

UNIVERSITÉ DE MONTPELLIER II
— SCIENCES ET TECHNIQUES DU LANGUEDOC —

Spécification et implantation d'un modèle
d'objets avec points de vues dans un langage
à classes

Application à Smalltalk

Éric BURGHARD

Rapport de Stage de DEA informatique

21 janvier 2000

Remerciements

Je tiens à remercier Christophe DONY et Daniel BARDOU qui ont été tous deux, mes encadrants pendant ces longs mois de stage.

Je les remercie pour les nombreuses critiques fort *utiles* et *instructives*, concernant la présentation de mon travail, qui m'ont permis de rendre un rapport plus *clair* et plus *accessible*, et plus particulièrement Christophe DONY pour tout le temps qu'il m'a accordé, parfois même sur son temps libre.

Table des matières

1	Introduction	5
2	Analyse: Objets et points de vue	7
2.1	Contexte	7
2.2	Rappels sur le modèle à classes	8
2.3	Rappels sur le modèle à prototypes	9
2.3.1	Mécanismes de partage	9
2.3.2	Reproches à l'encontre des langages à prototypes	10
2.4	Objets et points de vue	11
2.4.1	Le concept de vue	11
2.4.2	Intérêt des points de vue en programmation par objets	12
2.4.3	Représentation de points de vue dans un modèle à prototypes	13
2.4.3.1	Représentation éclatée	13
2.4.4	Représentation des points de vue dans un modèle à classes	14
2.4.4.1	Systèmes n'intégrant pas la notion de point de vue	14
2.4.4.2	Systèmes intégrant la notion de point de vue	14
2.5	Les objets morcelés	15
3	Implantation des objets morcelés dans un modèle à classes	17
3.1	Classes d'objets morcelés	17
3.1.1	Morceaux et modèle à classes	18
3.1.2	Classes d'objets morcelés et héritage	19
3.1.2.1	Classe à morceaux partagés	19
3.1.2.2	Classe à morceaux privés	20
3.1.2.3	Comparaison des deux propositions	21
3.1.3	Envoi de message avec point de vue	21
3.1.3.1	Stratégies d'envoi de message avec point de vue	22
3.1.3.2	Ambiguïtés sur l'envoi de message avec point de vue	23
3.1.4	Point de vue et variables de morceau	24
3.1.4.1	Accès implicite	24
3.1.4.2	Accès explicite	25
3.1.4.3	Problèmes engendrés par la première stratégie	25
3.1.4.4	Avantages de la seconde stratégie	26
3.2	Implantation des objets morcelés et choix du système	27
3.2.1	Systèmes à création explicite de métaclasses	29
3.2.1.1	Organisation type des classes du noyau	29
3.2.1.2	Implantation des classes d'objets morcelés	30
3.2.1.3	Modèle réflexif d'objets morcelés	30

3.2.2	Systèmes à création implicite de métaclasses	31
3.2.2.1	Organisation type des classes du noyau	31
3.2.2.2	Implantation des classes d'objets morcelés	33
3.2.2.3	Solution empêchant l'héritage avec une classe normale	33
3.2.2.4	Solution autorisant l'héritage avec une classe normale	33
3.3	Exemple pratique	35
4	Implémentation dans un système Smalltalk	39
4.1	Efficacité de l'implémentation	39
4.2	Composantes d'un système Smalltalk	40
4.3	Squeak	40
4.4	Analyse des besoins	41
4.4.1	Factorisation de comportement	41
4.5	Nouvelles primitives	42
4.5.1	Instanciation d'une classe d'objets morcelés	42
4.5.1.1	Structure des objets dans Squeak	42
4.5.1.2	new et l'instanciation	44
4.5.2	L'envoi de message avec point de vue	44
4.5.2.1	Point de vue implicite	44
4.5.2.2	Point de vue explicite	45
4.5.2.3	Analyse des besoins	45
4.6	Modification de la machine virtuelle	45
4.6.1	Accès aux variables d'instances	46
4.6.2	Envoi de message avec point de vue	46
4.6.3	Les bytecode dédiés à l'envoi de message	46
4.6.4	Les nouveaux bytecode	48
4.7	Modification du compilateur et de l'interpréteur	48
4.7.1	La chaîne de compilation	49
4.7.2	Le problème de l'accès aux variables d'instances	50
4.7.3	Syntaxe associée aux envois de messages	51
5	Conclusion	53
A	Objets: Composition inter-niveaux	55
A.1	Compatibilité ascendante	56
A.2	Compatibilité descendante	56

1

Introduction

L'utilisation du *modèle à objets* a sans conteste apporté de nombreuses qualités aux programmes informatiques: organisation modulaire, compréhension accrue, programmes modifiables et facilement extensibles découlent des propriétés remarquables de cette approche:

- *unicité* entre objets conceptuels et objets informatiques,
- *abstraction* et *catégorisation*,
- *hiérarchisation* des concepts.

C'est la recherche de nouveaux mécanismes d'*abstraction* et de *modularisation*, à la base de la propriété de *réutilisabilité* des programmes écrits dans un langage à objets, ainsi que l'émergence de nouveaux *besoins* en bases de données ou bases de connaissances, qui ont permis d'aboutir à la notion de point de vue sur des objets, de manière à pouvoir leur associer des propriétés différentes suivant un certain contexte d'utilisation. Exemple: un objet *personne* peut répondre différemment au message *numéro_de_téléphone*, suivant qu'on le considère comme un *employé* ou comme un *étudiant*.

Cette notion se retrouve sous diverses formes dans plusieurs domaines de la recherche informatique:

- en *représentation de connaissance*, nous avons des systèmes comme LOOPS [BS83], TROPES [INR95], ...
- en *conception*, nous avons les *modèles de rôles* [AR92], la *programmation par sujets* [HO93], ...
- en *base de données*: O2Views [Pro95], COCOON [SLR⁺92], MultiView [Run94], pour n'en citer que quelque uns;
- en *programmation par objets*: ROME [CDG90], ...

[Bar98, DB98] proposent une nouvelle approche dans le domaine de la programmation par objets. Ils ont en effet montré que la notion de point de vue trouvait un sens parmi les interprétations que l'on pouvait donner aux *mécanismes de partage* dans les *langages à prototypes*, alors qu'elle n'existe pas en tant que telle dans les

langages à classes de base¹. C'est des suites de cette constatation qu'a été défini un nouveau concept d'objet, capable de représenter et de manipuler des entités sous de multiples point de vue: l'*objet morcelé*.

D. BARDOU proposait à la fin de sa thèse de doctorat [Bar98], plusieurs ébauches d'implémentation d'objets morcelés dans un langage à classes. Ce sont ces solutions qui ont constitué le point départ du travail d'étude et d'implémentation réalisé au cours de ce stage de DEA.

Dans une première partie, consacrée à l'étude des points de vue dans les systèmes objets, nous présentons quelques rappels sur les différents modèles qui sont à la base de la caractérisation de la solution des objets morcelés et nous montrons d'où viennent leurs aptitudes à représenter effectivement des entités en fonction de points de vue.

Dans une deuxième partie, nous confrontons plusieurs possibilités d'implantation des objets morcelés dans un système à classes dynamiquement typé, de manière à orienter notre stratégie d'implantation dans la bonne direction.

Dans la troisième partie, nous abordons les modifications a apporter à un système Smalltalk, et plus spécifiquement à Squeak, qui est un système Smalltalk domaine public, de manière a y implémenter les objets morcelés suivant la solution retenue précédemment. Nous y abordons notamment les modifications a apporter à son noyau, et à sa machine virtuelle pour laquelle on défini un nouveau jeu d'instructions et de nouvelles primitives.

¹Ceux qui n'implémentent pas spécifiquement la notion de point de vue: soit parce que ce concept n'a pas été réifié, soit parce qu'aucun mécanisme ne permet de l'exprimer au sein du langage.

2

Analyse: Objets et points de vue

On s'intéresse depuis un certain temps déjà, au « *mariage* », des objets et des points de vue, comme en témoignent les diverses propositions émanant de domaines aussi variés que la programmation, la représentation de connaissances, ou les bases de données.

Pour permettre d'appréhender correctement le problème, un rappel sur deux grandes familles de langages à objet, est d'abord proposé. L'examen des principaux systèmes intégrant la notion de point de vue, servira à mieux souligner l'originalité de l'approche du problème par les *objets morcelés*.

2.1 Contexte

Une application informatique se conçoit, dans sa forme la plus traditionnelle, en termes de *données*, manipulées par différentes procédures, et de *résultats*, produits par celles-ci. La méthodologie objet, quant à elle, privilégie plutôt une approche *ontologique* de la réalité, dans laquelle des objets, décrits en *hiérarchie*, répondent à certains comportements et disposent d'une certaine mémoire (ou état). Les résultats émergent, de cette manière, des interactions entre les objets qui composent l'application, alors que les données sont *encapsulées* par les objets au travers de leur état, et manipulées de façon transparente, par l'activation de certains comportements associés aux dits objets.

Les langages à classes et les langages à prototypes représentent deux façons différentes d'organiser les objets du domaine d'application:

- dans les *langages à classes*, un concept est représenté par une classe, et ses représentants sont les instances de cette classe. La classe factorise les comportements et décrit la structure de ses instances (voir 2.2), et empêche de ce fait l'évolution individuelle d'une instance sur ces deux domaines. La modification d'une classe entraîne la modification de tous ses instances. L'avantage d'une telle organisation est que toutes les instances sont immédiatement caractérisées comme représentantes d'un concept, tout au long de l'exécution;
- dans les *langages à prototypes*, on abandonne la classe, qui représentait un concept, pour ne raisonner que sur les exemplaires de ce concept, qui du même coup n'est plus aussi formellement identifié. Cependant la démarche qui consiste à raisonner sur les exemples pour en identifier l'abstraction, semble être plus

proche du raisonnement humain, et rend les langages à prototypes plus adéquats aux systèmes à base de connaissances, par exemple. Un prototype décrit son propre état et son propre comportement, qu'il peut tout de même partager avec d'autres par divers mécanismes (voir 2.3). Se pose alors le problème de la nature de ce partage [Bar98], qui induit un problème d'identité des objets: la modification de la valeur d'un *slot*¹ d'un objet, doit-il toucher également ceux avec qui il les partage ?

De plus, en l'absence de classes, pour pouvoir regrouper des objets en tant qu'exemples d'un même concept, on doit raisonner en terme d'ensemble de prototypes admettant des propriétés communes. Cette recherche, bien que facilitée par l'existence de *lien de partage* de propriétés entre les prototypes, n'est plus aussi évidente que dans le cas des modèle à classes.

2.2 Rappels sur le modèle à classes

On se place, par la suite, dans le cadre du modèle à classes de Smalltalk [GR83], et même si à juste titre, on le considère comme le modèle de « base », certaines propriétés énoncées peuvent ne pas faire figure de vérité pour d'autre systèmes. [Per98] propose une bonne introduction aux langages à classes.

On distingue en général deux entités, les objets et les classes reliées entre elles par un lien, dit d'*instanciation*:

- un *objet* est le représentant d'un concept dont la structure et le comportement sont entièrement décrits par sa classe. Un objet n'a en général qu'une seule classe. Il est créé par *instanciation* de sa classe;
- une *classe* décrit donc la structure de ses instances, et factorise les comportements qui leur sont associés. Une classe peut inclure la structure et les comportements associés aux instances d'une ou de plusieurs autres classes, grâce à un mécanisme que l'on appelle *héritage*. On appelle sa super-classe, la classe dont elle hérite.

L'activation d'un comportement d'un objet est réalisé au moyen d'un *envoi de message* qui s'applique sur cet objet, et que l'on désigne par *receveur* du message; cet envoi de message, identifié par un *sélecteur*, donne lieu à une *recherche*² pour trouver le code de la méthode associée, suivant le(s) lien(s) d'héritage, et en partant de la classe d'instanciation de l'objet.

Dans les systèmes à classes réflexifs, une classe peut également être considéré comme un objet, instance d'une classe que l'on appelle pour la distinguer *méta-classe*. Cette métaclass, à son tour, peut être considérée comme un objet, instance d'une *méta-méta-classe*. Pour stopper la régression infinie liée à ce type raisonnement, on s'arrange pour placer une *boucle d'instanciation* qui spécifie, par exemple, qu'une métaclass est instance d'elle-même (voir 3.2.1.1).

¹une méthode ou une variable.

²ou *lookup*.

2.3 Rappels sur le modèle à prototypes

La catégorisation « langages à prototypes » a été inventée bien après que les premiers systèmes aient vu le jour, et il est difficile de dégager un ensemble minimal de propriétés les représentant tous, sans tomber dans une simplification maladroite et inadéquate pour cet exposé. C'est pourquoi, on se place par la suite dans le cadre du modèle de SELF [US91].

L'approche par *prototypes* se veut une simplification à l'extrême du modèle à classes. On supprime les classes, pour ne disposer que d'objets que l'on appelle *prototypes*³.

En l'absence de classe, un objet doit détenir sa propre description; un prototype est défini par un certain de nombre de slots:

- des slots *valeurs*, qui contiennent l'état du prototype; ils correspondent aux variables d'instances du modèle à classes;
- des slots *fonctions*, correspondant aux méthodes, dans ce même modèle;
- des slots *parents*, qui correspondent à des liens vers des prototypes utilisés comme complément de description⁴.

2.3.1 Mécanismes de partage

Deux mécanismes permettent la réutilisation de l'état et des comportements entre prototypes:

- le *clonage*, spécifie une source et une cible. Tous les slots de la source sont recopiés dans la cible. La cible est donc un nouveau prototype, qui dispose, au moment de sa création, du même état et des mêmes comportements que la source. Ce qui peut éventuellement changer par la suite, au cours des modifications d'un des deux prototypes. On parle de *partage ponctuel*;
- la *copie différentielle* permet au prototype copié, de n'exprimer que ses différences par rapport à un deuxième prototype avec lequel il partage les autres slots. C'est un mécanisme très proche de celui de l'*héritage* des langages à classes. Ce lien est représenté au niveau du prototype par l'existence d'un slot parent⁵. Un mécanisme dit de *délégation* résout l'accès aux slots partagés. On parle de *partage persistant*.

Si la modification d'un slot du père, au niveau de ce dernier, affecte également tous les fils qui n'ont pas redéfini l'attribut (voir figure 2.1), la demande de modification d'un slot du père, au niveau d'un fils peut elle, s'interpréter de deux façons:

- soit, elle entraîne la modification de la valeur du slot du père. Cette interprétation peut vite s'avérer difficile à contrôler, car les objets peuvent se trouver

³Historiquement, la démarche qui a conduit à la notion de prototypes, telle qu'elle est employée ici, fut très différente. Cette notion a été le point de convergence de nombreux travaux dans différents domaines de recherche comme la psychologie, ou les systèmes à base de connaissances.

⁴Lorsque le langage implémente la délégation.

⁵Analogie avec le lien *superclass* dans le modèle à classes.

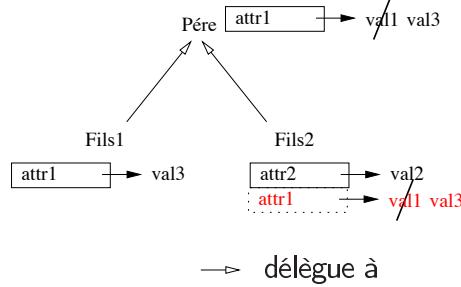


FIG. 2.1: La modification de la valeur d'un slot du père affecte tous les fils qui ne l'ont pas redéfini.

modifiés par inadvertance (voir figure 2.2(a)). Le partage induit est un *partage de propriétés* [Bar98]; cela revient à considérer qu'il n'y a qu'une seule entité du domaine d'application, représentée par plusieurs prototypes. On parle alors de représentation éclatée (voir 2.4.3.1);

- soit, elle entraîne la redéfinition du slot au niveau du fils (voir figure 2.2(b)), de manière à ce que le fils, et les petits-fils s'il y en a, en soient affectés. Le partage induit est un *partage de valeurs* [Bar98].



- (a) La demande de modification de la valeur de l'attribut **attr1** au niveau du fils, répercute la modification au niveau du père.
 (b) La demande de modification de la valeur de l'attribut **attr1** au niveau du fils, entraîne la redéfinition de **attr1** dans ce dernier.

FIG. 2.2: Deux interprétations possibles de l'affectation dans un langage à prototypes implémentant la délégation.

2.3.2 Reproches à l'encontre des langages à prototypes

Les problèmes engendrés par l'utilisation de langages à prototypes sont de deux sortes [DB98]:

- l'identité des objets, dans le cas où le langage intègre la délégation, devient une notion floue à cause d'*effets de bords* indésirables pouvant être provoqués par des affectations au niveau des slots d'un prototype;
- l'organisation des programmes devient difficile à appréhender à cause de l'absence de concepts clairement définis, c'est à dire définis en intention.

2.4 Objets et points de vue

Le concept de vue est utilisé dans de nombreux domaines de l'informatique, et c'est son utilisation dans les domaines qui sont en rapport direct avec le monde des objets qui nous intéressent plus particulièrement.

Dans les bases de données, les vues servent à masquer certaines relations du *schéma*, ou plus généralement, à l'adapter pour le faire correspondre à ses besoins [BD98]. Concernant les systèmes à objets, un point de vue est une vision particulière que l'on peut avoir sur un objet.

La caractérisation de ce qu'est une vue, nous permettra de mettre en évidence les mécanismes fondamentaux qui sont derrière leur fonctionnement, dans une optique d'implantation au sein d'un système objet, bien entendu.

2.4.1 Le concept de vue

Dans un sens très général, une *vue* est une abstraction qui désigne ce que l'on observe au travers d'une fenêtre ouverte sur le monde. Si l'on néglige le fait qu'avancer ou reculer par rapport à la fenêtre, agrandi ou rétréci notre champ de vision, on peut dire que le cadre de la fenêtre détermine le contenu de la vision partielle, que l'on a sur le monde. Une fenêtre peut même servir à déformer la réalité, si l'on imagine une fenêtre avec des vitres teintées ou déformantes, ou bien faire croire en une seconde réalité: est-ce un bateau au loin, ou une trace sur la vitre ?

En résumé de ce que l'on attend d'une fenêtre (qui fait la vue):

1. Qu'elle puisse *occulter* les aspects de la réalité que l'on ne souhaite pas voir, parce qu'on aime que ce qui est beau.
2. Qu'elle puisse *déformer* la réalité, pour qu'elle adopte une forme que l'on préfère, parce qu'on n'aime pas être choqué.
3. Qu'elle puisse *faire apparaître* une réalité qui n'existe pas vraiment; parce qu'on aime croire aux mirages.

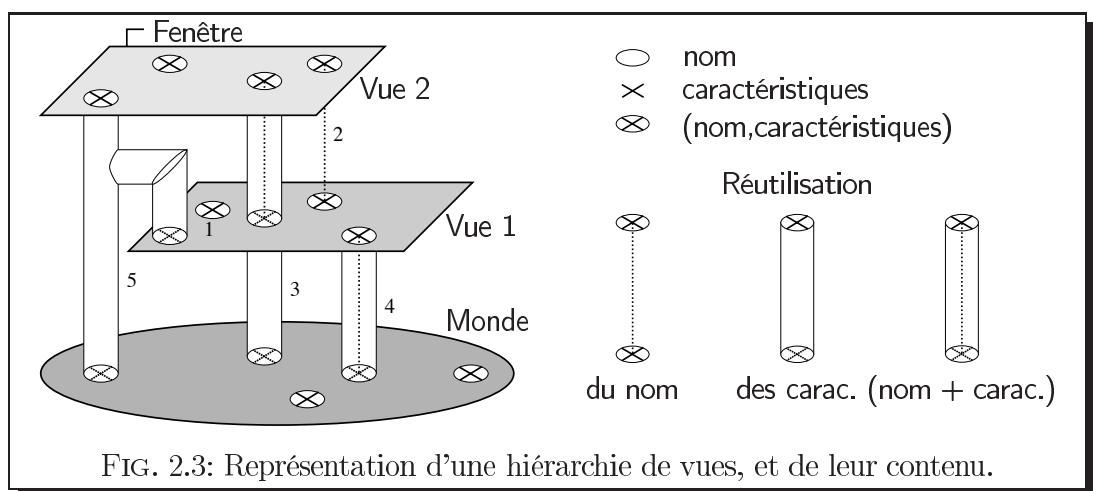


FIG. 2.3: Représentation d'une hiérarchie de vues, et de leur contenu.

