ClassTalk : Une transposition des métaclasses ObjVlisp à Smalltak-80

Pierre Cointe Jean-Pierre Briot

Equipe mixte Rank Xerox France - LITP,
Université Pierre et Marie Curie,
4 place Jussieu, 75005 PARIS, France
cointe/briot@rxf.ibp.fr

Résumé: Ce papier traite de l'implémentation du modèle ObjVlisp en Smalltalk-80. Nous décrivons précisement la transposition des métaclasses ObjVlisp à Smalltalk. Le résultat obtenu est la plateforme ClassTalk, support à l'élaboration d'une méthodologie de la programmation par (méta)classes. Nous détaillons le flâneur ClassTalk dédié à la manipulation de (méta)classes explicites dans l'environnement Smalltalk. Nous proposons une première bibliothèque de métaclasses choisies pour leur généralité et conçues pour leur réutilisabilité. Nous reprenons les résultats de Ingalls & Borning pour expérimenter la combinaison des métaclasses par héritage multiple. Nous concluons sur les limites d'un architecture de métaclasses explicite non couplé à un mécanisme de création uniforme et également explicite.

Mots Clefs: objet, classe, métaclasse, ObjVlisp, Smalltalk-80, CLOS, instanciation, héritage, uniformité, rendre explicite, noyau, integration, bibliothèque, compatibilité, flexibilité, enseignement, apprentissage.

Abstract: This paper discusses the introduction of explicit metaclasses à la ObjVlisp into the Smalltalk-80 language. The rigidity of Smalltalk metaclass architecture motivated this work. We decided to implement the ObjVlisp model into the standard Smalltalk-80 system. The resulting combination defines the Classtalk platform. This platform provides a full-size environment to experiment with class-oriented programming by combining implicit metaclasses à la Smalltalk and explicit metaclasses à la ObjVlisp. Obviously, these experiments are not limited to the Smalltalk world and will be useful to understand and practice the metaclass concept advocated by modern object-oriented languages such as ObjVlisp and CLOS.

Keywords: objet, class, metaclass, ObjVlisp, Smalltalk-80, CLOS, instantiation, inheritance, uniformity, to make explicit, kernel, integration, library, compatibility, flexibility, to teach, to learn.

1 Introduction

Ce papier est au carrefour de deux préoccupations: la programmation par objets et les langages pour l'intelligence artificielle.

Dans le cadre de la programmation par objets en général, et de la programmation par classes en particulier, nous souhaitons promouvoir la programmation par métaclasses au travers du modèle ObjVlisp tel qu'il a été présenté à RFIA'85, ECAI'86 et IJCAI'87.

Dans le cadre des architectures logicielles pour l'IA, nous sommes convaincus que Smalltalk-80 devrait former avec Lisp et Prolog l'un des trois langages majeurs pour l'écriture et le développement de nouveaux logiciels. Nous souhaitons montrer que Smalltalk-80 demeure le système (à objets) le plus complet, le plus riche, le plus flexible, le plus extensible.

1.1 Les métaclasses comme outils d'implémentation

La première préoccupation nous a d'abord conduits à explorer les métaclasses comme outils d'implémentation des systèmes à objets. Nous avons développé le modèle ObjVlisp [3], [5] qui propose une architecture de (méta)classes simple, minimale et générale. Nous avons montré comment enrichir ce noyau minimal par l'adjonction de métaclasses décrivant de nouveaux mécanismes tels la combinaison des méthodes et l'héritage multiple traité par linéarisation du graphe d'héritage. Le résultat [11], [6] est un système en couches, auto-défini implémentant une version réduite du "Common Lisp Object System" (CLOS), i.e. le système objet pour CommonLisp normalisé par l'ANSI [9]. CLOS repose aussi sur une architecture de métaclasses le représentant lui-même comme un ensemble d'objets. Ces métaclasses auto-d'écrivent le langage, elles sont spécialement conçues pour permettre aux implémenteurs de modéliser des sous-systèmes intégrant de nouvelles constructions sémantiques [1].

1.2 Les métaclasses comme outils de développement

Cette première préoccupation nous a ensuite conduit à nous interroger sur l'emploi des métaclasses du point de vue de "l'utilisateur lambda". A cet effet, nous avons souhaité mettre en place les premiers éléments d'une méthodologie de la programmation par (méta)classes. Cette méthodologie doit s'appliquer à tout système traitant les classes comme des objets à part entières. S'est alors posé le choix du langage d'expérimentation, le critère de sélection étant la richesse des bibliothèques de (méta)classes disponibles.

De ce point de vue ObjVlisp souffre de sa minimalité. Pour sa part, et du fait de son statut de futur standard industriel, CLOS a pour vocation de proposer des bibliothèques de classes plus riches. Néanmoins ce langage est encore en cours de gestation, il n'existe actuellement que bien peu d'implémentations et son environnement de programmation est inexistant. Enfin Smalltalk-80, l'archétype du langage à objets et l'inventeur du concept de métaclasse, utilise bien celles-ci pour ses besoins propres mais ne fait rien pour promouvoir leur emploi auprès de ses utilisateurs. Smalltalk-80 bénéficie cependant d'un environnement de programmation inégalé construit à partir de bibliothèques d'une grande valeur. Pour cette raison, et pour son extensibité, nous l'avons retenu comme base de notre expérimentation.

Ce choix répond également à notre deuxième préoccupation: promotion du langage Smalltalk-80 dans la communauté de l'IA. Pour ce faire, il nous a semblé important de clarifier ses choix sémantiques et d'étudier dès à présent son aptitude (et celles de ses métaclasses) à intégrer de nouveaux formalismes adaptés aux problèmes de la représentation des connaissances [15].

1.3 Les métaclasses Smalltalk-80

"Motivated primarily by the desire to have class-specific creation messages that did not require making up a "blank" instance and the initializing it, metaclasses have proven confusing to many users and perhaps in the balance more confusing than valuable" [8].

En fait, tant au niveau des choix terminologiques (méthodes-variables d'instances, méthodes-variables de classes), qu'à celui des choix ergonomiques qui ont présidé à l'élaboration du flâneur (une métaclasse est anonyme et n'apparait que par sélection d'une vue d'échange) tout est fait par Smalltalk pour éluder (cacher) le problème (la présence) des métaclasses.

De ce fait, plusieurs experts travaillant dans le domaine de l'apprentissage de la programmation par objets en général et de Smalltalk en particulier clament que le concept de métaclasse complexifie inutilement le modèle objet et réclament soit sa clarification soit sa suppression pure et simple [2].

Pourtant, l'étude des arcanes du système Smalltalk "révèle" que si l'architecture du noyau Smalltalk-80 limite volontairement l'usage des métaclasses elle demeure suffisamment extensible pour supporter un autre modèle de métaclasses [7]. Nous avons donc entrepris de transposer l'architecture des métaclasses ObjVlisp à Smalltalk.

1.4 Plan de notre étude

Ce texte traite successivement des limitations introduites par le couplage classe-métaclasse qui introduit une discontinuité dans le mécanisme d'instanciation Smalltalk (2). Il rappelle ensuite l'avantage des métaclasses explicites en décrivant les architectures modernes à la ObjVlisp et à la CLOS (3). Il discute alors l'introduction de l'architecture des métaclasses ObjVlisp dans le langage Smalltalk et décrit deux implémentations dites par "greffon" et par "fusion" (4). Il montre comment spécialiser le flâneur Smalltak pour l'adapter à la manipulation explicite des métaclasses ClassTalk (5). Il présente et commente une première bibliothèque de métaclasses ClassTalk (6). Il décrit l'héritage multiple Smalltalk-80 et sa version ClassTalk (7). A partir d'un exemple développé, il expose comment composer les métaclasses entre elles par instanciation et héritage (8). Il montre les limites d'une architecture de métaclasses non couplée à un mécanisme de création unifié et introduit les problèmes de compatibilité intervenant lors d'héritage "mixte" entre classes Smalltalk et classes ClassTalk.

