Université Montpellier-II
UFR des Sciences - Département Informatique
Master Informatique - UE FMIN 107 -

Ingénierie Logicielle - Concepts et Outils de la modélisation et du développement de logiciel par et pour la réutilisation.

Partie No 3 : **Vue générale de Smalltalk : Typage Dynamique, Réflexivité, Métaprogrammation, IDM**

Notes de cours - 2007-2012
Christophe Dony

1 Généralités

1.1 Liens historiques et pratiques

Créé au Xerox Parc, définitions : 1972, 1976, 1980

- Point d'entrée général : http://www.smalltalk.org/main/, contient entre autres la listes de toutes les versions gratuites, libres et commerciales du langage.
- http://www.smalltalk.org/versions/ANSIStandardSmalltalk.html : la norme "ANSI Smalltalk" normalise le coeur du langage.
- http://wiki.squeak.org/squeak/64 : "Smalltalk-80 : The Language and its Implementation" (Smalltalk Blue Book), par Adele Goldbeg et David Robson est le livre historique qui définit le langage Smalltalk-80 et le coeur de son implantation. Le "purple book" (http://wiki.squeak.org/squeak/692) est le même sans le chapître 4 sur l'implantation.
- Le "Smalltalk orange book" : The Smalltalk-80 programming environment, de Adèle Goldberg et David Robson définit l'environnement de programmation historique Smalltalk. Vous y trouverez les idées matérialisées aujourd'hui dans Eclipse dont le plus beau debugger du monde à l'époque avec son "hot code replacement".
- http://www.esug.org/ : ESUG - European Smalltalk User Group.
- http://www.pharo-project.org/home. Pharo is a clean, innovative, open-source Smalltalk-inspired environment.
- http://stephane.ducasse.free.fr/FreeBooks.html : catalogue de livres sur Smalltalk dont certains en ligne.
- www.squeak.org : Squeak est un Smalltalk gratuit réalisé par une équipe dirigée par Alan Kay, l'inventeur premier du langage.
- http://www.cincomsmalltalk.com/main/ : Cincom, Smalltalk industriel (ObjectStudio - Visualworks)
- http://www.threeriversinstitute.org : site de Kent Beck, un des pionniers du "Agile development", méthodologie initialement développée autour de Smalltalk.

- http://www.chimu.com/publications/JavaSmalltalkSyntax.html : Une comparaison des syntaxes Java et Smalltalk.
- http://www.csci.csusb.edu/dick/samples/smalltalk.syntax.html : Une présentation concise dela syntaxe de Smalltalk.

1.2 Caractéristiques

- Smalltalk est un **langage à objets** basé sur un petit nombre de concepts.
- C'est un environnement de programmation interactive et multimedia
- C'est un système complet : compilateur, Metteur au point, éditeur de programmes, détection d'erreurs, aide en ligne, gestion des références croisées, ...
- C'est un environnement intégré de prototypage pour le développement rapide, agile, extrême, ...
- Langage à objets influencé par Simula (abstraction de données, surcharge des noms d'opérations, polymorphisme d'inclusion)
- influencé par Lisp
  - typage dynamique,
  - manipulation généralisée de "références",
  - gestion automatique de la mémoire, allocation dynamique et ramasse-miette,
  - style applicatif, toute instruction est une expression,
  - Langage compilé en instruction d'une machine virtuelle dédiée.

1.3 Interprétation, Compilation, machine virtuelle

Principe d'exécution : compilation des instruction en byteCodes ou instructions d'une machine virtuelle dédiée à la programmation par objets puis interprétation de ces instructions.

Machine virtuelle Smalltalk : ensemble d'instructions dédiées à la programmation par objets et interpréteur associé.

Le langage Java a repris ce schéma d'exécution. JVM (Java Virtual Machine).

Exemple de méthode et de bytecode généré

```
fact
    ^self = 0
        ifTrue: [1]
        ifFalse: [self * (self - 1) fact]
```

```
normal CompiledMethod numArgs=0 numTemps=0 frameSize=12
literals: (#fact )
1 <44> push self
2 <49> push 0
3 <A6> send =
4 <EC 07> jump true 13
6 <44> push self
7 <44> push self
8 <4A> push 1
9 <A1> send -
10 <70> send fact
11 <A8> send *
```

```
12 <66> pop
13 <60> push self; return
```

2 Syntaxe

2.0.1 constantes littérales

"Constantes" car leur valeur ne peut être modifiée et "littérales" car elles peuvent être entrées littéralement dans le texte des programmes.

- nombres : 3, 3.45, -3,
- caractères : $a, $M, $$
- chaines : 'abc', 'the smalltalk system'
- symboles : #bill, #a22
- tableaux de constantes littérales :
  #(1 2 3)
  #('vincent' 'francois' 'paul' 'autres')

2.1 Expressions. Instructions

Smalltalk peut être qualifié de langage applicatif à l'image de *Scheme* : toute instruction est une expression (ayant donc une valeur) et tout calcul peut s'exprimer comme une suite d'application de méthodes à des arguments. Voir la fonction "factorielle" présentée plus haut.

