Les langages à prototypes[1]

Christophe Dony (1), Jacques Malenfant (2), Daniel Bardou (1)

(1) *L.I.R.M.M – 161, rue Ada*
34392 Montpellier Cedex 5 – FRANCE

(2) *Ecole des Mines de Nantes – 4, rue A. Kastler*
44070 Nantes Cedex 03 – FRANCE

Résumé

Les langages à prototypes proposent une forme de programmation par objets s'appuyant sur la représentation d'objets concrets plutôt que sur celle de concepts, ou plus pragmatiquement, une forme de programmation par objets sans classes. Ce chapitre décrit ces langages, les raisons qui ont conduit à leur émergence, les possibilités nouvelles qu'ils apportent, mais aussi les problèmes qu'ils posent. Nous répondons en premier lieu aux questions suivantes. Qu'est-ce qu'un prototype? Qu'est-ce qu'un objet concret? Pourquoi chercher à se passer des classes? Quelle est l'architecture primitive d'un langage à prototypes et quelle est la genèse de ces langages? Nous caractérisons plus précisément les deux mécanismes de création d'objets : le clonage, et la description différentielle. Cette caractérisation nous permet de différencier l'héritage dans les hiérarchies d'objets (la délégation) de l'héritage via des hiérarchies de classes (qui est l'héritage classique des langages à objets).

Elle nous permet également de présenter les spécificités ainsi que les problèmes que pose la programmation par prototypes. Au travers de cette présentation critique, nous posons finalement la question de l'intérêt et de l'avenir de cette forme de programmation par objets.

1 Introduction

Les langages à objets les plus utilisés aujourd'hui (en particulier dans le monde industriel) sont issus de SIMULA (1967) et de SMALLTALK (1972). Ce sont des langages au sein desquels la classe, modélisation intensionnelle d'un concept, est l'unité fondamentale autour de laquelle s'organise la représentation de connaissances et se structurent les programmes.

Parmi les autres familles de langages de programmation par objets effectivement utilisées ou largement étudiées dans des travaux de recherche figure la famille des langages dits à prototypes dont nous traitons dans ce chapitre. Un prototype est un *représentant typique* d'une famille ou d'une catégorie d'objets [Cohen et Murphy, 1984]. Les langages à prototypes, apparus au milieu des années 80, proposent une approche de la programmation par objets reposant sur la notion de prototype plutôt que sur celle de classe. Ils ont été inspirés par les premiers langages de *frames* utilisés en représentation de connaissances **cref**{Euzenat}) et par certains langages d'acteurs, issus des travaux de C. Hewitt, utilisés en programmation distribuée (**cref**{Briot}) . Ils ont été étudiés et développés, en réaction à un certain nombre de limitations propres au modèle à classes, avec les objectifs suivants:

– permettre une description simple des objets ne nécessitant pas la description préalable de modèles abstrait,

1. Paru dans le livre *"Langages et Modèles à Objets : Etats des recherches et perspectives"*, editeurs : R.Ducournau, J.Euzenat, G.Masisi et A.Napoli. INRIA - Collection Didactique en 1998.

– offrir un modèle de programmation plus simple que celui des classes, qui jouent de trop nombreux rôles,

– enfin, offrir de nouvelles possibilités de représentation de connaissances.

Les langages à prototypes sont issus de ces objectifs ainsi que du constat d'un certain nombre de limitations du modèle à classes [Borning, 1986], avec l'espoir d'obtenir une plus grande puissance d'expression dans des langages par ailleurs plus simples. Depuis le milieu des années 1980, de nombreux langages ont vu le jour : SELF [Ungar et Smith, 1987; Agesen *et al.*, 1993; Agesen *et al.*, 1995; Chambers *et al.*, 1991; Smith et Ungar, 1995], KEVO [Taivalsaari, 1991; Taivalsaari, 1993], NAS [Codani, 1988], EXEMPLARS [LaLonde *et al.*, 1986; Lalonde, 1989], AGORA [Steyaert, 1994], GARNET [Myers *et al.*, 1990; Myers *et al.*, 1992], MOOSTRAP [Mulet et Cointe, 1993; Mulet, 1995], CECIL [Chambers, 1993], OMEGA [Blaschek, 1994], NEWTON-SCRIPT [Smith, 1994] . D'autres langages, tels OBJECT-LISP [Allegro Common Lisp, 1989] ou YAFOOL [Ducournau, 1991] ne se réclamant pas de l'approche par prototypes offrent néanmoins des mécanismes proches.

La caractérisation très générale et informelle des langages à prototypes est relativement aisée : ce sont des langages dans lesquels on trouve en principe une seule sorte d'objets dotés d'attributs[2] et de méthodes, trois primitives de création d'objets : création *ex nihilo*, *clonage* et *extension* (ou *description différentielle*), un mécanisme de calcul, l'envoi de message, intégrant un mécanisme de *délégation*. Ceci étant posé, leur caractérisation, leur utilisation et leur compréhension précise posent en fait un certain nombre de problèmes.

- Il existe diverses interprétations de ce qu'est un prototype, objet concret ou représentant moyen d'un concept, qui peuvent conduire à des langages assez différents [Malenfant, 1995].
- La sémantique des mécanismes de base (clonage, copie différentielle, délégation) n'est pas unifiée et autorise différentes interprétations [Dony *et al.*, 1992; Malenfant, 1995; Bardou et Dony, 1996; Bardou *et al.*, 1996; Malenfant, 1996].
- La description différentielle rend les objets interdépendants, ce qui pose de nouveaux problèmes et autorise diverses interprétations quant au statut des objets [Dony *et al.*, 1992; Malenfant, 1996; Bardou et Dony, 1996; Bardou *et al.*, 1996].
- En même temps que les classes, a été supprimée par exemple, la possibilité d'exprimer que deux concepts partagent certaines caractéristiques. Cette seconde possibilité est si importante en terme d'organisation des programmes que de nombreux langages à prototypes ont cherché à la réintroduire, ce qui a été fait de façon plus ou moins appropriée. Cette réinsertion de formes d'abstraction dans le modèle a remis en cause certains postulats initiaux et a brouillé les frontières entre langages à prototypes et langages à classes [Malenfant, 1996].

Nous nous proposons de décrire ces langages, de juger des possibilités qu'ils offrent, d'étudier dans quelle mesure ils satisfont les objectifs que leurs concepteurs s'étaient fixés et à quel prix. Nous nous demandons si ces langages sont viables (peut-on se passer de la représentation des concepts), dans l'affirmative lesquels utiliser et, dans la négative, si certaines des idées qu'ils ont introduites peuvent être appliquées dans d'autres contextes? Le paragraphe 2 rappelle ce qu'est la notion de prototype en science cognitive. Le paragraphe 3 présente les premières utilisations de cette notion en représentation de connaissances par objets ainsi qu'en programmation distribuée. Nous y décrivons les primitives de clonage et de description différentielle. Le paragraphe 4 expose les motivations qui ont conduit les chercheurs à concevoir des langages à objets sans classes ; il montre l'intérêt potentiel de la programmation par prototypes. Le paragraphe 5 décrit les premières propositions de langages sans classes. Le paragraphe 6 fait le point sur les concepts et les mécanismes de base de la programmation par prototypes. Le paragraphe 7 propose une caractérisation plus fine de ces concepts et de ces mécanismes. Cette caractérisation permet de différencier les hiérarchies d'objets des hiérarchies de classes et de mieux comprendre les différentes évolutions des langages à prototypes. Le paragraphe 8 décrit les problèmes liés à l'identité des objets et le paragraphe 9 les problèmes liés à l'organisation des programmes. En conclusion nous présentons un bilan de l'expérience ainsi les axes de recherche que notre analyse fait apparaître.

2. Nous utilisons ce terme pour désigner une caractéristique non comportementale d'un objet ou d'un *frame*, nous aurions pu utiliser les équivalents que sont « champ » ou « slot ».

2 Notion de prototype

On trouve en sciences cognitives l'idée de représenter un concept, ou une famille d'entités, par un représentant distingué ainsi que l'idée de copie différentielle. Dans ce contexte, différents modèles de la notion de concept ont été proposés [Smith et Medin, 1981; Cohen et Murphy, 1984; Kleiber, 1991].

Un de ces modèles est fondé sur la théorie des ensembles : à chaque concept correspond une collection d'entités (extension), et chaque concept admet une définition qui caractérise son « essence » et définit les conditions nécessaires et suffisantes à l'appartenance d'une instance à ce concept (intension), . La relation qui lie une instance à un concept et celle qui lie un concept plus spécifique à un concept plus général s'y apparentent respectivement aux relations ensemblistes d'appartenance et d'inclusion. Ce modèle de concepts conduit à une mise en œuvre basée sur les classes.

Un autre modèle (développé en linguistique) permet de ne pas valuer systématiquement toutes les caractéristiques d'une instance. Il y a toujours des conditions nécessaires et suffisantes pour l'appartenance à un concept, mais on s'accorde la possibilité de ne pas savoir : on sait qu'une instance appartient à un concept, qu'elle n'y appartient pas, ou bien on n'en sait rien (**cref**{euzenat}). La « théorie des prototypes » est une extension de cette approche dans laquelle la relation d'appartenance est une certaine relation de ressemblance plus ambiguë. Dans cette théorie, les concepts ne sont décrits ni en intension ni en extension mais indirectement au travers de prototypes du concept, c'est-à-dire d'exemples. Cette théorie découle du principe selon lequel l'humain se représente mentalement un concept, identifie une famille d'objets et mène des raisonnements sur ses membres en faisant référence, au moins dans un premier temps, à un objet précis, typique de la famille. Ma « 2CV » est, par exemple, un prototype du concept de « voiture », comme « netscape » l'est pour le concept de « navigateur internet ». On trouve aussi dans la théorie des prototypes la notion de description différentielle qui désigne la possibilité de décrire 'un nouveau représentant du concept via l'expression de ses différences par rapport à un représentant existant.

