA'87, Orlando, Florida. Special Issue of ACM SIGPLAN Notices 22(12)*, pages 156–167, 1987.

[Cor97] Rationale Corporation. UML Notation Guide 1.1, Septembre 1997. Technical report, September 1997.

[CUCH91] C. Chambers, D. Ungar, B.W. Chang, and U. Hölzle. Parents are Shared Parts of Objects : Inheritance and Encapsulation in SELF. *LISP and Symbolic Computation*, 4(3) :207–222, 1991.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4) :471–522, 1985.

[DB96] Pierre Cointe Didier Badouel. Java : Introduction au langage, comparaison avec smalltalk et c++. In *Actes de "Objets'96", Ecole des Mines de Nantes*, 1996.

[DCC+94] Christophe Dony, Bernard Carré, Pierre Cointe, Roland Ducournau, Michel Habib, Marianne Huchard, Amedeo Napoli, Mourad Oussalah, Roger Rousseau, and Jean-Claude Royer. Rapports d'activités 1994 du pôle objet du gdr programmation. Rapport d'activité, 1994. rédacteur principal pour les activités du groupe ELO et coordonnateur du rapport du pôle.

[DCC+96] Christophe Dony, Bernard Carré, Pierre Cointe, Roland Ducournau, Michel Habib, Marianne Huchard, Amedeo Napoli, Mourad Oussalah, Roger Rousseau, and Jean-Claude Royer. Rapports d'activités 1996 du pôle objet du gdr programmation. Rapport d'activité, Edité par le CNRS, 1996. rédacteur principal pour les activités du groupe ELO et coordonnateur.

[DDHL94a] H. Dicky, C. Dony, M. Huchard, and T. Libourel. Ares, un algorithme d'ajout avec restructuration dans les hiérarchies de classes. In *Actes de LMO'94, Langages et Modèles à Objets, Grenoble*, pages 125–136, October 1994.

[DDHL94b] H. Dicky, C. Dony, M. Huchard, and T. Libourel. Ares, un algorithme d'ajout avec restructuration dans les hiérarchies de classes. Technical Report 94039, LIRMM, May 1994. version étendue de l'article LMO'94.

[DDHL95] H. Dicky, C. Dony, M. Huchard, and T. Libourel. Ares, adding a class and restructuring inheritance hierarchies. In *Actes des 11ièmes Journées Bases de Données Avancées, BDA'95*, pages 25–42, Nancy, 1995.

[DDHL96] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. *ACM Sigplan Notice - Proceedings of ACM OOPSLA'96, Object-Oriented Programming Languages, Systems and Applications*, 31(10) :251–267, October 1996. Egalement en Rapport de Recherche LIRMM No 95054, Février 1996.

[DE83] R. Reiter D. Etherington. On inheritance hierarchies with exceptions. In *Proceedings of AAAI'83*, pages 104–108, August 1983.

[DHH+95] R. Ducournau, M. Habib, M. Huchard, M.L. Mugnier, and A. Napoli. Le point sur l'héritage multiple. *Technique et science informatiques*, 14(3) :309–345, 1995.

[DHL97] C. Dony, M. Huchard, and T. Libourel. Automatic hierarchies reorganization, an algorithm and case studies with overloading. Technical Report 97279, LIRMM, 1997. Version étendue de [DDHL96] - SOUMIS À PUBLICATION.

[DMB97] C. Dony, J. Malenfant, and D. Bardou. Les langages à prototypes. In R. Ducournau, J. Euzenat, and A. Napoli, editors, *Langages et Modèles d'Objets*. INRIA - Collection Didactique, 1997. A paraître.

[DMB98] C. Dony, J. Malenfant, and D. Bardou. Classification of object-centered languages. In Ivan Moore, James Noble, and Antero Taivalsaari, editors, *Prototype-Based Object-Oriented Programming*. Springer-Verlag, 1998. A paraître.

[DMC92] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages : From a new taxonomy to constructive proposals and their validation. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'92*, 27(10) :201–217, October 1992.