2 Les arcanes de l'architecture Smalltalk-80

Le système Smalltalk utilise deux types de métaclasses. Les métaclasses que nous appellerons "primitives" qui sont réservées aux implémenteurs pour élaborer le noyau (elles figurent dans la catégorie `Kernel-Classes` du flâneur) et les métaclasses standards qui définissent l'image standard et les extensions utilisateurs.

2.1 Les métaclasses primitives

Il est en fait méconnu que que Smalltalk utilise au niveau le plus interne les métaclasses `Class` et `Metaclass` pour construire un système de classes auto-décrit et extensible, assez similaire à celui de langages plus modernes comme ObjVlisp, CLOS et Lore. Le rôle de ces deux métaclasses est de définir la structure et le comportement des objets "de type classes" et les objets "de type métaclasses". Ces classes sont toutes deux sousclasses de `ClassDescription` et `Behavior` qui définissent donc les structures et les comportements communs aux classes et métaclasses standards.

2.2 Les métaclasses standards

Cette méconnaissance des métaclasses primitives est dûe au choix des designers Smalltalk relatif au status des métaclasses standards. Ce choix consiste à réduire l'usage de celles-ci en associant à chaque classe standard un module implicite classe-métaclasse gérant par défaut la métaclasse. Cette approche réductrice vise à cacher au débutant le système de métaclasses tout en réservant un

usage limité de celui-ci au programmeur confirmé. Cette décision facilite évidemment les premiers pas du débutant mais rend également sa compréhension globale du système très difficile. Enfin cette décision complexifie inutilement la sémantique du langage donc son apprentissage et son enseignement.

Les métaclasses standards - contrairement aux classes - sont anonymes et créées implicitement par le système. Quand l'utilisateur définit une classe par l'intermédiaire du flâneur (par exemple la classe `Thinglab`), le système crée également automatiquement une nouvelle métaclasse (la métaclasse `Thinglab class`) qui est couplée avec son unique instance. Cette métaclasse est anonyme et ne peut être accédée que par l'intermédiaire de son unique instance: le message `class` envoyé à une classe retourne un "pointeur" sur sa métaclasse privée.

Au moment de la définition d'une classe, le flâneur se charge du couplage classe-métaclasse en créant successivement la métaclasse et sa classe, puis en liant physiquement la première à la seconde [7]. Cette métaclasse décrit normalement la structure et le comportement de sa classe par l'intermédiaire de ses variables d'instances et de ses méthodes. La sous-vue d'échange ("SwitchView") *instance/class* permet de "sauter" de la définition de la classe à celle de sa métaclasse privée. Comme dans le cas d'une classe, l'utilisateur peut définir des méthodes au niveau de la métaclasse. Le plus souvent il s'agit de messages d'initialisation (`initialize`), de création explicite (variante sur le `new`) ou d'exemplification (`example`). Toujours au niveau de la métaclasse, il est possible de définir des variables d'instances pour enrichir la structure des classes standards. Néanmoins l'usage de ces variables d'instances de métaclasses (à ne pas confondre avec les variables dites de classes [5]) demeure exceptionnel[1].

Les métaclasses standards étant créées implicitement, leurs règles d'héritage doivent obligatoirement obéir à un défaut. La règle d'héritage implicite est la suivante: la hiérarchie d'héritage des métaclasses est parallèle à celle des classes. Smalltalk connecte ces deux hiérarchies en définissant la métaclasse la plus générale, `Object class`, comme une sousclasse de la métaclasse primitive `Class`.

Du point de vue de l'instanciation, chaque métaclasse standard est obligatoirement définie comme une instance de la métaclasse primitive `Metaclass`. Toutes ses métaclasses partageront donc la même stucture et le même comportement.

La figure suivante résume la hiérarchie d'héritage des classes et des métaclasses Smalltalk-80. Chaque nom de classe préfixe l'ensemble de ses variables (méthodes) d'instances délimité par une paire de () (<>), les métaclasses primitives sont indiquées en italique :

```
Object () <...>
  Thinglab () <...>
  Behavior (superClass methodDict format subclasses) <... new ...>
    ClassDescription (instanceVariables organization) <...>
      Metaclass (thisClass) <... new ...>
      Class (name classPool sharedPools) <...>
        Object class () <...>
          Thinglab class () <...>
          Behavior class () <...>
            ClassDescription class () <...>
              Metaclass class () <...>
              Class class () <...>
```

L'exemple suivant témoigne des limites du couplage classes-metaclasses standards et des contraintes posées par l'introduction d'une règle d'héritage implicite pour les métaclasses.

L'exemple des classes abstraites

"Abstract class: a class that specifies protocol, but it not able to fully implement it; by convention instances are not created of this kind of classes" [10].

[1] A tel point qu'elles n'interviennent même pas dans la terminologie du langage!

Le terme *classe abstraite* est utilisé dans la littérature pour désigner des classes définies dans la hierarchie d'héritage dans le seul but de spécifier un comportement commun à une ensemble de classes. Ces classes sont donc destinées à être héritées mais pas à être instanciées. En Smalltalk, leur instanciation est pourtant possible même si par construction les objets générés sont forcement incomplets. Nous souhaitons définir une métaclasse modélisant des classes abstraites REELLEMENT non instanciables. Pour ce faire nous devons leur interdire l'accès à la méthode de création standard.

Un exemple de classe abstraite est celui qui intervient lors de la modélisation des nombres complexes. Deux modèles de représentations sont utilisés, le modèle (des coordonnées) Cartesian et le modèle Polaire. Il est donc normal de définir deux classes, respectivement `Cartesian` et `Polar` rendant compte de ces deux aspects d'un nombre complexe. Il est également utile de faire intervenir une troisième classe abstraite (`Complex`) exprimant le comportement des complexes indépendamment de leur modèle de représentation. `Cartesian` et `Polar` sont alors définies comme deux sousclasses de `Complex` :

```
Object () <   ...   >
  Complex () <+ - * ...conjugate negated>
    Cartesian (x y) <x y rho theta  ...  printOn:>
    Polar (rho theta) <x y rho theta  ...  printOn:>
```

Pour exprimer le fait que la classe `Complex` est abstraite Smalltalk ne propose qu'un mécanisme: celui de l'héritage. Nous devons obligatoirement définir la métaclasse de `Complex` comme sous-classe d'une métaclasse (par exemple `AbstractClass class`) redéfinissant la méthode standard de création. Pour ce faire, les métaclasses étant créer implicitement, il nous faut d'abord définir la classe "vide" `AbstractClass` puis sa sousclasse `Complex`. A ce moment là seulement, nous pouvons redéfinir les méthodes de classes `new` et `new:` pour qu'elle signalent une erreur. Nous sommes donc contraint à la hiérachie suivante:

```
Object < ... >
  AbstractClass <>
    Complex < ... >
      Cartesian < ... >
      Polar < ... >
  Behavior < ... new new: ... >
    ClassDescription < ... >
      Class < ... >
        Object class < ... >
          AbstractClass class <new new:>
            Complex class <>
              Cartesian class <>
              Polar class <>
```

Cet exemple est bien évidemment un contre exemple à l'héritage parallèle classe-métaclasse proposé par Smalltalk. En effet si `Complex` est bien une classe abstraite `Cartesian` et `Polar` le sont également car leurs métaclasses héritent aussi - par l'intermédiaire de `Complex class` - des nouvelles définitions de `new` et `new:`. La règle implicite retenue pour l'héritage des métaclasses va à l'encontre de notre intuition !!. Tout le problème viens du fait que Smalltalk ne permet pas de définir explicitement une métaclasse unique modélisant l'ensemble des classes abstraites. La solution consistera évidemment à "casser" les modules classes, métaclasses privées.

2.3 Non uniformité du protocole de création

Nous distingons dans le processus de création d'un objet deux étapes; l'étape d'allocation et l'étape d'initialisation. Nous dirons que le processus de création est uniforme s'il permet d'expliciter au niveau du langage même la composition d'un allocateur unique avec différentes formes d'initialisateurs.