La FIG. 2.3 représente tout ceci, schématiquement.

- Les *vues* sont constituées par tout ce que l'on peut y observer. Le « *monde* » est la vue initiale.
- Les « *chooses* » observables, peuvent être totalement identifiées par un nom, et ensemble de caractéristiques, dans le contexte d'une vue.

Le contenu d'une vue peut reprendre celui d'une autre vue. Une réutilisation engendre une dépendance de la vue qui réutilise, vis-à-vis de la vue qui définit. Sur la FIG. 2.3, ces formes de réutilisation sont numérotées, et nous tentons de mieux les expliquer à présent:

1. Pas de dépendance; Une nouvelle chose apparaît dans la Vue 1, et n'est visible, compte tenu de ce qui est représenté, que dans la Vue 1. C'est le principe de *génération spontanée*.
2. On observe deux choses qui sont identifiées par le même nom dans la Vue 1 et dans la Vue 2, mais qui peuvent avoir des caractéristiques totalement différentes. Ce procédé sert à *masquer* une réalité.
3. Quelque chose, dans la *Vue 1*, ressemble⁶ à ce que l'on peut voir sur le *Monde*, mais n'est pas identifiée sous le même nom. Ce procédé sert à *reformuler* une réalité.
4. Quelque chose observable sur le *Monde* est observable sous le même nom, et avec des caractéristiques similaires. Si les caractéristiques sont identiques, il s'agit du *principe de transparence*⁷, sinon ce procédé permet de *déformer* une réalité.
5. Cette forme de partage est une généralisation de celle décrite en 3. On *compose* une nouvelle chose à partir de deux autres choses.

2.4.2 Intérêt des points de vue en programmation par objets

Dans le monde des objets, on peut vouloir cacher ou exhiber certaines propriétés d'un objet suivant un contexte d'utilisation particulier. Ces contextes seraient autant de perceptions différentes que l'on pourrait avoir d'un même objet. Le Monde du §2.4.1 est assimilé à l'ensemble des propriétés les plus générales, ou les plus courantes d'un objet, suivant la politique choisie (on parle aussi de référentiel [Van94]), et un point de vue contient alors un ensemble de propriétés utiles à un certain contexte d'utilisation, qu'il partage éventuellement avec d'autres points de vue.

Pour reprendre notre exemple, choisissons une personne qui, au cours de ses activités journalières, est amenée à se comporter comme un étudiant ou un employé. Il est raisonnable de penser qu'elle ne répondra pas de la même façon à la question « *donne moi ton numéro de téléphone* » suivant le contexte dans lequel elle se trouve. Il est cependant nécessaire qu'elle se « *souvienne* » de tous ses numéros au cas où quelqu'un de sa fac demande explicitement son numéro en tant qu'employé.

Même si l'on peut toujours lever l'ambiguïté en donnant des noms différents aux attributs, `num_tel_lab`, `num_tel_bureau`, et en en faisant une demande explicite,

⁶Le terme *ressemble* n'est pas utilisé innocemment ici, puisqu'on ne sait pas dans quelles mesures les caractéristiques de la chose initiale sont reflétées.

⁷Si l'on voit la réutilisation des caractéristiques comme une fonction, il s'agit dans ce cas de l'identité.

il est plus naturel de considérer la valeur de ces attributs comme étant deux valeurs, ou *facettes*, différentes d'une même abstraction: un numéro de téléphone. Le point de vue permet, dans une certaine mesure, une économie de vocabulaire, en émettant un *sous-entendu* qui permet, le cas échéant, de lever l'ambiguïté quant à la réponse à apporter.

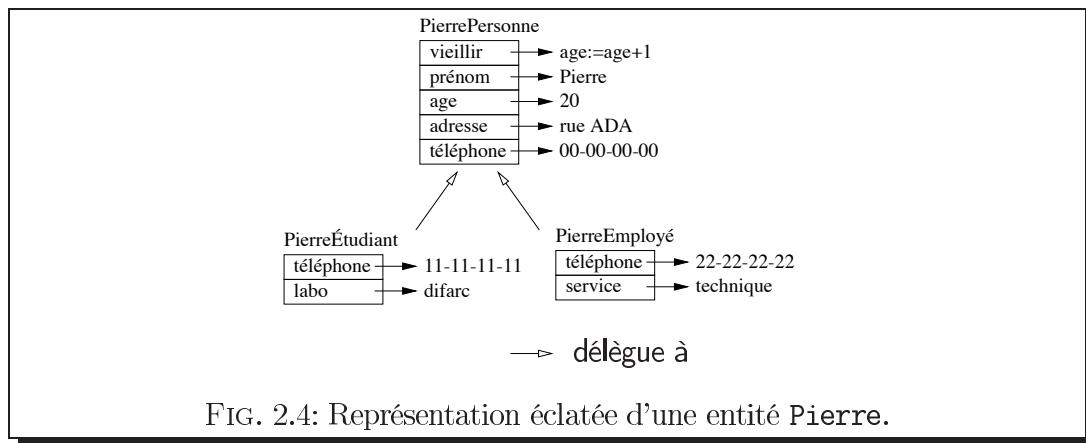
2.4.3 Représentation de points de vue dans un modèle à prototypes

Le §2.3.1 a montré différentes interprétations possibles du partage réalisé par le mécanisme de délégation. Le §2.4.1 a montré que l'essentiel des mécanismes du fonctionnement des vues étaient dans la notion de réutilisation, ou partage si on préfère.

[DB98, Bar98] ont mis en évidence les potentialités des prototypes à représenter des points de vue, dans le cas où le système implémentait la délégation avec partage de propriétés. On parle alors de *représentation éclatée* d'un entité.

2.4.3.1 Représentation éclatée

La FIG. 2.4 montre un exemple d'une telle représentation, dans laquelle une entité **Pierre**, que l'on ne voit pas, est représenté sous trois points de vue qui sont représentés directement par des prototypes. Le point de vue **PierrePersonne** partage ses propriétés avec **PierreÉtudiant** et **PierreEmployé**.



Cette interprétation du mécanisme de délégation associée à un partage de propriété pose cependant les problèmes suivants:

- **Pierre** n'est pas réifié. On ne peut lui envoyer un message;
- rien n'indique que **PierreÉtudiant**, est un point de vue. Une application qui mélangerait des prototypes *normaux* et des prototypes *point de vue* serait difficile à organiser, ne serait ce que pour le clonage: on aurait besoin de savoir ce que l'on doit cloner (un prototype seul, ou un ensemble de prototypes point de vue);
- rien n'indique que **PierrePersonne**, **PierreÉtudiant**, et **PierreEmployé**, sont points de vue d'une même entité. Autrement dit: rien ne dénote l'ensemble des points de vues représentants l'entité du domaine.

2.4.4 Représentation des points de vue dans un modèle à classes

Certains modèles intègrent la notion de point de vue, même si l'on peut tout de même utiliser les mécanismes de base des langages à classe pour tenter représenter des points de vue.

2.4.4.1 Systèmes n'intégrant pas la notion de point de vue

Les langages à classes standards, tel que Smalltalk par exemple, ne proposent que du partage de comportement aux instances d'une même classe: chaque instance dispose de son propre état qu'elle ne partage avec nulle autre. Le partage de propriété permet aux prototypes de représenter naturellement les points de vue (voir 2.4.3). Pour simuler un partage de propriété, dans un langage à classe, on dispose des solutions suivantes [DB98]:

- par *composition*; les classes `Étudiant` et `Employé` qui sont indépendantes, déclarent une variable d'instance de type `Personne`. Il faut cependant redéfinir chaque méthode de `Personne` dans ces dernières, pour qu'elles réalisent l'indirection vers l'instance de `Personne`.

On se rend cependant compte que c'est là une mauvaise utilisation de la composition, car peut-on dire qu'un étudiant est composé d'une personne ? De plus l'entité se trouve sous plusieurs identités (instances) différentes au sein du système, pouvant répondre indépendamment les unes des autres aux messages qui leur sont envoyés;

- à l'aide de *l'héritage multiple*, avec tous les problèmes que cela implique pour les applications un tant soit peu complexes. On crée une classe `Personne` qui hérite de `Étudiant` et de `Employé`.

Cependant, même si les instances résultantes héritent bien de la description associée aux « *pseudo points de vue* » représentés par les super-classes, il n'est pas possible d'en identifier clairement les différentes composantes;

- en utilisant des *variables de classes*⁸ au niveau de `Personne`, on peut faire partager des valeurs d'attributs par des instances des classes `Étudiant` et `Employé` qui héritaient de `Personne`.

Malheureusement, toutes les instances de ces trois classes partageraient les mêmes valeurs.

2.4.4.2 Systèmes intégrant la notion de point de vue

Les solutions présentées au §2.4.4.1 n'étant pas satisfaisantes, plusieurs systèmes qui intègrent la notion de point de vue ont été développés. Soit le concept même de point de vue a été réifiée, soit le système intègre de nouveaux mécanismes, permettant de les représenter plus facilement. On peut en citer quelques-uns:

⁸On entend par variables de classes, celles au sens de Smalltalk qui sont plus des dictionnaires d'associations (`nom`, `valeur`), partagés le long d'une même chaîne d'héritage, que de véritables variables de classes, définies au niveau de la métaclassse.

LOOPS [BS83] est un langage hybride qui intègre la notion d'*objet composite*, décrit par une classe. Les parties d'un objet composite sont elles même des objets. Une classe d'objets composites s'instancie récursivement de manière à instancier chacune de ses parties, et se laisse généraliser/spécialiser par héritage.

Ces objets composites permettent de représenter des objets avec points de vue, que l'on appelle *perspectives*. La solution de LOOPS ressemble à celle de la *composition* décrite dans le §2.4.4.1, mais est intégrée au sein du langage.

ROME [CDG90] est un langage qui intègre la notion de représentation multiple; un objet peut être lié à *plusieurs classes de représentation* qui peuvent contenir chacunes, des descriptions différentes de l'entité. Ces classes de représentations sont sous-classes de l'unique *classe d'instanciation* de l'objet, qui se trouve être sa *classe de représentation la plus générale*. Cette hiérarchie de classe est prise en compte lors de l'envoi de message.

Par exemple **Personne** serait la classe d'instanciation de pierre. Elle serait super-classe de **PersonneEmployé** et **PersonneÉtudiant**, toutes deux, classes de représentation.

Les inconvénients de tels systèmes est que leur capacité à représenter les points de vue est basé sur des mécanismes qui ne sont pas courants dans les langages à classes standard; la notion de point de vue résulte d'une utilisation particulière de ces mécanismes: on pense notamment à la multi-instanciation de ROME.

2.5 Les objets morcelés

Rappelons que ni les systèmes à classes, ni les systèmes à prototypes ne paraissent adaptés au concept de point de vue même si quelque uns disposent de facilités pour les représenter. Les *objets morcelés* constituent une approche spécifique au problème de la représentation des points de vue dans les systèmes objets, et leur implantation au sein d'un système objet ne fait appel qu'aux mécanismes standards qui ont fait l'objet des rappels du §2.2 et du §2.3.

Les *objets morcelés*, présentés dans [Bar98], fonctionnent suivant le même principe que les *représentations éclatées*, c'est à dire qu'ils sont eux aussi basés sur le partage de propriétés induit par le mécanisme de délégation. Quelques améliorations permettent cependant d'en outrepasser les limitations:

1. Un objet morcelé détient la liste de ses points de vue. On appelle *morceau*, l'entité qui représente un point de vue de l'objet morcelé, et les morceaux d'un objet morcelé sont organisés en une hiérarchie simple de délégation. Un point de vue n'est pas exclusivement représenté par un morceau unique, mais peut également prendre la forme d'un ensemble de morceau: si on considère un objet morcelé dans sa globalité, le point de vue correspondant est constitué par l'ensemble de ses morceaux.
2. Un *morceau* ne peut répondre directement à un message que si la demande lui parvient de l'objet qui le détient. On peut cependant lui envoyer des messages d'*introspection*.

On vérifie facilement que ces deux principes permettent d'éviter les problèmes de la représentation éclatée énumérés au §2.4.3.1.

Ce type d'objet pourrait s'intégrer aussi bien au niveau d'un langage à prototypes qu'au niveau d'un langage à classes auquel on aurait rajouté un mécanisme de partage de propriétés par délégation. Dans les deux cas, aucune implémentation n'a jamais été réalisée. Notre choix s'est cependant porté sur les langages à classes, qui permettent une meilleure organisation des entités du domaine des applications (voir 2.3.2), et parce que l'on se place plus dans un contexte de *programmation par objet* que dans celui de *représentation de connaissances* pour lequel le modèle à prototypes aurait été plus adéquat.

3

Implantation des objets morcelés dans un modèle à classes

Implanter des objets morcelés dans un modèle à classes peut se faire de diverses façons. Ce chapitre présente une étude qui confronte plusieurs de ces possibilités et qui valide, en soulignant les avantages, celles qui nous ont semblé les plus adéquates pour une implantation.

Le cadre de cette étude est plus précisément celui des langages à classes réifiées, car ils disposent de facilités intrinsèques d'extension, bien que ce qui est dit par la suite peut tout aussi bien s'appliquer aux langages à classes non réifiées.

3.1 Classes d'objets morcelés

La classe d'un objet détient la description et les comportements associés à cet objet. La classe d'un objet morcelé détient donc la structure et les comportements de l'objet lui-même ainsi que de chaque morceaux qui le constituent.

Pour en revenir à notre exemple, on peut faire de PersonnePDV une classe d'objets morcelés dans laquelle `Étudiant` et `Employé` sont sous-morceaux de `Personne`. Ces morceaux représenteraient trois points de vue différents¹ d'une personne.

La structure d'un objet morcelé est constituée:

- par les structures associées à tous les morceaux organisés en hiérarchie. Ces structures détiennent les valeur associées aux variables qui y sont déclarées, voir redéclarées.
- par le *corps* de l'objet morcelé, qui est la structure censée contenir les valeurs des variables déclarées en dehors de tout morceaux. Ces variables ne peuvent être redéclarées sauf au sein des morceaux, bien entendu.

[Bar98] propose deux solutions d'implantation des classes d'objets morcelés²:

Une classe à morceaux partagés est décrite par un certain nombre de morceaux extérieurs à la classe et partageables entre différentes classes.

¹Il y en d'autres: toute partie de l'ensemble des morceaux peut être considérée, à juste titre, comme un point de vue.

²Les noms à l'origine étaient respectivement *classes morcelées* et *classes d'objets morcelés*, qui ont été renommés pour des raisons évidentes d'ambiguïté.

Une classe à morceaux privés est décrite par un certain nombre de morceaux internes à la classe.

Mais dans un premier temps, il nous faut définir un cadre à l'utilisation des morceaux.

3.1.1 Morceaux et modèle à classes

Les morceaux, au même titre que les classes constituent des entités capables de contenir des descriptions ou des bouts de descriptions d'autres entités du domaine d'application. Définir une classe d'objets morcelés revient à donner une description de la dite classe ainsi que celles de tous ses morceaux constituants et de leur hiérarchie.

Travaillant sur un système à classes réifiées, on se doit également de réifier le concept de morceau. Cependant, de manière à restreindre les utilisations incohérentes que l'on pourrait faire des morceaux ainsi rendus manipulables sous la forme d'objets, on doit s'assurer de plusieurs choses:

1. La création, suppression, modification d'un morceau ou de la hiérarchie des morceaux, s'exprime par l'intermédiaire du protocole associé aux classes d'objets morcelés. L'exemple suivant montre une partie de ce protocole suivant la syntaxe Smalltalk (`addMrc:withSuperMrc:` correspond au nom de la première méthode qui admet 2 arguments `nomMorceau` et `pere`):

```
addMrc: nomMorceau withSuperMrc: pere
addMrc: nomMorceau withSuperMrc: pere desc: description
addMrcs: hierarchie
delMrc: nomMorceau
addMethod: methode onMrc: nomMorceau
delMethod: method onMrc: nomMorceau
addInstVar: nomvar onMrc: nomMorceau
delInstVar: nomvar onMrc: nomMorceau
```

2. L'envoi de message n'a de sens que dans le contexte d'un objet morcelé: un morceau tout seul ne peut répondre de lui-même au messages associés au point de vue qu'il est censé représenter. Il peut cependant répondre à des messages d'*introspection* ou de *manipulations bas niveau*:

```
at: index onMrc: name
at: index onMrc: name put: val
```

Un morceau doit pouvoir contenir le lien vers son super-morceau, son dictionnaire des méthodes, ainsi que son dictionnaire de noms de variables. Voici un exemple de déclaration minimale de la classe des morceaux:

```
Object
subclass: #Mrc
instanceVariablesNames: 'super methodDict instVarNames'
classVariablesNames: ''
poolDictionary: ''
category: 'PDVKernel-Objets'
```

3.1.2 Classes d'objets morcelés et héritage

Une classe d'objet morcelé peut hériter d'une classe normale. On considérera donc deux formes d'héritage:

- entre une classe d'objets morcelés et une classe normale;
- entre deux classes d'objets morcelés.

La première forme d'héritage empêche, en l'absence d'héritage multiple, de regrouper les comportements communs aux objets morcelés dans une *classe abstraite* dont toutes les classes d'objets morcelés héritaient, puisqu'une classe normale, choisie pour être super-classe d'une classe d'objets morcelés, n'en hérite pas forcément.

Une solution concrète à ce problème sera proposée au §4.4.1, mais on peut d'hors et déjà en révéler le principe: une classe particulière est utilisée comme un *patron* pour les classes morcelées qui n'en héritent pas.

La sémantique associée à l'héritage entre classes d'objets morcelés dépend du modèle choisi, et plus précisément dépend du choix que l'on fait, d'accepter ou d'interdire le partage de morceaux entre classes qui ne sont pas en relation par un lien d'héritage.

3.1.2.1 Classe à morceaux partagés

Dans ce modèle, le concept de morceau est réifié, et une classe *pointe* un certain nombre d'*objets morceaux* qui sont organisés en hiérarchie. Deux classes, sans lien particulier, peuvent faire appel aux mêmes morceaux puisqu'ils existent en tant qu'objets dans le système.

