Université de Montréal

Modélisation de langages à classes à l'aide de Proto-Reflex

par

Katia Montero

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès Sciences (M.Sc.)
en informatique

Mars 1998

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Modélisation de langages à classes à l'aide de Proto-Reflex

présenté par:

Katia Montero

a été évalué par un jury composé des personnes suivantes:

[redacted] (président-rapporteur)
Jacques Malenfant (directeur de recherche)
Guy Lapalme (co-directeur de recherche)
[redacted] (membre du jury)

Mémoire accepté le 21.05.1998

Sommaire

Ce mémoire comprend une étude comparative des modélisations de trois langages à classes à l'aide de Proto-Réflex, un langage à prototypes réflexif.

En profitant des caractéristiques distinctives à chaque langage, chacune des modélisations a été construite en fonction des aspects particuliers de chaque langage, de façon à les rendre plus évidentes à une analyse ultérieure.

Dans ObjVLisp, nous étions intéressés à conserver l'homogénéité du processus de création des objets du système, tout en gardant avec les prototypes, la structure des méta-classes, des classes, des instances et des liens entre celles-ci.

Pour CLOS, les aspects à étudier sont reliés à la structure du modèle, tant à la structure de ses entités qu'à la définition des méta-structures de programmation capables de modifier dynamiquement le comportement du système. Dans le premier cas, nous nous sommes intéressés à modéliser les méta-objets (méta-classes, classes, méthodes, etc) et son héritage multiple. Une attention spéciale a été fait à la modélisation du mécanisme de communication qui, à la différence de la plupart des langages à objets, n'appelle pas la fonction polymorphique d'envoi de messages, mais des fonctions génériques. Dans le deuxième cas, à cause de sa complexité qui tombe hors des limites de ce travail, nous n'avons fait qu'une brève comparaison entre le mécanisme de réflexion et le protocole des méta-objets de CLOS.

Pour la modélisation du troisième langage, un sous-ensemble minimal de Java, nous avons concentré notre attention sur le mécanisme d'héritage et la structure de

ses entités. D'autres caractéristiques ont été considerées sommairement, comme les capacités réflexives du langage et sa modularité.

Les trois expériences nous ont donné des éléments pour, tout d'abord, valider l'utilisation de Proto Reflex dans le but de modéliser les noyaux minimaux des trois langages à classes. Ensuite, la comparaison des modélisations a mis en évidence certaines lignes directrices de conception des langages, comme par exemple, le rôle qui jouent les mécanismes de communication et d'héritage dans les systèmes. Finalement, cette étude nous a permis d'explorer les frontières partagées par les modèles des langages à classes et des langages à prototypes.

Table des matières

Table des figures

Remerciements

Tout d'abord j'aimerais remercier la personne sans laquelle je n'aurais jamais terminé ce mémoire: M. Guy Lapalme. Son encouragement, intérêt et appui ont été déterminants pour finir ce travail. Je me souviendrai longtemps de son support moral et de sa générosité à mon égard.

Je dois aussi remercier Marco Jacques qui, en plus de me donner l'accès à l'interprète de Proto Reflex, a toujours été disponible à mes fréquentes demandes de "consultation" supposément courtes. Il a aussi beaucoup contribué à mes recherches bibliographiques au cours de ma maîtrise.

De même, je suis reconnaissante envers M. Jacques Malenfant pour la conceptualisation du sujet de recherche, ainsi comme pour avoir accepté la direction de ce mémoire.

Je réserve une affection spéciale à François-Nicola Demers qui m'a guidé tant vers la rigueur scientifique qu'à travers les merveilleux chemins du Québec. Son amitié et son aide resteront toujours présents en moi.

J'aimerais aussi exprimer mon amitié et ma gratitude à l'égard des copains du laboratoire Incognito, qui tout au long de mon séjour à Montréal m'ont aidé à me sentir membre d'une énorme famille en ayant comme intérêt commun la joie d'apprendre.

Il me reste seulement à remercier à Mme Lucie Duranceau du Ministère de l'Éducation pour son support et sa confiance inébranlables.

À George

MATHEMATICIANS hunt elephants by going to Africa, throwing out everything that is not an elephant, and catching one of whatever is left.

EXPERIENCED MATHEMATICIANS will attempt to prove the existence of at least one unique elephant before proceeding to previous step as a subordinate exercise.

PROFESSORS OF MATHEMATICS will prove the existence of at least one unique elephant and then leave the detection and capture of an actual elephant as an exercise for their graduate students.

COMPUTER SCIENTISTS hunt elephants by exercising Algorithm A:

1. Go to the Cape of Good Hope.
2. Work northward in an orderly manner, traversing the continent alternately east and west.
3. During each traverse pass:
   - Catch each animal seen.
   - Compare each animal caught to a known elephant.
   - Stop when a match is detected.

EXPERIENCED COMPUTER PROGRAMMERS modify Algorithm A by placing a known elephant (named *Clyde*) in Cairo to ensure that the algorithm will terminate.

ASSEMBLY LANGUAGE PROGRAMMERS prefer to execute Algorithm A on their hands and knees.

HARDWARE SALES PEOPLE catch rabbits, paint them gray, and sell them (very expensive) as desktop elephants.

Chapitre 1

Introduction

Dès que le concept des langages à prototypes a été proposé comme une alternative aux langages à classes, un ensemble de questions pour comparer les deux modèles se sont posées. Des études de simulation des caractéristiques des langages à classes à l'aide des langages sans classe [Ste87, Bor86, LaL89, MGZ92] ont exploré les frontières et les similitudes entre les deux modèles.

La modélisation de certains mécanismes comme celui de l'héritage n'a pas seulement montré qu'il y avait des variantes d'implantation de solutions (dépendantes du point de vue et de la philosophie de l'implanteur), mais aussi que ces autres modalités ont mis en évidence d'autres styles de programmation, comme la programmation par méta-classes explicites en ObjVLisp ou la programmation à l'aide de protocoles pour manipuler des méta-objets en CLOS.

Après ces premières approches, d'autres éléments dans la modélisation des langages à objets ont attiré l'attention des chercheurs: communication entre entités, homogénéité des objets, hiérarchisation de la structure des objets, propriétés des objets et sa dynamique, amalgame des langages à classes et langages à prototypes, etc. Même le coût d'implantation des projets des langages à prototypes versus celui des langages à classes a été considéré.

Dans cette étude, nous analyserons les caractéristiques principales de trois langages à classes. Le choix de langages entraîne la recherche de caractéristiques appropriées dans le langage qui doit être un langage à objets avec un noyau simple à modéliser et un ensemble minimal de primitives.

Nous avons choisi trois langages qui ont des classes pour grouper le comportement des instances. Dans les trois, c'est le mécanisme d'héritage qui permet le partage des connaissances entre entités.

Le premier sera ObjVLisp, un langage à classes développé par Pierre Cointe en 1987. Dans ce langage, l'homogénisation du processus de création des méta-classes, classes et instances rend visible la programmation par méta-classes explicites [Coi87].

Common Lisp Object System (CLOS) [KdRB91] est le deuxième langage étudié. Developpé à Xerox Parc dans les années 80s, CLOS représente un effort de standarisation des langages de la famille Lisp enrichi avec le paradigme de programmation par objets. L'introduction des métaprotocoles dans CLOS le rend suffisamment attirant pour la présente étude. Nous allons modéliser CLOS sous la forme de *Closette.* Closette, aussi connu comme *Tiny CLOS* est un interprète simplifié de CLOS. Malgré ses simplifications, Closette est représentatif de l'architecture de CLOS et de ses implémentations.

Finalement Java [AG96, Sun95], un langage à classes de création récente par *Sun Microsystems Computer Corporation* a été choisi pour sa diffusion croissante et son utilisation répandue dans la communauté informatique et son application au World Wide Web.

Nous modéliserons le noyau minimal de chacun des langages à l'aide de Proto-Reflex, un langage à prototypes developpé à l'Université de Montréal par Marco Jacques [Jac95]. Proto-Reflex est un langage avec réflexion de comportement basé sur le modèle conceptuel présenté par Malenfant, Dony et Cointe [MDC92]. Proto-Reflex est un langage robuste qui conjugue la puissance de la programmation orientée

objets et la flexibilité du mécanisme de réflexion. Une raison de ce choix de Proto-Reflex a été de mettre en relief ses qualités et d'y attirer l'attention d'autres usagers. Un autre intérêt pour l'équipe REFLEX au sein de laquelle j'œuvre, est le fait que je suis la première utilisatrice de Proto-Reflex en dehors des concepteurs et implanteurs.

Notre tâche sera de modéliser la structure des systèmes à objets avec des classes à l'aide des objets sans classes, c'est-à-dire, des prototypes. Aussi, nous allons modéliser les différents mécanismes d'héritage dans les langages à classes à l'aide du mécanisme de délégation et de passage de messages à la manière des langages à prototypes. Cela va nous permettre de montrer comment les langages à classes peuvent être simulés à l'aide des prototypes en utilisant le passage de message entre entités comme mécanisme principal de communication.

Pour finaliser notre étude, nous allons comparer ces trois modèles pour en tirer des conclusions sur trois aspects essentiels:

1. traitement d'entités, comme prototypes, classes, méta-classes, superclasses et instances.

2. mécanismes de communication (passage des messages).

3. mécanismes de partage d'information (héritage et délégation)

Le noyau minimal de chaque langage sera étendu selon les besoins pour montrer les comportements généraux des langages. Certaines structures spécifiques de programmation seront conservées au fur et à mesure qu'ils mettent en relief certains comportements particuliers de chaque langage.

Ainsi, l'ensemble de fonctions de chaque langage contiendra des fonctions générales nécessaires pour écrire des programmes, mais aussi un petit sous-ensemble propre à ses particularités expressives.

La comparaison des résultats entre les trois expériences nous donnera des éléments

pour évaluer la performance des langages à prototypes comme "langages assembleurs", c'est-à-dire, des langages pour exprimer tous les concepts des langages à classes.

Dans l'approche des modèles à classes, les détenteurs du comportement des objets sont les classes. Lorsqu'une instance reçoit un message, elle communique avec sa classe pour commencer la recherche du comportement correspondant au message. La réaction peut se résumer de la façon suivante: l'objet suit le lien vers sa classe et là, il essaie d'identifier la méthode à travers son nom. S'il ne la trouve pas, il suit le lien de la classe vers sa superclasse et essaie d'identifier la méthode. Le processus se repète en remontant vers les superclasses. Ce processus est implanté implicitement et ne peut être modifié par l'usager.

Dans le cas du modèle à objets sans classes, c'est-à-dire dans les langages à prototypes, ce sont les objets, eux-mêmes, qui contiennent leur comportement dans leur structure. Quand le message arrive au prototype, il y a un mécanisme implicite pour trouver la méthode demandée. Le mécanisme dicte de suivre les liens de son parent jusqu'à l'objet qui possède la méthode. Comme dans le modèle à classes, l'objet monte vers ses ancêtres jusqu'à l'objet racine.

Chacun des modèles applique sa méthodologie de recherche pour trouver ce qui est demandé dans la communication, mais cette méthodologie est impossible à changer et à redéfinir.

On pourrait voir la programmation à objets comme un paradigme de programmation où les instructions sont donnés aux objets qui ont la tâche de trouver les moyens de les appliquer. Alors, on peut considérer deux niveaux dans le modèle général de programmation à objets: le premier contient des objets, unités encapsulés capables de répondre aux messages, et l'autre, l'ensemble des instructions de contrôle qui définissent comment les messages vont être traités. Le niveau des objets est statique, c'est-à-dire, les objets dans la programmation à objets sont des entités passives qui deviennent actives seulement au moment de la réception des messages.

Les mécanismes de recherche sont rigides parce qu'ils ont été codés dans la structure même du système, inaccessibles à l'usager. La structure rigide des deux modèles impose implicitement une infrastructure aux objets et détermine sa fonctionnalité. L'impossibilité de changer cette structure est liée au fait qu'elle est l'engin principal du processus de communication dans le langage. Pourtant, ses arguments peuvent être rendus accessibles au programmeur qui pourrait donc avoir du contrôle dans cet aspect. Nous allons explorer les possibilités de rendre disponible ce type du contôle au usager au moment de l'analyse du langage CLOS.

Dans le processus de modélisation des langages à classes à l'aide d'un langage à protoypes, la structure des objets liés aux classes doit être conservée, ainsi comme les rôles que les autres entités du système (classes, méta-classes) jouaient. D'une partie nous sommes intéresés à modéliser la structure des objets à classes; de l'autre partie, nous voulons profiter de la modélisation du mécanismes de contrôle de messages pour étudier et explorer d'autres comportements et réponses possibles dans les systèmes. À cette fin, on doit procéder en étapes:

1. Homogénéiser tous les objets du système. Nous allons modéliser tous les objets comme des prototypes qui contiennent des données et des méthodes. Les liens des objets à classes seront modélisés à l'aide des champs spécials dans les prototypes.

2. Établir les fonctions de contrôle de passage des messages à l'aide des éléments de Proto-Reflex qui simulent le comportement caractéristique dans le langage à classes à modéliser.

3. Modéliser le langage dans le reste de ses aspects et explorer la souplesse de son structure et comportement: types d'héritage, des entités, traitement de messages, ses capacités de changer son comportement (réflexivement ou non), etc.

L'information et les résultats que chaque modélisation nous donnent seront la source pour en tirer des conclusions à propos de la conception générale des langages à objets, sur les mécanismes de communication entre entités et les différences des comportements des objets dans les langages à classes et à prototypes.

Chapitre 2

Généralités sur les langages à objets.

L'approche à objets considère que le plus petit élément qui peut contenir des données et des instructions pour manipuler ces données, est une unité appelée *objet.* Le fait de réunir dans un seul élément de l'information et des opérations sur cette information se définit comme *l'encapsulation.* Dans l'ensemble des données et des fonctions encapsulées dans l'unité, il y en a certaines qui peuvent être appelées de l'extérieur et certaines qui font partie des opérations internes. La différenciation des types de données et des opérations entre publiques et privés introduit le concept *d'interface publique* dans l'objet. Chaque objet est une unité qui a une interface de communication et un ensemble de données. Ces données peuvent se trouver dans l'objet soit sous la forme d'information statique (des valeurs associées aux variables), soit sous la forme d'information dynamique: des opérations à réaliser, généralement appelées *méthodes.* L'abstraction du concept d'objet nécessite un mécanisme de communication pour accéder aux données. La façon de communiquer avec un objet, c'est le *passage des messages.* Un message doit exprimer le receveur, l'opération demandée et les paramètres nécessaires. À son tour, l'objet répond suivant ses instructions. L'ensemble

des messages auxquels l'objet peut répondre se nomme *comportement.*

L'objet peut aussi diriger des messages sur lui-même. Ces actions ont pour but d'accéder aux fonctions internes, de mettre au point ses propres structures, ou de répondre aux opérations d'accès public.

Même si chaque objet a un ensemble différent de données, son comportement peut être semblable à celui d'autres objets. Aussi, ils peuvent avoir besoin de la même méthode ou d'un ensemble determiné. Dans ces cas, nous parlons d'objets de la même classe. Une classe est un cadre qui regroupe des comportements venant d'objets semblables et comme elle comporte un ensemble général de méthodes qui caractérisent des objets comme un groupe. La figure 2.1 montre un exemple de classes, sous-classes et instances. On peut aussi penser à une classe comme une usine à objets. Dans ce cas, on appelle *instance* un objet fabriqué selon la structure dictée par la classe. Toutes les instances d'une classe se comportent de la même façon selon l'ensemble de comportements détenus par sa classe.

L'héritage dans les langages à classes est un mécanisme par lequel une classe définit la structure des données de ses instances comme un sous-ensemble de la définition d'une autre classe ou de plusieurs. Bien sûr, nous pouvons nous servir du mécanisme d'héritage pour la définition des taxonomies de classes et créer des classifications.

En général, l'héritage nous permet de raffiner les propriétés d'une classe en en prenant une autre comme base et en ajoutant différents comportements et données au fur et à mesure des besoins. Cette caractéristique de "producteur de classes" aide à ce que l'héritage soit une des techniques utilisées et préférées pour la ré-utilisation de code.

Le concept d'héritage nous amène au concept de sous-classe. Une sous-classe est une classe qui hérite du comportement d'une autre classe. De même, une superclasse est une classe de laquelle héritent une ou plusieurs classes.

Il y a différents types d'héritage selon la façon de modifier le comportement d'une

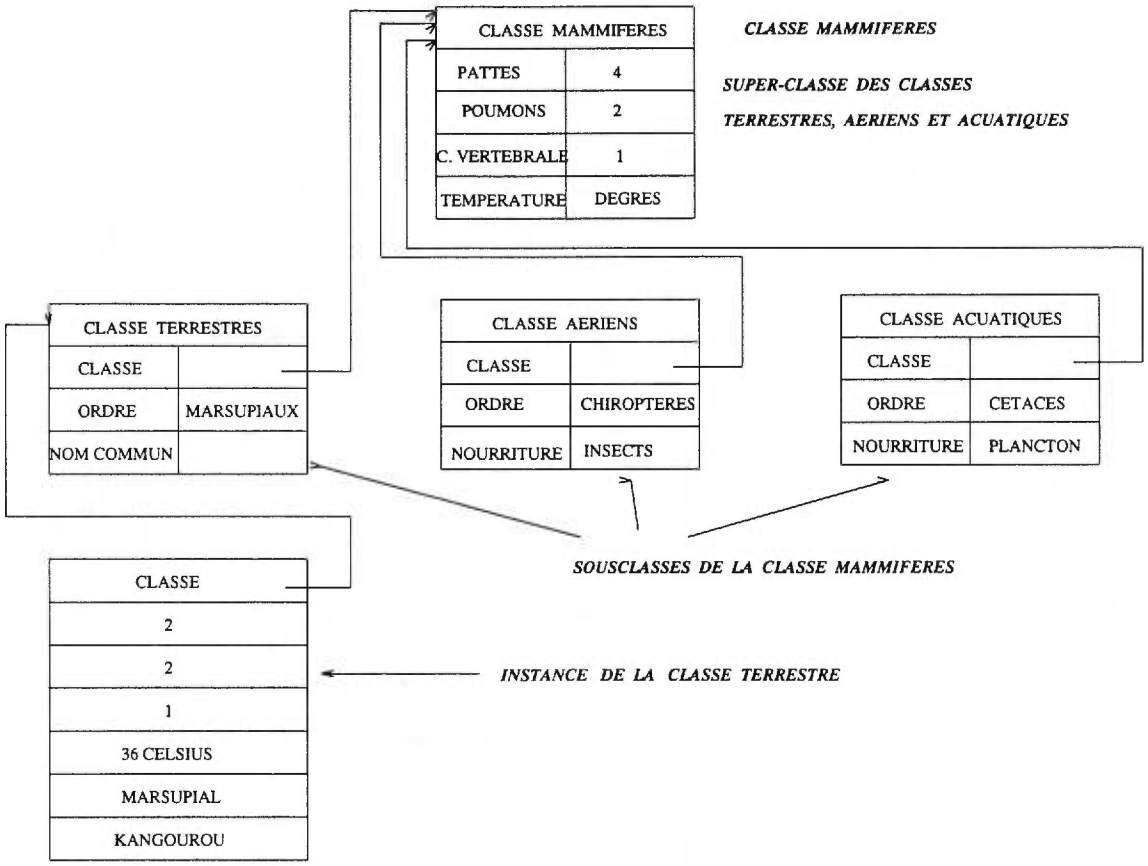

FIG. 2.1 - *Classes, sous-classes et instances*

classe fille à partir des méthodes qu'elle hérite d'une classe mère ou ancêtre.

La plus générale est celle où une classe est un cas particulier de la classe parent. La nouvelle classe a accès aux données et aux méthodes de la classe parent et donc spécialise un aspect de son comportement. Dans cette situation idéale on peut substituer la nouvelle instance à une instance de la classe parent, selon le *principe de substituabilité* (*"principle of substitutability"* [Bud97]).

Le *principe de substituabilité* nous dit que si on a deux classes A et B, B étant une sous-classe de A, il est possible de substituer les instances de la classe B à des instances de la classe A dans toute situation et sans effets observables.

Les formes d'héritage qu'une classe peut présenter sont les suivantes, (tiré de [Bud97]):

spécialisation La nouvelle classe est un cas spécial de la classe parent mais qui satisfait toutes les spécifications de la classe d'origine. Cette sorte d'héritage est aussi nommée *sous-type* (*"subtype"*) et suit le principe de substituabilité.

spécification La classe parent définit des comportements qui sont impleméntés dans la nouvelle classe mais non pas dans la classe parent.

construction La classe fille utilise le comportement de la classe parent mais elle n'est pas un sous-type de la classe parent, donc elle ne suit pas le principe de substituabilité.

généralisation La nouvelle classe modifie ou corrige quelques méthodes de la classe parent.

par extension La classe fille ajoute une nouvelle fonctionalité à la classe parent mais elle ne change aucun comportement hérité.

limitation La nouvelle classe restreint l'utilisation de quelques méthodes de la classe parent.

variance La classe parent et la classe fille sont variantes l'une de l'autre et la relation entre classe et sous-classe est arbitraire.

combinaison La classe fille hérite de plus d'une classe parent; ce type d'héritage est mieux connu sous le nom *d'héritage multiple.*

Les *classes abstraites* sont des classes spéciales qui n'ont pas nécessairement d'instances. Elles regroupent des comportements communs à un ensemble de classes et leur but est de définir une collection de méthodes générales ou "de base" pour un type de classes. Le concept de classes abstraites nous amène au concept de méta-classe. La méta-classe est une "classe de classes". Puisque certains langages donnent aux classes le statut d'objet, les classes devient alors instances d'un modèle, qui est sa méta-classe.