[Don85] Christophe Dony. Etude des environnements de programmation lisp - implantation d'un outil graphique d'évaluation en pas-à-pas pour la machine maia. Rapport de dea, LITP - Laboratoires de Marcoussis, CRCGE, September 1985.

[Don88a] Christophe Dony. An object-oriented exception handling system for an object-oriented language. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP '88*, LNCS 322, pages 146–161, Oslo, August 15-17 1988. Springer-Verlag.

[Don88b] Christophe Dony. Une implantation réflexive du debugger de lore, utilisant le système de gestion des exceptions. Rapport de recherche No L9.0, Laboratoires de Marcoussis, CRCGE, June 1988.

[Don89a] Christophe Dony. Apports du formalisme objet aux problèmes de la gestion des exceptions. In EC2, editor, *Actes des Deuxièmes Journées Internationales : Le Génie Logiciel et ses Applications*, pages 401–416, Toulouse, France, December 1989.

[Don89b] Christophe Dony. *Langages à objets et génie logiciel. Applications à la gestion des exceptions et à l'environnement de mise au point.* Thèse d'université, Université de Paris VI, June 1989.

[Don90a] Christophe Dony. Etude critique du système de gestion des exceptions de smalltalk - objectworks v2.5. Technical report, Rapport de recherche de l'equipe RXF-LITP, June 1990.

[Don90b] Christophe Dony. Exception handling and object-oriented programming : Towards a synthesis. *ACM SIGPLAN Notices - Proceedings of the joint conference ECOOP-OOPSLA'90*, 25(10) :322–330, October 1990. Egalement disponible en Rapport de Recherche RXF-LITP No 90-101.

[Don90c] Christophe Dony. Improving exception handling with object-oriented design. In *Proceedings of IEEE COMPSAC'90, Fourteenth Computer Software and Applications Conference*, pages 36–42, Chicago, USA, November 1990.

[Don97] Christophe Dony. Prototalk : A framework for the design and the operational evaluation of prototype-based languages. Technical Report 97254, LIRMM, 1997. SOUMIS À PUBLICATION.

[DPW92] C. Dony, J. Purchase, and R. Winder. Report on the ecoop'91 workshop on exception handling and object-oriented programming. *ACM OOPS Messenger*, 3(2) :17–30, April 1992.

[Duc91] R. Ducournau. *Y3 : YAFOOL, le langage à objets, et YAFEN, l'interface graphique.* SEMA GROUP, Montrouge, 1991.

[EDQD96] Babak Esfandiari, Gilles Deflandre, Joel Quinqueton, and Christophe Dony. Agent-oriented techniques for network supervision. *Annals of Telecommunications*, 51(9-10) :521–529, 1996.

[Fer89a] J. Ferber. *Objets et agents : une étude des structures de représentation et de communications en Intelligence Artificielle.* Thèse de doctorat d'etat, Université de Paris 6, 1989.

[Fer89b] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 317–326, October 1989. Published as Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, number 10.

[Fla96] David Flanagan. *Java in a Nutshell : A Desktop Quick Reference for Java Programmers.* Nutshell handbook. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel : +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel : +1 617 354 5800, February 1996.

[FV88] J. Ferber and P. Volle. Using Coreference in Object-Oriented Representations. In *Proceedings of ECAI'88, Munich, Germany*, pages 238–240, 1988.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Massachusetts, 1995.

[GM93] R. Godin and H. Mili. Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'93*, 28(10) :394–410, 1993.

[Goo75] John B. Goodenough. Exception handling : Issues and a proposed notation. *Communications of the ACM*, 18(12) :683–696, December 1975.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, the Language and its Implementation*. Addison Wesley, Reading, Massachusetts, 1983.

[HNR97] M. Habib, L. Nourine, and O. Raynaud. A new lattice based heurisitc for taxonomy encoding. In *Actes de Kruse'97*, pages 60–81, 1997.

