Université Montpellier-II
UFR des Sciences - Département Informatique

Objets Avancés

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

Notes de cours - 2007-2018
Christophe Dony

1 Introduction

1.1 Idées clé

1. Environnement intégré de génie logiciel → agilité, prototypage rapide,

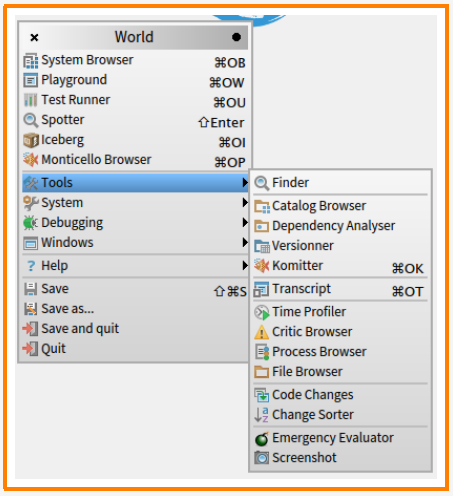

Figure *(1)* – *Environnement Pharo-6.1*

2. Langage à objet simple, (5 idées)
3. Langage réflexif → Systèmes auto-adaptables, IDM à runtime, Expérimentation recherche

1.2 Liens historiques et pratiques

— Alan Kay (https://fr.wikipedia.org/wiki/Alan_Kay).
— 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 Eliott 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)

— http://www.threeriversinstitute.org : site de Kent Beck, fondateur du "Agile development" avec Smalltalk.

— POURQUOI ? :
http://stackoverflow.com/questions/1821266/what-is-so-special-about-smalltalk

1.3 Le langage

— Objet : influencé par *Simula* (abstraction de données, polymorphisme d'inclusion)

— influencé par Lisp

  — typage dynamique (pas de problèmes de covariance, pas de contrôles statique, pas de preuves),

  — références simples généralisées (repris par Java),

  — allocation dynamique et récuparation dynamique et automatique de la mémoire,

  — style applicatif (toute instruction est une expression),

  — compilation en instructions d'une machine virtuelle,

  — réflexif : programmes = données.

Le langage en cinq 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'attribus 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 (liaison dynamique généralisée)
4. Toute classe (sauf `ProtoObject`) est définie comme une spécialisation d'une autre. La relation de spécialisation définit un arbre d'héritage.
5.

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.

Exemple de méthode et de bytecode généré

```
  caption={définition de la méthode {\tt factorial} sur la classe {\tt PositiveInteger}}]
fact
    ^self = 0
        ifTrue: [1]
        ifFalse: [self * (self - 1) fact]
```

```
normal CompiledMethod numArgs=0 numTemps=0 frameSize=12
literals: (#fact )
1 <44> push self
2 <49> push 0
3 <A6> send =
4 <EC 07> jump true 13
6 <44> push self
7 <44> push self
8 <4A> push 1
9 <A1> send -
10 <70> send fact
11 <A8> send *
12 <66> pop
13 <60> push self; return
```

1.5 Premiers exemples

Classes et Instances

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

initialize
    valeur := 0.

get
    ^valeur

set: anInt
    valeur := anInt.

incr
    valeur := valeur + 1.
```

```
20 decr
21     valeur := valeur − 1.
```

Listing (1) *– classe* `Compteur`*, définition et méthodes d'instance*

```
1   caption={méthodes de classe (ou static en Java)}]