2.1 Langages à classes et langages à prototypes.

Le modèle à prototypes a été introduit par H. Lieberman [Lie86] comme alternative au modèle à classes. Les objets, appelés prototypes, gardent l'encapsulation caractéristique des unités des langages à objets, mais ils ne sont pas liés à aucun objet décrivant leur structure. Les langages à prototypes n'ont pas des classes qui définissent des comportements, structures, ou des aspects abstraits pour des ensembles des objets. Le comportement est encapsulé dans chaque prototype, ce qui lui permet de répondre aux messages qui lui sont envoyés. Le mécanisme de partage de comportements communs se nomme *délégation.* Si le prototype ne peut répondre au message réçu, il en *délègue* le traitement aux autres objets, appelés ses *objets-parents*, avec qui le prototype a des liens de délégation. La figure 2.1 montre trois prototypes et ses liens. La délégation est aussi la manière de définir incrémentalement le comportement des objets.

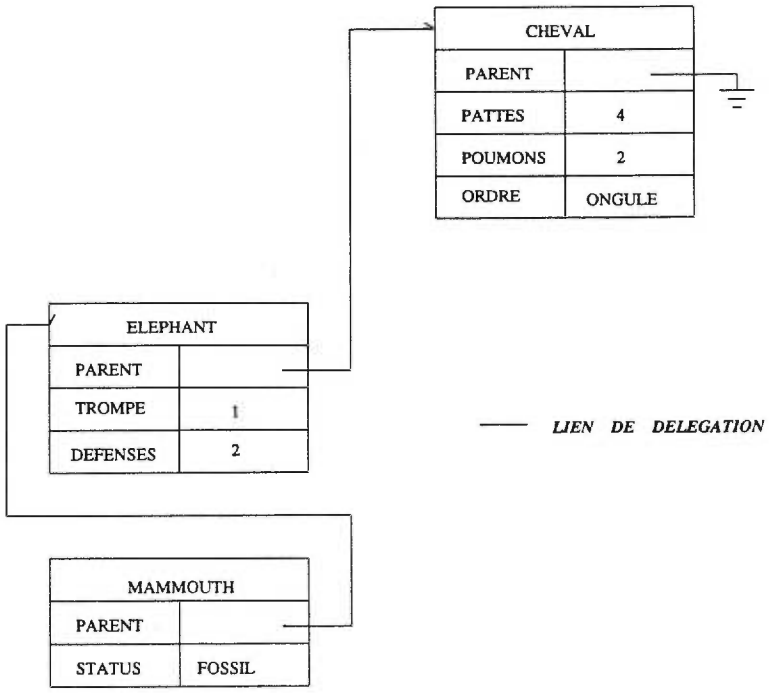

FIG. 2.2 - *Prototypes et lien de délégation*

La façon de générer un objet en copiant un autre déjà existant pour lui modifier et ajuster postérieurement au besoin est connue sous le nom de *clonage*. Le clonage n'est pas la seule manière de générer des nouveaux prototypes. Une autre alternative est la création *ex nihilo* (c'est-à-dire, à partir de rien). Cette alternative est nécessaire lorsque le nouvel objet représente un concept neuf, absent dans l'ensemble des prototypes.

Voici quelques variantes de création ex nihilo de prototypes:

- créer des prototypes vides, c'est-à-dire sans aucun champ, et fournir des fonctions primitives pour ajouter *a posteriori* les champs et ses valeurs.
- créer des prototypes avec des champs initiaux mais ne pas permettre de modifications à la structure, seulement aux valeurs.
- créer des prototypes avec des champs initiaux et permettre des modifications *a posteriori*.

Comme dans le modèle à classes, dans les langages à prototypes les objets communiquent à l'aide de messages. Si le receveur du message est incapable d'y répondre, il va communiquer avec son objet-parent pour chercher la méthode demandée et, de cette façon, déléguer la réaction à l'ancêtre qui la possède. Ainsi, on dit que les prototypes "héritent" de ses ancêtres un ensemble de méthodes qui définit leurs comportements.

La délégation est un mécanisme d'extension et de raffinement des propriétés dans lequel les nouveaux prototypes deviennent une "partie" des prototypes parents [UCCH91b, Mal95].

Dans les langages à classes, l'héritage permet le partage d'information mais toujours au niveau de l'abstraction. En effet, ce qui est partagé est une description des objets dont les valeurs seront assignées aux instances.

Ceci est particulièrement vrai en ce qui concerne la structure commune des nouveaux objets. Au moment de la création des nouvelles instances, la structure commune aux instances est remplie avec les valeurs qui vont constituer le corps de l'instance. Les instances ne partagent qu'une description abstraite et non des valeurs.

Par contre, dans les langages à prototypes il n'existe pas de descriptions abstraites à partager. Ce sont les valeurs mêmes des champs que les prototypes partagent. Le nouvel objet est généré comme une copie d'un prototype existant et ses champs sont (au moins au début) les mêmes que l'objet copié. Les valeurs de ses champs resteront identiques jusqu'au moment de la mise à jour, moment où la différenciation des prototypes commence.

En résumé, dans les langages à classes le partage d'information est fait au niveau abstrait alors que dans les langages à prototypes il est fait au niveau concret.

Malgré des différences de niveau où est fait le partage d'information, l'héritage et la délégation, les deux, utilisent des incréments différentiels pour enrichir le comportement des nouveaux objets.

Nous pensons que les relations d'héritage entre une classe et une sous-classe dé-

crites à la section précédente (de spécialisation, spécification, d'extension, etc) ont une *équivalence* dans le mécanisme de délégation dans le modèle à prototypes. Dans cette étude, au moment de l'analyse de chaque modélisation, nous allons explorer succinctement la capacité de chaque implémentation d'incorporer certains de ces relations.

2.2 Communication entre entités.

Bien que dans les langages à classes et les langages à prototypes la communication entre entités soit faite à l'aide des messages, ceux-ci ne sont pas traités de la même façon dans les deux modèles. Dans les langages à classes, le message est traité par la classe de l'objet receveur alors que dans les langages à prototypes l'objet lui-même répond au message. Cette autonomie d'action apporte au modèle une richesse expressive qui complique toutefois les relations entre les objets.

Nous sommes interessés à essayer des variations sur le mécanisme général de passage de messages pour étudier son comportement. Dans nos modélisations, l'emphase sera mise sur le processus de communication entre entités. Parmi d'autres, nous nous intéresserons aux avantages et désavantages de l'application des *messages implicites* et *messages explicites* dans les modèles à objets et à prototypes.

Les messages implicites sont des messages comme ceux qui ont été décrits plus tôt. Sa syntaxe ne se distingue pas de celle d'un appel de fonction quelconque.

Les messages explicites ont une syntaxe plus riche pour manipuler ses éléments: le receveur, le message et ses attributs, d'autres paramètres relatifs au contrôle des messages et des mots réservés pour différencier sa tâche des autres fonctions.

Ceci entraîne un plus grand pouvoir de contrôle qui permet à l'utilisateur d'exprimer différents mécanismes de communication. Parmi eux, la capacité de diriger des messages vers un objet différent de celui de défaut (par exemple, pour établir

des variations dans la délégation) ou pour définir des envois des messages à plusieurs objets à la fois. Cette richesse expressive n'implique pas nécessairement une meilleur clarté.

Dans les deux modèles, à classes et à prototypes, la communication entre entités est prédéfinie. En rendant le mécanisme des messages plus souple, nous allons en étudier les relations et l'impact qu'il a pour le système en général. Nous croyons qu'il deviendra, dans nos modélisations, la colonne vertébrale du fonctionnement du système.

Nous étudierons les types d'interaction entre entités pour explorer les mécanismes d'envoi des messages, mais développer l'ensemble des fonctions de communication pour un système est au-delà de cette étude. Cependant, dans les modélisations un sous-ensemble sera implémenté.

2.3 Réflexion et langages réflexifs.

La réflexion est l'ensemble des outils d'un système lui permettant de se regarder introspectivement, de se poser des questions sur lui-même et d'agir éventuellement pour modifier son comportement. À partir de la connaissance de soi-même et de son environnement (situation dans le monde, selon [Smi82] et [Mae87]) le système peut s'adapter à des nouvelles conditions.

Dans les systèmes informatiques, on applique l'adjectif réflexif s'il incorpore des structures représentant le système lui-même et s'il en permet l'accès et la modification.

Les exigences d'un système réflexif impliquent une *connexion causale* dans sa représentation. Cette connexion causale implique que si le domaine dont le système a une représentation interne change, cette dernière doit changer en conséquence. On garantit ainsi une représentation toujours exacte du domaine. Le calcul du système se fait en fonction de cette représentation.

La réflexion est en général très coûteuse en termes de temps et de performance du système. Ses applications vont du développement des outils de déverminage jusqu'à la gestion de la concurrence, en passant par l'expression de nouvelles sémantiques d'héritage.

On reconnaît deux types de réflexion pour les langages de programmation. La première, la *réflexion de structure* fait référence à la représentation du programme et des structures des données utilisées par le langage. La *réflexion de comportement* par contre, exige la représentation de la sémantique du langage. Par exemple, la première est obtenue en ayant accès à la représentation des classes ou des méta-classes du système. La deuxième implique un accès aux algorithmes de recherche des méthodes dans la classe et à la possibilité de les modifier. Une autre façon de traiter la réflexion de comportement est de permettre l'accès et des modifications à l'interprète du langage. Les changements sont faits à l'aide d'un autre interprète, sous-jacent au premier qui à son tour pourrait avoir un interprète, et ainsi de suite, constituant ce qui a été défini comme une *tour réflexive* [Smi82].

L'action de rendre disponible comme données les structures qui représentent l'état d'un programme pour sa manipulation (et son éventuelle modification) est connu sous le nom de *réification.* L'action contraire, où on réintègre les structures en forme de données à sa forme originale pour son fonctionnement est nommée *réflexion.*

Proto-Reflex est un langage réflexif et les objets du système sont des prototypes réflexifs à l'aide des méta-objets.

Parmi les avantages de Proto-Reflex, on retrouve sa souplesse en termes de traitement et manipulation des entités. En effet, dans Proto-Reflex presque tous les éléments du système sont des objets (même les environnements des variables) qui peuvent être, grâce à sa nature réflexive, réifiés, manipulés et changés. Proto-Reflex est donc approprié pour notre but de modélisation des structures externes de différents systèmes.

2.4 Introduction à Proto-Reflex.

Proto-Reflex [Jac95] est un langage à prototypes implantant la réflexion de comportement. Proto-Reflex comprend un interprète et un ensemble d'objets qui constituent le noyau du système.

Dans Proto-Reflex presque toutes les entités sont représentées comme des objets, qui sont en fait des prototypes, soit un ensemble de champs chacun contenant une donnée ou une méthode.

Les éléments de base, comme les nombres, caractères ou vecteurs, n'ont pas été implantés comme des prototypes en Proto-Reflex, mais comme des structures de données avec une syntaxe semblable à celle du langage Scheme. L'activation d'une méthode ou l'accès à un champ se fait à l'aide de l'envoi de messages.

Un prototype est construit comme une extension d'un autre en utilisant un lien de délégation. Ce lien est défini par un champ qui pointe vers un objet parent. Proto-Reflex comporte une fonction primitive de clonage et une autre de création *ex nihilo* pour générer des nouveaux prototypes.

L'objet ROOT est la racine de la hiérarchie de délégation. Son méta-objet est BMO, le méta-objet de base.

Similairement à l'objet ROOT, chaque prototype possède un méta-objet qui décrit la procédure de recherche d'une méthode ou d'un champ de donnée. Au moment de la réception du message, le méta-objet du prototype lui indique la façon de parcourir les protoypes selon les liens de délégation pour chercher la méthode ou la donnée demandées. Le méta-objet qui appartient à chaque objet contient la méthode *lookup* qui effectue le processus de recherche.

Chaque méthode est représentée par un prototype qui possède une méthode *apply*. Quand la méthode est trouvée, la deuxième partie du protocole *lookup-apply* se réalise et la méthode est exécutée.

Dans le cas de l'objet ROOT, l'objet BMO qui est son méta-objet contient un champ *lookup:in:* qui pointe vers un prototype appelé BL pour "Basic Lookup" ou *recherche de base.* Il représente le *lookup* ou processus de recherche par défaut de tout le système. À son tour, BL a un champ *applyTo:wA:wC* qui pointe vers une autre prototype nommé BA pour "Basic Apply" ou *application de base.* Il représente le processus d'éxécution des méthodes. BL comme BA peuvent être définies localement grâce au mécanisme de délégation.

ROOT contient un ensemble de primitives accessibles aux autres objets. Ils sont:

`size`***(obj)*** retourne le nombre des champs de *obj.*

`nameAt`***(obj,i)*** retourne le nom du i-ème champ de *obj.*

`contentsAt`***(obj,i)*** retourne la valeur du i-ème champ de *obj.*

`contentsAtPut`***(obj,i)*** modifie la valeur du i-ème champ de *obj.*

`isMethodAt`***(obj,i)*** vérifie si le i-ème champ de *obj* est une méthode.

L'envoi de message est composé de trois parties: le receveur du message, le sélecteur et les paramètres. Sa syntaxe est comme suit:

$$([\texttt{send}]\textit{expression selecteur } arg_1 \cdots arg_n)$$

Le mot clef [send] est facultatif et le sélecteur représente le nom du champ. L'expression est évaluée pour connaître le receveur du message. La procédure à suivre pour rechercher la méthode est spécifiée dans le méta-objet attribué à l'objet. Dans le cas de base, elle commence par l'objet lui-même. Si la méthode n'est pas trouvée, la recherche se poursuit dans le parent de l'objet, et ainsi de suite, à travers le lien de délégation, jusqu'à la racine de la hiérarchie de délégation (objet ROOT).

La pseudo-variable self indique à l'interprète de lui donner la valeur de l'objet courant. Si elle est trouvée comme résultat de l'évaluation de l'expression du receveur, la recherche de la méthode commence par l'objet courant.

super est aussi une pseudo-variable qui désigne l'objet courant mais la recherche de la méthode commence à partir de son parent. Ceci a été implementé de façon à donner plus de souplesse au mécanisme de la délégation en Proto-Reflex, en particulier en permettant à la méthode m d'appeler la méthode m de son parent sans risque de récursivité infinie.

La syntaxe de base d'un programme en Proto-Reflex est la suivante:

Expr Top Level	::=	(`define` *Variable Expression*)
	|	(`exit`)
	|	*Expression*
Expression	::=	*Élément de base*
	|	*Variable*
	|	(`method` (*Args ou vars*) *Expression*)
	|	(`block` (*Args ou vars*) *Expression*)
	|	(`set!` *Variable Expression*)
	|	([`send`] *Expression Selecteur Arguments*)
	|	(`self` *Expression Selecteur Arguments*)
	|	(`super` *Selecteur Arguments*)
Élément de base	::=	*Entier*
	|	*Réel*
	|	*Caractère*
	|	*Chaîne de caractères*
	|	*Symbole*
	|	*Vecteur*
Variable	::=	*Identificateur*
Selecteur	::=	*Identificateur*
Args ou vars	::=	(*Variable* |(*Variable Expression*))

Les listes de messages auxquels peuvent répondre les éléments de base sont décrits dans [Jac95].

Chapitre 3

Modélisation d'ObjVLisp en Proto-Reflex.

Dans ce chapitre nous allons modéliser ObjVLisp à l'aide de Proto-Reflex. ObjVLisp est un interprète developpé à partir d'un dialect de Lisp, *VLisp*, créé à l'Université de Vincennes sous la direction de Patrick Greussay [MNC+90]. ObjVLisp est un langage à classes et Proto-Reflex est un langage à prototypes. Nous avons choisi ObjVLisp parce qu'il offre un traitement original et systématique des méta-objets, particulièrement des méta-classes. On dénote cette fonctionalité *programmation par métaclasses explicites.*

Certaines caractéristiques sont partagées par les deux langages, comme la communication par l'envoi de messages et une structure générale hiérarchique à partir des objets "racine". Nous y reviendrons plus tard.

ObjVLisp n'est pas complètement réflexif, même si son architecture permet la réflexion de structure. L'ensemble des capacités réflexives de Proto-Reflex seront utilisées pour aider la modélisation des langages. Au cours de l'implantation d'ObjVLisp les fonctions réflexives pour manipuler les environnements se sont avérées très utiles.

3.1 Le modèle ObjVLisp

ObjVLisp a été développé par P. Cointe pour uniformiser le traitement des objets d'un système à classes. Pour arriver à ce but, ObjVLisp uniformise d'abord le processus de création des nouveaux objets. En partant du modèle de Smalltalk, le modèle ObjVLisp traite les classes comme des objets de plein droit associés à une méta-classe. Il donne l'accès aux méta-classes du système parce qu'elles deviennent les "productrices" des classes. La régression infinie vers les méta-classes antécédentes est arrêtée par une classe de départ, l'objet CLASS, qui à travers un lien circulaire, est instance d'elle même. L'objet CLASS est la racine du graphe d'instanciation du système.

L'initialisation des classes est faite par les métaclasses car elles détiennent le format des nouvelles classes. Par contre, l'instanciation des instances terminales (c'est-à-dire des instances qui ont seulement des valeurs dans leurs champs) est réservée à l'objet OBJECT qui est la racine du graphe d'héritage. Parallèlement au cas de l'objet CLASS, l'objet OBJECT limite le graphe en supprimant son lien vers le "super".

Le langage ObjVLisp est construit autour de la relation fonctionnelle entre deux entités. Le bootstrap d'ObjVLisp utilise essentiellement deux objets racine: CLASS et OBJECT. Chacun est à l'origine de graphes de fonctions différents.

Dans le modèle, OBJECT est une instance de CLASS. En même temps, CLASS elle-même un objet, hérite de OBJECT. Cette interrelation est illustrée dans la figure 3.1.

Afin d'éviter les conflits de termes, on va appeler Classe-racine l'objet CLASS et Objet-racine l'objet OBJECT.

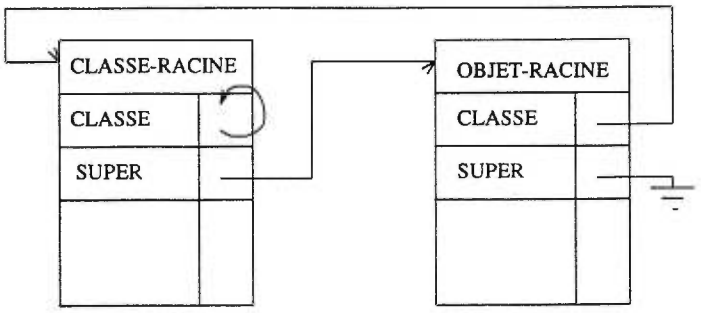

FIG. 3.1 - *Bootstrap d'ObjVLisp*

3.2 Les six postulats d'ObjVLisp.

Six postulats [Coi87] décrivent le modèle d'ObjVLisp et sa philosophie:

P1 Un objet représente une entité de connaissance et un ensemble d'actions.

$$\text{objet} = \langle data, procedures \rangle$$

P2 Le seul protocole pour rendre un objet actif est le mécanisme de passage de messages: un message spécifie la procédure à appliquer et ses arguments (dénotée par son nom):

$$(\text{send } objet\ sélécteur\ Args_1 \ldots Args_n\ )$$

P3 Chaque objet appartient à une classe qui spécifie ses données (des attributs appelés champs) et son comportement (des procédures appelées méthodes). Les objets qui seront générés dynamiquement à partir de ce modèle sont appelés les instances de la classe.

P4 Une classe est aussi un objet instanciée à son tour par une autre classe appelée sa méta-classe. Conséquemment, avec P3, chaque classe est associée à une méta-classe laquelle décrit son comportement comme un objet. La méta-classe initiale primitive est la classe Classe-racine qui est sa propre instance.

P5 Une classe peut être définie comme une sous-classe d'une ou plusieurs classes. Ce mécanisme de hiérarchie permet le partage des variables d'instances et des méthodes et reçoit le nom d'héritage. La classe Objet-racine représente le comportement commun partagé par tous les objets.

P6 Si les variables d'instance qui appartiennent à un objet définissent un environnement local, il y a aussi des variables de classe qui définissent un environnement global partagé par toutes les instances d'une même classe. Ces variables de classe sont définies au niveau de la méta-classe selon l'équation suivante:

variable de classe d'un objet = variable d'instance de la classe de l'objet

3.3 Implantation de la modélisation .

Nous allons implanter un noyau minimal du langage ObjVLisp avec Proto-Reflex. Ce noyau restera complètement séparé du langage récepteur mais il utilisera son environnement du travail. Le noyau pourra par la suite être étendu selon les besoins de l'utilisateur.