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

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

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

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

3.1 Les prototypes en représentation des connaissances

Les langages à prototypes existaient avant que l'appellation n'apparaisse. On trouve ainsi les notions de prototype et de copie différentielle dans la théorie des *frames* de Minsky [Minsky, 1975] et dans certains systèmes inspirés de cette théorie comme les langages de *frames* tels KRL [Bobrow

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

FIG. 1 – *Exemple de* Frame

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

FIG. 2 – *Description différentielle*

et Winograd, 1977] ou FRL [Roberts et Goldstein, 1977] .

> « *Les* frames *sont un formalisme de représentation créé pour prendre en compte des connaissances qui se décrivent mal ... [dans d'autres formalismes] ... comme la typicalité, les valeurs par défauts, les exceptions, les informations incomplètes ou redondantes. La structure d'un* frame *... doit pouvoir évoluer à tout moment, par modification, adjonction ou modification de propriétés.* » [Masini *et al.*, 1989]

Nous allons donner ici une vision simplifiée à l'extrême de ce que sont les *frames*, sans illustrer leur richesse et leur diversité; notre but est de montrer en quoi ils utilisent la théorie des prototypes et comment ils ont influencés certains des langages à prototypes utilisés aujourd'hui. Le lecteur se reportera aux articles précédemment cités et au chapitre proposé dans [Masini *et al.*, 1989] pour plus de précisions.

• **Structure d'un frame.** Un *frame* est un ensemble d'attributs; chaque attribut permet de représenter une des caractéristiques du *frame* et se présente sous la forme d'un couple « nom d'attribut – ensemble de facettes ». La facette la plus courante étant la valeur de l'attribut, nous ne considérerons que celle-ci dans nos exemples. La figure 1 propose un exemple de définition d'un *frame* dotée de 4 attributs représentant de façon minimale une baleine.

• **Description différentielle.** La description (ou création) différentielle permet de décrire un nouveau *frame* en exprimant ses différences par rapport à un *frame* existant [3]. Elle met en relation le nouveau *frame* avec celui sur lequel sa description différentielle s'appuie et qui est appelé son prototype ou son *parent*. Cette relation est matérialisée par un lien généralement appelé *est-un*. Nous avons représenté ce lien dans nos exemples par l'intermédiaire d'un attribut supplémentaire [4] également nommé *est-un*. La figure 2 montre la définition d'un *frame* représentant `Moby-Dick` qui est comme la baleine précédente à ceci près qu'elle est blanche et que son ennemi est défini plus précisément.

• **Héritage et Hiérarchies de *frames*.** La relation *est-un* est une relation d'ordre définissant des hiérarchie de *frames* [Brachman, 1983]. Un *frame* hérite de son parent un ensemble d'attributs et on trouve dans les systèmes de *frames* des hiérarchies d'héritage, comme celle de la figure 3, très similaires aux hiérarchies de classes [5], à ceci près que les nœuds de cette hiérarchie représentent des exemples plutôt que des description de concepts. Au sommet de la hiérarchie se trouvent généralement des représentants moyens de concepts (par exemple `Animal`) et dans le bas de la hiérarchie des représentants concrets (par exemple `Moby-Dick`). On trouve des hiérarchies similaires dans les programmes réalisés avec les langages à prototypes.

3. « *The object being used as a basis for comparison (which we call the prototype) provides a perspective from which to view the object being described. (...) It is quite possible (and we believe natural) for an object to be represented in a knowledge system only through a set of such comparisons.* » [Bobrow et Winograd, 1977]

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

5. Notons que la définition d'une sous-classe est également une description différentielle.

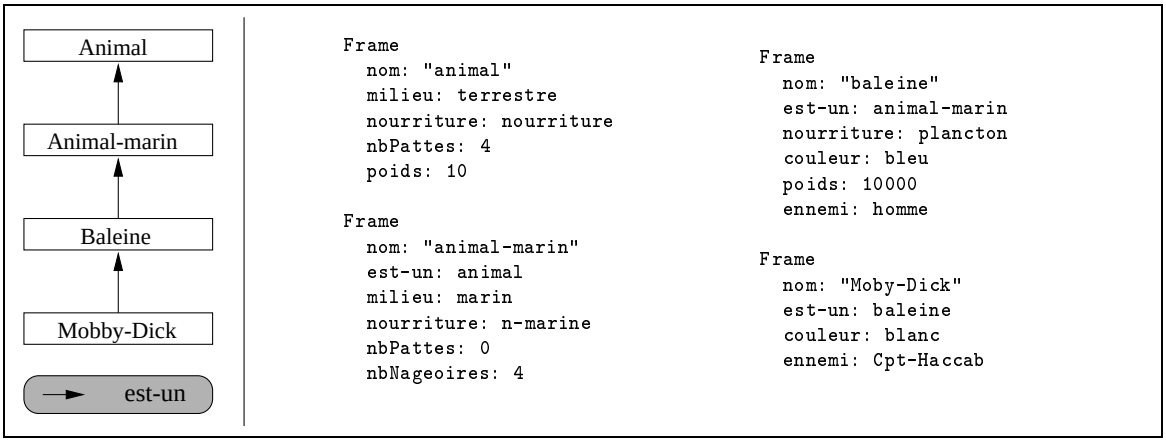


FIG. 3 – *Exemple de hiérarchie de* frames.

3.2 Langages d'acteurs

L'idée de représenter des entités du monde par des objets sans classes a également été appliquée dans le langage ACT 1 [Lieberman, 1981], bien qu'il ne soit fait mention, dans les articles relatifs à ce langage, ni de la notion de prototype ni de l'utilisation qui a pu en être faite dans les langages de *frames* qui lui sont antérieurs[6], tel que KRL. On trouve cependant dans ACT 1 des idées et des mécanismes assez similaires à ceux évoqués précédemment ainsi qu'une part des caractéristiques essentielles des langages à prototypes actuels.

• **Structure d'un acteur.** Les objets dans ACT 1 sont appelés « acteurs », ils possèdent des attributs (appelés *accointances*) référencés par un nom et possédant une valeur. Act1 est un langage de programmation, les acteurs sont donc également dotés de comportements (nous utilisons les termes classiques de « méthode » pour désigner un comportement et celui de « propriété » pour désigner indiferemment un attribut ou une méthode). Les méthodes peuvent être invoquées en envoyant des messages aux acteurs[7].

Un acteur, objet sans classe ne pouvant être créé par instanciation, est créé par copie ou extension d'un acteur existant. La figure 4 montre un acteur[8] appelé `point` possédant deux attributs `x` et `y` et une méthode `norm` calculant la distance de ce point a l'origine. Trois primitives `create`, `extend` et `c-extend` permettent de créer de nouveaux acteurs [Briot, 1984]. Ce sont ces trois primitives et la mise en œuvre de la copie différentielle qui nous intéressent ici. Nous en discutons au travers des exemples proposés dans [Briot, 1984].

• **Clonage.**

Bien que des primitives de copie d'objets aient existé dans les langages à classes (par exemple en SMALLTALK) antérieurement à ACT 1, ce langage a introduit la copie superficielle[9] (primitive `create`), ou *clonage*, comme moyen primitif de création d'objets. La primitive `create` permet ainsi de créer `point2` par copie de `point` (Fig. 4), de spécifier de nouvelles valeurs de propriétés, par exemple `x` et `y`, et d'en définir de nouvelles, par exemple la méthode `move`.

• **Extensions.** La création par description différentielle en ACT 1 est conceptuellement similaire à celle des *frames*. Elle s'effectue en envoyant à un acteur existant le message `extend`, qui crée un nouvel acteur, que nous appellerons donc « extension » du premier, lui même appelé en ACT 1 le *mandataire*

6. Ceci peut être dû en partie au fait qu'il n'est pas évident d'isoler clairement l'utilisation faite des prototypes dans KRL. D'autre part, l'objectif de ACT 1, mettre en œuvre un outil de programmation parallèle à base d'objets, est notablement éloigné de celui de KRL.

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

8. Créé par extension d'un acteur pré-défini.

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

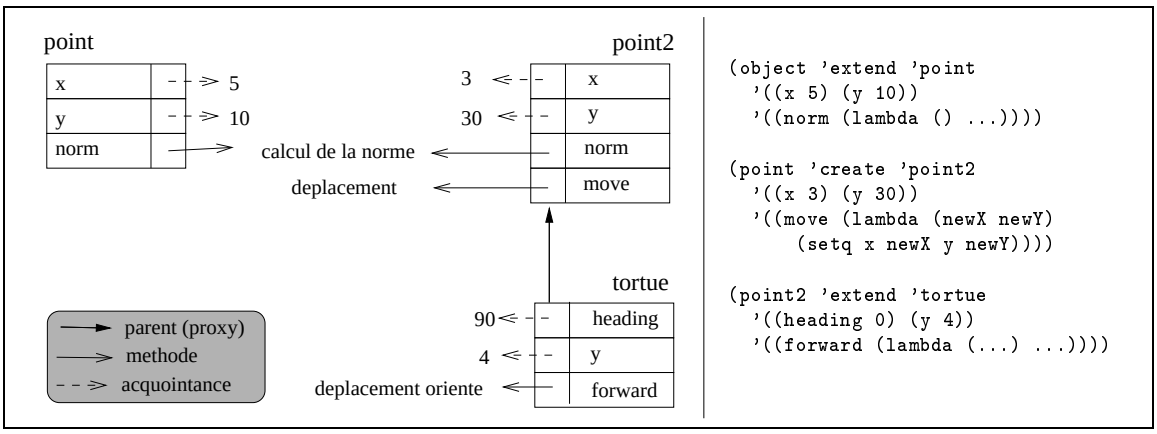


FIG. 4 – *Clonage et extension en* ACT 1.