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

7  example
8      | c |
9      c := Compteur new.
10     c set: 33.
11     ^c
```

1.6 La classe Compteur dans l'environnement

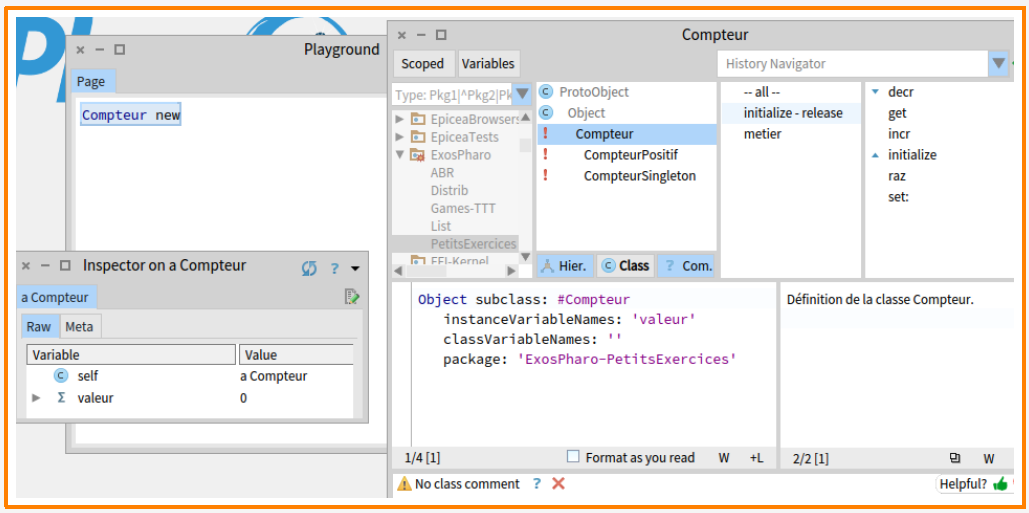

Figure ***(2)*** *– Définition de la classe* `Compteur` - Pharo *Env.*

1.7 L'exception universelle : `doesNotUnderstand`

```
1  'abc' factorial
```

Instance of byteString did not undestrand factorial

Conduite à tenir :

— lire le message

— si besoin, 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 (2) *– 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 : #(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 (3) *– à la Algol, ou à la Pascal, comme avant C*

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 ";").

```
m "une méthode''
    i := 1.
    j := 2.
```

2.5 Retour à l'appelant

```
^33 "(return)"
```

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

2.6 Envoi de message

Définitions :

— Conceptuelle : demande à un objet d'exécuter un de ses comportements implanté sous la forme d'une méthode,

— Pratique : appel de méthode avec sélection 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"* a $n (n >= 2)$ arguments dont le receveur.
   Leur forme syntaxique (originale) permet de représenter un envoi de message comme une "petite conversation" (*small talk*) 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.`

2.7 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>* ]`

2.7.1 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]
```

