Comprendre et interpréter la délégation, une application aux objets morcelés*

Daniel Bardou *bardou@lirmm.fr*
Christophe Dony *dony@lirmm.fr*
L.I.R.M.M – 161, rue Ada
34392 Montpellier Cedex 5 – FRANCE

Jacques Malenfant
Jacques.Malenfant@emn.fr
École des Mines de Nantes – 4, rue A. Kastler
44070 Nantes Cedex 03 – FRANCE

Résumé: Le but de cet article est d'améliorer la compréhension du mécanisme de délégation tel qu'il est défini dans [19]. Nous proposons une nouvelle caractérisation de la délégation basée sur les notions de *partage de noms*, *partage de propriétés* et *partage de valeurs*. Cela nous permet (1) de différencier clairement la délégation et l'héritage de classes et (2) d'expliquer comment une utilisation fondée de la délégation repose sur une sémantique correcte pour le partage de variables entre objets connectés par un lien de délégation. Nous décrivons ensuite un modèle d'objets morcelés qui est proposé comme un exemple d'utilisation disciplinée et sémantiquement fondée de la délégation, dans laquelle le partage de propriétés exprime des points de vues au sein d'un objet.

1 Introduction

En dépit de leur diversité [6, 27], tous les mécanismes d'héritage dans les langages de programmation ou de représentation par objets, ont au moins les points communs suivants [12]:

- Ils sont basés sur une relation $\mathcal{H}$ entre objets ou entre concepts (la relation de sous-typage entre types abstraits qui est représentée par le lien "super-classe" entre les classes d'un langage à classes, par exemple).
- Le mécanisme lui-même utillise cette relation pour réaliser l'héritage (le mécanisme d'envoi de messages d'un langage à classes, par exemple, qui effectue un *lookup* — quelle que soit la forme qu'il prend — le long des liens "super-classe" pour retrouver les propriétés héritées).
- Une interprétation ou une sémantique qui est donnée à la relation $\mathcal{H}$ pour justifier le mécanisme du point précédent. Par exemple, dans les langages à classes, la propriété formelle qui donne un sens au lien "super-classe" et qui est à la base de l'héritage est le polymorphisme d'inclusion [1] [7]. D'autres interprétations, telle que la spécialisation de concepts ou l'inclusion ensembliste ont été données dans d'autres contextes.
- Finalement, une caractéristique de tous les mécanismes d'héritage est qu'ils sont utilisés pour réaliser un certain type de partage [27].

Nous nous intéressons dans cet article au mécanisme de "délégation", tel qu'il est défini dans [18, 19, 27]. Prise dans ce contexte, la délégation est un mécanisme d'héritage [10, 27] qui est généralement associé aux langages à prototypes [19, 30, 31], des langages sans classes [2]. Le mécanisme de délégation est basé sur un lien (généralement appelé lien "parent" ou "lien de délégation") défini directement au niveau des objets [3], plutôt qu'à celui des descriptions (classes). En programmation par objets, la délégation est présentée comme un mécanisme de retransmission de messages [18, 19], décrite informellement de la façon suivante: "*un objet qui ne peut pas*

* Ce travail a été en partie financé par Marché CNRS/CNET 93 1B 142, Projet 5115 "Réseau futé".

Journées du GDR Programmation.
20, 21 et 22 novembre 1996.
Orléans.

1. La raison pour laquelle il est fondé de rechercher des méthodes dans une super-classe est que toute fonction applicable aux objets d'un type $\mathcal{T}$ (instances d'une classe $\mathcal{C}$) est supposée être également applicable aux objets d'un type $\mathcal{T}'$, sous-type de $\mathcal{T}$ (instances de sous-classes de $\mathcal{C}$).

2. Etre sans classes et être basé sur la délégation sont en fait deux caractéristiques orthogonales pour un système[20]: il existe des langages sans classes excluant la délégation [29], tandis que la délégation est partie intégrante de nombreux systèmes qu'ils aient des classes [11, 16, 17] ou non [1].

3. Par "objets", nous désignons les entités qui contiennent des valeurs et qui sont utilisées pendant l'exécution des programmes, ces entités peuvent être des classes mais considérées en tant qu'objets.

répondre à une question peut la déléguer à son parent, si le parent peut répondre, la réponse est donnée dans le contexte des valeurs de l'objet auquel la question a été posée" (voir Fig. 3-a, page 6, pour un exemple).

Si l'on compare cette description très générale aux quatres points décrits plus hauts, il apparaît que deux choses très importantes sont manquantes et limitent notre compréhension de la délégation: (1) quel type de partage est effectivement réalisé par ce mécanisme et (2) comment doit-on interpréter la relation sur laquelle le mécanisme est basé?

1. Le type de partage réalisé par la délégation n'a pas encore été bien caractérisé. En particulier, la différence entre la délégation et l'héritage de classes [4] est encore floue. A l'opposé de certaines idées généralement acceptées, nous affirmons que la délégation n'est pas de l'héritage de classes, et que la délégation n'est pas pour les objets sans classes ce qu'est l'héritage de classes pour les classes. La délégation n'est pas de l'héritage de classes parce que celle-ci et celui-ci n'induisent pas la même relation de partage entre les objets: une instance i_1 d'une classe α et une instance i_2 d'une classe β, superclasse de α, ne partagent pas les mêmes choses que ce qui est partagé par deux objets o_1 et o_2, o_2 étant le parent de o_1.

2. Une fois le type de partage effectivement réalisé par la délégation caractérisé, la question suivante est: qu'est-ce que cela signifie pour deux entités d'être reliées par un lien de délégation? Plus précisément: quelle sémantique ou quelle interprétation doit être donnée à la relation représentée par ce lien?

Dans cet article nous traitons de ces deux points. En ce qui concerne le partage, il est déjà connu que la délégation induit un partage de valeurs de variables mais une caractérisation formelle de la délégation est nécessaire pour identifier précisément et simplement les différences avec d'autres mécanismes de partage à base d'héritage. En ce qui concerne l'interprétation du lien de délégation, le problème est d'utiliser correctement le partage de variables qui, lorsqu'il est utilisé sans précautions, pose le problème de l'identité des objets [10, 20, 28]. Que cela signifie-t-il pour une entité de partager des variable avec une autre? Une première réponse peut être trouvée dans certains systèmes existants: la délégation y est utilisée pour réaliser une forme de partage permanent de valeurs par défaut entre les concepts et les objets. Nous introduisons et nous développons une deuxième réponse [3, 21] en définissant ce que nous appelons des "objets morcelés" **au sein** desquels la délégation est utilisée pour exprimer du partage entre différentes perspectives ou points de vue.

