Université Montpellier-II
UFR des Sciences - Département Informatique

Objets Avancés

Vue générale de Smalltalk : Tout Objet, Environnement de Génie Logiciel, Réflexivité, Métaprogrammation, IDM,

Notes de cours - 2007-2021
Christophe Dony

1 Introduction

Smalltalk est à la fois (i) un langage premier sur différents aspects du génie logiciel dont l'adaptabilité dynamique des programmes et la pratique "agile", (ii) un modèle d'évolution pour les autres langages de par la simplicité de ses concepts et, (iii) avec Simula, un des deux langages historiques à l'origine de l'approche Objet. Ce sont trois raisons pour l'étudier.

1.1 Idées clé

1. Simplicité, Uniformité (4 grands fondements dont "tout est objet", voir section 1.3)
2. Langage de programmation ET Environnement interactif de génie logiciel dédié au développement "agile" ()

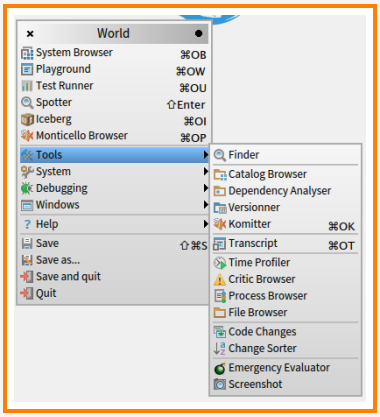

Figure *(1)* – *Ce cours utilise le langage et environnement Pharo, une émanation actuelle de Smalltalk - https ://pharo.org/*

3. Ingénierie Dirigée par les Modèles, modèles à l'exécution, adaptabilité dynamique des programmes ... (" *L'intelligence c'est la faculté de s'adapter au changement.*" *Stephen Hawking*).

1.2 Quelques liens historiques et pratiques

— Alan Kay (https://fr.wikipedia.org/wiki/Alan_Kay), créateur du langage, médaille Turing

— Histoire, l'idée de base : https://fr.wikipedia.org/wiki/Dynabook

— Histoire, les livres : http://www.world.st/learn/books

— La machine virtuelle pour le tout objet réflexif : CoG de Eliot Miranda (http://www.mirandabanda.org/cogblog/about-cog/)

— Le centre d'information Européen : http://www.esug.org/

— Le langage utilisé utilisé dans ces notes : http://pharo.org/ (Pharo is a clean, innovative, open-source Smalltalk-inspired environment)

— Un cours interactif en ligne : http://mooc.pharo.org,

— 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

— les versions industrielles : http://www.cincomsmalltalk.com/main/ : Cincom, Smalltalk industriel (ObjectStudio - Visualworks)

— POURQUOI c'est intéressant :

  — Simplicité : http://stackoverflow.com/questions/1821266/what-is-so-special-about-smalltalk
  — Adaptabilité Dynamique, Ingénierie dirigée par les modèles : http://moosetechnology.org/
  — Agilité : https://en.wikipedia.org/wiki/Extreme_programming. site de Kent Beck, fondateur de "Agile development" : http://www.threeriversinstitute.org :
  — Innovation : https://blog.appacademy.io/pharo-future-software-development/

1.3 Le style de langage

— héritier de *Simula* (Objet, abstraction de données, polymorphisme d'inclusion)
— héritier de *Lisp-Scheme*
  — style applicatif (toute instruction est une expression),
  — ordre supérieur
  — typage dynamique (pas de problèmes de redéfinitions covariantes, pas de surcharge statique),
  — passage par référence (à la *Lisp)* généralisé,
  — allocation et récupéraration dynamiques et automatiques de la mémoire,
  — compilation en instructions d'une machine virtuelle,
  — réflexivité : Programmes et Environnement = données,

Le langage en quatre idées

1. Toute entité est un objet.
   Un objet est une entité individuelle, repérée par une adresse unique, possédant un ensemble de champs (autant que d'attributs sur la classe), connus par leurs noms et contenant une valeur qui peut changer (mutable).
2. Tout objet est instance d'une classe,
   qui définit sa structure (définie par un ensemble attributs privés) et ses comportements (définis par un ensemble de méthodes).
3. Tout comportement d'un objet (méthode) est activé par un envoi de message (avec liaison dynamique).
4. Toute classe (sauf `ProtoObject`) est définie comme une spécialisation d'une autre. La relation de spécialisation définit un arbre d'héritage.

1.4 Interprétation, Compilation, machine virtuelle

Principe d'exécution : compilation des instructions 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. *Java* a repris ce schéma d'exécution. Voir par exemple COG (http://www.mirandabanda.org/cogblog/about-cog/).

Exemple de méthode et de bytecode généré

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

Listing (1) *– définition de la méthode* `fact` *sur la classe* `PositiveInteger`

```
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
```

1.5 Introduction pratique

Classes et instances

```
Object subclass: #Compteur
    instanceVariableNames: 'valeur'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'ExosPharo-PetitsExercices'

initialize
    valeur := 0.
get
    ^valeur
incr
    valeur := valeur + 1.
decr
    valeur := valeur − 1.
```

Listing (2) *– classe* `Compteur`*, définition de classe et de méthodes d'instance*

```
new
    "redéfinition de la méthode new pour appeler automatiquement une méthode d'initialization"
    ^super new initialize

example
    | c |
    c := Compteur new.
    c incr.
    ^c
```