[IBH+79] J. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Bruecker, O. Roubine, and B. A. Wichmann. Rationale for the design of the ADA programming language. *SIGPLAN Notices*, 14(6), June 1979.

[JCP97] J.D.Ruvini, C.Dony, and P.Reitz. The apprentice and the assistant : two interface agents for smalltalk. Technical report, LIRMM, 1997. Accepté à PAAM'98 : The Third International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents - A paraître.

[JF88] Ralph E. Johnson and Brian Foot. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2) :22–35, 1988.

[JM95] Marco Jacques and Jacques Malenfant. Proto-reflex : un langage à prototypes avec réflexion de comportement. In A. Napoli, editor, *Actes du Colloque Langages et Modèles à Objets*, pages 75–89. Unité de Recherche Inria Lorraine, Nancy, 1995.

[KdB91] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[Kic96] Gregor Kiczales. Aspect-Oriented Programming. *ACM Computing Surveys*, 28(4), 1996.

[Kle91] G. Kleiber. Prototype et prototypes : encore une affaire de famille. In D. Dubois, editor, *Sémantique et cognition – Catégories, prototypes, typicalité*, pages 103–129. Editions du CNRS, Paris, 1991.

[KMMPN87a] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. Classification of Actions or Inheritance also for Methods. In *Proceedings of ECOOP'87, Paris. Lecture Notes in Computer Science 276*, pages 109–118, 1987.

[KMMPN87b] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. The BETA Programming Language. In B. Shriver and P. Wegner, editors, *Research Directions in Object Oriented Programming*, pages 7–48. MIT Press, Cambridge, Massachusetts, 1987.

[KS90] Andrew Koenig and Bjarne Stroustrup. Exception handling for C++. In USENIX, editor, *C++ Conference Proceedings, April 9–11, 1990. San Francisco, CA*, pages 149–176, Berkeley, CA, USA, April 1990. USENIX.

[Lal89] W.R. Lalonde. Designing Families of Data Types Using Examplars. *ACM Transactions on Programming Languages and Systems*, 11(2) :212–248, 1989.

[LBSL90] K. J. Lieberherr, P. Bergstein, and I. Silva-Lepe. Abstraction of object-oriented data models. *Proceedings of International Conference on Entity-Relationship*, pages 81–94, 1990.

[LBSL91] K. J. Lieberherr, P. Bergstein, and I. Silva-Lepe. From objects to classes : Algorithms for optimal object-oriented design. *Journal of Software Engineering*, pages 205–228, 1991.

[LC96] T. Ledoux and P. Cointe. Explicit metaclasses as a tool for improving the design of class libraries. *Lecture Notes in Computer Science*, 1049 :38–? ?, 1996.

[Lie81] H. Lieberman. A Preview of Act 1. AI Memo 625, Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1981.

[Lie86] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proceedings of OOPSLA'86, Portland, Oregon. Special Issue of ACM SIGPLAN Notices 21(11)*, pages 214–223, 1986.

[Lie90] H. Lieberman. *Habilitation à diriger des recherches*. PhD thesis, Université Pierre et Marie Curie, Paris VI, Institut Blaise Pascal, 1990.

[LS79] Barbara H. Liskov and Alan Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6) :546–558, November 1979.

[LTP86] W.R. LaLonde, D.A. Thomas, and J.R. Pugh. An Exemplar Based Smalltalk. In N.K. Meyrowitz, editor, *Proceedings of OOPSLA'86, Portland, Oregon. Special Issue of ACM SIGPLAN Notices 21(11)*, pages 322–330, 1986.

[Mae87] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87, Orlando, Florida. Special Issue of ACM SIGPLAN Notices 22(12)*, pages 147–155, 1987.

[Mal95] J. Malenfant. On the Semantic Diversity of Delegation-Based Programming Languages. In *Proceedings of OOPSLA'95, Austin, Texas. Special Issue of ACM SIGPLAN Notices (30)10*, pages 215–230, 1995.

[Mal96] J. Malenfant. Abstraction et encapsulation en programmation par prototypes. *Technique et science informatiques*, 15(6) :709–734, 1996.