La suite de cet article est organisée comme suit. Dans la section 2, nous définissons formellement trois notions fondamentales de partage. La section 3 propose une caractérisation, dans les termes de ces trois notions, du partage dans les systèmes à classes et de celui réalisé par la délégation dans des systèmes sans-classes. Elle montre comment la délégation induit du *partage de propriétés* pour les variables et pour les méthodes. Dans la section 4, nous rappelons que le partage de variables peut avoir de mauvaises conséquences sur la modularité des objets. La section 5 rapelle une utilisation et une interprétation connue du partage de variables et la sémantique qui lui est associée dans certains systèmes existants. Dans la section 6, nous abordons une seconde utilisation fondée de la délégation qui permet d'exprimer des points de vue. La section 7 décrit ce que pourrait être un modèle dans lequel cette dernière utilisation de la délégation est maîtrisée au sein de ce que nous appelons des objets morcelés. Enfin, la section 8 renvoie à une brève comparaison entre les objets morcelés et quelques systèmes incluant la notion de points de vue.

2 Un formalisme simple pour les relations de partage

Nous définissons ici les notions de *partage de noms*, *partage de propriétés* et *partage de valeurs* [5] entre objets, ainsi que le formalisme dans lequel nous les utilisons. Nous considérons uniquement l'héritage simple [6].

Propriétés. Nous utilisons le terme "propriété" pour désigner ce que l'on appelle généralement slot, attribut, champ, variable d'instance ou encore méthode. Nous ne considérons pas des propriétés génériques telles qu'elles apparaissent dans certains langages (e.g. Clos), mais nous considérons plutôt les variables `x` de deux points différents (par exemple) comme étant à priori deux propriétés différentes.

Les propriétés sont déclarées et définies pour les objets. Un nom de propriété est ce qui identifie une propriété dans le contexte d'un objet. Des propriétés différentes peuvent avoir le même nom dans des objets différents.

Les propriétés peuvent être des entités complexes mais elles ont au moins une valeur (elles pourraient de plus avoir un type, une signature, un domaine, etc). Chaque

4. Donnons ce nom au mécanisme d'héritage qui peut être trouvé dans les langages de classes.

5. Ces notions ont été inspirées par des distinctions similaires (mais néammoins différentes) portant sur l'héritage qui ont été introduites dans [12].

6. Les trois notions de partage ne sont pas radicalement différentes en présence de l'héritage multiple, mais elles sont plus difficiles à exprimer de façon formelle.

propriété a une et une seule valeur (mais une valeur peut être la valeur de plusieurs propriétés).

Formellement, considérons un système dans lequel $\mathcal{O}$ est l'ensemble des objets, $\mathcal{P}$ l'ensemble des propriétés, $\mathcal{N}$ l'ensemble de noms de propriétés et $\mathcal{V}$ l'ensemble des valeurs. Les fonctions suivantes mettent ces ensembles en relation:

$$\begin{array}{lcccc} NomDe & : & \mathcal{P} & \longrightarrow & \mathcal{N} \\ RefProp & : & \mathcal{N} & \longrightarrow & 2^{\mathcal{P}} \\ Prop & : & \mathcal{O} \times \mathcal{N} & \longrightarrow & \mathcal{P} \\ Val & : & \mathcal{P} & \longrightarrow & \mathcal{V} \end{array}$$

Etant donnés un objet $o \in \mathcal{O}$, une propriété $p \in \mathcal{P}$ et un nom de propriété $n \in \mathcal{N}$: $NomDe(p)$ est le nom de p, $RefProp(n)$ est l'ensemble des propriétés de nom n, et $Prop(o,n)$ est la propriété de o nommée n (la propriété identifiée par n dans le contexte de o). $Val(p)$ est la valeur de p et nous notons $Valeur(o,n) = Val(Prop(o,n))$, la valeur de la propriété de o nommée n. $Val(Prop(o,n))$ peut être définie au niveau de o (valeur localement définie) ou héritée.

Nous définissons également deux ensembles: $\mathcal{N}_o$ l'ensemble des noms de propriété qui identifient une propriété dans le contexte de o (l'ensemble des propriétés déclarées pour o) et $\mathcal{N}^o$ l'ensemble des noms de propriété dont la valeur est localement définie pour o. $\mathcal{N}_o - \mathcal{N}^o$ est donc l'ensemble des noms de propriété déclarés pour o dont la valeur est héritée.

Partage. Nous définissons le partage comme une relation $\mathcal{S}$ de $\mathcal{O}$ dans $\mathcal{O}$, et nous notons $<_{\mathcal{S}}$ (resp. $\leq_{\mathcal{S}}$) la fermeture transitive (resp. réflexive et transitive) de $\mathcal{S}$.

Dire qu'un objet o a une propriété n de valeur v signifie que: (1) o a **une** propriété nommée n ($n \in \mathcal{N}_o$), (2) cette propriété est identifiée dans o par n comme étant une certaine propriété p ($Prop(o,n) = p \in \mathcal{P}$), et (3) la valeur de **la** propriété de o qui est nommée n est v ($Valeur(o,n) = Val(p) = v \in \mathcal{V}$). Le partage intervient donc à ces trois niveaux: ce qui peut être effectivement partagé sont les noms de propriété, les propriétés elles-même et les valeurs de propriété.

1. **Partage de noms**
   Ce qui est partagé dans le *partage de noms* est le fait d'avoir une propriété d'un nom donné, i.e. la déclaration, l'existence d'une propriété. Une relation de partage $\mathcal{S}$ est dite relation de *partage de noms* si:

   $$\forall (o_1, o_2) \in \mathcal{O}^2,\ o_1 <_{\mathcal{S}} o_2 \Rightarrow \mathcal{N}_{o_1} \supseteq \mathcal{N}_{o_2}$$

   ou: si $o_1 <_{\mathcal{S}} o_2$, alors si o_2 a une propriété nommée n alors o_1 en a une aussi.

2. **Partage de propriétés**
   Le *partage de propriétés* concerne le partage des propriétés elles-même. Il intervient lorsque n, un nom donné, identifie exactement la même propriété dans deux objets différents ou plus. Le *partage de propriétés* implique le *partage de noms*. Nous caractérisons de relation de *partage de propriétés*, toute relation de *partage de noms* $\mathcal{S}$ telle que:

   $$\forall (o_1, o_2) \in \mathcal{O}^2,\ o_1 \mathcal{S} o_2 \Rightarrow \\ (\forall n \in \mathcal{N}_{o_1} - \mathcal{N}^{o_1},\ Prop(o_1, n) = Prop(o_2, n))$$

   ou: si $o_1 \mathcal{S} o_2$ et si o_2 a une propriété nommée n non localement définie, alors la propriété de o_1 nommée n (avec $n \in \mathcal{N}_{o_1} - \mathcal{N}^{o_1}$) **est** également la propriété de o_2 qui est nommée n.

3. **Partage de valeurs**
   Ce qui est partagé dans le *partage de valeurs* sont les valeurs de propriétés. Une relation de *partage de noms* $\mathcal{S}$ est relation de *partage de valeurs* si:

   $$\forall (o_1, o_2) \in \mathcal{O}^2,\ o_1 \mathcal{S} o_2 \Rightarrow \\ (\forall n \in \mathcal{N}_{o_1} - \mathcal{N}^{o_1}, \\ Valeur(o_1, n) = Valeur(o_2, n))$$

   ou: si $o_1 \mathcal{S} o_2$ et si o_2 a une propriété nommée n non localement définie, alors la propriété de o_1 nommée n et la propriété de o_2 nommée n ont la même valeur. Le *partage de propriétés* implique le *partage de valeurs* mais la réciproque n'est pas nécessairement vraie. Le *partage de valeurs* est plus amplement abordé en § 5.