Listing (3) *– définition de méthodes de classe*

1.6 La classe Compteur dans l'environnement

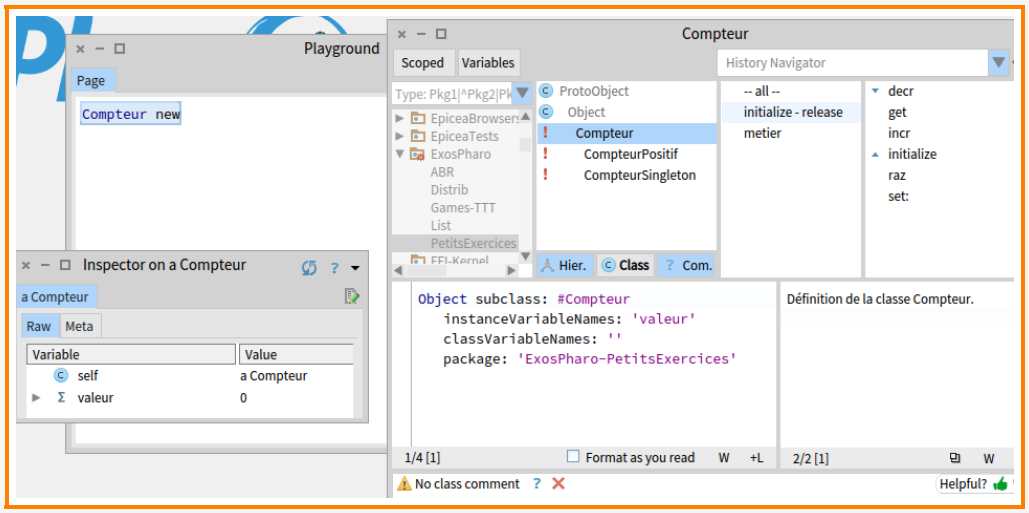

Figure *(2)* – *Définition de la classe* `Compteur` - Pharo *Env.*

1.7 L'exception universelle : `doesNotUnderstand`

```
3.4 factorial
```

→ *Instance of SmallFloat64 did not undestrand factorial*

Conduite à tenir :

— lire le message

— si non compréhension, ouvrir le debbugger (one click), corriger, relancer

1.8 Tout objet et réflexivité

```
1 class "-> SmallInteger"
1 class class "->SmallInteger class"
1 class class class "->Metaclass"
1 class class class class "->Metaclass class"
1 class class class class class "->Metaclass"
```

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

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

2 Syntaxe

```
exampleWithNumber: x
    "A method that illustrates every part of Smalltalk method syntax except
    primitives. It has unary, binary, and keyboard messages, declares arguments
    and temporaries, accesses a global variable (but not an instance variable),
    uses literals (array, character, symbol, string, integer, float), uses the pseudo
    variables true, false, nil, self, and super, and has sequence, assignment,
    return and cascade. It has both zero argument and one argument blocks."
    | y |
    true & false not & (nil isNil) ifFalse: [self halt].
    y := self size + super size.
    #(\$a #a "a" 1 1.0)
        do: [ :each |
            Transcript show: (each class name);
                       show: ' '].
    ^x < y
```

Listing (4) – *from :* http://wiki.c2.com/?SmalltalkSyntaxInaPostcard

— 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 de la syntaxe de Smalltalk.

2.1 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

— booléens : true, false

— tableaux de constantes littérales (construit à l'analyse syntaxique) : #(3 3.45 -3 aM 'abc' #bill true #(am stram gram))

— commentaires : "ce commentaire sera ignoré par l'analyseur syntaxique"

2.2 Identificateur

— Suite de chiffres et lettres commençant par une lettre.

— Manipulables dans le texte des programmes via leur symbole éponyme.

```
Integer methodDictionary at: #factorial
```

— Non typés dans le texte des programmes (typage dynamique).

2.3 Affectation

```
i := 3
```

Listing (5) – *à la* Algol, *ou à la* Pascal

2.4 Séquence d'Instructions

Il y a une séquence implicite associée à tout bloc (corps de méthode par exemple). Dans un bloc, Les instructions sont séparées par un "." (et pas un ";").

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

2.5 Tableaux littéraux

```
a := 3.
{ a . a+1 . a*a } "--> #(3 4 9)
```

2.6 Retour à l'appelant

```
^33 "return 33"
```

Toute invocation de méthode rend une valeur, celle de `self` par défaut, le type `void` n'existe pas.

2.7 Envoi de message

Définitions :

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

Trois sortes de messages (distinction syntaxique) :

1. *unaires* (à un seul argument : le receveur)
   `1 class`
   `5 fact`
2. *binaires* (à deux arguments, syntaxe pratique pour les opérations arithmétiques en infixé)
   `1 + 2`
3. *"keywords"* (à $n (n >= 2)$ arguments dont le receveur)
   Leur forme syntaxique (originale) permet de représenter un envoi de message comme une "petite conversation" (*small talk* en anglais) entre un objet et un autre.
   `1 log: 10`
   `anArray at: 2 put: 3`
   `monCalendrier enregistre: unEvenement le: unJour à: uneheure`
   Les familiers de *Java* traduiront en :
   `monCalendrier.enregistre(unEvenement, unJour, uneHeure)`
   Noms des méthodes utilisées dans ces exemples :
   `log:`,
   `at:put:`,
   `enregistre:le:à:`.

- 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.`

2.8 Structures de contrôle

Il n'existe aucune forme syntaxique particulière pour les structures de contrôle ; le contrôle se réalise par envois de messages.

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 utiliser des *blocks* (fermetures lexicales), soit en position de receveur, soit en position d'argument.

Une fermeture lexicale est une fonction anonyme (lambda-expression) capturant son environnement lexical de définition (toute variable libre y est interprétée relativement à l'environnement dans lequel la fermeture a été définie).