2.7.2 Plus sur les fermetures

Les fermetures en Smalltalk sont implantées 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 *lvalue* (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
```

2.7.3 Fermeture et retour de méthode

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

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

"la méthode de classe test2 définie également sur Object
invoque la fermeture"
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.

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* (pas d'équivalent en Java (à l'heure où j'écris ces lignes)).

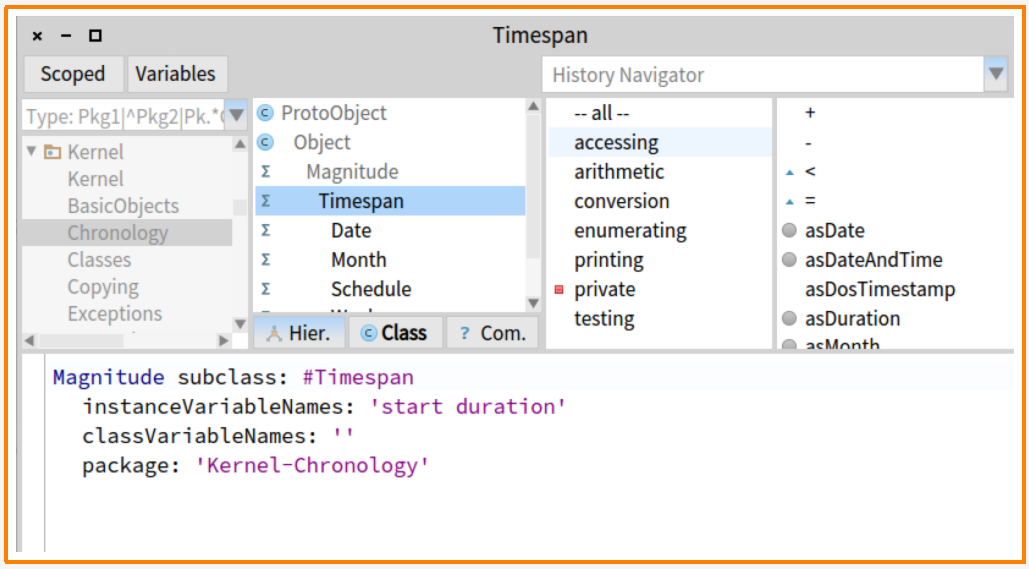

Figure *(3)* – *Définition de la classe* `TimeSpan` - Pharo *env.*

Méthodes d'instance

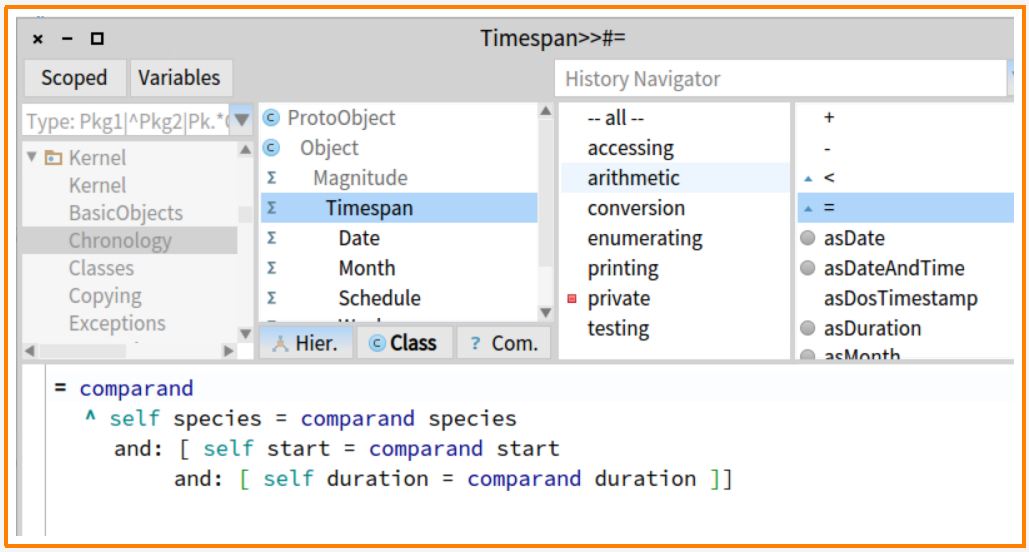

Figure *(4)* – *Méthode d'instance "=" de la classe* `TimeSpan`

Méthodes de classe

Méthodes invocables par envoi de message aux classes [1]

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.

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

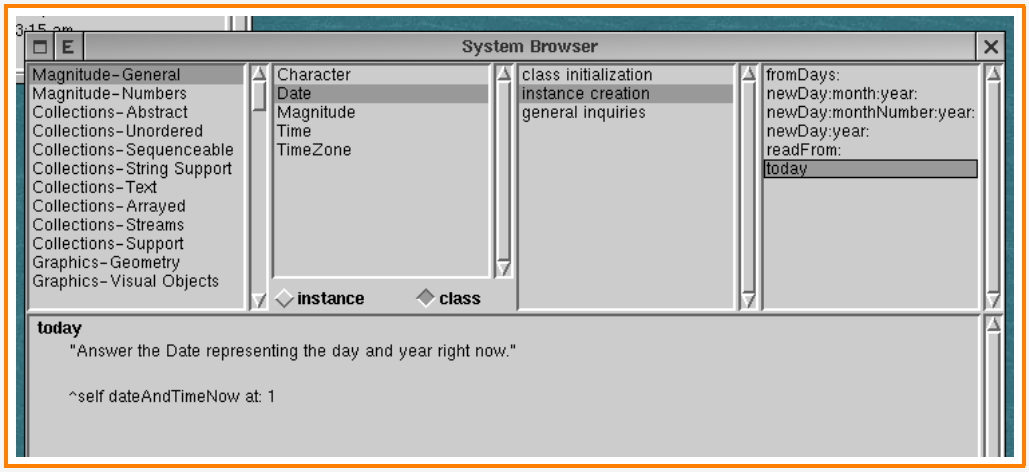

Figure ***(5)*** – *Méthode de classe "today" de la classe Date (En fait une méthode d'instance de la (méta)-classe "Date class")*

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) définis sur la classe du receveur,
- 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.

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

Exemples : 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 être rendue abstraite en la définissant de la façon suivante :