3 Relations de partage dans les systèmes à classes et les systèmes à délégation

Nous utilisons ici les notions de *partage de noms* et *partage de propriétés* pour caractériser le partage dans les langages à classes et dans les systèmes à délégation. Cette caractérisation montre que les deux mécanismes d'héritage sont différents.

3.1 Le partage dans les systèmes à classes

Classes, variables et méthodes. Dans les langages à classes, on fait la distinction entre les propriétés d'état (les variables) et les propriétés comportementales (les méthodes). Les déclarations de noms de variable et de noms de méthode, et les définitions des valeurs de méthodes[7] sont faites dans les classes et s'appliquent aux instances[8].

Etant donné un objet o (une instance), notons $Classe(o)$ la classe de o. Etant donnée une classe α, $\mathcal{NV}_\alpha$ est

7. On peut considérer que les valeurs de méthodes sont des lambda-expressions.

8. Nous ne considérons pas les méta-classes, ni les variables de classe.

l'ensemble des noms de variables déclarés pour les instances de α, et $\mathcal{NM}_\alpha$ l'ensemble des noms de méthodes déclarés pour les instances de α. Conformément aux définitions précédentes (voir § 2), $\mathcal{N}_o = \mathcal{NV}_{Class(o)} \cup \mathcal{NM}_{Class(o)}$ pour tout objet o.

Héritage de classes. Les classes sont organisées dans un graphe d'héritage de classes que nous notons $G_\mathcal{H} = (\mathcal{C}, \mathcal{H})$ où $\mathcal{C}$ est l'ensemble des classes et $\mathcal{H}$ la relation d'héritage. Nous notons $<_\mathcal{H}$ la fermeture transitive de $\mathcal{H}$. Etant données deux classes α et β, si $\alpha <_\mathcal{H} \beta$ (resp. $\alpha \mathcal{H} \beta$) alors α est dite sous-classe (resp. sous-classe directe) de β.

Le partage dans les systèmes à classes est du partage de noms de variable.
L'héritage de classes est utilisé pour déterminer l'ensemble $\mathcal{NV}_\alpha$ pour toute classe α. Il garantit que tout nom de variable dont la déclaration est détenue par α ou l'une de ses super-classes est élément de $\mathcal{NV}_\alpha$:

$$\forall(\alpha, \beta) \in \mathcal{C}^2,\ \alpha <_\mathcal{H} \beta \Rightarrow \mathcal{NV}_\alpha \supseteq \mathcal{NV}_\beta$$

De ce fait, il est facile de prouver l'existence de $\mathcal{S}_\mathcal{H}$, une relation relation de partage de noms de variable:

$$\forall(o_1, o_2) \in \mathcal{O}^2,\ o_1 \mathcal{S}_\mathcal{H} o_2 - \left\{ \begin{array}{c} Classe(o_1) = Classe(o_2) \\ or \\ Classe(o_1) \mathcal{H} Classe(o_2) \end{array} \right.$$

L'existence des variables de toute instance d'une classe α est partagée avec toute autre instance de α ou d'une sous-classe de α. Chaque instance détient une valeur propre pour chaque variable déclarée dans sa classe. Il n'y a donc pas de partage de propriétés ou de valeurs pour les variables.

Le partage dans les systèmes à classes est du partage de propriétés pour les méthodes.
De même que pour les déclarations de noms de variable, l'héritage de classes est utilisé pour déterminer $\mathcal{NM}_\alpha$ pour toute classe α:

$$\forall(\alpha, \beta) \in \mathcal{C}^2,\ \alpha <_\mathcal{H} \beta \Rightarrow \mathcal{NM}_\alpha \supseteq \mathcal{NM}_\beta$$

Il est donc trivial d'établir que $\mathcal{S}_\mathcal{H}$ est aussi une relation de partage de noms de méthode.

De plus, l'héritage de classes est utilisé pour l'activation des méthodes. Les méthodes qui peuvent être activées en envoyant un message à un objet donné o sont celles dont la valeur est définie dans $Classe(o)$ ou dans l'une des super-classes de $Classe(o)$. Nous les distinguons en appelant $\mathcal{NM}^{Class(o)}$ l'ensemble des premières (l'ensemble des dernières étant $\mathcal{NM}_{Class(o)} - \mathcal{NM}^{Class(o)}$).

Etant donné un objet o_1, lorsqu'un message est envoyé afin d'activer la méthode m de o_1 nommée n, un lookup débute dans la classe de o_1 et est éventuellement continué le long des liens d'héritage jusqu'à ce que la définition de la valeur de m soit trouvée dans une certaine classe β. β ne peut contenir qu'**une** seule définition de valeur pour une méthode de nom n, cela implique que n identifie une seule méthode dans le contexte de β, cette méthode est m. Comme nous aurions pu choisir, à la place de o_1, n'importe quel autre objet o_2 parmi les instances de n'importe quelle classe α, telle que $Class(o_2) \leq_\mathcal{H} \beta$ et $\alpha \leq_\mathcal{H} \beta$, pour que le lookup se termine en β, nous pouvons conclure que:

$$\forall(o_1, o_2) \in \mathcal{O}^2,\ o_1 \mathcal{S}_\mathcal{H} o_2 \Rightarrow (\forall n \in \mathcal{NM}_{Classe(o_1)} - \mathcal{NM}^{Classe(o_1)}, Prop(o_1, n) = Prop(o_2, n))$$

Le partage dans les systèmes à classes peut donc être caractérisé comme du partage de propriétés pour les méthodes. Toutes les instances d'une classe β ont le même comportement et, pour toute α sous-classe directe de β, les méthodes des instances de β non redéfinies dans α sont aussi des méthodes des instances de α.

3.2 La délégation réalise du partage de propriétés (pour les variables et les méthodes)

Dans les systèmes à délégation, l'héritage est déterminé à l'exécution, implicitement ou explicitement et directement au niveau des objets [27].

Les objets sont liés entre eux par des liens de délégation (liens parent) dans un graphe de délégation $G_\mathcal{D} = (\mathcal{O}, \mathcal{D})$. $\mathcal{D}$ est la relation de délégation, et nous notons $<_\mathcal{D}$ sa fermeture transitive et $\leq_\mathcal{D}$ sa fermeture réflexive et transitive. Si, étant donnés deux objets o_1 et o_2, $o_1 <_\mathcal{D} o_2$ (resp. $o_1 \mathcal{D} o_2$) alors o_1 est appelé un descendant (resp. un fils) de o_2, et o_2 un ancêtre (resp. le parent) de o_1.