[MC93] P. Mulet and P. Cointe. Definition of a Reflective Kernel for a Prototype-Based Language. In *Proceedings of the 1st JSSST International Symposium on Object Technologies for Advanced Software, ? ?. Lecture Notes in Computer Science 742*, pages 128–144, 1993.

[MCDM91] J. Malenfant, P. Cointe, C. Dony, and P. Mulet. Reflection in prototype-based object-oriented programming languages. In *Proceedings of OOPSLA'91 workshop on reflection and meta-level architectures*, October 1991.

[MCDM92] J. Malenfant, P. Cointe, C. Dony, and P. Mulet. Etude de la réflexion de comportement dans le langage self. In *Actes de RPO'92, Représentation par objets : le point sur la recherche et les applications, La Grande-Motte*, page ? ?, June 1992.

[MDC92] J. Malenfant, C. Dony, and P. Cointe. Behavioral reflection in a prototype-based language. In Akinori Yonezawa and Brian. C. Smith, editors, *Proceedings of International Workshop on ¡¡ New Models for Software Architecture : Reflection and Meta-Level Architectures ¿¿*, pages 143–153, Tokyo, Japan, November 1992. ACM Sigplan, JSSST, IPJS.

[MDC96] Jacques Malenfant, Christophe Dony, and Pierre Cointe. A semantic of introspection in a prototype-based language. *Kluwert International Journal on Lisp And Symbolic Computation*, 9(2/3) :153–179, 1996.

[Mey90] B. Meyer. *Conception et programmation par objets, pour du logiciel de qualité*. InterEditions, Paris, 1990.

[Mey92] B. Meyer. *Eiffel : The Language*. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, United Kingdom, 1992.

[MGD+90] B.A. Myers, D.A. Giuse, R.B. Dannenberg, B. Van der Zanden, D. Kosbie, E. Previn, A. Mickish, and P. Marchal. Garnet : Comprehensive Support for Graphical Highly-Interactive User Interfaces. *IEEE Computer*, 23(11) :71–85, 1990.

[MGdZ92] B.A. Myers, D.A. Giuse, and B. Van der Zanden. Declarative Programming in a Prototype-Instance System : Object-Oriented Programming Without Writing Methods. In A. Paepcke, editor, *Proceedings of OOPSLA'92, Vancouver, Canada. Special Issue of ACM SIGPLAN Notices (27)10*, pages 184–200, 1992.

[Min75] M. Minsky. A Framework for Representing Knowledge. In P. Winston, editor, *The Psychology of Computer Vision*, pages 211–281. McGraw-Hill, New York, 1975.

[MMS81] J. G. Mitchell, W. Maybury, and R. Sweet. Mesa language manual. Technical Report CSL-78-1, Xerox, Palo Alto, CA, February 1981.

[MNC+89] G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Les langages à objets*. InterEditions, Paris, 1989.

[Moo95] Ivan Moore. Guru - A Tool for Automatic Restructuring of Self Inheritance Hierarchies. *TOOLS USA 1995 Proceedings, Prentice-Hall*, 1995.

[Moo96] Ivan Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'96*, 1996.

[MRU90] O. Mariño, F. Rechenmann, and P. Uvietta. Multiple Perspectives and Classification Mechanism in Object-Oriented Representation. In *Proceedings of ECAI'90, Stockholm, Sweden*, pages 425–430, 1990.

[MS89] M. Missikoff and M. Scholl. An Algorithm for Insertion into a Lattice : Application to Type Classification. *Proc. 3rd Int. Conf. FODD'89*, pages 64–82, 1989.

[Mul95] Philippe Mulet. *Réflexion & langages à prototypes*. Thèse d'université, Université de Nantes, 1995.

[Nix83] B.A. Nixon. A Taxis Compiler. Technical Report 33, Comp. Sci. Dept., Univ. of Toronto, 1983.