La syntaxe de définition d'une fermeture est `[ <parametre>* | <instruction>* ]`

2.8.1 Exemples

Conditionnelle

```
(aNumber \\ 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].
```

Itérateur généralisé

```
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 Classes et Méthodes

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

Variables d'instance, variables de classes (équivalent des "statiques" de C++ et Java), variables de *pool*.

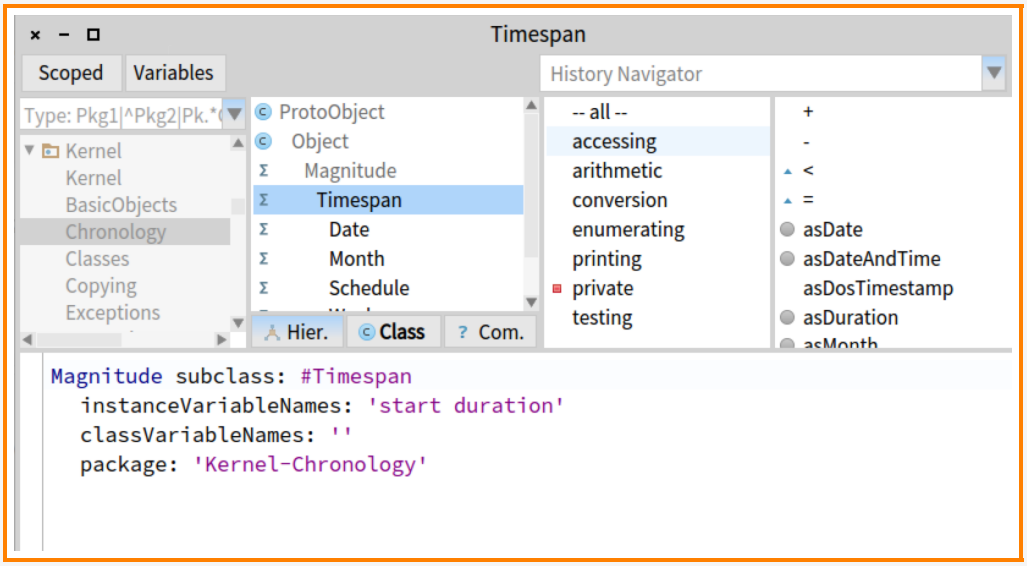

Figure *(3)* – *Définition de la classe* `TimeSpan` - Pharo *env.*

Méthodes d'instance

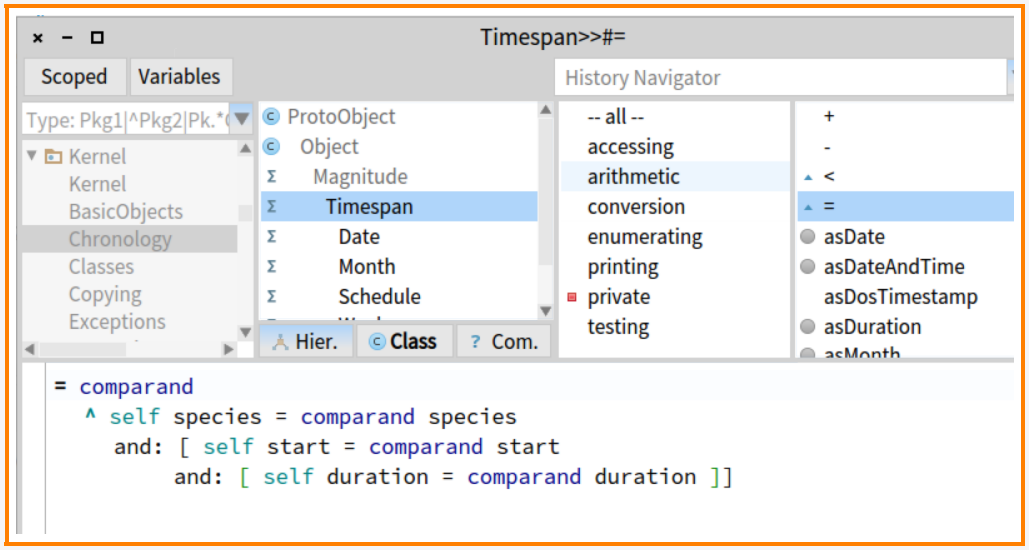

Figure *(4)* – *Méthode d'instance "=" de la classe* `TimeSpan`

Méthodes de classe

Méthodes invocables par envoi de message aux classes [1]

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

1. Les méthodes de classe sont en fait des méthodes d'instance définies sur les métaclasses, voir la section 4.3. En première approche, elles ressemblent aux *static* en `C++` puis `Java`, mais elles sont en fait des méthodes standard définies sur les classes des classes (tout est objet), utilisables de façon standard.

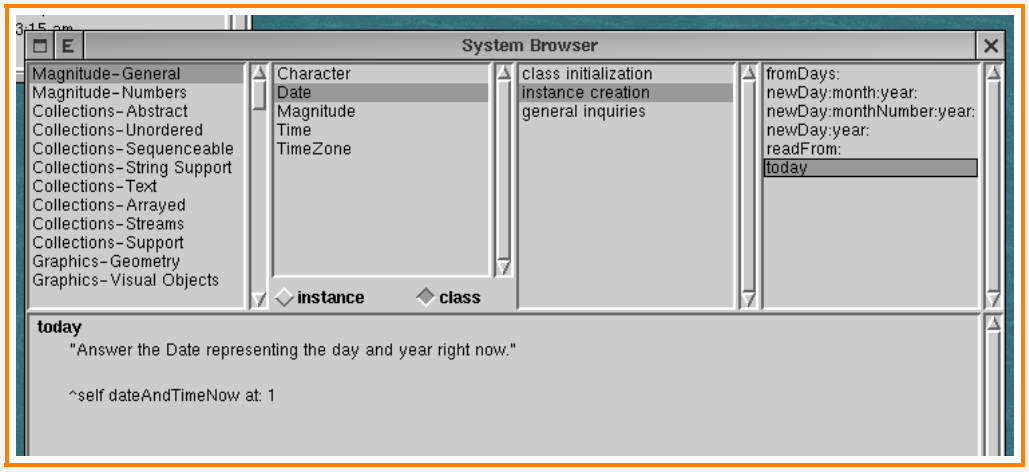

Figure ***(5)*** – *Méthode de classe "today" de la classe Date (En fait une méthode d'instance de la (méta)-classe "Date class")*

Différentes sortes de variables accessibles au sein des méthodes

Les identificateurs accessibles au sein d'une méthode M :

- les paramètres de M,
- les variables temporaires de M,
- les attributs (variables d'instance) déclarés sur la classe du receveur courant,
- les identificateurs prédéfinis `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 (voir `PoolDictionary`),
- les variables globales du système, dont celles référençant toutes les classes.

3.2 Envoi de message

Demande à un objet d'exécuter un de ses services.

Procédé, réalisé à l'exécution, menant a l'exécution d'un méthode de même nom que le *selecteur*.

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

Diverses techniques d'optimisation de l'envoi de messages sont implantées dans les diverses machines virtuelles dont la première est la technique de cache (voir le *green book*).

3.3 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 la référence sur le nouvel objet.
- Exemple : `Date new`, `d := Date new.`

Il n'y a pas de constructeurs spécifiques ;

l'initialisation s'effectue via des méthodes standard, souvent invoquées automatiquement dans des redéfinitions de la méthode de classe `new`.

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

3.4 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
```