$\mathcal{D}$ est également une relation de partage de noms entre objets. Effectivement, pout tout objet o, une valeur peut être calculée pour tout nom de propriété déclaré dans tout ancêtre de o: soit cette valeur est explicitement définie pour o, soit elle est déduite en appliquant le mécanisme de délégation. De ce fait tout nom de propriété qui est le nom d'une propriété d'au moins l'un des ascendants de o est aussi le nom d'une propriété de o:

$$\forall(o_1, o_2) \in \mathcal{O}^2,\ o_1 <_\mathcal{D} o_2 \Rightarrow \mathcal{N}_{o_1} \supseteq \mathcal{N}_{o_2}$$

Nous pouvons aussi qualifier $\mathcal{D}$ de partage de propriétés. Considérons le lookup effectué à la réception d'un message envoyé à un objet o_1, demandant la valeur d'une propriété p nommée n: ce lookup finit dans un certain objet o_2. o_2 est l'objet détenant la définition de la valeur de p, et comme o_2 ne peut détenir qu'**une seule** définition de valeur pour la propriété nommée n, n identifie

également p dans le contexte de o_2 ($Prop(o_1, n) = p = Prop(o_2, n)$). Remarquons que nous aurions pu choisir n'importe quel objet o_3, tel que $o_1 \leq_{\mathcal{D}} o_3$ et $o_3 \leq_{\mathcal{D}} o_2$, pour pouvoir appliquer le même raisonnement:

$$\forall (o_1, o_2) \in \mathcal{O}^2,\ o_1 \mathcal{D} o_2 \Rightarrow \\ (\forall n \in \mathcal{N}_{o_1} - \mathcal{N}^{o_1},\ Prop(o_1, n) = Prop(o_2, n))$$

Un objet partage chacune de ses propriétés avec chacun de ses fils qui ne détient pas une définition de la valeur de cette propriété.

4 Critique du partage de propriétés pour les variables

Nous avons montré que la délégation réalise du partage de propriétés à la fois pour les variables et les méthodes tandis que l'héritage de classes ne réalise du partage de propriétés que pour les méthodes et du partage de noms pour les variables. La différence entre les deux mécanismes est donc le partage de propriétés pour les variables qui est une caractéristique de la délégation. Nous faisons une critique de ce partage avant de nous concentrer sur la sémantique du lien parent.

4.1 Le partage de propriétés pour les variables est utile

Considérons, dans un système à delégation, des objets représentant une personne — appelons-là *Pierre* — voir Fig. 1 (un tel exemple est également considéré dans [3, 10, 21]). Supposons que nous ayons d'abord voulu considérer *Pierre* comme une simple personne, nous avons créé l'objet `PierrePersonne` avec les variables `adresse`, `age`, `nom` et `téléphone`, et une méthode `vieillir`. Puis l'objet `PierreSportif` a été créé, avec les variables `endurance` et `poids`, et un lien parent vers `PierrePersonne`. Le lien de délégation entre `PierreSportif` et `PierrePersonne` garantit que toutes les propriétés de ce dernier sont partagées par les deux objets, et nous pouvons considérer alternativement:

- *Pierre* en tant que sportif (dont les propriétés sont `poids`, `endurance`, `adresse`, `age`, `nom`, `téléphone` et `vieillir`),
- et *Pierre* en tant que personne (n'ayant pas les propriétés `poids` et `endurance`).

De plus, il est possible de créer d'autres fils de `PierrePersonne` suivant d'autres aspects: nous créons l'objet `PierreCinéphile`, fils de `PierrePersonne`, et détenant les définitions des variables `acteurFavori`, `filmFavori` et `réalisateurFavori`. Dans cet exemple, les trois objets sont supposés dénoter la même entité du monde

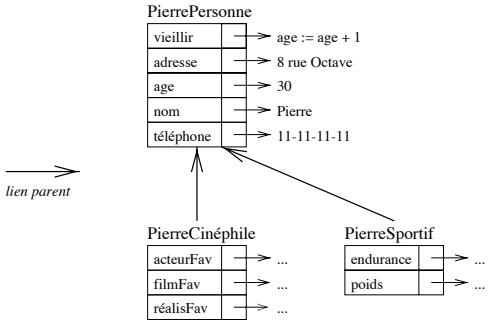

FIG. 1 – *Une représentation de* Pierre *éclatée en trois objets.*

réel, la personne *Pierre*. Par exemple, la variable `adresse` détenue par `PierrePersonne` est supposée être aussi la variable adresse de `PierreSportif` et de `PierreCinéphile`. Pour cette raison, le partage de propriétés pour les variables réalisé par la délégation est, **dans ce cas**, bien adapté: on peut modifier la valeur de `adresse` en envoyant un message d'affectation à l'un des objets, la modification a lieu dans `PierrePersonne` et est immédiatement effective pour les trois objets.

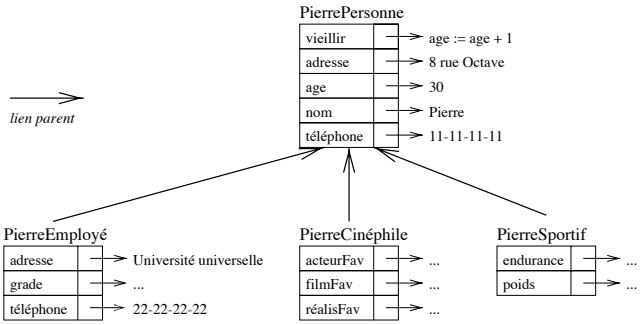

FIG. 2 – *Une autre représentation de* Pierre *complétée par un aspect employé.*

De plus, toute propriété peut être redéfinie dans l'un des fils de `PierrePersonne`. Par exemple, on peut créer un nouvel objet `PierreEmployé`, fils de `PierrePersonne`, et dénotant *Pierre* en tant qu'employé, dans lequel les variables `adresse` et `téléphone` sont redéfinies (voir Fig. 2). Du fait que les propriétés redéfinies ne sont pas partagées, *Pierre* est alors représenté comme ayant une adresse et un numéro de téléphone différent lorsqu'il est au travail.

On ne peut pas obtenir une représentation équivalente de façon simple dans un langage à classes puisque l'héritage de classes ne réalise pas de partage de propriétés pour les variables.

4.2 Le partage de propriétés pour les variables pose le problème de l'identité des objets

Nous rappelons ici comment le partage de propriétés pour les variables brisent les frontières des objets et par là même viole le principe d'encapsulation [28]. De façon plus générale, nous soulevons la question de l'interprétation du lien de délégation.

Considérons un autre exemple dans un langage à délégation. Un point à 5@10 est représenté par un objet `Point1`, et une tortue à 10@10 et orientée à 90° par un objet `Tortue1` (voir Fig. 3-a). Parce qu'un objet-tortue est comme un objet-point ayant une variable (`orientation`) et deux méthodes (`tourner` et `avancer`) de plus, et, plus spécifiquement, parce que `Tortue1` a la même valeur (du moins lors de sa création) pour `y` que `Point1`, nous avons fait de `Tortue1` un fils de `Point1`. Si l'on demande alors à `Tortue1` d'avancer jusqu'à 10@14, la valeur de son `y` doit être changée. La définition de la valeur de `y` n'est pas trouvée dans `Tortue1` mais dans `Point1` où la modification est effectuée (voir Fig. 3-b). Comme la délégation réalise du partage de propriétés le lien parent octroie à `Tortue1` un accès, non seulement en lecture mais aussi en écriture, à la variable `y` détenue par `Point1`.