Dans le cas de Smalltak-80, il existe deux types d'objets, les objets à taille fixe dont la structure est définie par un ensemble de variables d'instances et les objets à taille variable (chaînes, tableaux, ...) dont la structure est définie par une dimension et un index. A ces deux types d'objets correspondent deux allocateurs primitifs représentés par les deux méthodes `new` et `new:` (et leurs synonymes `basicNew` et `basicNew:`) définies dans la classe `Behavior`. Les différents objets du système sont tous alloués par appel de l'un de ces allocateurs. En particulier l'allocation des classes et des métaclasses utilise toujours le `new` de `Behavior` [7].

Cependant si le processus d'allocation est explicite (i.e décrit dans `Behavior`) le processus d'initialisation ne l'est pas (en particulier la classe `Object` ne prévoit pas "d'initialisateur" standard). Il est donc impossible de parler d'un processus de création explicite et uniforme en Smalltalk.

Le langage, l'environnement et la méthodologie Smalltalk conduisent à distinguer création des classes et création des objets terminaux. En ce qui concerne les classes, elles partagent toutes la même structure définie par la classe `Class`. Smalltalk propose donc une méthode de création commune à toutes les classes standards. Cette méthode nommée (`subclass:instanceVariablesNames:...:category:`) sert de patron à la définition d'une classe dans le flâneur. Elle est propriété de la classe `Class`, et permet d'initialiser certaines variables d'instances de la nouvelle classe.

En ce qui concerne les objets terminaux, leur structure est définie par leur classe. Smalltalk suggère d'utiliser le couplage classe-métaclasse, pour expliciter au niveau de chaque module la création des instances d'une même classe. A cet effet on définira une méthode de classe composant l'allocateur standard avec une méthode d'instance chargée de l'initialisation[2]. La méthode de classe `x:y:` permet de créer de nouveaux `Cartesien` dont les valeurs du `x` et du `y` sont explicites. Elle compose l'allocateur standard ("super new") avec la méthode d'instance `setX:setY:` dédiée à l'initialisation des `Cartesien` :

```
Cartesian methodsFor: 'initializing'
setX: xValue setY: yValue
   x := xValue. y := yValue

Cartesian class methodsFor: 'creation'
x: xValue y: yValue
   ^(self new) setX: xValue setY: yValue
```

Une autre approche proposée par ObjVlisp et développée en 4.2 consiste à remarquer que la structure de tout objet est connue (elle est déterminée par sa classe). Il est donc possible de prévoir un protocole général (défini dans la classe `Object`) chargé d'initialiser cette structure.

3 L'alternative ObjVlisp

Une solution aux limitations Smalltalk est le modèle ObjVlisp [3] [5]. Ce modèle unifie les concepts de classe et de métaclasse, une métaclasse étant traitée comme une classe à part entière. De ce fait le couplage strict classe-métaclasse n'a pas lieu d'être et différentes classes peuvent partager la même métaclasse. La création d'un objet s'effectue toujours par un message envoyé à sa classe. Ce message permet d'expliciter les valeurs d'initialisation des différentes variables d'instances.

ObjVlisp représente le modèle minimal dans la mesure où deux classes seulement servent à le construire et à l'auto-définir. Ces deux classes définissent respectivement la racine de l'arbre d'instanciation (`Class`) et la racine de l'arbre d'héritage (`Object`). En tant qu'objets, les classes `Object` et `Class` sont instances de la première (méta)classe `Class`. `Class` est donc sa propre

[2]Il serait possible d'associer automatiquement à chaque module classe-métaclasse ces deux méthodes d'initialisation et de création. Les sélecteurs correspondant étant obtenus par concaténation des noms variables d'instances.

instance, l'expression de cette propriété au niveau d'ObjVlisp même lui confère ses caractères d'uniformité et de réflexivité. Le système est également extensible, une nouvelle métaclasse étant simplement créée par héritage d'une métaclasse existante. En conséquence, la profondeur de l'arbre d'instanciation est virtuellement infinie.

3.1 Uniformité et flexibilité des métaclasses ObjVlisp

En ObjVlisp une métaclasse est une classe ayant accès à l'allocateur standard soit directement parce qu'elle le détiens soit indirectement par héritage. `Class`, en tant que propriétaire de cet allocateur standard est la première métaclasse.

La version ObjVlisp des classes abstraites

Nous donnons en trois étapes successives la solution ObjVlisp au problème des classes abstraites. Pour ce faire, nous conservons la syntaxe Smalltalk et anticipons l'extension ClassTalk qui sera présentée ultérieurement.

- Création d'une nouvelle métaclasse `AbstractClass` modélisant l'ensemble des classes abstraites. Cette classe est à la fois instance ET sousclasse de la métaclasse primitive `Class`. `AbstractClass` redéfinit simplement les méthodes d'allocations `new` et `new:` pour signaler une erreur :

```
Class newName: #AbstractClass
   superclass: Class
   instanceVariableNames: ''
   category: 'Metaclass-Library'
AbstractClass methodsFor: '(forbidden) allocation'
new
   self error: 'no instance, I am an abstract class'
new: size
   self error: 'no instance, I am an abstract class'
```

- Création de la classe abstraite `Complex`, comme l'une des instances de `AbstractClass` :

```
AbstractClass newName: #Complex
   superclass: Object
   instanceVariableNames: ''
   category: 'Numeric-Complex'
```

- Création des classes `Cartesian` et `Polar`, instances de `Class` et sousclasses de `Complex` :

```
Class newName: #Cartesian
   superclass: Complex
   instanceVariableNames: 'x y'
   category: 'Numeric-Complex'
```

```
Class newName: #Polar
   superclass: Complex
   instanceVariableNames: 'rho theta'
   category: 'Numeric-Complex'
```

3.2 Uniformité du protocole de création

En ObjVlisp (comme en CLOS) le processus de création des objets est défini comme la composition de l'allocation et de l'initialisation : `create` = `initialize` o `allocate`

`Class` détient la méthode d'allocation standard `allocateInstance` ainsi que la methode de création standard, appelée `makeInstance:`. Il existe deux méthodes d'initialisation, toutes deux nommées `initializeInstance:`. La première est détenue par la classe `Object` et définit l'initialisation standard des objets terminaux. La seconde, détenue par `Class`, définit l'initialisation des classes. L'initialisation des classes utilise celle des objets terminaux mais doit également prendre en compte la compilation statique des variables d'instances héritées (le calcul de la

liste de précédence en CLOS) ainsi que la compilation des méthodes d'instances. Cette seconde méthode d'initialisation spécialise (et appelle par la construction "super") la méthode `initializeInstance:` générale détenue par `Object`. L'architecture du noyau ObjVlisp est donc la suivante :

```
Object <initializeInstance:>
  Class <allocateInstance initializeInstance: initializeClass: makeInstance:>

makeInstance: = initializeInstance: o allocateInstance
Class.initializeInstance: = initializeClass: o Object.initializeInstance:
```

La méthode `makeInstance:` reçoit en argument les variables d'instances de l'objet receveur suffixées (par le caractère ":") et leurs valeurs d'initialisation [5].

4 ClassTalk: ObjVlisp en Smalltalk-80

Deux problèmes sont posés par l'implémentation du modèle ObjVlisp en Smalltalk. Celui de l'introduction d'une architecture de classes explicites non réduite au couplage automatique classe-métaclasse et celui d'une méthode de création unifiée. Ce papier traite principalement du premier problème, et aborde seulement le deuxième.

4.1 Création explicite des classes

Pour uniformiser le processus de création des classes, celles-ci seront créées comme les autres objets par envoi d'un message à leur (méta)classe et non plus à la classe dont elles héritent. Dans la tradition ObjVlisp, les arguments de ce message sont conformes à la structure générale du receveur et servent à donner le nom de la classe, celui de sa super-classe, l'ensemble de ses variables d'instances et sa catégorie dans le flâneur. A la manière Smalltalk, les différentes méthodes seront définies ultérieurement par l'intermédiaire de la sous-vue "texte" du flâneur ClassTalk. A la manière ObjVlisp les variables de `classes` et de `pools` sont supprimées. Le sélecteur associé au message de création est donc le suivant: `newName:superclass:instanceVariableNames:category:`.

