PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

A Tutorial Introduction to Metaclass Architecture as provided by Class Oriented Languages

Pierre COINTE
Rank Xerox France & LITP
12 Place de L'Iris, 92071 Paris La Défense
cointe@inria.inria.fr seismo!mcvax!inria!litp!pc

Abstract

This paper investigates the relation between computational reflection and class-based object-oriented programming. We study the relation between the abstraction mechanism provided by a class language and the reflective technic introduced by B.C. Smith.

We observe that object-oriented programming promotes the encapsulation of a structure and of a behavior to represent uniformly - as an object - all the entities manipulated by the language whereas computational reflection aims at describing the structure and behavior of all implicit aspects of a language.

We call structural (behavioral) reflection the way to reason upon and interact with the object structure (behavior). We observe that the uniform class taxonomy proposed by languages such as Smalltalk-80, ObjVlisp and the CommonLisp ObjectSystem (CLOS) induces a metaclass architecture which introduces both structural and behavioral reflections. Because they manipulate classes as true objects and because they have the ability to treat programs as datas, these three languages are the privileged candidates to investigate the power of metaclass.

Consequently we introduce the kernel architectures provided by Smalltalk, ObjVlisp and CLOS metaclasses. After having explained these architectures we can deduce the main features of these languages and therefore we develop a metaclass programming style. This style allows us to extend a class system through new mechanisms such as "differential error handling", generic classes and partwhole hierarchies.

1 Reflection and Object-Oriented Programming

"As described in Smith a reflective computational system is one in which otherwise implicit aspects of the system's structure and behavior are available for explicit inspection and manipulation" [15].

Today two terms are becoming very popular : `Computational Reflection` and `Object-Oriented Programming` even if their meanings are not yet formalized. The goal of this paper is to discuss the balance between the encapsulation principle as developed by object-oriented languages and the intrinsic reflection associated with the manipulation of an object representation.

Because they are now the best defined and the best known, we focus our study on languages using the `class abstraction` to represent, create and manipulate objects.

1.1 The Object Paradigm

"A central new concept in SIMULA 67 is the "object". An object is a self-contained program (block instance), having its own local data and actions defined by a "class declaration". The class declaration defines a program (data and action) pattern, and objects conforming to that pattern are said to "belong to the same class"" [13].

The common definition of an object is the encapsulation of a structure and of a behavior. The structure expresses the state of the object while the behavior defines a set of potential actions which are used to interface the object with the external world. Compared with procedural programming, object oriented programming associates a structure defined by a set of variables (also called fields, instance-variables, or slots ...) with a functional behavior defined by a set of procedures (also called methodDictionary or script).

Message passing is the protocol used to communicate with the object. Sending a message to an object (the receiver) means to select one action and then to execute it in the context of this object. Obviously, this action can modify the local state of the receiver or-and compute a value.

Programming with objects means to simulate the real world by abstracting objects interacting and communicating with each other through message passing. This simulation leads to study and solve the following topics [4] :

1. object creation, inspection and destruction,
2. object organization,
3. object specialization,
4. object activation.

Each language claiming the object-oriented label must explain how it answers the previous questions. For instance, objects can be created from an abstract generator or from every previous object, organized in classes or sets, specialized by using inheritance or delegation, activated sequentially or concurrently. In fact object-oriented languages could be classified into - at least - four sub-families. The `class` family led by Smalltalk, the `abstract data-type` family led by Clu, the `actor` family led by Act* and ABCL* and the `frame` family initiated by KRL & FRL.

Independently of this classification, we have to observe that in order to increase the integrity of the object model, more and more object-oriented languages attempt to solve the "Uniformity postulate". The idea is simply to extend the object paradigm to all the entities manipulated by the language and its interpreter-compiler. Consequently, "rock bottom" entities such as Numbers, Strings or Vectors, but also more idiosyncrasic ones such as Messages, Methods, Environments, and Continuations are represented as objects.

Then, to exhibit and make accessible the structure and the behavior of every object, the next step is to use a metaobject to control (represent and modify) each object. This approach was pioneered by the Smalltalk metaclasses and is currently in progress in the actor world [25] [16]. and in the frame world [24] Obviously in those systems, because each entity is an object and because each object is defined by a metaobject, a metaobject must also be an object described by a metametaobject. Traditionally, this scheme leads to an infinite tower of metaobjects and requires some specific implementation technics such as "explicit bootstrap" (cf 3.2.a) or lazy evaluation.