En conséquence, si nous exigeons seulement d'un objet qu'il soit une entité capable de recevoir des messages, alors `Tortue1` et `Point1` peuvent être considérés comme deux objets différents, mais cela devient faux dès lors que nous exigeons également d'un objet qu'il soit une entité individuelle et indépendante. Dans la plupart des langages à délégation, il n'y a pas de prérequis pour pouvoir créer un objet comme étant le fils d'un autre. L'accès total aux propriétés d'un objet peut alors être obtenu de façon inattendue simplement en créant un fils de celui-ci. Ou bien le partage de propriétés réalisé, à la base, par la délégation doit être restreint à du partage de valeurs (voir définition en § 2), ou bien tous les objets connectés par des liens parent doivent toujours être considérés comme les parties de la représentation d'une seule entité globale (comme dans le cas de notre exemple précédent § 4.1). Choisir entre ces deux alternatives revient à choisir entre deux utilisations différentes, mais toutes deux fondées, de la délégation.

5 Une première interprétation: objets individuels et valeurs par défaut

Une première interprétation fondée du lien de délégation peut être trouvée dans certains langages d'acteurs (e.g. Act1 [18]), des langages de frames (e.g. Y3 [11]) mais aussi des langages à prototypes (e.g. le langage KR dans Garnet [23]). Dans ces langages, les propriétés d'un objet ne peuvent pas être modifiées en envoyant un message d'affectation à l'un de ses descendants. Un objet a au moins autant de propriétés que son parent, chacune de ses propriétés est identifiée par le même nom dans le contexte de chacun des deux objets et a la même valeur par défaut.

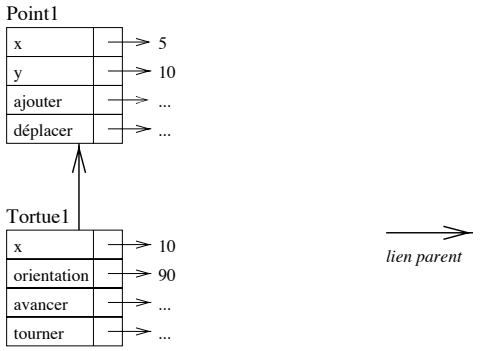

FIG. 3 —a: *avant la modification de* `y`.

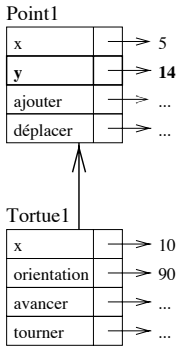

FIG. 3 —b: *après la modification de* `y` *avec du partage de propriétés.*

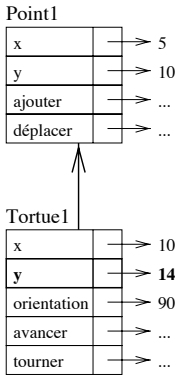

FIG. 3 —c: *après la modification de* `y` *avec du partage de valeurs.*

FIG. 3 – `Point1` *et* `Tortue1` *partage la variable* `y`. *La modification de la valeur du* `y` *de* `Tortue1` *mène résulte en 3-b avec du partage de propriétés, ou en 3-c avec du partage de valeurs.*

Considérons à nouveau l'exemple du point et de la tortue (voir Fig 3-a). La raison pour laquelle on a fait de `Tortue1` un fils de `Point1` est clairement dans ce cas la réutilisation de la définition des propriétés de `Point1`. Nous ne voulons pas que ces propriétés soient aussi les propriétés de `Tortue1`, nous voulons seulement que `Point1` fournisse des définitions par défaut à `Tortue1`. Un message d'affectation de `y` envoyé à `Tortue1` ne devrait donc pas résulter en la modification de la valeur du `y` de `Point1` comme en Fig. 3-b mais plutôt en la redéfinition de `y` dans `Tortue1` comme en Fig. 3-c. Une telle interprétation de l'affectation revient à restreindre le partage de propriétés (qui est à la base réalisé par la

délégation) à du partage de valeurs. Cela est généralement réalisé en Y3 et en KR en considérant systématiquement l'affectation d'une variable non localement définie comme la création et l'initialisation d'une nouvelle variable.

La délégation octroie toujours un accès aux variables du parent en lecture mais plus en écriture. Les frontières entre les objets sont alors claires: dans notre exemple `Point1` et `Tortue1` sont vraiment deux objets différents et ils peuvent évoluer individuellement. La seule manière de modifier la valeur du `y` de `Point1` est d'envoyer explicitement un message à `Point1`. Bien sûr, cela modifierait également la valeur de `y` pour `Tortue1`, mais ce n'est pas inattendu puisque cette valeur est supposée n'être qu'une valeur par défaut.

6 Une deuxième interprétation: des points de vue sur une seule entité

Nous avons expliqué comment le partage de propriétés peut être restreint à du partage de valeurs afin de résoudre le problème de l'identité des objets. Mais nous avons aussi remarqué l'utilité du partage de propriétés au travers d'un exemple (voir § 4.1). Nous nous intéressons maintenant à une deuxième interprétation fondée de la délégation qui conserve le partage de propriétés.

Considérons à nouveau la représentation de *Pierre* qui est éclatée en quatre objets (voir Fig. 2). Que cela signifie-t-il exactement? Selon Ferber, des objets dénotant une même entité (objets corréférents) dénotent des points de vue sur cette entité [14]. Il devrait être clair que `PierrePersonne` dénote *Pierre* en tant que personne, `PierreEmployé` *Pierre* en tant qu'employé, `PierreCinéphile` *Pierre* en tant que cinéphile, et `PierreSportif` *Pierre* en tant que sportif.

Eclater une représentation en plusieurs objets dans une hiérarchie de délégation est simplement une façon naturelle de représenter des points de vue. Comme dans une hiérarchie de descriptions, le point de vue le plus général est dénoté par le sommet de la hiérarchie tandis que les plus spécifiques sont dénotés par les feuilles de celle-ci. Dans notre exemple, personne est un point de vue plus général que employé, sportif ou cinéphile.

Toutefois, ce type de représentation pose des problèmes puisque nous ne pouvons pas manipuler la représentation de *Pierre* dans son ensemble. Nous ne pouvons envoyer de messages à la représentation globale mais seulement à l'un des objets dénotant un point de vue de *Pierre*. Nous ne pouvons pas non plus manipuler cette représentation globale en tant que structure, pour la dupliquer par exemple. Le problème de l'identité des objets n'est toujours pas résolu, mais nous savons où placer la frontière d'une telle représentation: autour de celle-ci toute entière, c'est à dire autour des quatre objets dans notre exemple. Une solution à ces problèmes est de donner le statut d'objet à la représentation globale et enlever ce statut à chacune de ses parties. En effet, nous ne pouvons plus considérer `PierrePersonne`, par exemple, comme un objet, sinon il y aurait conflit entre la frontière de cet objet et celle de la represésentation entière. L'inadéquation entre le système et le monde réel doit être éliminée: il faut garantir une correspondance d'un à un entre les objets dans le système et les entités du monde réel qui y sont représentées. Nous proposons dans ce but les *objets morcelés*.