Nous commencerons par créer le bootstrap ou "embryon" d'ObjVLisp. Les deux objets (Classe-racine et Objet-racine) peuvent être modélisés à l'aide des prototypes avec des liens vers d'autres prototypes. On considère deux types des liens: liens-valeur et liens de délégation, propres à la philosophie des prototypes de Proto-Reflex. Les prototypes seront aussi utilisés pour représenter des propriétés des objets, comme l'ensemble des méthodes qui caractérisent le comportement de l'objet. Dans Proto-Reflex, les liens de délégation doivent être mis en place au moment de la création des prototypes. Ceci explique le fait que le processus d'initialisation du noyau est divisé en étapes.

La première étape envisage la création du prototype qui caractérise le comporte-

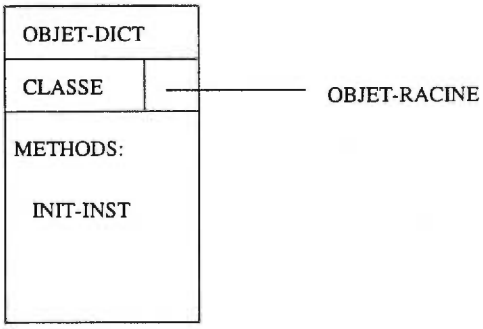

FIG. 3.2 - *Structure du dictionnaire de l'Objet-racine*

ment de l'Objet-racine: son dictionnaire Objet-dict. Après, on crée le prototype dictionnaire de la Classe-racine. Dans le modèle d'ObjVLisp, les dictionnaires gardent les méthodes des objets, c'est-à-dire, leur comportement. Le dictionnaire de l'objet Classe-racine est particulièrement important parce qu'il détient la méthode de création universelle new.

Une deuxième étape correspond à l'instanciation des nouveaux objets. Dans le modèle original d'ObjVLisp, le processus d'initialisation est divisé en deux méthodes: une méthode d'instanciation traite les cas de méta-classes et classes et l'autre méthode s'occupe de l'initialisation des instances terminales. Cette dernière méthode appartient au dictionnaire de l'Objet-racine. Nous avons gardé cette convention telle qu'elle est décrite en [Coi87]. Au moment du bootstrap, on crée d'avance les dictionnaires. Le premier est le dictionnaire de l'objet Objet-racine qui comporte une structure montrée dans la figure 3.2.

Le suivant c'est le dictionnaire de la Classe-racine dont sa structure est montrée dans la figure 3.3. À noter les méthodes new et initialize, cette dernière ne servant qu'aux méta-classes et classes, mais non aux instances terminales.

Après la création des dictionnaires, c'est le tour des objets de base: à partir de la méthode **new** localisée dans Classe-dict, Objet-racine et Classe-racine sont construits avec les structures et les relations montrées dans la figure 3.4.

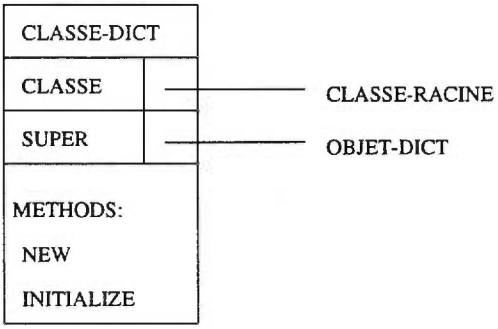

FIG. 3.3 - *Structure du dictionnaire de la Classe-racine*

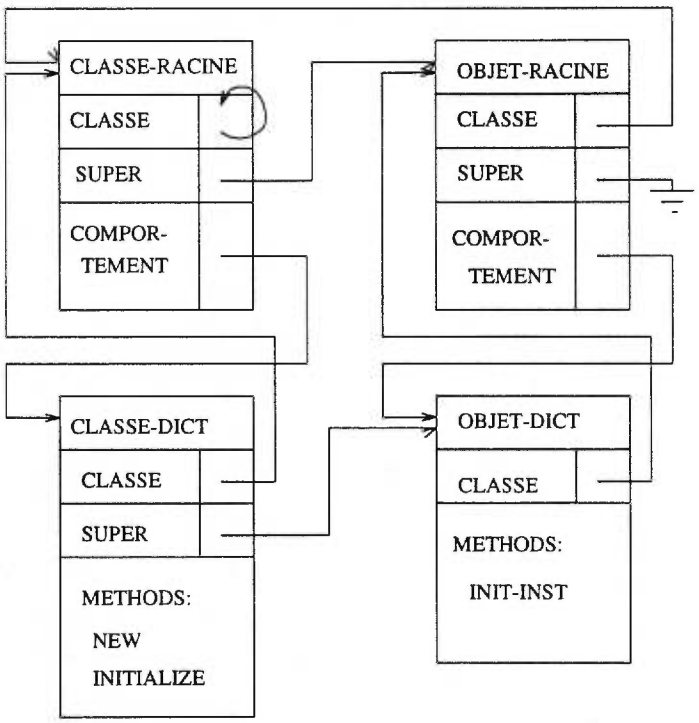

FIG. 3.4 - *Relations du noyau après le bootstrap*

3.4 Unification des méthodes de création pour les classes et les instances.

Un des buts d'ObjVLisp est d'uniformiser la nature des objets dans le système, c'est-à-dire, de donner le même comportement et structure aux méta-classes et aux classes et partager cette structure avec les objets terminaux (instances terminales). Cet effort se reflète fortement dans le mécanisme de création, qui doit s'appliquer uniformement tout au long du système. Il faut donc avoir une seule méthode pour créer les nouvelles instances, soit une classe (instance de méta-classe) soit un objet (instance terminale d'une classe).

Cette méthode de création utilise comme modèle makeInstance qui est la fonction primitive de la Machine Virtuelle pour créer de nouvelles instances. Dans notre système, cette action se réalise en envoyant un message à la classe correspondant avec le sélecteur new et l'information de départ du nouvel objet. L'information de départ doit contenir une variable, denommée isit pour établir le lien de l'objet vers sa classe. isit est la première des variables d'instance dans la structure, suivie du reste de l'ensemble des variables.

Puisque les classes servent à générer d'autres classes, l'ensemble minimal des variables d'instance qu'elles peuvent avoir est comme suit:

name Le nom de la classe. Par définition en ObjVLisp, les classes ne sont pas des objets anonymes.

supers la liste des superclasses de la classe.

var-inst la liste des variables d'instance de la classe spécifique.

dict le dictionnaire des méthodes propres à la classe.

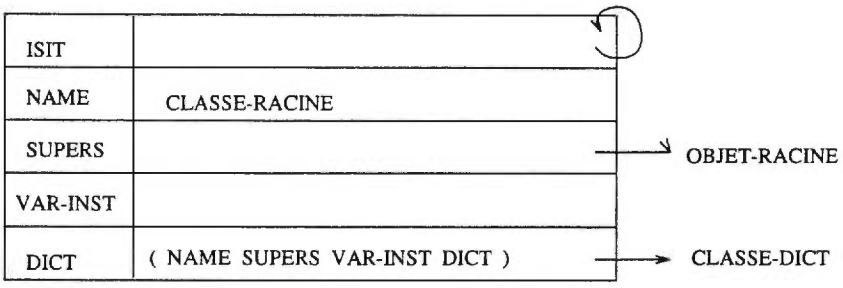

FIG. 3.5 - *Structure de la Classe-racine avec son format de création des métaclasses*

Le dictionnaire des méthodes est défini comme un ensemble de paires de la forme:

$$\langle symbole, \lambda - expression \rangle$$

Dans la section suivante, la description des messages de création sera présentée en pseudocode (qui ressemble beaucoup à Proto-Reflex) où chaque variable sera précédée d'un symbole qui permet une identification précise. La syntaxe réelle du message en Proto-Reflex garde et respecte seulement la positions des valeurs.

Quelques directives pour comprendre le pseudocode sont:

- # indique un vecteur. Un vecteur est une structure de données (généralement implémentée comme une liste) où chaque élément peut avoir un type différent. On peut accèder aux éléments par indiçage.
- un objet est un vecteur de vecteurs.
- les symboles (pour nommer un champ ou un objet) sont précédés d'un apostrophe.
- la fonctionalité des parenthèses est semblable à celle du Lisp.

En ObjVLisp, le message de création est établi uniformement pour les métaclasses, classes et instances. Particulièrement, la classe Class, racine de la hiérarchie d'héritage détient la méthode de création new. La figure 3.5 montre sa structure.

Comme point de départ, nous considérons l'envoi du message new à la classe Classe-racine afin d'obtenir une nouvelle méta-classe:

1. Méta-classes

```
(send Classe-racine new
              (#('name 'Nom-MetaClasse)
               #('supers (superclasse1 ... superclasseN))
               #('var-inst (#(vimc1 v-vimc1) #(vimc2 v-vimc2)
                         ... #(vimcN v-vimcN)))
               #('dict  ('Nom-MetaClasse-dict)
                         ('m1 (method () ...))
                         ('m2 (method () ...))
                                 ...
                         ('mN (method () ...)))))
```

Cet message a comme résultat l'objet suivant:

```
#(#('isit Classe-racine )
  #('name 'Nom-MetaClasse)
  #('supers (superclasse1 ... superclasseN))
  #('var-inst #(name supers var-inst dict
                #(vimc1 v-vimc1)
                #(vimc2 v-vimc2)
                     ... #(vicmN v-vicmN)))
                #('dict  Nom-MetaClasse-dict))
```

et Nom-MetaClasse-dict est l'objet:

```
#( #('isit Nom-MetaClasse)
   #('name Nom-Meta-Classe-dict)
   #('m1 (method () ...))
   #('m2 (method () ...))
```

```
                ...
        ('mN (method () ...)))
```

Dans cette figure, la séquence

vimc1 vimc2 .. vimcN

représente l'ensemble des variables d'instance propres de la méta-classe.

Pour le cas de méta-classes, l'ensemble de variables d'instance est obtenu comme l'union des deux ensembles suivants:

(a) L'ensemble composé par

name, **supers**, **var-inst**,**dict**

définit le "squelette" ou moule pour construire les nouvelles instances, et

(b) l'ensemble des variables propres à la méta-classe, défini dans le message new.

2. Classes

Pour les classes, le message de création a la même structure:

```
(send UneMetaClasse new
          (#('name 'Nom-Classe)
           #('supers (Object))
           #('var-inst (#(vic1 v-vic1)
                        #(vic2 v-vic2)
                            ...
                        #(vicN v-vicN)))
           #('dict  ('Nom-Classe-dict)
                    ('m1 (method () ...))
                    ('m2 (method () ...))
                        ...
                    ('mN (method () ...)))))
```

Et le résultat nous donne un objet décrit dans le champ **var-inst** de sa méta-classe:

```
#(#('isit Classe )
  #('name 'Nom-Classe)
  #('supers (Object))
  #('var-inst #(isit
                #(vic1 v-vic1)
                #(vic2 v-vic2)
                    ...
                #(vicN v-vicN)))
  #('dict  'Nom-Classe-dict)
  #(vimc1 v-vimc1)
  #(vimc2 v-vimc2)
      ...
  #(vimcN v-vimcN))
```

Comme d'habitude, le dictionnaire est formé par un autre prototype:

```
#(#('isit Nom-Classe)
  #('name Nom-Classe-dict)
  #('m1 (method () ...))
  #('m2 (method () ...))
          ...
  #('mN (method () ...)))
```

Dans cette figure, la séquence

```
(#(vic1 v-vic1) #(vic2 v-vic2) ... #(vicN v-vicN))
```

représente la liste des variables d'instance de la nouvelle classe. Ici on montre chacune des variables d'instance accompagnée par sa valeur *v-vc1 v-vc2 ... v-vcN v-vm1 v-vmc2 ... v-vmcN* dans une structure de vecteurs.

Par contre, cette autre séquence

```
vimc1 vimc2 .. vimcN
```

représente l'ensemble des *variables d'instance de la méta-classe* à laquelle on a envoyé le message.

Il faut noter que dans ces processus l'équation:

variables d'instance de la classe = variables de classe de l'instance

est toujours respectée.

3. Instances terminales.

   Similairement, on peut créer des instances en utilisant la même méthode new avec la même structure d'envoi des messages. Ceci garantit une uniformité de création entre les classes et les instances.

```
(set! Nom-Instance (send UneClasse new
                         (#(vic1 v-vic1)
                          #(vic2 v-vic2)
                               ...
                          #(vicN v-vicN))))
```

donnant comme résultat l'objet:

```
#(#(isit Classe)
  #(vic1 v-vic1)
  #(vic2 v-vic2)
       ...
  #(vicN v-vicN))
```

Dans la Machine Virtuelle l'initialisation était implémentée avec deux méthodes différentes: une pour les classes (laquelle se trouvait dans l'objet Classe-racine) et une autre pour les instances (localisée dans l'objet Objet-racine). Nous avons gardé cette caractéristique en implémentant deux méthodes d'initialisation, un dans l'objet racine du graphe d'héritage, **Objet-racine** et l'autre dans l'objet racine du graphe d'instantiation **Classe-racine**.

3.5 Examples

Comme exemple, l'envoi du message new à la classe Classe-racine crée la nouvelle classe Vehicule, avec deux variables d'instance (qui vont devenir variables de classe) et un dictionnaire des méthodes appelé vehicule-dict qui contient deux méthodes.

```
P-R> (Classe-racine new #('Vehicule
                          #('Classe-racine)
                          #('marque 'prop)
                          #( 'vehicule-dict
                             #('marche (method () ... ))
                             #('arrete (method () ... )))))

Value: #[nil]
```

Le nouvel objet a été ajouté à l'environnement global grâce aux facilités réflexives de Proto-Reflex, qui permet l'instruction suivante:

(`:global-env :add-var` *symbole valeur*)

On teste le nouvel objet crée:

```
P-R> (vehicule isit)

Value: classe-racine
```

```
P-R> (vehicule var-inst)

Value: #(name supers var-inst dict marque prop)

P-R> (vehicule dict)

Value: vehicule-dict
```

Le dictionnaire associé au nouvel objet est le prototype vehicule-dict composé par le vecteur suivant:

```
#(#(marche #[method]) #(arrete #[method]))
```

auquel on peut aussi envoyer des messages pour avoir accès aux méthodes. Cette souplesse a été ajoutée comme facilité aux activités de soutien et modification des méthodes, mais le moyen le plus correct pour changer son information est à travers son objet associé.

Maintenant, on crée une classe de la méta-classe vehicule. L'ensemble des méthodes est étendu et on donne des valeurs aux variables de classe prédéfinies dans la méta-classe:

```
(Vehicule new #('Voyager
                #('Objet)
                #('couleur)
                #('voyager-dict
                  #('droit (method () ... ))
                  #('gauche (method () ... )))
                'Chrysler
                'Gelman))
```

et on teste le résultat:

```
P-R> (voyager isit)

Value: vehicule

P-R> (voyager marque)

Value: chrysler

P-R> (voyager var-inst)

Value: #(couleur)
```

Finalement, on démontre la création des instances terminales avec le message new:

```
P-R> (Voyager new #('rouge))

#(rouge)#(#(isit voyager) #(couleur rouge))

Value: #[object]
```

le résultat est un prototype qui peut être lié à une variable comme dans l'expression:

```
(set! cadeau (Voyager new #('rouge)))
```

Chapitre 4

Modélisation du CLOS en Proto-Reflex

Dans ce chapitre nous allons modéliser CLOS, un langage à classes, à l'aide de Proto-Reflex. CLOS sera modélisé sous sa forme *Closette* [KdRB91]. Closette est un sous-ensemble didactique du CLOS qui malgré ses simplifications est représentatif du langage original. Alors que CLOS a été développé comme un standard ANSI, Closette a été défini pour étudier et explorer les possibilités de l'application des méta-protocoles dans un langage à objets. Closette garde les principes fondamentaux de CLOS en éliminant certaines facilités qui compliquent l'implantation du langage. Gardant l'essentiel du modèle CLOS, Closette est donc approprié pour une modélisation à l'aide de prototypes.

4.1 Introduction à CLOS et à Closette.

Common Lisp Object System (CLOS) a été développé par un comité[1] chargé de créer l'extension orientée objet pour Common Lisp en tant que standard ANSI. CLOS

1. Le comité X3J13 de l'ANSI qui a réuni des équipes des sociétés Xerox, Symbolics, Lucid, Hewlett-Packard et Texas Instruments pour cet projet.

est un langage de programmation de haut niveau orienté objets avec une syntaxe héritée de Common Lisp. Il possède un mécanisme d'héritage multiple complémenté par des mécanismes de combinaisons de méthodes. CLOS est un langage en premier lieu applicatif qui a comme des éléments principals les fonctions. Basé sur le lambda calcul, Common Lisp (et par extension CLOS) fait l'application des fonctions sur n'importe quel élément du système: des nombres, des structures de données ou d'autres fonctions. Toutes les actions de transformation de l'état du système sont faites à l'aide de l'application d'une fonction.

CLOS est organisé autour de quatre types d'éléments: des classes, des instances, des méthodes et des fonctions génériques.

Les classes sont composées d'un ensemble de variables qui définissent la structure des instances. Chaque *variable d'instance* se trouve dans un champ et elle est accompagnée d'une liste de propriétés. Les classes doivent être créées avant leurs instances. Les instances ne portent que les données correspondantes à la structure de leurs classes respectives.

Par exemple, la classe `elephant` ci-dessous definit la structure de ses instances à l'aide de la forme spéciale defclass. Suite au nom de la nouvelle classe, on indique entre parenthèses la liste de superclasses auxquelles elle est attachée. Dans notre exemple, la liste est vide pour indiquer que la classe `elephant` a comme superclasse la classe standard-class, racine de la graphe d'héritage. Elle sera expliquée en détail plus tard. Les options :initform, :initarg et :accessor définissent des propriétés de chaque variable d'instance. Les formes :initform et :initarg sont utilisées au moment de l'initialisation de la nouvelle instance. La valeur de l'option :accessor représente le nom d'une fonction générique automatiquement créée qui sert à accéder la valeur du champ où elle apparaît.

```
(defclass elephant ()
  ((pattes :initform 4
```

```
        :initarg n-pattes-initiales
        :accessor :n-pattes)
  (defenses :initform 0
            :initarg n-defenses-initiales
            :accessor :n-defenses )
  (couleur :initform gris
           :initarg couleur-initiale
           :accessor :couleur)))
```

Les nouvelles instances sont créées à l'aide de la fonction make-instance comme suit:

```
(setq 'Clyde (make-instance 'elephant n-defenses-initiales 2))
```

Dans ce cas, l'instance `Clyde` aura trois champs avec les valeurs { 4 2 gris }.

Quant aux méthodes, chaque classe est associée à une ou plusieurs méthodes qui décrivent son comportement. Contrairement à Proto-Reflex, les méthodes ne font pas partie de la structure de la classe. Elles font le lien avec leurs classes en les recevant comme paramètres.

Les méthodes sont des objets indépendants groupés dans des fonctions génériques. Celles-ci sont des structures avec une interface commune aux méthodes permettant leur invocation. En effet, la définition d'une fonction générique est composée de son nom et de sa liste d'arguments qui constitue l'interface.

Une méthode est associée à une classe si un paramètre de la méthode est *spécialisé* à cette classe. Être *spécialisé* veut dire que le paramètre doit recevoir une instance de la classe associée ou bien d'une de ses sous-classes. Dans ce sens, les méthodes sont des unités d'implémentation de fonctions spécifiques à une ou plusieurs classes. Dans notre exemple, la classe `elephant` est associée à la méthode `dessine` qui dessine la figure d'un elephant selon les données du paramètre `silhouette`. `silhouette` doit être une instance de la classe `elephant` ou une de ses sous-instances:

```
(defmethod dessine ((silhouette elephant) type-ligne)
    (dessine-elephant  (n-pattes figure)
                       (n-defenses figure )
                       (couleur figure )
                       type-ligne)))
```

La liste d'arguments d'une méthode peut contenir ou bien des noms de paramètres (comme `type-ligne`, dans l'exemple) ou bien des *paires paramètre-classe* qui représentent l'instance et sa classe. Dans la paire *paramètre-classe*, le nom de la classe est appelé en Closette *spécialisateur*. Dans notre example, le *spécialisateur* est `elephant`[2]

Les méthodes ne sont pas restreintes à seulement recevoir comme arguments les classes et les sous-classes. Elles peuvent recevoir aussi d'autres sortes de données: structures de données indépendantes, des champs associés à la classe ou aux sous-classes, etc.

Parallèlement à la relation classes-instances il existe une relation entre les fonctions génériques et les méthodes. Comme mentionné précédemment, les méthodes sont associées et regroupées par fonctions génériques. En plus de la méthode *dessine-elephant* déjà décrite, il pourrait avoir des méthodes *spécialisées* aux autres classes. Par exemple:

```
(defmethod dessine ((silhouette chameau) type-ligne)
    (dessine-chameau  (n-pattes figure)
                      (n-bosses figure )
                      type-ligne)))
```

Parce que la structure de la classe `chameau` est différente de la classe `elephant`, il a été nécessaire de construire une nouvelle méthode qui gère les différences. Cepen-

2. L'utilisation du substantif *spécialisateur* et du verbe *spécialiser* dans ce contexte est différent de celle qui désigne régulièrement un aspect du mécanisme de l'héritage dans les langages à objets. On a gardé cette terminologie propre à Closette en l'indiquant en italique dans son contexte pour éviter des confusions.

dant, les caractéristiques communes aux deux classes sont regroupées dans une même interface définie par leur fonction générique:

```
(defgeneric dessine (silhouette type-ligne))
```

En ce qui concerne l'application des méthodes, on trouve en Closette un schéma bien différent que celui dans d'autres langages à objets. Les méthodes en Closette ne sont pas envoyées à un objet avec un message pour entrer *"en communication"* avec l'objet qui va ensuite *"gérer"* la situation et satisfaire la requête. Au lieu de ça, et en renforçant le point de vue des langages applicatifs, elles sont *invoquées* avec les classes et d'autres données comme arguments. Comme les fonctions génériques représentent l'interface commune des méthodes, la façon d'appliquer une méthode est d'appeler sa fonction générique. Celle-ci analyse le type d'arguments de la requête pour ensuite appliquer la méthode adéquate.