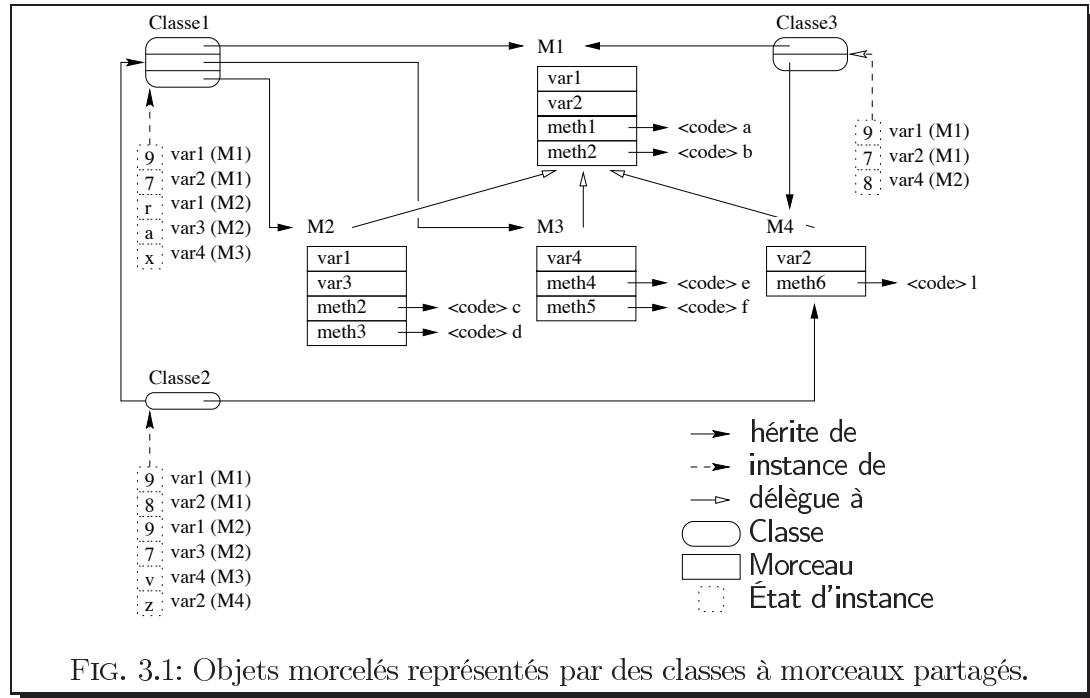


FIG. 3.1: Objets morcelés représentés par des classes à morceaux partagés.

Le concept associé à une classe d'objets morcelés se laisse généraliser/spécialiser par héritage. Une classe hérite de la description induite par la hiérarchie des morceaux. La seule opération permise lors de l'héritage, est la prise en compte, dans la sous-classe,

de nouvelles hiérarchies qui sont directement connectées à la hiérarchie héritée (voir figure 3.1).

Sur la FIG. 3.1, les instances sont représentées par des tableaux de valeurs. Ces valeurs sont associées aux variables qui sont identifiées par un nom et un morceau d'appartenance entre parenthèses.

3.1.2.2 Classe à morceaux privés

Le concept associé au morceau peut être réifié, mais ce qui importe surtout, c'est que les morceaux ne puissent pas être partagés³ entre différentes classes qui ne sont pas en relation via un lien d'héritage.

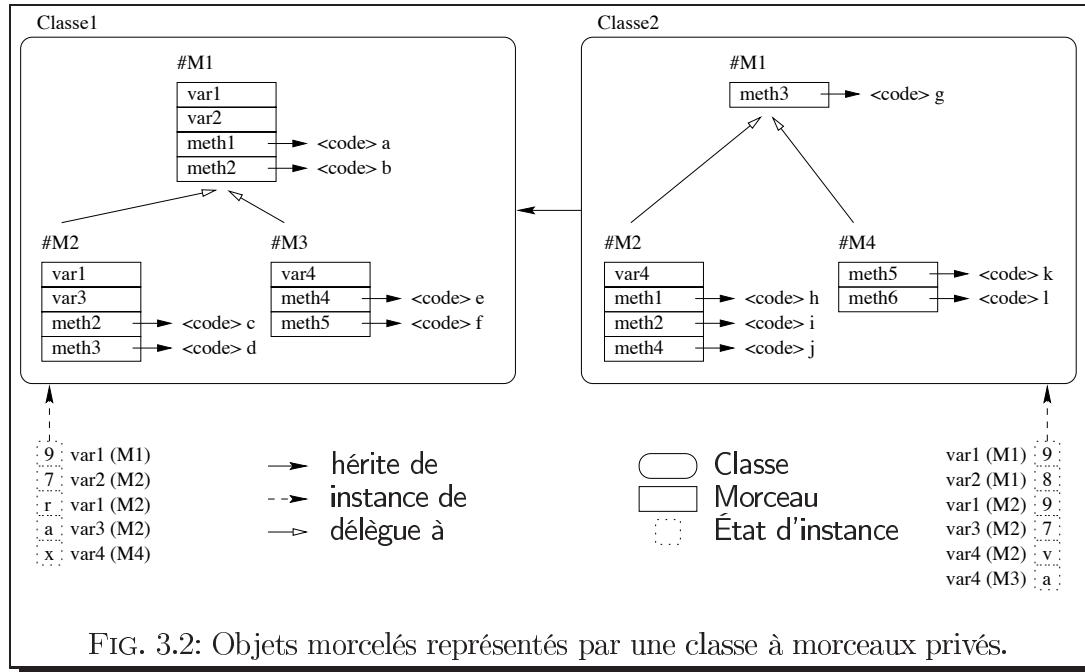


FIG. 3.2: Objets morcelés représentés par une classe à morceaux privés.

Le concept associé à une classe à morceaux privés se laisse généraliser/spécialiser par héritage, comme c'est le cas avec une classe normale. Plus précisément, on peut généraliser/spécialiser le concept représenté par la hiérarchie des points de vue (voir figure 3.2):

- en ajoutant un nouveau morceau et en précisant son morceau parent.
- en ajoutant un nouveau slot⁴ à un morceau existant, c.a.d. un morceau dont la description a été héritée; cette modification ne touche évidemment que la classe dans laquelle elle est réalisée et ses sous-classes éventuelles.
- en redéfinissant une méthode d'un morceau.

³On entend ici partage de propriété puisque le partage de valeur peut toujours être obtenu en recopiant les morceaux d'une classe à une autre.

⁴Terminologie des langages à prototypes pour exprimer indifféremment un attribut ou une méthode.

3.1.2.3 Comparaison des deux propositions

La sémantique opérationnelle est plus riche dans le cas des classes d'objets à morceaux privés du fait que les morceaux ne soient connus que de leur classe d'appartenance; on autorise de ce fait des opérations de spécialisation/généralisation sur les morceaux eux-mêmes, sans se limiter uniquement à leur hiérarchie. L'exemple de la FIG. 3.14 montrera un cas pratique qui tire partie de cette sémantique, et qui ne peut pas être adapté aux classes à morceaux partagés sans avoir à créer de nouveaux morceaux.

Dans le cas du modèle de classes à morceaux partagés, le système s'organise autour de hiérarchies de classes et de hiérarchies de morceaux. Une classe peut être représentée par un sous-arbre partiel de l'arbre d'une hiérarchie, et l'envoi de message en est rendu un peu plus complexe, sachant qu'on ne doit plus suivre *aveuglément* les liens de délégation au sein d'une hiérarchie, mais se limiter à la partie utilisée par la classe à morceaux partagés de l'objet qui a reçu le message. Les classes à morceaux partagés permettent une meilleure factorisation de description et de comportement, mais on est en droit de se demander s'il le fait d'admettre un partage de description entre deux classes, qui ne sont pas en relation via un lien d'héritage, ne favorise pas une mauvaise organisation des programmes.

Au vues de ses limitations et des interrogations concernant le bien fondé de son approche, la solution des classes à morceaux partagées a été écarté au profit de celle des classes à morceaux privés.

3.1.3 Envoi de message avec point de vue

On a dit qu'un point de vue pouvait être caractérisé par un nom de morceau unique comme par un ensemble de nom de morceaux. L'envoi de message avec point de vue sur des objets morcelés doit prendre en compte cette particularité. On distingue donc trois types d'envois de messages:

- l'envoi de message spécifiant un seul nom de morceau: c'est le *point de vue simple*;
- l'envoi de message spécifiant plusieurs noms de morceaux: c'est le *point de vue combiné*;
- l'envoi de message spécifiant tous les noms de morceaux de l'objet morcelé chargé d'y répondre: c'est le *point de vue global*, et c'est celui qui est considéré par défaut, en l'absence de précision.

Au sein du langage, on peut préciser le point de vue de quatre manières:

1. Sous la forme d'une succession de *chaînes de caractères* qui contiennent les identifiants des morceaux constituants le point de vue.
2. Sous la forme d'une succession de *noms de variables* qui contiennent des chaînes de caractères.
3. Sous la forme d'une succession entremêlée de *chaînes* et de *noms de variables*.

4. Sous la forme du nom d'une *pseudo-variable* qui n'a de valeur que dans le contexte d'une méthode de morceau, ou d'un envoi de message avec point de vue:

`methodViewPoint` dénote le même point de point de vue que celui qui est associé au contexte d'*utilisation* de cette pseudo-variable. C'est à dire, le point de vue associé au morceau qui déclare la méthode dans laquelle la pseudo-variable est utilisée. Cette variable permet d'éviter d'écrire en dur le nom du morceau, tout en se limitant au point de vue associé à ce seul morceau⁵.

`thisViewPoint` dénote le même point de vue que celui qui forme le contexte d'*évaluation* de cette pseudo-variable. L'intérêt est de pouvoir envoyer un message à un autre objet suivant le même point de vue que le point de vue courant.

`genericViewPoint` dénote le point de vue le plus spécifique parmi les plus génériques que celui qui forme le contexte d'*utilisation* de cette pseudovariable.

`globalViewPoint` dénote le point global. Cette variable peut être utilisée conjointement avec le pseudo-recepteur `self` pour régler une certaine ambiguïté sur les envois avec point de vue implicite: un envoi de message sur `self`, sans expliciter le point de vue, ne change pas le point de vue courant, alors qu'un envoi avec point de vue implicite, sur un recepteur normal, considère le point de vue global.

3.1.3.1 Stratégies d'envoi de message avec point de vue

Pour l'envoi global et combiné, une stratégie de résolution spécifique doit être mise en place lors de la recherche de la méthode à appliquer. En effet, on ne doit plus faire face à une recherche parcourant le lien unique de délégation à partir d'un seul morceau, comme c'est le cas d'un envoi avec point de vue simple, mais bien à une recherche sur une partie de l'arbre de délégation dont les feuilles sont les morceaux spécifiés. On doit également certainement s'attendre (car c'est là l'intérêt des points de vue) à des propriétés redéfinies, donc à des ambiguïtés.

[Bar98] propose deux stratégies pour trouver la bonne méthode:

La stratégie du point de vue le plus spécifique considère les morceaux, caractérisant le point de vue, ainsi que leur descendants dans la hiérarchie des morceaux de l'objet. Plusieurs cas sont à envisager.

- Le nom de méthode est introuvable. Il faut signaler l'erreur.
- Le nom de la méthode est trouvé dans un seul morceau, elle est appliquée de facto.
- Le nom de la méthode est trouvé dans plusieurs morceaux se trouvant sur la même chaîne de délégation. La méthode appliquée est celle du morceau en début de chaîne, ou le plus spécifique si l'on se place dans un contexte de généralisation/specialisation.

⁵On évite ainsi toutes les erreurs de programmation qui résultent d'opérations de *couper/coller* un peu trop hâties sur le code d'une application.

- Le nom de la méthode est trouvé dans plusieurs morceaux dont certains n'ont aucun lien de parenté. Il faut signaler l'ambiguïté.

La stratégie du point de vue le plus spécifique parmi les plus généraux considère également les descendants des morceaux caractérisant le point de vue, mais au lieu de les parcourir à partir des morceaux du point de vue, on calcule au préalable un *plus petit commun ancêtre* qui leur est, par définition, plus général. Deux cas peuvent se présenter.

- La méthode recherchée est définie pour le PPCA⁶. On remonte alors le lien de délégation à la recherche du premier ancêtre qui la déclare. Il peut s'agir du PPCA.
- La méthode recherchée n'est pas définie pour le PPCA. On descend alors chacune des branches formées par le lien de délégation récursivement. La récursion s'arrête lorsque l'on rencontre une déclaration de la méthode.
 - Si plusieurs morceaux sont candidats, il faut signaler l'ambiguïté.
 - Si aucun morceau n'est candidat, il faut signaler l'erreur.

Notre étude a permis de caractériser un certain nombre de problèmes et de nécessités qui n'avaient pas encore été mis en évidence, et de formuler une nette préférence quant à la stratégie à adopter. C'est ce que nous voyons dans la section suivante.

3.1.3.2 Ambiguïtés sur l'envoi de message avec point de vue

Voyons à présent, plusieurs configurations peuvent engendrer des ambiguïtés sur un envoi de message combiné⁷.

Les exemples de la FIG. 3.3 représentent toutes une classe d'objets morcelés d'un type très simple nommé `ClassePDV`, ainsi qu'une instance `i`. Dans les exemples qui suivent, on note le point de vue entre accolades, après le receveur du message et avant le sélecteur. On précise le morceau d'appartenance de la méthode sélectionnée entre parenthèses.

FIG. 3.3(a) `i{m2,m3} meth` correspond à `meth (m2)` pour la première stratégie et `meth (m1)` pour la seconde.

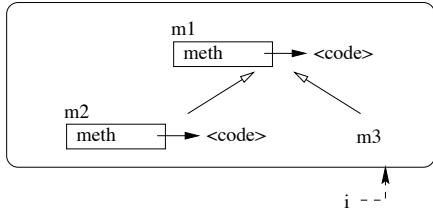
FIG. 3.3(b) L'ambiguïté sur `i{m2,m3} meth` pour la première stratégie est due au fait que `meth` est trouvé dans `m2` et `m3` qui sont incomparables vis-à-vis de la relation induite par le lien de délégation. La seconde stratégie recherche quant à elle, à partir du plus petit ancêtre commun qui est `m1` et qui déclare `meth`.

FIG. 3.3(c) Pour les mêmes raisons que précédemment, `i{m2,m3} meth` reste ambigu pour la première stratégie mais le devient également pour la seconde puisque `m1` ne déclare plus `meth` (Cette méthode est trouvée en deux points incomparables de la hiérarchie sur `m2` et `m3`).

⁶Plus petit commun ancêtre

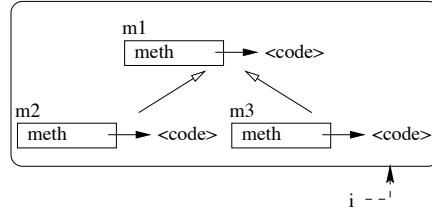
⁷Ces remarques sont également valables pour l'envoi de message global, sachant que celui-ci est une forme particulière d'envoi combiné.

ClassePDV



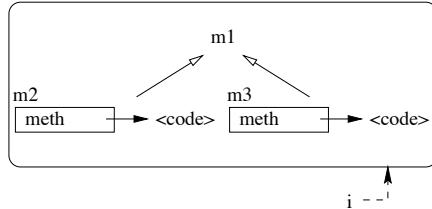
(a) Non ambigu, mais les résultats diffèrent suivant la stratégie.

ClassePDV



(b) Ambigu pour la première stratégie, non ambigu pour la deuxième.

ClassePDV



(c) Ambigu pour les deux stratégies.

FIG. 3.3: Exemples d'objets morcelés sur lesquels un envoi de message avec point de vue combiné peut aboutir à des résultats différents ou à des ambiguïtés suivant la stratégie choisie.

Ces exemples permettent déjà de prononcer une petite préférence en faveur de la seconde stratégie puisque certains cas ambigus pour la première, ne le sont pas pour la seconde et que le cas inverse ne peut pas se produire. Cette préférence sera accentuée par la suite de l'étude.

3.1.4 Point de vue et variables de morceau

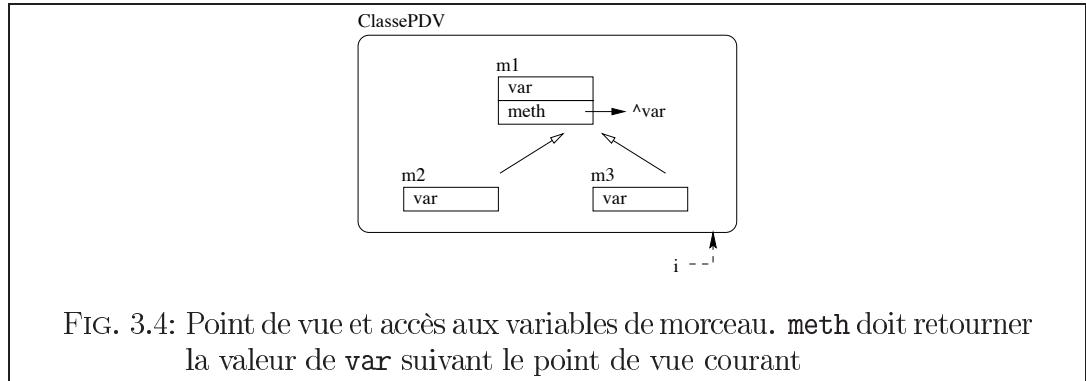
Un problème soulevé par notre étude concerne les accès aux variables de morceau. On montre, dans un premier temps, la nécessité de considérer un point de vue *implicite* lors des accès aux variables de morceaux faisant suite à un envoi de message avec point de vue. Dans un deuxième temps, on montre la nécessité de fournir au programmeur un mécanisme lui permettant de réaliser des accès *explicites*.

3.1.4.1 Accès implicite

Considérons que l'on demande le numéro de téléphone d'une **Personne** en tant qu'**Étudiant**. Une méthode accesseur **Téléphone** définie dans **Personne** retourne la valeur de la variable **téléphone**. Si on ne prend pas en compte le point de vue de l'envoi du message **Téléphone**, cette méthode retournerait invariablement la valeur détenue sous le point de vue de **Personne**. D'autre part, redéfinir cette méthode dans chaque morceau qui redéclare **téléphone** de manière à retourner la valeur correspondant au point de vue, rendrait inutile les possibilités de partage de comportement

induites par la délégation. Il devient donc nécessaire de prendre en compte, de façon *implicite*, le point de vue pour l'accès aux variables de morceaux.

Cette configuration correspond à celle représentée sur la FIG. 3.4.



3.1.4.2 Accès explicite

Définir un accesseur dans un super-morceau pour chaque variable redéclarée dans les sous-morceaux, permet, grâce à l'envoi de message avec point de vue explicite, d'obtenir la valeur particulière de la variable selon ce point de vue. Pour des raisons évidentes d'*encapsulation*, une telle méthode n'est pas toujours souhaitable.

Il faut donc fournir au programmeur d'une méthode de morceau, un moyen pour *expliciter* les accès qu'il fait sur les variables privées déclarées dans les super-morceaux. On se limite aux super-morceaux car il n'y a aucune raison évidente de permettre d'accéder à toutes les variables privées d'une classe d'objets morcelés donnée à partir d'une méthode de morceau de cette même classe. On encourage plutôt le programmeur à structurer correctement ses morceaux.

On spécifie un point de vue sur une variable, de la même façon que sur le receveur d'un envoi de message: entre accolades après le nom de la variable: `var1←var2{m1}`. L'utilisation des pseudo-variables (voir 3.1.3) a la même signification que pour l'envoi de message.

Le fait de devoir considérer un point de vue lors de l'accès aux variables engendre un certain nombre de problèmes lors d'un envoi de message avec point de vue combiné. Les problèmes ne sont pas les mêmes suivant la stratégie de désambiguïsation choisie (voir 3.1.3.1): celle du point de vue le plus spécifique ou celle du point de vue le plus spécifique parmi les plus généraux.

3.1.4.3 Problèmes engendrés par la première stratégie

Exhibons maintenant les problèmes qui découlent du fait que des ambiguïtés, concernant les accès aux variables de morceaux, peuvent apparaître même lorsque l'envoi de message est non ambigu (dans le cas de la stratégie du point de vue le plus spécifique):

- le premier type de problème est une généralisation du problème de conflit, lors d'un envoi de message combiné, aux variables de morceau;

- le deuxième type de problème rend compte de la difficulté d'organisation des programmes lorsque l'on utilise la stratégie du point de vue le plus spécifique.