This representation problem is related to the idea of describing the architecture of an interpreter in the language itself and is consequently very close to Smith's work about Reflection and Reification as materialized by 3-LISP.

1.2 The Goal of Reflection

3-LISP idea was to build a reflective tower supporting a series of interpreters, interpreting each other and connected by two metalevel operations: `reflection` and `reification`. The `reification` process makes avalaible the data structures of an interpreter to the program it is running. On the contrary, the `reflective` process allows to use the program to alter its interpreter structure. Among the meta-level facilities required by a reflective language we can mention :

1. the ability to manipulate a program as a data (cons-cell in Lisp, string in Smalltalk),
2. the ability to call the interpreter-compiler explicitly (the Lisp function `eval` or the Smalltalk methods `compile:` & `compile:classified:`),
3. the ability to build new control structures (the Lisp `fexpr` and the Smalltalk couple `(block,value)`),
4. the ability for the interpreter to describe itself by using metacircular definitions,
5. the ability to control the allocation and lifetime of every entity [23].

Some abilities are linked more precisely to the description of the control structure while others are more related to the specification of the structures of the data. To distinguish between them we will use the two terms: behavioral reflection and structural reflection.

1.3 The Class model from a Reflective point of view

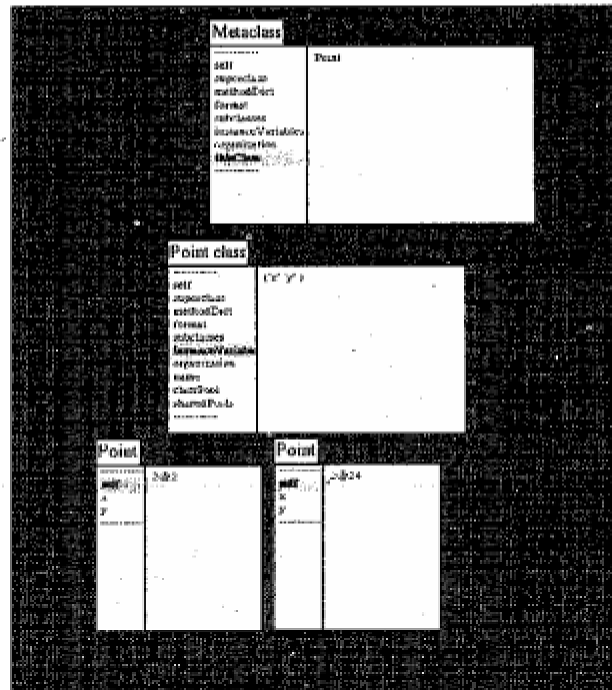
"The purpose of the Smalltalk project is to support children of all ages in the world of information. The challenge is to identify and harness metaphors of sufficient simplicity and power to allow a single person to have access to, and creative control over, information which ranges from numbers and text through sounds and image. In our experience, the Simula notion of class and instance is an outstanding metaphor for information `structure`. *To describe* `processing`, *we have found the concept of message passing to be correspondingly simple and general"[20].*

First Simula and then Smalltalk have developed a class taxinomy to describe and organize the objects of the world. A "normalized" definition of a class, as accepted by the CLOS committee, is the following one: *"a class is a new object that determines the structure and behavior of a set of other objects which are called its instances" [26].*

The class model obeys to both philosophical and implementory motivations. Philosophical, because since Plato, the abstraction of "a Form" is a way to represent (modelize) the real world. Implementory, because the use of an abstraction is the manner to describe the structure and the behavior of the set of objects which are instances of a same class.

From a reflective analysis a class can be seen as the partial descriptor of its instances. This class defines their common structure by a set of instance variables and their common behavior by a set of methods. Nevertheless each instance holds its own set of values isomorphic to the instance variables of its class.