Dans notre exemple, pour dessiner une figure on fait l'appel à la fonction générique (du même nom que les méthodes) qui va *discriminer* automatiquement entre les deux méthodes et qui va *appliquer*, celle qui est la plus appropriée. CLOS unifie donc l'appel de fonction et l'envoi de messages avec ses fonctions génériques.

La figure 4.1 montre les relations entre les éléments qui composent un système en CLOS. Le lien entre les méthodes et sa fonction générique symbolise leur attachement à une méta-structure qui définit leur communication avec les autres éléments du système. En d'autres mots, la fonction générique definit leur interface.

D'autre part, le lien entre les classes et les méthodes qui les reçoivent est un lien de comportement. L'architecture du système, avec son comportement "externe" à la structure des classes permet à l'usager de créer des programmes où le contrôle des actions est hors des objets. Tout le contrôle du système réside dans le côté droit de la figure 4.1, dans la structure hiérarchique des méthodes et des fonctions génériques.

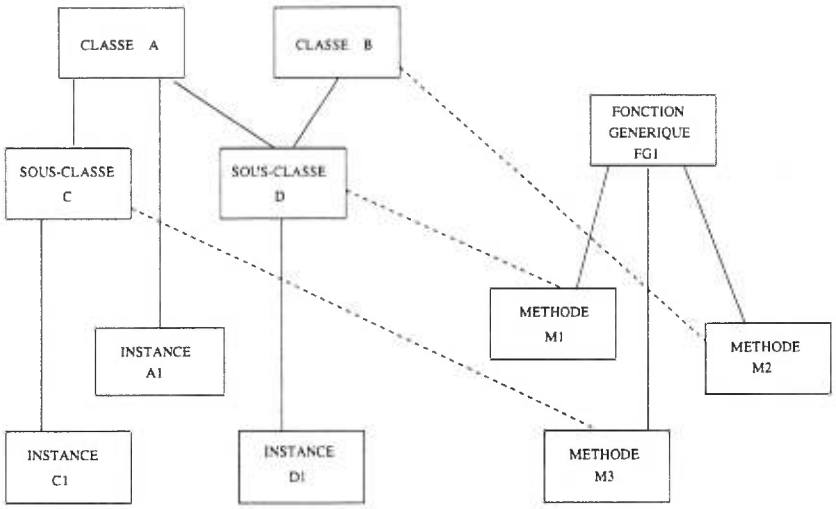

FIG. 4.1 - *Relations entre les éléments de CLOS*

L'autre côté (classes et instances) est plutôt statique. Elles gardent la forme et la structure des objets du système pendant que le coté droit réalise les transformations qui changent dynamiquement le système. En plus, cette architecture est fortement enracinée dans le modèle de la programmation fonctionnelle. En fait, on pourrait visualiser le tout comme des actions du côté droit qui s'exécutent sur la structure du côté gauche. Le scénario de Closette est un ensemble de fonctions groupés dans des structures génériques pour être appliquées sur des classes statiques.

En Closette presque tout le codage d'opération se trouve hors des objets comme les classes ou sous-classes. Ce codage est implementé indépendamment sous la forme des méthodes et des fonctions génériques. L'application des fonctions est privilégiée de telle sorte que même pour accéder aux données d'un objet on recommande plutôt la fonction générique créée à partir de l'option :accessor que la primitive slot-value qui renforce une action provenant de l'objet lui-même. La méthode lookup, ansi comme d'autres méthodes de base du système est implémentée comme une partie d'une fonction générique.

Malgré cette complexité et la séparation des rôles des éléments, l'architecture du système a l'avantage d'être particulièrement appropriée pour l'addition des actions

dans un méta-niveau. En effet, nous allons voir que cette architecture "à deux vitesses" permet aisément l'inclusion de la méta-programmation et par extension, de la réflexion de structure et de comportement.

Nous allons distinguer en CLOS deux types d'éléments selon les rôles qu'ils jouent dans le système. Les premiers forment un *niveau d'objets* statique et composent le domaine du programme. Dans ce niveau, on trouve les instances. qui sont créées, manipulées et exploitées à volonté par l'usager. Un deuxième niveau appelé *méta-niveau* comprend les éléments dont leur création, manipulation et exploitation ne sont pas faits par l'usager, mais par le programmeur ou l'implanteur du système. Appelés *méta-objets* par les concepteurs du système CLOS, ces éléments forment le code du système en ayant ainsi le contrôle de son opération. Les classes, fonctions génériques et méthodes se trouvent à un *méta-niveau* souvent comparé à l'arrière-scène d'un théatre (*backstage* dans la description originale). Certains éléments à ce niveau sont créés automatiquement à partir des actions internes, c'est le cas des certaines fonctions génériques. D'autres éléments font partie du contrôle du système, par exemple les méthodes qui retournent la valeur d'un champ. Enfin, on trouve aussi au méta-niveau toutes les fonctions générales de comportement. La figure 4.2 résume la structure qui nous venons de décrire.

Les classes sont représentées de façon interne (dans le *backstage*) comme un méta-objet. Ce méta-objet suit la description faite par l'usager à l'aide de la commande defclass. Avec defclass on définit le nom de la classe, les paramètres, le nombre et les noms des champs qui la composent et le traitement de ces champs. Ce traitement comprend l'initialisation à l'aide des mots réservés :initform et :initarg et la définition de l'accès au valeur du champ à l'aide de :accessor.

Cette action provoque la création automatique d'un méta-objet et ses liens vers ses super-classes. Il est composé typiquement des champs montrés dans la figure 4.3.

Les champs direct superclasses, effective slots et class precedence list sont nécessaires

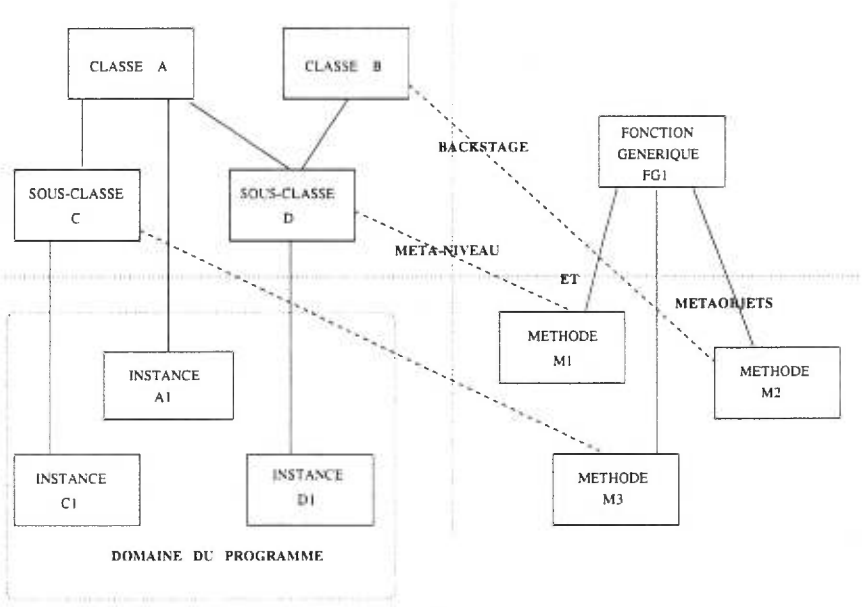

FIG. 4.2 - *Domaine et méta-objets dans CLOS*

CLASS NAME
DIRECT SUPERCLASSES LIST
DIRECT SLOTS
CLASS PRECEDENCE LIST
EFFECTIVE SLOTS
DIRECT SUBCLASSES
DIRECT METHODS

FIG. 4.3 - *Méta-objet standard pour les classes en CLOS*

parce qu'une classe peut hériter de deux classes en même temps, comme on peut le voir avec la sous-classe **D** dans la figure 4.1.

CLOS utilise les mécanismes de l'héritage pour exprimer le partage des comportements entre une classe et ses super-classes. L'héritage est définie en CLOS comme la spécification d'une nouvelle classe par modifications incrémentales d'une ou plusieurs classes existantes. Donc, la vraie définition de cette nouvelle classe est la combinaison de ce qui est explicitement défini dans son corps et dans les classes dont elle hérite. Cette combinaison peut être obtenue de deux façons différentes en CLOS: comme une *union* d'attibuts hérités ou comme une *substitution* d'attributs hérités.

Par exemple, en considérant la définition de la classe `elephant`, déjà faite, nous allons définir une sous-classe `elephant-albinus`:

```
(defclass elephant ()
   ((pattes :initform 4
            :initarg n-pattes-initiales
            :accessor pattes)
    (defenses :initform 0
              :initarg n-defenses-initiales
              :accessor defenses )
    (couleur :initform gris
             :initarg couleur-initiale
             :accessor couleur)))

(defclass elephants-albinus (elephant)
   (couleur :initform blanc
            :initarg couleur-elephant-initiale
            :accessor couleur-elephant)
   (couleur-yeux :initform rose
                 :initarg couleur-yeux-initiale
                 :accessor couleur-yeux-eleph)
```

De la même façon qu'avec la classe `elephant`, nous allons créer une instance de la sous-classe `elephant-albinus`:

```
(setq 'Fred (make-instance 'elephant-albinus n-pattes-initiales 3
                                             n-defenses-initiales 1))
```

Le redéfinition de l'option :initform du champ `couleur` de la sous-classe provoque la substitution de la valeur initiale `gris` de la super-classe avec la valeur `blanc` dans l'instance. Par contre, la redéfinition des options :initarg et :accessor donnent comme résultat des ensembles avec l'union des valeurs des options. De cette façon, l'instance `Fred` aura quatre champs, un propre et trois hérités. La valeur initiale du champ `couleur` est blanc et pour initialiser ce champ on peut utiliser n'importe quel mot de l'ensemble { `couleur-initiale` , `couleur-eleph-initiale` }. Finalement, pour accéder à la valeur du champ `couleur` le système crée une fonction générique qui peut être appelée à l'aide de n'importe quel mot dans l'ensemble { `couleur` , `couleur-elephant` }.

Comme pour defclass, les autres *formes de définition* defgeneric et defmethod contiennent l'information qui va être utilisée par le système afin de lier les méthodes et les fonctions génériques et de relier les classes et les méthodes.

Dans Closette, standard-class est le méta-objet qui représente la structure de toutes les classes. Pour chaque définition de classe il y a un méta-objet qui est une instance du méta-objet *standard-class*. Contrairement à CLOS, en Closette les fonctions génériques ne sont pas créées automatiquement à partir de l'information existante dans les méthodes. Il est de la responsabilité de l'implanteur de définir explicitement une fonction générique là où elle est nécessaire.

Il y a certaines d'autres différences entre CLOS et le sous-ensemble Closette. D'abord, la redéfinition de classes et de méthodes n'est plus permise en Closette, ainsi comme les références à l'avance des classes qui ne sont pas encore définies. Une autre restriction établit qu'aucun champ peut être partagé parmi les instances. Aussi, chaque fonction générique doit être introduite explicitement avant que ses méthodes ne soient définies. Ceci simplifie le système CLOS en éliminant l'inférance

d'une fonction générique à partir des définitions des ses méthodes.

En supprimant des fonctions non essentielles au système mais en gardant l'aspect réflexif de CLOS, Closette aboutit à un modèle didactique aussi approprié pour étudier des méta-structures que le modèle d'ObjVLisp.

4.2 Implémentation de Closette en Proto-Reflex

Toute implémentation de CLOS (et de Closette) est structurée en trois couches. La première couche, qui est en contact avec l'usager, traite les *formes de définition*: defclass, defmethod, defgeneric, etc. Cette couche fait le dépliage de la définition et éxécute la macro-instruction afin de générer les méta-objets correspondant aux définitions.

Les métaobjets ainsi créés seront gérés seulement par la troisième couche du système, chargée du processus au méta-niveau. Au milieu, la deuxième couche est responsable de donner le support entre la couche de traitement des définitions et celle qui fait le processus. Donc, la couche du milieu est composée des fonctions pour communiquer, par exemple, la fonction qui cherche le méta-objet d'une classe à partir de son nom ou la fonction qui retourne la valeur d'un champ dans un objet. La figure 4.4 montre la structure à trois couches de Closette.

Tous les méta-objets correspondant aux classes du système sont des instances de la classe standard-class, racine de la hiérarchie de classes. Elle doit être créée à l'avance, durant la première étape du *bootstrap*.

Le code de la classe *standard class* est le suivant:

```
1. (defclass standard-class ()
2.    ((name :initarg :name
             :accessor class-name)
3.     (direct-superclasses :initarg :direct-superclasses
                            :accessor class-direct-superclasses)
```

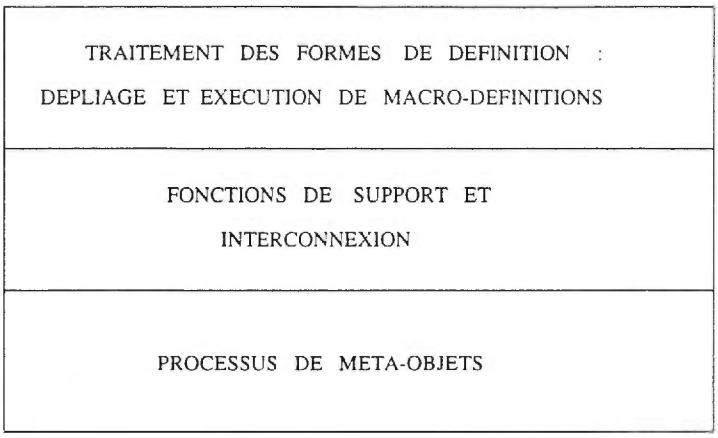

FIG. 4.4 - *Structure en couches d'une implémentation de CLOS et Closette*

```
4.      (direct-slots :accessor :class-direct-slots)
5.      (class-precedence-list :accessor class-precedence-list)
6.      (effective-slots :accessor class-slots)
7.      (direct-subclasses :initform ()
                           :accessor class-direct-subclasses)
8.      (direct-methods :initform ()
                        :accessor class-direct-methods)))
```

La modélisation de la classe standard-class en Proto-Reflex est à continuation. Elle a comme parent l'objet root de Proto-Reflex (ligne 2).

```
1. (define standard-class
2.  ((:root :clone ) :new-initials
3.   #('name   #('standard-class
                 #(':initarg #('name))
                 #(':accessor #(':class-name))))
4.   #('direct-superclasses #(#()
                              #(':initarg #('direct-superclasses))
                              #(':accessor
                                   #(':class-direct-superclasses))))
5.   #('direct-slots  #(#()
                        #(':accessor #(':class-direct-slots))))
6.    #('class-precedence-list #(#()
                                 #(':accessor
                                    #(':class-precedence-list))))
7.   #('effective-slots #(#()
                          #(':accessor #(':class-slots))))
```

```
8.   #('direct-subclasses #(#()
                            #(':initform #())
                            #(':accessor
                                  #(':class-direct-subclasses))))
9.   #('direct-methods #(#()
                        #(':accessor #(':class-direct-methods))))))
```

Le bootstrap de Closette diffère du bootstrap d'ObjVLisp au point de vue complexité et homogénéité. Le bootstrap d'ObjVLisp est extraordinairement "propre": il est réalisé d'une façon simple, complétement automatisé et à la fin du processus tous les objets sont prêts à fonctionner sans modification future. Par contre, le bootstrap de Closette n'est pas complétemente automatisé car il suppose l'éxistence de la classe standard class. Alors que dans ObjVLisp la classe CLASS est générée par une version minimale d'elle-même qui garde la méthode de création new, en Closette la méthode de création est hors de la classe standard class et ne peut traiter ce cas spécial.

L'étape suivante concerne la création des méta-objets initiaux, des méthodes standards et des fonctions génériques standards. À ce moment nous trouvons un des problèmes du bootstrap de Closette qui vient de sa circularité: la création de certaines fonctions génériques implique l'existence *à priori* de certaines classes, supposément créées par ces mêmes fonctions génériques.

Ces dépendances circulaires qui en ObjVLisp se trouvent seulement entre l'objet OBJECT et la classe CLASS, dans le bootstrap de Closette sont nombreuses à cause de la quantité d'éléments qui interviennent et de ses interrelations.

On résout le dilemme en remarquant que le mécanisme du lookup au cœur de ces fonctions génériques, n'est pas nécessaire au moment du bootstrap parce que tous les méta-objets initiaux sont des instances immédiates de standard-class et de ses structures connues. En effet, aucune propriété ne requiert l'extensibilité ou la flexibilité caractéristique du lookup dans ces fonctions génériques de base. Lorsque il y aura besoin d'appliquer une fonction générique, elle sera appellée sans utiliser

le mécanisme du lookup. Ce dernier ne sera disponible qu'au moment où toutes les classes, fonctions génériques et méthodes initiales auront été créées. Nous devons donc être capables d'éxécuter defclass, defgeneric et defmethod sans faire intervenir lookup.

```
(define defclass
 ((:root :clone) :new-initials
  #(#(':
      (method (name
               direct-superclases
               direct-slots)
      (ensure-class : name
                    direct-superclases
                    direct-slots)
)))))
```

defclass appelle la fonction ensure-class qui à son tour appelle make-instance. ensure-class est une fonction de support qui s'occupe de vérifier les données (i.e., super-classes existantes, nom de classe non répétée, etc) avant de créer le nouvel objet:

```
(define ensure-class
 ((:root :clone) :new-initials
  #(#(':
        (method (name
                 dir-superclasses
                 dir-slots)

          (((find-class : name) :eq? :true)
          if-true-false
          (block ()
           ("Erreur: on ne peut redefinir la classe" :display))
          (block ()
           (((canonicalize-direct-superclasses
                              : dir-superclasses) :eq? :true)
           if-false-true
           (block ()
            (dir-superclasses :display)
            ("Erreur : superclasses non valides" :display))
           (block ()
            (set! direct-slots
```

```
                (canonicalize-direct-slots : dir-slots))
            (make-instance : name
                             direct-superclasses
                             direct-slots)))))
)))))
```

make-instance est la principale fonction générique dans le processus de création, d'où presque toutes les autres fonctions génériques importantes sont appelées.

Lorsque que make-instance reçoit une classe comme argument au moment du bootstrap, la seule classe existante est standard-class. Ainsi, nous pouvons prévoir quelles méthodes de quelles fonctions génériques seront invoquées par la suite car nous connaissons la structure de standard-class. Cette connaissance nous amène à modifier la fonction ensure-class pour attraper le cas de standard-class et le passer, non à make-instance, mais à make-instance-standard-class. De la même façon, on peut créer des fonctions textitspécialisées pour standard-class tout au long de la séquence à suivre pour finaliser son initialisation. Voici l'addition du traitement de l'exception (lignes 5 - 8) pour standard-class dans ensure-class:

```
1   (define ensure-class
2   ((:root :clone) :new-initials
3     #(#(':
4           (method (name
                    dir-superclasses
                    dir-slots)
5             ((name :eq? 'standard-class)
6             if-true-false
7             (block ()
8               (make-instance-standard-class :))
9             (block ()
10              (((find-class : name) :eq? :true)
11              if-true-false
12              (block ()
13               ("Erreur: on ne peut redefinir la classe" :display))
14              (block ()
15               (((canonicalize-direct-superclasses
```

```
16                                  : dir-superclasses) :eq? :true)
17                 if-false-true
18                 (block ()
19                   ("Erreur : superclasses non valides" :display))
20                 (block ()
21                   (set! direct-slots
22                      (canonicalize-direct-slots : dir-slots))
23                 (make-instance : name
24                                  direct-superclasses
25                                  direct-slots)))))))
)))))
```

Par rapport au bootstrap d'ObjVLisp, le bootstrap de Closette présente une non-homogénité de traitement des objets initiaux et des objets réguliers, en ayant comme conséquence une addition de codage. L'homogenité du bootstrap d'ObjVLisp est possible, au moins en partie, grâce à la structure de ses objets qui permet l'encapsulation de méthodes.