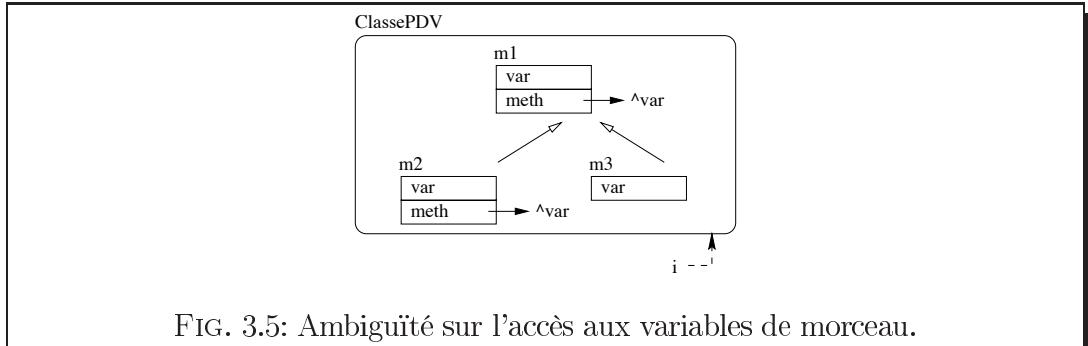


FIG. 3.5: Ambiguïté sur l'accès aux variables de morceau.

La FIG. 3.5 montre un exemple d'objet morcelé sur lequel l'envoi `i-{m2,m3} meth` n'engendre pas de conflit sur la recherche de la méthode, mais engendre un conflit sur l'accès à la variable `var`. Il faut donc également attacher une attention particulière aux conflits sur les accès aux variables de morceau qui s'opèrent dans le corps des méthodes de morceau.

Sur la FIG. 3.6(a), on peut voir l'exemple d'un objet morcelé pour lequel l'envoi de message `i-{m2,m3} meth` sélectionne `meth` (`m2`) et `var` (`m3`) qui se trouvent sur deux branches différentes de la hiérarchie des morceaux, d'où la difficulté d'associer un sens à cet envoi de message car la méthode et la variable sélectionnées ne sont pas mises en relation par le lien de délégation.

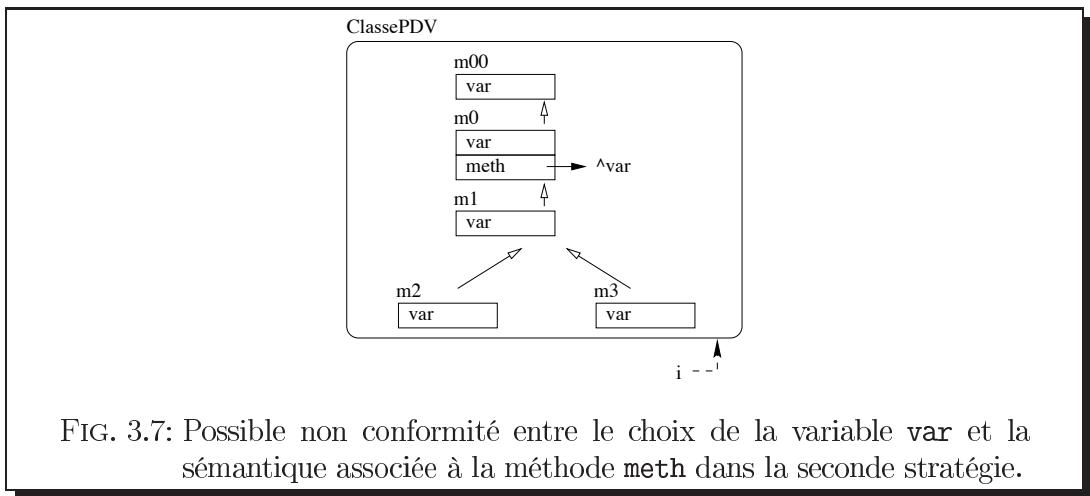
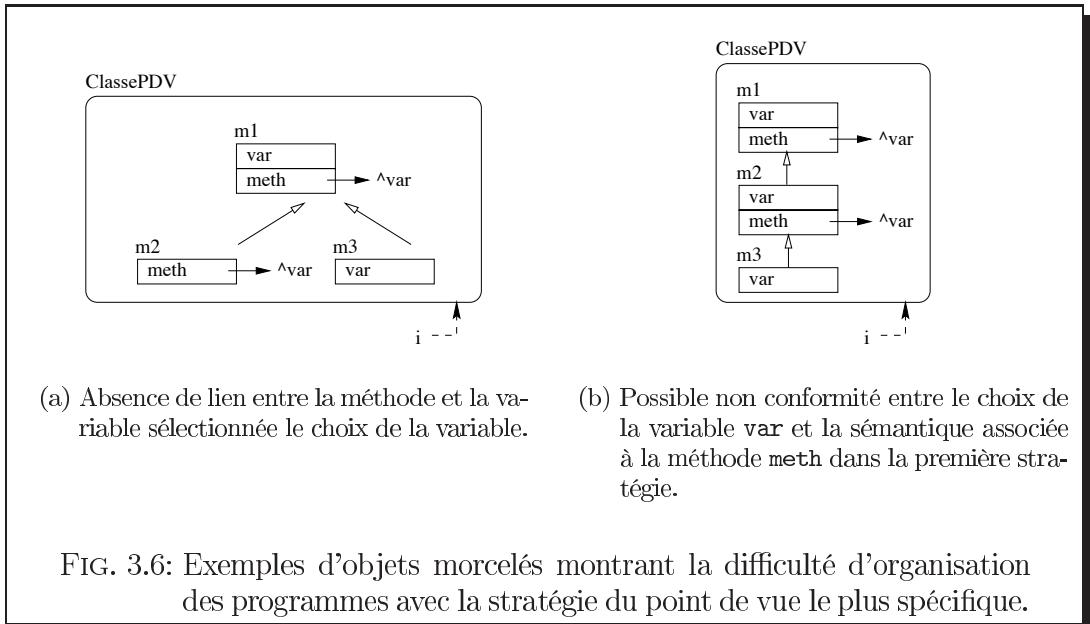
Sur le deuxième exemple de la FIG. 3.6(b), `i-{m2,m3} meth` sélectionne `meth` (`m2`) qui accède à `var` (`m3`) située à un niveau inférieur sur la même branche de délégation. Il s'agit certainement d'une opération non conforme avec ce que le programmeur de `meth` (`m2`) avait en tête: si celui-ci est habitué au modèle à classes standard, dans lequel la redéclaration de variable n'est pas permise, il pense très certainement que toutes les variables accédées par ses méthodes se trouvent en amont de sa méthode, puisque dans le modèle à classes standard, la redéclaration de variable est interdite. Il a cependant la charge de préciser de manière explicite les accès sur les variables (voir 3.1.4.2): l'utilisation des pseudo-variables `thisViewPoint` ou `genericViewPoint` sur l'accès à la variable `var` dans le corps de `meth` (`m2`) permet de régler le problème.

Considérons ces mêmes configurations avec la stratégie du point de vue le plus spécifique parmi les plus généraux.

3.1.4.4 Avantages de la seconde stratégie

Le problème soulevé par la FIG. 3.6(b), persiste toujours comme le montre la FIG. 3.7: la méthode sélectionnée par `i-{m2,m3} meth` est `meth` (`m0`), mais la variable accédée est `var` (`m1`). L'ambiguïté sémantique se lève de la même manière, en utilisant les pseudo-points de vue.

Par contre les autres disparaissent avec la seconde stratégie pour la raison suivante: l'utilisation de noms de variables au sein d'une méthode de morceau est conditionnée par le fait que la variable y soit définie. C'est à dire déclarée dans la classe, dans une super-classe, dans le même morceau ou un super-morceau, de la même classe, ou



d'une super-classe. Autrement dit une variable est déclarée nécessairement au même niveau ou en *amount* de la méthode qui l'utilise. Comme la seconde stratégie débute la recherche par *le plus petit commun ancêtre*, on est assuré que la méthode et chacune des variables accédées sont mises en relation par un lien de délégation. Cet état de fait permet également de mieux optimiser les accès aux variables puisque l'on peut se servir du morceau qui détient la méthode sélectionnée, comme point de départ de la recherche des morceaux qui contiennent les variables accédées.

3.2 Implantation des objets morcelés et choix du système

Comment représenter une classe d'objets morcelés au sein d'un système à classes standard ? On a visiblement affaire à un nouveau *type de classe* qui engendre naturellement un nouveau *type d'objet*:

1. Une classe d'objets morcelés détient des informations supplémentaires par rapport à une classe normale, comme par exemple la *hiérarchie des morceaux* qui la définit.
 2. On souhaite pouvoir manipuler des *classes d'objets morcelés*, pour, par exemple, leur ajouter ou enlever des morceaux, ou simplement les instancier. On doit donc disposer à cette fin, d'un *ensemble de méthodes* communes à toutes les classes d'objets morcelés (cette dernière remarque est également valable pour les objets morcelés eux-mêmes).
- L'opération d'instanciation, par exemple, est nécessairement différente de celle d'une classe normale puisqu'elle doit prendre en compte des informations spécifiques aux classes d'objets morcelés.
3. L'*envoi de message* aux objets morcelés est une opération qui ne peut pas se confondre avec l'envoi de message sur des objets standards, puisqu'elle doit parcourir la hiérarchie des morceaux à la recherche du code de la méthode sémantiquement adéquate.

Le choix du système à modifier peut maintenant s'orienter suivant que ce système dispose de *facilités* d'extension sur ces trois points fondamentaux, mais le paramètre de l'efficacité est également à prendre en compte. Dans un système réflexif et correctement réifié, ces modifications s'expriment en créant de nouvelles classes qui héritent des classes du système dont on veut étendre le comportement:

1. Introduire un nouveau type de classe dans un système réflexif, s'exprime par la définition d'une nouvelle méta-classe chargée de les instancier. Une classe d'objet morcelés en tant qu'objet, doit pouvoir contenir la *hiérarchie des morceaux* qui la caractérise: en plus des classiques dictionnaires de méthodes et de noms variables de classe et d'instance, un dictionnaire de morceaux est nécessaire; on déclare au niveau de sa méta-classe, une variable d'instance **pdvs** (voir figure 3.9).

Cette méta-classe des classes d'objets morcelés hériterait de celle des classes d'objets standards.

2. Les *comportements* associés aux classes d'objets morcelés font partie du protocole (de la description) de la nouvelle méta-classe. De même que la méthode **new**, chargée de l'instanciation, qui doit parcourir la hiérarchie des morceaux, à la recherche des variables d'instances et de morceaux déclarées.

les *méthodes de manipulation* des objets morcelés sont normalement déclarées dans une classe, disons abstraite, utilisée comme *patron* pour toutes les classes d'objets morcelés qui n'en héritent pas (voir 3.1.2).

3. Si l'*envoi de message* est réifié, une nouvelle classe d'envoi de message, qui hérite de celle associée aux envois normaux, est ajouté au système. Une instance d'envoi de message avec point de vue est automatiquement créé, chaque fois qu'un message est envoyé à un objet morcelé.

Malheureusement, l'envoi de message est un concept réifié uniquement dans les systèmes qui sacrifient du même coup l'efficacité: si chaque envoi de message donne lieu à une instanciation, on en comprend aisément les raisons. Pour les autres, l'envoi

de message est traité par des primitives au niveau de l'interpréteur. C'est cette configuration qui est discutée plus en détail dans le Chap. 4. D'autre part, tous les langages à classes réflexifs, ne permettent pas la création explicite de métaclasses, et Smalltalk en fait partie.

Le *noyau* d'un système désigne le plus petit sous-ensemble de ce même système qui permet de le faire fonctionner. On peut alors parler du noyau en termes complémentaires:

- d'ensemble minimal de classes,
- d'ensemble minimal de primitives,
- d'ensemble minimal d'instructions machine, etc...

L'organisation des classes du noyau renseigne sur les capacités du système à créer *explicitement* ou *implicitement* les métaclasses. Pour cette raison, un *noyau type* est présenté pour chacun des deux types de systèmes étudiés.

3.2.1 Systèmes à création explicite de métaclasses

Dans cette catégorie, on trouve les systèmes tels que Classtalk [Coi90], NeoClass-talk [Riv97] ou ObjVLisp [Coi87]. ObjVLisp a été le premier à unifier la notion de classe et métaclass, et à ne pas les distinguer. Dans ce système l'emphase est mise plutôt sur les objets capables de créer d'autres objets, c.a.d. répondant au message `new`, et les autres, que l'on appelle objets terminaux. On peut librement choisir la classe d'instanciation, et évidemment la super-classe, d'une classe au moment de l'introduire dans le système.

3.2.1.1 Organisation type des classes du noyau

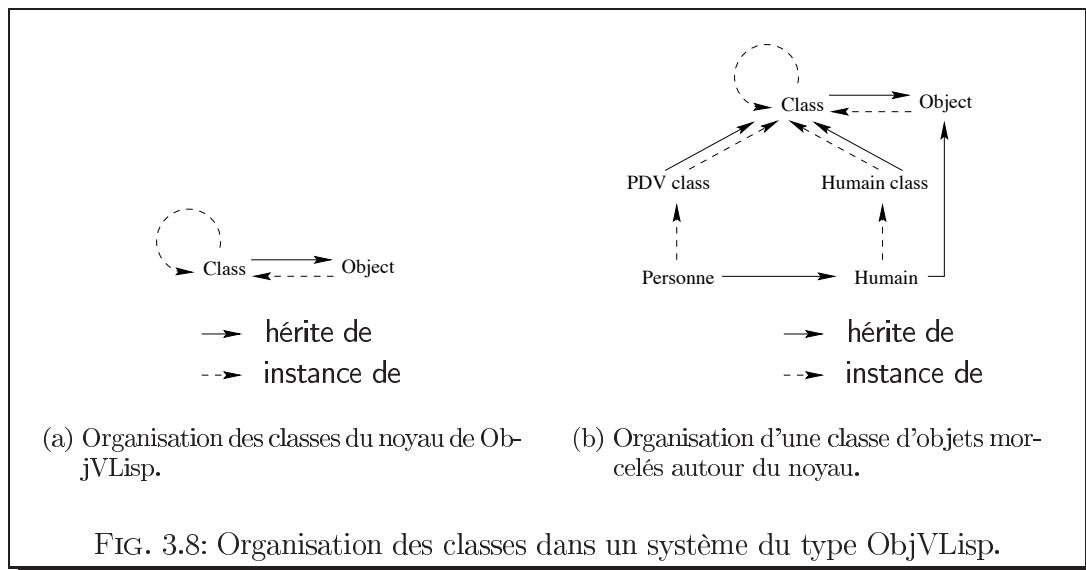


FIG. 3.8: Organisation des classes dans un système du type ObjVLisp.

Le noyau type n'est constitué que de deux classes: `Object` et `Class` (voir figure 3.8(a)). Sur une hiérarchie de classes, construite autour de ce type de noyau, on peut identifier (voir figure 3.8(b)):

- une *méta-classe*, qui hérite, directement ou indirectement, de **Class**, et qui est instance de **Class** ou d'une de ses sous-classes.

Le niveau d'abstraction d'une méta-classe et la longueur de la chaîne d'instanciation qui la relie à **Class**. Le premier niveau correspond à celui des classes, le troisième à celui des méta-méta-classes...

- une *classe*, qui n'hérite pas de **Class**, et qui est instance de **Class** ou d'une autre méta-classe;
- un *objet* qui est instance de **Object** où d'une de ses sous-classes. **Object** est la racine de la hiérarchie d'héritage de tous les niveaux d'abstraction grâce à la boucle d'instanciation sur **Class** qui hérite de **Object**. Tous les objets du système se comportent donc comme tels;
- un *objet terminal*, qui est instance d'une classe.

La contrepartie d'une telle organisation est que l'on laisse le soin au programmeur de s'assurer des problèmes de *compatibilité ascendante* et *descendante* qui peuvent subvenir entre classes (voir A).

3.2.1.2 Implantation des classes d'objets morcelés

Après avoir défini la nouvelle méta-classe PDV **class** associée aux classes d'objets morcelés, une nouvelle classe d'objets morcelés s'introduit de manière usuelle, par instantiation de cette méta-classe:

```
PDV class
super: #Humain
name: 'Personne'
i_v: #(...)
methods: #(...)
category: 'User-Object'
pdvs: #('personne' ('étudiant',
'employé'))
```

La hiérarchie partielle des classes correspondrait à celle représentée sur la FIG. 3.9.

Bien que la solution présente convienne tout à fait, on s'est posé la question suivante: pourquoi ne pas considérer une classe d'objets morcelés également comme un objet morcelé ? La section suivante montre les difficultés qu'engendrerait d'une telle organisation.

3.2.1.3 Modèle réflexif d'objets morcelés

L'idée de base est q'un objet morcelé est instance d'un objet morcelé de structure semblable: c.a.d. avec le même nombre de morceaux, organisés suivant une hiérarchie identique. On se place, pour des raisons de facilités de mis en oeuvre, dans le contexte des systèmes à création explicite de méta-classes.

Une telle organisation permet de changer la structure des morceaux représentés (en admettant qu'il leur faut au minimum les champs **super_m**, **name**, **m_v** et **methods** pour répondre au nom de morceau), et pourquoi pas d'associer à une classe d'objets morcelés, une hiérarchie de morceaux de types différents.

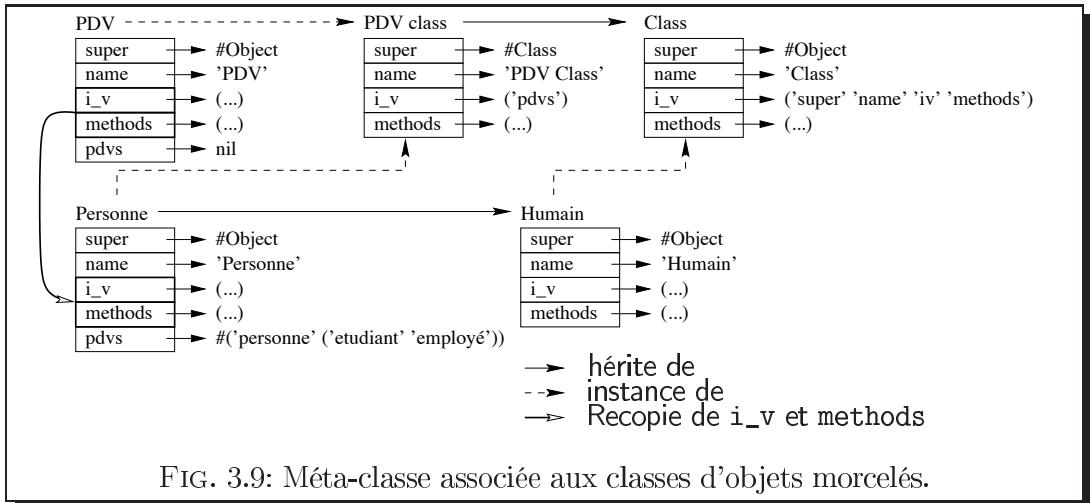


FIG. 3.9: Méta-classe associée aux classes d'objets morcelés.

Si l'idée est séduisante, la réalisation est beaucoup plus dure: pour arrêter la régression, il faut placer une boucle d'instanciation sur un objet morcelé de la chaîne d'instanciation dont la description est stable et qui peut par conséquent s'*autodécrire*: par exemple la méta-classe morcelée de la FIG. 3.10. Le problème est qu'il y a autant de méta-classes morcelées que de hiérarchie de morceaux différentes, et les descriptions ne peuvent plus « *converger* » vers celle de l'unique objet qui s'autodécrirait, comme dans le cas de **Class** du noyau d'ObjVLisp (voir 3.2.1.1).

Or ces classes qui s'instancient elles-mêmes sont construites en plusieurs étapes durant l'initialisation du système⁸. La mise à la disposition de l'utilisateur d'un tel mécanisme rendrait particulièrement instable le système.

Il faut donc plutôt considérer une classe d'objets morcelés comme un objet standard capable de stocker, à l'aide d'une variable déclarée dans sa méta-classe, une hiérarchie de morceaux. C'est la solution que nous considérons pour l'implémentation.