7 Objets morcelés

Nous avons montré dans la section précédente que la délégation peut être utilisée pour obtenir des représentations avec points de vue. Nous avons également remarqué que une solution aux problèmes posés par ces représentations repose sur une correspondance un-à-un entre les objets du système et les entités du monde représenté. Nous proposons donc un modèle d'objets morcelés dans lequel cette correspondance est respectée.

7.1 Un modèle de base pour les objets morcelés

Un *objet morcelé* est défini comme une collection de *morceaux*. Les propriétés d'un objet morcelé sont stcokées dans ses morceaux. Les morceaux sont organisés au sein d'un objet dans une hiérarchie de délégation (avec partage de propriétés). Un objet morcelé dénote une seule entité du monde réel et ses morceaux dénotent des points de vue sur cette entité. **Les morceaux n'ont pas le statut d'objet, tandis que les objets morcelés l'ont.** Pour illustrer ce qu'est un objet morcelé, considérons une nouvelle représentation plus complète de *Pierre*, celle de la figure 4. *Pierre* est maintenant représenté par un seul objet morcelé `Pierre`, contenant neuf morceaux. **`Personne`, `Employé`, `Cinéphile` et `Sportif` ne sont plus des objets mais seulement des morceaux de `Pierre`.** Nous détaillons ci-dessous les principes fondamentaux qu'un langage centré sur l'objet, comportant le clonage, devrait inclure pour pourvoir à la manipulation d'objets morcelés. Une description plus complète de ce modèle peut être trouvée dans [3], de même que la sémantique dénotationnelle d'un modèle très similaire dans [21].

Nommage et accès. Les objets morcelés sont des entités de première classe, ils sont directement accessibles. Ce n'est pas le cas des morceaux, auxquels on ne peut accéder qu'au travers de l'objet morcelé englobant, en spécifiant un nom de morceau.

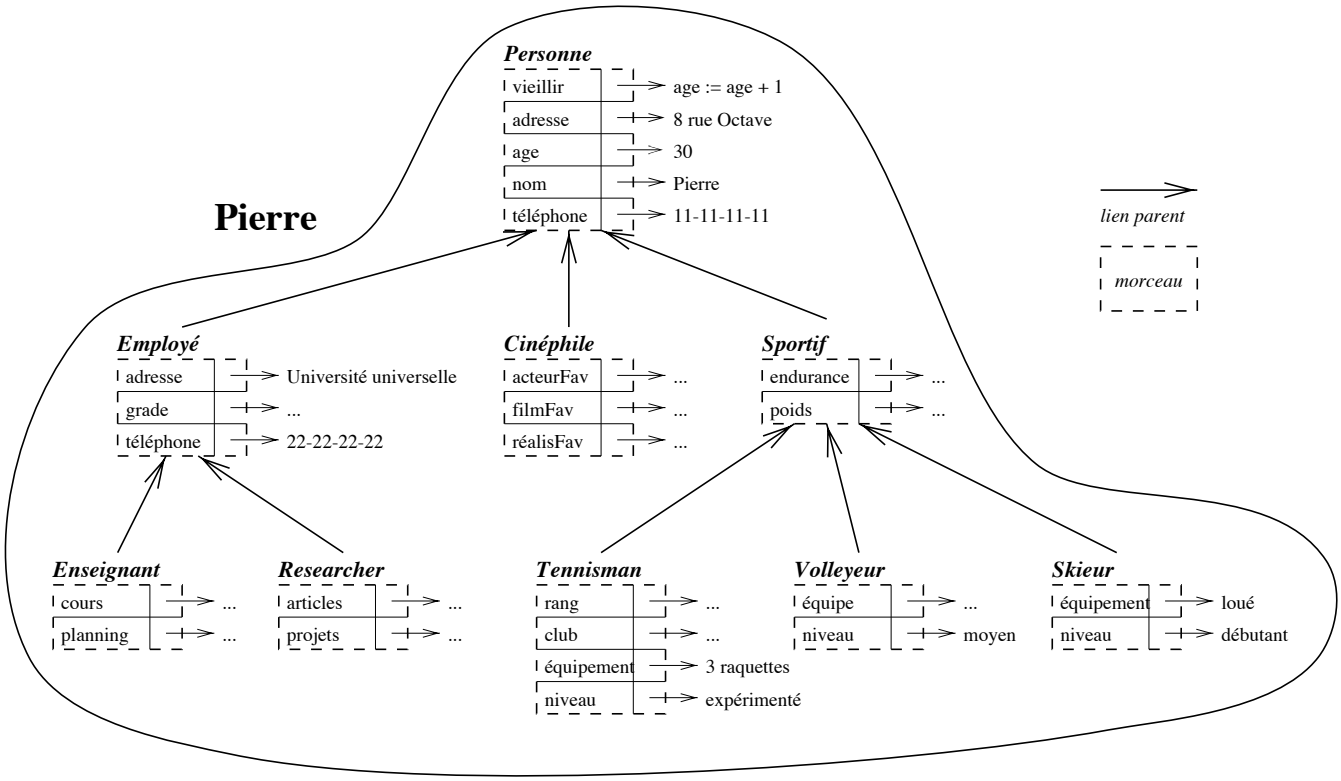

FIG. 4 – *Un objet morcelé représentant une personne, chaque morceau dénote un point de vue sur cette personne.*

Création. Les objets morcelés sont créés par clonage. Un cas particulier de création est la création ex-nihilo: elle est réalisée en clonant *l'objet morcelé vide* (un objet prédéfini dans le langage, vide de morceaux) et en effectuant autant d'ajout de morceaux que nécessaire.

Clonage. Cloner un objet morcelé résulte en un nouvel objet morcelé, initialement composé du même ensemble de morceaux détenant les mêmes propriétés. Le clonage d'un objet morcelé est donc une copie profonde de la hiérarchie des morceaux dans laquelle chaque morceau est copié superficiellement.

Modification. Les objets morcelés peuvent être modifiés sur la base d'un morceau. On peut ajouter, supprimer ou modifier un morceau dans un objet morcelé.

- L'ajout d'un morceau consiste à créer un morceau vide étant le fils d'un morceau existant. Le nom du morceau existant et celui du nouveau morceau doivent être spécifiés.
- La suppression d'un morceau entraîne la suppression de tous ses descendants dans la hiérarchie: rappelons qu'un morceau dénote un point de vue plus général que ceux dénotés par ses descendants. Si *Pierre* n'est plus un sportif, il est clair qu'il ne peut plus être un skieur, un volleyeur ou un tennisman.
- La modification d'un morceau consiste à ajouter ou supprimer une propriété, changer la valeur d'une variable ou modifier le corps d'une méthode détenue par ce morceau.