Après avoir passé par make-instance qui crée les trois premiers champs du méta-objet, le processus d'initialisation des classes se complète avec l'appel à méthodes initialize-instance:

```
1  (define make-instance
2   ((:root :clone) :new-initials
3    #(#(':
4          (method (name
                    direct-superclasses
                    direct-slots
                    (vect-init #()) )

5   (set! vect-init (vect-init concat
                       #(#('name
                          #(name
                            #(':initform name)
                            #(':initarg #('name))
                            #(':accessor #('name))))
6                        #('direct-superclasses
                          #( #(direct-superclasses)
```

```
                              #(':initform direct-superclasses)
                              #(':initarg #('direct-superclasses))
                              #(':accessor #('direct-superclasses))))
7                          #('direct-slots
                             #( #(direct-slots)
                                #(':initform direct-slots)
                                #(':initarg #(':direct-slots))
                                #(':accessor #(':direct-slots))))
                           )))
8      (initialize-instance  : name
                              vect-init
                              direct-superclasses
                              direct-slots)
   )))))
```

initialize-instance calcule la liste de précédence de classes (*class precedence list*) et la liste de champs effectifs (*effective slots*). A la fin de cette méthode, on fait l'appel à la méthode finalize-instance.

```
1   (define initialize-instance
2    ((:root :clone) :new-initials
3     #(#(':
4           (method (name
                     vect-init
                     direct-superclasses
                     direct-slots
                     (vect-super #())
                     (vect-slots #())
                     (vect-sub #()) )

5   (set! vect-super direct-superclasses)
6   (0 for-to (direct-superclasses size)
7             (block (i)
8              (set! liste
                    ((direct-superclasses at i)
                              class-precedence-list))
9              (set! vect-super
                    (vect-super concat #(liste)))))
```

Ensuite on élimine les super-classes dupliquées tout en gardant l'ordre:

```
10  (set! vect-super (elimine-duplicats : vect-super))
```

Ici on ajoute le champ class-precedence-list au vecteur initial:

```
11  (set! vect-super #('class-precedence-list
                       #( vect-super
                          #(':accessor #(':class-precedence-list)))))
12  (set! vect-init (vect-init concat vect-super))
```

Ici on crée le vecteur de tous les slots effectifs. Closette a, par opposition à CLOS, des règles très simples d'héritage multiple. Dans le cas où on a deux classes dans la liste de précédence de classes avec le même nom de champ, on prend seulement le champ de la classe la plus spécifique. En d'autres mots, il n'y a pas de mélange de propriétés de champs comme en CLOS.

```
13  (set! vect-slots direct-slots)
14  (0 for-to (direct-superclasses size)
15  (block (i)
16    (set! liste ((direct-superclasses at i)
                                  effective-slots))
17    (set! vect-slots (vect-slots concat #(liste)))))
18  (set! vect-slots (elimine-duplicats : vect-slots))
19  (set! vect-slots #('effective-slots
                       #( vect-slots
                          #(':accessor #(':effective-slots)))))
20  (set! vect-init (vect-init concat vect-super))
```

Finalement nous mettons à jour les liens d'héritage de toutes les super-classes de la nouvelle classe qu'on vient de définir dans cette partie du codage:

```
21  (0 for-to (direct-superclasses size)
22  (block (i)
23    (set! vect-sub ((direct-superclasses at i)
                                  direct-subclasses))
```

```
24    (set! vect-sub (vect-sub concat #(name)))
25    ((direct-superclasses at i) slot-set!
                          direct-subclasses vect-sub)))
26  (set! vect-sub #('direct-subclasses
                          #(#()
                            #(':initform #())
                            #(':initarg #('direct-subclasses))
                            #(':accessor #('direct-subclasses)))))
27  (set! vect-init (vect-init concat vect-sub))
```

Le calcul des méthodes directes est fait à l'aide du champ direct-slots:

```
28  (set! vect-slots #())
29  (0 for-to (direct-slots size)
30  (block (i)
31    (is-method? : (direct-slots at i))
32    if-true
33    (block ()
34      (set! vect-slots
                 (vect-slots concat #((direct-slots at i)))))))
35  (set! vect-slots #('direct-methods
                          #(vect-slots
                              #(':initform #())
                              #(':initarg #('direct-methods))
                              #(':accessor #('direct-methods)))))
```

De même, on ajoute le champ direct-methods au vecteur initial:

```
36  (set! vect-init (vect-init concat vect-slots))
```

Ici, on appel finalize-inheritance qui trait les derniers liens d'héritage en insérant le prototype à sa place dans l'ensemble d'objets:

```
37 (finalize-inheritance : name
                           vect-init)
)))))
```

```
1  (define finalize-inheritance
2   ((:root :clone) :new-initials
3    #(#(':
4          (method (name
                      vect-init
                      (N-Metaobj :nil))

5          (set! N-Metaobj (standard-class :new-initials vect-init))
6          (:global-env :add-var name N-Metaobj))))))
```

Une des dernières méthodes à être initialisée est lookup. Quoique non essentiel au bootstrap, le lookup fait partie indispensable des fonctions génériques de localisation de méthodes dans les champs des classes. En fait, Closette a tout un ensemble de méthodes pour accéder aux champs des objets. Les méthodes d'accès se trouvent hors de tout objet du système, groupés dans une hiérarchie de fonctions qui s'appuient les unes sur les autres. Parmi les plus importantes on y trouve slot-value, qui à son tour s'appuie sur les méthodes slot-location, slot-boundp et slot-exists-p, qui à son tour s'appuient sur des méthodes comme lookup. slot-value est un bon exemple de la force du modèle de Closette (et CLOS): la composition de fonctions. Grâce à la composition des fonctions il est possible d'identifier finement les actions qui interviennent dans une méthode et éventuellement les modifier. Étant donné que ni les méthodes ni les fonctions ne font pas partie d'un objet en particulier, elles sont disponibles pour être redéfinies d'une façon générale.

D'un autre côté, les méta-objets gardent toute leur information dans leur structure, en conséquence il est possible d'accéder à cette information pour transformer le méta-objet à volonté. La complexité structurelle de méta-objets gérés par des fonctions qui sont compositions d'autres fonctions a comme objectif de donner à l'usager le pouvoir d'altérer le langage radicalement.

Les deux facteurs, la composition des fonctions et l'accès à la structure des méta-

objets, contribuent à faire de Closette un modèle approprié pour l'implantation de la réflexion de structure et de comportement.

4.3 Implantation de Closette en Proto-Reflex.

Notre implantation de Closette en Proto-Reflex a confronté deux paradigmes de programmation: la programmation fonctionnelle et la programmation à classes.

Le modèle de Closette s'appuie sur les bases de la programmation fonctionnelle où tout changement de l'état du système est fait à l'aide de l'application de fonctions. L'encapsulation des méthodes dans les objets est faible et la façon d'appliquer une méthode associée à une classe est d'appeler la méthode et de passer comme argument une instance de la classe correspondante. Ainsi, la structure de contrôle du système se trouve hors des objets.

Par contre, Proto-Reflex est un langage à prototypes qui encapsulent les données et les méthodes dans leur structure. La seule façon d'appliquer une méthode est l'envoi des messages. Il n'y a pas de structure de contrôle indépendante des prototypes, car ils réagissent à la réception de messages en satisfaisant la requête selon ses méthodes.

D'abord nous avons traité l'implémentation de la structure des entités de Closette en Proto-Reflex. Tous les éléments de Closette ont été représentés à l'aide de prototypes en Proto-Reflex. Les classes et les méta-objets ont été composés par un vecteur de vecteurs, où le premier élément correspond au nom du champ de la classe et le deuxième à la valeur du champ. En ce qui concerne les méthodes, fonctions génériques et fonctions en général, nous avons utilisé des prototypes avec un seul champ. Ce champ est un vecteur dont le premier élément était le symbole ":" suivi du codage. Cette convention nous a permis de simuler la syntaxe de l'application de fonctions en Closette tout en gardant le style de communication de Proto-Reflex, par l'envoi de messages. Par exemple, la syntaxe de l'application de la méthode defclass en Closette

est:

(*Nom de la méthode Arguments*)

(defclass `name direct-superclasses direct-slots`)

Pour faire le même en Proto-Reflex où la syntaxe est celle de l'envoi de messages, nous utilisons:

(*Nom du prototype-méthode* : *Arguments*)

(defclass : `name direct-superclasses direct-slots`)

Le symbole ":" représente l'application de la méthode (du même nom) de l'objet defclass avec les arguments `name direct-superclasses direct-slots`. Le caractère ":" a été déjà utilisé dans les langages à objets pour signaler une méthode.[3]

Ainsi, avec le mécanisme de l'envoi de messages en Proto-Reflex nous avons implémenté l'application des fonctions de Closette.

Un autre aspect intéressant de Closette modélisé en Proto-Reflex est l'héritage multiple. Les procédures qui composent les mécanismes d'héritage sont en général des opérations de localisation des données et de méthodes sur une structure hiérarchique des classes. Ce sont donc les fonctions de localisation lookup, slot-value et les fonctions qui les composent qui nous avons utilisés pour gérer l'héritage multiple.

Pour gérer la délégation simple, Proto-Reflex s'appuie sur le lien que chaque prototype a vers son prototype parent afin de parcourir toute le graphe d'héritage. En Closette, la complexité structurelle des méta-objets garde presque toute l'information requise pour gérer l'héritage. Les liens qu'une classe a vers ses super-classes se trouvent dans les champs direct superclasses list et class precedence list du méta-objet qui la représente. En plus, les champs dont hiérarchiquement elle a hérité de ses super-classes sont déjà calculés et disponibles dans le champ effective slots. Pour connaître

3. Smalltalk a été le premier langage à utiliser les deux points au début d'une variable pour indiquer qu'elle représentait une méthode et non pas un champ de données.

quels sont ses champs directs et combien d'entre eux sont des méthodes, nous avons les champs direct slots et direct methods. Les mécanismes d'héritage se réduisent donc à des procédures de manipulation des listes. De cette façon, Closette est non seulement bien équipée pour gérer aisément l'héritage multiple, mais d'autres formes plus compliquées de mécanismes d'héritage.

Les difficultés de ce schéma se présentent au moment de la création de la liste de précédences de classes, lorsque le méta-objet est créé. À ce moment, on doit être capable de résoudre les conflits de priorités de super-classes pour positioner correctement les éléments dans la liste. Les règles implantées de résolution des conflits d'héritage dans cette étude ont été les plus simples possibles pour éviter de compliquer inutilement le modèle. Nous avons considéré la priorité positionnelle dans la liste de classes comme critère de séléction dans les procédures de localisation des données et des méthodes. À cette fin nous avons utilisé tant la méthode lookup comme des autres méthodes plus simples qui sont apropriées dans le modèle comme slot-value, is-method? et slot-location.

Chapitre 5

Modélisation de Java en Proto-Reflex

Java est un langage à classes avec une conception modulaire. Nous allons modéliser Java en Proto-Reflex en le considérant composé de deux modules: un interprète de base minimal et un petit module de support de classes.

5.1 Introduction à Java

Le projet initial de Java, dirigé par James Gosling et Patrick Naughton de Sun Microsystems, avait pour but de développer un logiciel portable et attirant qui pourrait être utilisé dans plusieurs types d'architectures de machines et d'appareils électroniques. L'idée a évolué jusqu'à concevoir un langage-plate-forme modulaire qui pourrait être distribué dans un réseau d'ordinateurs. La sécurité et la réponse en temps-réel dans un environnement de processus concurrents a été indispensable pour garantir l'entrée du produit dans le World Wide Web.

En 1994 Naughton et Jonathan Payne ont défini le langage Java pour fonctionner comme une plate-forme de développement des applications dans un environnement

distribué et ont écrit le fureteur d'internet (*"browser"*) *HotJava*. L'équipe de création du langage Java et ses compléments, a mis une emphase spéciale sur la sécurité, la modularité, la robustesse, la portabilité, le traitement simultané (*"multithreading"* et l'optimisation de l'espace mémoire.

La conception du langage Java et son architecture a été inspirée par plusieurs langages, principalement de C++ et C, mais aussi par Smalltalk, Objective C, Eiffel et Cedar/Mesa.

Des caractéristiques avantageuses ont popularisé Java parmi la communauté informatique. Le fait de disposer d'un *garbage collector* automatique et d'une gestion adéquate de la mémoire lui confère une rapidité de réponse peu commune pour un interprète, ainsi qu'une simplification des tâches pour le programmeur. La portabilité de codage (grâce à la traduction en *bytecodes* des applications et à la spécification interne des types des données de base) encourage le partage des librairies et d'applications communes à un groupe d'usagers dans un réseau afin d'optimiser les ressources.

La plate-forme du langage Java, avec une architecture portable est connue comme la Machine virtuelle de Java (*"The Java Virtual Machine"*). Essentiellement, elle est la spécification d'une machine abstraite pour laquelle les compilateurs et interprètes du langage Java peuvent générer du code. La notion d'une Machine virtuelle n'est pas nouvelle, mais le langage Java amène le concept dans un environnement réel, efficace, sécure et distribué. Bien que la Machine virtuelle ait besoin d'un interprète de base spécifique à chaque architecture, elle peut même être implémentée en *hardware*, c'est-à-dire, dans un microprocesseur specialisé (*chip*), sans sacrifier sa portabilité ou ses capacités de réponse. La Machine virtuelle de Java est basée sur la spécification d'interface POSIX [1]

Bien que le langage Java consiste de plusieurs modules distribués dans des environnements hétérogènes, pour des fins didactiques nous allons le considérer pour notre

1. POSIX constitue la définition standard industrielle d'une interface portable pour l'informatique.

modélisation comme composé de deux modules. Le premier module est l'interprète de base et le deuxième le support de classes du langage. Java nécessite ces deux modules comme ensemble minimal pour fonctionner. Dans une première version, la taille de l'interprète de base et le module de support de classes était de 40 Kbytes. Le paquet de base mis à disposition des usagers comprenait les deux modules déjà mentionnés, les librairies standard et un module de support des activités concurrentes (*"thread support"*). Le tout a occupé approximativement 220 Kbytes. En plus, pour exécuter une application Java, l'interprète suppose la présence d'un petit logiciel spécifique à l'architecture de la machine hôte: le système de *runtime.*

Le langage Java reconnaît quatre sortes des types de données: les types des données primitifs (*primitiveType*), de classe (*classeType*), d'interface (*interfaceType*) et de vecteur (*arrayType*). Sauf les types de données primitifs, tout dans le langage Java est un objet. Il y a trois types de données primitives: les nombres (*"numeric type"*), les données booléennes (*"boolean type"*) et les caractères (*"char type"*). Ces derniers sont encodés en *Unicode.*[2] Les types primitifs peuvent être encapsulés dans des objets du système, au besoin de l'usager, à l'aide d'un processus appelé *wrapping.* L'objet résultant appartient à la classe *wrapper class* et a l'avantage de participer dans des méthodes qui ne savent traiter que des objets (ou ses références). Contrairement au type *char*, les chaînes de caractères (*"strings"*) sont des objets à plein droit.

Java est un langage fortement typé. Chaque variable en Java a un type de données associé, appelé *compile-time type.* Les transformations d'un type de données dans un autre type de données ne sont pas supportés. Par exemple, si on veut transformer un nombre réel dans un nombre entier, il faudra faire une "conversion" (*"casting"*) du nombre en indiquant le type à obtenir:

2. Unicode c'est le système de codage de caractères standard créé afin de supporter l'interchange, le procédé et la visualisation du texte de divers langages du monde.

```
nombre-entier = (int)nom-reel
```

Même s'il est possible faire des transformations entre les différents types de données numériques, il n'y a pas de transformations entre ceux-ci et le type de données caractère (*char*) ou booléen.

La façon de créer d'autres types de données et d'objets en Java est la définition de nouvelles classes. Les classes sont des constructions abstraites qui définissent l'état et le comportement des objets concrets qui y sont attachés.

L'opérateur new alloue de la mémoire pour les nouveaux objets mais il n'y a que le *garbage collector* automatique pour la libérer. new peut allouer dynamiquement seulement deux types d'objets selon son type de données: des instances de classes et des vecteurs. La différence entre les classes et les vecteurs est que les premières ont des champs contenant des variables et des méthodes et les deuxièmes ont des *composants* qui contiennent seulement des variables. La classe à laquelle l'instance est attachée est appelée le *runtime type* de l'objet.

La classe Object est la racine du graphe d'héritage. L'héritage en Java est simple: la nouvelle classe avec ses modifications comportementales ne peut avoir qu'une seule superclasse. Même si l'héritage multiple n'est pas explicitement indiqué en Java, il est réalisé à l'aide des *"interfaces"*. L'interface en Java est une définition des méthodes qu'une ou plusieurs classes implémentent. Dans les interfaces, il n'y a pas de déclaration de variables: seulement des méthodes et des constantes. Inversement, une classe peut être liée à plusieurs interfaces. Les interfaces peuvent avoir, comme les classes, des sous-interfaces, et aussi elles ont des types de données *runtime*. Les transformations entre ces types de données et ses substitutions d'objets (selon le *principe de substituabilité* du chapitre 2) sont possibles, mais elles suivent des règles plus complexes qui prennent en considération les mécanismes d'héritage.

Dans le but de contrôler l'accès aux méthodes et variables d'instance d'une classe,

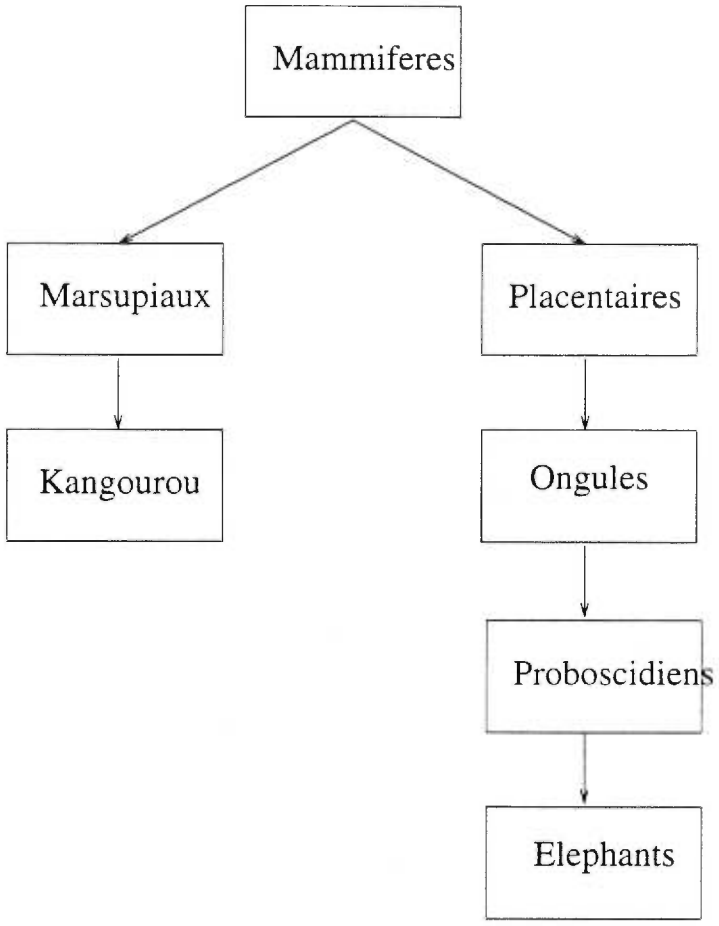

FIG. 5.1 - *Taxonomie en hierarchie de classes*

Java a les mots réservés public, protected et private. Un quatrième niveau d'accès (souvent appelé *"friendly"*) est obtenu si l'implanteur n'utilise aucun de ces mots; dans ce cas, la méthode ou la variable n'est accessible que par des éléments de son *"package"*. Un package en Java est une collection de descriptions de classes similaires entre elles. Le package associe des classes semblables et facilite l'implémentation de ses interrelations.

À part des méthodes et des variables d'instance, une classe peut avoir des variables et des méthodes locales dont la valeur n'est pas copiée dans ses instances. Ces variables et méthodes, appelées "de classe" (*"class methods, class variables"*) sont déclarées avec le mot réservé static et elles sont partagées par toutes les instances de la classe.

Par exemple, en considérant une classe `Mammiferes` qui est la racine d'une hiérarchie taxonomique, elle a des champs communs aux sous-classes, avec des valeurs initiales communes à tous les mammifères: 4 pattes (on suppose que les mammifères sont tous tétrapodes), 2 oreilles, etc. Comme les valeurs initiales peuvent changer dans des cas particuliers (un animal peut perdre une patte) ces champs doivent avoir une copie dans chaque instance. Par contre, la variable de classe `total-mammiferes`, qui enregistre le nombre total de mammifères, est unique. Localisée seulement dans la classe `Mammifères`, elle est marquée avec le modificateur `public`, donc accessible même au code hors du `package`:

```
class Mammiferes extends Object {
   protected int n-pattes;
   protected int n-oreilles;
   static public int total-mammiferes;
            ...
}

   Mammiferes() {          // Le constructeur de la classe
     n-pattes = 4;         // initialise ici les valeurs des
     n-oreilles = 2;       // champs de la classe.
}
```

Les constructeurs (*constructors*) sont des méthodes spéciales qui ne sont pas héritées et qui servent à initialiser les champs d'une classe. Ils portent le même nom que sa classe. Le nombre d'arguments entre ses parenthèses et ses types déterminent le constructeur à utiliser dans un appel.

Une technique alternative pour initialiser les nouveaux objets sont les méthodes d'accès *accessor methods*. Ils sont utilisés pour accèder aux variables d'instance declarées `private` et ainsi permettre la lecture de ses valeurs au code à l'extérieur de la classe. Les méthodes d'accès évitent de cette façon les modifications non-permises des champs protégés.

Une des sous-classes de la classe `Mammifères`, la classe `Marsupiaux` définit trois champs. Comme le niveau d'accès des champs `n-pattes` et `n-oreilles` de sa super-classe est protected, `Marsupiaux` et toutes ses sous-classes héritent ces deux champs, mais ils ne sont pas accessibles hors du package.

```
class Marsupiaux extends Mammiferes {
   protected double taille;
   protected double poids;
   static private String[] habitat;
            ...
}

class Kangourou extends Marsupiaux {
   public String couleur;
            ...
   public void Sauter (int distance){
            ...
   }
}
```

Puisque le vecteur `habitat` est marqué private, il ne peut être ni hérité ni accédé par aucun code hors de sa classe. La variable `couleur` et la méthode `Sauter` ont un modificateur d'accès public, qui permet l'accessibilité par des références d'objets dans le code de ses sous-classes et aussi par d'autres methodes (par exemple, graphiques) hors du ce package.