Il existe une instruction "return" utilisable dans toute méthode avec la même sémantique qu'en Java. Si elle n'est pas utilisée, la valeur implicitement rendue par toute méthode est une référence sur le receveur courant, par ailleurs accessible via l'identificateur (`self`. Le type `void` n'existe pas.

Les instructions sont séparées par un point.

```
i := 1. j := 2.
```

Principales instructions : affectation, retour de méthode, envoi de message. Les conditionnelles se réalisent avec des envois de messages.

```
i := 3 :: affectation

^33 :: équivalent du "return" java
```

2.2 Envois de message

Définitions :

- Vision : demande à un objet d'exécuter un de ses comportements implanté sous forme d'une méthode
- Pratique : appel de méthode avec sélection selon le type dynamique du premier argument qualifié de receveur et syntaxiquement distingué.

On distingue les messages

- unaires, a un paramètre : le receveur
  `1 class`
  `5 fact`
- binaires, pour les opérations arithmétiques en infixé,
  `1 + 2`
- "keywords" a $n, n >= 2$ paramètres dont le receveur.
  Leur forme syntaxique (originale) permet de représenter un envoi de message comme une "petites conversations" entre un objet et un autre.
  `1 log: 10`
  `anArray at: 2 put: 3`
  `monCalendrier enregistre: unEvenement le: unJour a: uneheure`
  Les familiers de Java traduiront en : `monCalendrier.enregistreLeA(unEvenement, unJour, uneHeure)`
- Précédence :
  unaire > binaire > keyword,
  exemple : `1  5 fact+` égale 121 et pas 720.
- Associativité A précédence égale les messages sont composés de gauche à droite.
  exemple :
  `2 + 3 * 5 = 25`
  `2 + (3 * 5) = 17`
- Cascade de messages
  `r s1; s2; s3.` est équivalent à : `r s1. r s2. r s3.`

2.3 Identificateurs

Les identicateurs sont des symboles. Le langage étant dynamiquement typé, il n'est associé aucun type statique aux identificateurs dans le texte d'un programme.

2.4 Structures de contrôle

Il n'existe aucune forme syntaxique particulière pour les structures de contrôle qui sont implantées comme des méthode standard même si leur compilation est optimisée.

Les fermetures lexicales

Un *block* est l'implantation Smalltalk de la notion de fermeture lexicale. Une fermeture lexicale est une fonction anonyme capturant son environnement lexical de définition - Toute variable libre dans la fonction est interprétée relativement à l'environnement dans lequel la fermeture a été définie. Elle est exécutable n'importe où et n'importe quand dans cet environnement.

Le propre d'une structure de contrôle est d'avoir une politique non systématique d'évaluation de ses arguments. Implanter des structures de contrôle comme des méthodes standard nécessite donc que les arguments de ces méthodes ne soient pas évalués systématiquement au moment de l'appel. La solution Smalltalk à ce problème consiste à passer des blocks, soit en postion de receveur, soit en position d'argument.

Un block s'évalue à lui-même.

```
[2 + 3]
= [2+3]
```

Un bloc est une entité de première classe qui peut être passée en argument.
Exécution d'un bloc :

```
[2 + 3] value
= 5
```

Application d'un bloc à des arguments :

```
[:i | 2 + i] value: 33
= 35
```

Les *blocks* sont utilisés pour écrire toutes sortes de fonctions d'ordre supérieur dont les structures de contrôle.

Conditionnelle

```
(number \\ 2) = 0 ifTrue: [parity := 0] ifFalse: [parity := 1]
```

Boucle for

La boucle for utilise un block à un paramètre qui tient lieu de compteur de boucle

```
1 to: 20 do: [:i | Transcript show: i printSting].
```

Itérateurs

```
untableau := #(3 5 7 9).
untableau do: [:each | Transcript show: each; cr]
```

Formes de gestion d'exceptions

```
untableau findKey: x ifAbsent: [0]
```

3 Pratique basique du langage

3.1 Une vision unifiée du développement par objets

- Toute entité est un objet.

Un objet est une entité individuelle, repérée par une adresse unique, formé de plusieurs champs, connus par leurs noms (en nombre fixé) et contenant une valeur qui peut varier au cours du temps.

- tout objet appartient à (est instance) d'une classe qui définit sa structure et ses comportements.

- un comportement d'un objet est activé par un envoi de message.

Plus généralement, **tout calcul** en Smalltalk s'effectue par l'envoi d'un message à un objet.

- Toute classe est définie comme une spécialisation d'une autre. La relation de spécialisation définit un arbre d'héritage.

3.2 Classes et Méthodes

Toute classe est créée comme sous-classe d'une autre classe.

Variables d'instance, variables de classes (statiques), variables de *pool*.