3.2.2 Systèmes à création implicite de méta-classes

Dans cette catégorie, on range le système Smalltalk-80 et tous ceux qui sont basés sur le même type d'organisation du noyau des classes.

3.2.2.1 Organisation type des classes du noyau

Le noyau de Smalltalk-80, ne permet pas la création explicite de méta-classe. On remarque sur la FIG. 3.11 que la séparation des classes et des méta-classes, y est très nette, et que l'on retrouve d'ailleurs, ce *découpage* au niveau du *browser*⁹ qui les catégorise séparément. Sur une hiérarchie de classes basée sur une représentation semblable, on peut identifier:

- une *méta-classe* qui est instance de **MetaClass**;
- une *classe* qui est instance de **Class** ou d'une de ses sous-classes. Les classes **ClassDescription**, **Behavior**, **Object** et **Class** s'autodécrivent;

⁸ou *bootstrap*.

⁹Outil de développement permettant de consulter les descriptions des objets au sein du système.

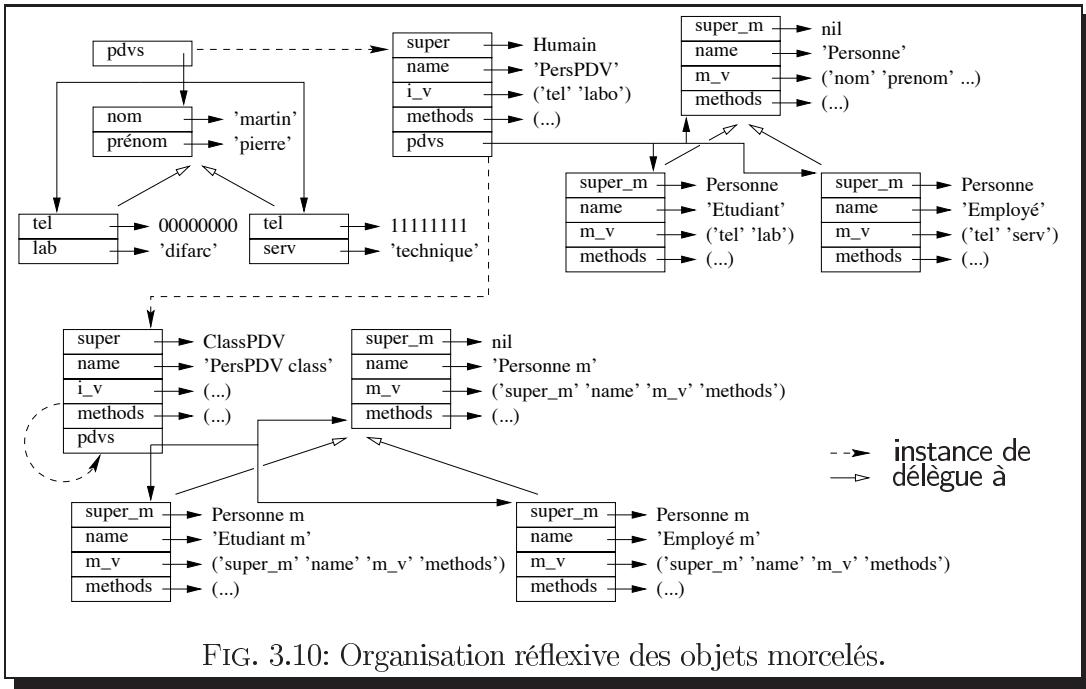


FIG. 3.10: Organisation réflexive des objets morcelés.

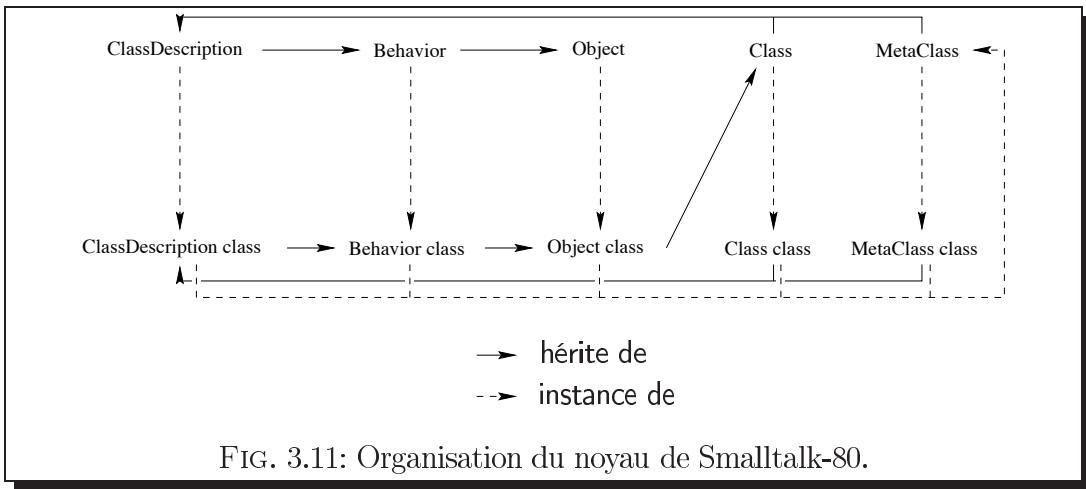


FIG. 3.11: Organisation du noyau de Smalltalk-80.

- un *objet* qui est instance de **Object** ou une de ses sous-classes. Le lien d'héritage entre **Object class** et **Class**, qui traverse étrangement un niveau d'abstraction, et le lien d'instanciation entre **MetaClass class** et **MetaClass**, qui relie en sens inverse les niveaux d'abstraction, font en sorte que **Object** soit la racine des hiérarchies d'héritage des deux niveaux d'abstraction, et donc que tous les objets du système se comportent comme tels.
- un *objet terminal* qui est instance d'une classe.

En créant implicitement une métaclass à chaque création de classe, Smalltalk assure la compatibilité ascendante et descendante (voir A) en maintenant l'héritage parallèle entre classes et métaclasses. C'est la raison essentielle qui rend impossible la création explicite de nouvelles métaclasses (voir 3.2.2).

3.2.2.2 Implantation des classes d'objets morcelés

On a vu (§3.2) que l'implantation des classes d'objets morcelés nécessitait la définition d'une méta-classe pour décrire les variables de classes nécessaires à la gestion de la hiérarchie des points de vue.

Smalltalk, fait partie des systèmes qui ne permettent pas la création explicite de méta-classes. Pour contourner cette difficulté, nous avons exhibé deux méthodes:

- en créant une classe abstraite d'objets morcelés de laquelle hériterait toutes les classes d'objets morcelés, on s'assure, grâce à l'héritage parallèle des classes et des méta-classes de Smalltalk, que toutes les méta-classes partagent du même coup les caractéristiques de la méta-classe de la classe abstraite. L'inconvénient majeur est qu'il devient impossible de faire hériter une classe avec points de vue d'une classe normale (voir 3.2.2.3) en l'absence d'héritage multiple;
- pour pallier cet inconvénient, il suffit de recopier les caractéristiques d'une classe d'objets morcelés *abstraite* et de sa méta-classe, respectivement dans la classe nouvellement créée et dans sa méta-classe (voir 3.2.2.4). L'inconvénient est que cela engendre une mauvaise factorisation des comportements, puisque des parties de description identiques existent en de multiples points de la hiérarchie des classes et des méta-classes.

La seconde solution est préférable, pour ne pas trop limiter le pouvoir expressif associé aux classes d'objets morcelés à l'intérieur du système.

3.2.2.3 Solution empêchant l'héritage avec une classe normale

La création d'une classe avec points de vue se fait de la manière suivante:

```
PDV
subclass: #Personne
instanceVariablesNames: ''
classVariableNames: ''
poolDictionary: ''
category: 'User-Object'.

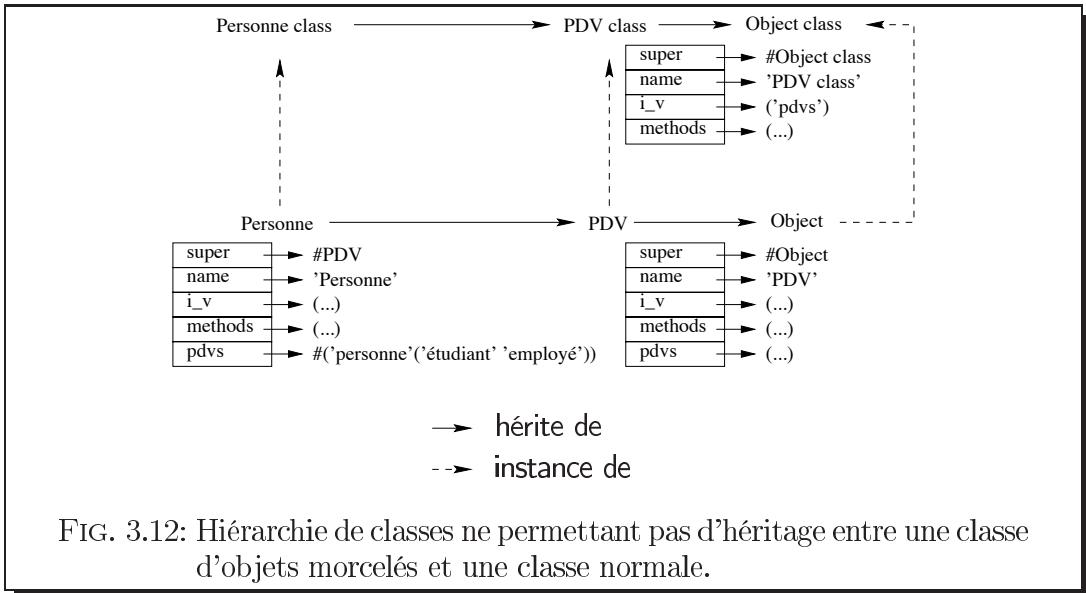
Personne
addPDVs: #('personne' ('étudiant' 'employé')).
```

Toute classe avec points de vue hérite directement ou indirectement de la classe PDV, de telle sorte que sa structure et son comportement, définis dans sa méta-classe, soient également hérités de la de PDV class (voir figure 3.12). C'est le seul critère caractérisant une classe d'objets morcelés. En l'absence d'héritage multiple, il est impossible de faire hériter une classe d'objets morcelés d'une classe normale.

L'adjonction de morceaux se fait par les méthodes définies dans PDV class, par exemple addPDVs::

3.2.2.4 Solution autorisant l'héritage avec une classe normale

La classe PDV existe toujours, et on peut toujours en faire la même utilisation que dans §3.2.2.3 à l'aide de l'héritage, mais elle peut également être utilisée comme un



patron, dans le cas où l'on souhaite faire hériter une classe avec points de vue d'une classe normale.

Les variables déclarées et les méthodes définies dans **PDV** et **PDV class** seront copiées respectivement dans la nouvelle classe et dans sa métaclass (voir figure 3.13) à la suite d'un message spécifique (**pdvsubclass:...**), utilisé pour demander à une classe de retourner une sous-classe d'objets morcelés¹⁰:

```
Humain
pdvsubclass: #Personne
instanceVariablesNames: ''
classVariableNames: ''
poolDictionary: ''
category: 'User-Object'
pdvs: #('personne' ('étudiant' 'employé'))
```

Ce procédé permet d'associer la même structure de base et le même comportement aux objets morcelés et à leur classes tout en laissant libre l'unique lien d'héritage qui peut être utilisé pour hériter d'une classe normale.

On notera que **pdvsubclass:...** et **subclass:...** fonctionnent de la même façon lorsqu'ils sont envoyés à une classe d'objets morcelés puisque, dans ce cas, une sous-classe est nécessairement une classe d'objets morcelés et que la recopie de **i_v** et **methods** est inutile puisque la sous-classe en question en hérite déjà.

Il est nécessaire d'introduire ce message au niveau de la classe décrivant les comportements des classes, de manière à pouvoir demander à n'importe laquelle de retourner une sous-classe avec point de vue. On parlera au Chap. 4 des modifications à apporter au *noyau du système*.

¹⁰Le message **subclass:...** sert habituellement à demander à une classe, une sous-classe standard.

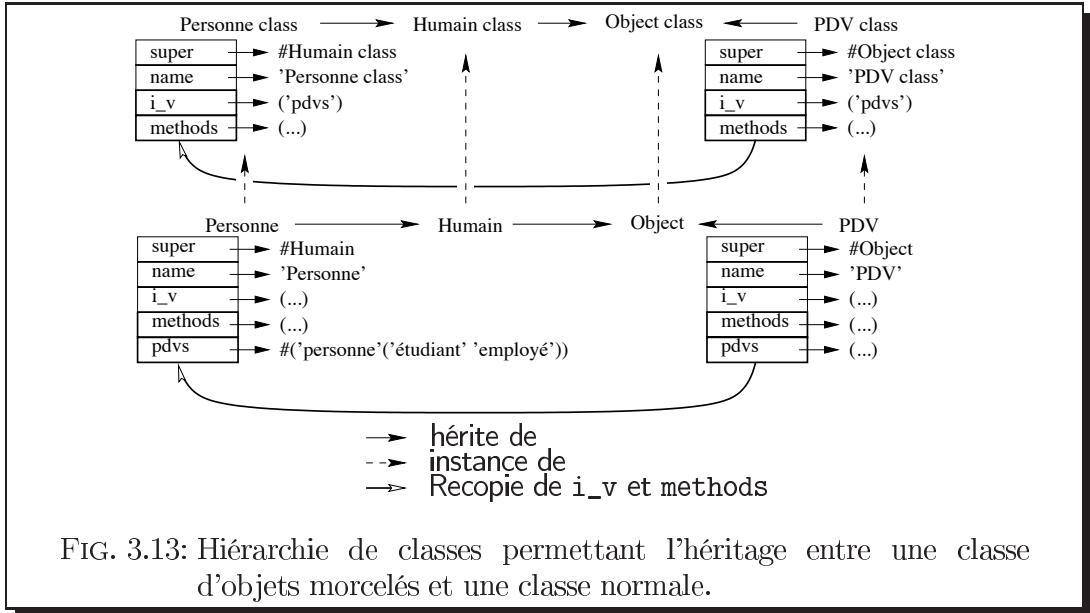


FIG. 3.13: Hiérarchie de classes permettant l'héritage entre une classe d'objets morcelés et une classe normale.

3.3 Exemple pratique

Pour rendre la suite de notre exposé plus clair, il apparaît nécessaire de montrer les possibilités de notre modèle sur un exemple pratique.

Notre exemple tente de représenter le maximum d'opérations permises, tout en restant proche des exemples qui ont déjà été présentés tout au long de ce rapport. Voici une brève description de ce qui est représenté sur la FIG. 3.14:

- une classe normale;
- deux classes d'objets morcelés avec leur hiérarchie de points de vue;
- deux formes d'héritage:
 1. Une entre la classe standard et une classe d'objets morcelés.
 2. Une entre deux classes d'objets morcelés. Cette deuxième forme d'héritage permet de montrer diverses opérations permises sur la hiérarchie des morceaux:
 - adjonction d'un nouveau point de vue;
 - adjonction de nouvelles propriétés sur un morceau existant;
- deux instances: une instance pour chaque classe d'objets morcelés.

On a choisi de ne pas représenter tous les accesseurs aux variables déclarées. L'accesseur `Telephone` montre le principe général pour leur définition: ils sont déclarés dans le premier morceau qui déclare la variable à laquelle ils accèdent. Les sous-morceaux disposent de cet accesseur par délégation. Pour la convention: leur noms sont ceux des variables accédées dont la première lettre a été capitalisée.

Le code suivant correspond aux instructions possibles qui ont permis la création de la hiérarchie de la FIG. 3.14:

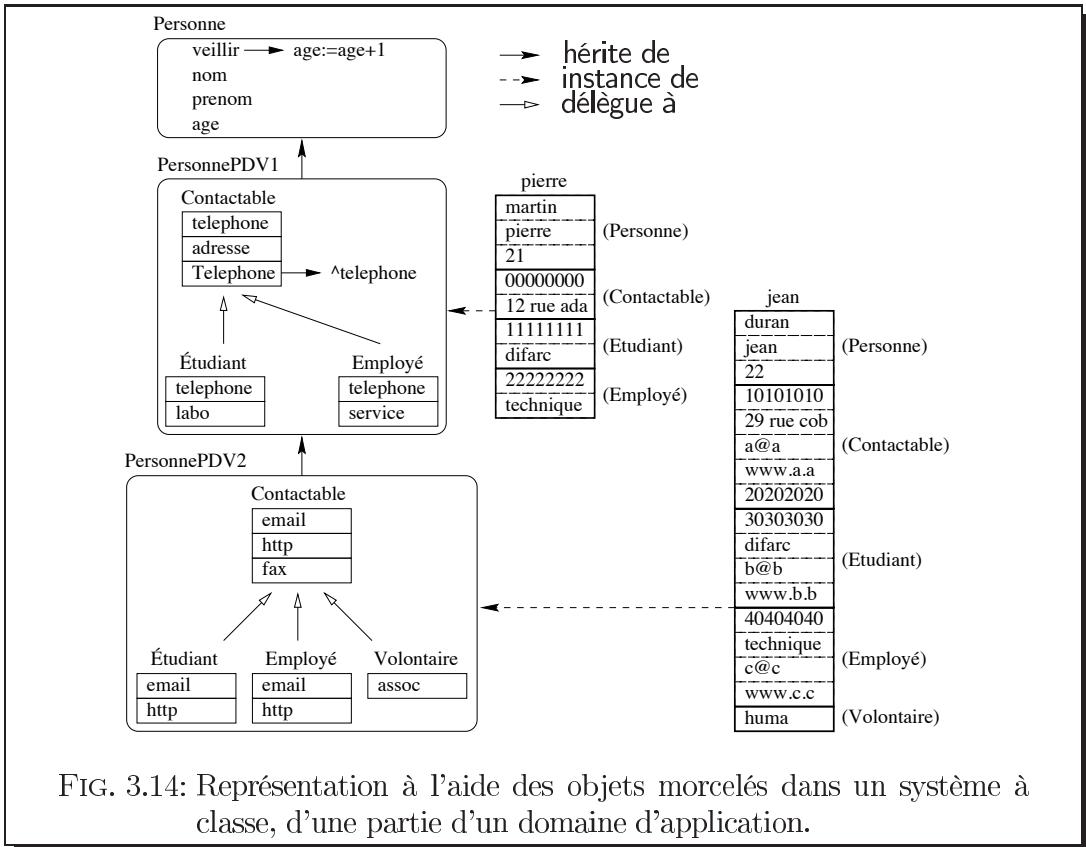


FIG. 3.14: Représentation à l'aide des objets morcelés dans un système à classe, d'une partie d'un domaine d'application.

```

Personne pdvsubclass: #PersonnePDV1
  instanceVariablesNames: ''
  classVariablesNames: ''
  poolDictionary: ''
  category: 'User-Object'
  pdvs: #('Contactable' ('Étudiant', 'Employé')).

PersonnePDV1 addInstvars: 'telephone adresse' onMrc: #Contactable.
PersonnePDV1 addMethod: Telephone_meth onMrc: #Contactable.
PersonnePDV1 addInstvars: 'telephone labo' onMrc: #Étudiant.
PersonnePDV1 addInstvars: 'telephone service' onMrc: #Employé.

PersonnePDV1 pdvsubclass: #PersonnePDV2
  instanceVariablesNames: ''
  classVariablesNames: ''
  poolDictionary: ''
  category: 'User-Object'
  pdvs: #('Contactable' ('Volontaire'))..

PersonnePDV2 addInstvars: 'email http fax' onMrc: #Contactable.
PersonnePDV2 addInstvars: 'email http' onMrc: #Étudiant.
PersonnePDV2 addInstvars: 'email http' onMrc: #Employé.
  