3.5 Classe abstraite

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

Ceci pose néanmoins de sérieux problèmes pour ses futures sous-classes concrètes. (utilisation obligatoire de `basicNew`). Ceci est une limite de la solution Smalltalk pour les métaclasses (voir section 4.3).

3.6 Methode abstraite

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

```
method
    self subclassResponsibility
```

3.7 Polymorphisme d'inclusion, Héritage entre classes

— Créer une sous-classe SC d'une classe C, revient à définir un sous-type, au sens du polymorphisme d'inclusion du type définit par C. Ainsi, SC hérite de C et toute instance de SC possède les caractéristiques définies par C.

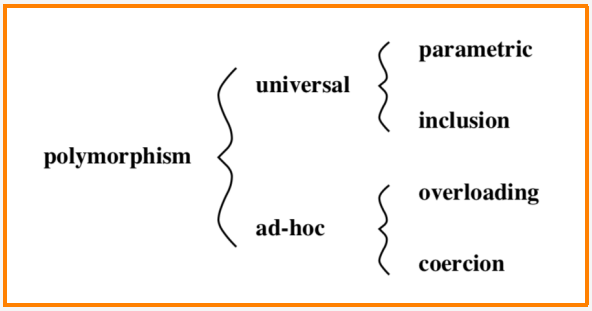

Figure *(6)* – *Variétés de polymorphismes (extrait de Luca Cardelli, Peter Wegner : "On Understanding Types, Data Abstraction, and Polymorphism". ACM Comput. Surv. 17(4) : 471-522 (1985)*

— Héritage simple : toute méthode définie sur une surclasse est applicable (via envoi de message) aux instances des sous-classes.

— **Règle de redéfinition/spécialisation** : toute méthode de nom m d'une sous-classe SC d'une classe C définissant m en est une redéfinition.

— de plus :
pas de règles de covariante/contravariance, pas de *downcast*,
pas de surcharge,
polymorphisme paramétrique universel mais sans contrôle du compilateur,

— avantages et défauts à corréler à la pratique de tests systématiques et élaborés ? (Plus globalement voir *Agile Development*, ou la classe `TestCase` en *Pharo*).

3.8 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')
            IdentitySkipList #()
        WeakRegistry #('valueDictionary' 'accessLock')
```

3.9 Héritage et description différentielle

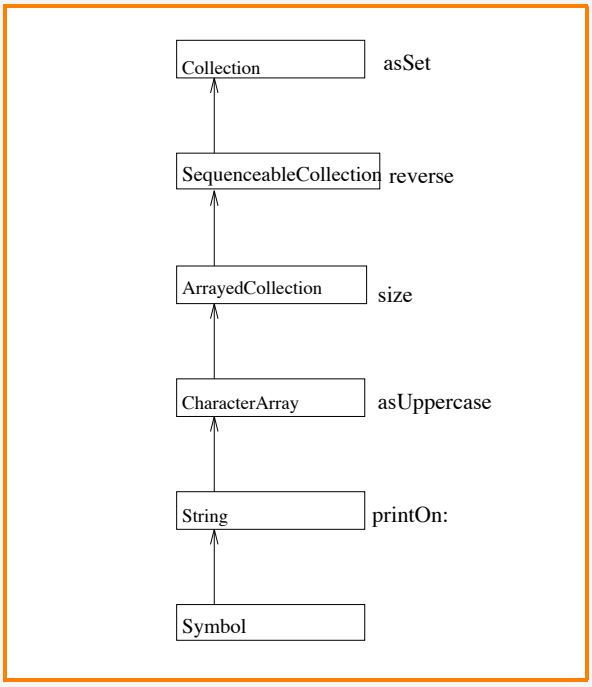

Figure ***(7)*** – *Hiérarchie d'héritage avec ajouts de fonctionnalités. Pour plus de détails sur la description différentielle, voir notes de cours "Réutilisation/Frameworks".*

3.10 Schémas de spécialisation utilisant l'héritage

— Masquage : spécialisation d'une méthode sur une sous-classe.

— Masquage partiel : utilisation de la pseudo-variable `super`.

— Paramétrage par spécialisation (pseudo-variable `self`), ou par composition, classiques (voir cours "Réutilisation/Frameworks").