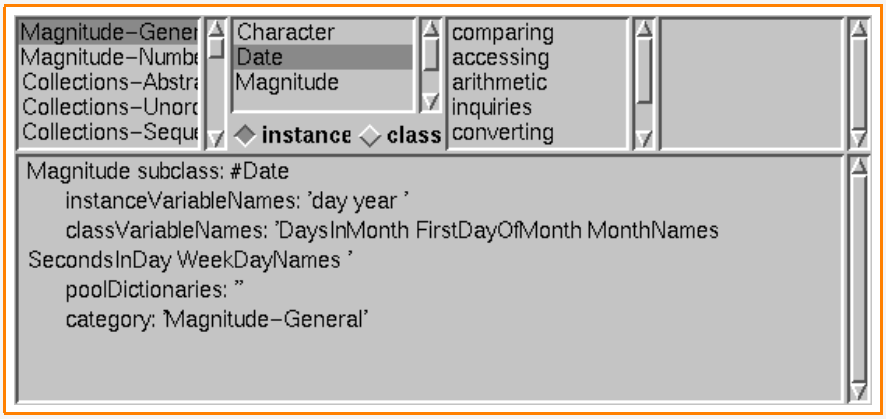

Fig. (1): Définition de la classe Date

Définition de méthodes d'instance

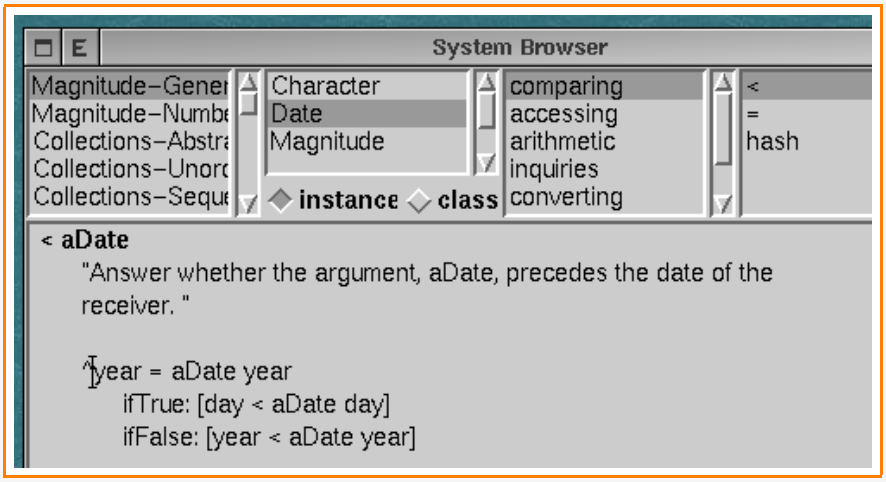

Fig. (2): Méthode d'instance < de la classe Date

Définition de méthodes de classe

Méthodes applicables aux classes et invocable par envoi de message aux classes[1]

```
Date new. Date today.
```

Les méthodes *static* de `C++` puis `Java` sont des erzats sans cohérence au niveau du langage des méthodes de classe Smalltalk.

Les méthodes de classe Smalltalk sont cohérentes dans le modèle et sont fondées sur l'existance de méta-classes.

[1] Les méthodes de classe sont en fait des méthodes normales (d'instance) définies sur les métaclasses, voir la section 4.4.

Toute classe C est instance d'une métaclasse, générée automatiquement au moment de la création de C et nommée “C class”.

Une méthode de classe est une méthode d'instance de la métaclasse.

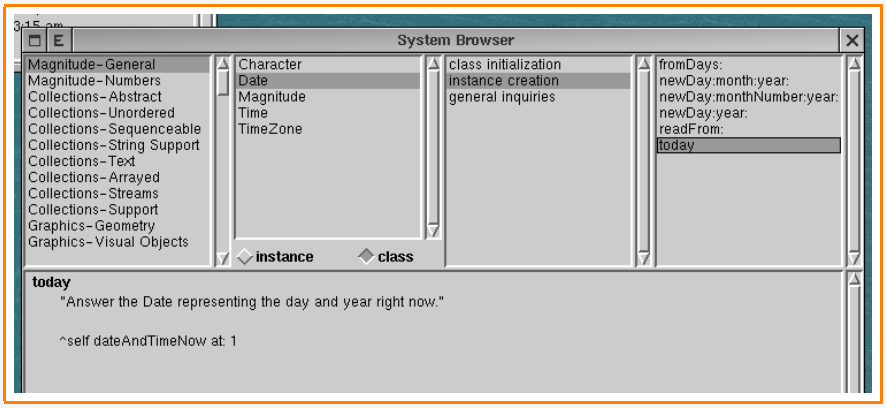

Fig. (3): Méthode de classe “today” de la classe Date (En fait une méthode d'instance de la (méta)classe “Date class”)

Différentes sortes de variables accessibles au sein des méthodes

Les variables accessibles au sein d'une méthode M :

- les paramètres de M,
- les variables temporaires de M, (exemple, réécrire théta en utilisant des variables temporaires)
- les variables d'instance de l'objet receveur O,
- les pseudo-variables self, super, true, false, nil.
- les variables de classe de la classe C de O (celle ou est définie M)
- les variables partagées entre la classe C et d'autres classes, dont les variables globales partagées entre toutes les classes,

3.3 Envoi de message

La résolution de l'envoi de message est dynamique.