Envoi de message. Les valeurs de propriétés d'un objet morcelé sont accessibles par envoi de message. Comme les variables et les méthodes sont détenues par les morceaux, et comme les morceaux dénotent des points de vue, les messages sont envoyés sur la base d'un point de vue. Lorsqu'un message est envoyé à un objet, un point de vue doit être spécifié en donnant le nom du morceau le dénotant. Un lookup est alors effectué, il commence dans ce morceau et est éventuellement continué dans les morceaux ascendants, afin de calculer et renvoyer la réponse. L'envoi de message peut être étendu de telle sorte que l'on puisse envoyer des messages sans spécifier de point de vue. Nous en discutons plus amplement en § 7.2.

Pseudo-variables. Comme dans tout langage à objet, une pseudo-variable *self* est liée, pendant l'activation d'une méthode, au receveur courant, c'est à dire un objet morcelé. Pour le cas particulier où l'on désire envoyer à *self* un message suivant le même point de vue que le message alors évalué, une seconde pseudo-variable *this-*

Viewpoint doit aussi être disponible. Pendant l'évaluation d'un message, *thisViewpoint* est liée au nom du morceau dénotant le point de vue courant.

Contrôles de cohérence de la structure. Lors d'une modification, il est possible de vérifier que la structure d'arbre du graphe des morceaux d'un objet morcelé est préservée.

7.2 Envoi de message étendu

Nous avons présenté un modèle de base dans lequel l'utilisateur a accès à chacun des points de vue dénoté par un morceau dans un objet morcelé. Une question intéressante alors est de savoir s'il y a seulement autant de points de vue qui sont dénotés dans un objet morcelé que le nombre de morceaux qui le composent. Nous montrons qu'il y a plus de points de vue que de morceaux. Nous donnons quelques indices sur la manière d'envoyer des messages selon tous ces points de vue, et en particulier le *point de vue global*, celui que l'on peut considérer être la "réunion" de tous les autres.

L'objet morcelé `Pierre` (voir Fig. 4) est une collection de neuf morceaux et chacun de ces morceaux dénote un point de vue sur *Pierre*. `Pierre` dénote donc au moins neuf points de vue. Nous pouvons accéder à chacun de ces points de vue en envoyant des messages. Nous pouvons, par exemple, demander à *Pierre* en tant que chercheur de quels articles il est auteur en envoyant un message à `Pierre`, en indiquant le morceau `Chercheur` et le sélecteur `articles`. Mais comment obtenir cette information sans savoir que la propriété `articles` de `Pierre` est détenue dans le morceau `Chercheur`? Ne devrions nous pas être capable d'accéder à *Pierre* dans son ensemble, selon le point de vue global depuis lequel toute propriété définie dans les morceaux de `Pierre` est accessible? Nous voudrions répondre oui à cette question, mais il n'y a pas de morceau qui dénote *Pierre* en entier: le point de vue global n'est qu'implicitement dénoté dans `Pierre`, et nous ne pouvons pas à priori envoyer de messages suivant ce point de vue.

Si nous permettons aux morceaux d'avoir plusieurs parents, une solution naïve à ce problème serait de créer un morceau dénotant *Pierre* en entier. Ce morceau vide pourrait être le fils de chacune des feuilles de la hiérarchie afin d'avoir accès par délégation à toutes les propriétés définies pour `Pierre`.

Toutefois, une remarque importante est que le point de vue global de *Pierre* peut être considéré comme étant la composition de tous les points de vue de *Pierre* explicitement dénotés par les morceaux. Nous pouvons alors considérer que toutes les combinaisons possibles de morceaux dénotent des points de vue intéressants. Par exemple, nous pourrions vouloir considérer *Pierre* selon le point de vue employé-chercheur (dénoté par la combinaison des deux morceaux `Employé` et `Chercheur`). Remarquons également que le point de vue personne-et-employé (dénoté par la combinaison de `Personne` et `Employé`) n'est pas le même point de vue que le point de vue employé: demander l'adresse de *Pierre* en tant qu'employé est non ambigü, mais le devient pour l'adresse de *Pierre* en tant qu'employé et personne.

Compter les points de vue (implicitement ou explicitement) dénoté dans un objet morcelé revient donc à compter le nombre de sous-ensembles non vides de morceaux de l'objet morcelé, Dans notre exemple, `Pierre` a 9 morceaux et dénote donc 2^9-1, c'est à dire 511, points de vue sur *Pierre*. Nous en concluons que créer un morceau vide pour dénoter explicitement chaque point de vue implicitement dénoté n'est pas une solution applicable en regard de la taille mémoire alors nécessitée.

L'envoi de message peut être étendu de telle sorte que l'on puisse envoyer des messages suivant tout point de vue dénoté dans un objet morcelé. Les points de vue implicitement dénotés peuvent être spécifiés par une liste de noms de morceaux. Il est également possible de considérer systématiquement un message envoyé sans spécification de point de vue comme un message envoyé suivant le point de vue global. Les propriétés d'un objet morcelé peuvent alors être activées sans connaître le nom des morceaux par lesquels elles sont détenues au profit de l'encapsulation.

Comme certains messages peuvent être ambigüs, il faut choisir une stratégie de lookup qui, au minimum, détecte ces ambiguïtés. Il semble raisonnable d'exiger de cette stratégie qu'aucun lookup ne soit effectué dans les morceaux qui ne pas en relation (morceaux non ascendants et non descendants des) avec les morceaux spécifiés comme point de vue. Nous avons proposé une telle stratégie dans [3], basée sur une sémantique unifiant l'envoi de messages selon des points de vue explicites ou implicites.

8 Notions de point de vue dans d'autres systèmes

Le but principal de ce travail n'était pas à l'origine de concevoir un nouveau modèle intégrant la notion de point de vue, mais plutôt de comprendre plus amplement la délégation et sa sémantique. Néammoins, la notion de point de vue a émergé naturellement lorsque nous avons pensé aux représentations éclatées. De nombreux autres travaux ont été menés sur les points de vue et les objets, et il est intéressant de les comparer avec que nous avons présenté ici.

Dans une précédente version de cet article [4], nous avons comparé la notion de point de vue apparaissant

dans le modèle des objets morcelés avec celles apparaissant dans d'autres systèmes: les perspectives de Loops [5, 6], la corréférentialité de Ferber [13, 14], la représentation multiple et évolutive de ROME [8, 9], les points de vue de Tropes, les sujets d'Harrison et Ossher [15, 24], la modélisation de rôles [2] et les perspectives de Us [26]. Les différences entre les objets morcelés et ces systèmes sont parfois fortes, parfois subtiles. Les plus fortes similitudes ont été relevées par rapport à ROME et Us.

9 Conclusion

Dans cet article, nous avons introduit un formalisme dans lequel les notions de *partage de noms*, *partage de propriétés* et *partage de valeurs* nous ont permis de caractériser précisément le type de partage réalisé dans les systèmes à classes et celui réalisé par la délégation.

Nous avons ensuite montré comment le *partage de propriétés*, caractéristique de la délégation, peut être utilisé pour obtenir des représentations avec points de vue. Nous avons aussi expliqué qu'une utilisation sûre de la délégation repose sur le choix d'une sémantique correcte des liens parent. Nous avons abordés deux interprétations fondées pour ces liens: nous avons rappelé la première qui a été adoptée dans plusieurs systèmes, et introduit la seconde dans laquelle des points de vue sont exprimés au sein d'objets morcelés.