(*proxy* en anglais) du nouvel acteur ; c'est l'équivalent du prototype ou du parent des *frames*. La figure 4 montre la définition de l'acteur nommé `tortue` représentant une tortue *Logo*[10] comme une extension de l'acteur précédent `point2` qui devient son *mandataire*. Une tortue est comme un point mais possède en plus un `cap` et une méthode `forward` lui permettant d'avancer dans la direction définie par son `cap`.

• **Héritage et première forme de délégation**. Le lien reliant une extension à son mandataire est tout à fait similaire au lien *est-un* des *frames*. L'extension peut hériter des propriétés de son parent. L'héritage est mis en œuvre lorsqu'un acteur ne sait pas répondre à un message parce qu'il ne possède pas la propriété demandée, auquel cas le système demande à son mandataire de répondre à sa place. Le mandataire est ainsi habilité à répondre à un message en lieu et place de ses extensions. Ce passage du contrôle au mandataire est appelé « délégation »[11]. Les articles décrivant ACT 1 laissent dans l'ombre un point important de la problématique des langages à prototypes (cf. paragraphe 5.1) en ne précisant pas le contexte d'exécution une méthode après qu'il y ait eu une délégation. On trouve quoi qu'il en soit dans ACT 1 une première forme de ce qui deviendra la délégation dans les langages à prototype.

• **Copie-extension.** Une des caractéristiques des hiérarchies d'objet est la dépendance qu'établit le lien de délégation entre un parent et une de ses extensions :

> « *Les accointances (les attributs) et le* script *(l'ensemble des méthodes) de mon mandataire (mon parent) sont aussi les miens.* » [Briot, 1984]

Les propriétés du parent sont partagées par ses extensions. Pour permettre la création différentielle d'un nouvel acteur indépendant, une troisième primitive de ACT 1, nommée `c-extend`, compose un clonage[12] et une extension du clone.

4 Motivations et intérêts de la programmation par prototypes

On trouve dans les systèmes que nous venons de décrire l'essence de ce qui a été appelé programmation par prototypes. Les études relatives à l'introduction d'objets sans classes dans les langages de programmation par objets ont été réalisées au milieu des années 80. Elles visaient à proposer des

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

11. « *Whenever an actor receives a message he cannot answer immediately on the basis of his own local knowledge (...), he delegates the message to another actor, called his proxy* ». [Lieberman, 1981]

12. Notons que ce clonage peut être complexe : si l'on souhaite par exemple créer une tortue colorée (traçant des traits colorés) comme une « c-extension » de l'acteur tortue, il faut d'abord cloner `point2` puis cloner `tortue`, lier `clone-tortue` à `clone-point2` par un lien *est-un* et enfin créer le nouvel objet par extension de `clone-tortue`. Cet exemple introduit le problème du clonage d'un objet placé dans une hiérarchie d'objets (cf. paragraphe 8.3).

alternatives au style usuel de programmation par classes utilisé avec SIMULA, SMALLTALK, C++ ou les FLAVORS. Ces études portaient en premier lieu sur la complexité du monde des classes [Borning, 1986] et les limitations que celles-ci imposent en terme de représentation.

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

4.1 Description simplifiée des objets

Le processus de raisonnement humain fait souvent passer l'exemple avant l'abstraction [Lieberman, 1986], l'accumulation d'exemples menant à terme à une généralisation. Or le modèle à classes oblige le programmeur à formaliser un concept, une abstraction, avant de permettre la manipulation de représentants (d'exemples, d'instances) de ce concept. Les langages à prototypes proposent un modèle de programmation plus proche de la démarche cognitive, s'appuyant sur les exemples, attribuée à l'humain face à un problème complexe. Un langage à prototypes permet la description et la manipulation d'objets sans l'obligation préalable d'avoir à décrire leur modèle abstrait.

4.2 Modèle de programmation plus simple

Le modèle à classes est complexe [Borning, 1986; LaLonde *et al.*, 1986; Lalonde, 1989; Stein *et al.*, 1989], parce que les classes y jouent différents rôles qu'il est parfois difficile de dissocier et qui peuvent rendre leur conception, leur mise en œuvre et leur maintenance difficile. Notons parmi ces rôles : descripteur de la structure des instances, bibliothèque de comportements pour les instances, support de l'encapsulation et support à l'implantation de types abstraits, à l'organisation des programmes, à la modularité, au partage entre descriptions de concepts, à la réutilisation. De plus, le même lien entre classes est utilisé pour modéliser différentes relations, entre concepts ou entre types abstraits (suivant la vision que l'on a d'une classe à un instant donné), subtilement différentes les unes des autres [Lalonde, 1989] : héritage de spécifications, héritage d'implantations, sous-typage ou ressemblance [13]. Enfin, dans un système intégrant des méta-classes (comme SMALLTALK ou CLOS), la classe se voit de plus dotée du rôle d'instance, pouvant recevoir des messages et mener, si l'on peut dire, sa propre vie. La « sur-utilisation » du même support (la classe) tend à rendre complexe la programmation par objets, et plus encore la réutilisation de programmes existants.

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

4.3 Expressivité

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

- **Instances différenciées ou exceptionnelles.**

13. Par exemple en SMALLTALK, la classe des ensembles (*Set*) est une sous-classe de la classe des collections non ordonnées quelconques (`Bag`) : « *A set is like a bag except that duplicates are not allowed* » [Lalonde, 1989].

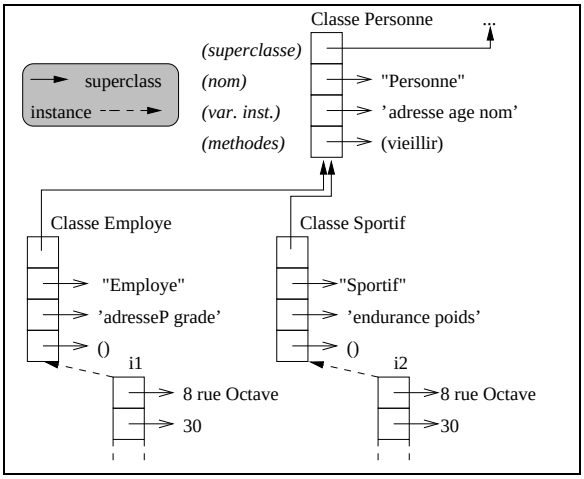


FIG. 5 – *Duplication de la valeur de la propriété* `âge`.

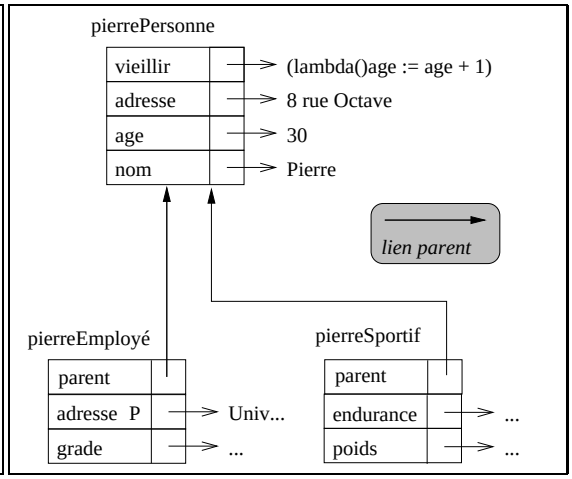


FIG. 6 – *Partage de propriété.*

Considérons en premier lieu le cas des instances exceptionnelles [D.Etherington, 1983; Borgida, 1986; Dony, 1989] ayant des caractéristiques propres que les autres objets de la même famille n'ont pas. La manière standard de représenter une instance exceptionnelle, par exemple `Jumbo` l'éléphant qui sait voler, ou la liste vide dont les méthodes *car* et *cdr* s'implantent différemment de celles des listes non vides, est de créer une nouvelle classe pour les représenter. On trouve, en SMALLTALK par exemple, des classes n'ayant qu'une seule instance : `True`, `False` ou `UndefinedObject`. Même si des alternatives existent pour définir des propriétés au niveau d'un objet, elles sont restées marginales [14].

La représentation d'objets exceptionnels pose évidemment moins de problèmes dans un langage basé sur la description d'individus. On peut par exemple y définir l'éléphant sachant voler par clonage d'un autre éléphant et ajout de la propriété. La représentation de booléens évoquée ci-dessus est tout à fait naturelle dans un langage à prototypes. La possibilité de ne pas créer une classe pour chaque instance exceptionnelle est particulièrement intéressante lorsque celles-ci sont nombreuses (on peut penser par exemple à la modélisation des règles de la grammaire de la langue française [Habert et Fleury, 1993]).