L'instance de cette hiérarchie de classes est un objet concret, un kangourou gris appelé "Skippy"qui a cinq champs au total, dont les deux premiers déjà initialisés. Au moment de l'allocation de l'espace mémoire avec l'opérateur new, s'il y a des variables d'instance ou des vecteurs qui n'ont pas une valeur initiale spécifiée, Java leur donne la *valeur par défaut selon son type*. Dans notre exemple, les doubles auront zéro positif (0.0d) et les types de référence null.

```
Kangourou Skippy;
```

```
Skippy = new Kangourou();

Skippy.couleur = gris;
Skippy.taille = 1.05;
Skippy.poids = 40.5;
```

Si les méthodes de classe sont accessibles par toutes les instances de la classe, elles n'ont toutefois pas accès aux variables d'instance. Cette interdiction existe afin d'éviter le mélange des opérations au niveau de classes avec des opérations au niveau des instances. Malgré les barrières, il est assez facile de passer un objet (ou sa référence, c'est-à-dire, son "pointeur" en Java) à une méthode de classe et utiliser les variables d'instance au niveau de classe. Donc, la restriction est plus de style de programmation que de fond, mais elle est spécialement appuyée par les mécanismes de contrôle d'accès en Java. Le modificateur **protected** assure que l'héritage des champs soit réalisé *seulement en ligne directe* tout le long des sous-classes. Plus précisément, un champ marqué **protected** ne peut être accédé par une sous-classe qu'avec une référence d'objet possédant le même type ou le type d'une de ses sous-classes. Pour illustrer ce fait, nous montrons la figure 5.1 où la classe `Mammifères` a deux sous-classes: `Marsupiaux` et `Placentaires` (la troisième sous-classe, non montrée, est la `Prototheriens`, fossiles à exceptions des ornithorhynques). `Marsupiaux` a une seule sous-classe dans notre exemple, `Kangourou`, mais `Placentaires` comprend 11 ordres, dont seule la sous-classe `Ongulés` est dessinée avec un de ses 5 sous-ordres: la sous-classe `Proboscidiens`, qui est la superclasse des éléphants. Le champs **protected** `n-pattes` est accessible au code de la classe `Placentaires` avec une référence *au moins du type* `Placentaires`, mais aussi du type `Ongulés`, ou de ses sous-classes. Par contre, ce code de la classe `Placentaires` ne peut accéder au champ `n-pattes` avec une référence du type `Marsupiaux`. Cette restriction évite que les références d'objets qui utilisent les champs **protected** d'une ligne d'une hiérarchie soient "mélangées" avec les références d'objets d'autres lignes de la même hiérarchie. On dirait que les types

ont des lignages différents.

À l'inverse, si dans la classe `Marsupiaux` il y a du code avec une référence du type `Mammifères`, la seule façon d'accéder au champs `n-pattes` est de faire une "conversion" (*casting*) du type plus général (`Mammifères`) vers le type `Marsupiaux` et d'utiliser cet objet résultant. En faisant cette sorte d'accès avec conversion on doit être soigneux avec la relation entre les classes: tous les marsupiaux sont des mammifères, mais pas tous les mammifères sont de marsupiaux.

Une notion importante pour la conception de programmes en Java est celle de la classe abstraite. Une classe (ou mieux, superclasse) abstraite est une classe qui groupe des caractéristiques communes aux autres sous-classes mais qui ne sert pas à créer d'instances. Ses méthodes sont des généralisations des comportements communs de ses sous-classes. Ils ne sont pas conçus pour être utilisés directement, mais modifiés à l'aide du mécanisme de l'héritage dans ses sous-classes pour les rendre spécifiques à chaque sous-classe.

Nous déclarons une superclasse abstraite en Java avec le mot réservé abstract. Les superclasses abstraites ne peuvent avoir d'instances concrètes, mais seulement des sous-classes, qui ont à leur tour des instances. Par exemple, la classe abstraite `Mammifères` est celle qui décrit des caractéristiques communes aux mammifères et où tous les individus sont des animaux:

```
abstract class Mammiferes extends Object {
 final String individu = animal;          // individu est une
 protected int n-pattes;                  // constante et
 protected int n-oreilles;                // total-mammiferes
 static public int total-mammiferes;      // est une variable
          ...                             // de classe.

 Mammiferes() {
  n-pattes = 4;
  n-oreilles = 2;
            ...
 }                                    // la methode abstraite
```

```
                                        // peut avoir une
 abstract void dessinerAnimal();        // definition partielle
                                        // ou comme ici,
                                        // n'avoir aucune
}                                       // definition.
```

Ici, le mot réservé final déclare la chaîne de caractères individu comme une constante qui sera accédée par toutes les sous-classes de la classe abstraite. Elle doit être initialisée au moment de sa définition.

La méthode abstraite dessinerAnimal n'est pas définie dans cette classe abstraite; elle est seulement declarée. Chaque sous-classe qui a besoin de dessiner un animal spécifique devra définir sa propre méthode dessinerAnimal.

```
class Ongules extends Mammiferes {
 protected Object type-onglon;
            ...
}

class Proboscidiens extends Ongules {
 protected Object type-trompe;
            ...
}

class Elephants extends Proboscidiens {
 protected String couleur;

 Elephants() {                          // Initialisation des
  type-onglon = sabot;                  // valeurs a l'aide d'un
  type-trompe = trompe-elef;            // constructeur.
  couleur = "gris";
 }

 public void dessinerAnimal() {
  private int n-parts = 4;
  private int[] quantite = new int[n-parts];
  private Object[] part = new Object[n-parts];

  part[1] = silhouette-pattes-elef;     // On suppose que nous
  quantite[1] = n-pattes;               // disposons des silhouettes
```

```
  part[2] = siluette-oreilles-elef;    // specifiques pour
  quantite[2] = n-oreilles;            // dessiner des partiess
  part[3] = type-onglon;               // des animaux.
  quantite[3] = 4;
  part[4] = type-trompe;
  quantite[4] = 1;

  for (int i = 0; i < n-parts; i++)    // La methode dessiner
     dessiner(part[i], quantite);      // trace la silhouette de
                                       // l'objet passee comme
                                       // argument.

  metCouleur(couleur);                 // metCouleur peint l'objet
                                       // avec la couleur passee
 }                                     // comme argument.
}
```

De la même façon que la classe `Proboscidiens` implémente sa méthode spécifique au dessin des éléphants, une autre possible classe `Périssondactyles` (d'où appartient le cheval) pourrait redéfinir la méthode abstraite spécifiquement selon ses besoins.

L'instance de la classe `Proboscidiens` est un éléphant appelé "Clyde" qui nous pouvons dessiner:

```
Proboscidiens Clyde = new Proboscidiens;

Clyde.dessinerAnimal();
```

La *signature* d'une méthode (où d'un constructeur) est composée de son nom, ses arguments et leur types. Il est possible de définir plusieurs fois une même méthode pour décrire différents comportements selon les différents arguments qu'elle reçoit. Comme le nom reste égal, le compilateur de Java compare le nombre et le type des arguments pour discriminer la méthode à appliquer. La propriété des objets de se comporter de différentes façons sous différentes circonstances est connue comme *polymorphisme*. Une forme d'extension du comportement d'une méthode en Java est

de fournir plus d'un code de méthode avec le même nom mais avec une signature différente. Ce mécanisme est appelé *overloading.*

Il y a essentiellement deux formes d'héritage en Java: *overloading* et *overriding.* Le premier, *overloading* est le mécanisme expliqué précédement où on fournit plus d'une méthode avec le même nom mais avec signature différente. Le mécanisme d'*overriding* est le remplacement du code d'une méthode dans une superclasse par une nouvelle méthode locale. Le cas de la substitution d'un champ de variable dans une sous-classe est plutôt appelée *shadowing.*

Dans Java, this est la référence à l'objet courant. this est une référence implicite, la référence à soi-même de l'objet qui est en train de recevoir un message. super est une référence à l'objet courant comme une instance de sa superclasse. Lorsque une invocation de super.*method* est faite, le système de *runtime* de Java remonte la hiérarchie d'héritage pour trouver la superclasse et utiliser *son implémentation* de la méthode *method.*

5.2 Modélisation du module de support à classes

Une des forces de Java est son module de support des classes, qui permet de partager des classes appartenant à des librairies communes et de les charger à travers d'un réseau d'ordinateurs comme Internet. Il garde trace de classes qui sont actives et travaille directement avec le compilateur et la Machine virtuelle (interprète de base) pour garantir l'intégrité des classes et la disponibilité de l'espace mémoire.

Dans d'autres langages à objets, comme C++, la modification d'un champ dans une classe (surtout une superclasse) ou l'addition d'une variable d'instance implique la re-compilation de toutes les classes qui ont une référence à la superclasse. Sinon, le programme ne marche pas. En Java, l'addition ou modification d'un champ est traitée par étapes. Le compilateur de Java compile le code juste avant d'arriver aux références

numériques (c'est-à-dire, pas profondement et chaque objet séparément), et passe *de l'information symbolique* sur les références des nouveaux objets au Vérificateur de bytecode (*Bytecode Verifier*) et à l'interprète de base. Ceci fait la résolution finale de noms de classes une seule fois, au moment de lier les classes. Une fois les noms résolus, les références sont écrites comme une extension (*offset*) numérique. Comme la disposition de l'espace mémoire n'est pas faite par le compilateur mais par l'interprète de base, elle est retardée jusqu'au moment du *runtime*. Bien sûr, Java est protégé avec plusieurs mécanismes de vérification de types au moment de la compilation (*compile-time type verification*) qui permettent la gestion des nouvelles références d'une façon symbolique avec confiance. En conséquence, les modifications ou additions de champs peuvent être faites sans affecter le code existant.

Ici, nous allons modéliser le module de support à classes à l'aide d'une structure de Proto Reflex, les environnements de variables, qui permettent la gestion de classes actives. Le module aura certains comportements essentiels mais il ne pourra charger des classes définies au delà de la machine hôte, comme en Java, parce que cette situation tombe hors des limites de ce travail. Cependant, cet objet de Proto Reflex pourrait être étendu à l'aide de mécanismes de la délégation pour avoir aussi ce comportement.

Les environnements de variables de Proto Reflex apportent certains avantages pour modéliser le module de support à classes de Java. Le premier avantage est que chaque environnement est un objet distinct. Il peut être manipulé séparément des autres objets. Le deuxième est la création de fermetures avec une durée de vie contrôlable et potentiellement illimitée, ce qui nous fournit une structure semblable à celle où les classes en Java sont traitées. Un troisième avantage est que son accès est réflexif, donc il est possible d'examiner, d'ajouter et de redéfinir ses composants d'une façon semblable à ce que fait Java au *runtime*.

Une nouvelle classe définit une espace mémoire nommée. Elle est composée de

champs declarés et de ceux hérités de toutes ses superclasses sauf ceux marqués private, les constructeurs et ceux qui ont le même nom que les champs de la nouvelle classe, selon le mécanisme du *"shadowing"*.

Nous avons déjà mentionné que toutes les classes ont un objet du type Class qui les représente au moment du *runtime*. La classe Class est la racine de cette hiérarchie d'objets, qui est souvent comparée à un système de navigation de types. Ce système devient partie de l'environnement du programme et, en plus d'apporter des mécanismes de création des objets, présente plusieurs avantages pour le déverminage éventuel, l'auto-documentation et pour la consistance générale du programme. Les objets servent à donner de l'information sur la classe et les interfaces auxquelles elles sont liées. Ce dernier aspect sera modélisé en Proto Reflex à l'aide d'un champ qui garde le vecteur d'interfaces implémentées par la classe représentée. Cette structure est aussi utile pour modéliser l'héritage multiple de Java.

Pour simuler le système à classes dans un langage à prototypes, nous avons implémenté deux prototypes pour chaque classe: un objet qui définit la structure de la nouvelle classe et un autre prototype qui a le rôle de l'objet qui garde l'information sur la classe et ses relations avec les interfaces. Cet objet représente la classe dans le système de types et il est une instance de la classe Class.

new est l'opérateur le plus commun pour créer des objets en Java. Une variation du processus de création est réalisée avec une vraie méthode, newInstance. Malgré son orientation vers les objets, Java présente une grande quantité d'opérateurs et modificateurs qui ne sont pas des objets et qui ont été implantés *en dehors d'objets*, comme par exemple, public, static et même new. Étant donné que dans Proto Reflex il y a une absence de ces éléments (tout est un prototype), nous avons modélisé l'opérateur new comme une méthode pour la création des nouveaux objets. La méthode newInstance a été implantée afin de créer des nouvelles instances. Les définitions par défaut de new et newInstance sont dans l'objet Class.

Pour céer une nouvelle classe, la méthode new est invoquée avec un message contenant le mot reservé new, le nom de la classe à créer, la liste d'interfaces qu'elle implémente et sa structure, c'est-à-dire, les champs définis par l'usager:

```
(Object new 'Mammiferes #('graphique)
                          #(#('n-pattes #(protected 4))
                          #('n-oreilles #(protected 2))))
```

En Proto Reflex, nous avons modélisé le processus à l'aide d'une méthode. Les champs peuvent être précedés ou non par les modificateurs public, private, protected, static ou final pour marquer les variables d'instances, de classe et les niveaux d'accès principaux. Cet ensemble minimal de modificateurs permet une implémentation sommaire des règles de base de l'héritage du Java au moment de la création des sous-classes.

```
1(define Class
2 ((:root :clone) :new-initials
3  #(#('class 'Object)
4    #('new
5           (method (Nom-Classe
                    interfaces
                    vect-Corps-Classe
                    (vect-herite     #())
                    (vect-struct     #())
                    (vectInit        #())
                    (vect-obj-classe #())
                    (vect-Classe     #())
                    (NouvClasse      :nil))

6             (set! vect-herite (super vect-struct))
7             (0 for-to (vect-herite size)
               (block (i)
8               ((((vect-herite at i) at 1) :eq? "method")
                if-false
                (block ()
9                ((((((vect-herite at i) at 1) size) > 1)
                 if-false-true
```

```
                (block ()                    ; Pas de modificateurs
10               (set! vectInit
                    (vectInit concat (vect-herite at i)))
                )                            ; fin de non-modificateurs
                (block ()                    ; Modificateurs
11               (set! modificateur (((vect-herite at i) at 1) at 0))
12               (((modificateur :eq? "public") or
                        (modificateur :eq? "private"))
                 if-true
                 (block ()
13                  (set! vectInit
                          (vectInit concat (vect-herite at i)))
                 ))
                ))                           ; fin de Modificateurs
               ))                            ; fin de non-method
              ))                            ; Loop de vecteur de Classe

14          (set! vect-struct (vectInit concat vect-Corps-Classe))
15          (set! vect-obj-classe #(#('classe self)
                                    #('interfaces interfaces)
                                    #('type vect-Corps-Classe)))
16          (set! vect-Classe  #(#('nom Nom-Classe)
                                  #('class vect-obj-classe)
                                  #('vect-struct vect-struct)))
17          (set! vect-Classe (vect-Classe concat
                                           vect-Corps-Classe))
18          (set! NouvClasse (self :new-initials vect-Classe))
19          (:global-env :add-var Nom-Classe NouvClasse)))
```

Le processus commence pour traiter les liens et la structure hérités du nouvel objet: sa classe et le type de champs hérités ou partagés (lignes 6 au 14). Cette information (avec les interfaces) est organisée pour créer l'*objet de classe*. Il est composé par trois champs (ligne 15) et un lien vers la classe Class. Le premier champ est la classe, le deuxième le vecteur d'interfaces et le troisième un vecteur qui représente le type, c'est-à-dire, la structure de la classe. Ce dernier élément est utilisé dans la modélisation pour la vérification de conversions de type et de références entre les objets. Ensuite, la classe est créée (ligne 16) aussi avec trois champs et deux liens. L'information dans cet objet est son nom (*l'objet de classe*, en concordance avec Java, n'a pas de nom),

le champ contenant un lien vers son *objet de classe*, et un champ contenant le vecteur de vecteurs qui est la classe même. La nouvelle classe a un lien de délégation vers le récepteur du message (self, ligne 18), afin d'utiliser la délégation dans le système pour implémenter l'héritage.

La méthode newInstance sert à créer des instances. Elle utilise le constructeur sans arguments (*no-arg constructor*) pour initialiser le nouvel objet. Dans notre modélisation, le constructeur sans arguments utilise le champs vect-struct (un vecteur de construction localisé dans l'*objet de classe*) pour construire et initialiser les instances (ligne 3). Après, l'objet est crée au ligne 11.

```
1    #('newInstance
2           (method( Nom-Instance
                     (vectInit     #())
                     (vect-Classe  #())
                     (modificateur :nil) )

3             (set! vect-Classe (self vect-struct))
4             (0 for-to (vect-Classe size)
               (block (i)
5               ((((vect-Classe at i) at 1) :eq? "method")
                if-false
                (block ()
6                (((((vect-Classe at i) at 1) size) > 1)
                 if-false-true
                 (block ()
7                 (set! vectInit
                     (vectInit concat ((vect-Classe at i) at 1)))
                 )
                 (block ()
8                 (set! modificateur (((vect-Classe at i) at 1) at 0))
9                 (((modificateur :eq? "public") or
                           (modificateur :eq? "private"))
                  if-true
                  (block ()
10                  (set! vectInit
                        (vectInit concat
                              (((vect-Classe at i) at 1) at 1)))
                  ))))))))
```

```
11              (set! NouvInst (self :new-initials vectInit))
12              (:global-env :add-var Nom-Instance NouvInst)))
```

La méthode new réserve de l'espace mémoire pour le nouvel objet et initialise ses champs selon les constructeurs de la superclasse. S'il n'y a pas une initialisation explicite, l'objet est initialisé avec des valeurs par défaut (en Proto Reflex le défaut pour les objets est :nil, en Java est null). Une fois crée, une classe étendue doit utiliser les constructeurs de ses superclasses pour son initialisation car la partie de l'objet contrôlée par la superclasse doit être construite ou initialisée correctement. À part des constructeurs spécifiques, la classe peut se servir de l'expression super() et ainsi appliquer les constructeurs ou structures définies dans la superclasse. Nous utilisons le même critère pour initialiser le nouvel objet dans notre modélisation: on cherche dans la superclasse l'information contenue dans le champ vect-struct (ligne 6 du code de new), après nous analysons le contenu des champs (méthodes, modificateurs, etc) et ensuite nous construisons la structure du nouvel objet. Cette construction ou "coquille" sera utilisée par la méthode newInstance pour initialiser les instances (ligne 3 du code de newInstance).

La méthode ClassLoader est la simulation dans un prototype de la partie du module de support à classes qui cherche et apporte des classes demandées par l'usager mais qui ne sont pas présentes dans le package ou dans la machine hôte. ClassLoader en Java permet de charger le code (en *bytecode*) des classes localisées dans d'autres systèmes à travers l'Internet. Nous n'avons pas modélisé cet aspect et la méthode ClassLoader est mentionée comme une partie de la classe Class pour des raisons didactiques.

À partir de ce moment, la classe est prête à fonctionner avec le reste des objets du système et le contrôle passe à la Machine virtuelle.

5.3 Modélisation de l'interprète de base

La classe Object est la racine de la graphe d'héritage et tous les objets peuvent accèder ses méthodes. Elle comprend les méthodes equals(Object obj), hashcode(), clone(), getClass() et finalize(), mais notre modélisation minimale implémente seulement les suivantes:

protected Object clone() Cette méthode crée un nouvel objet qui est la copie du récepteur.

public final Class getClass() Cette méthode retourne l'objet de type Class qui représent la classe du récepteur au moment de l'exécution (*runtime*) du programme.

La méthode Object.clone est la méthode de base pour créer un nouvel objet dont l'état initial est une copie de l'état courrant du receveur. Elle copie tous les champs de l'objet à cloner vers le nouvel objet. Ce comportement est très général et peut ne pas convenir dans tous les cas. En conséquence, Java offre une interface, l'interface Clonable, laquelle doit être implémentée dans une classe pour obtenir un résultat différent du défaut. L'argument pour avoir une interface Clonable est que les structures des données des objets peuvent devenir très complexes, donc clone doit pouvoir s'adapter aux caractéristiques particulières des objets pour les copier efficacement. Notre modélisation en Proto Reflex simplifie le problème. Le clonage en Java peut devenir énormément complexe parce que le langage impose au programmeur une gestion stricte des données selon ses types. Par contre, Proto Reflex est un langage à prototypes non-typé où tout est un objet. La vérification des types est faible et les types de données primitifs ont des interactions avec les objets à plein droit du système sans avoir besoin d'être transformés en objets (*wrapping*).