Elle consiste en la recherche de la méthode dans la classe du receveur du message puis en cas d'échec dans ses superclasses.

Diverses stratégies d'optimisation de l'envoi de messages sont implantées dans les diverses machine virtuelle dont la principale est la techniques de cache.

3.4 Instantiation et initialisation des objets

- On instancie une classe en lui envoyant le message `new`.
- `new` est une méthode définie sur une superclasse commune à toutes les métaclasses (cf. métaclasses). `new` réalise l'allocation mémoire et rend un nouvel objet.
- Exemple : `Date new`

Il n'y a pas de constructeurs.

L'initialisation s'effectue via des méthodes appelées dans des méthodes de classe dédiés ou des redéfinitions de la méthode de classe `new`.

Exemples : les méthodes `new` et `x :y :` de la classe Point.

3.5 Exemple

```
Object subclass: #Compteur
    instanceVariableNames: 'valeur'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Compteur-MVC'!

!Compteur methodsFor: 'acces lecture/ecriture'!
nombreUnites
    ^valeur!

nombreUnites: n
    valeur := n.! !

!Compteur methodsFor: 'imprimer'!

printOn: aStream
    aStream nextPut: $@.
    self nombreUnites printOn: aStream! !
```

```
!Compteur methodsFor: 'operations'!

decr
        self nombreUnites: valeur - 1!

incr
        self nombreUnites: valeur + 1!

raz
    self nombreUnites: 0! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Compteur class
    instanceVariableNames: ''!

!Compteur class methodsFor: 'creation'!
new
    ^super new raz! !
```

3.6 Classe abstraite

Une classe ne peut être déclarée abstraite mais elle être rendue abstraite en redéfinissant la méthode new pour signaler une exception.

Ceci pose néanmoins de graves problèmes pour les futures sous-classes concrètes. (utilisation obligatoire de `basicNew`). Ceci est une limite de la solution Smalltalk pour les métaclasses.

3.7 Methode abstraite

Une méthode ne peut être déclarée abstraite mais elle être rendue abstraite en la définissant de la façon suivante :

```
method
    self subclassResponsibility
```

3.8 Sous-types et Héritage

Mêmes principes que dans les autres langages à objet.
Typage dynamique : pas de problème de redéfinition covariante ou contra-variante mais pas de contrôles.
Héritage simple.

3.9 Schémas de spécialisation

- Masquage : redéfinition d'une méthode sur une sous-classe.
- Masquage partiel : la pseudo-variable `super`.
- Paramétrage par spécialisation (pseudo-variable `self`) ou par composition, classiques.

3.10 Un exemple de hiérarchie : Les collections

```
ProtoObject #()
    Object #()

        Collection #()
            Bag #('contents')
                IdentityBag #()
            CharacterSet #('map')
            SequenceableCollection #()
                ArrayedCollection #()
                    Array #()
                        ActionSequence #()
                        DependentsArray #()
                        WeakActionSequence #()
                        WeakArray #()
                    Array2D #('width' 'contents')
                    B3DPrimitiveVertexArray #()
                    Bitmap #()
                Heap #('array' 'tally' 'sortBlock')
                Interval #('start' 'stop' 'step')
                    TextLineInterval #('internalSpaces' 'paddingWidth' 'lineHeight' 'baseline')
                LinkedList #('firstLink' 'lastLink')
                    Semaphore #('excessSignals')
                MappedCollection #('domain' 'map')
                OrderedCollection #('array' 'firstIndex' 'lastIndex')
                    GraphicSymbol #()
                    SortedCollection #('sortBlock')
                    UrlArgumentList #()
                SourceFileArray #()
                    StandardSourceFileArray #('files')
            Set #('tally' 'array')
                Dictionary #()
                    HtmlAttributes #()
                    IdentityDictionary #()
                        SystemDictionary #('cachedClassNames')
                            Environment #('envtName' 'outerEnvt')
                                SmalltalkEnvironment #()
                    LiteralDictionary #()
                    MethodDictionary #()
                    PluggableDictionary #('hashBlock' 'equalBlock')
                    WeakKeyDictionary #()
                        ExternalFormRegistry #('lockFlag')
                        WeakIdentityKeyDictionary #()
                    WeakValueDictionary #()
                IdentitySet #()
                PluggableSet #('hashBlock' 'equalBlock')
                WeakSet #('flag')
            SkipList #('sortBlock' 'pointers' 'numElements' 'level' 'splice')
```

```
48              IdentitySkipList #()
49          WeakRegistry #('valueDictionary' 'accessLock')
```