• **Représentation de points de vues d'une même entité.**

Le modèle à classes ne permet pas à des instances de partager des attributs ou des valeurs d'attributs. Posons le problème de la représentation d'une personne et de divers points de vues sur cette personne, par exemple le point de vue « employé » ou le point de vue « sportif ». Il n'existe pas de solution standard dans un langage à classes pour résoudre ce problème. Le schéma classique de spécialisation de la classe `Personne` définissant un attribut `âge`, par deux sous-classes `Employé` et `Sportif` (Fig. 5), ne répond pas à la question car deux instances respectives de ces deux sous-classes représentent des entités différentes indépendantes en terme d'état ; si l'âge de l'une change, l'autre n'est pas affecté. D'autres solutions standard à ce problème (utilisant des constructions présentes dans la majorité des langages à classes) sont insatisfaisantes ; décrivons-en quelques-unes.

– On peut imaginer utiliser la composition : redéfinir les classes `Sportif` et `Employé` non plus comme des sous-classes de `Personne`, mais comme possédant une variable d'instance de type `Personne`. Le problème est alors reporté sur l'accès aux attributs et aux méthodes de la per-

14. L'expression « poser problème » que nous avons employée ne dénote en effet pas nécessairement une impossibilité de représentation : il est souvent possible d'adapter, dans une implantation donnée, le modèle à classes pour lui faire faire ce que l'on souhaite. Certaines de ces adaptations ne dénaturent pas le modèle mais affectent l'efficacité de la recherche de méthode comme par exemple le qualifieur « `eql` » de CLOS ; d'autres nécessitent des constructions spéciales et le rendent plus complexe comme les *patterns* d'objets du langage BETA [Kristensen *et al.*, 1987] ; d'autres enfin sont intrinsèquement contradictoires avec le modèle (de vraies instances différenciées [Stein *et al.*, 1989] par exemple) et font que le résultat de l'adaptation engendre d'autres problèmes sémantiques.

sonne: comment demander à un sportif son âge? D'une part, il y a un problème d'l'attribut `âge` est privé, d'autre part, il est nécessaire de redéfinir toutes les méthodes de `Personne` sur `Employé` et `Sportif` afin d'y mettre une ré-expédition de message[15].

- Une seconde solution consiste à créer une nouvelle classe `EmployéSportif`, sous-classe de `Employé` et de `Sportif`. Le défaut ici est que la hiérarchie résultante peut devenir rapidement inexploitable si l'on souhaite créer de multiple extensions de la classe `Personne` et les combiner.
- Les variables de classes à la SMALLTALK ou leurs équivalents permettent à des instances de partager des valeurs mais la portée de ces variables est trop large, définir une telle variable au niveau de la classe `Personne` confère le même âge à toutes les instances de `Personne`, `Sportif` et `Employé`.

En fait la représentation d'objets manipulables selon divers points de vues nécessite, dans un langage à classes, des mécanismes spécifiques tels que la « multi-instantiation » de ROME [Carré, 1989], ou ceux développés dans TROPES [Mariño *et al.*, 1990] . Dans un monde de prototypes, l'héritage entre objets permet de répondre simplement à la question posée (Fig. 6). Deux objets représentant la partie « employé » et la partie « sportif » de la personne peuvent être définis comme des extensions de l'objet représentant la « personne primitive » et détenant l'attribut `âge`; cet attribut et donc sa valeur étant alors partagés par les trois objets.

• **Objets incomplets.**

La possibilité pour un objet d'hériter les valeurs des attributs d'un autre objet est utilisée intensivement dans les langages de *frames* pour représenter des objets incomplets *i.e.* des objets dont certaines valeurs d'attributs ne sont pas connues mais dont des valeurs par défaut peuvent être trouvées dans les objets dont ils héritent. Par exemple si on cherche ce que mange `Moby-Dick`, on trouvera une valeur dans le *frame* `baleine` dont `Mobby-dick` hérite (cf. Fig. 3). Les valeurs héritées peuvent changer au gré de l'évolution des parents d'un objet.

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

5 Premières propositions de langages à prototypes

5.1 Langages fondés sur les instances prototypiques et la délégation

Henry Lieberman a repris dans [Lieberman, 1986; Lieberman, 1990] certaines idées de ACT 1 pour les appliquer à la programmation par objets. Il a proposé un modèle de programmation basé sur les instances prototypiques et l'a mis en œuvre ultérieurement dans le langage OBJECT-LISP [Allegro Common Lisp, 1989]. La figure 7 propose une version OBJECT-LISP de l'exemple « point-tortue » présenté au paragraphe 3.2. Le premier exemple concret d'un concept (`point`) sert de modèle pour définir les suivants. Les nouveaux objets sont créés par extension avec un équivalent de la primitive `extend` de ACT 1. Une extension possède un lien *est-un* vers son prototype. Les objets possèdent des attributs et des méthodes; ils communiquent en s'envoyant des messages (primitive `ask`). Il n'existe pas de mécanisme d'accessibles par envoi de message (`(ask tortue x)`) ou directement dans le corps des méthodes. Le mécanisme de délégation est mis en œuvre aussi bien lors de l'accès à la valeur d'une variable que pour l'activation d'une méthode: si la propriété n'est pas trouvée chez le receveur alors la demande est déléguée à son parent.

15. Le corps de la méthode `âge` de `Sportif` est un envoi du message `âge` à une instance de `Personne`.
16. Il est évidemment possible d'implanter une telle relation, ce qui a été fait par exemple dans certains langages de représentation [Rechenmann, 1988]

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

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

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

FIG. 7 – *Un exemple de langage à prototypes* – OBJECT-LISP.

Le point véritablement nouveau par rapport à ACT 1 est l'explicitation du mécanisme de liaison dynamique. Le mécanisme est conceptuellement parfaitement similaire à celui des langages à classes [17] . Dans notre exemple, l'envoi du message `norm` à `tortue` rend ainsi la valeur `5`, la méthode `norm` est trouvée dans le parent du receveur (`point2`) mais l'accès aux variables `x` et `y` est interprété dans le contexte du receveur initial (`tortue`), ce qui donne, via deux nouvelles délégations, `3` pour `x` et `4` pour `y`.

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

5.2 Langages fondés sur les instances prototypiques et le clonage

A la même époque, [Borning, 1986] a proposé une description informelle d'un monde d'objets sans classes organisé autour du clonage. Un prototype y représente un exemple standard d'instance et les nouveaux objets sont produits par copies et modifications de prototypes. Une fois la copie effectuée, aucune relation n'est maintenue entre le prototype copié et son clone. Conscient toutefois de la pauvreté du modèle ainsi obtenu, Borning proposait de l'étendre en instaurant une certaine forme d'héritage à base de contraintes [Borning, 1981]. Ce modèle n'a pas été développé par son auteur mais a inspiré les langages à prototypes basés sur le clonage comme KEVO [Taivalsaari, 1993],

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

OMEGA [Blaschek, 1994] ou OBLIQ [Cardelli, 1995].

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

Le problème a également été abordé plus directement sous l'angle de l'organisation des programmes : pour contourner les limitations que nous avons évoquées, a été imaginé [LaLonde *et al.*, 1986; Lalonde, 1989], à la même époque et parallèlement à celui de Lieberman, un modèle de programmation intégrant des classes et des instances (appelés cette fois *exemplars*, ce que l'on peut traduire par « exemplaires ») dotées d'une certaine autonomie. La principale raison d'être de cette proposition était d'expérimenter un découplage entre une hiérarchie de sous-typage, composée de classes et une hiérarchie de réutilisation d'implantations, composée d'instances. Les classes détiennent l'interface des types abstraits qu'elles implantent. Les instances détiennent les méthodes et sont organisées en une hiérarchie de délégation. L'héritage des méthodes se fait entre instances, celui des spécifications entre classes. Une classe peut donc avoir deux instances possédant des méthodes implantées différemment. Par exemple la classe `Liste` a une instance `listeVide` et une autre `listeNonVide` possédant des versions différentes de la méthode `append`, mais les deux instances ont la même interface définie par la classe. Par ailleurs, une instance peut hériter de n'importe quelle autre instance certaines méthodes privées nécessaires à l'implantation des méthodes composant son interface. Par exemple la classe `Dictionnaire` n'est logiquement pas définie comme une sous-classe de la classe `Ensemble`, mais une instance de `dictionnaire` peut hériter d'une instance de la classe `Ensemble` lorsque qu'un dictionnaire est implanté comme un ensemble de couples « clés – valeurs ». La hiérarchie de délégation entre instances n'est pas nécessairement isomorphe à celle d'héritage entre classes.

Cette proposition pose un certain nombre de problèmes qu'il serait trop long de décrire ici ; les auteurs l'ont d'ailleurs abandonnée, n'ayant pas réussi à faire la synthèse entre le rôle des classes et l'héritage entre instances. Cette étude reste cependant intéressante. D'une part, l'héritage entre instances est tout à fait similaire à celui proposé par Lieberman et a donc inspiré au même titre les langages à prototypes ultérieurs. D'autre part, elle proposait une première tentative d'utilisation de la délégation dans un monde de classes, idée qui revient aujourd'hui à l'ordre du jour.

5.4 Langages de frames pour la programmation

Certains langages, dits hybrides [Masini *et al.*, 1989], autorisent la définition de méthodes dans un monde dédié à la représentation et fondé sur les *frames* comme YAFOOL [Ducournau, 1991]. On peut donc les assimiler aux langages à prototypes. L'originalité de ces langages par rapport à ceux qui se voulaient uniquement basés sur les instances prototypiques (comme SELF ou OBJECT-LISP) est qu'ils permettent de définir des hiérarchies intégrant des représentants moyens (comme `Animal`) comme celle de la figure 3. Nous verrons que les concepteurs de la plupart des langages à prototypes ont dû réintroduire cette possibilité.