```
method
    self subclassResponsibility
```

3.7 Sous-types et Héritage

— Héritage simple.

— Redéfinition sur la base du nom des méthodes. Toute définition sur une sous-classe d'une méthode de même nom existant sur une superclasse est une redéfinition.

— Typage dynamique :
  – + pas de surcharge,
  – + pas de problème de redéfinition covariante, pas de *downcast*,
  – - aucun contrôle statique de correction des types.

— 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 Schémas de spécialisation

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

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

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

3.9 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 #()
```

```
32                  HtmlAttributes #()
33                  IdentityDictionary #()
34                      SystemDictionary #('cachedClassNames')
35                          Environment #('envtName' 'outerEnvt')
36                              SmalltalkEnvironment #()
37                  LiteralDictionary #()
38                  MethodDictionary #()
39                  PluggableDictionary #('hashBlock' 'equalBlock')
40                  WeakKeyDictionary #()
41                      ExternalFormRegistry #('lockFlag')
42                      WeakIdentityKeyDictionary #()
43                  WeakValueDictionary #()
44              IdentitySet #()
45              PluggableSet #('hashBlock' 'equalBlock')
46              WeakSet #('flag')
47          SkipList #('sortBlock' 'pointers' 'numElements' 'level' 'splice')
48              IdentitySkipList #()
49          WeakRegistry #('valueDictionary' 'accessLock')
```

3.10 Héritage et description différentielle

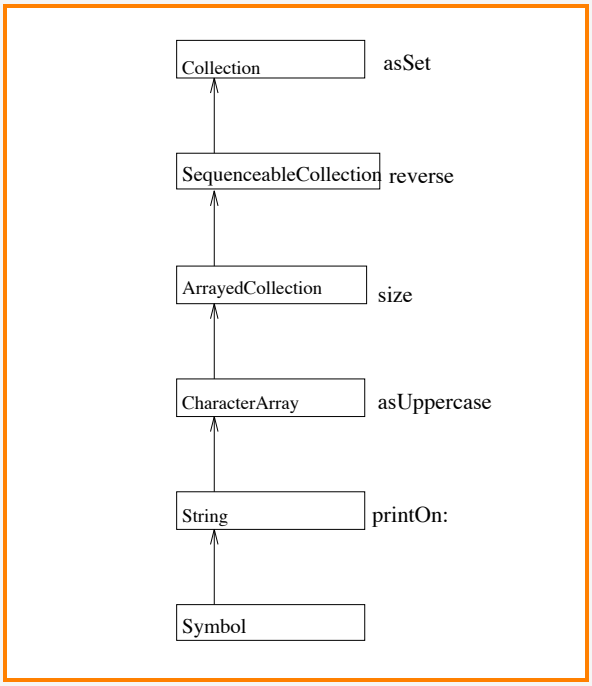

Figure *(6)* – *Hiérarchie d'héritage avec ajoûts de fonctionnalités. Pour plus de détails sur la description différentielle, voir notes de cours "Réutilisation/Frameworks".*

4 La méta-programmation en Smalltalk

4.1 Définitions

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

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

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

4.2 Méta-programmation basique

4.2.1 Les classes comme "rvalues" standards - Généricité

Le terme "généricité" désigne en Java la capacité à contraindre les structures de données de type collection 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 le 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]
```

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

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

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

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

La hiérarchie des classes représentant les nombres.