3.11 Héritage et description différentielle

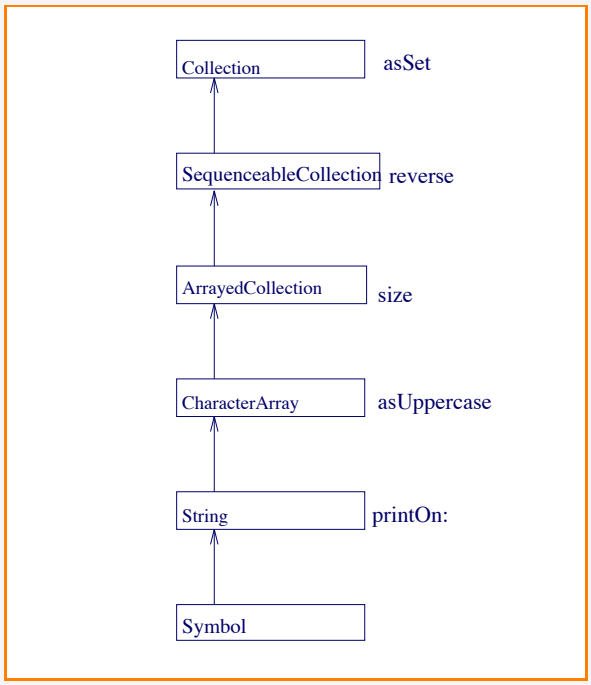

Fig. (4): Hiérarchie d'héritage avec ajoûts de fonctionnalités. Pour plus de détails sur la description différentielle, voir notes de cours "réutilisation et frameworks".

4 La méta-programmation en Smalltalk

4.1 Définitions

Réflexivité : Capacité qu'a un système à donner à ses utilisateurs une représentation de lui-même en connexion causale avec sa représentation effective en machine.

Entité de première classe : entité ayant une représentation accessible dans un programme, que l'on peut référencer, manipuler et inspecter, et éventuellement modifier.

En Smalltalk, les classes, les méthodes compilées, les méthodes, certaines structures de contrôle, éventuellement la pile d'exécution, l'environnement de programmation, le compilateur ... sont des entités de première classe.

4.2 Les objets primitifs (rock-bottom objects)

Il existe une classe décrivant chacun des types primitifs d'objets : String, Float, SmallInteger, ...

Il est possible de modifier les méthodes de ces classes, attention ce peut être fort dangereux, ou d'en créer de nouvelles (exemple : définir la méthode `factorielle` sur la classe `Integer`).

Les objets primitifs sont utilisables comme les autres objets du système (différence avec Java), ils sont représentés par une classe mais leur implantation n'est pas entièrement définie par cette classe.

La hiérarchie des classes représentant les nombres.

```
Number ()
    FixedPoint ('numerator' 'denominator' 'scale')
    Fraction ('numerator' 'denominator')
    Integer ()
        LargeInteger ()
            LargeNegativeInteger ()
            LargePositiveInteger ()
        SmallInteger ()
    LimitedPrecisionReal ()
        Double ()
        Float ()
```

4.3 Définition de nouvelles structures de contrôle

Les structures de contrôle sont définies en Smalltalk par des méthodes définies sur les classs `Boolean` (conditionelles, et, ou), `Block` ("tantque"), `Integer` (boucle "for").

Exercice : ajouter au système les méthodes `ifNotTrue :`, `ifNotFalse :`, `repeatUntil`.

4.4 Les Méta-classes en Smalltalk

4.4.1 Rappel : la solution Objvlisp

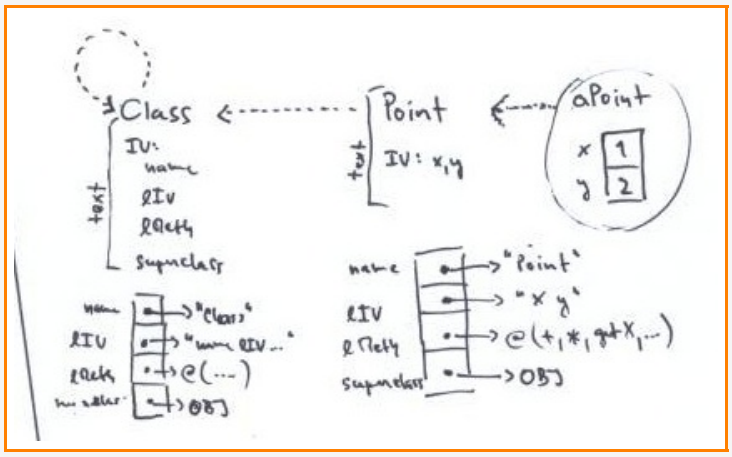

Fig. (5): Une classe et une méta-classe sont structurellement et fonctionnellement identiques

- `Object` est la racine de l'arbre d'héritage,
- `Class` est instance d'elle-même, sous-classe de `Object` et racine de l'arbre d'instantiation.

L'intérêt de cette solution est de permettre l'association explicite de n'importe quelle métaclasse à une classe C indépendamment de la relation d'héritage entre C et sa superclasse. Une classe peut ainsi être abstraite (instance de la méta-classe AbstractClass) sans que ses sous-classes le soient.

L'inconvénient de cette solution est qu'elle peut poser des problèmes de compatibilité (voir section 4.4.7). Clos laisse au développeur le soin de résoudre ces problèmes éventuels de compatibilité via la définition de fonctions ad.hoc.