6 Récapitulatif des concepts et mécanismes primitifs

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

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

• **État et comportements**. Les objets sont définis par un ensemble de propriétés. Une propriété

est à la base un couple « nom[18]– valeur[19] ». Les propriétés sont soit des attributs, auquel cas la valeur est un objet quelconque, soit des méthodes, la valeur est alors une fonction.

• **Envoi de messages**. Les objets communiquent par envoi de messages ; ils sont capables de répondre aux messages en appliquant un de leurs comportements (ou éventuellement en rendant la valeur d'un de leurs attributs). Ils peuvent être vus comme des *frames* sans facettes dotés de procédures et répondant à des messages comme les acteurs ACT 1.

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

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

• **Délégation**. La délégation est le nom donné au mécanisme qui met en œuvre l'héritage, c'est-à-dire qui cherche et active une propriété héritée. La délégation est généralement implicite[20] [Stein *et al.*, 1989; Dony *et al.*, 1992]. Dans ce cas, le terme « déléguer » est une image ; dans la pratique, le système, lorsqu'il ne trouve pas de propriété dans le receveur du message, la recherche dans ses parents successifs et s'il la trouve, l'active[21]

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

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

7 Caractérisation des mécanismes et interprétation des concepts

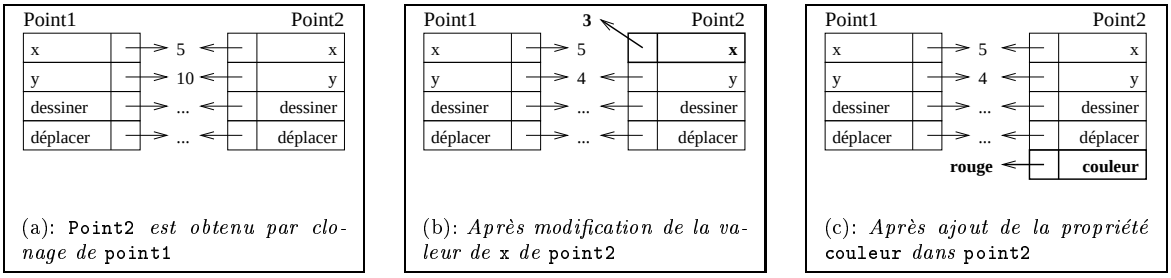
Les points, sources de confusions, qui réclament plus particulièrement des précisions sont : la caractérisation de la différence entre clonage et création différentielle, la caractérisation de la différence

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

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

20. Dans l'autre alternative, la délégation explicite, l'objet dispose d'un module de réception des messages et choisit lui-même la propriété à activer ou délègue lui-même le message à un autre objet. La délégation explicite est citée dans certains articles [Stein *et al.*, 1989] et est utilisée dans ACT 1 mais nous ne connaissons pas de langages à prototypes qui l'utilisent.

21. Le lecteur trouvera dans [Malenfant, 1995] une description formelle de la sémantique du mécanisme de délégation pour un langage à la Lieberman et dans la plate-forme *Prototalk* [Dony *et al.*, 1992] une mise en œuvre d'évaluateurs correspondants qu'il pourra étudier, étendre et modifier à sa guise. Prototalk est une plate-forme SMALLTALK permettant de simuler simplement la plupart des langages à prototypes ; elle est disponible à l'adresse : *http://www.lirmm.fr/~dony/research.html.*



(a): `Point2` *est obtenu par clonage de* `point1`

(b): *Après modification de la valeur de* `x` *de* `point2`

(c): *Après ajout de la propriété* `couleur` *dans* `point2`

FIG. 8 – *Clonage : partage ponctuel*

entre l'héritage dans les langages à classes et dans les langages à prototypes, et enfin la compréhension des diverses utilisations possibles du concept d' « extension ». Nos caractérisations sont fondées sur l'héritage et le partage, ce qui est partagé ou hérité est relatif aux propriétés. On distingue trois formes de partage[22].

- Il y a partage de noms lorsque deux objets ont chacun une propriété de même nom.
- Il y a partage de valeurs lorsque deux objets `o1` et `o2` ont chacun une propriété de nom x et que la valeur de la propriété `x` de o_1 et de o_2 est la même (égalité physique).
- Il y a partage de propriétés[23] lorsque deux objets possèdent la même propriété (même adresse et donc même nom et même valeur). Il peut y avoir partage de valeurs sans qu'il y ait partage de propriétés.

7.1 Distinction entre clonage et extension

La distinction entre clonage et extension, montrant que ces deux mécanismes ne sont pas redondants, provient de ce que le clonage et la création différentielle induisent deux formes de partage distinctes [Dony *et al.*, 1992].

• **Une caractérisation du clonage**. Tout objet cloné, partage avec son clone, au moment où celui-ci est créé, les noms et les valeurs de ses propriétés. Le clone et son original évoluent indépendamment, la modification d'un attribut du premier n'est pas répercutée sur le second : le partage est ponctuel. En effet, tout nom de propriété, par exemple `x`, `y`, `déplacer` ou `dessiner` (Fig. 8.a) de l'objet cloné, `point1` dans notre exemple, est aussi un nom de propriété du clone (`point2`) et, les deux propriétés désignées par chaque nom ont la même valeur. Mais il y a bien deux propriétés ; par exemple, si l'on modifie la valeur de la propriété `x` pour `point2`, la valeur de la propriété de même nom de `point1` n'est pas modifiée (Fig. 8.b) ; idem en cas d'ajout ou de retrait de propriété. (Fig. 8.c).

• **Caractérisation du mécanisme d'extension**. Un objet dans une hiérarchie d'objets hérite de ses parents un ensemble d'attributs et de méthodes. Tout parent partage avec ses extensions les propriétés[24] qu'il définit et que ces derniers n'ont pas redéfinies. Les propriétés du parent non redéfinies dans une extension sont aussi des propriétés de l'extension. L'extension est dépendante de son parent[25]. Cette dépendance et ce partage sont persistants, ils durent aussi longtemps qu'existe le lien entre les deux objets. Reconsidérons l'exemple de la figure 4 dans lequel l'objet `tortue` a pour parent l'objet `point2`. `Tortue` ne détient pas la propriété `x` mais hérite celle détenue par `point2`.

22. Une description plus détaillée et plus formelle en est donnée dans [Bardou et Dony, 1996].

23. L'idée de différencier le partage de noms du partage de propriétés est inspirée de la distinction entre héritage de noms et de valeurs introduite dans [Ducournau *et al.*, 1995]. Les deux distinctions n'ont en fait pas grand chose en commun car elle sont fondées sur des interprétations différentes de la notion de propriété.

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

25. La réciproque, à savoir l'indépendance du parent vis-à-vis de ses extensions sera discutée au paragraphe 7.3.

Cette propriété définit tout autant `tortue` que `point2`, elle est partagée par les deux objets. Toute modification de la valeur de cette propriété pour `point2` affecte également l'objet `tortue`.

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

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

L'étude comparative de l'héritage dans les hiérarchies de classes et dans les hiérarchies d'objets (hiérarchies de délégation) a fait l'objet de plusieurs travaux. D'après Lieberman, la délégation est un mécanisme plus général que l'héritage de classes, elle permet de le simuler. Dans [Stein, 1987], il est montré que la simulation inverse est possible à condition d'utiliser les classes comme des objets et de se servir des variables de classe à la SMALLTALK pour représenter les propriétés, ce qui est un cas très particulier. Cet article ne montre par ailleurs pas du tout ce que son titre laisse supposer, à savoir que la délégation est équivalente à l'héritage dans les hiérarchies de classes.

En fait les deux formes d'héritages sont distinctes car l'héritage dans les langages à classes n'induit aucun partage d'attributs entre instances[26]. Un objet, instance d'une classe `C` possède, via sa classe : d'une part un ensemble de méthodes (déclarées et définies dans les super-classes de `C`) et d'autre part un ensemble de noms d'attribut (déclarés dans les super-classes de `C`) dont il détient en propre la valeur. Deux objets dont les classes sont liés par un lien « sous-classe-de » partagent donc des méthodes mais uniquement des noms d'attribut. Ces objets sont indépendants en terme d'états.

Les deux formes d'héritage ne sont donc pas équivalentes. L'héritage d'attributs entre objets est caractéristique des hiérarchies d'objets ; il est à l'origine des possibilités nouvelles de représentation offertes par les langages à prototypes (cf. paragraphe 4), mais aussi de problèmes nouveaux.

7.3 Variations sur la dépendance entre parent(s) et extensions

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

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

Cette solution limite le partage d'attributs entre objets à du partage de valeurs : le parent partage avec ses extensions uniquement le nom et la valeur de ses propriétés non redéfinies. Elle rend le parent complètement indépendant de ses extensions[27].

Dans ce contexte, une extension représente systématiquement une entité différente de celle représentée par le parent, duquel elle hérite néanmoins certaines caractéristiques. Cette interprétation est ainsi adaptée à la mise en œuvre de l'exemple « point-tortue » dans lequel deux entités différentes,

26. Sauf pour le cas très particulier des variables de classe.
27. On n'obtient pas pour autant un équivalent du clonage car l'état de l'extension est toujours dépendant du parent.

représentées par les objets `point2` et `tortue` ont la même abscisse. Mais en restreignant le partage d'attributs, cette solution interdit également certaines utilisations de la délégation telle que celle utilisée dans l'exemple « personne, employé, sportif »[28] (cf. Fig. 6).