In a "pure" class model each entity is uniformly treated as an instance of a single class. Consequently classes are also true objects defined by other classes called metaclasses. The next figure illustrates the Smalltalk representation of simple instances and classes as viewed by the inspector :



We recognize :

- two objects, instances of the class `Point` . Each of them shares the same structure defined by the x and y instance variables plus a link towards their common class. This link is materialized by the label of the associated view. Each "point" holds its own value for the x, y variables and then maintains its own state,
- the class `Point` itself. As an object, the structure of `Point` is defined by the `superclass`, `methodDict`, `format`, `subclasses`, `instanceVariables`, `organization`, `name`, `classPool`, `sharedPools` variables. The value of `instanceVariables` is precisely the array `('x' 'y')` defining the common structure of `Point`'s instances. The value of `methodDict` is the dictionary of methods defining the common behavior of all the points. When an instance of Point receives a message, the "lookup" method will start at the `Point`'s methodDictionary. The structure of `Point` presented as an object establishes the existence of a subclassing mechanism operating at the class level. The two instance variables `superclass` and `subclasses` are used to handle a single inheritance hierarchy. The "method lookup" starting at the class level, will follow the superclass link,
- the (meta)class `Point class` instance of `Metaclass`.

Notice that both instances and classes use the pseudo-variable `self` which expresses the fact that each object has a (self)knowledge about itself.

1.4 Programming by Metaclasses

"Smalltalk uses classes to describe the common properties of related objects. Unfortunately, the use of classes and metaclasses is the source of a number of complications ... One source of the complexity surrounding classes in Smalltalk is the interaction of message lookup with the role of classes as the generators of new objects, which gives rise to the need for metaclasses" [8].

As an abstraction, a class is also a generator of objects. To be an instance of a class means having been created by this class. The creation of a new object operates by allocating a structure isomorphic to the instance variables of the class. Providing new allocators, e.g. new kinds of structures, leads to the need for new metaclasses. For instance, the implemention of `generic stacks` will require each class to hold a slot `type` (cf. 2.3.2) while the implementation of the parthwole hierarchy will require the additional slot `parts` (cf. 2.3.3).

The rest of this paper studies Smalltalk, ObjVlisp and CLOS metaclasses both from a structural and a behavioral point of view. Our goal is to demonstrate that metaclasses can be easily understood and that they provide :

- the way to design the self-descriptive architecture of an (object-oriented) system,
- the natural pedagogical tools to teach and learn this architecture,
- and the obvious hooks to extend this architecture towards new semantics.

Our chronology: Smalltalk, ObjVlisp and CLOS is historical. Smalltalk metaclasses are the most hairy probably because they were conceived more for the implementor than the user and consequently not integrated in a uniform way. ObjVlisp metaclasses are the cleanest because they take benefit of the Smalltalk and Loops [3] experiences of metaclasses. They were designed to support a unified but extensible class system. Finally, the oncoming CLOS metaclasses are looking promising because they are built to describe new objects such as `generic function`, `slots`, `method combinations`, `types` and to handle various extensions (persistents objects, framestyle languages).

2 Smalltalk-80 Metaclasses

Smalltalk-72 classes were not yet objects. Metaclasses were invented by Smalltalk-76 in order to express the behavior of classes as true objects able to handle message passing. Nevertheless all classes were instance of a sole metaclass (`Class`). Consequently all classes shared the same behavior and it was impossible to act on the structure of a class.

Defining classes as object-receivers conducts Smalltalk-80 to associate with each class a metaclass whose methodDictionary defines the protocol of this class. For instance, an object creation is performed by sending the message new to its class. This message allocates a new instance whose instance variables are (by default) initialized to "`nil`". Smalltalk-80 metaclasses are stil seldom used to control or describe the structure of classes themself. The first implementation of Smalltalk-80 realized by Apple on the Macintosh, as the first implementations of Smalltalk-V realized by Digitalk, do not support the definition of instance-variable at the metaclass level.

On the one hand, the standard use of (meta)classes is guided by the Smalltalk browser which partially hides the metaclass architecture. A switchView allows to switch between the class and the metaclass levels. Nevertheless, because a class creation automatically provides the creation of an implicit metaclass, "standard users" have some difficulties to discover and then to understand the metaclass concept. On the other hand, as demontrated by the reading of the Smalltalk system, implementors have relatively used the metaclass facilities, more generally to act on the initialization behavior, to control the access to the "so-called class-variables" and to increase the class structure by adding new instance-variables (see for example the instance variable `screenMask` of the metaclass `Form class`).

After recalling the Smalltalk postulates we shall comment on its class architecture and then propose "a guided tour"of the Smalltalk kernel. Then we develop several metaclass examples demonstrating how :

1. to control the initialization protocol of any object,
2. to use metadescription to parameterize abstraction itself (for instance to simulate generic classes à la Eiffel),
3. to control the error treatment definition within different metaclass levels (for instance to provide automatic method creation when a method is missing),
4. to graft new metaclasses simulating new sub-systems such as Borning's Thinglab.