4.4.2 Motivations de la solution Smalltalk

La solution Smalltalk a été conçue avec deux objectifs[2] :

- Donner la possibilité de définir des méthodes sur les métaclasses tout en rendant les métaclasses invisibles au développeur ou au chef de projet qui n'a pas envie de les voir. Ceci est fait via l'environnement de programmation.
- Empêcher les problèmes de non compatibilité.

4.4.3 Coeur du système de métaclasses Smalltalk

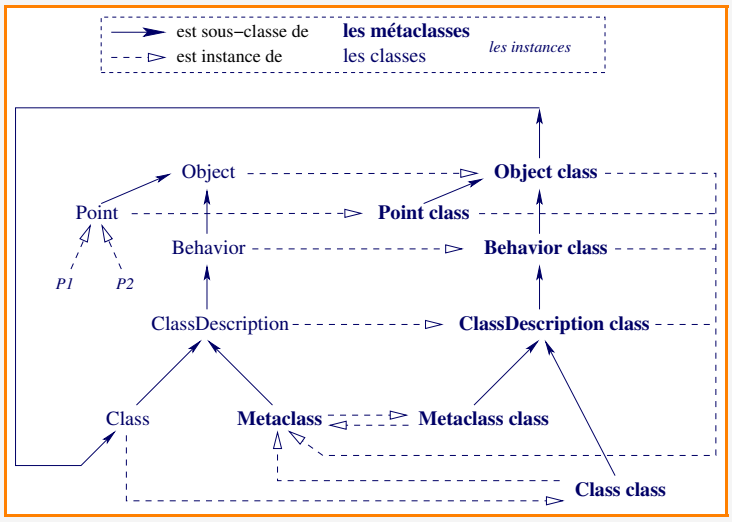

Fig. (6): Classes et Métaclasses : Hiérarchies d'héritage et d'instantiation. (figure : G.Pavillet)

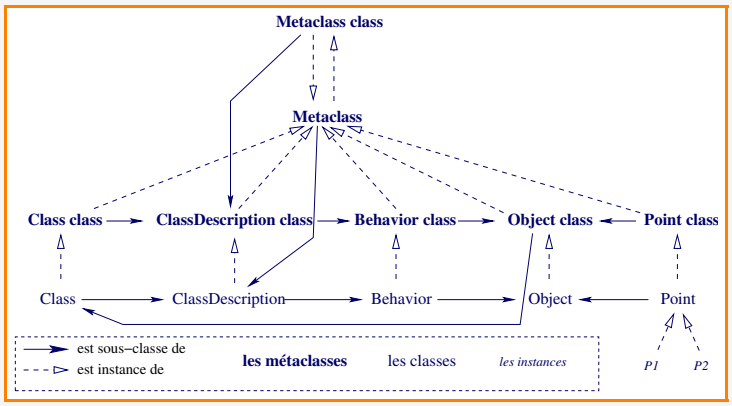

Fig. (7): Classes et Métaclasses : couches d'instantiation. (figure : G.Pavillet)

Les points clé :

[2]Il y a donc deux façons de considérer ce système, la première est de l'utiliser sans chercher à entrer dans le détail de sa réalisation et d'apprécier son efficacité. Le bouton "class" du browser est dédié à la programmation des méthodes de classe et il n'est pas utile de comprendre la mécanique interne pour l'utiliser. La seconde est d'aller y voir de plus près pour le plaisir de comprendre, ce qui prend tout son sens dans le cadre d'un cours sur la réflexivité.

- Toute classe `C` possède une unique métaclasse référençable via l'expression `C class`.
- Les deux hiérarchies d'héritage entre classes et d'héritage entre métaclasses sont isomorphes.
- `Object class` est la racine de la sous-hiérarchie des métaclasses.
- Le graphe d'héritage étant un arbre avec donc une racine unique (`Object`), le lien entre les hiérarchies de classes et de métaclasses est fait au niveau de la métaclasse `Object class` qui hérite de la classe `Class`. Ceci se comprends, selon la sémantique usuelle[3] du lien d'héritage, ainsi : une `Object class`, par exemple `Object)`, est une sorte de classe.
- Il y a 4 couches d'instantiation. Les objets (couche 1) (par exemple un point), instances des classes (par exemple `Point`, instances des métaclasses (par exemple `Point class`), instances d'une unique méta-méta-classe (`Metaclass`).
  Le problème de regression infinie dans les descriptions est reglé par le fait que `metaclass` est instance de `metaclass class`, qui est une métaclasse standard de la couche 3, instance de `Metaclass`. `Metaclass class` a exactement le même format que toutes les autres métaclasses.

4.4.4 Structure des classes et des métaclasses (cf. fig. 8)

```
ProtoObject #()
    Object #()
        Behavior #('superclass' 'methodDict' 'format')
            ClassDescription #('instanceVariables' 'organization')
                Class #('subclasses' 'name' 'classPool' 'sharedPools' 'environment' 'category')
                    [ ... all the Metaclasses ... ]
                Metaclass #('thisClass')
```

Fig. (8): Les classes "Class" et "Metaclass" (et leurs attributs) définissant les classes et les méta-classes. Elles ont en commun tout ce qui est hérité de "ClassDescription".