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

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

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

Les premiers problèmes que pose la programmation par prototypes sont relatifs à l'identité des objets. Nous utiliserons le terme « identité » pour désigner l'entité (ou les entités) du domaine qu'un objet représente. Avec le modèle classe-instance, un objet a une identité unique, il représente une et une seule entité du domaine. L'objet y est par ailleurs une unité d'détenant l'ensemble des valeurs des attributs ; toute modification de la valeur d'un de ces attributs est sous son contrôle. Avec l'héritage entre objets, cette bijection entre entités décrites et objets n'existe plus. En effet, dans une hiérarchie, un même objet peut représenter à la fois une entité, plusieurs entités ou des parties de plusieurs autres entités. Par exemple, l'objet `point2` de la figure 4 représente un point, mais également une partie d'une tortue puisqu'il détient son abscisse. Les représentations des entités point et tortue partagent les propriétés définies dans l'objet `point2`. Dans l'autre exemple, l'entité personne est représentée par les trois objets `personne`, `employé` et `sportif`.

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

8.1 Problèmes potentiels d'intégrité

Le premier problème potentiel est la modification d'une extension par l'intermédiaire de son parent. Il se pose avec les deux interprétations de la notion d'extension. Par exemple, l'envoi à l'objet `point2` du message `move` provoque la modifications d'attributs de `point2` et subséquemment des entités point et tortue, car l'attribut `x` est partagé[29]. Ce résultat peut néanmoins être considéré comme une conséquence naturelle de l'utilisation de la description différentielle. Si le programmeur de `tortue` ne souhaite pas que cet objet dépende de `point2`, il peut utiliser le clonage. Il y a bien modification indirecte d'une entité mais elle correspond à l'intention du programmeur.

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

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

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

Il est ainsi possible de modifier plusieurs entités en pensant n'en modifier qu'une. Plus généralement, il est impossible de placer une frontière nette entre des entités représentées par des objets appartenant à une même composante connexe d'une hiérarchie. Ces composantes devenant en pratique les réelles unités d'encapsulation des langages à prototypes ([Chambers *et al.*, 1991] emploie le terme d'*encapsulation de modules*). Une affectation peut entraîner la modification d'un très grand nombre d'entités sans qu'il soit aisé de prédire lesquelles ou même leur nombre. Briser l'encapsulation dans un tel contexte devient extrêmement simple. Il suffit, pour accéder en lecture et en écriture aux attributs d'un objet `O`, d'en créer une extension `E`, d'y définir une méthode réalisant un accès en écriture aux attributs définis sur `O` et d'envoyer le messages correspondant à `E`. Les variables d'un objet deviennent de fait des variables semi-globales, modifiables par n'importe lequel de ses descendants.

8.2 Solutions aux problèmes d'intégrité

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

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

• **Restriction du partage d'attributs.** En restreignant le partage d'attributs à du partage de valeurs (cf. paragraphe 7.3), les langages comme YAFOOL ou GARNET solutionnent le problème au détriment du pouvoir d'expression du langage.

• **Distinction entre liens de délégation.** Offrir en standard les deux possibilités est tentant, une solution mixte a ainsi été implantée dans le langage *NewtonScript* [Smith, 1995] où le programmeur a le choix entre deux sortes de liens de délégation. A l'un de ces liens (lien `_proto`) est associé du partage de valeurs et une sémantique de valeurs par défaut, tandis que du partage d'attributs est associé à l'autre (lien `_parent`). L'existence de deux types de liens de délégation complique cependant considérablement le modèle de programmation, la lisibilité des programmes et la recherche de sélecteurs. En effet, bien que le lien `_proto` soit prioritaire au lien `_parent`, il est difficile de prévoir ce qui peut se passer lorsque le sélecteur recherché est accessible en suivant deux chemins différents (incluant éventuellement des liens des deux types).

• **Contrôle des extensions.** Le langage AGORA [Steyaert, 1994] permet à chaque objet de contrôler la création de ses futures extensions. Il est impossible d'étendre un objet, s'il ne possède pas de méthodes particulières, appelées « *mixin-methods* » permettant de spécifier de façon précise les droits d'accès en lecture et en écriture aux propriétés qui seront octroyés à ses descendants. Cette solution rend au programmeur le contrôle total des accès aux propriétés d'un parent, son principal inconvénient réside dans le fait que celui-ci doit systématiquement prévoir toutes les possibilités d'extension, ce qui exclut toute réutilisation non anticipée; le problème est similaire à celui du choix de la « virtualité » des méthodes en C++.

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

8.3 Problème de la gestion des entités morcelées

Le partage d'attributs crée un autre problème, connexe aux précédents. Lorsqu'une entité est représentée par plusieurs objets, nous parlons alors d' « entité morcelée » [Dony *et al.*, 1992; Malenfant, 1996; Bardou et Dony, 1996] ; c'est par exemple le cas de l'entité tortue, dont la représentation utilise les objets `point2` et *tortue*. Le problème est qu'il n'existe aucun objet du langage représentant l'entité

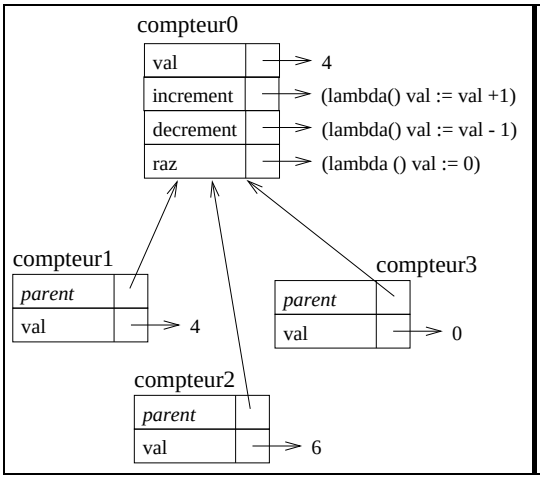


FIG. 9 – *Solution des instances prototypiques.*

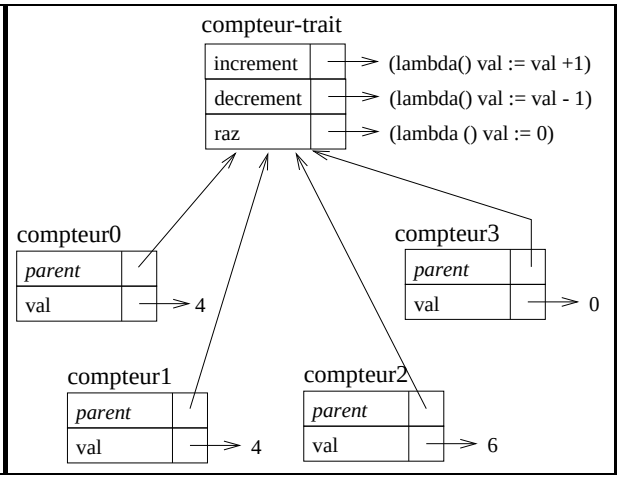


FIG. 10 – *Solution des* `traits`.

tortue dans sa globalité. On pourrait considérer que l'objet `tortue` joue ce rôle mais ceci lui confère un double statut, celui de représentant de tortue et en même temps celui de représentant d'une de ses parties. Ce double statut se manifeste si l'on demande à l'objet `tortue` de se cloner, demande-t-on un clonage de l'objet ou un clonage de l'entité? Dans le premier cas, seul l'objet doit être copié. Dans le second cas, il est nécessaire de réaliser une copie de toutes les parties de la tortue, c'est-à-dire une copie des objets `tortue` et `point2`, similaire à celle réalisée par la primitive `c-extent` de ACT 1. Le double statut se manifeste dans cet exemple par l'existence de deux primitives de clonage, ayant des noms différents, entre lesquelles le programmeur doit choisir et qui sont sources de confusion. Le problème se révèle encore mieux sur un exemple plus complexe, par exemple celui de la personne « pierre » (cf. Fig. 6) représentée par les objets `pierrePersonne`, `pierreEmployé` et `pierreSportif`. Il n'existe plus dans ce cas aucun objet susceptible de représenter l'entité « pierre » dans sa globalité. Si on souhaite la cloner, aucune primitive du langage n'est capable de réaliser cette opération puisque cette entité n'est pas réifiée; un tel clonage doit être réalisé de façon *ad hoc* par le programmeur. On ne trouve aucune solution à ce problème dans les langages existants. Des études sont en cours pour intégrer une représentation explicite d'objets morcelés dans des langages, à prototypes ou à classes [D.Bardou et C.Dony, 1995; Bardou et Dony, 1996; Bardou *et al.*, 1996] ou dans les bases de données à objets [H.Naja et N.Mouaddib, 1995]. Dans cette optique, la délégation a aussi été utilisée comme technique d'implantation du système de points de vue de *Rome* en SMALLTALK [G.Vanwormhoudt, 1994].

9 Discussion des problèmes liés à l'organisation des programmes

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

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

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

Le premier problème est celui du partage des méthodes communes à tous les objets d'une famille de clones. Considérons par exemple un ensemble de clones de l'objet `point` de la figure 4, ces objets

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

9.1.1 Gestion des familles de clones utilisant des prototypes

La méthode dite des « instances prototypiques », initialement proposée dans [Lieberman, 1986], consiste à élever un des membres au statut de représentant de la famille. Dans la pratique l'instance prototypique devient le parent de tous les autres membres. La figure 9 montre un exemple d'instance prototypique (`compteur0`) pour une famille de « compteurs ». Doter tous les membres de la famille d'une nouvelle propriété s'effectue alors simplement en la définissant sur cet objet. Cette solution, bien qu'utilisable dans la pratique, est peu satisfaisante car l'instance prototypique peut être condidérée tantôt comme un individu, tantôt comme l'ensemble des individus qu'elle représente. Le statut particulier de représentant de la famille n'est en rien matérialisé ; rien ne distingue l'instance prototypique des autres objets. Pourtant les autres membres de la famille hériteront de toutes ses évolutions. L'évolution personnelle de l'instance prototypique peut devenir contradictoire avec son statut de représentant si cette évolution fait qu'il n'est plus représentatif. Imaginons par exemple un ministre représentant de ses congénères ayant des démélés avec la justice, caractéristique nouvelle que les autres ne souhaiterons pas hériter.