```
1  Magnitude
2      Number ()
3          FixedPoint ('numerator' 'denominator' 'scale')
4          Fraction ('numerator' 'denominator')
5          Integer ()
6              LargeInteger ()
7                  LargeNegativeInteger ()
8                  LargePositiveInteger ()
9              SmallInteger ()
10         LimitedPrecisionReal ()
11             Double ()
12             Float ()
```

4.2.3 L'objet `nil`.

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

Elle est l'unique instance *(Singleton)* de la classe `UndefinedObject`, qui possède, et sur laquelle il est possible de définir, des méthodes.

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

L'exception `nullPointerException` n'existe pas !

```
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: 'TP3'!

!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 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), *Block* (boucle "tantque"), `Integer` (boucle "for").

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

4.3 Les Méta-classes en Smalltalk

4.3.1 Rappel : la solution Objvlisp

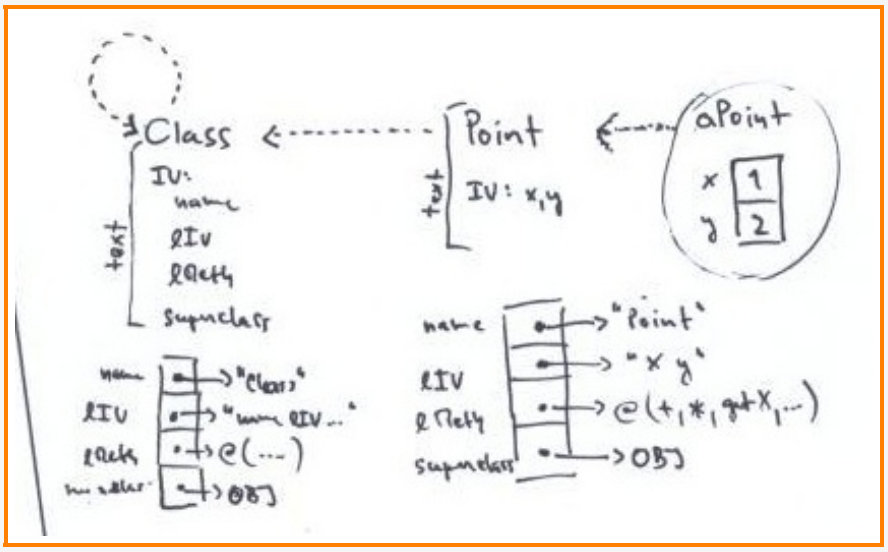

Figure *(7)* – *Une classe et une méta-classe sont structurellement et fonctionnellement identiques*

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

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

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

L'inconvénient de cette solution est qu'elle peut poser des problèmes de compatibilité (voir section 4.3.7). *CLOS* 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 [2] :

— 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 les incompatibilités.