La classe `Class` déclare, en fermant transitivement la relation "est-sous-classe-de" les attributs : "superclass methodDict format instanceVariables organization subclasses name classPool environment category" sharedPools.

Toute instance de `Class`, directe ou indirecte, pourra posséder une valeur pour chacun d'eux. Par exemple la valeur de l'attribut `instanceVariables` de la classe `Point` (instance de `Point class` qui hérite de `Class`) est `'x y z'`. Les variables de classes sont rangées dans l'attribut `classPool` qui dans le cas de la classe `Point` est vide. Il faudra que je trouve un meilleur exemple.

La classe `Metaclass` déclare les attributs : "superclass methodDict format instanceVariables organization thisClass".

Remarquons qu'une métaclasse possède un champs `instanceVariables`, un champs `thisClass` mais pas de champs `classPool`.

`thisClass`[4] référence la classe, unique instance (schéma Singleton) de la métaclasse. `thisClass` est utilisable dans les méthodes d'instance de la classe `Metaclass` qui n'a aucune sous-classe[5].

[3] AKO : a kind of, est une sorte de ... une voiture est une sorte de véhicule.

[4] Dont le nom est mal choisi.

[5] Exercice difficile : créer une sous-classe MC de la classe `Metaclass`, qui définirait un nouveau type généraliste de meta-classes. La difficulté n'est pas dans la création de cette classe mais, comme les metaclasses sont crées automatiquement par le système, de modifier le système de création des classes pour qu'une nouvelle lasse ne soit pas une instance de `Metaclass` mais de `MC`. Je n'ai pas encore fait cet exercice.

Une métaclasse peut déclarer des attributs mais pas de variable de classe.

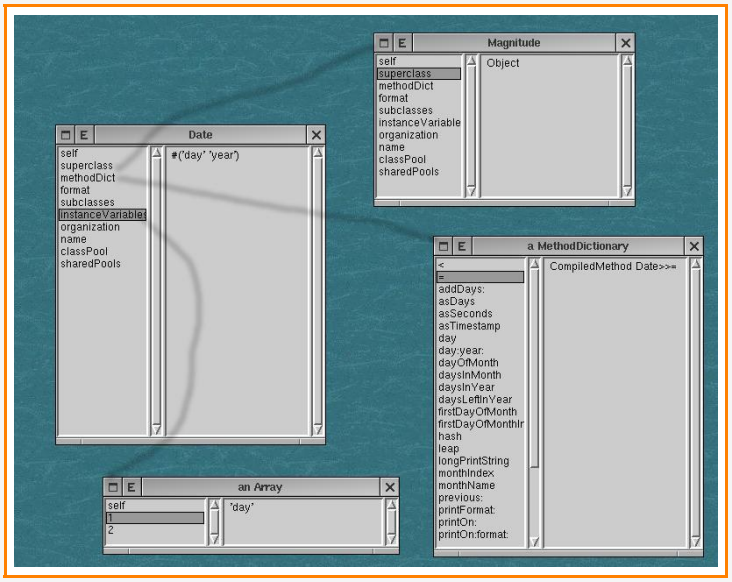

Fig. (9): l'Inspection d'une classe montre que ses champs sont conformes à la déclaration des attributs réalisés dans la classe `Class`.

4.4.5 Variable de classe et variable d'instance de métaclasse

Un point difficile est la compréhension de la différence entre variable de classe et variable d'instance[6] de métaclasse et ce d'autant plus qu'il n'y a pas de différence entre méthode de classe et méthode d'instance de métaclasse.

Les méthodes de classe d'une classe `C` sont les méthodes d'instance de `C class`. Les variables de classes d'une classe `C` n'ont rien à voir avec les variables d'instance de `C class`.

Les variables de classe sont accessibles dans les méthodes d'instance et dans les méthodes de classe. Elle servent à stocker des valeurs communes à toutes les instances d'une même classe, sans avoir à passer par la métaclasse.

Les variables d'instance de métaclasse ne sont accessibles que dans les méthodes de classe. Ce sont les attributs standard des métaclasses considérées comme des classes standard. Si l'on souhaite réaliser une "memo classe" qui mémorise la liste des ses instances, la solution naturelle est d'utiliser une variable d'instance de métaclasse.

4.4.6 Utilisation des méta-classes : création d'instances initialisées

Une classe complète avec ses méthodes de création d'instances : la classe `Pile`.

```
1 Object subclass: #Pile
2       instanceVariableNames: 'index buffer '
3       classVariableNames: ''
4       poolDictionaries: ''
5       category: 'Pile'!
6 ...
```

[6] On rappelle que le terme "variable d'instance" dénote ce qui est maintenant usuellement appelé "attribut" dans les autres langages.

```
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!
Pile class
        instanceVariableNames: ''!

!Pile class methodsFor: 'creation'!
example
        "self example"
        "Rend une pile d'entiers de taille 10, pleine."
        | aStack |
        aStack := self new: 10.
        1 to: aStack size do: [:i | aStack push: i].
        ^aStack!

new
        "On fixe la taille par défaut à 10"
        ^super new initialize: 10!

new: taille
        ^super new initialize: taille!

taille: taille
        ^self new: taille! !
```

4.4.7 Avantage du modèle Smalltalk : La compatibilité des métaclasses

La solution Smalltalk ne pose pas de problèmes de compatibilité.

Les problèmes de compatibilité surviennent avec les architectures à la `Objvlisp` lors de phase d'abstraction (passage au niveau meta - compatibilité ascendante) ou de la phase de réification (passage au niveau réifié - compatibilité descendante.

Problème de compatibilité Ascendante

Soient :

- une classe A, sa methode `foo` : 'self class bar'

- une classe B sous-classe de A

- la classe de A et sa méthode `bar`

- la classe de B, non sous-classe de la classe de A, et ne définissant pas `bar`

- une instance b de B

alors : 'b foo' provoque une erreur.

Cette situation est impossible avec le modèle Smalltalk où la classe de B ne peut pas ne pas être une sous-classe de la classe de A.

Problème de Compatibilité Descendante

Métaclasse MA, méthode bar : 'self new foo'

Métaclasse MB, sous classe de MA,

'mb bar' peut provoquer une erreur.

4.4.8 Inconvénient du modèle Smalltalk

Il est impossible de découpler les hiérarchies de classes et de méta-classes. Ce qui pose des problèmes d'expression.

Par exemple si on crée une méta-classe `AbstractClass` dont les instances ne peuvent être instanciées (par une redéfinition appropriée de *new)*. Alors toutes les sous-classes d'une instance de AbstractClass hériteront de cette redéfinition de *new*. Or, les classes abstraites ont bien sûr des sous-classes concrètes.

4.5 Les Méta-objets permettant d'accéder à la pile d'exécution

Smalltalk permet de manipuler chaque bloc (frame) de la pile d'exécution comme un objet de première classe avec une politique de "si-besoin" car la réification est une opération coûteuse.

Par exemple, le code suivant implante les structures de contrôle `catch` et `throw` permettant de réaliser des échappements à la Lisp (réalisation de branchements non locaux), que l'on peut voir comme les structures de contrôle de base nécessaires à l'implantation de mécanismes de gestion des exceptions.

`catch` permet de définir un point de reprise à n'importe quel point d'un programme et `throw` interromp l'exécution strandard et la fait reprendre à l'instruction qui suit le `catch` correspondant.

Application : interruption d'une recherche dés que l'on a trouvé l'élément recherché pour revenir à un point antérieur de l'exécution du programme.

```
1  !Symbol methodsFor: 'catch-throw'!
2
3  catch: aBlock
4      "le nom de la méthode et le receveur courant font office de marque dans la pile"
5      aBlock value.!
6
7  throw: aValue
8          "Looks for a catch, the mark of which is self,
9          if found, transfer control while executing recovery blocks."
10         | catchMethod currentContext |
11         currentContext := thisContext.
12         catchMethod := Symbol compiledMethodAt: #catch:.
13         [currentContext method == catchMethod and: [currentContext receiver == self]]
14                 whileFalse: [currentContext := currentContext sender].
15         thisContext sender: currentContext sender.
16         ^aValue! !
```

4.6 Méta-objets pour accéder au compilateur et aux méthodes compilées

Le compilateur Smalltalk est écrit en Smalltalk et utilisable dans tout programme. Cela permet de définir dynamiquement de nouvelles classes et méthodes.

L'exemple suivant est extrait de la réalisation d'un mini-tableur en Smalltalk. Dans cette exemple, les formules associées aux cellules sont représentées par des méthodes définies dynamiquement par le programme, analysées lexicalement et compilées à la volée

Pour savoir comment ajouter, une fois compilée, une méthode à une classe, il faut étudier les protocoles définis sur les classes `Behavior` et `ClassDescription`.

```
1  Model subclass: #Cellule
2      instanceVariableNames: 'value formula internalFormula dependsFrom '
```

```
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Tableur'!
```

```
!Cellule methodsFor: 'compile formula'!
compileFormula: s
    "Analyse lexicale, puis syntaxique puis generation de code pour la formule s"
    | tokens newDep interne methodNode |
    tokens := Scanner new scanTokens: s.
    newDep := (tokens select: [:i | self isCaseReference: i]) asSet.
    interne := 'execFormula\ | '.
    newDep do: [:each | interne := interne , each , ' '].
    interne := interne , '|\ '.
    newDep do: [:each | interne := interne , each , ' := (Tableur current at: #' , each , ')
        value.\ '].
    interne := (interne , '^' , s) withCRs asText.
    methodNode := UndefinedObject compilerClass new
                compile: interne
                in: UndefinedObject
                notifying: nil
                ifFail: [].
    internalFormula := methodNode generate.
    ^newDep! !
```

```
!Cellule methodsFor: 'exec formula'!
executeFormula
    formula isNil
        ifFalse:
            [UndefinedObject addSelector: #execFormula withMethod: internalFormula.
            ^nil execFormula]
        ifTrue: [self error]!

update: symbol
    symbol == #value ifTrue: [self setValue: self executeFormula]! !
```