4.1.1 Les deux alternatives: fusion ou greffon

La question qui reste posée est celle du choix de la métaclasse détentrice de cette nouvelle méthode de création de classes. Sachant qu'en ObjVlisp, la classe propriétaire de cette méthode est la métaclasse `Class`, cette question se généralise ainsi: quels sont les équivalents Smalltalk des classes `Class` et `Object` d'ObjVlisp. Deux réponses qui correspondent à deux stratégies d'implémentation peuvent être données :

a) fusionner (identifier) purement et simplement le noyau des deux classes primitives ObjVlisp (`Class` et `Object`) avec les classes Smalltalk-80 correspondantes (qui se trouvent d'ailleurs avoir le même nom). Cette solution réutilise au mieux le noyau Smalltalk-80, mais ne permet pas de supprimer les variables de `classes` et de `pools` définies dans la classe Smalltalk `Class`. Par ailleurs il sera plus difficile de distinguer les classes Smalltalk des classes ObjVlisp, toutes deux partageant la même structure.

b) greffer ObjVlisp dans Smalltalk-80 par définition d'une nouvelle métaclasse (`ClassTalk`) sous-classe directe de `ClassDescription`. Cette fois les variables de `classes` et de `pools` disparaissent réellement. L'adjonction de la variable `category` permet de lier directement la classe à sa catégorie dans le flâneur :


```
ClassDescription (instanceVariables organization)
  Metaclass (thisClass)
  Class (name classPool sharedPools)
  ClassTalk (name category)
```

Cette solution introduit néanmoins un inconvénient mineur: pour maintenir la consistence avec Smalltalk-80, certaines méthodes de `Class` (`name`, `format:`, `classPool` ...) devront être dupliquées ou redéfinies au niveau `ClassTalk`.

4.1.2 Expression de la circularité ObjVlisp

En ObjVlisp la métaclasse `Class` est instance d'elle même. Cette circularité permet, entre autre, d'utiliser une méthode de création unique pour les classes et les métaclasses.

L'observation de l'architecture Smalltalk ne fait apparaitre aucune circularité explicite. Pourtant, elle montre que la classe `Class` est à la fois l'instance et la super-classe indirecte de sa métaclasse `Class class`. Par conséquence, toute méthode définie dans `Class` sera visible de sa métaclasse `Class class`. Cette relation d'héritage entre `Class class` et `Class` établit une auto-description partielle de `Class` même si, contrairement à ObjVlisp, `Class class` et `Class` ne sont plus confondues.

Dans l'hypothèse de la fusion, pour assurer l'unicité de la méthode de création des (méta)classes, il nous suffit de placer celle-ci au niveau de la classe `Class`.

Dans l'hypothèse du greffon nous pouvons soit la dupliquer dans `ClassTalk` et sa métaclasse Smalltalk `ClassTalk class` soit modifier "à la main" (par "`ClassTalk class superclass: ClassTalk`") l'arbre d'héritage. Si `ClassTalk class` deviens une sousclasse directe de `ClassTalk`, nous réintroduisons comme pour `Class`, une auto-description partielle de `ClassTalk`.

Dans la suite du texte nous ne ferons pas d'hypothèse sur l'implémentation choisie. Cependant, par respect des traditions ObjVlisp et Smalltalk, les différents exemples présentés utilise `Class` comme première métaclasse.

4.1.3 Création unifiée des classes: "newName:...:category:"

L'implementation de la méthode `newName:...:category:` est dérivée de la méthode de création des classes Smalltalk (`subclass:...:category:`) dont elle constitue une version simplifiée. Contrairement à celle-ci, la création d'une classe n'est plus précédée de la création de sa métaclasse privée [7].

Comme la méthode Smalltalk standard, elle inclut un aiguillage destiné à distinguer entre classes à structure variable (variables d'instances indéxées) et classes à structure fixe (variables d'instances). Cet aiguillage conduit dans les deux cas à l'appel de la méthode `newName:environment:...category:` décrivant le processus de création des classes ClassTalk. Pour simplifier l'exposé nous ne présentons que la partie de cette méthode ayant trait au processus de création. Le code relatif à l'aiguillage et à l'environnement de programmation (vérification de la syntaxe, gestion des "changes"...) est donc remplacé par des commentaires :

```
Class methodsFor: 'ObjVlisp - class creation'
newName: className superclass: superClass instanceVariableNames: ivnString
                  category: categoryString
  "Here takes place a dispatch along classes with indexed variables."
  "self isVariable ..."
  ^self
      newName: className
      environment: Smalltalk
      superclass: superClass
      otherSupers: nil
      instanceVariableNames: ivnString
      variable: false
```

```
        words: true
        pointers: true
        category: categoryString

newName: className environment: env superclass: superClass
                    otherSupers: otherSuperClassArray instanceVariableNames: ivnString
                    variable: v words: w pointers: p category: categoryString
    | newClass "..." |
    "Here takes place syntax checking and redefinition management."
    "(1) Allocation of the new class."
    newClass := self new.
    "(2) Initialization of the new class - 1."
    newClass
       superclass: superClass
       methodDict: MethodDictionary new
       format: -8192
       name: className
       organization: ClassOrganizer new
       instVarNames: (Scanner new scanFieldNames: ivnString)
       classPool: nil
       sharedPools: nil.
    otherSuperClassArray isNil    "(3) Specification of remaining superclasses."
       ifFalse: [newClass otherSupers: otherSuperClassArray].
    newClass                      "(4) Initialization of the new class - 2."
       format: newClass allInstVarNames size
       variable: v
       words: w
       pointers: p.
    "Here takes place environment management."
    ObjVlispOrganization classify: newClass name under: categoryString asSymbol.
    "Here takes place hierarchy updating and change management."
    otherSuperClassArray isNil    "(5) Compilation of multiple inheritance."
       ifFalse: [newClass copyMethods].
    ^newClass
```

Quatre remarques peuvent être faites à la lecture de ce "code" :

1. cette méthode de création peut être définie indifféremment dans `Class` ou `ClassTalk` selon que l'on choisisse l'option "fusion" ou l'option "greffon",

2. comme le préconise ObjVlisp, la création d'une classe s'opère bien en deux étapes: allocation de la classe (1) puis initialisation de celle-ci (2) et (4). La classe nouvellement créée (variable temporaire `newClass`) apparait maintenant comme l'instance explicite de la métaclasse receveuse du message de création ("self new"). Comme en Smalltalk-80 standard, l'initialisation de la classe s'effectue alors en deux étapes successives, (2) et (4),

3. cette méthode prend également en compte le nouvel "organizer" `ObjVlispOrganization` destiné à la construction d'un flâneur dédié à la manipulation des classes ClassTalk,

4. cette méthode prévoit l'introduction de l'héritage multiple des (méta)classes telle qu'elle sera discuté ultérieurement. A cet effet le paramètre `otherSuperClassArray` recevra les autres super-classes directes de la nouvelle classe (cf. 7.1). Le défaut correspond au cas de l'héritage simple pour lequel la valeur de ce paramètre est `nil`. Les expressions (3) et (5) ne sont donc évaluées que dans le cas de l'héritage multiple: (3) initialise la variable d'instance otherSupers, (5) appelle le module de traitement de l'héritage multiple défini par [13].