Some of these examples will be reused to introduce ObjVlisp and CLOS metaclasses.

2.1 The Smalltalk-80 Postulates

Following the Ingalls presentation of Smalltak-76 [20], we characterize Smalltalk-80 by these five postulates :

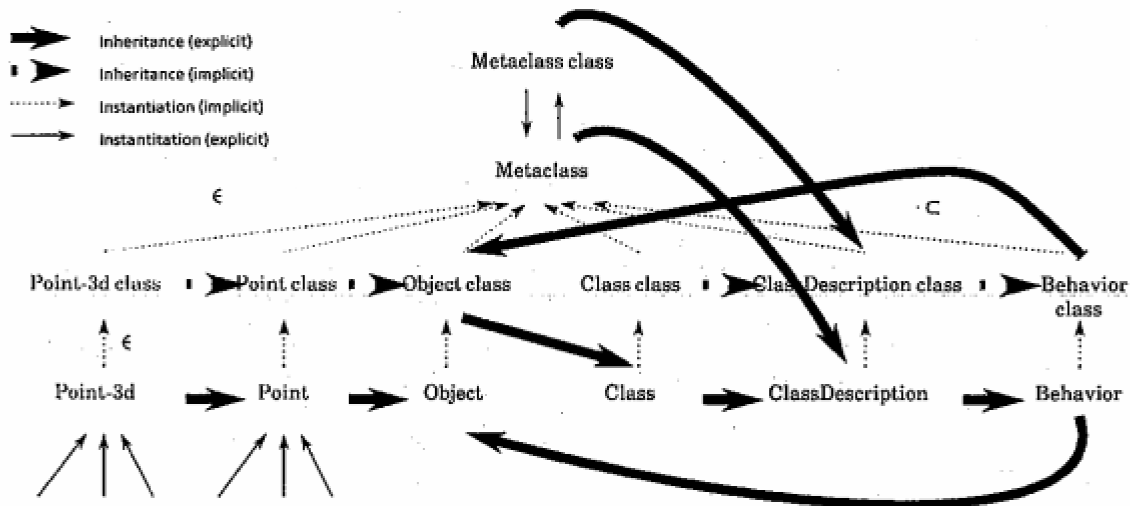
1. EVERY entity is UNIFORMLY defined as an object,
2. each object belongs to a single class. This class describes the common state (via instance variables) and behavior (via methods) of the set of its instances. Particularly, the message `class` sent to any object returns the name of its class,
3. classes are organized into a single inheritance hierarchy. Inheritance rules allow methods (dynamic) and instance variables (static) sharing. The root of the inheritance tree is the class `Object`. `Object` owns the set of methods shared by all the Smalltalk-80 objects, for instance `class`, `inspect`, `doesNotUnderstand:...`,

4. classes are also objects defined as the sole instance of their metaclasses. Metaclasses are IMPLICITLY created at the same time as their sole instance. Because of their implicit creation, the inheritance of metaclasses is parallel to the inheritance of classes: if a class is a subclass of another one, then its metaclass is the subclass of the metaclass of the other. The root of the metaclass's inheritance tree is the class `Object class` defined as a subclass of `Class`. Metaclasses are equally objects, instances of `Metaclass`,

5. message passing is the only way to communicate with objects. When a message is sent to an object the system looks up for a method defined by the selector name - first in the class of the receiver and - then in one of its superclasses. The associated method is then applied to the receiver and the arguments of the message. When the lookup fails, the `doesNotUnderstand:` held by `Object` is applied.

2.2 Smalltalk-80: the class architecture

The next figure summarizes the previous postulates by materializing the instantiation and subclassing relations introduced by the definition of the class `Point` and its subclass `Point-3d` :



To make the analyze of this complex "network" (see also Figure 16.5 page 272 of [17]!) easier we discuss separately the "instance-of" and "subclass-of" relations. Notice that because all metaclasses are anonymous we refer to a metaclass by the message `class` sent to their sole instance: `Point class` represents the metaclass of `Point`.