```

```
|PersonnePDV2 addInstvars: 'email http' onMrc: #Volontaire.
```

Le code suivant montre quant à lui, des exemples d'utilisation de ces objets nouvellement créés. On suppose que les deux instances pierre et jean ont déjà été créés, et que leurs états correspondent à la représentation qui en est faite FIG. 3.14:

```
pierre Telephone  
-> 00000000  
pierre{Etudiant} Telephone  
-> 11111111  
pierre{Etudiant,Employé} Telephone  
-> 00000000  
jean{Employé} Email  
-> c@c  
pierre{Employé} Email  
-> doesNotUnderstand
```

4

Implémentation dans un système Smalltalk

Même si certains choix, qui ont été fait dans l'étude précédente, tendent à favoriser plus un type de système à classes qu'un autre, on a essayé de rester le plus général possible de manière à ne pas contraindre le choix définitif du système hôte. C'est ce choix qu'il nous incombe de faire à présent, et c'est logiquement que l'on s'est tourné vers Smalltalk qui apparaissait comme le candidat le mieux placé compte tenu des orientations qui avaient été prises.

4.1 Efficacité de l'implémentation

Il est courant en Smalltalk d'intercepter certains messages émis par le système, lorsque celui se trouve dans un état d'exception: par exemple un envoi de message sur un receveur donné, n'a pu aboutir. C'est à ce niveau que l'on *greffe* le code chargé d'étendre la sémantique du langage: on s'arrange pour faire aboutir l'envoi de message qui a échoué, si celui-ci est conforme à la nouvelle sémantique. Cette stratégie que l'on qualifiera de *au plus tard* se situe à la *fin* du processus d'interprétation. Une stratégie qui agirait *au plus tôt* toucherait, elle, aux spécifications de la machine virtuelle ou du moins au compilateur, pour prendre en compte dès le *début* de l'interprétation la nouvelle sémantique du langage. D'autre part l'écriture de primitives pour répondre aux nouvelles spécifications de la machine virtuelle, nous assure d'une vitesse d'exécution optimale.

Il existe de nombreux systèmes Smalltalk tournant sur une multitudes de plate-formes, et ce sont les possibilités d'extension de ces systèmes qui doivent orienter notre choix définitif. Augmenter la sémantique associée à un langage de programmation est une affaire de compromis entre *facilité d'extension* et *vitesse d'exécution*:

- modifier un système à classes entièrement réflexif et réifié où, dans l'absolu, chacune de ses composantes prend la forme d'un objet, s'exprime par une relation d'héritage sur les classes associées aux composantes que l'on veut modifier. De tels systèmes, hautement réifiés et réflexifs, ont cependant la réputation, non usurpée, d'être lents à l'exécution;
- pour les systèmes partiellement réifiés, l'opération est plus délicates dans le sens, où il faut s'attaquer aux sources du systèmes qui sont, pour des raisons d'optimisation, rarement écrites dans le même langage que le système qu'elles

décrivent: le C++, C et même l'assembleur sont parfois utilisés. Mais la condition première reste cependant de disposer des sources.

De nombreuses extensions de Smalltalk ont été écrites comme Classtalk [Coi90], NeoClasstalk [Riv97], Prototalk [Don97], ROME [CDG90]. Même si certaines caractéristiques de ces extensions peuvent faciliter l'implantation des objets morcelés, il est difficile d'évaluer le bruit occasionné par les caractéristiques de l'extension qui ne nous sont d'aucune utilité, sur les performances à l'exécution.

Smalltalk dispose d'un noyau correctement documenté, et a été largement étudié. Repartir du noyau d'origine, signifie travailler avec des spécifications qui « *collent* » au plus près avec celles de [GR83].

4.2 Composantes d'un système Smalltalk

Un système Smalltalk est en général composé par

- les *sources* du système qui contiennent le code de toutes les classes du système,
- une *image* qui contient l'état sauvegardé d'une *session*, interprétable conformément aux spécifications de la machine virtuelle,
- les *modifications* relatives aux sources du système initial,
- un *exécutable* qui implémente l'interpréteur de codes de la machine virtuelle.

4.3 Squeak

Squeak, est une implémentation efficace de Smalltalk écrite en Smalltalk, et de surcroît domaine public. L'interpréteur Smalltalk est décrit par une classe au sein du système. Cette particularité offre, lorsque l'on est amené à modifier l'interpréteur, le double avantage suivant:

- un procédé de *génération de code C* permet de traduire le code Smalltalk de l'interpréteur pour pouvoir générer un exécutable rapide associé à la machine virtuelle, et ciblé sur une machine réelle. Squeak est donc *portable* sur à peu près toutes les machines qui existent.
- un *simulateur d'interpréteur*, dont la classe hérite de celle de l'interpréteur, permet de simuler la machine virtuelle que l'on est en train de modifier, au sein de la machine virtuelle associée à l'exécutable qui fait *tourner* le système. Cette façon de faire provoque un ralentissement non négligeable du système, mais permet de tester la « *viabilité* » des modifications apportées à l'interpréteur, avant de générer un exécutable, plus rapide, sur la machine cible. Cela permet également d'éviter la phase de génération/recompilation fastidieuse des sources C de l'interpréteur durant la phase de test.

Toutes ces facilités, ajoutées au fait que Squeak implémentait un noyau Smalltalk-80, ont fait de lui le candidat de choix.

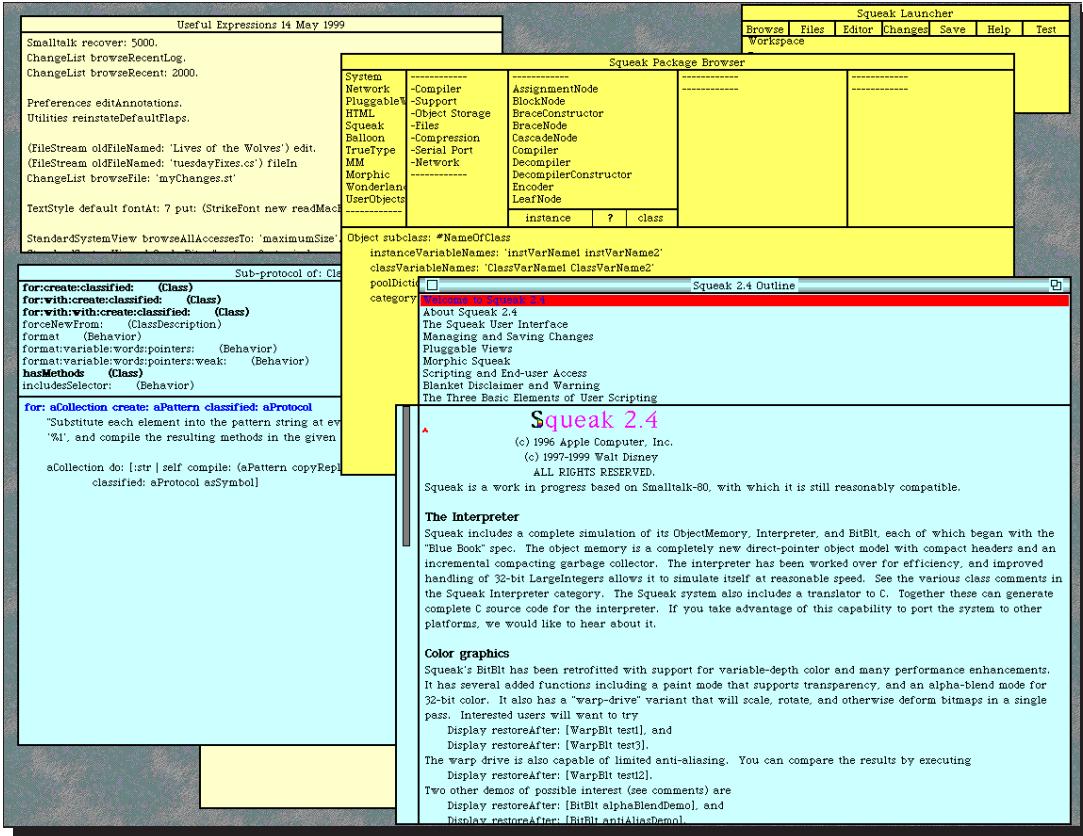


FIG. 4.1: Environnement de programmation de Squeak.

4.4 Analyse des besoins

Le §3.2 nous a déjà permis d’appréhender les modifications à apporter à un système pour l’implantation suivant trois points fondamentaux (*instanciation, manipulation, envoi de message*). Cependant, la description de l’implémentation réalisée est faite plutôt en suivant une démarche du *bas vers le haut*:

1. On établit les *besoins* en nouvelles primitives pour permettre l’existence des objets morcelés, et de l’envoi de message avec point de vue.
2. On détermine les besoins en nouveaux *bytecodes*, pour permettre la compilation efficace des envois de messages avec point de vue.
3. On détermine ensuite les modifications à apporter au compilateur et à l’interpréteur pour donner aux objets morcelés leur sémantique opérationnelle.

4.4.1 Factorisation de comportement

En l’absence de création explicite de métaclass, et de manière à pouvoir associer un comportement commun à toutes les classes d’objets morcelés, la solution du §3.2.2.4 préconise de se servir de la description de la métaclass PDV `class` comme d’un *squelette* pour les métaclasses des classes d’objets morcelés, en y recopiant les noms des variables de classes et en y recompilant les méthodes de classes. Pour cette raison,

et dans un soucis d'économie mémoire, il paraît évident de ne pas surcharger PDV `class`. La stratégie générale est plutôt de définir une méthode `isPDV` qui rend `faux` pour les classes normales et `vrai` pour les classes d'objets morcelés. On pourra de cette façon adapter le comportement de chaque méthode directement dans le noyau, au lieu de les redéfinir au sein de PDV `class`. Cette remarque prévaut également pour la classe PDV qui décrit les comportements de base des objets morcelés (voir 3.1.2).

Normalement cette façon de faire est à proscrire puisque l'on obtient la même chose, mais de façon beaucoup plus élégante, sans avoir recours à un test, à l'aide de la liaison dynamique. Mais en l'absence d'héritage multiple, cette solution paraît la plus réaliste. Notons également que l'image de Smalltalk contient presque 700 méthodes similaires (`isQuelqueChose`), utilisées par presque 56% des classes [Riv97], mais le fait est que certains concepts échappent à une modélisation *propre*¹ dans les langages à objets [Per98].

4.5 Nouvelles primitives

Les nouvelles primitives concernent essentiellement deux domaines fondamentaux de l'implémentation:

1. L'*instanciation* d'une classe d'objets morcelés;
2. L'*envoi de message* sur un objet morcelé.

4.5.1 Instanciation d'une classe d'objets morcelés

Instancier une classe d'objets morcelés nécessite de contrôler les mécanismes qui interviennent avant l'allocation mémoire, et plus précisément l'opération visant à calculer la taille allouée à l'instance, puisque dans notre cas, on doit, en plus, tenir compte des compléments de descriptions contenus dans les morceaux de la classe.

C'est dans cette optique que l'on examine au préalable, les mécanismes qui se trouvent derrière l'instanciation des classes normales.

4.5.1.1 Structure des objets dans Squeak

Examinons la structure des objets en mémoire dans le système Squeak [IKM⁺96]: un objet est représenté sous la forme d'un bloc en mémoire, qui débute par une entête (voir figure 4.2) de taille variable, et qui est éventuellement suivi des valeurs des variables qui définissent l'état de l'objet. L'entête de base est composée de champs de bits qui ont chacuns une signification particulière:

storage management est utilisé par l'algorithme de marquage du *garbage collector*;

object hash est une clé de hachage destiné à placer l'objet dans un dictionnaire et à optimiser les fonctions de recherches en tout genre;

cClass est un *pointeur compact* sur la classe d'instanciation de l'objet si celle-ci se trouve être une des 32 classes accessibles par ce type de pointeur. Par mesure d'optimisation, les classes les fréquemment instanciées, comme **Integer**, ont un

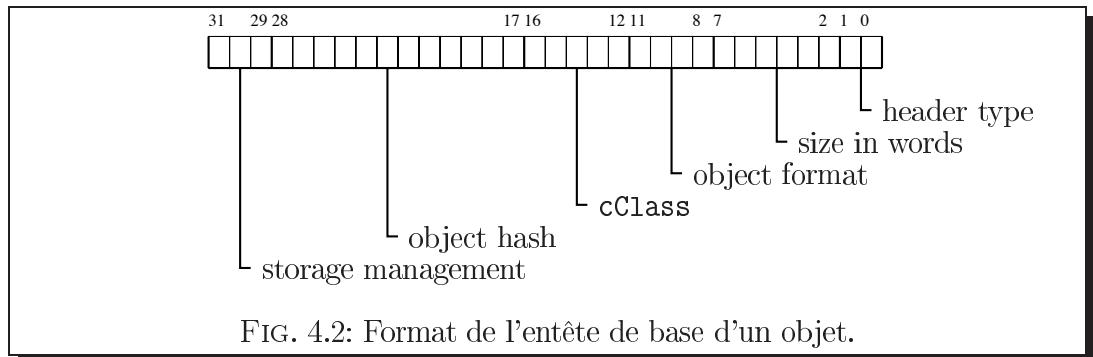
¹Voir à ce propos le cas des Booléens en Smalltalk.

pointeur compact². Pour les autres, cela nécessite que l'entête de leur instances soit étendu d'un mot ou deux pour stocker l'adresse entière;

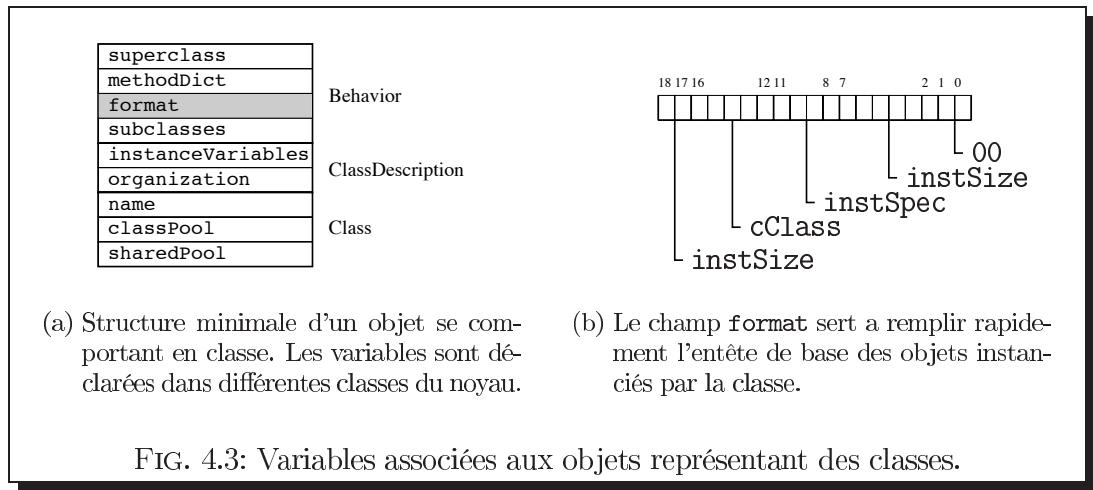
object format permet de savoir si l'état de l'objet est accessible par l'intermédiaire des noms de variables définies dans sa classe, par l'intermédiaire d'indexes, ou bien les deux.

size in words comme son nom l'indique, donne la taille en mots de 32 bits, de la zone alloué à l'objet, taille de l'entête non comprise;

header type permet de savoir si l'entête de l'objet est en 1,2 ou 3 mots de 32 bits.



Une classe est une instance de **Class** ou d'une de ses sous-classes (voir 3.2.2.1). Son état est donc au minimum décrit par les valeurs des variables déclarées dans **Class**, **ClassDescription** et **Behavior** (voir figure 4.3(a)).



Au moment de l'instanciation d'une classe, la primitive **primitiveNew**, décrite dans la classe **Interpreter**, utilise les informations détenues par cette dernière de manière à allouer une taille réaliste à l'instance et à lui affecter une entête conforme. La valeur de la variable **format** (voir figure 4.3(b)) joue un rôle essentiel dans ce processus puisque c'est elle qui est utilisée pour remplir l'entête de base des objets

²82% des objets d'une image standard Squeak sont instances de telles classes, ce qui leur permet d'avoir une entête qui fait seulement un mot, et réaliser une économie mémoire substantielle.

instanciés: on peut établir une correspondance directe entre les champs de bits de l'entête de base d'une instance et les champs de bits de la variable `format` de sa classe d'instanciation.

4.5.1.2 new et l'instanciation

Tout comme des instances de classes normales, les instances de classes d'objets morcelés ne contiennent que l'état des objets auxquels elles correspondent. Elles peuvent être représentées sous forme d'un seul bloc en mémoire: la séparation entre les états associés au différents morceaux d'un objet n'étant que virtuelle (voir figure 3.2).

La seule différence entre le mécanisme d'instanciation d'une classe normale et celui d'une classe d'objets morcelés réside dans la façon de compter les variables d'instance: on ne doit pas se limiter uniquement aux variables normales dans le cas des classes d'objets morcelés, mais on doit aussi parcourir tous ses morceaux à la recherche de celles qu'ils déclarent.

A la lumière de ce qui été vu au §4.5.1.1, la solution apparaît évidente: la valeur du champ `instSize` de la variable `format` d'une classe détermine la taille mémoire allouée à ses instances par la primitive d'allocation. C'est donc après la création d'une classe, ou après la modification de sa description, qu'il faut « *corriger* » cette valeur pour qu'elle tienne compte de la place à réservé pour les morceaux.

Une classe avec point de vue est créée en envoyant le message `pdvsubclass:...` à sa super-classe. Cette méthode fait appel à `subclass:...` qui effectue tous les calculs de champs comme s'il s'agissait d'une classe normale. On corrige ensuite la valeur du champ `instSize`.

`new` peut maintenant s'appliquer sur une classe d'objets morcelés de manière totalement transparente, et sans avoir à modifier son code.

4.5.2 L'envoi de message avec point de vue

L'envoi de message sur un objet morcelé possède un paramètre de plus qu'un envoi sur un objet standard; le point de vue, ou plutôt l'identifiant permettant de caractériser ce point de vue. Dans le Chap. 3, on avait écarté les systèmes réifiant le concept de message dans notre choix pour l'implémentation pour des raisons d'efficacité, bien que l'implémentation s'en serait trouvée considérablement simplifiée.

Dans Squeak, l'envoi de message est compilé, et représenté directement sous forme de *bytecodes* (voir 4.6.3). Ces bytecodes font partie des spécifications de la machine virtuelle Smalltalk, et expriment spécifiquement des envois de messages. Ils sont décodés, au niveau de l'interpréteur, par des primitives dédiées qui déclenchent la recherche du code de la méthode à appliquer dans le contexte du receveur, en fonction du sélecteur et de la sémantique de l'envoi de message. Dans le cas où la recherche aboutit, le code de la méthode est interprété. La recherche de sélecteur à partir d'une classe donnée, est implémentée dans la méthode `lookupMethodInClass` de `Interpreter`.

On doit considérer les deux types d'envoi de messages avec point de vue (explicite et implicite), car ils s'implémentent de façons différentes.

4.5.2.1 Point de vue implicite

Cette forme d'envoi de message ne se différencie pas, syntaxiquement parlant, d'un envoi de message normal. Comme on se place dans le cadre d'un langage dynami-

quement typé, il est impossible de savoir au moment de la compilation si le message sera envoyé à un objet morcelé ou à un objet standard. Et pourtant il faut bien compiler l'envoi de message... on doit donc modifier toutes les primitives qui ont attrait à l'envoi de message normal, pour traiter le cas de l'envoi sur un objet morcelé avec point de vue implicite.