4.2 Pour un protocole d'initialisation unifié

Pour réaliser une implémentation exacte du modèle ObjVlisp il nous faudrait définir une méthode générale de création ainsi que deux méthodes d'initialisation destinées respectivement aux objets

terminaux et aux classes. La solution développée dans [4] est fondée sur le schema suivant:

```
Object <initialize:  ...>
  Behavior <... new create:...>                create: initArray
    ...      Class(Talk) <... initialize:...>   ↑self new initialize: initArray
```

La méthode de création `create:` est placée au niveau de `Behavior` tandis que les deux méthodes d'initialisation `initialize:` sont situées respectivement dans `Object` et `Class(Talk)`. Ces méthodes recoivent en argument les valeurs nécessaires à l'initialisation de l'objet créé. Le nombre de celles-ci est bien sûr fonction de la structure de l'objet considéré. La syntaxe Smalltalk n'autorisant pas les méthodes à nombre variable d'arguments, ces valeurs sont regroupées en un argument unique (`initArray`).

La méthode d'initialisation générale de `Object` reproduit le mécanisme des keywords à la Lisp. Les valeurs initiales des instances variables - données dans un ordre quelconque - sont préfixées par le nom de la variable à initialiser. La création d'un Cartesien pourra donc prendre la forme : `Cartesian create: #(y: 2 x: 1)`, la classe `Cartesien` étant elle même définie par: `Class create: #(name #Cartesien superclass Object instanceVariables 'x y' ...)`. La spécialisation de `initialize:` dans `Behavior` traite des initialisations propres aux classes.

Au niveau implémentatoire [4] on utilisera la méthode `instVarAt:put:` (de `Object`) pour affecter chaque variable d'instance à la valeur produite par un appel explicite au compilateur.

On observera cependant que le protocole de création ainsi mis en place suppose de la part de l'utilisateur une connaissance exacte de la structure des objets manipulés (en particulier celle des noms des variables d'instance). Création uniforme et réflexion vont donc à l'encontre du principe d'encapsulation.

La suite de ce papier est consacrée à l'étude de l'architecture de métaclasses explicites à la ObjVlisp en Smalltalk. Nous souhaitons établir qu'une telle architecture n'est utilisable que si le mécanisme de création est unifié et explicite.

5 Adaptation du flâneur Smalltalk-80 à ClassTalk

Le flâneur Smalltalk-80 n'est pas adapté à la manipulation de classes ClassTalk. En premier lieu, le patron ("template") de création des classes correspond à celui d'une classe Smalltalk. En deuxième lieu, la disparition du couplage classe-métaclasse rend obsolète la vue d'échange *instance/class*, unique moyen d'accéder aux métaclasses implicites. Les métaclasses ClassTalk sont nommées et se repartiront donc normalement dans les différentes catégories du flâneur.

Nous avons donc choisi de proposer un nouveau flâneur spécialisé dans la gestion et la manipulation des classes ClassTalk. Ce flâneur modélisé par la classe `ObjVlispBrowser` est une spécialisation du flâneur standard tel qu'il est défini par la classe `Browser`. Bien évidemment `ObjVlispBrowser` est défini comme une sousclasse directe de `Browser`, contrairement à ce dernier il est couplé à l'organisateur des classes ClassTalk.

Comme le montre la figure 1, ce nouveau flâneur ne propose plus la sous-vue d'échange *instance/class*, et le patron affiché dans la zone `text` correspond maintenant au schéma de création des classes ClassTalk: (Class newName: ...) On observera également que ce flâneur est interfacé avec un éditeur d'arbres qui permet d'afficher et de parcourir les deux arbres d'héritage et d'instanciation [14].

6 Une bibliothèque de métaclasses

La première étape d'expérimentation et de validation du système ClassTalk a été la construction d'une bibliothèque de métaclasses. Nous avons choisi des métaclasses qui modélisaient des

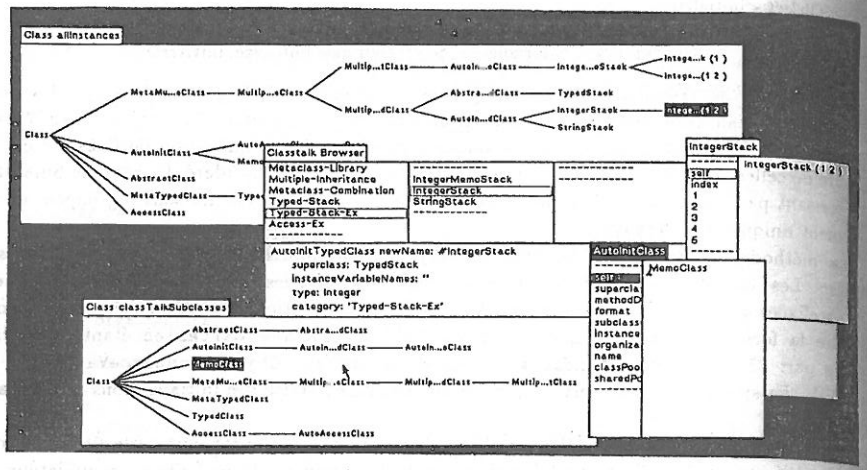


Figure 1: Le flâneur ClassTalk

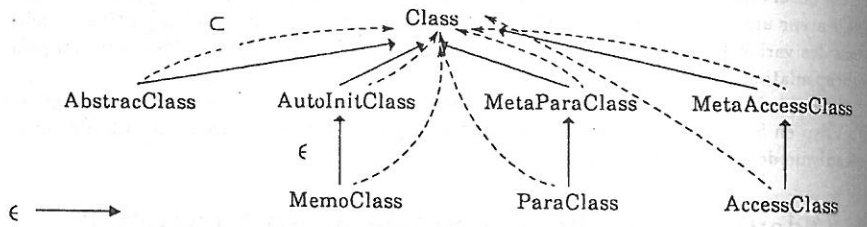


Figure 2: Une première bibliothèque de métaclasses.

"schémas de programme" classiques en Smalltalk. La figure 2 introduit ces différentes classes et détaille les relations d'instanciation et d'héritage qui les unissent. On conviendra que les noms des métaclasses ClassTalk se termineront toujours par le suffixe `Class`. On vérifiera finalement que chaque métaclasse hérite son comportement "méta" soit de la métaclasse originelle `Class` soit d'une autre métaclasse.

6.1 AutoInitClass

Tout programmeur Smalltalk a redéfini une fois au moins la méthode de classe `new` pour initialisation "à son défaut" les instances d'une classe donnée. Pour modéliser ce comportement nous proposons la métaclasse `AutoInitClass`. Elle definit les classes dont les instances recevront automatiquement un message d'initialisation après leurs allocations :

```
Class newName: #AutoInitClass
   superclass: Class
   instanceVariableNames: ''
   category: 'Metaclass-Library'
AutoInitClass methodsFor: 'allocation'
new
   ^super new init
```


`AutoInitClass` évite bien sûr de répéter les méthodes de classe `new` dont le corps se réduit à la composition de l'allocateur standard ("super new") avec une méthode d'initialisation spécifique (`init`). Cette construction correspond exactement au `isnew` de Smalltalk-72.

6.2 MemoClass

Un autre comportement dont la modélisation est souvent donnée en exercice aux débutants Smalltalk est celui des classes capables de mémoriser leurs instances.

En ClassTalk, nous introduisons la métaclasse `MemoClass` et nous lui associons une nouvelle variable d'instance `instances` chargée de la mémorisation des différentes instances créées. Nous définissons ensuite une méthode d'accès en lecture à cette variable. Finalement nous redéfinissons la méthode `new` pour qu'elle alloue ET mémorise tout nouvel objet créé.

La (ré)utilisation de `AutoInitClass` comme classe de `MemoClass` assure l'initialisation automatiquement de la variable d'instance `instances` à une nouvelle `OrderedCollection` par simple définition d'un `init` ad-hoc :

```
AutoInitClass newName: #MemoClass
   superclass: Class
   instanceVariableNames: 'instances '
   category: 'Metaclass-Library'
MemoClass methodsFor: 'init'
init
   instances := OrderedCollection new
MemoClass methodsFor: 'allocation'
new
   "Method add: returns the object added."
   ^instances add: super new
MemoClass methodsFor: 'accessing'
instances
   ^instances
```

6.3 Para(meterized)Class

La métaclasse `ParaClass` modèlise des classes paramétrables par une autre classe. Par exemple, la classe des "piles paramétrées" définie, on en dérivera la classe piles d'entiers, celle des piles de réels, celle des piles de piles ....