Afin de modéliser la méthode clone, nous nous sommes appuyés sur la méthode :clone du Proto Reflex, car elle a un comportement identique au comportement désiré. Après avoir crée le nouvel objet (ligne 7), nous associons le nom passé comme argument avec le nouvel objet. Ensuite, nous ajoutons le nouvel objet au environnement global (ligne 8):

```
1 (define Object
2  (Class :new-initials
3   #(#('class 'Class)
4     #('clone
5                (method (Nom-Objet)
6                        (Objet-clone :nil))

7                 (set! Objet-clone (self :clone))
8                 (:global-env :add-var Nom-Objet Objet-clone))
```

Comme toutes les références d'objets sont polymorphiques de la class Object, ceci devient la classe générique de n'importe quel objet:

```
Object refAnimal = new Ongules;

refAnimal = new Marsupiaux;
```

refAnimal peut recevoir références de Ongulés et de Marsupiaux parce qu'il est du type Object. Entre eux, les deux classes n'ont aucune relation sauf une superclasse commune.

La méthode getClass est déclarée en Java comme public et final, donc elle n'a aucune restriction d'accessibilité, mais elle ne peut être redéfinie. Ce comportement est obtenu dans notre modélisation en ajoutant le modificateur final dans le vecteur de structure de la classe Class:

```
#('vect-struct #(... #('getClass #(final #()))...))
```

getClass cherche simplement de l'information sur la classe de l'objet qui est dans l'*objet de classe* (ligne 4). Le lien vers l'*objet de classe* est dans le champ class (ligne 3).

```
1   #('getClass
2               (method ((obj-classe   :nil)
                         (nom-classe   :nil))

3                 (set! obj-classe (self class))
4                 (set! nom-classe ((obj-classe at 0) at 1))
5                 (nom-classe :display)))
```

L'implémentation des objets Class et Object constitue le *bootstrap* de cette modélisation. En utilisant cette configuration minimale il est possible de créer des nouveaux objets *à la Java* et obtenir des réponses selon son comportement. Seul les objets essentiels sont présents dans la modélisation, mais la structure d'objets définie est suffisamment flexible comme pour étendre le système jusqu'à avoir un système avec un comportement semblable à celui du langage Java.

5.4 Discussion et critique de l'implantation

Une conception assez répandue des systèmes à classes est de disposer d'un graphe d'héritage et d'un graphe d'instantiation. Chaque graphe a comme racine un objet spécial: l'objet Object (ou Root) et l'objet Class qui forment entre eux une *bootstrap*, c'est-à-dire, une relation fonctionnelle pour démarrer le système et maintenir son fonctionnement. Notre modélisation, en suivant la conception du langage Java, a aussi un graphe d'héritage, constitué par les liens de délégation des prototypes. Ce graphe est clair, facile à suivre et ne s'éloigne pas de l'implémentation typique. Par contre, l'autre graphe, qui nous appelerons d'instantiation parce qu'il garde cette fonction parmi d'autres, connecte les *objets de classe* jusqu'à la classe *Class*. Ce graphe est

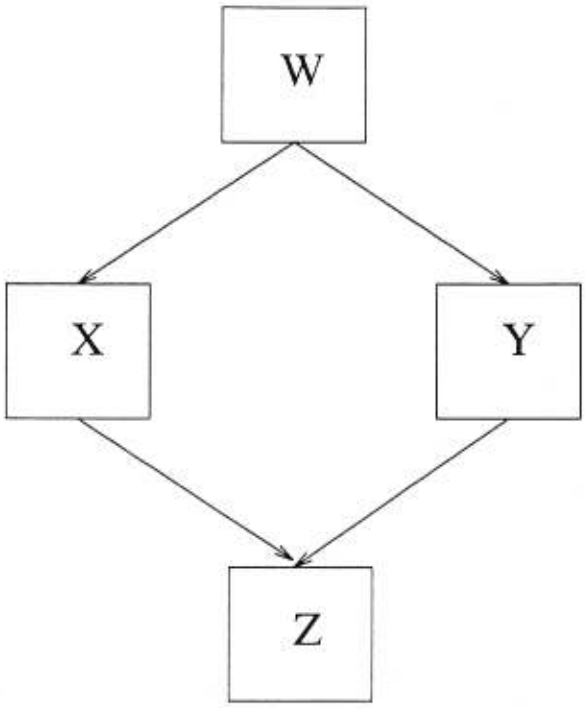

FIG. 5.2 - *Type d'héritage "de diamant".*

utile pour implémenter les liens vers les interfaces de Java, l'héritage multiple, des capacités de réflexion et les processus de création de nouvelles classes et instances. L'implémentation des *objets de classe* avec deux liens physiques dans notre modèle à prototypes nous a permis d'apprécier les avantages de cette structure. Cette hiérarchie d'*objets de classe* est particulièrement bien adaptée pour résoudre les conflits d'héritage multiple, car elle a toute l'information sur les classes et les interfaces que ses classes utilisent. Ainsi, ce graphe d'*objets de classe* permet la flexibilité de Java d'avoir deux sortes d'objets pour gérer l'aspect abstrait de comportement des programmes: les classes abstraites et les interfaces. Si ce scénario double complique le choix d'utilisation par le programmeur, en réalité il offre toute une gamme de combinaisons pour exprimer le mécanisme de l'héritage multiple sans ses inconvénients. L'exemple suivant aidera à éclaircir cette situation.

En considérant un type d'héritage multiple "de diamant" comme il est illustré dans la figure 5.2, on arrive facilement à la nécessité de résoudre le conflit suivant: Duquel

objet on doit héritier la méthode `Zref.Wchamp`? De l'objet `X` où de l'objet `Y`? Dans une situation typique, les deux objets héritent le champ et en plus ont le droit d'étendre ou de redéfinir son comportement. Dans l'approche de Java, déclarer un de deux objets `X` ou `Y` comme une interface résout le conflit: les interfaces n'ont pas de champs (seulement de déclarations et des constantes final), donc la seule possibilité de trouver le champ `Zref.Wchamp` est dans la classe. Sans risque de tomber dans un conflit, la classe pourrait être même abstraite, donc capable d'implémenter partiellement le champ. Dans ce cas, si aucune de deux objets n'implémente le champ, il doit être implémenté dans l'objet `Z`.

Les classes abstraites et les interfaces travaillent en combinaison pour supporter l'héritage multiple, mais seulement les deuxièmes permettent cette forme d'héritage. En effet, on peut implémenter plusieurs interfaces dans une même classe, mais une classe peut étendre seulement une classe. En compensation, plusieurs mécanismes de l'héritage simple sont contrôlés à l'aide des classes abstraites. La seule déclaration d'une classe abstraite dans une hiérarchie de classes "concrètes" peut invalider l'héritage à toute une section de l'arbre. Si bien les classes abstraites se trouvent le plus souvent dans les niveaux les plus hauts d'une hiérarchie et n'ont que des sous-classes aussi abstraites jusqu'à avoir une sous-classe *concrète*, le schéma contraire, celle d'une classe concrète qu'a une sous-classe abstraite est aussi possible. Le but de cette classe abstraite est de redéfinir (*override*) certaines méthodes et les déclarer abstract, en transformant une méthode abstraite dans une méthode concrète. La signification de ce changement est de rendre invalide une partie de l'implémentation de défaut d'une superclasse à tout un ensemble de sous-classes et instances.

Ce contrôle d'héritage est semblable à la substitution d'un champ selon le mécanisme de redéfinition (override) et en plus le déclarer final. Dans ce cas, le champ (ou toute une classe) est considéré comme "la version finale" et ne participe pas au processus d'héritage.

Aussi, mais d'une façon indirecte, l'héritage entre classes est affecté par le contrôle d'accès. Les modificateurs d'accès protected et private restreignent l'héritage et règlent ainsi les formes d'encapsulation des objets en Java. De même, et à cause de restrictions de conversion, il est important pour un programmeur de savoir en tout temps le type (c'est-à-dire, la classe) des objets qui participent dans le mécanisme de l'héritage.

Dans un langage comme C++, il est assez compliqué de savoir au moment de l'exécution (*runtime*) la classe à laquelle l'objet appartient et sa structure. En Java l'usager peut savoir la classe de ses objets grâce à la représentation au moment de l'exécution (*runtime representation*) des classes.

Comme nous avons déjà mentionné, toutes les classes en Java ont un *objet de classe* qui les représente. Il est une instance de la classe Class et contient des définitions et de l'information sur la classe qui représente. Nous avons modélisé ces *objets de classe* à l'aide de prototypes avec trois champs. Le premier contient la superclasse de la classe représentée. Le deuxième a le vecteur d'interfaces que la classe implémente et qui est nécessaire pour réaliser l'héritage multiple. Le troisième contient la structure de la classe, qui est techniquement le type de la classe. Cette représentation au moment d'exécution permet à l'usager de savoir la classe de l'objet, ses interfaces et son type. Cette information est indispensable pour résoudre des conflits d'héritage et pour réaliser la vérification de types de Java.

Le schéma d'avoir d'*objets de classe* qui gèrent l'information sur les classes permet une grande liberté de modifications, d'adaptation et de portabilité aux objets du système. Nous pensons que cette structure apporte aussi une base suffisamment flexible pour implémenter des comportements réflexifs. La réflexion de structure s'obtient à l'aide de la représentation que les *objets de classe* ont déjà de la structure de la classe à son champ `vect-struct`. L'autre aspect, la réflexion de comportement est possible d'implémenter avec la définition de quelques méthodes pour gérer la réification et la réflexion des objets. Ces opérations de base sont nécessaires pour établir une réflexion,

mais il y a aussi d'autres considérations à ajouter dans un modèle réflexif de Java, comme la résolution de conflits de types, modificateurs et de securité au moment de rendre disponible l'information, etc.

Bien que les capacités de réflexion ne fassent pas partie des caractéristiques de base de la Machine virtuelle du langage Java, elles ont été considerées dans d'autres parties du système dès le début. Par exemple, dans le développement du projet Java, (appelé à ce moment-là *"Glasgow"*), il y avait la description d'un mécanisme de délégation (*Aggregation/Delegation mechanism*). Finalement un *API (Application Programming Interfase)*, le *JavaCoreReflection*, a été ajouté à la Machine virtuelle de Java pour supporter de l'introspection sur les classes et objets, donner de l'information réflexive sur les objets et manipuler de façon sécuritaire les objets réflexifs.

Il y a d'autres éléments dans le système Java qui ont aussi des propriétés réflexives: les unités de code portable appelées *JavaBeans*. Conçus pour être réutilisables, les *JavaBeans* sont des composants écrits en Java et manipulables dans un environnement graphique. Ils ont, par défaut, les caractéristiques suivantes:

Introspection Permet à une interface graphique (*Builder Tool*) d'analyser comment un *Bean* fonctionne.

Customization Permet l'utilisation d'une interface graphique pour manipuler les *Beans*.

Events Permet aux *Beans* de la communication et de la connexion entre eux.

Properties Permet aux utilisateurs l'exploitation des *Beans* et sa programmation.

Persistance Permet la recupération d'un *Bean* après son utilisation dans l'interface graphique.

Les *JavaBeans* sont plutôt des composants que de micro-applications. Sa concep-

tion privilégie la programmation modulaire et l'intégration des applications distribuées dans un réseau d'ordinateurs.

Même si notre implantation en Proto Reflex des principaux aspects de Java présente le désavantage d'être un peu inefficace, elle démontre que la modélisation est parfaitement possible. L'inefficacité dépend surtout du fait de compter seulement avec les prototypes et le mécanisme de la délégation comme éléments de la modélisation. La vérification de types est particulièrement complexe parce que Proto Reflex ne contient pas toutes les restrictions de types et de sécurité existantes en Java. De plus, Proto Reflex n'a pas le schéma de vérification de types de Java (*au deux temps*, au moment de la compilation et du *runtime*) car il est un interprète.[3]

Nous nous sommes contentés ici d'un modèle minimal qui permet des améliorations futures et qui supporte presque tous les aspects de Java. La structure proposée des objets et la modélisation de ses relations montrent aisément qu'il est possible d'implémenter un système à prototypes avec le comportement de base qui caractérise le langage Java.

3. Le nouveau compilateur de Proto Reflex, par Marco Jacques, sera disponible bientôt.

Chapitre 6

Discussion.

Le processus de modélisation des classes par prototypes nous oblige à considérer toutes les entités d'un système dans un seul niveau. L'effet "d'applatissement" a été renforcé par le fait que dans Proto-Reflex presque tous les éléments du système sont des prototypes.

Tout en restant à un seul niveau et en n'utilisant que des prototypes attachés par liens de délégation, nous avons pu représenter des classes, métaclasses, instances, méthodes et d'autres objets comme des dictionnaires.

Le premier effet d'enlever les classes pour ne travailler qu'avec des prototypes a été un manque d'organisation parmi les entités. Ce problème organisationnel était évident au moment de passer de l'héritage à la délégation. Les relations d'héritage entre les classes établissent aussi des "règles implicites" de partage d'information. En gardant seulement la représentation abstraite d'un objet, la classe empêche le partage d'informations entre ses instances, qui représentent des cas concrets et particuliers. Par contre, dans les langages à prototypes, l'information partagée (entre les instances!) est de type concret et de "cas particuliers".

Par exemple, la façon de représenter un ensemble d'éléphants dans un système à prototypes et dans un système à classes est illustrée dans la figure 6.1. Elle met en

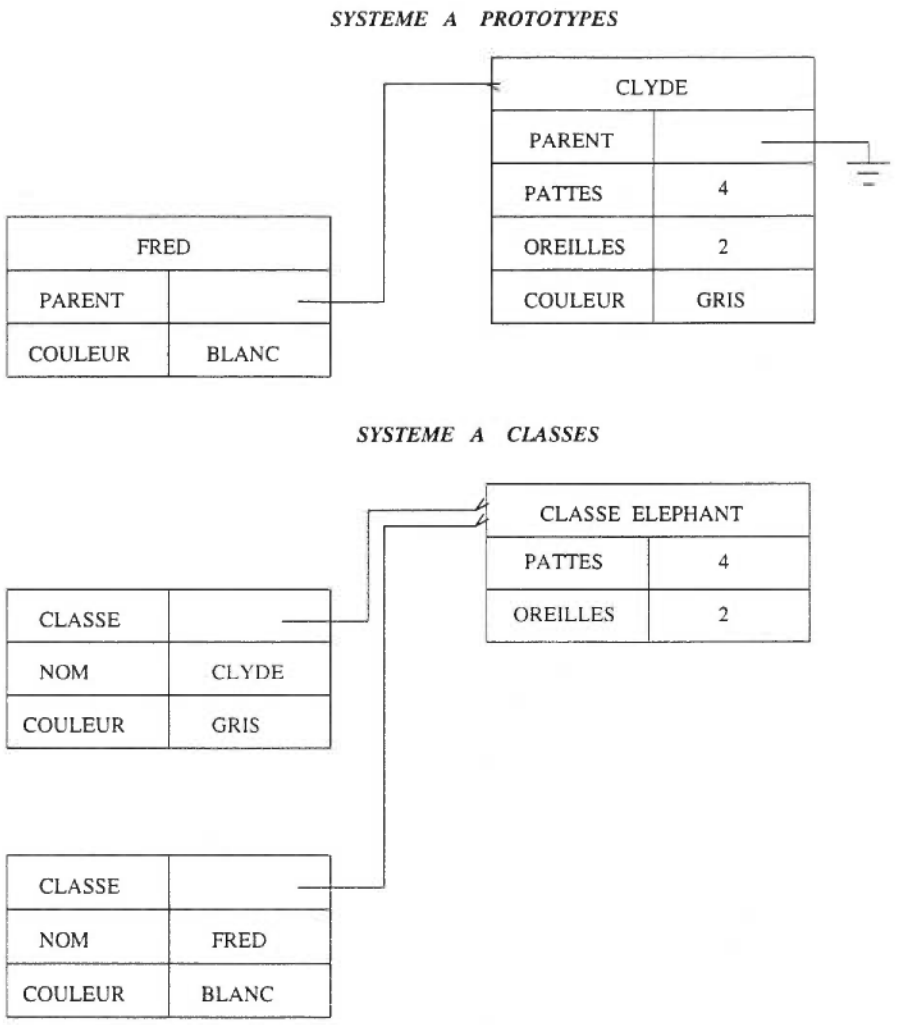

FIG. 6.1 - *Représentation d'un ensemble d'éléphants dans un système à prototypes et à classes*

évidence le manque de représentation abstraite dans le système à prototypes.

La mise à jour de l'information des prototypes parents peut causer des erreurs dans des prototypes attachés par liens de délégation. Si le nombre de pattes de Clyde change (dans le cas qu'il perde une patte, il y en aura seulement trois), Fred sera aussi représenté comme un éléphant handicapé. Ce type d'erreurs peut être évité en donnant aux prototypes l'habilité "d'extraire" la généralisation de l'information au parent pour ensuite la mettre à jour selon son information de cas particulier.

Il nous semble que les mécanismes d'héritage qui vont du général au particulier

dans les systèmes à classes sont possibles d'être modélisés avec les mécanismes de la délégation, qui vont du particulier au particulier.

Donc, il faut trouver des fonctions pour gérer les mécanismes de généralisation et de spécialisation qui sont au cœur de la délégation et de l'héritage. Elles vont nous permettre de trouver des équivalences entre les deux mécanismes, et par extension, passer d'un mécanisme à l'autre.

Nous proposons la définition d'une nouvelle primitive generalize qui puisse généraliser l'information commune à un ensemble des objets. Parallèlement on pourrait penser à son complément, une autre primitive specialize qui puisse rendre particulier à un objet l'information commune à un ensemble d'objets.

D'un certain point de vue generalize rend abstrait une propriété concrète commune à un ensemble d'objets et specialize rend concret un concept abstrait commun à un ensemble d'objets.

Par exemple, dans la figure 4.2, la primitive generalize trouvera que l'objet commun aux trois prototypes a quatre pattes, deux oreilles, une couleur variante et un moyen de locomotion.

Si Fred perd une patte, le système sera averti du changement et la primitive generalize determinera que l'objet commun aux prototypes a maintenant deux oreilles, un nombre de pattes variant, un couleur variant et un moyen de locomotion variant. Il ne propagera pas l'erreur que tous les éléphants ont trois pattes et il demandera au prototypes qui sont directement attachés à Fred d'étendre leur information avec un champ ``pattes'' actualisé.

Mais, est-il possible d'implémenter des fonctions comme celles-ci? Nous pensons que pour généraliser un concept dans le parent du prototype, nous avons besoin de:

- l'accès aux champs du parent et aux objects avec le même parent (*"siblings"*).
- une primitive pour savoir si un champ contient une donnée ou une méthode.

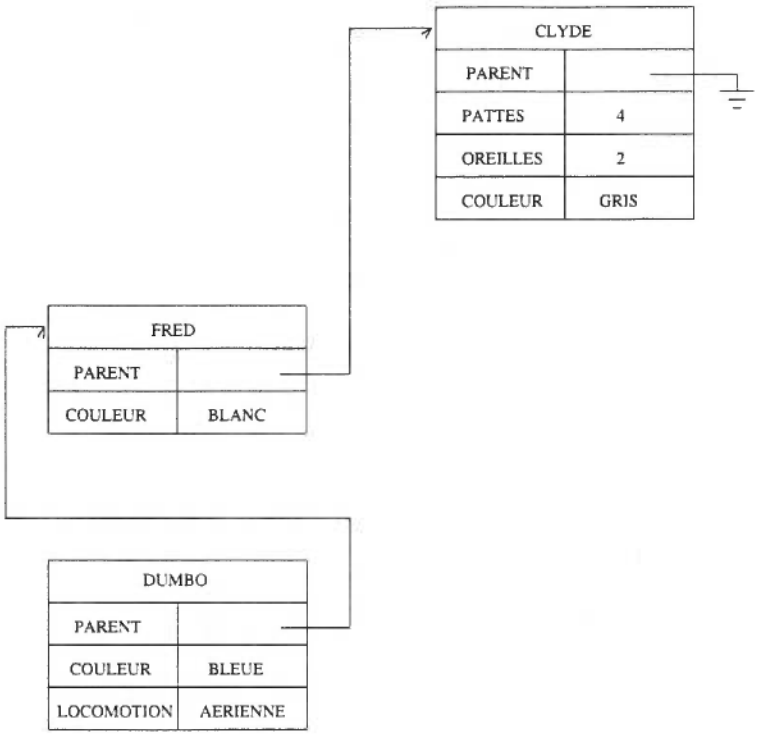

FIG. 6.2 - *Trois éléphants dans un système à prototypes*

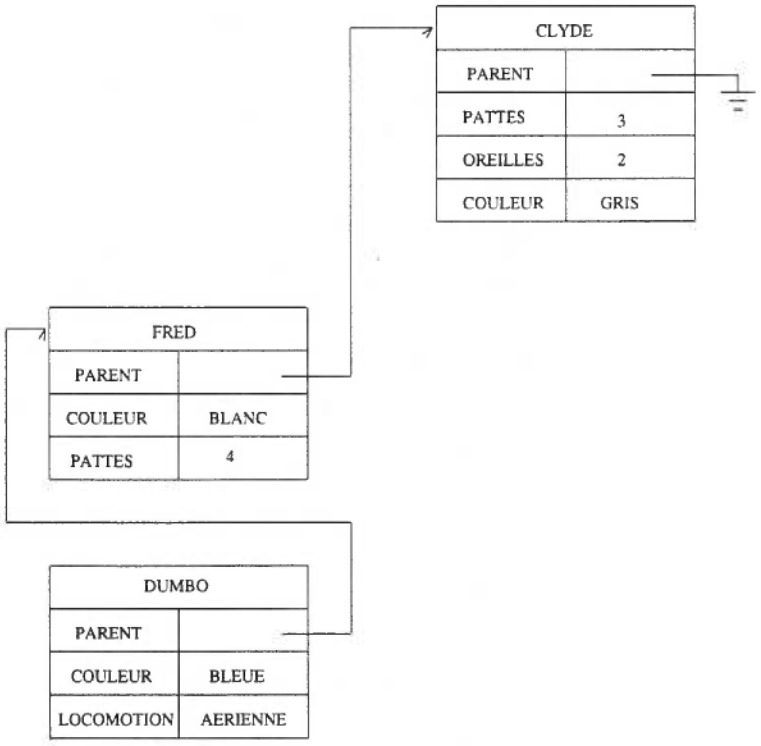

FIG. 6.3 - *Mise à jour de l'information après un changement dans le prototype parent*

- une primitive pour connaître le *nom* du champ par opposition à la *valeur* du champ.
- des primitives de comparaison et la capacité d'un prototype de comparer ses champs avec les champs des autres prototypes.
- une primitive pour créer un prototype à partir d'un sous-ensemble des champs du parent.