[NM95] H. Naja and N. Mouaddib. Un modèle pour la représentation multiple dans les bases de données orientées-objet. In A. Napoli, editor, *Actes de LMO'95*, pages 173–189. Unité de Recherche Inria Lorraine, Nancy, 1995.

[NT95] Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice Hall, 1995.

[OKH+95] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings OOPSLA '95, ACM SIGPLAN Notices*, pages 235–250, October 1995. Published as Proceedings OOPSLA '95, ACM SIGPLAN Notices, volume 30, number 10.

[Par90] ParcPlace Systems. *Objectworks Smalltalk Reference Manual -Exception Handling*, 1990.

[Pit88] K. Pitman. Error/Condition Handling. Contribution to WG16. Revision 18.Propositions pour ISO-LISP. Technical Report 22/WG 16N15, AFNOR, ISO/IEC JTC1/SC, 1988.

[PW90] Jan A. Purchase and Russel L. Winder. Message pattern specifications : A new technique for handling errors in parallel object oriented systems. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, pages 116–125, October 1990. Published as Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices, volume 25, number 10.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Objekt-oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

[RD97] Jean-David Ruvini and Christophe Dony. Ébauche de deux agents interfaces pour l'environnement smalltalk-80. In *Actes des Cinquièmes journées francophones sur l'intelligence artificielle distribuée et les systèmes multi-agents - Session Poster*, La colle-sur-Loup, April 1997.

[Rec88] F. Rechenmann. *SHIRKA : système de gestion de bases de connaissances centrées-objet. Manuel de référence*. INRIA/ARTEMIS, Grenoble, 1988. ftp ://ftp.inrialpes.fr/pub/sherpa/rapports/manuel-shirka.ps.gz.

[RG77] R.B. Roberts and I.P. Goldstein. The FRL Manual. AI Memo 409, Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts, 1977.

[Run92] E. A. Rundensteiner. A Class Classification Algorithm For Supporting Consistent Object Views. Technical report, University of Michigan, 1992.

[SLU89] L.A. Stein, H. Lieberman, and D. Ungar. A Shared View of Sharing : The Treaty of Orlando. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 31–48. ACM Press/Addison-Wesley, Reading, Massachusetts, 1989.

[Smi94] W.R. Smith. The Newton Application Architecture. In *Proceedings of the 39th IEEE Computer Society International Conference, San Francisco, California*, pages 156–161, 1994.

[Smi95] W.R. Smith. Using a Prototype-Based Language for User Interface : The Newton Project's Experience. In *Proceedings of OOPSLA'95, Austin, Texas. Special Issue of ACM SIGPLAN Notices (30)10*, pages 61–72, 1995.

[Ste87] L.A. Stein. Delegation IS Inheritance. In N.K. Meyrowitz, editor, *Proceedings of OOPSLA'87, Orlando, Florida. Special Issue of ACM SIGPLAN Notices 22(12)*, pages 138–146, 1987.

[Ste90] Guy L. Steele Jr. *Common Lisp - The Language*. Digital Press and Prentice-Hall, 12 Crosby Drive, Bedford, MA 01730, USA and Englewood Cliffs, NJ 07632, USA, second edition, 1990.

[Ste94] P. Steyaert. *Open Design of Object-Oriented Languages. A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel, Belgium, 1994.

[SU95] R.B. Smith and D. Ungar. Programming as an Experience : The Inspiration for Self. In Walter Olthoff, editor, *Proceedings of ECOOP'95, Aarhus, Denmark. Lecture Notes in Computer Science 952*, pages 303–330, Berlin, 1995. Springer-Verlag.

[Tai91] A. Taivalsaari. Cloning Is Inheritance. Computer Science Report WP-18, University of Jyväskylä, Finland, 1991.

[Tai93] A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming*. PhD thesis, University of Jyväskylä, Finland, 1993.

[UCCH91] D. Ungar, C. Chambers, B.W. Chang, and U. Hølzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3) :223–242, 1991.