```
indexOf: anElement startingAt: start ifAbsent: exceptionBlock
    "Answer the index of the first occurence of anElement after start
    within the receiver. If the receiver does not contain anElement,
    answer the  result of evaluating the argument, exceptionBlock."

    start to: (self size)
          do: [:index | (self at: index) = anElement ifTrue: [^ index]].
    ^ exceptionBlock value
```

Listing (6) – *Exemple sur la classe SequenceableCollection*

3.11 Héritage statique classes-traits et traits-traits

Les *traits*[2] sont des unités de réutilisation prédéfinies partageables et composables, intégrées par différents langages (Pharo, Php, Scals, ...).

Un *trait*, à distinguer d'une interface, définit des méthodes abstraites ou concrètes.

```
Trait named: #TComparable
    uses: {}
    package: 'Kernel-Traits'!

< aComparable
    "Answer whether the receiver is less than the argument."
    ^self subclassResponsibility

min: aComparable
    "Answer the receiver or the argument, whichever is lesser."
    self < aComparable
        ifTrue: [^self]
        ifFalse: [^aComparable]

... etc
```

Listing (7) – *Exemple de Trait*

```
Object subclass: #CompteurWTrait
   uses: TComparable
   instanceVariableNames: 'valeur'
   classVariableNames: ''

get
    ^valeur

< aComparable
    ^valeur < aComparable get
```

Listing (8) – *Exemple d'utilisation d'un Trait*

Toute classe utilisant (`use` relation) un trait T possède (héritage au moment de la définition) les méthodes définies par T :

- les méthodes de la classe utilisatrice (par exemple la méthode `=` de l'exemple) sont prioritaires par rapport à celles d'un trait utilisé (on peut donc redéfinir une méthode d'un trait),
- les méthodes d'un trait utilisé sont prioritaires par rapport à celles des superclasses de la classe utilisatrice.
- Une méthode définie sur un trait T n'a pas accès aux variables d'instances d'une classe qui utilise T.

Une classe peut utiliser plusieurs traits ; sous réserve de gérer (statiquement) les conflits de noms.

Un trait peut utiliser un autre trait.

Un trait peut être utilisé par un(e) ou plusieurs classes ou traits.

Un trait définir des attributs (résultat récent - *stateful traits*[3])

```
Trait named: #TSortable
    uses: {}
```

2. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, Andrew P. Black : Traits : Composable Units of Behaviour. ECOOP 2003 : 248-274.
Les traits n'existent pas en Smalltalk originel, la mise en oeuvre présentée ici est celle de *Pharo*.

3. Pablo Tesone, Stéphane Ducasse, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, **A new modular implementation for stateful traits**, Science of Computer Programming, Volume 195, 2020, 102470, ISSN 0167-6423.

```
    category: 'Collections-Abstract'

!TSortable methodsFor: 'sorting' !
sort
    "Sort this collection into ascending order using the '<=' operator."
    self sort: [:a :b | a <= b]
"-------------------------"
Collection subclass: #SequenceableCollection
    uses: TSortable
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'Collections-Abstract'
```

Listing (9) *– Traits, exemple no 2. En utilisant le trait* `TSortable`*, la classe* `SequenceableCollection` *possède la méthode* `sort`*, toutes ses instances peuvent l'utiliser.*

```
c := OrderedCollection new.
c add: 1; add: 19; add: 8; add: 11.
c sort -> an OrderedCollection(1 8 11 19)
c isKindOf: TSortable -> false
```

Listing (10) *– Utilisation de la méthode* `sort` *du trait* `TSortable` *pour une instance de* `OrderedCollection`*, sous-classe de* `SequenceableCollection`*.*

4 Le tout-objet et la méta-programmation

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

`méta-entité` : entité constitutive du méta-niveau donc du langage ou de son environnement

Entité de première classe : entité, possiblement une méta-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, le source des méthodes, certaines structures de contrôle, éventuellement la pile d'exécution, l'environnement de programmation, le compilateur, ... sont des entités de première classe.

méta-programmation : programmation des entités constitutives du système (méta-entités) en utilisant le système.

4.2 Méta-programmation basique

4.2.1 Les classes comme "rvalues" standards

La "généricité paramétrique" est la capacité à contraindre les structures de données composites, comme les collections par exemple, à ne contenir que des éléments appartenant à des types identifiées dans le texte du programme.

En typage statique (voir les génériques *Java*, les templates *C++*) la vérification de telles contraintes peut être faite à la compilation.

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

Exemple :

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

```
push: element
    (element isKindOf: typeElements)
        ifTrue: [ super push: element ]
        ifFalse: [ self error: 'Impossible d''empiler ' , element printString ,
                                ' dans une pile de ' , typeElements printString]
```

Listing (11) *– test du type d'un objet stocké dans une variable. A ne pas confondre avec instanceOf de Java.style*

4.2.2 Manipulation standard des objets primitifs (rock-bottom objects)

Chaque élément 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''
false ifFtrue: [#useless]
nil isNil "null pointer or UndefinedObject''
selecteur := #facto , #rial ``concaténation de 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 représentant les nombres.

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

4.2.3 L'objet `nil`.

`nil` (`null` en *Java)* est la valeur par défaut du type référence, donc affectée à toute *l-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 définir des 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`.