4.5.2.2 Point de vue explicite

Les envois de messages normaux et les envois avec point de vue explicite se diffèrent au cours d'une simple analyse syntaxique. Cet état de fait permet de traiter ces derniers de deux manières:

1. Par le truchement d'une *macro-expansion*, qui permet, avant la compilation proprement dite, de traduire un envoi de message avec point de vue explicite, sous la forme d'un envoi de message normal avec un paramètre en plus: l'identifiant du point de vue. Cette méthode (`perform:withArguments:withPDV:`) devrait s'implémenter au niveau de PDV. Cependant, pour les raisons avancées au §4.4.1, on l'implémenterait plutôt au niveau de `Object` en utilisant le test `isPDV` sur la classe d'instanciation de l'objet morcelé, seul susceptible de répondre à ce type de message.
2. Par la compilation directe de l'envoi de message avec point de vue sous la forme de bytecodes qui intègrent, dans leurs codages, les informations du point de vue proprement dit. Cette technique est la plus délicate car elle touche en profondeur le système, mais c'est également la plus performante.

4.5.2.3 Analyse des besoins

S'il on veut traiter les envois de messages avec point de vue il faut:

1. Modifier les spécifications la machine virtuelle (voir 4.6), si l'on choisit la méthode des bytecodes, et étendre l'interpréteur pour qu'il reconnaisse ces nouveaux bytecodes (ce qui passe par l'écriture de nouvelles primitives).
2. Introduire une nouvelle primitive dans l'interpréteur, chargée de rechercher la méthode au niveau de la hiérarchie des classes, mais aussi au niveau de la hiérarchie des morceaux de chaque classe d'objets morcelés: `lookupMethodInPDVClass`.
3. Modifier le code de `lookupMethodInClass` pour prendre en compte un envoi de message avec point de vue lorsque que celui-ci est implicite.
4. Étendre la syntaxe du langage pour pouvoir exprimer les envois selon un point de vue explicite, et modifier le compilateur pour qu'il *traduise* ce type d'envoi de message avant de les compiler ou pour qu'il les compile directement (suivant la solution retenue).

4.6 Modification de la machine virtuelle

Par le terme machine, on entend une description de la structure interne et des instructions d'un interpréteur de codes informatiques. Dans le cas où cet interpréteur

possède une architecture matérielle, on parle de machine cible ou de machine réelle. Si l'interpréteur est simplement représenté par une couche logicielle, on parle de machine virtuelle. L'interpréteur d'une machine virtuelle, est alors fatallement représenté par un programme informatique ciblé sur une machine réelle.

Tous les systèmes Smalltalk sont basés sur une machine virtuelle car cela leur permet:

- d'utiliser des instructions particulières, introuvables sur la plupart des machines réelles, pour représenter, par exemple, les envois de message, et obtenir un code compilé compact;
- de pouvoir exécuter n'importe quel programme Smalltalk compilé sur n'importe quelle machine, à partir du moment que celle-ci est pourvue d'un interpréteur natif Smalltalk.

Maintenant que les objets morcelés existent physiquement, il reste à pourvoir la machine virtuelle de nouvelles opérations permettant de les manipuler.

On peut répertorier les besoins:

- accéder aux variables d'instances déclarées dans la classe et les morceaux;
- répondre aux envois de messages dirigés explicitement selon un point de vue.

4.6.1 Accès aux variables d'instances

La structure des objets morcelés étant exactement la même que celle des objets normaux, la machine virtuelle est d'hors et déjà capable d'accéder au contenu d'un objet morcelé: aucune instruction supplémentaire n'est nécessaire. C'est au niveau du compilateur et de l'interpréteur qu'il y aura par contre, de nombreuses modifications.

4.6.2 Envois de message avec point de vue

Si l'on décide d'implémenter les envois de messages avec point de vue explicite, directement au sein de la machine virtuelle, il faut leur affecter un ensemble d'instructions libres et déterminer la façon de les coder.

4.6.3 Les bytécodes dédiés à l'envoi de message

Les *bytécodes* sont des entiers compris entre 0 et 255, qui préfixent les instructions de la machine virtuelle.

Affecter un nouveau jeu d'instructions aux envois de message avec point de vue peut se faire de différentes manières:

- modifier des instructions existantes pour prendre en compte la nouvelle sémantique. Cette opération est délicate dans le sens où les anciennes applications compilées, ne pourront plus être exécutées par l'interpréteur basé sur la nouvelle machine virtuelle. C'est par exemple le cas de l'image Smalltalk.

Le décodage en devient un peu plus compliqué, car un même bytecode préfixe alors plusieurs instructions sémantiquement différentes, mais on réalise de cette façon l'économie d'un bytecode;

- ajouter de nouveaux codes à son jeu d'instructions. Cela nécessite cependant qu'il y en reste quelque uns de libres. C'est heureusement le cas de la machine Smalltalk, qui n'associe aucune opération aux codes 126 et 127.

C'est cette solution, qui ne remet pas en cause la compatibilité des applications existantes, qui est étudié par la suite.

Il faut maintenant définir le codage des nouvelles instructions, qui rappelons le, sont préfixées par les bytécodes nouvellement alloués. On examine dans un premier temps, le codage des instructions d'envoi de message normaux, de manière à rester cohérent avec le modèle existant.

Les *bytecodes* dédiés aux envois de messages, peuvent être placés en deux grandes catégories suivant le type d'envoi de message qu'ils représentent:

- les envois normaux qui déclenchent la recherche de méthode à partir de la classe d'instanciation du receveur;
- les envois au *pseudo-recepteur super* qui permettent de commencer la recherche à partir de la super-classe de la classe déclarant la méthode qui a effectué l'envoi.

L'envoi de message est le principe essentiel dans un système à classes, et le nombre de bytécodes qui y sont consacrés, dans la machine virtuelle de Smalltalk, est important. Pour des raisons d'efficacité, il est préférable que l'interpréteur passe plus de temps à exécuter du code proprement dit, qu'à décoder des bytécodes d'instructions compliquées.

Associés au bytécode proprement dit, viennent plusieurs octets, en général un ou deux, qui sont en quelque sorte les arguments associés à l'opération, et l'ensemble représente une instruction en langage machine. La structure associée à ce(s) octet(s) est en général, pour ce qui concerne des envois de messages, un compromis entre le nombre d'arguments pouvant accompagner l'envoi de message, et le nombre de sélecteurs pouvant être référencés, car le sélecteur d'un envoi de message est un index sur un tableau de noms de sélecteurs. (*voir TAB. 4.1*).

bytécode	arguments	sélecteurs	envoi	primitive
208-223	0	16	normal	sendLiteralSelectorBytecode
224-239	1	16	normal	sendLiteralSelectorBytecode
240-253	2	16	normal	sendLiteralSelectorBytecode
131	[0-7]	31	normal	singleExtendedSendBytecode
132	[0-31]	256	normal/super	doubleExtendedDoAnythingBytecode
133	[0-7]	31	super	singleExtendedSuperBytecode
134	[0-3]	64	normal	secondExtendSendBytecode

TAB. 4.1: Répartition des bytécodes associés à l'envoi de message.

Chaque code identifie un envoi de message normal ou super, avec un nombre fixé ou variable d'arguments, pouvant indexer les n premiers sélecteurs contenu dans un tableau, et déclenche l'exécution d'une primitive lorsqu'il est interprété. Cette primitive se charge du décodage du bytécode et de sa suite.

Certains bytécodes sont associés à un grand nombre d'opérations quelque fois très différentes, ce qui rend le décodage plus long, mais permet l'économie de bytécodes qui sont évidemment très précieux. C'est par exemple le cas du bytécode 132 dédié, entre autres, aux envois de messages normaux (voir figure 4.4).

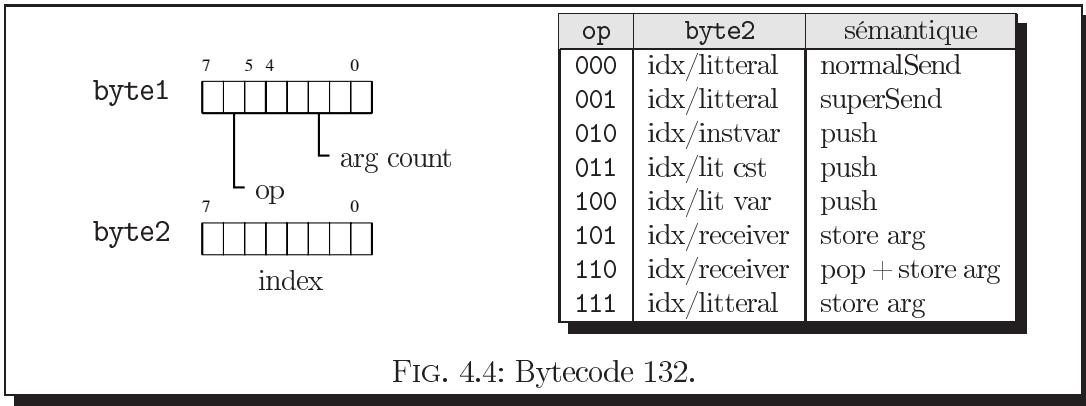


FIG. 4.4: Bytecode 132.

4.6.4 Les nouveaux bytecodes

La structure proposée pour les bytecodes d'envois de message avec point de vue (voir figure 4.5(c)), reprend celle des bytecodes 133 et 134 (voir figure 4.5(a)), en l'augmentant d'un second bytecode pour indexer les sélecteurs, comme dans le cas du bytecode 132 (voir figure 4.4).

Ils permettent donc d'exprimer des envois de messages possédant jusqu'à 8 arguments, sous 1 point de vue parmi 32 différents, et représentés par un sélecteur parmi 256 possibles.

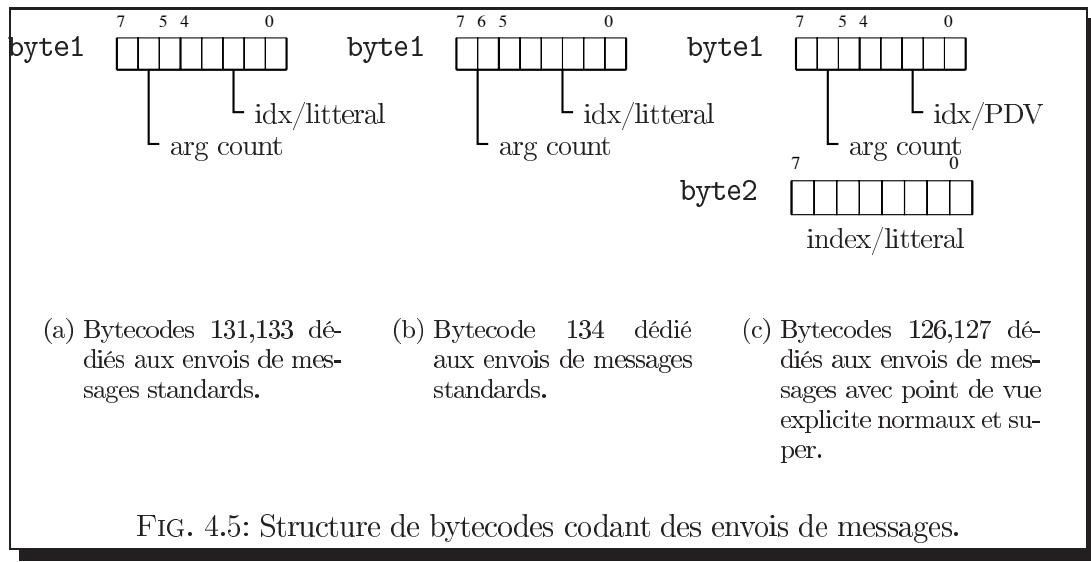


FIG. 4.5: Structure de bytecodes codant des envois de messages.

4.7 Modification du compilateur et de l'interpréteur

On dispose à présent de tout les éléments pour pouvoir créer des objets morcelés et leur envoyer des messages avec points de vue. Il reste cependant à faire le lien entre un objet morcelé, son état, et l'exécution de méthodes préalablement compilées, au moyen d'une syntaxe appropriée dont les grandes lignes ont été vues aux §3.1.3 et §3.1.4.

Avant toute chose, on doit être capable de compiler les méthodes au sein des morceaux des classes d'objets morcelés, et notamment:

- les accès aux variables de morceaux implicites et explicites, et
- les envois de message avec point de vue explicite.

Le code d'une méthode qui, par abus de langage, est plutôt un tableau de *bytecodes*, est la représentation en langage machine d'un texte informatique. Pour passer de la représentation textuelle à la représentation machine, on utilise un processus de compilation.

4.7.1 La chaîne de compilation

La compilation se déroule en deux phases qui constituent la chaîne de compilation:

1. L'analyse syntaxique, qui produit un arbre à partir d'un texte source, dans lequel chaque noeud représente une unité syntaxique du langage.
2. La génération de code réalise un parcours de l'arbre, pour générer le code des instructions associées à chaque noeud.

Smalltalk possède une chaîne de compilation totalement réifiée. Il est facile d'effectuer des modifications au niveau de l'analyseur syntaxique, ou du compilateur: chaque classe dispose d'une méthode `compilerClass` chargée de retourner la classe du compilateur qui a la charge de compiler ses méthodes.

On doit disposer d'au moins deux classes de compilateur:

1. Le compilateur standard `Compiler` que l'on doit modifier puisque n'importe quelle classe est susceptible d'utiliser des envois de message avec point de vue explicite, sur des objets morcelés, et qu'il faut être capable de compiler.
2. Le compilateur associé aux morceaux: `PDVCompiler` qui est sous-classe de `Compiler` et qui permet de compiler les accès aux variables avec point de vue implicite ou explicite, uniquement au sein des méthodes de morceaux.

<pre>Point>>rho ^((x*x + (y*y)) sqrt</pre>	<pre>1 <00> push inst 0 2 <00> push inst 0 3 <A8> send * 4 <01> push inst 1 5 <01> push inst 1 6 <A8> send * 7 <A0> send + 8 <E0> send sel 0</pre>	<p><code>inst 0</code> référence la première variable d'instance déclarée dans <code>Point</code> qui en déclare 2: <code>x</code> et <code>y</code>.</p> <p><code>sel 0</code> référence un sélecteur dans le contexte de la méthode <code>rho: sqrt</code>.</p>
--	--	---

FIG. 4.6: Compilation d'une méthode au sein d'une classe.

La FIG. 4.6 montre un exemple de compilation d'une méthode simple au sein d'une classe (normale) qui compte deux variables. On rappelle qu'un objet est représenté sous forme d'un bloc continu en mémoire, capable évidemment de contenir les valeurs de toutes les variables définies dans la classe de l'objet. La méthode `rho` réalise trois envois de message et quatre accès en lecture sur des variables de l'objet³:

³Il peut aussi y avoir des variables locales à la méthode.

- un envoi de message s'applique sur l'objet qui se trouve en haut de la pile au moment où l'interpréteur atteint le bytecode exprimant l'envoi de message. Les arguments de l'envoi de message se retirent également de la pile;
- on récupère la valeur d'une variable en accédant à la structure mémoire du receveur courant⁴. Cet accès s'effectue grâce à un indice qui indique l'emplacement de la valeur à l'intérieur de la structure de l'objet.

4.7.2 Le problème de l'accès aux variables d'instances

L'accès aux valeurs des variables se fait de la même façon pour un objet morcelé que pour un objet normal (voir 4.6.1): par l'intermédiaire d'un index que l'on peut voir comme une adresse relative à l'adresse mémoire de l'entête de l'objet, sous réserve que toutes les variables occupent la même place au sein de l'objet.

Le §3.1.4.1 nous a montré que pour les variables de morceaux, on devait tenir compte du point de vue du contexte d'exécution, avant de réaliser l'accès proprement dit. Ce qui signifie qu'un accès à une variable, au sein d'une méthode de morceau, n'est pas directement compilable et qu'il faut *lier dynamiquement* la variable au bon morceau. Les solutions pour résoudre ce problème sont les mêmes que celles qui sont mis en oeuvre pour compiler les envois de message dans les langages dynamiques [Duc97].

La solution retenue pour une première implémentation est une *compilation en ligne* basée sur l'utilisation de dictionnaires d'indices associés à chaque morceaux. On procède en plusieurs temps:

1. On calcule l'*ensemble* de tous les noms de variables définis pour une classe d'objets morcelés, au moment de sa création. Les modifications éventuelles altèrent bien entendu la liste. On associe à chaque nom, un indice en partant de 0. Cet indice code l'accès à la variable au moment de la compilation, quelque soit le point de vue que l'on considère. On peut maintenant compiler toutes les méthodes.
2. On établit pour chaque morceau, un *dictionnaire* contenant les indices réels (ceux qui indexent la position de la valeur dans la structure de l'objet) de chaque variable qu'il définit. Les clefs sont les indices de ces mêmes variables, calculés à la première étape.
3. On établit pour chaque méthode compilée, une *table* d'indices qui localise les emplacements, par rapport au tableau de bytecodes, des instructions qui réalisent des accès nécessitant la prise en compte d'un point de vue. Cette table servira de *table de relocation* pour corriger les indices au moment de l'exécution.

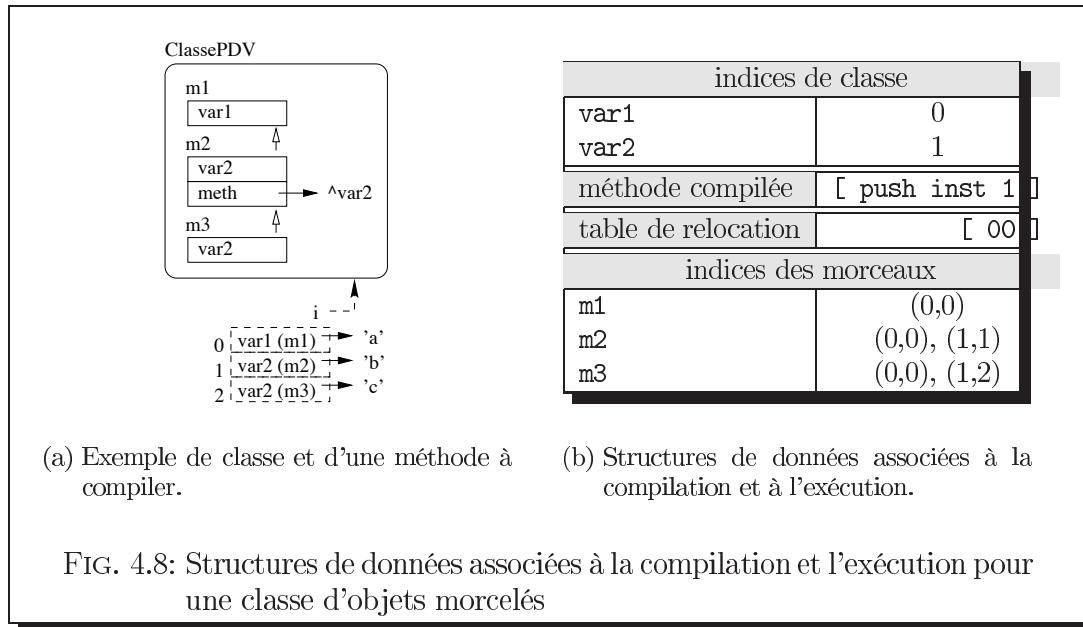
Au moment de l'exécution d'une méthode de morceau:

1. On sauvegarde le tableau de bytecodes original (on peut utiliser une technique de sauvegarde différentielle au fur et à mesure des modifications qui seront faites sur cette table).

⁴L'objet qui a reçu le message dont le code est en cours d'interprétation.

2. On inspecte la table de relocation. Pour chaque emplacement, on récupère l'indice qui a servi au moment de la compilation (c'est une sorte de processus de décompilation d'un type très simple), et on s'en sert comme d'une clef pour récupérer le bon indice, relativement au morceau qui représente le point de vue du contexte d'exécution. On replace cet indice dans la méthode.
3. On interprète les bytecodes du tableau modifié. On peut placer les modifications réalisées sur la méthode dans un cache, de sorte que l'on peut sauter directement à l'étape 3 si le point de vue n'a pas changé entre deux appels successifs.