Par rapport aux classes standards, ce paramétrage nécessite l'ajout d'une nouvelle variable d'instance que nous nommerons `type`. Deux méthodes d'instances donnent accès (en lecture/écriture) à celle-ci. La création et plus précisemment l'initialisation des classes paramétrées nécessite l'expression de leur paramètre. A cet effet, nous spécialisons la méthode de création standard `newNamed:...:category:` pour qu'elle permette également l'expression du "`type`". Le problème est alors de "placer" cette nouvelle méthode. Celle-ci nécessite en fait la construction d'une nouvelle métaclasse `MetaParaClass` selon l'architecture suivante:

```
Class (name classPool sharedPools) <... newName:...:category: ...>
   MetaParaClass () <newName:...:type:category:>
   ParaClass (type) <type type:>
```

Bien évidemment `newNamed:...:type:category:` compose `newNamed:...:category:` de `Class` avec `type:` de `ParaClass`.

```
Class newName: #MetaParaClass
   superclass: Class
   instanceVariableNames: ''
   category: 'Metaclass-Library'
```

```
MetaParaClass methodsFor: 'creation'
newName: className superclass: superClass instanceVariableNames: ivnString
                  type: aClass category: categoryString
   ^(self newName: className superclass: superClass
         instanceVariableNames: ivnString category: categoryString)
      type: aClass

MetaParaClass newName: #ParaClass
   superclass: Class
   instanceVariableNames: 'type '
   category: 'Metaclass-Library'
ParaClass methodsFor: 'accessing'
type
   ^type
type: aClass
   type := aClass
```

Cette exemple laisse déjà entrevoir les limites du système de metaclasses étudié. En l'absence d'un protocole d'initialisation standard nous devrons redéfinir une méthode de création à chaque extension de la structure des classes standards (par ajout de variables d'instance au niveau méta). Cette redéfinition a pour effet pervers de réintroduire le couplage classe-métaclassse que nous souhaitions justement supprimer. Ainsi la définition de `ParaClass` induit celle de `MetaParaClass`.

6.4 AccessClass

Un autre aspect "repétitif" de la programmation Smalltalk-80 est la définition de méthodes d'accès aux variables d'instances. Si l'on accepte que ces méthodes prennent le nom des variables dont elles contrôlent l'accès (et sans rentrer dans la polémique du génie logiciel!) il est facile de les générer automatiquement.

Les pages 289–290 de [10] donnent une solution à cette génération automatique qui procède par ajout d'une méthode de création spécifique dans `Class`. Cette solution n'est donc malheureusement pas intrinsèque à un ensemble de classes données.

Nous proposons ici la métaclasse `AccessClass` modélisant les classes qui seront automatiquement créées avec des méthodes d'accès aux variables d'instances déclarées publiques (les variables déclarées par le keyword `instanceVariableNames:` seront considérées comme privées). L'exemple ci-dessous définit la classe `Record` [10] comme instance de `AccessClass`. Cette classe est dotée à sa création des deux méthodes d'accès à la variable "public" `name` :

```
AccessClass newName: #Record
   superclass: Object
   instanceVariableNames: 'name address'
   public: 'name '
   category: 'Access-Example '
```

Comme pour l'exemple des classes paramétrées nous devons introduire la nouvelle métaclasse `MetaAccessClass`. Cette fois la méthode de création standard est composée avec la méthode `makeIvAccessOn:` chargée de la génération des accesseurs :

```
Class newName: #MetaAccessClass
   superclass: Class
   instanceVariableNames: ''
   category: 'Metaclass-Library'
MetaAccessClass methodsFor: 'creation'
newName: className superclass: superClass instanceVariableNames: ivnString
                  public: publicIvnString category: categoryString
```



```
   ^(self newName: className superclass: superClass
         instanceVariableNames: ivnString category: categoryString)
      makeIvAccessOn: (Scanner new scanFieldNames: publicIvnString)

MetaAccessClass newName: #AccessClass
   superclass: Class
   instanceVariableNames: ''
   category: 'Metaclass-Library'
AccessClass methodsFor: 'access generation'
makeIvAccessOn: ivNameArray
   ivNameArray isNil ifFalse:
      [ivNameArray do: [:ivString |
         self compile: ivString , '\ ^' withCRs , ivString
               classified: #accessing;
               compile: ivString , ': aValue\
' withCRs , ivString , ' := aValu\accent19e
               classified: #accessing]]
```

Une instance de la classe `Scanner` analyse la déclaration des variables publiques donnée sous la forme d'une chaine de caractères et retourne un tableau à partir duquel sont calculés les sélecteurs et les corps des méthodes compilées automatiquement.

7 Introduction de l'héritage multiple

Les exemples précédents ont montré comment composer des (méta)classes par héritage simple. Pourtant l'héritage simple a ses limites et les systèmes à classes modernes (C++, CLOS) proposent aujourd'hui l'héritage multiple. Dans le cadre de notre étude méthodologique, il nous a semblé important d'investiguer également la composition des (méta)classes par héritage multiple.

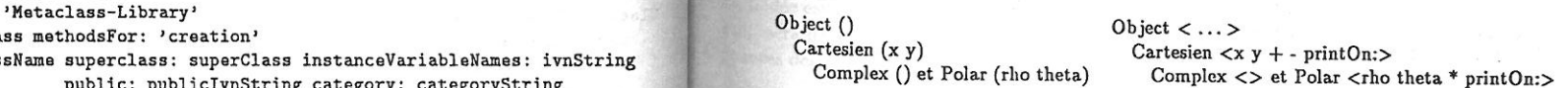
7.1 Héritage multiple en Smalltalk-80

L'héritage multiple n'existe pas en "primitif" en Smalltalk-80 ce qui signifie qu'il n'est pas pris en compte par les primitives de la machine virtuelle et que les différentes bibliothéques de l'environnement Smalltalk sont écrites en héritage simple. Néanmoins une extension traite de l'héritage multiple. Cette extension est compatible avec le système standard et elle est fournie avec lui.

Le principe est de préserver l'architecture des classes standards et l'héritage simple (en particulier la recherche des méthodes) pour greffer l'héritage multiple par introduction d'une nouvelle métaclasse gérant les "autres" superclasses directes.

La première superclasse joue le rôle de la superclasse unique en héritage simple. Les autres superclasses sont mémorisées dans la métaclasse privée de la classe. A la création d'une classe à superclasses multiples, le système détecte les méthodes des superclasses qui ne sont pas accessibles par héritage simple et procède à leur recompilation dans le dictionaire de la classe (même si par la suite le flâneur les occulte volontairement). Il se charge également de signaler les conflits (méthodes de même nom) en générant automatiquement des "méthodes de conflits" que l'utilisateur devra résoudre [13].

Pour illustrer ce principe, nous reprenons l'exemple des complexes que nous traitons cette fois en héritage multiple. La classe `Complex` est définie comme sousclasse directe de `ComplexCartesien` et de `PolarCartesien` :



Pour `Complex`, `Cartesien` joue le rôle de la première superclasse et ses méthodes sont directement accessibles par le "look-up" standard. Par contre, les méthodes `rho`, `theta` et `*` de `Polar` ne sont pas accessibles et doivent être recompilées dans `Complex`. La méthode `prinOn:` étant définie dans les deux superclasses, une méthode de conflit est générée dans `Complex`.