Nous avons expliqué comment les objets morcelés peuvent être manipulés dans un langage sans classes, où la création est essentiellement effectuée par clonage. Toutefois, nous pensons qu'il existe un modèle à classes équivalent: on peut penser à des classes d'objets morcelés. En effet, la délégation a lieu au sein des objets morcelés, entre morceaux, et nous n'avons rien imposé sur l'héritage entre objets morcelés, ou même sur leur organisation relative dans le système. Nous sommes en train d'étudier le pour et le contre du choix d'un langage à classes ou à prototypes pour implanter les objets morcelés. Les facteurs déterminant de ce choix incluent:

- les solutions alors possibles pour la factorisation de propriétés communes, un problème récurrent des langages à prototypes [3, 10];
- la manière dont les points de vue sont exprimés dans les objets morcelés, le niveau (celui des classes ou celui des objets) auquel sont définis les noms de morceaux;
- la création et la modification dynamique d'objets morcelés.

Un autre point important est l'encapsulation. Dès lors qu'il est possible de mettre des frontières autour des objets morcelés, l'encapsulation peut être garantie dans notre modèle mais nous n'avons pas donné de détails à ce sujet. De plus, il pourrait y avoir deux niveaux d'encapsulation: l'encapsulation des morceaux et celle (traditionnelle) des propriétés.

Remerciements

Nous remercions Bernard Carré, Roland Ducournau et Gilles Vanwormhoudt pour les discussions que nous avons eu sur les points de vue et la délégation.

Références

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

[2] E.P. Andersen et T. Reenskaug. System Design by Composing Structures of Interacting Objects. Dans *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP '92), Utrecht NL, Lecture Notes in Computer Science 707*, pages 133–152, 1992.

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

[4] D.Bardou et C. Dony. Split Objects: a Disciplined Use of Delegation within Objects. Dans *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '96), San Jose, California, USA, ACM SIGPLAN Notices (31)10*, pages 122–137, 1996.

[5] D.G. Bobrow, M. Stefik. *The LOOPS Manual*. Memo KB-VLSI-81-13, Xerox Palo Alto Research Center, 1983.

[6] D.G. Bobrow et M. Stefik. Object-Oriented Programming: Themes and Variations. Dans *The AI Magazine (6)4*, pages 40–62, American Association for Artificial Intelligence, 1986.

[7] L. Cardelli et P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. Dans *ACM Computing Surveys (17)5*, pages 472–522, 1985.

[8] B. Carré. The Point of View Notion for Multiple Inheritance. Dans *Proceedings of the OOPSLA/ECOOP Conference, Ottawa, Canada, ACM SIGPLAN Notices (25)10*, pages 312–321, 1990.

[9] B. Carré, L. Dekker et J.M. Geib Multiple and Evolutive Representation in the ROME Language. Dans *Proceedings of TOOLS2, Paris*, pages 101–109, 1990.

[10] C. Dony, J. Malenfant, et P. Cointe. Prototype-Based Languages : From a New Taxonomy to Constructive Proposals and Their Validation. Dans *Proceedings of*

the 7th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '92), Vancouver, British Columbia, ACM SIGPLAN Notices (27)10, pages 201–217, 1992.

[11] R. Ducournau. Y3/YAFOOL*: Le langage à objets*. Sema Group 1989.

[12] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, et A. Napoli. Le point sur l'héritage multiple. Dans *Techniques et sciences informatiques (14)3*, pages 309–345, 1995.

[13] J. Ferber et P. Volle. Using Coreference in Object Oriented Representations. Dans *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 238–240, 1988.

[14] J. Ferber. *Objets et agents : une étude des structures de représentation et de communications en Intelligence Artificielle*. Thèse d'informatique, Université Pierre et Marie Curie, Paris 6, 1989.

[15] W. Harrison et H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). Dans *Proceedings of the 8th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93), Washington, DC, USA, ACM SIGPLAN Notices (28)10*, pages 411–428, 1993.

[16] W.R. LaLonde, D. Thomas, et J.R. Pugh. An Exemplar Based Smalltalk. Dans *Proceedings of the 1st Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86), Portland, Orergon, ACM Sigplan Notices (21)11*, pages 322–330, 1986.

[17] W.R. LaLonde. Designing Families of Data Types Using Exemplars. Dans *ACM TOPLAS (11)2*, pages 212–248, 1989.

[18] H. Lieberman. *A preview of Act1*. AI memo No 625, Massachusetts Institute of Technology, 1981.

[19] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. Dans *Proceedings of the 1st Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86), Portland, Oregon, ACM SIGPLAN Notices, (21)11*, pages 214–223, 1986.

[20] J. Malenfant. On the Semantic Diversity of Delegation-Based Programming Languages. Dans *Proceedings of 10th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95), Austin, TX, USA, ACM SIGPLAN Notices (30)10*, pages 215–230, 1995.

[21] J. Malenfant. *Split Objects: Taming Value Sharing in Object-Oriented Languages*. Rapport de recherche IRO-968, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, 1995.

[22] O. Mario. *TROPES*. Thèse d'informatique, Université Joseph Fourier, Grenoble 1, 1993.

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

[24] H. Ossher, M. Kaplan, W. Harrison, A. Katz et V. Kruskal. Subject-Oriented Composition Rules. Dans *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95), Austin, TX, USA, ACM SIGPLAN Notices (30)10*, pages 235–250, 1995.

[25] F. Rechenmann, O. Mario et P. Uvietta. Multiples Perspectives and Classification mechanism in Object Representation. Dans *Proceedings of the 10th European Conference on Artificial Intelligence, Stockholm*, pages 425–430, 1990.

[26] R.B. Smith et D. Ungar. A Simple and Unifying Approach to Subjective Objects. À paraître dans *TAPOS special issue on Subjectivity in Object-Oriented Systems (2)3*, 1996.

[27] L.A. Stein, H. Lieberman, et D. Ungar. A Shared View of Sharing: The Treaty of Orlando. Dans *Object-Oriented Concepts, Applications and Databases, W. Kim et F. Lochovosky eds., Addison-Wesley*, 1988.

[28] P. Steyaert. et W. De Meuter. A Marriage of Class- and Object-Based Inheritance Without Unwanted Children. Dans *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95), Aarhus, Denmark, W. Olthoff ed., LNCS 952, Springer-Verlag*, pages 127–144, 1995.

[29] A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming*. Thèse d'informatique, Université de Jyväskylä, No 23, Finlande, 1993.

[30] D. Ungar et R.B. Smith. SELF: The Power of Simplicity. Dans *Proceedings of 2nd Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87), Orlando, FL, ACM SIGPLAN Notices (22)12*, pages 227–242, 1987.

[31] P. Wegner. Dimensions of Object-Oriented Language Design. Dans *Proceedings of 2nd Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87), Orlando, FL, ACM SIGPLAN Notices (22)12*, pages 168–182, 1987.