Pour la plupart des langages, on a abandonné l'idée de ne manipuler que des objets représentant des entités concrètes. La méthode des « représentants moyens » repose sur le même principe que celle des instances prototypiques mais un représentant moyen (cf. paragraphe 2) est choisi comme représentant de la famille, il est éventuellement doté d'attributs et éventuellement incomplet.

La méthode des `traits`, proposée par SELF [Ungar *et al.*, 1991], proche de la précédente consiste à créer des objets, appelés `traits`, ne contenant que les méthodes partagées par les objets de la famille (la figure 10 en montre un exemple). La méthodologie des `traits` suggère de diviser la représentation d'une nouvelle sorte d'entités en deux parties : un objet `trait` qui contient les méthodes factorisées et un prototype contenant les attributs et ayant le `trait` pour parent. Obtenir un nouvel objet de la famille de clones consiste alors à cloner le prototype mais pas le `trait`.

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

Le clonage met bien en lumière les problèmes que pose le double statut des instances prototypiques, des traits ou des représentants moyens. Il est impossible d'écrire une primitive de clonage capable de cloner automatiquement et correctement un objet ayant pour parent (direct ou indirect) un tel objet. Par exemple, pour cloner le compteur représenté par les objets `compteur1` et `compteur-trait` (Fig. 10), il faut cloner `compteur1` mais pas `compteur-trait`. Plus généralement, le problème est de savoir, à partir d'un objet donné, jusqu'où la copie doit remonter afin d'obtenir une nouvelle entité sans dupliquer la bibliothèque de comportements, ce qui est impossible puisque rien ne distingue

généralement la bibliothèque des autres objets[30].

Si l'on cumule les différents problèmes, cloner automatiquement une entité morcelée membre d'une famille de clones représentée par un `trait` supposerait de pouvoir déterminer d'une part quel est l'ensemble des objets représentant l'entité et d'autre part, parmi ceux-ci, lesquels sont concrets ou abstraits.

9.1.2 Gestion automatique des familles de clones.

Une seconde approche est l'automatisation. Dans le langage KEVO, tout objet appartient à une famille de clones automatiquement gérée par le système. Si une méthode est ajoutée ou retirée à un objet, celui-ci change de famille. SELF propose une construction appelée `map`, elle aussi gérée automatiquement et invisible au programmeur. Un `map` détient les méthodes d'un objet ainsi que pour chaque attribut, son nom et l'indice auquel la valeur est rangée dans l'objet. Les `maps` permettent d'obtenir une implantation mémoire des objets identique à celle des instances d'une classe ; ils réduisent l'espace mémoire occupé par chaque objet dans lesquels seules les valeurs des attributs sont stockées (Fig. 11, partie droite). Un nouveau `map` est créé à chaque création *ex nihilo* d'un nouvel objet et tous les membres d'une famille de clone dont la structure n'a pas été modifiée partagent le même `map`.

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

9.1.3 Conclusion

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

9.2 Problème du partage entre familles de clones

Une dernière forme de partage essentielle pour l'organisation des programmes a dû être introduite dans les langages à prototypes : il s'agit du partage entre différentes familles de clones. Si on prend l'exemple d'un programme représentant des « gaulois » et des « romains », le problème est de savoir sur quel objet définir les propriétés communes à ces deux familles d'entités comme par exemple la méthode `voyager`. Dans le monde des classes, ce type de partage est réalisé par des super-classes, souvent abstraites, communes à plusieurs classes. Dans notre exemple la méthode `voyager` serait définie sur une classe `Personnage` super-classe de `Romain` et `Gaulois`. Dans les langages à prototypes,

30. Profitons de cette occasion pour discuter d'un cas particulier. YAFOOL solutionne ce dernier problème par certaines adaptations et restrictions. Les représentants moyens sont créés à l'aide d'une primitive spécifique (`defmodele`) et les objets sont dotés d'un attribut booléen indiquant s'ils ont ou non le statut de bibliothèque (ou d'abstraction). Par ailleurs il est impossible de créer des extensions d'objets non définis par `defmodele`. Ce booléen, tout en permettant de différencier les objets, met en lumière le problème du double statut. L'impossibilité de créer des extensions d'objets quelconques limite par ailleurs sérieusement les possibilités de représentations d'objets exceptionnels. YAFOOL est à la base un langage à prototypes et « au sommet », autre chose, assez proche d'un langage à classes sans en être vraiment un.

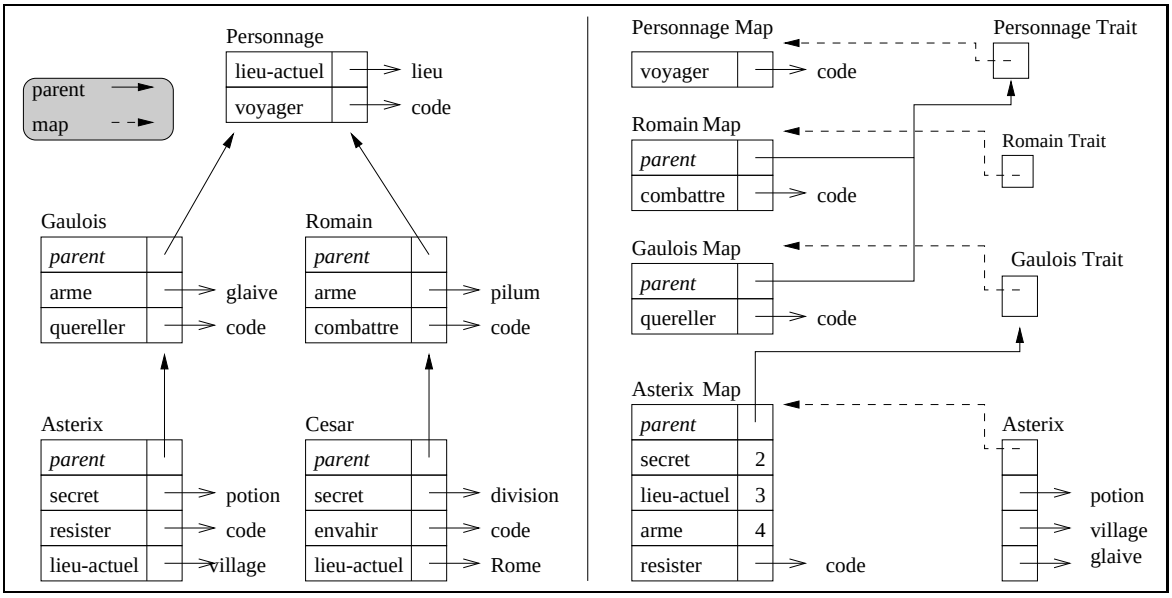


FIG. 11 – *Hiérarchies réalisées avec des représentants moyens d'une part et des* `traits` *et des* `maps` *à la* SELF *d'autre part.*

il a été mis en œuvre à l'aide des constructions que nous venons de décrire, instances prototypiques, représentants moyens, `traits` ou `maps`.

Utiliser la solution des instances prototypiques est à nouveau peu satisfaisant. En effet, il n'existe en principe aucun représentant typique de concepts abstraits (les concepts représentés par des classes abstraites dans les langages à classes), on ne peut alors en élever un au rang d'instance prototypique. Il est bien sûr possible de choisir un représentant d'un sous-concept comme représentant du concept abstrait, par exemple choisir un `gaulois`, que l'on doterait de la méthode `voyager`, comme instance prototypique de personnage, ce qui implique de définir tous les romains comme ses descendants. Ceci implique finalement une multiplication des masquages et des redéfinitions dans les descendants et rend les hiérarchies assez obscures.

La partie gauche de la figure 11 [31] montre une hiérarchie de représentants moyens permettant de répondre au problème posé. Dans cet exemple, un représentant moyen du concept personnage détient les propriétés `lieu-actuel` et `voyager`. Cette utilisation des représentants moyens ne pose a priori pas de problèmes nouveaux.

La partie droite de la figure 11 montre un équivalent en SELF, les représentants moyens sont remplacés par des `traits` ne contenant que des méthodes. De plus chaque objet possède un `map`. Le lien `parent` d'un objet est stocké dans son `map`. La gestion automatique du partage entre familles pose de nouveaux problèmes. Dans KEVO, non seulement le système gère automatiquement les familles de clones, mais il doit les organiser en une hiérarchie. La maintenance automatique de cette hiérarchie est basée sur un ensemble de pré- et post-conditions à la modification d'un objet, ou de toute une famille. Lorsqu'un objet est modifié, il doit changer de famille: le système doit déterminer quelle est cette famille et éventuellement la créer. Dans le cas d'une modification collective, le problème ne consiste plus à faire migrer un seul objet, mais une famille entière: le système vérifie donc si la famille modifiée doit ou non fusionner avec l'une de ses voisines dans la hiérarchie. En ce qui concerne les `maps` de SELF, le même problème de migration se pose, mais aucune solution spécifique n'a été mise en œuvre pour le résoudre. Plusieurs `maps` redondants peuvent donc cohabiter à la suite de modifications équivalentes appliquées à des objets différents. Deux objets ayant même structure et même comportement ne sont donc pas systématiquement considérés comme appartenant à la même

31. Extrait d'un exemple YAFOOL réalisé par J. Quinqueton

famille de clones.