instantiation: When `Point` (`Point-3d`) is created, its metaclass `Point class` (`Point-3d class`) is automatically generated by the system. As an object, `Point class` is instance of `Metaclass`, which itself is instance of `Metaclass class`. The loop between `Metaclass` and `Metaclass class` materializes the Smalltalk solution to the infinite "tower of metaclasses". We can distinguish five different levels:

1. the "instance level" defined by the leaves of the tree. These objects are not generators i.e. cannot be instantiate,

2. the "class level" defined by named generators which describes the behavior and structure of the instance level. Potentially, each class can produce an infinite set of instances,

3. the "metaclass level" isomorphic to the "class level" defined by anonymous generators which create and describe only one instance,

4. the "metametaclass level" reduced to the sole class `Metaclass` whose instances are all the Smalltalk metaclasses,

5. the "metametametaclasslevel" reduced to the root i.e. `Metaclass`.

Because classes and metaclasses are strongly coupled, the user has no control over the metaclass level which is automatically handled by the system. The instantiation tree could be simplified by contracting the class and metaclass levels, i.e. by decoupling metaclasses and their classes. This contraction corresponds to the ObjVlisp solution which conducts to uniformize the class and metaclass concept. Contrary to Smalltalk, an ObjVlisp metaclass can instantiate several classes and can be defined explicitly as an instance (and a subclass) of a previous metaclass.

inheritance: The inheritance mechanism provided by Smalltalk is single inheritance (even if multiple inheritance is also supported cf. 2.3.4). Consequently a standard class can inherit directly from only one superclass. We can check that the inheritance of metaclasses is parallel to the inheritance of classes: `Point-3d class` is automatically defined as a subclass of `Point class` which is itself a subclass of `Object class`. `Object class`, which is the root of the metaclass inheritance lattice, is defined as a subclass of `Class`.

In fact, each Smalltalk entity shares a default behavior defined by the class `Object` level. Specialized objects such as classes and metaclasses are defined by the two classes `Class` and `Metaclass` which explicit the structure&behavior of standard classes and standard metaclasses. Obviously these common behaviors and structures are expressed through the inheritance rules between `ClassDescription` and `Behavior`.

Let us recall that inheritance rules are static for the object structure and dynamic for the object behavior. When creating a class, the resulting instance variables are calculated as the union of the instance variables owned by its superclass with the instance variables explicitly specified in the class definition. On the other hand, method inheritance is dynamic. When the method lookup fails in the receiver class then the search continues along the inheritance path.

To fully explicit the Smalltalk kernel we detail the instance variables and the methods defining the standard structure (S) and behavior (B) of classes and metaclasses.

2.2.1 The kernel : Behavior, ClassDescription, Class & Metaclass

`Class` and `Metaclass` are defined at the same hierachy level and share the structure&behavior explicited by the path: `Object, Behavior and ClassDescription` :

```
Object ()
      Behavior (superclass methodDict format subclasses)
            ClassDescription (instanceVariables organization)
                  Metaclass (thisClass)
                  Class (name classPool sharedPools)
                        Object class ()
                              Behavior class ()
                                    ClassDescription class ()
                                          Metaclass class ()
```

Behavior (superclass methodDict format subclasses)

S: `Behavior` handles the functional behavior of objects via the method descriptions `methodDict` and the lookup path as expressed by the inheritance links (`superclass` & `subclasses`). The `format` is used both to distinguish between indexed or non indexed classes and to memorize the size of the object structure.

B: The primitive `new` is the GENERAL allocator which creates an object by allocating a structure isomorphic to the descriptor of its class's instance-variables. This method is inherited by all classes and metaclasses. When you create an object, let's say a Point, by "`Point new`", this allocator is used because `Point class` is an undirect subclass of Behavior.

Class (name classPool sharedPools)

S: classes are named (`name`), they maintain a first global environment shared by all the instances of a class hierarchy (`classPool`) and a second environment (`sharedPools`) shared by a set of arbitrary classes.

B: the method `subclass:instanceVariableNames:...:category` is used to create simultaneously the classes and their private metaclasses (cf. 2.2.2).

Metaclass (thisClass)

S: `Metaclass` defines classes which are anonymous and constrained to have a single instance. The instance variable `thisClass` handles the backward pointer between a metaclass and its sole instance.