La complexité du processus n'est pas linéaire à cause de l'utilisation de dictionnaires. On peut utiliser une matrice noms d'attributs \times noms de morceaux, pour stocker les indices, si l'on accepte un certain gaspillage mémoire au profit d'un accès en temps linéaire sur les indices. Le mieux serait d'utiliser une table compactée dont plusieurs algorithmes sont présenté dans [Duc97] et qui engendre une complexité, en terme d'utilisation mémoire, quasi optimale.



La classe `Encoder` joue un rôle particulier pendant la compilation. C'est une de ses instances qui est chargée, pendant la compilation, d'affecter des indexées aux noms de variables et noms de littéraux. La classe `PDVEncoder` en hérite et implémente l'encodage pour les classes d'objets morcelés. C'est elle qui encode les littéraux associés aux noms de points de vue pour pouvoir compiler les envois de message avec point de vue explicite (2^{nd} octet des bytecodes 126&127, voir 4.6.3).

4.7.3 Syntaxe associée aux envois de messages

On précise un point de vue entre accolades conformément à la syntaxe déjà utilisée pour les exemples qui ont illustré le Chap. 3 et conformément à la sémantique décrite au §3.1.3: on peut préciser un point de vue après l'identificateur d'un objet morcelé pour un envoi de message, ou après un nom de variable au sein d'une méthode de morceau.

L'analyse syntaxique est prise en charge par une instance de la classe `Parser`. Celle-ci est capable de générer un arbre syntaxique à partir du texte des méthodes à compiler dans lequel, chaque noeud est réifié, et dont la racine est instance de `MethodNode`. Pendant la génération de code chaque noeud est contacté pour générer le code qui lui correspond.

Tout comme le faisait `compilerClass`, une méthode `parserClass` permet d'associer une classe d'analyseur à n'importe quelle classe du système. `PDVParser` implémente donc l'analyse syntaxique sur les méthodes des classes d'objets morcelés et génère des noeuds syntaxiques de types nouveaux:

`PDVMessageNode` sous-classe de `MessageNode` qui permet de contenir toutes les informations sur l'envoi de message avec point de vue, avec sélecteur et arguments.

`PDVVariableNode` sous-classe de `VariableNode` qui permet de traiter les variables avec point de vue explicite.

5

Conclusion

C'est avant tout la *réactivité* d'un système objet qui permet d'étendre son champ d'utilisation, en faisant passer son statut de simple curiosité informatique à un outil capable d'exprimer plus facilement des problèmes concrets, et nombreuses sont les études [Duc97, USCH92, HU94] qui témoignent de l'activité dans le domaine des optimisations sur les langages objets.

Les objets morcelés sont nés d'un besoin simple de considérer des entités sous plusieurs point de vue, et posent leur problématique principalement sur un plan conceptuel, sans rien enlever aux classiques confrontations entre système à classes et système à prototypes, qui peuvent tous deux, accueillir les objets morcelés. On peut donc dire que des champs d'applications spécifiques existent pour les objets avec points de vue, et qu'ils concernent directement les objets morcelés. Reste à en développer une implémentation *réaliste*, vis-à-vis de besoins, et *efficace*, vis-à-vis de nos moyens.

Notre contribution a été de montrer, par une approche originale qui partait d'une abstraction de la notion de vue, les aptitudes des objets morcelés à représenter des entités avec points de vue. Nous avons également « *déblayer* » quelque peu le terrain des solutions qui avaient été proposées pour les classes d'objets morcelés, en posant dès que possible les questions du pouvoir expressif, de la cohérence ou de complexité, associés à telle ou telle solution. Enfin, l'étude d'implémentation en Smalltalk a permis de mettre le doigt sur les points sensibles, où se jouent les performances du système, et qui n'avaient pas été abordés dans les parties précédentes. Cette étude constitue, à notre avis, une bonne base pour une implémentation sérieuse.

La solution des classes d'objets morcelés n'a encore jamais été implémentée et les perspectives d'améliorations, d'optimisations, et d'études spécifiques sont certes nombreuses pour palier les défauts (de jeunesse) de l'implémentation:

- utilisation d'un *nouveau type de noyau* autorisant la création explicite et implicite des métaclasses [Riv96],
- optimisation des accès aux variables d'instances en utilisant une méthode de compactage de table [Duc97],
- étude de technique de cache à deux niveaux (sur les classes et les morceaux ou les points de vue, parallèlement à l'étude d'optimisation),
- technique de *garbage collector* spécifique aux objets morcelés (pointeurs mous comme en JAVA),

- étude de méthodes de conception à base de point de vue: *Designs Patterns* intégrant les points de vue, etc...

Les avantages que procurent une organisation réflexive des objets n'est plus à démontrer, et sans être arrivée à en exhiber une convenable, la §3.2.1.3 laisse tout de même entrevoir qu'une telle organisation est possible sur des objets morcelés, mais dans un système hybride classes-prototypes. Des hiérarchies de prototypes, organisées en ensemble de représentations éclatées, pourraient servir de métaréprésentations pour les classes et métaclasses d'objets morcelés (de manière à résoudre le problème de l'auto-instanciation multiple). L'idée mérite certainement d'être approfondie.

De nombreuses portes sont maintenant ouvertes sur toutes sortes d'investigations pour le moins passionnantes...

Annexe A

Objets: Composition inter-niveaux

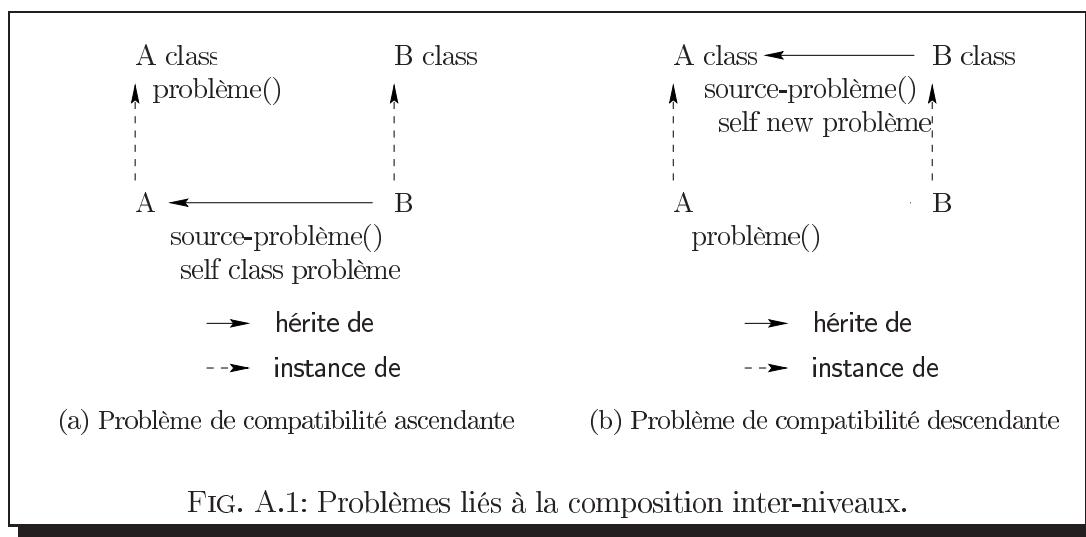
Les instances, les classes, les métaclasses, organisent une application en plusieurs niveaux d'abstraction. Chaque entité d'un niveau est relié à l'entité qui la contrôle par un lien d'instanciation, située au niveau supérieur. Le nombre de niveau dépend de l'organisation du noyau du système considéré (voir 3.2.1.1).

Le langage met à la disposition du programmeur, des messages qui permettent de monter ou descendre dans la hiérarchie d'instanciation: on parle de *composition inter-niveaux*, lorsqu'on les utilise. En Smalltalk par exemple:

`new` retourne une instance du receveur, qui nécessairement se comporte comme une classe, et permet donc de descendre dans la hiérarchie.

`class` retourne la classe du receveur, et permet de monter dans la hiérarchie.

Un certain nombre de problèmes, parfaitement classifiés, peuvent intervenir lorsque l'on n'y prend pas garde. C'est pourquoi quelques langages, comme Smalltalk, tendent y remédier, en assurant de manière automatique, ce que l'on nomme la compatibilité ascendante et descendante (voir figure A.1).



A.1 Compatibilité ascendante

Sur la FIG. A.1(a), la classe B hérite de la classe A, donc la méthode **source-problème** peut être appliquée à toutes instances de B. C'est à ce niveau qu'apparaît l'incompatibilité ascendante, car cette méthode envoie le message **problème** à la métaclass de la classe du receveur, que l'on atteint avec **self class**. Si le receveur en question est la classe B, sa métaclass B **class** ne sait pas y répondre; on obtient alors une erreur à l'exécution.

Ce problème est désigné sous le nom d'incompatibilité ascendante [Gra89, BSLR96].

La compatibilité ascendante entre A class et B class est vérifiée si toutes les hypothèses faites par A sur le comportement de sa classe A class sont valides pour B class: métaclass de B avec B sous-classe de A.

A.2 Compatibilité descendante

Sur la FIG. A.1(b), la classe B **class** hérite de la classe A **class**, donc la méthode **source-problème** peut être appliquée à toutes instances de B **class**, autrement dit B elle-même. C'est à ce niveau qu'apparaît l'incompatibilité descendante, car cette méthode envoie le message **problème** à une instance du receveur, créé pour l'occasion par **self new**. Si le receveur en question est la classe B **class**, cette instance ne sait pas y répondre; on obtient une erreur à l'exécution.

Ce problème est désigné sous le nom d'incompatibilité descendante [BSLR96].

La compatibilité descendante entre A et B est vérifiée si toutes les hypothèses faites par A class sur le comportement de ses instances (du type de A), sont valides pour les instances de B class (du type de B), avec B class sous-classe de A class.

Liste des figures

2.1	La modification de la valeur d'un slot du père affecte tous les fils qui ne l'ont pas redéfini.	10
2.2	Deux interprétations possibles de l'affectation dans un langage à prototypes implémentant la délégation.	10
2.2(a)	La demande de modification de la valeur de l'attribut \texttt{tattr1} au niveau du fils, répercute la modification au niveau du père.	10
2.2(b)	La demande de modification de la valeur de l'attribut \texttt{tattr1} au niveau du fils, entraîne la redéfinition de \texttt{tattr1} dans ce dernier.	10
2.3	Représentation d'une hiérarchie de vues, et de leur contenu.	11
2.4	Représentation éclatée d'une entité Pierre	13
3.1	Objets morcelés représentés par des classes à morceaux partagés.	19
3.2	Objets morcelés représentés par une classe à morceaux privés.	20
3.3	Exemples d'objets morcelés sur lesquels un envoi de message avec point de vue combiné peut aboutir à des résultats différents ou à des ambiguïtés suivant la stratégie choisie.	24
3.3(a)	Non ambigu, mais les résultats diffèrent suivant la stratégie.	24
3.3(b)	Ambigu pour la première stratégie, non ambigu pour la deuxième.	24
3.3(c)	Ambigu pour les deux stratégies.	24
3.4	Point de vue et accès aux variables de morceau. meth doit retourner la valeur de var suivant le point de vue courant	25
3.5	Ambiguïté sur l'accès aux variables de morceau.	26
3.6	Exemples d'objets morcelés montrant la difficulté d'organisation des programmes avec la stratégie du point de vue le plus spécifique.	27
3.6(a)	Absence de lien entre la méthode et la variable sélectionnée le choix de la variable.	27
3.6(b)	Possible non conformité entre le choix de la variable \texttt{var} et la sémantique associée à la méthode \texttt{meth} dans la première stratégie.	27
3.7	Possible non conformité entre le choix de la variable var et la sémantique associée à la méthode meth dans la seconde stratégie.	27
3.8	Organisation des classes dans un système du type ObjVLisp.	29
3.8(a)	Organisation des classes du noyau de ObjVLisp.	29
3.8(b)	Organisation d'une classe d'objets morcelés autour du noyau.	29
3.9	Méta-classe associée aux classes d'objets morcelés.	31
3.10	Organisation réflexive des objets morcelés.	32
3.11	Organisation du noyau de Smalltalk-80.	32

3.12 Hiérarchie de classes ne permettant pas d'héritage entre une classe d'objets morcelés et une classe normale.	34
3.13 Hiérarchie de classes permettant l'héritage entre une classe d'objets morcelés et une classe normale.	35
3.14 Représentation à l'aide des objets morcelés dans un système à classe, d'une partie d'un domaine d'application.	36
4.1 Environnement de programmation de Squeak.	41
4.2 Format de l'entête de base d'un objet.	43
4.3 Variables associées aux objets représentant des classes.	43
4.3(a) Structure minimale d'un objet se comportant en classe. Les variables sont déclarées dans différentes classes du noyau.	43
4.3(b) Le champ \texttt{format} sert à remplir rapidement l'entête de base des objets instanciés par la classe.	43
4.4 Bytecode 132.	48
4.5 Structure de bytecodes codant des envois de messages.	48
4.5(a) Bytecodes 131,133 dédiés aux envois de messages standards.	48
4.5(b) Bytecode 134 dédié aux envois de messages standards.	48
4.5(c) Bytecodes 126,127 dédiés aux envois de messages avec point de vue explicite normaux et super.	48
4.6 Compilation d'une méthode au sein d'une classe.	49
4.8 Structures de données associées à la compilation et l'exécution pour une classe d'objets morcelés	51
4.7(a) Exemple de classe et d'une méthode à compiler.	51
4.7(b) Structures de données associées à la compilation et à l'exécution.	51
A.1 Problèmes liés à la composition inter-niveaux.	55
A.1(a) Problème de compatibilité ascendante	55
A.1(b) Problème de compatibilité descendante	55

Bibliographie

- [AR92] E. ANDERSEN et T. REENSKAUG. «*System Design by Composing Structures od Interacting Objects*». Dans O. MADSEN, rédacteur, *ECOOP'92 conference proceedings*, pp. 133–152 (Springer-Verlag, 1992).
- [Bar98] D. BARDOU. *Étude des langages à prototypes, du mécanisme de délégation, et de son rapport à la notion de point de vue*. Thèse de doctorat, UNIVERSITÉ DE MONTPELLIER II, avril 1998.
- [BD98] Z. BELLAHSÈNE et R. DUCOURNAU. «*Vues et points de vue*», 1998.
- [BS83] D. G. BOBROW et M. STEFIK. *The LOOPS Manual*. Xerox Palo Alto Research Center, 1983.
- [BSLR96] N. BOURAQADI-SAÂDANI, T. LEDAUX, et F. RIVARD. «*Metaclasses Composability*». Dans *ECOOP'96 Workshop*, juillet 1996.
- [CDG90] B. CARRÉ, L. DEKKER, et J. M. GEIB. «*Multiple and Evolutive Representation in the ROME Language*». Dans *TOOLS2*, pp. 101–109, 1990.
- [Coi87] P. COINTE. «*Metaclasses are First Class: The ObjVlisp Model*». Dans *OOPSLA '87 conference proceedings*, pp. 156–167. Rank Xerox & LITP (ACM Sigplan Notices, 1987).
- [Coi90] P. COINTE. «*The ClassTalk System: a Laboratory to Study Reflection in SmallTalk*». Dans *Informal Proceedings of the First Workshop on Reflection and Meta-Level Architectures in Object-Oriented Programming*. OOPSLA/ECOOP'90, 1990.
- [CU91] C. CHAMBERS et D. UNGAR. «*Making pure Object-Oriented languages practical*». Dans *OOPSLA'91 conference proceedings*, pp. 1–15 (ACM Sigplan Notices, 1991).
- [DB98] Ch. DONY et D. BARDOU. «*Les langages à prototypes*». Dans R. DUCOURNAU, J. EUZENAT, G. MASINI, et A. NAPOLI, rédacteurs, *Langages et modèles à objets, État des recherches et perspectives*, Collection didactique, chapitre 8 (INRIA, 1998).
- [DMC92] Ch. DONY, J. MALENFANT, et P. COINTE. «*Prototype-Based Languages: From a new Taxonomy to Constructive Proposals and their Validation*». Dans A. PAEPCKE, rédacteur, *OOPSLA'92 conference proceedings*, pp. 201–217 (ACM Sigplan notices, 1992).

- [Don97] Ch. DONY. «*A framework for the design and operational evaluation of prototype-based languages*». *Rapport technique 97254*, LIRMM, 1997.
- [Duc97] R. DUCOURNAU. «*La compilation de l'envoi de message dans les langages dynamiques*». Dans *L'objet*, tome 3, pp. 241–276 (1997).
- [GR83] A. GOLDBERG et D. ROBSON. *Smalltalk-80, The Language and its Implementation* (Addison-Wesley Publishing Company, 1983).
- [Gra89] N. GRAUBE. «*Metaclass compatibility*». Dans *OOPSLA'89 conference proceedings*, pp. 305–315, octobre 1989.
- [HO93] W. HARRISON et H. OSSHER. «*Subject-Oriented Programming (A Critic of Pure Objects)*». Dans A. PAEPCKE, rédacteur, *OOPSLA'93 conference proceedings*, pp. 411–428 (ACM sigplan notices, 1993).
- [HU94] U. HÖLZLE et D. UNGAR. «*A Third Generation SELF Implementation: Reconciling Responsiveness with Performance*». Dans *OOPSLA'94 conference proceedings* (ACM Sigplan Notices, 1994).
- [IKM⁺96] D. INGALLS, T. KAEHLER, J. MALONEY, *et al.*. «*Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself*». *Rapport technique*, Apple Computer & Walt Disney Imagineering, 1996.
- [INR95] INRIA Rhône-Alpes. *Tropes 1.0 reference manual*, imag-lifia grenoble édition, 1995.
- [Per98] J.-F. PERROT. «*Objets, classes, héritage: définition*». Dans R. DUCOURNAU, J. EUZENAT, G. MASINI, et A. NAPOLI, rédacteurs, *Langages et Modèles à objets, État des recherches et perspectives*, Collection didactique, chapitre 1 (INRIA, 1998).
- [Pop98] S. T. POPE. «*The Do-It-Yourself Guide to Squeak primitives*». *Rapport technique*, CREATE, avril 1998.
- [Pro95] Projet VERSO - INRIA Rocquencourt, Le Chesnay, France. *O2Views User Manual*, version 2 édition, 1995.
- [Riv96] F. RIVARD. «*A New Smalltalk Kernel Allowing Both Explicit And Implicit Metaclass Programming*». Dans *OOPSLA'96 conference proceedings*, 1996.
- [Riv97] F. RIVARD. *Évolution du comportement des objets dans les langages à classes réflexifs*. Thèse de doctorat, UNIVERSITÉ DE NANTES, juin 1997.
- [Run94] E. RUNDERSTEIN. «*A Classification Algorithm for Supporting Object-Oriented Views*». Dans *CIKM'94 conference proceedings*, pp. 18–25 (ACM Press, Gaithersburg, Maryland, 1994).
- [SLR⁺92] M. SCHOLL, C. LAASCH, C. RICH, *et al.*. «*The COCOON Object Model*». *Rapport technique 192*, ETH Zurich, Switzerland, 1992.
- [Squ] Squeak documentation. *Pluggable Primitives*.

- [US91] D. UNGAR et R. B. SMITH. «*Self: The Power of Simplicity*». Dans N. K. MEYROWITZ, rédacteur, *OOPSLA'87 conference proceedings*, pp. 187–206 (ACM Sigplan Notices, 1991).
- [USCH92] D. UNGAR, R. B. SMITH, C. CHAMBERS, *et al.*. «*Object, Message, and Performance: How they coexist in SELF*». *IEEE Computer*, tome 25(10), octobre 1992.
- [Van94] G. VANWORMHOUDT. «*Points de vue, représentation multiple et évolutive en Smalltalk*». *Rapport technique*, LIFL, mars 1994.