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

10 Conclusion

Ce chapitre présente la genèse, les caractéristiques, les potentialités et les problèmes posés par les langages à prototypes. Nous y avons présenté la notion de prototype, telle qu'elle a été définie en sciences cognitives, montré comment cette notion a d'abord été utilisée en représentation de connaissances et en programmation distribuée. Nous avons enfin montré comment ces utilisations ont été intégrées au monde de la programmation par objets, pour former la famille des langages à prototypes. Les caractéristiques primitives de ces langages ont ainsi été décrites en en respectant la diversité.

Nous avons ensuite proposé des caractérisations du clonage, du mécanisme d'extension et de la différence entre l'héritage dans les hiérarchies d'objet et celui dans les hiérarchies de classes. Ces caractérisation nous a permis d'expliquer les différentes évolutions proposées par les langages, les différentes utilisations possibles de la délégation ainsi que de mieux appréhender les avantages et les problèmes que pose la programmation basée sur la délégation et les objets sans classes.

Quels constats peut-on tirer de cette étude?

Les langages à prototypes ont permis de découvrir ou de redécouvrir des mécanismes peu utilisés comme le clonage ou l'héritage entre objets (la délégation). Les objets sans classes, moins contraints, permettent ou simplifient la représentation d'un nombre important de situations. La délégation doit être considérée indépendamment des langages dans lesquels elle est utilisée. C'est un mécanisme utilisant un héritage entre objets ayant ses spécificités et ses applications propres, différentes des applications de l'héritage entre classes.

L'héritage entre objets induit un partage des propriétés d'état (attributs) et des propriétés comportementales (méthodes). Le partage d'attributs entre objets, absent entre instances de classes, est à la base des possibilités originales de représentation d'objets exceptionnels, d'objets incomplets ou d'objets morcelés avec points de vues. Ce même partage est aussi la cause d'une première série de problèmes que posent les langages à prototypes: inter-dépendance entre objets, d'encapsulation ou de gestion d'entités représentés par des ensembles d'objets. Limiter le partage d'attributs pour le restreindre à du partage de valeurs d'attributs apporte une solution à certains de ces problèmes tout en restreignant les possibilités de représentation.

Une seconde grande série de problèmes est liée à la disparition de la possibilité de description intensionnelle des concepts. Cette disparition se manifeste dès que l'on souhaite partager des propriétés communes à des familles d'objets. Tous les langages à prototypes ont réintroduits des objets ayant en plus du statut d'objet standard, celui de représentant d'une famille d'objets (les instances prototypique des OBJECT-LISP, les représentants moyens de YAFOOL ou les `traits` de SELF), ou celui de descripteur d'objets (les `maps` de SELF ou les familles de clone de KEVO), ou encore celui de bibliothèque de comportements (les `traits` ou les représentants moyens) ou plusieurs de ces statuts à la fois. Dans des langages en principe dédiés à la manipulation d'exemples, de prototypes, ce double statut pose des problèmes qui apparaissent clairement dès que l'on cherche à définir des langages à objets réflexifs [Mulet et Cointe, 1993].

Reconsidérons les motivations initiales ayant amené le développement des langages à prototypes. En ce qui concerne la description simplifiée des objets et les possibilités de représentation, le constat est favorable. Si un certain nombre de langages ont été et continuent à être développés, c'est parce qu'ils répondent à des besoins. Le constat est moins favorable en ce qui concerne la simplicité du modèle de programmation. Sur ce point, l'argumentation en faveur des langages à prototypes était fondée sur le fait que le modèle des prototypes est un modèle de programmation minimaliste dans lequel

tout concept jugé inutilement complexe a été supprimé[32]. La réduction du nombre et la simplicité des concepts de base ne s'est pas avérée être un facteur de plus grande simplicité de programmation. (les modèles les plus minimalistes que l'on aie pu concevoir, la machine de Turing par exemple, ne sont d'ailleurs pas ceux dans lesquels il est le plus aisé de programmer). Le monde des classes est plus rigide, celui des prototypes offre plus de liberté mais ne demande pas moins d'efforts de compréhension. Le développement d'applications importantes requiert manifestement la possibilité de décrire des abstractions. A l'heure actuelle aucun des langages à prototypes existants ne le permet de façon réellement satisfaisante.

Les problèmes relatifs à l'organisation des programmes condamnent-ils la programmation par prototypes? Si l'on réduit ce courant à des langages manipulant uniquement des « objets concrets » la réponse est certainement positive. Mais cette réduction serait une erreur. Les langages à prototypes existants, avec ou sans les solutions apportées aux problèmes d'intégrité et de partage, ont permis l'implantation de logiciels importants, on peut penser à l'environnement SELF [Agesen *et al.*, 1995], à la construction d'interfaces graphiques [Myers *et al.*, 1992] en GARNET ou AMULET, à l'architecture logicielle du *Newton* d'*Apple*). Ils ont également été utilisés comme couche basse (ou assembleurs de haut niveau) d'autres langages à objets: les langages YAFOOL ou OBJECT-LISP sont à la base des langages à prototypes mais disposent d'une couche logicielle permettant au programmeur de penser jusqu'à une certaine limite en termes de classes et d'instances[33]. Des applications importantes ont été réalisées en YAFOOL (par exemple ReSyn).

En résumé, Si les langages à prototypes présentent encore des défauts, les motivations qui ont conduit à leur développement sont toujours d'actualité, les recherches les concernant restent assez peu nombreuses. Il nous semble maintenant intéressant de travailler à une réconciliation entre prototypes et abstractions, entre la souplesse de représentation qu'offrent les objets sans classes et les possibilités de structuration offertes par des objets abstraits assumant pleinement leur statut. Une des pistes à suivre pour parvenir à ce résultat est d'arriver à définir des descripteurs d'objets et des bibliothèques de comportements qui soient eux-mêmes des objets capables de recevoir des messages, sans pour autant contraindre les objets autant que ne le font les classes.

Remerciements

Les auteurs remercient Michel Dao, Roland Ducournau, Jérome Euzenat et Amédéo Napoli pour leurs relectures avisées.

Références

[Agesen *et al.*, 1993] O. Agesen, L. Bak, C. Chambers, B.W. Chang, U. Hølzle, J. Maloney, R.B. Smith, D. Ungar, et M. Wolczko. *The SELF 3.0 Programmer's Reference Manual.* Sun Microsystems Inc. and Stanford University, 1993.

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

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

[Bardou *et al.*, 1996] D. Bardou, C. Dony, et J. Malenfant. Comprendre et interpréter la délégation. Une application aux objets morcelés. In *Actes des Journées du GDR Programmation, Orléans*, 1996. Session Programmation Objet.

32. En parlant de la conception du langage SELF : « *We employed a minimalist strategy, striving to distill an essence of object and message* » [Smith et Ungar, 1995].

33. Dans les deux cas, ces couches logicielles ne transforment pas ces langages en véritables langages à classes, le partage de valeur d'attributs et ses conséquenses étant toujours présent.

[Bardou et Dony, 1996] D. Bardou et C. Dony. Split Objects: A Disciplined Use of Delegation Within Objects. In *Proceedings of OOPSLA'96, Sans Jose, California. Special Issue of ACM SIGPLAN Notices (31)10*, pages 122–137, 1996.

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

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

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

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

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

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

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

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

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

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

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

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

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

[D.Bardou et C.Dony, 1995] D.Bardou et C.Dony. Propositions pour un nouveau modèle d'objets dans les langages à prototypes. In *Actes du Colloque Langages et Modèles à Objets, Nancy*, éditeur A. Napoli, pages 93–109. Unité de Recherche Inria Lorraine, Nancy, 1995.

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

[Dony *et al.*, 1992] C. Dony, J. Malenfant, et P. Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. In *Proceedings of OOPSLA'92, Vancouver, Canada. Special Issue of ACM SIGPLAN Notices (27)10*, éditeur A. Paepcke, pages 201–217, 1992.

[Dony, 1989] C. Dony. *Langages à objets et génie logiciel. Applications à la gestion des exceptions et à l'environnement de mise au point.* Thèse d'université, Université Pierre et Marie Curie, Paris 6, 1989.

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

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

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

[Habert et Fleury, 1993] B. Habert et S. Fleury. Suivi fin de l'analyse automatique du langage naturel basé sur l'héritage et la discrimination multiple. In *Actes de la Conférence Représentations Par Objets, La Grande Motte*, page??, 1993.

[H.Naja et N.Mouaddib, 1995] H.Naja et N.Mouaddib. Un modèle pour la représentation multiple dans les bases de données orientées-objet. In *Actes du Colloque Langages et Modèles à Objets, Nancy*, éditeur A. Napoli, pages 173–189. Unité de Recherche Inria Lorraine, Nancy, 1995.

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

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

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

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

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

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

[Lieberman, 1990] H. Lieberman. Habilitation à diriger des recherches. Mémoire de synthèse. Université Pierre et Marie Curie, Paris 6 / LITP, Institut Blaise Pascal, 1990.

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

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

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

[Masini *et al.*, 1989] G. Masini, A. Napoli, D. Colnet, D. Léonard, et K. Tombre. *Les langages à objets.* InterEditions, Paris, 1989.

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

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

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

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

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

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

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

[Smith et Medin, 1981] E.E. Smith et D. Medin. *Categories and Concepts.* Harvard University Press, Cambridge, Massachusetts, 1981.

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

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

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

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

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

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

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

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

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

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