```
type List =
    Empty |
    Tuple of Object * List
```

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

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

Mise en oeuvre (extrait 1)

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

!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)

```
!UndefinedObject methodsFor: 'ListManipulation'!
```

```
addFirst: element
    ^List with: element

length
    ^0

append: aList
    ^aList
```

4.2.4 Fermetures Lexixales - Addendum

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"), et utilisant des fermetures lexicales.

Une fermeture lexicale est une fonction anonyme créée dans l'environnement lexical de sa définition et exécutée dans cet environnement quel que soit son point d'activation.

Les fermetures en Smalltalk sont implantées en *Smalltalk* par la classe *BlockClosure*.

— Une fermeture est une constante littérale dont la valeur est elle-même.

```
[:x | x + 1]
= [:x | x + 1]
```

— Une fermeture peut ainsi être utilisée comme *rvalue* (affectation, passage en argument) :

```
V := [:x | x + 1]
(date = '24/12') ifTrue: [Transcript show: 'Noël'.]
```

— Une fermeture est une fonction, que l'on peut donc invoquer. Comme c'est par ailleurs aussi un objet (tout est objet) on l'invoque en lui envoyant un message, ici le message `value`.

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

— Si la fermeture possède des paramètres, on l'invoque en lui passant des arguments avec les messages `value:` , `value:value:`, etc (c'est moins beau qu'en *Scheme* mais c'est conforme au paradigme).

```
V value: 23
=24
[:i | i + 1] value: 33
= 34
```

4.2.5 Fermetures et retour de méthode

Une fermeture capture le point de retour du *block* ou elle est créée.

```
1  "une méthode de classe définie sur la classe Object
2  qui créée une fermeture et la passe en argument à la méthode test2"
3  test2Main
4      self test2: [^1 + 2].
5      Transcript show: '33333333'; cr.

7  "la méthode de classe test2 définie également sur Object
8  invoque la fermeture"
9  test2: aBlock
```

```
    Transcript show: '11111111'; cr.
    aBlock value.
    Transcript show: '22222222'; cr.

"exécution de Object test2Main"
Object test2Main
11111111
=3
```

A condition que le point de retour existe toujours lors de l'exécution du *block*.

```
test1Main
    | b |
    b := self test1.
    Transcript show: '11111111'; cr.
    Transcript show: b value.
    Transcript show: '22222222'; cr.

test1
     ^[^1 + 2]
```

— The receiver tried to return result to homeContext that no longer exists.

4.2.6 Fermeture et Structures de contrôles

De par leur auto-évaluation, les fermetures lexicales permettent la définition de structures de contrôle.

```
"une méthode abstraite sur la classe Boolean"

ifTrue: alternativeBlock
    "If the receiver is false (i.e., the condition is false), then the value is the
    false alternative, which is nil. Otherwise answer the result of evaluating
    the argument, alternativeBlock. Create an error notification if the
    receiver is nonBoolean. Execution does not actually reach here because
    the expression is compiled in-line."