L'héritage multiple est réalisée par la métaclasse `MetaclassWithMultipleInheritance`. Cette métaclasse hérite de `Metaclass` et introduit une nouvelle variable d'instance `otherSupers` chargée de mémoriser les "autres" superclasses (la première est toujours gérée par la variable d'instance `superClass`). L'idée est donc qu'une classe à héritage multiple ne détient pas directement ses superclasses directes qui sont placées dans sa métaclasse privée. Une nouvelle méthode de création pour les classes à héritage multiple (`named:superClasses:...:category`) est introduite dans `Class class` ainsi que différentes méthodes auxilliaires (dont `subclass:otherSupers:...:category` dans `Class`) :

```
ClassDescription (instanceVariables organization) <...>
  Metaclass (thisClass) <...>
    MetaclassWithMultipleInheritance (otherSupers) <...>
  Class (name classPool sharedPools) <subclass:otherSupers:...:category:>
    Object class () <...>
    ...
            Class class () <... named:superclasses: ...:category:>
```

Il est intéressant d'observer que comme ClassTalk, cette extension préconise d'expliciter la création d'une classe par envoi d'un message à sa métaclasse (et non pas à sa première superclasse comme dans le Smalltalk standard). Nous définirions ainsi la classe `Complex` en héritage multiple :

```
Class named: #Complex
   superclasses: 'Cartesien Polar'
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'MultipleInheritanceExample '
```

7.2 Héritage multiple en ClassTalk

Par souci de compatibilité avec Smalltalk, l'héritage multiple ClassTalk reprend le principe de l'extension d'Ingalls & Borning mais l'adapte à sa vision explicite des métaclasses. La classe `MultipleInheritanceClass` est introduite pour modéliser les classes ClassTalk à héritage multiple. A la différence de Smalltalk, celles-ci auront connaissance de leurs superclasses directes. `MultipleInheritanceClass` hérite donc de `Class(Talk)` et non pas de `Metaclass`.

L'enrichissement de la structure des classes à héritage multiple est réalisé par la nouvelle variable d'instance `otherSupers`. L'initialisation de celle-ci conduit à introduire la métaclasse `MetaMultipleInheritanceClass` qui détiendra la méthode de création des classes à superclasses multiples. Comme nous souhaitons que `MultipleInheritanceClass` accède également à cette méthode nous la définissons à la fois comme instance et sousclasse de `MetaMultipleInheritanceClass`:

```
Class(Talk) (...) <...>
  MetaMultipleInheritanceClass () <newName:superclasses:...:category:>
    MultipleInheritanceClass (otherSupers) <... otherSupers: superclasses>
```

Cette architecture se traduit ainsi en ClassTalk par :

```
Class newName: #MetaMultipleInheritanceClass
   superclass: Class
   instanceVariableNames: ''
   category: 'Multiple-Inheritance '
```

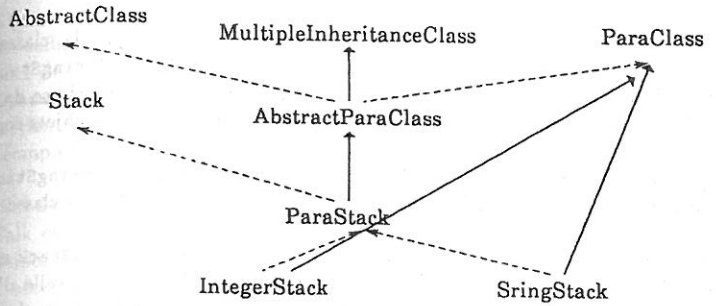



Figure 3: Architecture de Piles Paramétrées.

```
MetaMultipleInheritanceClass methodsFor: 'creation'
newName: className superclasses: superclassNames instanceVariableNames: ivnString
                 category: categoryString
   | superclasses firstSuperclass |
   "Find the superclasses corresponding to the superclassNames"
   superclasses := Class getSuperclasses: superclassNames.
   firstSuperclass := superclasses first.
   "Here takes place a dispatch along classes with indexed variables."
   ^self
      newName: className
      environment: Smalltalk
      superclass: firstSuperclass                                    "(1)"
      otherSupers: (superclasses copyFrom: 2 to: superclasses size) "(2)"
      instanceVariableNames: ivnString
      variable: false
      words: true
      pointers: true
      category: categoryString

MetaMultipleInheritanceClass newName: #MultipleInheritanceClass
   superclass: MetaMultipleInheritanceClass
   instanceVariableNames: 'otherSupers '
   category: 'Multiple-Inheritance '
```

La méthode `newNamed:superclasses:...:category:` est dédiée à la création des classes à héritage multiple. L'implémentation (et la syntaxe) de cette méthode est dérivée de celle de la méthode utilisée pour la création des classes à héritage simple. Comme dans l'implémentation d'Ingalls & Borning la première superclasse joue un rôle privilégié et elle est traitée à la façon de l'héritage simple (1). Les autres superclasses sont extraites et associées à l'élément de keyword `otherSupers` qui se charge de les affecter à la variable d'instance de même nom (2). Le traitement spécifique à l'héritage multiple (gestion des conflits, recompilation des méthodes) est défini dans la méthode `newNamed:superclass:...:category:` dont le code complet est donné en 4.

8 La méthodologie ClassTalk

Nous avons choisi de développer un exemple mettant en jeu suffisamment de niveaux de métaclasses pour être révélateur de la méthodologie ClassTalk. L'architecture de cet exemple, intitulé "Les piles paramétrées", est détaillée par la figure 3.

L'idée est de modéliser des piles qui ne pourront contenir que les objets d'une classe donnée. On supposera qu'il existe déjà une classe `Stack` (Smalltalk-80 ou ClassTalk) modélisant la structure de données des Piles. Ce pourra être par exemple une sousclasse de `Array` ou de `OrderedCollection`.

Le comportement commun à l'ensemble des piles est modélisé par la classe abstraite (ParaStack) dont hériteront toutes les piles paramétrées (IntegerStack, StringStack etc ...). ParaStack est donc définie comme une instance de AbstractClass et sous classe de Stack. La méthode push: est redéfinie pour vérifier la classe de son argument. Seuls les objets conformes au "type" de la classe seront empilés.

Chaque classe définissant un type de pile paramétrée (IntegerStack, StringStack etc ...) est décrite par ParaClass, sa variable d'instance type servant à spécifier la classe des objets empilables.

Interviens alors un problème de consistence entre la classe abstraite ParaStack et ses sous-classes concrètes. En effet la structure de ParaStack doit être logiquement celle d'une classe paramétrée. Idéalement TypesStack devrait être abstraite et paramétrée. Nous aboutissons là à une incohérence: un objet ne pouvant appartenir à plus d'une classe. Pourtant, l'héritage multiple permet de combiner le modèle des classes abstraites et celui des classes paramétrées. Du fait de la non symétrie de cet héritage on pourra privilégier la composante paramétrage (ParaAbstractClass) ou la composante abstraite (AbstractParaClass). Dans ce dernier cas de figure on obtiendra:

```
MultipleInheritanceClass newName: #AbstractParaClass
   superclasses: 'AbstractClass ParaClass '
   instanceVariableNames: ''
   category: 'Metaclass-Combination'
AbstractParaClass methodsFor: 'conflicting inherited methods'
new
   ''^super new''
   ^self AbstractClass.new

AbstractParaClass newName: #ParaStack
   superclass: Stack
   instanceVariableNames: ''
   category: 'Stack-Collection'
ParaStack methodsFor: 'operations'
push: x
   (x isMemberOf: self class type)
      ifTrue:  [super push: x]
      ifFalse: [self error: 'wrong type ']

ParaClass newName: #IntegerStack
   superclass: ParaStack
   instanceVariableNames: ''
   type: Integer
   category: 'Stack-Collection'

ParaClass newName: #StringStack
   superclass: ParaStack
   instanceVariableNames: ''
   type: String
   category: 'Stack-Collection'
```

Les méthodes de création (new et new:) provoquent un conflit d'héritage, celles-ci étant définies dans AbtractClass et dans une super-classe de ParaStack (Behavior). Ce modèle privilégiant l'abstraction par rapport à la paramétrisation, nous résolvons donc ce conflit par redirection sur la méthode new (new:) de AbstractClass. Nous explicitons cette redirection en utilisant soit la notation pointée AbstractClass.new proposée par [13] soit la construction super qui fonctionne ici AbstractClass étant définie comme la première superclasse directe.

9 Premier bilan de l'expérience ClassTalk

Nous avons présenté comme motivation principale de ce papier les limitations du modèle de métaclasses Smalltalk.