[US87] D. Ungar and R.B. Smith. Self : The Power of Simplicity. In N.K. Meyrowitz, editor, *Proceedings of OOPSLA'87, Orlando, Florida. Special Issue of ACM SIGPLAN Notices 22(12)*, pages 227–242, 1987. Also published in Lisp and Symbolic Computation 4(3), Kluwer Academic Publishers, pages 187–205, 1991.

[Van94] G. Vanwormhoudt. Points de vue, représentation multiple et évolutive en Smalltalk. Rapport interne, Laboratoire d'Informatique Fondamentale de Lille, 1994.

[VBL97] G. Vanwormhoudt, B.Carré, and L.Debrauwer. Programmation par objets et contextes fonctionnels. Application de CROME à Smalltalk. In Serge Garlatti Roland Ducournau, editor, *Actes de LMO'97*, pages 223–236, 1997.

[Win79] Terry Winograd. Beyond programming languages. *Communications of the ACM*, 7(22) :391–401, July 1979.

[WM81] D. Weinreb and D. Moon. *Lisp Machine Manual*. Symbolics Inc., ? ?, 1981.

Chapitre 9

Article relatif à la gestion des exceptions.

– [Don90b] Christophe Dony.
Exception handling and object-oriented programming : Towards a synthesis.
ACM SIGPLAN Notices - Proceedings of the joint conference ECOOP-OOPSLA'90, 25(10) :322–330, Octobre 1990.
Un système de gestion des exception développé pour et avec la programmation par objets. Appliqué à et implanté en SMALLTALK.

Chapitre 10

Articles relatifs à la programmation sans classes

- [DMC92] Christophe Dony, Jacques Malenfant, and Pierre Cointe.
  Prototype-based languages : From a new taxonomy to constructive proposals and their validation.
  Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'92, 27(10) :201–217, October 1992.
  Cet article est le résultat de note première étude sur les langages à prototypes, elle pose les problèmes sur lesquels nous avons travaillé par la suite.
- [BD96] Daniel Bardou, Christophe Dony.
  Split objects : a disciplined use of delegation within objects.
  ACM Sigplan Notice - Proceedings of ACM OOPSLA'96, 31(10) :122–137, Octobre 1996.
  Compréhension du mécanisme de délégation, des diverses interprétations associées au lien de délégation, de la différence entre délégation et héritage classique. Applications de la délégation à la gestion de points de vues.
  applications
- [Don97] C. Dony.
  Prototalk : A framework for the design and the operational evaluation of prototype-based languages.
  Rapport de recherche LIRMM, Novembre 1997. Soumis à publication.
  Un article nouveau, que j'aurais du écrire depuis un bon moment, qui décrit l'architecture de la plate-forme Prototalk de simulation de langages à prototypes. Cette architecture a évoluée depuis l'article [DMC92]. Les langages à prototypes ne servent que de support à cet article dont le sujet est réalisation d'un framework pour l'implantation rapide d'interprètes par réutilisation de code.

Chapitre 11

Article relatif à la réorganisation de hiérarchies

– [DHL97] C.Dony, M.Huchard, and T.Libourel.
Automatic hierarchies reorganization, an algorithm and case studies with overloading.
Rapport de recherche LIRMM, Soumis à publication.
Article décrivant le problème de l'insertion automatique d'une classe dans une hiérarchie d'héritage et la version actuelle de l'algorithme Ares. Version longue de [DDHL96] incluant des études de cas.

Chapitre 12

Article relatif aux systèmes réflexifs

– [MDC92]
Jacques Malenfant, Christophe Dony, and Pierre Cointe.
A semantic of introspection in a prototype-based language.
Kluwert International Journal on Lisp And Symbolic Computation, 9(2/3) :153–179, 1996.
L'article [MDC92] présente un système à méta-objets permettant aux programmeurs de paramétrer, dans un langage à prototypes, l'envoi de message et l'application des méthodes. Il démontre également l'arrêt des diverses régressions infinies induites par la représentation réflexive. Cet article reprend ces résultats et propose une nouvelle version de la démonstration dans un cadre formel basé sur les systèmes de réécriture.