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

4.3.3 Architecture du système de métaclasses Smalltalk

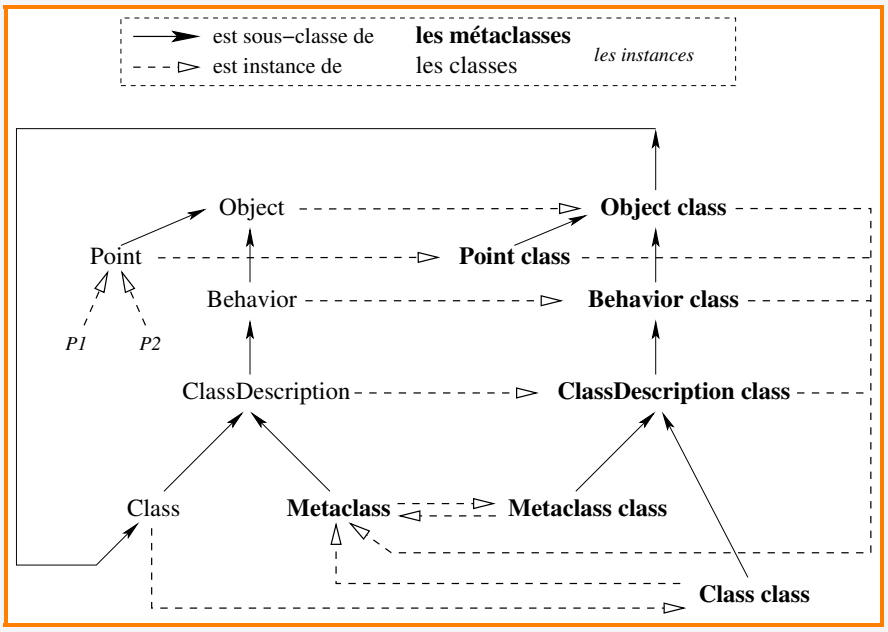

Figure ***(8)*** *– Classes et Métaclasses : Hiérarchies d'héritage et d'instantiation. (figure : G.Pavillet)*

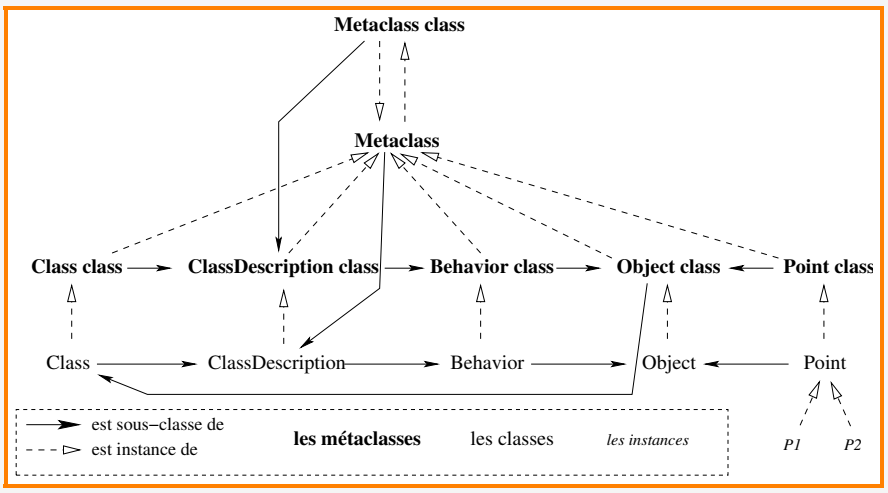

Figure ***(9)*** *– Classes et Métaclasses : couches d'instantiation. (figure : G.Pavillet)*

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 [3] du lien d'héritage, ainsi : une `Object class`, par exemple `Object)`, est une sorte de classe.

— 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. 10)

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

Figure *(10)* – *Les classes "Class" et "Metaclass" (et leurs attributs) définissant les classes et les méta-classes. Elles ont en commun tout ce qui est hérité de "ClassDescription".*

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

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

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 [4].

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

4. Exercice difficile : créer une sous-classe MC de la classe `Metaclass`, qui définirait un nouveau type généraliste de meta-classes. La difficulté n'est pas dans la création de cette classe mais, comme les metaclasses sont crées automatiquement par le système, de modifier le système de création des classes pour qu'une classe qui en relève ne soit pas une instance de `Metaclass` mais de `MC`.