Notre première expérience ClassTalk a montré qu'une architecture de métaclasses explicites n'était pas suffisante et devait s'accompagner nécessairement d'un protocole de création unifié. En l'absence de ce dernier, le couplage classe-métaclasse préconisé par Smalltalk réapparait (cf. *AccessClass, *ParaClass et *MultipleInheritanceClass) et viens alourdir considérablement la construction des méta-niveaux. La programmation par métaclasses explicite requiert donc que ces deux composantes soient réunies simultanément [4].

Du point de vue pédagogique, la présence dans un même environnement des flâneurs Smalltalk et ClassTalk est riche d'enseignement. Elle permet de comparer les possibilités offertes par les deux systèmes de métaclasses. En particulier, elle montre que la programmation par métaclasses explicites nécessitent de composer en permanence les mécanismes d'héritage et d'instanciation. Il est donc indispensable de doter l'environnement ClassTalk d'éditeurs d'arbres permettant de visualiser et d'explorer simultanément les hiérarchies d'héritage et d'instanciation.

La coexistence dans un même système de classes Smalltalk et ClassTalk pose des problèmes de compatibilité [12] lorsqu'une classe d'un système hérite d'une classe de l'autre système. Quand une classe ClassTalk hérite d'une classe Smalltalk, la règle d'héritage parallèle ne s'applique plus, et des méthodes de classes deviennent inaccessibles. Dans le cas des piles paramétrées, un tel problème interviendra si nous définissions la classe Stack comme une sousclasse de la classe Smalltalk OrderedCollection. En effet, OrderedCollection détient la méthode setIndices chargée d'initialiser ses deux variables d'instances. Il se trouve que la métaclasse OrderedCollection class prend implicitement soin de rédéfinir la méthode d'allocation new: pour qu'elle appelle bien la méthode setIndices :

```
OrderedCollection class methodsFor: 'instance creation'
new: anInteger
   ^(super new:anInteger) setIndices
```

Pour éviter une telle erreur d'initialisation, l'utilisateur ClassTalk doit s'assurer que la métaclasse ParaStack commune à l'ensemble des piles paramétrées redéfinit correctement la méthode d'allocation new:.

Un travail futur serait de rajouter à ClassTalk un module chargé de signaler et de résoudre ces problèmes de compatibilité.

Finalement l'utilisation de l'héritage multiple nécessite l'introduction d'un mécanisme de combinaison des méthodes à la CLOS. Ainsi lors de la définition d'une sous-classe de AutoInitClass et MemoClass nous ne devrions pas résoudre un conflit mais exprimer une combinaison des new.

Nous remercions Francis Wolinski pour nous avoir permis d'interfacer son éditeur d'arbres avec le flâneur ClassTalk.

Références

[1] D.G. Bobrow and G. Kiczales, The Common Lisp Object System Metaobject Kernel - A Status Report, ACM Conference on Lisp and Functional Programming (*LFP'88*), pages 309-315, July 1988.

[2] A. Borning and T. O'Shea, Deltatalk: An Empirically and Æsthetically Motivated Simplification of the Smalltalk-80 Language, *ECOOP'87*, *LNCS*, No 276, pages 1-10, Springer-Verlag, June 1987.

[3] J.-P. Briot and P. Cointe, A Uniform Model for Object-Oriented Languages Using The Class Abstraction, *IJCAI'87*, Vol. 1, pages 40-43, August 1987.

[4] J.-P. Briot and P. Cointe, Programming with Explicit Metaclasses in Smalltalk-80, *OOPSLA'89*, New Orleans, October 1989.

[5] P. Cointe, Metaclasses are First Class: the ObjVlisp Model, *OOPSLA'87*, pages 156-167, Orlando, USA, October 87.

[6] P. Cointe and N. Graube, Programming with Metaclasses in CLOS, *First CLOS Users and Implementors Workshop*, Xerox Parc, Palo Alto CA, USA, October 1988.

[7] P. Cointe, A Tutorial Introduction to Metaclass Architectures as Provided by Class Oriented Languages, International Conference on Fifth Generation Computer Systems (*FGCS'88*), Vol. 2, pages 592-608, November-December 1988.

[8] P. Deutsch The Past, Present and Future of Smalltalk *invited paper ECOOP'89*, pages 73-87, edited by Steve Cook, Cambridge University Press.

[9] L.G. DeMichiel and R.P. Gabriel, The Common Lisp Object System: An Overview, *ECOOP'87*, *LNCS*, No 276, pages 151-170, Springer-Verlag, June 1987.

[10] A. Goldberg and D. Robson, Smalltalk-80: the Language and its Implementation, *Series in Computer Science*, Addison Wesley, 1983.

[11] N. Graube, Reflexive Architectures: From ObjVlisp to CLOS, *ECOOP'88*, *LNCS*, No 322, pages 110-127, Springer-Verlag, August 1988.

[12] N. Graube, Metaclass Compatibility, *OOPSLA'89*, New Orleans, October 1989.

[13] D.H.H. Ingalls and A.H. Borning, Multiple Inheritance in Smalltalk-80, Proceedings of the National Conference on Artificial Intelligence, pages 234-237, August 1982.

[14] F. Wokinsky The MV2C metaphor: modeling and generating some man-machine interfaces *rapport de recherche LAFORIA*, Université Pierre et Marie Curie

[15] F. Wokinski Gestion de contraintes induites dans la structuration des objets en sous-objets *même conférence.*

Actalk: une Plateforme de Modélisation de Langages d'Acteurs en Smalltalk-80

Actalk: a Platform to Model Actor Languages in Smalltalk-80

Jean-Pierre BRIOT

Equipe Mixte LITP - Rank Xerox France,
Université Pierre et Marie Curie,
4 place Jussieu, 75005 Paris, France
briot@litp.ibp.fr

RESUME - Cet article décrit une plateforme de modélisation, classification et expérimentation de langages d'acteurs, premiers pas vers les systèmes multi-agents. Ce système, nommé *Actalk*, qui signifie *act*eurs en Small*talk*-80[1] étend les objets passifs et séquentiels activés par envois de messages synchrones vers des acteurs actifs et parallèles communiquant par messages asynchrones. Le noyau d'Actalk est aisément étendu pour simuler les principaux modèles de calcul et langages d'acteurs, notamment le modèle de calcul de Agha, et le langage Abcl/1 de Yonezawa, ainsi implicitement classifiés par héritage. L'environnement standard de Smalltalk-80 est spécialisé pour visualiser et contrôler dynamiquement l'exécution des acteurs.

Mots clés : objet, Smalltalk-80, parallélisme, acteur, agent, plateforme, spécification, expérimentation, modularité, combinaison, Act*, Abcl/1, classification, environnement.

ABSTRACT - This paper describes a platform to model, classify and experiment with *actor* languages, foundations for multi-agents systems. This system, named *Actalk*, which stands for *act*ors in Small*talk*-80, extends passive and serial objects activated through synchronous message passing towards active and concurrent actors communicating through asynchronous messages. The kernel of Actalk is easily extended to simulate the main actor computation models and languages, Agha's actor computation model, and Yonezawa's Abcl/1 language, thus implicitly classified with inheritance. Smalltalk-80 standard programming environment is specialized in order to dynamically visualize and control execution of actors.

Keywords : object, Smalltalk-80, parallelism, actor, agent, platform, specification, experiment, modularity, combination, Act*, Abcl/1, classification, environment.

1 Introduction

Cet article décrit la modélisation de langages d'*acteurs*. Les langages d'acteurs sont basés sur la notion d'objets actifs et autonomes qui interagissent selon des protocoles de communication fondés sur l'envoi de message. Le concept d'*acteur* est sans doute la fondation la plus appropriée pour décrire et implémenter des systèmes multi-agents [Ferber 87]. Par opposition aux

[1] Smalltalk-80 est une marque déposée de ParcPlace Systems.

Cette recherche a bénéficié du soutien du GRECO Programmation.