    self subclassResponsibility
```

4.3 Les Méta-classes en Smalltalk

4.3.1 Rappel : la solution Objvlisp

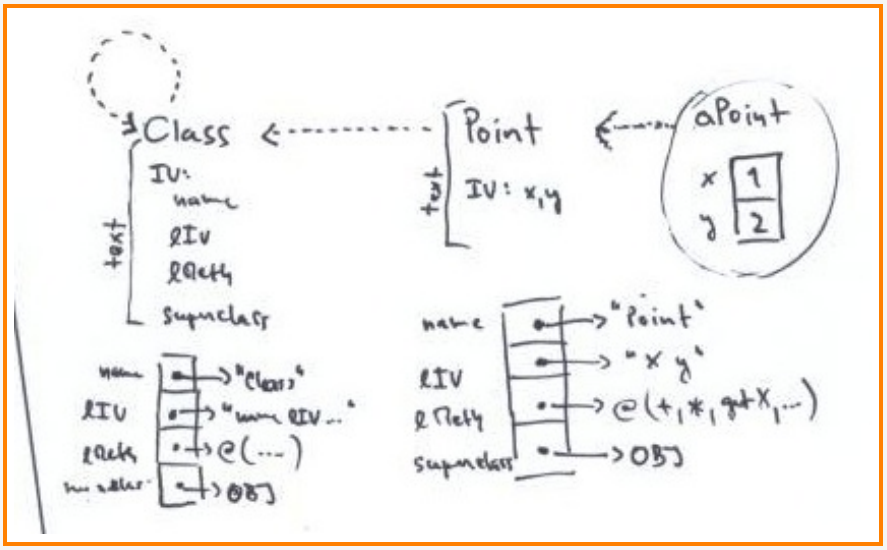

Figure ***(8)*** – *Une classe et une méta-classe sont structurellement et fonctionnellement identiques*

— `Object` est la racine de l'arbre d'héritage,

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

On parle à son propos d'un système à méta-classes explicites car le créateur d'une classe peut choisir la méta-classe qu'il souhaite instancier.

L'intérêt de cette solution est de permettre le découplage entre la superclasse de la méta-classe et la méta-classe de la super-classe. Une classe peut ainsi et par exemple 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 4.3.8). Le langage *CLOS* qui réalise un sur-ensemble de la solution *Objvlisp* laisse au développeur le soin de résoudre ces problèmes éventuels de compatibilité via la définition de fonctions ad.hoc.

4.3.2 Motivations de la solution Smalltalk

La solution Smalltalk a été conçue avec deux objectifs [4] :

— Donner la possibilité de définir des méthodes sur les métaclasses tout en rendant les métaclasses invisibles au développeur qui n'a pas envie de les voir. L'invisibilité des méta-classes pour le développeur standard est réalisé par l'éditeur de programmes également nommé *browser*.

— Rendre impossible la création de métaclasses incompatibles.

4. 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 nécessaire 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.

4.3.3 Architecture du système de métaclasses

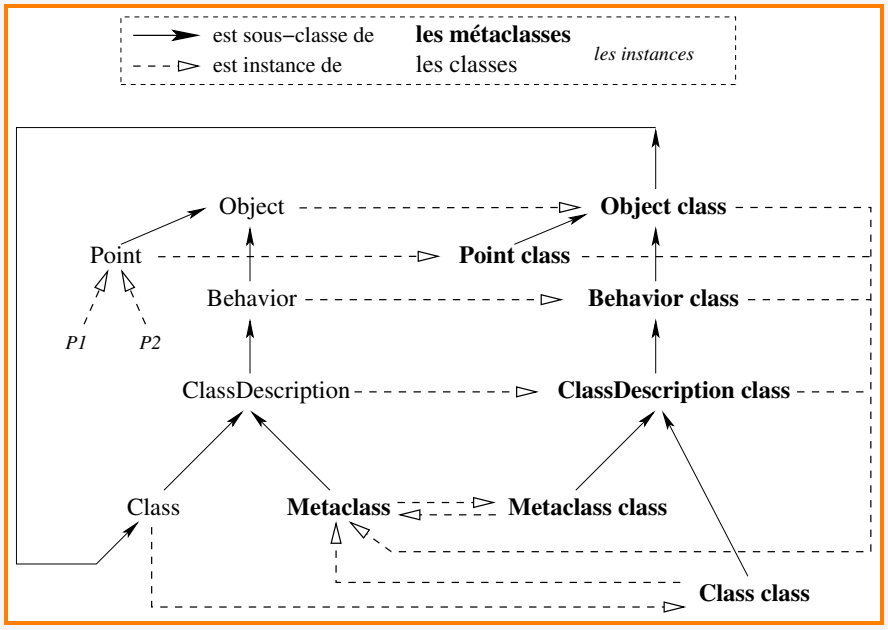

Figure ***(9)*** – *Classes et Métaclasses : Hiérarchies d'héritage et d'instantiation. (figure : Gabriel Pavillet)*

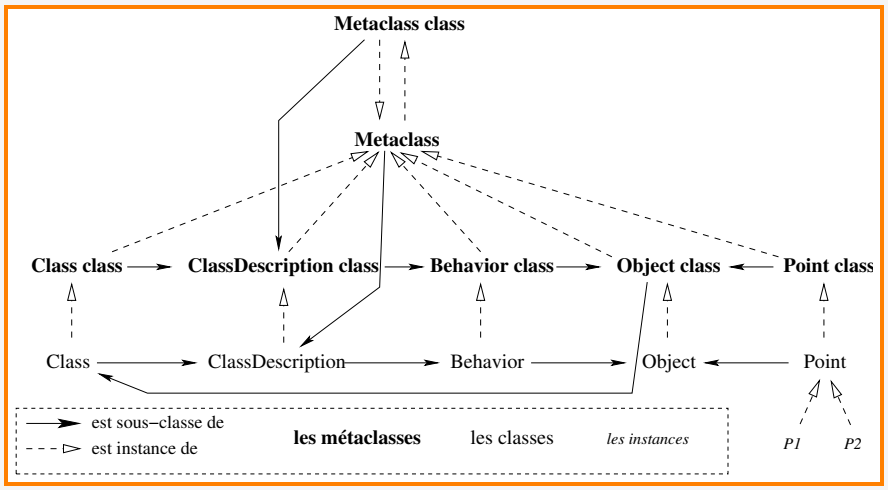

Figure ***(10)*** – *Classes et Métaclasses : couches d'instantiation. (figure : Gabriel Pavillet)*

Les points clé :

- 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 comprend, selon la sémantique usuelle [5] du lien d'héritage, ainsi : une `Object class`, par exemple `Object`), est une sorte de classe.

5. AKO : a kind of, est une sorte de ... une voiture est une sorte de véhicule.

— Il y a 4 niveaux (conceptuels) d'instantiation. Un objet (couche 1) (par exemple un point), instances d'une classe (dans cet exemple `Point`), instance d'une métaclasse (dans cet exemple `Point class`), instance de l'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.

4.3.4 Structure des classes et des métaclasses (cf. fig. 11)

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

Figure *(11)* – *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 sorte de `Class`, 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`.

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

`thisClass` (nom peu évocateur) référence la classe, unique instance (schéma *Singleton)*) d'une métaclasse dans un contexte donné.

`thisClass` est utilisable dans les méthodes d'instance de la classe `Metaclass` qui n'a aucune sous-classe[6].

6. Exercice : créer une sous-classe MC de la classe `Metaclass`, qui définirait un nouveau type généraliste de meta-classes. La difficulté principale 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 classe qui en relève ne soit pas une instance de `Metaclass` mais de `MC`.

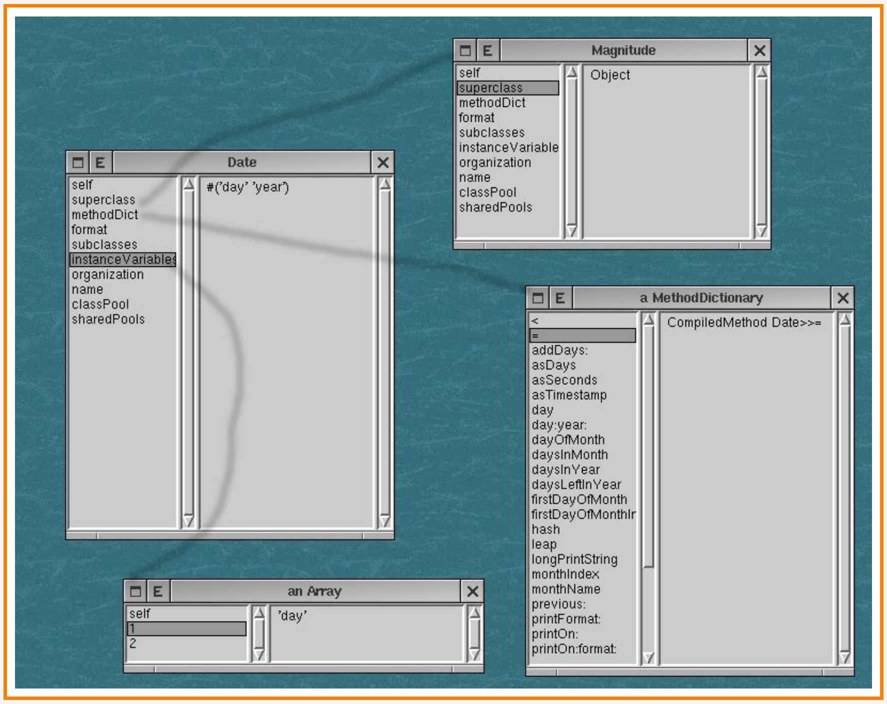

Figure *(12) – l'Inspection d'une classe montre que ses champs sont conformes à la déclaration des attributs réalisés dans la classe* `Class`.

4.3.5 Variable de classe versus variable d'instance de métaclasse

— **Rappel** : une variable de classe *(static en Java)* définie sur une classe `C` détient une valeur partagée par toutes les instances (au sens large) de `C` ;
  elle est accessible dans les méthodes d'instances et dans les méthodes de classes de `C` (et de ses sous-classes).

— **Nouveauté** : une variable d'instance [7] `I` définie sur une classe `C` permet à chaque instance de C de détenir une valeur propre pour I;
  Si `C` est une métaclasse, on dira que I est une **variable d'instance de métaclasse** ; toute instance de `C` est une classe et possède une valeur pour `I`.

4.3.6 Utilisation standard des méta-classes : 1) création d'instances initialisées

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

```
Object subclass: #Pile
       instanceVariableNames: 'index buffer '
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Exercices'
...
```

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

!Pile class methodsFor: 'creation'!
example
       "self example"
```

7. On rappelle que le terme "variable d'instance" dénote en Smalltalk ce qui est usuellement nommé "attribut".

```
        "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
```

4.3.7 Utilisation standard des méta-classes : 2) création de nouvelles méta-classes

Si l'on souhaite faire d'une classe une "memo classe" afin qu'elle mémorise la liste de ses instances, la solution naturelle est d'utiliser une variable d'instance de métaclasse.

```
EtudiantMemo class
    instanceVariableNames: 'tous'

initialize
tous := OrderedCollection new.

new
   | i |
   i := super new.
   tous add: i.
   ^i

tous
   ^tous
```

4.3.8 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, définissant une 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 de méthode `bar`

- une instance b de B

alors : 'b foo' lève une l'exception "B ne comprend pas le message bar".

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 et hérite donc de la méthode `bar`.

Problème de Compatibilité Descendante

Soient :

- une méta-classe MA définissant une méthode `bar` : 'self new foo'

- une classe A instance de MA définissant une méthode `foo`.

- une méta-classe MB, sous classe de MA

- une classe B instance de MB

'B bar' peut lever une exception si aucune méthode foo n'est définie dans une superclasse de B, en particulier si B n'est pas sous-classe de A.

4.3.9 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.

4.4 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).

Exemple d'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.

4.4.1 Version Visualworks7

```
1  !Symbol methodsFor: 'catch-throw'!
2
3  catch: aBlock
4      "execute aBlock with a throw possibility"
5      aBlock value.
6
7  returnToCatchWith: aValue
8          "Look down the stack for a catch, the mark of which is self,
9          when found, transfer control (non local branch)."
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
```

Listing (12) *– version Visualworks-Smalltalk*

4.4.2 Version Pharo6

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

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

returnToCatchWith: aValue
        | catchMethod currentContext |
        currentContext := thisContext.
        catchMethod := Symbol compiledMethodAt: #catch:.
        [currentContext method == catchMethod and: [currentContext receiver == self]]
                whileFalse: [currentContext := currentContext sender].
        currentContext return: aValue.
        ^aValue
```

Listing (13) *– version Pharo-6*

Exemple d'utilisation :

```
^#Essai catch: [
        Transcript show: 'a';cr.
        Transcript show: 'b';cr.
        Transcript show: 'c';cr.
        #Essai returnToCatchWith: 22.
        Transcript show: 'd';cr.
        33]
```

Listing (14) *– should display a b c (not d) in the Transcript and return 22 (not 33)*

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

Le compilateur Smalltalk est écrit en Smalltalk et est un objet de première classe. 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 cet exemple, les formules associées aux cellules sont représentées par des méthodes définies dynamiquement par le programme, lexicalement analysées 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`.

```
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]
```

Remerciements

Merci à René Paul Mages pour ses relectures attentives et remarques pertinentes. https://renemages.wordpress.com/smalltalk/pharo/