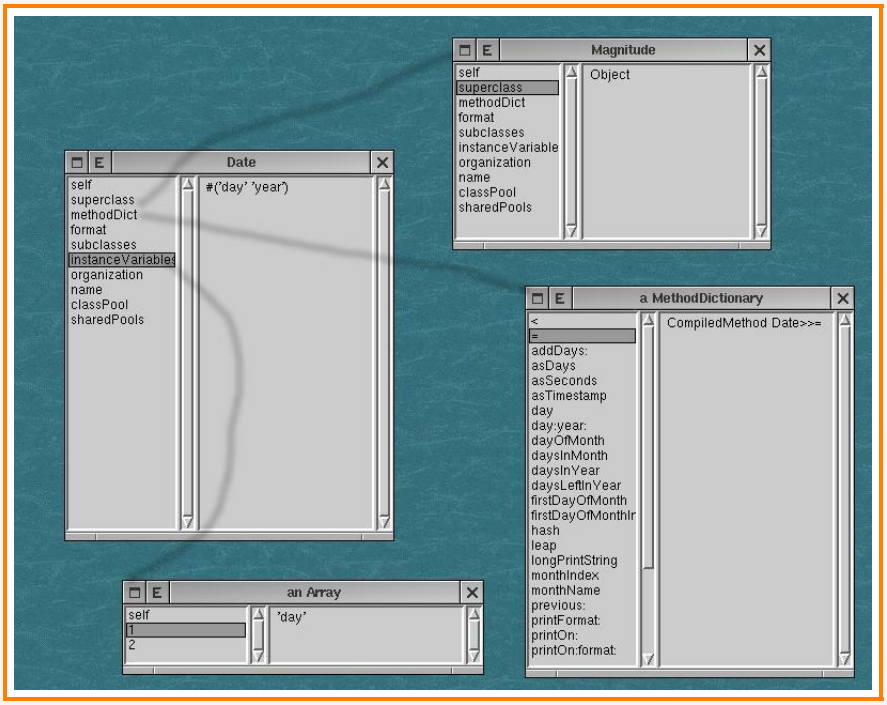

Figure ***(11)*** – *l'Inspection d'une classe montre que ses champs sont conformes à la déclaration des attributs réalisés dans la classe* `Class`.

4.3.5 Variable de classe et variable d'instance de métaclasse

Un point clé est la différence entre variable de classe et variable d'instance [5] de métaclasse et ce d'autant plus qu'il n'y a pas de différence entre méthode de classe et méthode d'instance de métaclasse.

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

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

Les variables d'instance de métaclasse ne sont accessibles que dans les méthodes de classe. Ce sont les attributs définis par les métaclasses. 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.

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

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

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

5. On rappelle que le terme "variable d'instance" dénote ce qui est usuellement nommé "attribut" dans les autres langages.

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

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

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

new: taille
	^super new initialize: taille!

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

4.3.7 Avantage du modèle Smalltalk : la compatibilité des métaclasses

La solution Smalltalk ne pose pas de problèmes de compatibilité.

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

Problème de compatibilité Ascendante

Soient :

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

- une classe B sous-classe de A

- la classe de A et sa méthode `bar`

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

- une instance b de B

alors : 'b foo' lève une exception.

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

Problème de Compatibilité Descendante

Métaclasse MA, méthode bar : 'self new foo'

Métaclasse MB, sous classe de MA,

'mb bar' peut lever une exception.

4.3.8 Inconvénient du modèle Smalltalk

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

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

4.4 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 `throw` interromp l'exécution strandard et la fait reprendre à l'instruction qui suit le `catch` correspondant.

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

```
1  !Symbol methodsFor: 'catch-throw'!
2
3  catch: aBlock
4      "execute aBlock with a throw possibility"
5      aBlock value.!
6
7  throw: 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! !
```

Exemple d'utilisation, dans un "workspace" :

```
1  #Essai catch: [
2      Transcript show: 'a';cr.
3      Transcript show: 'b';cr.
4      Transcript show: 'c';cr.
5      #Essai throw: 22.
6      Transcript show: 'd';cr.
7      33
8      ]
```

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

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

L'exemple suivant est extrait de la réalisation d'un mini-tableur en Smalltalk. Dans 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]! !
```