B: The constraint forcing a metaclass to own a single instance is reflected by the redefinition of the method `new`. *"Before calling the primitive new, check if the metaclass has already an instance. Create it by calling Behavior.new or complain if that one already exists [18] :"*

```
!Metaclass methodsFor: 'instance creation'!
new
      thisClass == nil
            ifTrue: [↑ thisClass ←  super new]
            ifFalse: [self error: 'A Metaclass should only have one instance!!'] !
```

2.2.2 Instantiation: The magic of new revelated

Each Smalltalk beginner has asked himself the following question: where are defined the new methods creating instances, classes and metaclasses. The answer to this mysterious question is that Smalltalk uses only one allocator defined by `Behavior` and redefined by `Metaclass`.

We want to discuss here this paradox: Smalltalk's reflective architecture is partially hidden by the programming environment and more precisely by the ergonomy of the browser. Every entity (an instance, a class or a metaclass) is REALLY created by executing the primitive method new. Nevertheless, this uniformity does not appear in the standard pattern proposed by the browser to define a new (meta)class. For instance, the definition of the class Point uses the following template :

```
Object subclass: #Point
       instanceVariableNames: 'x y'
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Graphic-Primitives'
```

which is far from the general template used to create instances of Point : "Point new". It would be more uniform and rational to create a class by sending new to its metaclass. But because this metaclass does not yet exist (it will be created just before the class cf. below) this scenario is impossible and leads to define "a parasite" method defined in Class and named subclass:instanceVariableNames:classVariableNames:poolDictionary:category This method has to call the new allocator twice : the first time to create the metaclass, the second time to create its sole instance. A simplified definition of the scenario used to create the class Point could be the following one :

```
|aMetaclass aClass anInstance anotherInstance|
"1" aMetaclass ← Metaclass new.
    aMetaclass superclass: Object class.
    aMetaclass methodDict: MethodDictionary new.
    aMetaclass format: (Object class) format.
    aMetaclass instanceVariables: nil.
"2" aClass ← aMetaclass new.
    aClass superclass: Object
           methodDict: MethodDictionary new
           format: -8192
           name: #Point
           organization: ClassOrganizer new
           instVarNames: #('x' 'y')
           classPool: nil
           sharedPools: nil.
    aClass format: aClass format + aClass instVarNames size.
    Smalltalk at: #Point put: aClass.
    SystemOrganization classify: #Point under: #Graphic-Primitive.

"3" anInstance ← aClass new . anotherInstance ← aclass new.
```

1. the metaclass Point class is created as an instance of Metaclass, Because Point is a subclass-of Object the superclass slot must be initialized to Object class. The slot methodDict is initialized to an empty instance of MethodDictionary while format is computed from Object class,

2. Point class is instantiated to create its sole instance. The method new allocates an anonymous class aClass which becomes the value of thisClass. This anonymous class is then explicitly initialized (and named) by the message: methodDict:...:sharedPools:. This initialization is achieved through the computation of the value of format and the addition of the class in the global environment: the Smalltalk systemDictionary,

3. the class is now instantiable and can produce instances by receiving the general new.

2.2.3 Object & Behavior: Error handling

Many Smalltalk extensions use a redefinition of the error handler. From a reflective point of view, we have to observe that this error handler is represented by a method (doesNotUnderstand:) which can be partially parameterized at the metaclass level.

doesNotUnderstand: When a message "lookup" fails the method doesNotUnderstand: held by Object is applied. Before creating a Notifier allowing the user to open a debuger, tryCopyingCodeFor: is called (cf. below). This method has been implemented at the metaclass level "self class tryCopyingCodeFor:... to allow differential error treatments. This indirection towards the metaclass gives the hook to support generic handler and will be used by our partwhole implementation (cf. 2.3.3).

```
!Object methodsFor: 'error handling'!
doesNotUnderstand: aMessage
     | status gripe |
     status ← self class tryCopyingCodeFor: aMessage selector.
```

```
status==#OK ifTrue:
    [↑ self perform: aMessage selector withArguments: aMessage arguments].
....
NotivierView
        openContext: thisContext
        label: gripe , aMessage selector
        contents: thisContext shortStack.
self perform: aMessage selector withArguments: aMessage arguments.!
```

Independently of the metaclass use, the analysis of doesNotUnderstand: reveals the intrinsic reflective character of Smalltalk-80. The computational environment (thisContext), and the message itself (aMessage) are represented as true objects explicitly defined by their classes, respectivly BlockContext and Message.

tryCopyingCodeFor: The default tryCopyingCodeFor: implemented in Behavior allows the treatment of "compound selectors". