Université de Montpellier
UFR des Sciences - Département Informatique
Master Informatique

Méta-Programmation et Réflexivité.

Models@Runtime

Notes de cours

Christophe Dony

1 Contenu du cours

But : Utilisation et Construction de Langages Réflexifs autorisant la Méta-Programmation.

(#meta-programming #dynamic adaptability #model@runtime)

1. **Définitions** : modèles, méta-modèles, réflexivité, méta-programmation,
2. **Utilisation de systèmes réflexifs**
   contexte du développement par objet.
   (a) Méta-Programmation, Utilisation de méta-objets (Smalltalk, Clos, ... Ruby, Pharo, Python, ...), pour interroger ou manipuler :
   — tout ou partie des programmes (par exemple la classes, les hiérarchies de classes, les attributs, les méthodes), tels que représentés en machine,
   — tout ou partie de la machinerie d'exécution des programmes (par exemple le compilateur, la pile d'exécution).
   (b) Méta-Programmation, Réalisation, par programme, de (méta-)modèles exécutables, de nouveaux types de classes, de nouveaux types de méthodes, de nouvelles structures de contrôle, de nouveaux langages (eg. DSL).
   ...
3. **construction de systèmes réflexifs**
   (a) Intégration de méta-classes explicites dans un langage à objets, avec son bootsrap.
   (b) construction d'un interpréteur méta-circulaire
   (c) Construction d'un interpréteur méta-circulaire réflexif
4. **Projets Possibles** - Exploration ouverte multi-langages et multi-thématique

2 Préambule

Définitions préalables

Réflexivité

"La réflexivité est une démarche méthodologique en sciences sociales consistant à appliquer les outils de l'analyse à son propre travail ou à sa propre réflexion et donc d'intégrer sa propre personne dans son sujet d'étude." [wikipedia : Réflexivité-(sciences-sociales)]

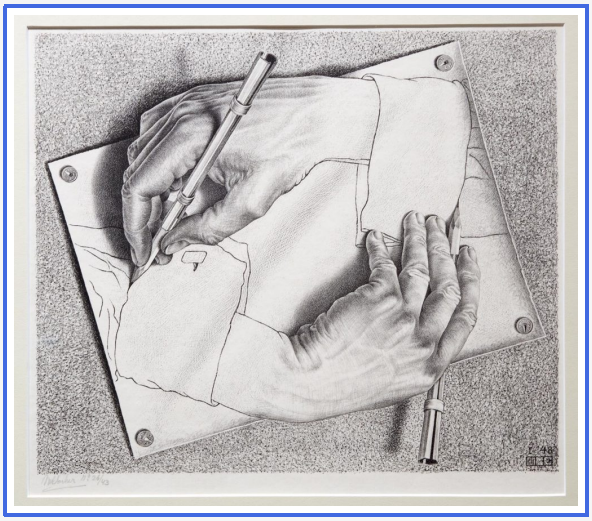

Figure (1) *– Maurits Cornelis Escher, "Drawing Hands", 1948*

Système Réflexif (informatique) : Système offrant à ses utilisateurs une représentation, un modèle, de lui-même, utilisable pendant son exécution.

Méta-programmation : programmation d'un système informatique utilisant sa nature réflexive, i.e. le modèle, qu'il offre de lui-même.

Idée : La peinture ou la photographie sont des langages suffisamment puissants pour créer des systèmes réflexifs (auto-portraits par exemple).

Figure (2) *– Diego Velazquez "Las-Meninas", 1656*

Figure (3) *– Vivian Maier, auto-portrait*

Les systèmes réflexifs et la méta-programmation, Intérêt, Impact

— Réaliser des Systèmes auto-adaptables (*Self-Adaptive Software Systems*)
https://conf.researchr.org/event/seams-2018/seams-2018-papers-self-adaptive-software-systems-are-essent
— ...
— Permettre la construction d'environnements de développement du logiciel efficaces,
Exemple : réalisation d'un mode d'évaluation en pas à pas.

Exemple : déverminer des programmes non interruptibles (thèse de S. Costiou)
Adaptation non-anticipée de comportement : application au déverminage de programmes en cours d'exécution

— ...
— Appliquer l'Ingénierie dirigée par les modèles à l'exécution *(models@runtime)*
https://mrt21.bitbucket.io/
— ...
— Permettre aux utilisateurs d'étendre, corriger, améliorer les systèmes qu'ils utilisent
— Exemple de systèmes adaptable dynamiquement et par ses utilisateurs : tout logiciel possédant un menu "Préférences".
...
— Exemple de systèmes adaptable dynamiquement, par ses utilisateurs et par programme : l'éditeur *Emacs*.
Emacs inclus un langage (Emacs-Lisp) offrant des fonctions pour accéder aux structures de données de l'éditeur [1].

```
(defun split5 ()
  (interactive)
  (split-window-horizontally)
  (split-window-vertically)
  (split-window-horizontally)
  (split-window-vertically)
  )

(split5)
```

Listing (1) *– une méta-fonction* `Emacs-Lisp`*, qui enrichit les fonctions offertes par l'éditeur*

— Réaliser des **systèmes** à **(méta-)modèle** extensible et/ou modifiable.

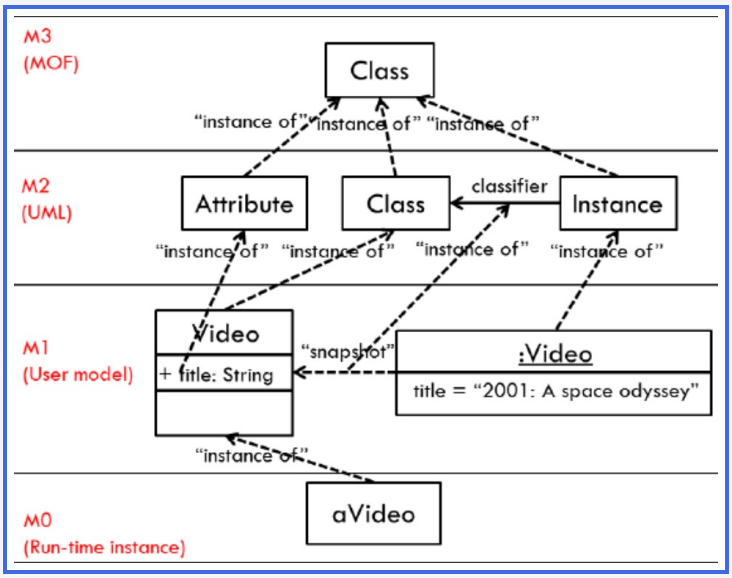

Figure (4) *– M2 : Méta-Modèle de la programmation Objet*

— Un exemple de système à méta-modèle extensible : CLOS

1. "L'idée intéressante à propos d'Emacs était qu'il possédait un langage de programmation et que les commandes d'édition de l'utilisateur étaient écrites dans ce langage de programmation interprété, de telle sorte qu'on pouvait charger de nouvelles commandes dans l'éditeur tout en éditant. On pouvait éditer les programmes qu'on utilisait tout en continuant à s'en servir pour l'édition. Donc, nous avions un système qui était utile à autre chose qu'à programmer et qu'on pouvait programmer pendant qu'on l'utilisait." [Richard.M.Stallman - http ://www.gnu.org/gnu/rms-lisp.html]

```
(defclass singleton-class (standard-class)
... exercice ..
)

(defclass singleton1 (singleton-class)
...)

defclass singleton2 (singleton-class)
...)
```

Listing (2) – *Singleton en CLOS*

à ne pas confondre avec l'implantation du schéma *Singleton* dans les langages non réflexifs :

```
public class Singleton{
   private static Singleton singleton = new Singleton();
   private Singleton() { }
   public static Singleton getInstance( ) { return singleton; }
}
```

Listing (3) – *Singleton en Java*

— **Réaliser de nouveaux langages de programmation**
"Metacircular evaluator : an evaluator written in the same language it evaluates."
"Reasons to look at metacircular evaluators : better understanding of language semantics ; allows us to experiment with different semantics."
https://courses.cs.washington.edu/courses/cse341/98au/scheme/eval-apply.html

```
(define (happy-eval e)
  (if (equal? e 'happy?)
      'YES
      (eval e)))

(define (toplevel)
  (while true
    (begin
      (print (happy-eval (read)))
      (display "\n"))))

(toplevel)
```

Listing (4) – *Un évaluateur méta-circulaire d'un nouveau langage, nommé* **happy**, *écrit en DrScheme*

— **Réaliser de nouveaux langages de programmation spécialisés**
Domain Specific Languages :
https://www.jetbrains.com/mps/concepts/domain-specific-languages/

3 Définitions

3.1 Modèles, méta-Modèles

modèle : description d'un système (une carte routière, le plan d'une machine, le code source d'un programme ...)

modèle Objet : Informatique, modèle décrivant un système par les objets qui le constituent et les opérations qu'ils savent effectuer

transformation de modèle (IDM) : modifier, traduire, enrichir un modèle

méta : (du grec) après, à propos de, qui parle de ...

meta-donnée : donnée relative à une (des) donnée(s), par exemple méta-donnée indiquant que 60% des données de telle application sont de type numérique

méta-modèle : modèle relatif à, décrivant, un (des) modèle(s)

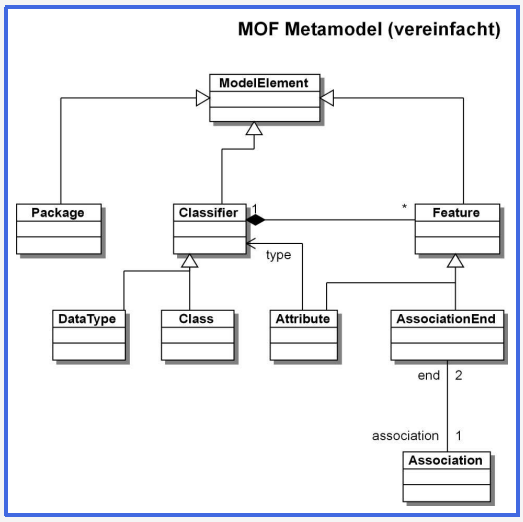

Figure (5) – *Exemple : le méta-modèle MOF de l'OMG pour l'ingénierie dirigée par les modèles (https://en.wikipedia.org/wiki/File:MOF_Metamodel_144dpi.jpg)*

méta-langage : langage permettant de décrire et/ou de manipuler des langages, par exemple *Lex* ou *Yacc*

méta-niveau : relativement à un niveau donné D où se situent des entités que l'on utilise, le méta-niveau est celui où se trouvent (où sont définies) les entités qui modélisent (décrivent) celles de D

Dans la pratique tout commence donc par un axiome ou un amorçage *(bootstrap)*.

programmation de niveau méta : programmation des entités d'un niveau méta, par exemple programmation d'un compilateur ou d'une machine virtuelle.

Par exemple, un programmeur d'un compilateur *Java* définit les structures pour représenter les classes, leurs instances, la pile d'exécution, et plus globalement les entités qui permettent d'écrire et d'exécuter un programme *Java*.

méta-programmation : ...

3.2 Méta-Programmation

méta-programmation : programmation de niveau méta réalisée au niveau de base,

exemples :

- programmation d'un compilateur Java en Java
- programmation d'un interpréteur *Scheme* en *Scheme*
- accès aux modèles, classes ou méthodes, dans le texte et durant l'exécution des programmes

```
1        Person p = new Person(''Jean'');
```

```
2       Class pClass = p.getClass();
3       Class pSuperclass = pClass.getSuperclass();
4       Method gMeths[] = pClass.getDeclaredMethods();
5       Method getAge = pClass.getDeclaredMethod("getAge", null);
6       System.out.println(getAge.invoke(p, null));
```

Listing (5) – *Exemple de méta-programmation en Java utilisant le* package `reflect`

Plongement : Offrir la méta-programmation nécessite de "plonger" le niveau méta dans le niveau de base :

- de le représenter

- de rendre cette représentation accessible au niveau de base.

3.3 Représentation par objets d'un méta-niveau : méta-objets

Une description d'un monde en terme d'objets repose sur 3 entités fondamentales : les objets (instances), les classes et les propriétés (attributs, méthodes), comme l'explicite le niveau M2 de l'architecture de méta-modélisation de l'OMG.

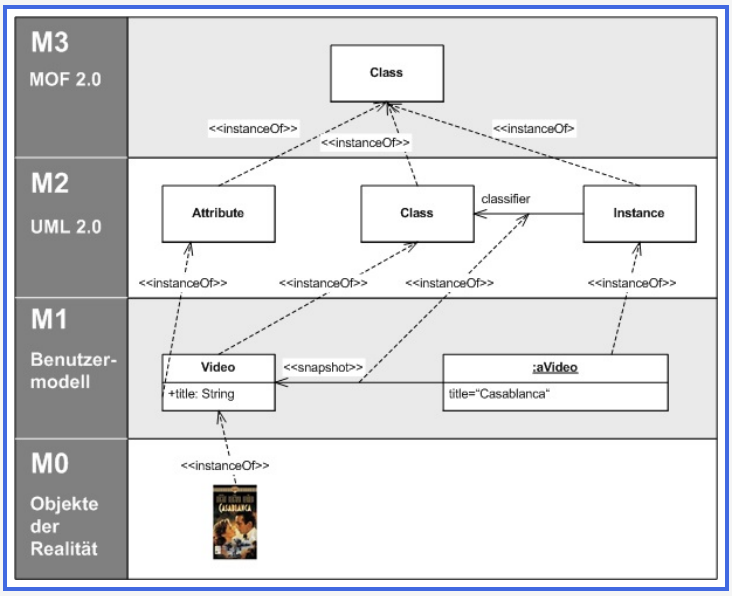

Figure (6) – *OMGs-four-layer-metamodel-architecture - Wikipedia*

Meta-objet objet du niveau de base représentant une entité du niveau méta modélisée en OOD.
Exemples : méta-objets représentant les classes, attributs, méthodes, exceptions, voire les structures de la machine virtuelle telle la pile d'exécution.
Les meta-objets accompagnés des méthodes définies sur leurs classes (des méta-classes) offrent une **bibliothèque** et un **protocole** pour la méta-programmation.

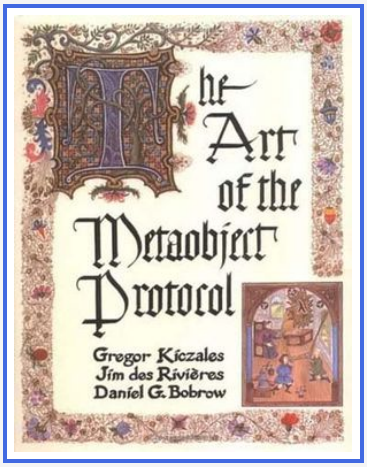

Figure (7) *– Livre décrivant la bibliothèque de méta-programmation du langage Common-Lisp Object System*

Voir : Javascript's Meta-object Protocol

Attention : un "meta-objet" est un objet standard, donc utilisable comme un autre ; son nom signifie qu'il représente une entité du niveau méta.

La méta-programmation s'intéresse aussi bien aux structures de données qu'aux mécanismes de calcul et donc à l'observation et modification du processus d'interprétation, ainsi qu'à la génération de code à la volée.

La méta-programmation nécessite des systèmes ou des langages réflexifs.

3.4 Réflexivité

Réflexivité : Capacité qu'a un système à donner à ses utilisateurs une représentation, un modèle, de lui-même en relation (ou connexion) causale avec sa représentation effective (en machine dans le cas d'un système informatique).

Connexion causale si la représentation effective (RE) change, la représentation que voit l'utilisateur (RU) change également, ... et inversement quand la relation est symétrique.

Réification : (latin *res* : la chose) procédé consistant à à "chosifier" (rendre tangible, accessible) une entité du nivau méta.

Dans le cas d'une réification dans un système à objet, réifier est fabriquer un méta-objet.

Introspection : connexion causale unidirectionnelle, RU reflète RE, si RE change RU change mais la réciproque est fausse. *(Java Reflect*, inspecteur *Smalltalk)*

Intercession : connexion causale complète. (exemple : classes en Smalltalk ou Pharo)

Réflexivité structurelle : réification des stuctures (données), par exemple classes, méthodes.

Exemple :

```
1  import java.lang.Reflect;

3  public class TestReflect {

5      public static void main (String[] args) throws NoSuchMethodException{
```

```
        GraphicCptBean g = new GraphicCptBean();
        Class gClass = g.getClass();
        Class gSuperclass = gClass.getSuperclass();
        Method gMeths[] = gClass.getDeclaredMethods();
        Method getCompteur = gClass.getDeclaredMethod("getCompteur", null);
        try {System.out.println(getCompteur.invoke(g, null));}
        catch (Exception e) {}
    }
}
```

Réflexivité comportementale : réification des mécanismes de calcul, donnant par exemple accès au processus d'interprétation.

Exemple :

```
(defun mon-evaluateur (form env)
  (format t "Eval --> ~S ~%" form)
  (let ((res (evalhook form 'mon-evaluateur nil env)))
    (format t "Eval <-- ~S~%" res)
    res))

(defun mon-debugger (form)
  (let ((*evalhook* 'mon-evaluateur))
    (eval form)))

(defun exemple1 () (mon-debugger '(+ 3 (* 4 5))))x1

(defun exemple2 () (mon-debugger '(factorial 4)))
```

Listing (6) *– hook = crochet, possibilité de "crocheter" l'interpréteur en* CLisp*, pour écrire un debuggeur d'évaluation en pas à pas.*

```
(defun my-ev2 (form env)
    (format t "Eval --> ~S , ~S~%" form (aref env 0))
    (let ((res (evalhook form 'my-ev2 nil env)))
      (format t "Eval <-- ~S~%" res)
      res))

(evalhook '(factorial 4) 'my-ev2 nil)
```

Listing (7) *– hook = crochet, possibilité de "crocheter" l'interpréteur en* CLisp*, pour voir les environnements d'évaluation*

Réflexivité à la Compilation/Chargement/Execution L'accès au méta-modèle peut être réalisé

- à la compilation (voir par exemple *OpenJava)*)
- au chargement du code (voir par exemple *Javassist)*
- ou à l'exécution (tous les langages réflexifs).

, **Langage réflexif** : nom généralement donné aux langages offrant la réflexivité à l'exécution.

4 Utilisation de systèmes réflexifs #1, Métaprogrammation en Pharo

4.1 Tout est Objet

Un ensemble complet de méta-objets et de méthodes d'introspection et intercession offrent la possibilité de méta-programmer presque tout le système.

```
1 class "-> SmallInteger"
2 class class "->SmallInteger class"
3 class class class "->Metaclass"
4 class class class class "->Metaclass class"
5 class class class class class "->Metaclass"
```

4.2 Méta-objets représentant les éléments primitifs (rock-bottom objects)

Chaque éléments d'un type primitif (nombres, caractères, booléens, chaînes, tableaux) est représenté par un méta-objet qui permet son utilisation comme un objet standard. On peut en particulier lui envoyer des messages.

```
5 factorial "entier"
'abcde' at: 3 "chaîne de caractère"
#($a $b) reverse "tableau"
true ifTrue: [#ofCourse] "booléen"
nil isNil "null pointer ou UndefinedObject"
selecteur := #facto , #rial "symboles"
```

Un type primitif est représenté par une classe mais son implantation n'est pas entièrement définie par cette classe et réside partiellement dans la machine virtuelle.

Il est possible de modifier les méthodes de ces classes (forcément dangereux) et d'en créer de nouvelles (exemple, la méthode `factorial` sur la classe `Integer`).

La hiérarchie des classes définissant les nombres.

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

4.3 Méta-objets représentant les éléments non primitifs

4.3.1 Les symboles et l'envoi de message calculé

Les symboles sont des chaînes de caractères immutables (non modifiables) et uniques (il n'y a pas deux symboles constitués des mêmes caractères).

Les symboles [2] sont les méta-objets des identificateurs des programmes.

```
sel := #facto , #rial
sel := sel asSymbol
5 perform: sel
```

Listing (8) – *Envoi de message dont le sélecteur est calculé*

```
m := Integer compiledMethodAt: #factorial.
m class "->CompiledMethod"
m name "->'Integer>>#factorial' "
m valueWithReceiver: 5 arguments: #() "->120"
```

Listing (9) – *Récupération d'une méthode via son nom et invocation directe*

4.3.2 Les classes comme des objets (donc comme des "r-values" standards)

Une classe peut être utilisée universellement en position de *r-value*.

```
1 isKinfOf: Integer "-> true''
```

```
v := Integer.
1 isKindOf: v. "-> true''
```

```
Integer isKindOf: Class. "-> true''
```

Les méta-objets permettant d'utiliser les classes comme des objets standards sont définies par les méta-classes, que nous étudieront plus tard.

Application au contrôle de la généricité paramétrique

La "généricité paramétrique" est la capacité à permettre le paramétrage des structures de données composites, commes les collections par exemple, par le type des éléments qu'elles utilisent, soit pour réutiliser (en typage statique) soit pour contraindre (en typage dynamique).

En typage statique (voir les génériques *Java*, les templates *C++*) elle évite d'avoir à réécrire du code et permet la vérification des contraintes à la compilation.

Les classes comme "rvalues", accompagnées d'une primitive de test de sous-typage, permettent en typage dynamique un contrôle dynamique sur les types.

Exemple :

```
Pile subclass: #PileTypee
    instanceVariableNames: 'typeElements'
    classVariableNames: ''
    category: 'TP1'

push: element
    (element isKindOf: typeElements)
        ifTrue: [ super push: element ]
```

2. Note : les symboles en JavaScript, https://www.keithcirkel.co.uk/metaprogramming-in-es6-symbols/

```
        ifFalse: [ self error: 'Impossible d''empiler ' , element printString ,
                               ' dans une pile de ' , typeElements printString]
```

4.3.3 Un méta-objet pour représenter la valeur `nil` (`null`)

`nil` (`null` en *Java)* est la valeur par défaut du type référence, donc affectée à toute *r-value* (variable, case de tableau, etc) non explicitement initialisée.

Nil est l'unique instance *(Singleton)* de la classe `UndefinedObject` sur laquelle il est possible de définirdes méthodes.

Il est donc possible d'envoyer un message à `nil`.

L'exception `nullPointerException` n'existe pas !

Le nom de la classe est certainement discutable.

```
nil isNil "- >true"
nil class "-> UndefinedObject"
```

Application à l'implantation des collections

La classe *UndefinedObject* offre une alternative originale pour l'implantation de certains types récursifs, par exemple `List` ou `Arbre`.

```
1 type List =
2     Empty |
3     Tuple of Object * List
```

dont certaines fonctions se définissent par une répartition sur les différents constructeurs algébriques :

```
1 length :: List -> Int
2 lentgh Empty          = 0
3 length Tuple val suite = 1 + (lentgh suite)
```

Mise en oeuvre (extrait 1)

```
Object subclass: #List
    instanceVariableNames: 'val suite'
    classVariableNames: ''
    poolDictionaries: ''
    category: '...'!

!List methodsFor: 'accessing' !
suite
    ^suite !
first
    ^val !
```

```
first: element suite: uneListe
    val := element.
    suite := uneListe. !!

!List methodsFor: 'manipulating' !
addFirst: element
    ^List new first: element suite: self !
length
    ^1 + suite length !
append: aList
    ^(self suite append: aList) addFirst: self first ! !
```

Mise en oeuvre (extrait 2)

```
!List class methodsFor: 'instance creation' !
with: element
    ^super new first: element suite: nil ! !
```

Mise en oeuvre (extrait 3)

```
!UndefinedObject methodsFor: 'ListManipulation'!
addFirst: element
    ^List with: element !

length
    ^0 !

append: aList
    ^aList !
```

4.3.4 Les fermetures et la définition de nouvelles structures de contrôle

Les structures de contrôle sont réalisées en Smalltalk par des méthodes définies sur les classes `Boolean` (conditionnelles, ET, OU, *BlockClosure* (boucle "tantque"), `Integer` (boucle "for").

```
!True methodsFor: 'Controlling'!

ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
    "Answer with the value of trueAlternativeBlock. Execution does not
    actually reach here because the expression is compiled in-line."

    ^ trueAlternativeBlock value
```

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

```
!Boolean methodsFor: '*ExosPharo-7-controlling' !
```

```
ifNotTrue: aBlock
    "la lambda passée en argument sera exécutée si le receveur est false"
    ^self ifFalse: aBlock! !
```

4.4 Les objets comme données de base

```
!Object methodsFor 'Introspection'!

instVarAt: index
instVarAt: index put: aValue
instVarNamed: aString
instVarNamed: aString put: aValue
```

4.5 Les méta-objets pour représenter les classes

Il existe au niveau de base un méta-objet pour toute classe définie dans le système.

Un nom de classe est un symbole réservé à l'identification du méta-objet correspondant.

Les classes `Behavior`, `ClassDescription` et `Class`, définissent les méthodes permettant de manipuler les classes comme des objets

4.5.1 `Behavior` et la méthode `new`

```
1 Object subclass: #Behavior
2     instanceVariableNames: 'superclass methodDict format layout'
3     classVariableNames: 'ClassProperties ObsoleteSubclasses'
4     package: 'Kernel-Classes'
```

My instances describe the behavior of other objects. I provide the minimum state necessary for compiling methods, and creating and running instances. Most objects are created as instances of the more fully supported subclass, Class, but I am a good starting point for providing instance-specific behavior.

exemple :

```
1 new
2     "Answer a new initialized instance of the receiver (which is a class) with no indexable variables.
          Fail if the class is indexable."

4     ^ self basicNew initialize
```

4.5.2 ClassDescription

I add a number of facilities to basic Behaviors :

— Named instance variables

— Category organization for methods

— The notion of a name of this class (implemented as subclass responsibility)

— The maintenance of a ChangeSet, and logging changes on a file

— Most of the mechanism for fileOut.

4.5.3 Class

I add a number of facilities to those in ClassDescription :

— A set of all my subclasses (defined in ClassDescription, but only used here and below)

— A name by which I can be found in a SystemDictionary

— A *classPool* for class variables shared between this class and its metaclass

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

Classes : **CompiledMethod**, **Compiler** (toute variante locale), Parser

4.6.1 Exemple1

```
testCompile
    "BaseExos testCompile"
    | multMethod res |
    multMethod := OpalCompiler new
        source: 'mult: y\ ^self = 0 ifTrue: [0] ifFalse: [^y + ((self - 1) mult: y)]' withCRs;
                class: Integer;
                compile.
    Integer addSelector: #mult: withMethod: multMethod.
    res := multMethod valueWithReceiver: 2 arguments: #(3).
    ^(3 mult: 4) + res
```

4.6.2 Exemple2

L'exemple suivant est extrait de la réalisation dans une ancienne version de Smalltalk d'un mini-tableur. Dans cet exemple, les formules associées aux cellules sont représentées par des méthodes définies dynamiquement par le programme, lexicalement analysées et compilées à la volée. L'interface avec le compilateur est à adapter selon la version du langage utilisée (voir l'exemple précédent avec OpalCompiler dans le cas de *Pharo*.

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

```
Model subclass: #Cellule
    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]! !
```

4.7 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 `returnToCatchWith:` interromp l'exécution strandard et la fait reprendre à l'instruction qui suit le `catch` correspondant (le symbole receveur détermine la correspondance).

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.

```
!Symbol methodsFor: 'catch-throw'!

catch: aBlock
    "execute aBlock with a throw possibility"
    aBlock value.!

returnToCatchWith: aValue
        "Look down the stack for a catch, the mark of which is self,
        when found, transfer control (non local branch)."
        "Version Visualworks"
        | catchMethod currentContext |
        currentContext := thisContext.
        catchMethod := Symbol compiledMethodAt: #catch:.
        [currentContext method == catchMethod and: [currentContext receiver == self]]
                whileFalse: [currentContext := currentContext sender].
        thisContext sender: currentContext sender.
```

```
17          ^aValue! !
```

Listing (10) *– version Visualworks-Smalltalk*

```
1  !Symbol methodsFor: 'catch-throw'!

3  catch: aBlock
4      "execute aBlock with a throw possibility"
5      aBlock value.!

7  returnToCatchWith: aValue
8          "version Pharo6.1."
9          | catchMethod currentContext |
10         currentContext := thisContext.
11         catchMethod := Symbol compiledMethodAt: #catch:.
12         [currentContext method == catchMethod and: [currentContext receiver == self]]
13                 whileFalse: [currentContext := currentContext sender].
14         currentContext return: aValue.
15         ^aValue
```

Listing (11) *– version Pharo7*

Exemple d'utilisation :

```
1  ^#Essai catch: [
2          Transcript show: 'a';cr.
3          Transcript show: 'b';cr.
4          Transcript show: 'c';cr.
5          #Essai returnToCatchWith: 22.
6          Transcript show: 'd';cr.
7          33]
```

Listing (12) *– should display a b c (not d) in the Transcript and return 22 (not 33)*

5 Etude de différents modèles de méta-classes

5.1 Plongement des classes dans le niveau de base

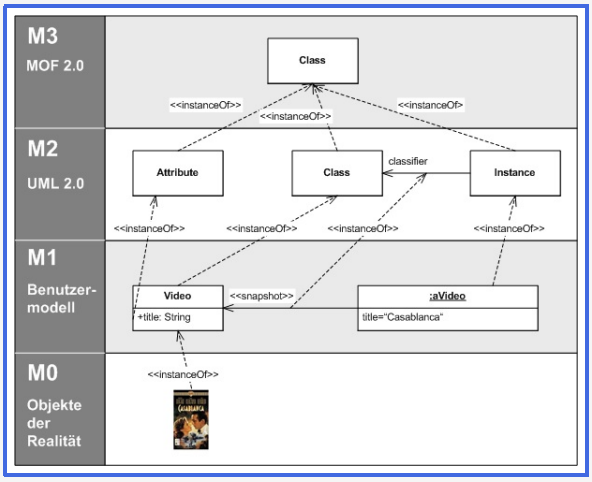

Figure (8) – *OMGs-four-layer-metamodel-architecture - Wikipedia*

Offrir les classes au niveau de base revient à plonger le niveau 2 [3] dans le niveau 1 en y introduisant des méta-objets qui les représentent ... c'est-à-dire en faisant en sorte en premier lieu qu'une classe soit un objet.

Faire en sorte qu'une classe soit un objet :

1. qu'une classe soit instance d'une classe (décrivant des classes), conceptuellement une méta-classe .

```
1 1 class "-> SmallInteger"
2 SmallInteger class "->SmallInteger class"
```

2. qu'il soit possible d'envoyer un message à une classe :

```
1 SmallInteger superclass "->Integer"
2 m := Integer compiledMethodAt: #factorial "-> Integer>>#factorial"
3 m selector "-> #factorial"
```

Listing (13) –

5.2 Solution à métaclasses implicites (Smalltalk, ...)

Voir http://www.lirmm.fr/~dony/notesCours/smalltalkOverview.s.pdf, section 4.3.

En Synthèse :

3. et le 3 ? ...

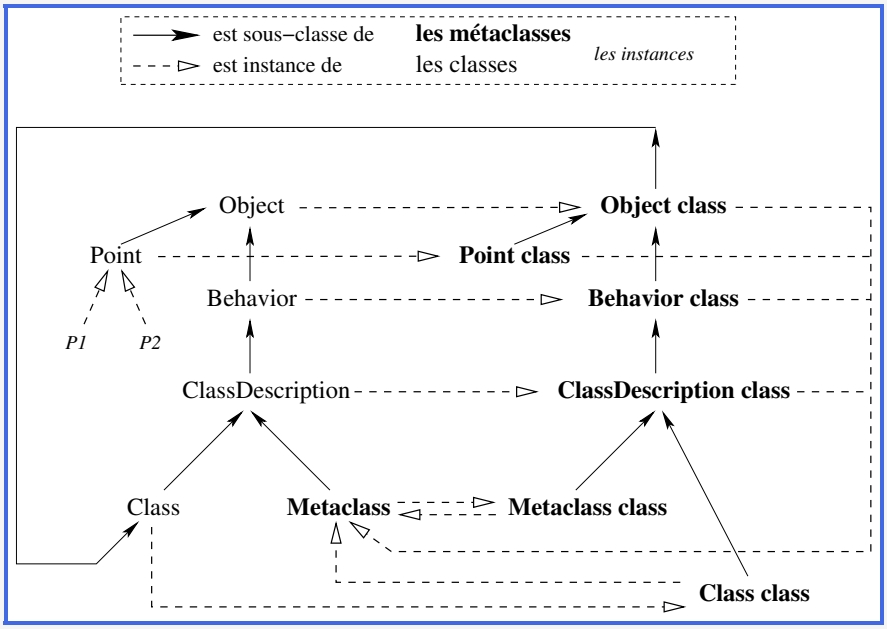

Figure (9) – *Classes et Métaclasses : Hiérarchies d'héritage et d'instantiation. (figure : Gabriel Pavillet)*

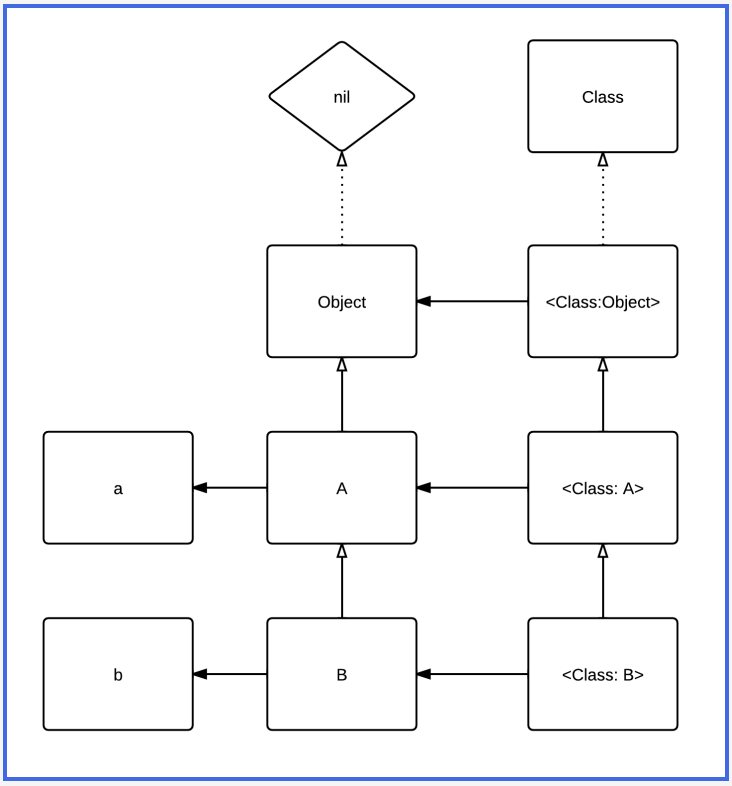

Figure (10) – *RUBY metaclasses inheritance applies Smalltalk's solution : http://timnew.me/blog*

5.3 Solution à méta-classes explicites (Objvlisp - Common-Lisp (CLOS))

Notons :

ako : a kind of - la relation “est-une-sorte-de” ou “est-sous-classe-de”

isa : is a - la relation “est-un” ou “est-instance-de”

La solution *Objvlisp*[4] pour plonger le méta-niveau M3 dans le niveau M2 s’exprime ainsi :

```
1 Object isa Class

2 Class isa Class ako Object
```

5.3.1 Class instance d’elle-même, pourquoi ? comment ?

— Class instance d’elle-même, pourquoi ? :
stopper la régression infinie induite par : “tout objet est instance d’un descripteur, tout descripteur est un objet”.

— Class instance d’elle-même, comment ? ...

Class instance d’elle-meme, une mise en oeuvre opérationnelle

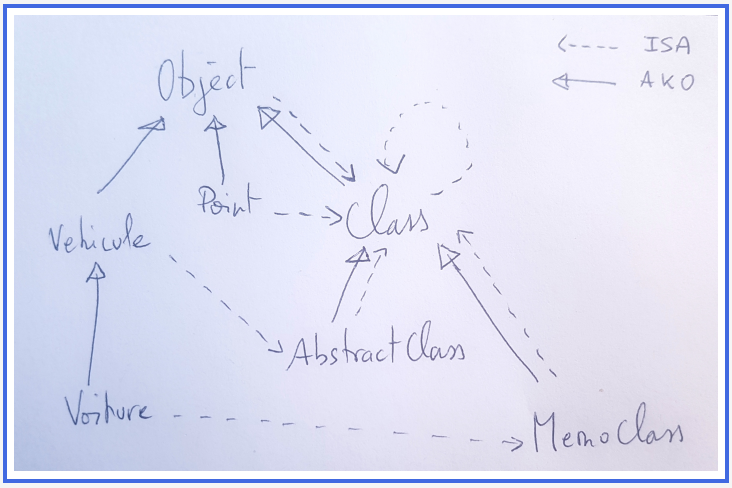

Figure (11) – *Il est possible de faire en sorte qu’une méta-classe* `(Class)` *et une méta-méta-classe (la classe de* `Class`*) soient : (i) structurellement identiques et (ii) fonctionnellement identiques modulo un paramétrage.*

5.3.2 Création d’un Point, utilisation de `newInstance`

— Soit **un point**,

```
1 p := Point newInstance (2, 3) ;;envoi du message newInstance
2 p setx(33)
```

4. Pierre Cointe, “Metaclasses are First Classes : the ObjVlisp Model”. OOPSLA 1987 : 156-167

Listing (14) – *création d'un point en syntaxe objvlisp*

— les valeurs passées en arguments (2 et 3) permettent d'initialiser les attributs `x` et `y` de `p`.

— la structure de p `(x, y)` et les méthodes qu'on peut lui appliquer `(getx(), gety(), setx(..), sety(..))` sont définies par sa classe (`Point`).

— la méthode `newInstance` est définie, quand à elle, sur la classe de la classe `Point`

5.3.3 Création de la classe `Point` et de sa méthode getX()

```
1  Class newClass (
2       #Point, ;; son nom
3       Object, ;; sa superclasse
4       (x, y), ;; la liste des attributs qu'elle déclare
5       (getx(){...}, gety(){...}, setx(..){...}, sety(..){...})) ;; liste des méthodes qu'elle définit
```

Listing (15) – **Point** *(créée comme instance de* `Class`*)*

— les 4 valeurs passées en arguments permettent d'initialiser les attributs de `Point` qui sont son nom, sa superclasse, les attributs qu'elle déclare (x et y), les méthodes qu'elle définit (setX, ...).

— les attributs de `Point` : `nom`, `superclasse`, `attributs`[5], `methods`)) et les méthodes qu'on peut lui appliquer `(getNom, addMethod, newInstance, etc)` sont définies par sa classe, `Class`.

— la méthode `newClass` est, quand à elle, définie sur la classe de la classe `Class` (ou sur une de ses superclasses)

5.3.4 Création de la méta-classe `Class` et de sa méthode `newInstance(...)`

```
1  MetaClass newMetaClass (
2       #Class, ;; son nom
3       Object, ;; sa superclasse
4       (nom, superclasse, attributs, methods), ;; la liste des attributs qu'elle déclare
5       (getNom(), getSuperclasse(), addMethod(...), newInstance(...), ...))
```

Listing (16) – `Class` *créée comme instance de* `Metaclass`*)*

— `Class` est une méta-classe, ses instances sont des classes.

— La méta-classe `Class` définit la méthode `newInstance` permettant à ses instances (par exemple la classe `Point`) de créer des instances (par exemple `p`).

```
1  newInstance (&rest listeArguments)
2      i := self allocateInstance(self instanceSize()).
3      i initInstance ( listeArguments).
4      return (i).

6  allocateInstance(size) {malloc ... }

8  méthode instanceSize()
```

5. pour instance variables ... ou attributs

```
9       return (1 + attributs size() + superclass instanceSize())
```

Listing (17) – *Code, en syntaxe Objvlisp, des méthodes* newInstance, `allocateInstance` *et* `instanceSize`*, définies sur* `Class`

`InitInstance`, utilisée par `newInstance`, est définie sur `Object` ; son argument l est une liste de valeurs d'attributs.

```
1  initInstance (l) ;; l est la liste des valeurs des attributs de l'instance en création
2      n := 1.
3      while (l empty() not())
4          ;; affectation de attribut n avec la valeur n
5          self instVarAtPut (n, l car()).
6          l := l cdr().
7          n := n + 1.
```

Listing (18) – *Code de initInstance(...) définie sur Object*

5.3.5 Création de la méta-méta-classe `Metaclass` et de sa méthode newClass(...)

```
1  MetaMetaClass newMetaMetaClass (
2      #MetaClass, ;;son nom
3      Object, ;;sa superclasse
4      (nom, superclasse, attributs, methods), ;; la liste des attributs qu'elle déclare
5      (getNom(), getSuperclasse(), addMethod(...), newClass(...), ...)) ;;méthodes
```

Listing (19) – **MetaClass** *(créée comme instance de* `MetaMetaclass`*)*

— `MetaClass` définit la méthode `newClass`

— la méthode `newClass` définie sur la méta-méta-classe `MetaClass` permet à ses instances (par exemple la méta-classe `Class`)) de créer des instances (par exemple la classe `Point`).

```
1  newClass(&rest listeArguments)
2      //self vaut Class
3      c := self allocateClass (self classSize()).
4      c initClass (listeArguments).
5      return (c).

7  allocateClass(size) {malloc ...}

9  classSize()
10     return (1 + attributs size() + superclass classSize())
```

Listing (20) – *Code des méthodes définies sur* `Metaclass`

5.3.6

La méthode `initClass` est définie sur `Class`, ou une de ses super-classes. On remarque son identité (modulo les noms) avec la méthode *initInstance* de *Object* (voir listing 18).

```
method initClass(l)
    n := 1.
    while (l empty() not())
        self instVarPut (n, l car()).
        n := n + 1.
        l := l cdr().
```

Listing (21) *– Code de initClass(...) définie sur Object*

5.3.7 Synthèse : Metaclass == Class

— Identité structurelle, `Class` et `Metaclass` déclarent les mêmes attributs.

```
MetaClass newMetaClass (
    #Class,
    Object,
    (nom, superclasse, attributs, methods),
    (getNom(), getSuperclasse(), addMethod(...),
        newInstance(...), ...))
```

```
MetaMetaClass newMetaMetaClass (
    #MetaClass,
    Object,
    (nom, superclasse, attributs, methods),
    (getNom(), getSuperclasse(), addMethod(...),
        newClass(...), ...))
```

— Identité comportementale (même méthodes) modulo ...

```
newInstance (&rest listeArguments)
    i := self allocateInstance(self instanceSize()).
    i initInstance ( listeArguments).
    return (i).
```

```
newClass(&rest listeArguments)
  c := self allocateClass (self classSize()).
  c initClass (listeArguments).
  return (c).
```

— Identité comportementale

Les méthodes `newInstance`, `newClass`, `newMetaClass` sont identiques modulo un paramétrage par spécialisation et composition en présence de liaison dynamique (voir cours de M1) :

- spécialisation - elles allouent une zone mémoire `self allocate()` dont la taille est définie par `self attributs size()` (la taille de la liste des attributs du receveur).
- composition - elles demandent au nouvel objet créé d'initialier ses attributs (`c init(listargs)`.

`newInstance` et `newClass` peuvent être remplacées par une unique méthode `new`.

```
new(&rest listeArguments)
    i := self allocate (self size()).
    i init (listeArguments).
    return (i).

allocate(size) {malloc ...}

size()
    return (1 + attributs size() + superclass size())
```

Listing (22) *– Cette méthode new(..) définie sur* `Class`*, remplace newInstance(..), newClass(..), newMetaClass(..), ..*

— La méthode `init(l)` définie sur `Object` s'applique à tous les objets nouvellement créés (que ce soient des classes ou des instances terminales).

```
method init(l)
```

```
    n := 1.
    while (l empty() not())
        self instVarPut (n, l car()).
        n := n + 1.
        l := l cdr().
```

Listing (23) *–*

En synthèse :

```
Class new (
    #Class,
    Object,
    (nom, superclasse, attributs, methods),
    (getNom(), getSuperclasse(), addMethod(...), new(...), ...))
```

Listing (24) *– Class instance d'elle-même (vue d'artiste). On note l'égalité entre le nombre (4) d'attributs déclarés sur la classe et le nombre d'arguments passés à la méthode new.*

5.4 Bootstrap d'un système réflexif à méta-classes explicites - La poule et l'oeuf

Boucles :

— `Class` instance de `Class`

— `Objet` instance de `Class`, `Class` sous-classe de `Objet`.

Solution : fabrication "à la main", c'est-à-dire dans le code d'iplamtation de la machine virtuelle, et avec son langage d'implantation, d'une première version de la classe `Class`, dotée d'une méthode `new` de base.

```
(setq CLASS ;; affection à la variable CLASS
   '( ;; d'une liste implantant la classe
       CLASS ;; son type
       CLASS ;; son nom
       (OBJECT) ;; la liste de ses sur-classes
       (isit name supers attributs methods) ;; ses attributs
       ( (new ;; ses méthodes
           (lambda (listeArguments) (make-object (name self) ...))
        ...
       )
   )
```

Listing (25) *– 1) Création "à la main" dans le langage d'implantation de la machine virtuelle (ici Lisp) de* `Class` *instance d'elle-même. Extrait de "Metaclasses are first classes, the Objvlisp Model" - Pierre Cointe*

Note : on remarque un attribut supplémentaire `isit`, par rapport à la vue d'artiste présentée au listing 24, il sera plus tard défini sur la classe Object et sert à stocker la classe de chaque objet. Cet attribut n'est explicite dans aucun langage mais l'information correspondante est bien présente en mémoire, elle est généralement obtenue par l'envoi à un objet du message `class` ou *type* ou `instanceOf`.

```
(send CLASS 'new
   :name 'OBJECT
   :supers '()
   :attributs '()
   :methods ... toutes les méthodes de la classe Object
```

Listing (26) *– 2) Création normale, dans le langage utilisateur (ici Objvlisp), de la classe* `Object` *et de ses méthodes*

```
(send CLASS 'new
   :name 'CLASS
   :supers '(OBJECT)
   :attributs '(name supers attributs methods)
   :methods
       '( new (lambda i-v* (make-object name ...))
       ... toutes les autres méthodes de la classe Class
```

Listing (27) *– 3) RE-Création, normale dans le langage à objets résultant (ici Objvlisp et sa syntaxe) de la classe* `Class`*, sous-classe de* `Object`*, et de ses méthodes*

6 Programmation des méta-classes

6.1 Propriétés (attributs et méthodes) des méta-classes.

Remarques générales valides avec un système à méta-classes implicites ou explicites.

6.1.1 Méthodes

Méthode d'instance de méta-classe

Définition : toute méthode définie sur une méta-classe applicable à ses instances.

Une méthode d'instance de méta-classe :

- s'applique à ses instances, qui sont des classes. Exemple : `Point new`, invoque `new` définie sur `Class`,
- le receveur courant (`self` ou `this`) est une classe,
- peut accéder aux attributs de la méta-classe où elle est définie,
- est héritée et s'invoque par envoi de message en liaison dynamique,
- ne s'applique pas aux instances de la classe instance de la méta-classe qui la définit, sauf à passer explicitement au niveau méta :

```
1 name ``--> erreur''
1 class name ``-->SmallInteger''
```

Listing (28) *–* `name` *est une méthode d'instance définie sur la métaclasse* `Class` *...*

- les méthodes de classe de Smalltalk sont en fait des méthodes d'instance de métaclasses.

Versus Méthodes “static” (c++ ou Java)

Méthode “static” : fonction factuellement rattachée à une classe, n’ayant que peu à voir avec une méthode d’instance de méta-classe.

— accède aux attributs “static”,

— ne s’invoque pas par envoi de message donc pas de receveur courant, héritage mais pas de liaison dynamique

```
1  class A{
2      static int m1() { return m2(); }
3      static int m2() {return 1;} }
4
5  class B extends A{
6      static int m2() {return 2;} }
7
8  B.m1(); //rend 1, pas de liaison dynamique
```

Listing (29) – `static` *en* Java *ou* C++

6.1.2 Attributs

A - Attribut d’instance de méta-classe

Attribut défini sur une méta-classe dont la valeur est propre à chaque classe qui en est instance.

Exemple : l’attribut `name` défini sur `Class` et ayant valeur pour chaque classe instance de `Class`.

B - attribut partagé - “static” ou “attribut d’instance à allocation dans la classe” (CLOS) - ou “attribut de classe” (Smalltalk)

Si un attribut a la même valeur pour toutes les instances d’une classe, il est intéressant de le partager.

Un attribut “*static*” de *C++-Java* est partagé à la façon d’un “attribut de classe” de *Smalltalk* ou d’un “attributs d’instance à allocation dans la classe” de *CLOS* (voir plus loin).

Il est traditionnellement accessible dans les méthodes d’instance de la classe et de la méta-classe.

Il n’est pas un attribut d’instance de méta-classe.

```
1  Object subclass: #Citoyen
2     instanceVariableNames: 'nom age adresse'
3     classVariableNames: 'president'
4     package: 'ExempleCours'
```

Listing (30) – *Exemple d’attribut d’instance partagé*

6.2 Utilisation d’un système à méta-classes implicites

Archétype : les méta-classes de *Smalltalk*, voir section 5.2

6.2.1 Spécialisation des comportements par défaut hérités des méta-classes de base

Exemple, faire d'une classe une Mémo-Classe, une classe qui mémorise la liste de ses instances.

```
1  Pile class
2      instanceVariableNames: 'listeInstance'
3
4      new
5          ^self new: tailleDefaut.
6
7      new: taille
8          | newInst |
9          newInst := super new initialize: taille.
10         listeInstance add: newInst.
11         ^newInst
12
13     initialize
14         tailleDefaut := 5.
15         listeInstance := OrderedCollection new.
16
17     getListeInstances
18         ^listeInstances
```

Listing (31) – *Faire de la classe Pile une Mémo-classe*

6.3 Utilisation d'un système à méta-classes explicites

6.3.1 Création de nouvelles méta-classes

Possibilité de créer explicitement une nouvelle méta-classe comme sous-classe de `Class`.

Il n'y a aucun isomorphisme entre la hiérarchie des classes et celle des méta-classes.

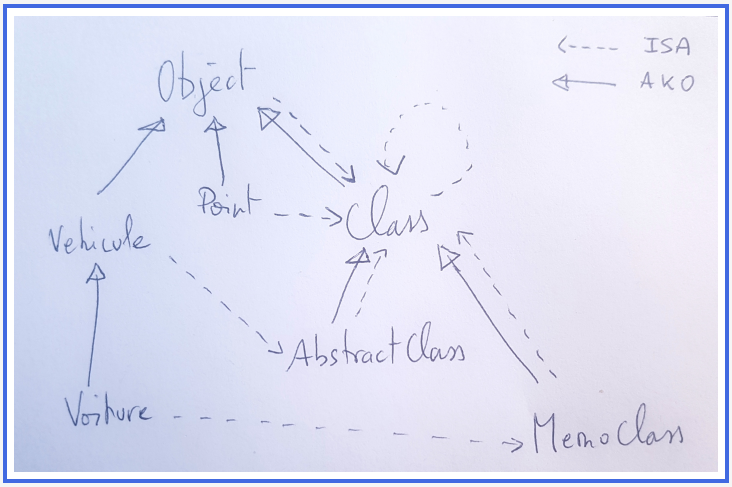

Figure (12) – *AbstractClass et MemoClass sont des méta-classes explicitement créés par le méta-programmeur.*

```
Class new (
    #MemoClass, //son nom
    Class, //sa superclasse
    (listeInstances), //les attributs qu'elle déclare
    (new(...), initialize(), ...) //les méthodes qu'elle définit
```

Listing (32) – *Création d'une méta-classe :* `MemoClass` *en syntaxe* Objvlisp.

```
MemoClass, method new (&rest args)
      i := super new(args)
      listeInstances add(i).
      return(i)
```

Listing (33) – *Les méthodes définies sur* `MemoClass`

```
(defclass memo-class ( standard-class ) ;; hérite de standard-class
  ((ListInstances :initform nil
                  :accessor get-listInstances)) ;; déclare un attribut
                  (:metaclass standard-class)) ;; est instance de standard-class

 (defmethod make-instance ((mc memo-class) &rest args)
  (let ((newI (call-next-method)))
    (setf (get-listInstances mc)
          (cons newI (get-listInstances mc)))
    newI))
```

Listing (34) – `MemoClass` *en* CLOS.

```

class MemoClass(type):
    instances = [] # List of instances
    def __call__(cls, *args, **kwargs):
        instance = super(MemoClass, cls).__call__(*args, **kwargs)
        MemoClass.instances.append(instance)
        return instance

 class Stack(object, metaclass = MemoClass):
    def __init__(self, capacity = 5):
        self.capacity = capacity # Size of the stack, 5 if no value given by the user
        self.content = [None] * capacity # We sqtore the content in a list
        self.index = 0 # Index for the next element
```

Listing (35) – `MemoClass` *en* Python3.

6.3.2 Les n chemins vers la méthode `new`

Suivre le lien d'instanciation 1 fois puis éventuellement `m` fois un lien d'héritage puis éventuellement un envoi de message à `super`, pour finalement arriver à la méthode *new* du système (celle de la classe `Class`).

1.
```
Class new (
    #Point,
    Object,
    (x, y),
```

```
5       (...) )
```

Listing (36) – *création d'une classe standard*

2.

```
1  Class new (
2      #MemoClass,
3      Class,
4      (listesInstances)
5      ( new(...) ) )
```

Listing (37) – *création d'une méta-classe, voir listing 35*

3.

```
1  MemoClass new (
2       #Voiture,
3       Vehicule,
4       (nom, cylindrée),
5       (...) )
```

Listing (38) – *création d'une MemoClass*

4.

```
1  Voiture new (
2       #C5,
3       9)
```

Listing (39) – *création d'une instance d'une memo-classe*

6.4 Méta-niveaux, Héritage et Compatibilité

En présence de méta-classes, la question se pose des relations entre les relations d'instanciation (ISA) et d'héritage (AKO).

Elle se pose en parculier dans un langage réflexif de par le plongement du méta-niveau dans le niveau de base et la possibilité subséquente de passer d'un niveau à l'autre, dans un sens base vers méta dit ascendant ou méta vers base dit descendant.

```
1  //invoquer ma méthode m du niveau meta depuis le niveau de base
2  self class m

4  //invoquer la méthode m du niveau de base depuis le niveau meta
5  self new m
```

Listing (40) – *Passer du niveau de base au niveau méta et inversement.*

Les questions se posent de savoir si la superclasse de la métaclasse est, ou pas, la même que la métaclasse de la superclasse

Le méta-niveau de Smalltalk et celui d'Objvlisp (et de CLOS) proposent des solutions différentes à cette question dite de "compatibilité des métaclasses".

Objvlisp-CLOS : Problème de non compatibilité “ascendante”

Soient :

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

(la classe de A doit définir ou hériter une méthode `bar`)

- une classe B sous-classe de A

- la classe de B, non sous-classe de la classe de A, et ne définissant ni n’héritant `bar`

- une instance b de B

alors : `b foo` lève une exception.

Smalltalk : pas de problème de non compatibilité “ascendante”

Cette situation est impossible avec le modèle à méta-classes implicites de *Smalltalk* (la classe de B ne peut pas ne pas être une sous-classe de la classe de A).

Objvlisp-CLOS : Problème de non compatibilité “descendante”

Soient :

- une Métaclasse MA, méthode bar : ’self new foo’

- la méthode foo doit être définie sur chaque classe instance de MA, par exemple sur A

- une Métaclasse MB, sous classe de MA

- une classe B instance de MB, non sous-classe de A

alors `B bar` lève l’exception “un B ne comprends pas le message foo”.

Smalltalk : pas de problème de non compatibilité “descendante”

Cette situation est impossible avec le modèle à méta-classes implicites de Smalltalk (la classe de B ne peut pas ne pas être une sous-classe de A).

Smalltalk : problèmes d’incompatibilités semantiques

Hiérarchies classes/mémaclasses isomorphes (voir figure 9) avec des méta-classes créés automatiquement

Problème de compatibilité sémantique :

— quand la classe d’une superclasse de C, ne devrait pas être (sémantique) la superclasse de la classe de C, par exemple si la classe de `Véhicule` est `AbstractClass`, celle-ci ne devrait pas être la superclasse de la classe de `Voiture`, car `Voiture` n’est pas abstraite.

— ou quand la superclasse de la classe de C ne devrait pas être (sémantique) la classe de la superclasse de C.

Un modèle à méta-classes explicites permet de traiter ces questions.

7 Le système de métaclasses explicites de Common-Lisp-Object-System

Lire :

- *The Art of the Metaobject Protocol*. Gregor Kiczales, Jim des Rivieres Daniel G. Bobrow.

- Programmation Par Objets : des Concepts Fondamentaux à leur Application dans les Langages. Chapitre 11 et 12. R. Ducournau

- https://lispcookbook.github.io/cl-cookbook/clos.html

7.1 Classes et instances

```
(defclass point () ;;une classe
  (x y z)) ;;3 attributs, dits ''slots''
#<STANDARD-CLASS POINT> ;; c'est une standard-class

(setf my-point (make-instance 'point)) ;;#<POINT 205FA53C>

(type-of my-point) ;;POINT

(setf (slot-value my-point 'x) 33) ;;33

(slot-value my-point 'x) ;;33
```

Listing (41) – *Définition de classe, version 1.*

```
(defclass point (standard-object)
  ((x :initform 1 ;;valeur par défaut de l'attribut
      :initarg :x ;;nom de l'initialiseur
      :accessor getx ;;nom de l'accesseur en lecture (et écriture via setf)
   )
   (y :initform 2 :initarg :y :accessor gety)
   (z :accessor getz :initarg :z :allocation :instance)))

(setf p1 (make-instance 'point :x 19)) ;;#<POINT #x000000020024BB61>
(getx p1) ;;19
(setf (getx p1) 33) ;;33
(getx p1) ;;33
```

Listing (42) – *Définition de classe, version 2*

7.2 Variable d'instance versus variable de classe

```
(defclass person ()
  ((name :initarg :name :accessor name)
   (species
      :initform 'homo-sapiens
      :accessor species
      :allocation :class)))
```

Listing (43) *– Le mot-clé " :allocation" permet de spécifier où sera stockée la valeur d'un attribut. S'il est stocké dans la classe, c'est un attribut partagé par toutes les instances, équivalent d'une variable de classe de* Smalltalk.

7.3 Sous-classes et Héritage

```
(defclass person ()
  ((name :initarg :name :accessor name)
   (species
      :initform 'homo-sapiens
      :accessor species
      :allocation :class)))

(defclass child (person)
  ((can-walk-p
     :accessor can-walk-p
     :initform t)))

(setf p1 (make-instance 'person :name "Pierre"))
(setf c1 (make-instance 'child :name "Lisa"))
(type-of c1) ;; CHILD
(subtypep (type-of c1) 'person) ;; T
```

Listing (44) *– .*

7.4 fonction-générique, multi-méthodes, redéfinitions, liaison dynamique.

```
(defgeneric toString (obj)
  (:documentation "say hello to an object"))
```

Listing (45) *– La fonction générique* `toString` *représente la collection de toutes les méthodes de nom* `toString` *à un paramètre.*

```
(defmethod toString ((p person))
  (format t "Hello ~a !" (name p)))

(defmethod toString ((p child))
  (call-next-method p) ;; équivalent de l'envoi de message à "super"
  (format t "young friend!~&"))
```

Listing (46) *– Deux méthodes à deux paramètres, dites multi-méthodes, appartenant à la fonction générique* `toString`*. Une pour la classe* `person` *et sa redéfinition pour la classe* `child`

```
(setf p1 (make-instance 'person name: 'Pierre))
(tostring p1)
;;Hello Pierre !
```

```
(setf c1 (make-instance 'child name: 'Lisa))
(tostring c1)
;;Hello Lisa ! young friend!
```

Listing (47) – *Liaison dynamique.*

7.5 Appel de méthode avec *multiple-dispatch*, Liaison dynamique généralisée

L'appel de méthode prend en compte les types dynamique de tous les arguments (et pas uniquement celui du receveur).

```
(defclass stockage ()
  ((name :initarg :name :accessor name)))

(defclass dossier (stockage) ;; dossier sous-classe de stockage
  ((contenu :initform nil :reader name)))

(defclass fichier (stockage) ()) ;; fichier sous-classe de stockage

(defclass visitor () ())

(defclass razVisitor (visitor) ())

(defclass findVisitor (visitor) ())

(defclass countVisitor (visitor) ())
```

Listing (48) – *Application au schéma Visiteur*

```
(defgeneric visit (unVisiteur unStockage)) ;; 2 paramètres

(defmethod visit ((v visitor) (d stockage)) 1) ;; 2 paramères, nom et type. Rend 1.

(defmethod visit ((v razVisitor) (d dossier)) (+ (call-next-method) 2))
(defmethod visit ((v razVisitor) (f fichier)) (+ (call-next-method) 3))

(defmethod visit ((v findVisitor) (d dossier)) (+ (call-next-method) 4))
(defmethod visit ((v findVisitor) (f fichier)) (+ (call-next-method) 5))

(defmethod visit ((v countVisitor) (s stockage)) (+ (call-next-method) 6))
```

Listing (49) – *Application au schéma Visiteur - Suite*

```
(setf d1 (make-instance 'dossier))
(setf f1 (make-instance 'fichier))

(setf v1 (make-instance 'razVisitor))

(visit v1 d1) ;;3
(visit v1 f1) ;;4
```

7.6 Multi-méthodes (+ héritage multiple) et linéarisation

L'algorithme d'appel d'une méthode `m` avec liaison dynamique généralisée considère le n-uplet constitué des types dynamiques de tous les arguments passés lors de l'appel et on cherche la méthode la première méthode compatible dans la linéarisation de la fonction générique `m`.

La fonction générique `m` ordonne les méthode de nom `m` de la plus spécifique à la plus générale selon la hiérarchie des types des paramètres, le premier paramètre étant prioritaire sur le suivant pour la détermination de l'ordre.

7.7 Les classes sont aussi des objets

```
(defclass point (standard-object)
  (x y z))

(find-class 'point)
#<STANDARD-CLASS POINT>

(class-name (find-class 'point))
POINT

(setf my-point (make-instance 'point))
(class-of my-point)
#<STANDARD-CLASS POINT 275B78DC>

(typep my-point (class-of my-point))
T

(class-of (class-of my-point))
#<STANDARD-CLASS STANDARD-CLASS 20306534>
```

7.8 Définition de nouvelles métaclasses en CLOS

7.8.1 Définition + spécialisation de la méthode d'instantiation

Une métaclasse est une (1) instance (éventuellement indirecte) et une (2) sous-classe (éventuellement indirecte) de `Standard-class`.

(1) une métaclasse est une classe.

(2) une métaclasse hérite de *new* (nommée *make-instance* en CLOS) ; les instances des métaclasses sont des classes.

```
(defclass singleton-class (standard-class)
  ((UniqueInstance :accessor get-instance :initform nil))
  (:metaclass standard-class))
```

Listing (50) – `Singleton-class` *est la classe des classes qui ne peuvent avoir qu'une seule instance. Elle possède un attribut (d'instance de métaclasse) nommé* `UniqueInstance`.

L'instantiation basée sur `make-instance` de `standard-class` et `initialize-instance` de `Standard-object`, peut être spécialisée en redéfinissant la méthode `make-instance` pour la nouvelle méta-classe.

```
(defmethod make-instance ((aSClass singleton-class) &rest args)
  (or (get-instance aSClass)
```

```
3       (let ((newInstance (call-next-method))) ;; ''super new''
4             (setf (get-instance aSClass) newInstance) ;; modifie UniqueInstance
5             newInstance)))
```

Listing (51) *– Spécialisation de l'instantiation sur* singleton-class, (call-next-method') *réalise l'appel de la méthode masquée par la redéfinition.*

Création d'une `SingletonClass`

```
1  (defclass test (standard-object)
2    (()) ;;pas d'attributs
3    (:metaclass singleton-class))

5  >(eq (make-instance 'test) (make-instance 'test))
6  =T
```

7.8.2 Compatibilité méta-classe/super-classe

CLOS implante une version du système à méta-classes explicites d'*Obvlisp*.

Le programmeur peut choisir la classe de la classe qu'il crée et peut donc faire une erreur de compatibilité.

`validate-superclass` (*callback* du protocole d'instantiation), message envoyé pour comparer successivement toute nouvelle instance d'une nouvelle méta-classe avec chacune de ses superclasses.

Fonction générique : (defmethod validate-superclass ((cl standard-class) (super standard-class))

`cl` est une classe et `super` une de ses super-classes directes ; rend vrai si les classes de `cl` et de `super` sont compatibles et nil sinon.

```
2  (defclass memo-object ( standard-object )
3    ()
4    (:metaclass standard-class))

6  (defclass memo-class ( standard-class ) ;; hérite de standard-class
7    ((ListInstances :initform nil
8                    :accessor get-listInstances)) ;; déclare un attribut
9                    (:metaclass standard-class)) ;; est instance de standard-class

11  (defmethod make-instance ((mc memo-class) &rest args)
12   (let ((newI (call-next-method)))
13     (setf (get-listInstances mc)
14           (cons newI (get-listInstances mc)))
15     newI))
```

Listing (52) *– Une nouvelle méta-classe* `memo-class`*, on pose comme contrainte que ses instances doivent hériter de la classe standard* `memo-object`*, au lien de* `standard-object`

```
1  ;; une nouvelle memo-classe hérite d'une classe standard
2  (defmethod validate-superclass ((cl memo-class) (sup standard-class))
3       ;; ok si cl hérite de memo-object.
4       (eq 'memo-object (class-name sup)))

6  ;; une nouvelle mémo-classe hérite d'une mémo-classe
7  (defmethod validate-superclass ((cl memo-class) (sup memo-class))
8      ;; ok
```

```
    t)

;; une nouvelle classe standard hérite d'une mémo-classe
(defmethod validate-superclass ((cl standard-class) (sup memo-class))
    ;; interdit
    ())
```

Listing (53) *– Méthodes indiquant à quelle condition un nouvelle classe est compatible avec une autre en présence de mémo-classes*

8 Python

https://docs.python.org/3/reference/datamodel.html#objects-values-and-types

https://docs.python.org/3/reference/datamodel.html#customizing-class-creation

A suivre

Table des matières