Nous pensons qu'avec ces conditions et les capacités déjà existantes dans des langages comme Proto-Reflex on pourrait implémenter des fonctions de généralisation et de spécialisation. Avec ces fonctions nous pourrions définir toute la gamme de relations d'héritage pour le mécanisme de délégation sans faire des modifications à la structure simple de prototypes. La souplesse de la programmation à prototypes est aussi gardée mais enrichie avec des nouvelles relations entre prototypes.

Le problème de mélange d'abstraction et d'information concrète (de "cas particulier") dans les prototypes a été déjà étudié [UCCH91a, UCCH91b, Bor86, Mal95]. Traditionnellement, le problème avait été résolu par l'introduction du concept du *"traits"* (à la Self) et de *"maps"* [US87, Mal95]. La critique générale est qu'elles enlèvent la flexibilité des prototypes, changent le concept de prototype et s'approchent du modèle à classes. Malenfant [Mal95] propose le concept "d'objets morcelés", mais cette solution ajoute beaucoup de complexité à la structure naturelle de prototypes. Nous pensons que la solution des primitives generalize et specialize est plus simple et aussi efficace qui pourrait être utilisée dans d'autres types des systèmes informatiques.

Dans la même ligne de pensée, on pourrait aller plus loin avec cette solution et implanter les primitives generalize et specialize à l'aide de fonctions réflexives. Ceci en considérant qu'un prototype puisse arriver à faire une généralisation d'un concept au parent à l'aide d'un processus d'introspection. Dans ce processus, il compare la structure de son parent (ses champs, sa nature et ses valeurs), avec sa propre structure

et celle des autres instances voisines. Ceci est garanti par la réflexion de structure dans un langage. Ensuite, il peut se poser des questions sur la comparaison des valeurs et appliquer la généralisation et/ou la spécialisation.

Comme nous avons vu antérieurement, ObjVLisp n'est pas un langage réflexif, mais sa définition comme langage montre les caractéristiques de la réflexion de structure. Ceci parce que les classes et les méta-classes, d'une façon homogène, possèdent la représentation de la structure de ses instances.

En regardant de plus proche son architecture, il est à noter qu'elle permettrait d'étendre le langage afin que ce dernier puisse exprimer une réflexion de comportement.

La réflexion de comportement est obtenue en Proto-Reflex via le méta-objet que tout prototype possède. Le lien vers son méta-objet est simplement un champ vers lui mais le prototype tire de cette relation des capacités de réflexion et de réification très importants. En Proto-Reflex, le méta-objet est le responsable des aspects réflexifs comportementaux de l'objet. Cette architecture vient du langage 3-KRS [Mae87] où les aspects réflexifs des objets sont divisés en deux: les structurels sont gardés dans la classe et les comportementaux dans les méta-objets.

Comme le modèle d'ObjVLisp est tellement homogène, il pourrait être étendu en ajoutant l'ensemble des primitives qui gèrent les mécanismes de la réflexion selon le modèle du 3-KRS.

Un étude semblable a été développée par Ferber [Fer89]. Il a introduit la réflexion de comportement dans un langage à classes en utilisant ObjVLisp comme modèle et le concept des méta-objets. Le modèle résultant ne permet pas d'avoir une grande souplesse d'utilisation des capacités réflexives dû à l'utilisation de la classe de l'objet comme méta-objet. Pour lever les limites de cette approche, nous laisserons dans les classes seulement les aspects structurels de la réflexion, ainsi que les aspects comportementaux devront être gérés dans d'autres structures qui jouent le rôle de méta-objets.

Chapitre 7

Conclusion

Dans cette étude nous avons implémenté trois langages à classes à l'aide d'un langage à prototypes. Un des buts du projet, ce de modéliser les principales caractéristiques des langages a été réussi, mais le processus pour y arriver nous a ouvert des nouvelles motivations de recherche.

Nous avons constaté l'importance de la structure des entités d'un langage pour atteindre certaines capacités, comme la méta-programmation et les capacités réflexives. Aussi, nous nous sommes interrogés sur la flexibilité que chaque solution de structure apporte vers la redéfinition de comportements des entités d'un système.

La comparaison des différentes conceptions d'architecture des langages nous a permis d'évaluer le rôle que la distribution des données et de l'information joue pour rendre un système plus souple vers l'adoption des nouveaux comportements, comme l'héritage multiple, ou des modifications à ses mécanismes de base, comme sa façon de traiter les messages.

Dans les trois langages à classes modélisés, nous avons observé certaines lignes de conception utiles pour les classifier et analyser. Elles sont décrites à continuation.

7.1 Structure des objets.

Quant à la structure de ses objets, nous avons observé les conceptions suivantes:

- des objets qui encapsulent de l'information sur leur structure et leur comportement. (ObjVLisp et Java)
- des objets *statiques* qui n'encapsulent que leur information structurelle (ses champs de valeurs), sans avoir des champs de méthodes ou d'autre information de comportement. (CLOS)
- des fonctions et des méthodes associées qui décrivent le comportement des objets *statiques*. (CLOS)

Dans ObjVLisp, les instances sont les seuls éléments *statiques* qui n'encapsulent que des valeurs, mais ses classes et méta-classes gardent les méthodes qui décrivent leur comportement et l'information sur leur structure. En fait, la couche d'éléments composée par les classes et méta-classes joue un rôle dynamique dans le système, et la couche statique, composée par les instances, forme le domaine du système. Etant donné leur information, les classes et méta-classes sont les cibles pour effectuer des modifications dans le comportement des objets du système. Java suit le même modèle, mais avec une couche dynamique un peu plus complexe (enrichie avec d'autres éléments comme les interfaces, les classes abstraites, etc) qui lui donne plus de souplesse.

Par contre, en CLOS les instances, classes et méta-classes forment une hiérarchie purement structurelle associée au graphe des méthodes et des fonctions génériques. Il n'y a pas de méthodes encapsulées dans ces objets et tout le contrôle du comportement du système se trouve dans le graphe des fonctions génériques. Ici, la division des aspects structurels et comportementaux permet de contrôler séparément les processus qu'impliquent la structure des objets et ceux qui touchent la dynamique du

comportement. Par exemple, le mécanisme d'héritage en CLOS est présent dans la partie statique du système comme un champ dans chaque classe, qui garde la liste de précédence de classes (class-precedence-list). Cette liste est activée par une fonction générique associée (compute-class-precedence-list) qui réalise l'héritage multiple à l'aide de ses méthodes. C'est ici qu'il existe une flexibilité pour modifier le régime d'héritage sans avoir des conflits avec la structure des entités. Il faudra seulement modifier les méthodes qui contrôlent les règles de l'héritage, car elles sont été implémentées indépendamment de la structure des objets. La partie structurelle, c'est-à-dire, le champ class-precedence-list ne se modifie pas, (il est seulement consulté), mais sa fonction générique associée, compute-class-precedence-list est modifiée pour gérer d'autres sortes d'héritage.

De la même façon, d'autres comportements peuvent être modifiés assez facilement dans les systèmes qui ont une séparation explicite de ses aspects structurels et comportementaux.

En ce qui concerne les prototypes, sa structure encapsule les aspects structurels et compotementaux dans une seule entité. En fait, pour détecter et contrôler séparément chaque partie il faudra établir des mécanismes assez coûteux en temps, espace mémoire et efficacité, en plus d'éliminer la simple élégance des prototypes. Par ailleurs, la simplicité des prototypes a été utile pour modéliser les aspects séparés de Closette. Grâce à la flexibilité dans leur structure, quelques prototypes ont été implémentés comme des fonctions génériques, d'autres comme des méthodes et d'autres comme des classes, méta-classes et instances.

La force des prototypes est sa simplicité. Ses entités encapsulent une partie de sa structure complète (la structure complète d'un prototype devrait comprendre toute sa ligne de délégation) et une partie de son comportement total. Si on veut séparer ces deux parties pour les exploiter et contrôler efficacement, il serait plus adéquat d'utiliser la flexibilité de la structure des prototypes: définir des prototypes "purement

comportementaux" comme par exemple, Basic-Lookup, Basic-Apply ou Basic-Meta-Object. En fait, nous trouvons un exemple approprié dans l'implémentation de la délégation multiple en Proto-Reflex par Marco Jacques [Jac95]. Il a défini une prototype *mo-multiple-delegation* qui est composé des méthodes pour gérer la délégation multiple. Les méthodes, aussi des prototypes "purement comportementaux" redéfinissent les comportements de base comme le lookup (dans la syntaxe du Proto-Reflex mo-mult::lookup-in et mo-mult::base-lookup).

Dans le modèle d'ObjVLisp et Java, l'héritage est un comportement *hard wired*, c'est-à-dire, établi dans la définition de la *Machine virtuelle*. Pour modifier le mécanisme au niveau de règles de classes et méta-classes il faudra implémenter un élément structurel pour garder la liste de classes et sa précédence et la définition ou modification des méthodes pour gérer le comportement. L'extension de la partie structurelle est facile à implémenter (Java le fait à l'aide de ses *interfaces*, qui gardent une liste des classes), mais la partie comportementale est difficile à modifier car elle n'est pas explicite dans le langage.

7.2 Conception du système.

Quant à la conception du système, nous avons observé:

- une architecture basée sur le concept d'une *Machine virtuelle*. La structure des langages ainsi conçus peut être dupliquée car la description totale du langage est disponible (ObjVLisp et Java).

- l'existence d'une *racine* du système, composée par l'interrelation des objets constituant un noyau de base: objets Objet et Classe. Ils font l'initialisation et le démarrage (bootstrap) du système. (ObjVLisp et Java).

- l'existence d'un objet standard-object, qui est attaché à son tour de l'objet t. Ils ne font aucune action d'initialisation (bootstrap) dans le système. L'objet standard-object est la super-classe par défaut (CLOS).

Les langages ObjVLisp et Java sont basés dans le concept d'une *Machine virtuelle.* Par conséquence, ils sont capables de reproduire la description de leur interprètes pour recréer leur systèmes. Donc, ils peuvent avoir des capacités reflexives à l'aide d'une succession de machines virtuelles, redéfinies une après l'autre. Des comportements *hard-wired* comme le mécanisme d'héritage peuvent être modifiés de cette façon.

L'organisation des objets du système selon des *objets-racine* des graphes d'instantiation et d'héritage sert à plusieurs buts, principalement de consistance et de contrôle. La concentration de méthodes de base uniques dans les *objets-racine* garantit l'homogénéité de comportement dans certaines tâches importantes comme la création des instances, définition de liens et champs des objets, etc. Le contrôle de ces tâches est plus facile à gérer en gardant une seule copie dans un objet commun, accessible et protégé. Dans ce cas, l'homogénéité de structure est un effet de l'homogénéité de comportement. La spécialisation de méthodes sans perte de contrôle est tout à fait possible dans la ligne d'héritage, étant donné la nature hiérarchique de ce type de graphes.

Dans le modèle d'ObjVLisp et Java, la distribution des méthodes de base dans les deux *objets-racine* est fait selon un critère de séparation de méthodes reliés au processus d'instanciation ou bien au processus d'héritage. Ainsi, la méthode de création se trouve dans l'objet Class, mais l'initialisation des instances ou le clonage se trouvent dans l'objet Object (aussi appelé Root). En CLOS, la distribution des méthodes suit le critère de séparation d'aspects structurels et comportementaux, en conséquence, toutes les méthodes se trouvent groupés dans un graphe de fonctions génériques, hors des objets qui forment la structure du système. Voici la raison d'avoir seulement une *racine* composée de deux *objets-racine*: standard-object et t. Ils n'encapsulent pas des

méthodes, mais ses fonctions génériques associées gardent les méthodes de base des comportements les plus importants du système.

7.3 Structures et méta-structures.

Quant à leur capacité d'implémenter des méta-structures, nous avons observé:

- l'utilisation des classes comme des méta-objets pour redéfinir le comportement (ObjVLisp et Java).
- la définition de méta-objets attachés à chaque élément du système pour redéfinir le comportement et créer des extensions au langage.

L'utilisation des classes comme des méta-objets pour modifier le comportement des entités pose de restrictions à la redéfinition de mécanismes. D'abord, les modifications individualisées sont difficiles à réaliser car elles sont appliquées par groupes, c'est-à-dire, à tous les membres d'une même classe. Ensuite, certains mécanismes de comportement du système ne sont pas explicitement définis dans les classes et par conséquent, impossibles d'accéder et modifier. Finalement, les classes dans ObjVLisp et Java encapsulent de l'information sur leur structure en plus de l'information sur son comportement, en compliquant ainsi la localisation, le traitement et la modification de l'aspect comportemental.

CLOS, par contre, introduit les méta-protocoles pour définir des extensions au langage et modifier ainsi toutes sortes de comportements. Tous les éléments du système à l'exception des instances (domaine du système) ont un méta-objet attaché qui garde son information. Ainsi, le méta-objet attaché aux classes a de l'information sur la structure de la classe, la liste de précédence de classes, ses champs et ses méthodes associées. Les méta-objets attachés aux objets qui déterminent le comportement du système (comme les méthodes ou les fonctions génériques) gardent de l'information

comme le nom de la fonction (ou méthode), ses *spécialisateurs* (classes associées en CLOS), liens vers les méthodes (ou fonctions génériques), environnement d'évaluation, liste d'arguments (*lambda list* en CLOS), code de la méthode, etc. Avec toute cette information, le programmeur a toute la liberté et la flexibilité nécessaires pour modifier profondément le comportement du langage. La réflexion de structure et de comportement sont assez simples à définir dans ce modèle, car elles peuvent être implémentées à l'aide des méta-objets. En fait, étant donné le pouvoir de modification, le risque d'aboutir à une inconsistance du système est élevé. Les changements, donc, doivent être faits avec soin et planification.

7.4 Communication.

Quant aux communications entre entités, nous avons observé les conceptions suivantes:

- la classe est le receveur du message et elle s'occupe de son traitement (ObjVLisp et Java).
- la communication est l'application d'une fonction et le méta-objet associé est le receveur du message et il garde l'information de comment traiter le message (CLOS).

CLOS utilise l'application des fonctions génériques pour établir de la communication entre entités et transformer le système d'un état à l'autre. Le passage de messages n'est pas défini dans le modèle original. Pour faire l'application d'une fonction, le système utilise le protocole suivant: lorsque la fonction générique est invoquée, une fonction de discrimination détérmine selon les arguments la ou les méthode(s) à appliquer. Chaque partie du protocole est décomposée dans des sous-fonctions, toutes redéfinissables au besoin de l'usager.

Bien que ce mécanisme de communication n'est pas défini, l'application d'une méthode sous la forme d'un passage de messages pourrait être éventuellement implémentée à l'aide des méta-objets. Les protocols de méta-objets de CLOS sont suffisamment flexibles pour cela.

ObjVLisp et Java présentent un modèle plus restrictif pour modifier le passage de messages. Java permet des variations dans les communications des objets, comme les messages concurrents, mais ce sont des facilités déjà établies dans le langage. Le passage des messages dans ces langages est un mécanisme implanté dans la *Machine virtuelle* et sa définition n'est pas explicite pour sa modification. Néanmoins, les mécanismes d'éxtension incrémentale des langages à objets sont assez puissants comme pour monter au niveau des classes et métaclasses un mécanisme modifié de passage de messages en redéfinissant les méthodes accessibles qui participent dans le mécanisme standard. Naturellement, certaines parties du mécanisme restent inaccessibles au changement. Une autre possiblité de modifier le passage de messages serait à l'aide de la réflexion, en profitant qu'ObjVLisp et Java peuvent recréer ses systèmes et implémenter la réflexion dans une succession de *Machines virtuelles* (*tour réflexive*). Même si elle est possible, la modification du mécanisme de passage des messages dans ce genre des systèmes est dangereuse. Le passage des messages est la seule façon d'établir une communication entre les objets et sa modification a le risque de mener le système à l'inconsistance.

Le mécanisme de passage de messages est un élément approprié pour implémenter la réflexion [Fer89]. ObjVLisp et Java ont une primitive standard send pour être réifié, mais CLOS n'a pas accès à cette façon d'implémentation de la réflexion car il n'utilise pas le passage de messages. Comme nous avons déjà montré, CLOS utilise l'appel des fonctions génériques à la place de la communication à l'aide du passage des messages.

Pour en terminer, nous constatons que la modélisation de langages à classes à l'aide de Proto-Reflex nous a permis d'observer et d'étudier beaucoup plus d'aspects

et caractéristiques du systèmes que nous avions prévus au début du travail. Le projet nous a conduit à la comparaison de plusieurs structures, architectures et mécanismes dans trois langages à classes avec deux modèles différents et un langage à prototypes. Les résultats obtenus pourraient être récupérés, approfondis et postérieurement utilisés pour la conception des nouveaux systèmes ou bien pour le dévéloppement des nouveaux comportements des systèmes dans le fertile monde des langages à objets.

Bibliographie

[AG96] Ken Arnold et James Gosling. *The Java Programming Language.* The Java Series. Addison Wesley, 1996.

[Bor86] A.H. Borning. Classes versus prototypes in object-oriented languages. Dans *Proceedings of IEEE/ACM Joint Conference*, pages 36–40, 1986.

[Bud97] T. Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, 1997.

[Coi87] Pierre Cointe. Metaclasses are first class: The ObjVLisp model. Dans *Proceedings of OOPSLA'87*, pages 156–167. ACM Sigplan Notices, December 1987.

[Fer89] J. Ferber. Computational reflections in class-based object-oriented languages. Dans *Proceedings of OOPSLA'89*, pages 317–326. ACM Sigplan Notices, October 1989.

[Jac95] Marco Jacques. Implantation d'un langage à prototypes avec réflexion de comportement. Mémoire de maîtrise, Département d'informatique et de recherche opérationnelle. Université de Montréal, février 1995.

[KdRB91] G. Kiczales, J. des Rivières et Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, Massachusetts, 1991.

[LaL89] W.R. LaLonde. Designing families of datatypes using examplars. Dans *ACM TOPLAS*, pages 212–248. ACM Sigplan, avril 1989.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. Dans *Proceedings of OOPSLA'86*, pages 214–223. ACM Sigplan Notices, November 1986.

[Mae87] Pattie Maes. Concepts and experiments in computational reflexion. Dans *Proceedings of OOPSLA'87*, pages 147–155. ACM Sigplan–Notices, december 1987.

[Mal95] Jacques Malenfant. Programmation par prototypes. Rapport technique, Département d'informatique et de recherche opérationnelle, Université de Montréal, avril 1995. publication 970.

[MDC92] J. Malenfant, Christophe Dony et Pierre Cointe. Behavioral reflection in a prototype-based language. Dans *Proceedings of the International Workshop on New Models for Software Architecture '92.*, Reflection and Meta-Level Architecture, Rise(Japan), November 1992. ACM Sigplan,JSSST, IPSJ.

[MGZ92] B.A. Myers, D.A. Giuse et B. Vander Zander. Declarative programming in a prototype-instance system: Object oriented programming without writing methods. Dans *Proceedings of OOPSLA'92*, pages 184–200. ACM Sigplan Notices, octobre 1992.

[MNC+90] G. Masini, A. Napoli, D. Colnet, D. Léonard et K. Tombre. *Les langages à Objets.* InterEditions, 1990.

[Smi82] Brian C. Smith. *Reflection and Semantics in Procedural Langages.* Thèse de doctorat, Massachussets Institute of Technologie, 1982.

[Ste87] Lynn A. Stein. Delegation is inheritance. Dans *Proceedings of OOPSLA '87*, volume 22, pages 138–146. ACM Sigplan Notices, December 1987.

[Sun95] Sun Microsystems Computer Corporation. *The Java Language Specification*, Octobre 1995.

[UCCH91a] D. Ungar, C. Chambers, B.-W. Chang et U. Hölzle. Organizing programs without classes. Dans *Lisp and Symbolic Computation*, pages 223–242. Kluwer Academic Publishers, 1991.

[UCCH91b] D. Ungar, C. Chambers, B.-W. Chang et U. Hölzle. Parents are shared parts of objects. Dans *Lisp and Symbolic Computation*, pages 207–222. Kluwer Academic Publishers, 1991.

[US87] D. Ungar et R. Smith. Self:the power of simplicity. Dans *Proceedings of OOPSLA '87*, pages 227–242. ACM Sigplan Notices, Decembre 1987.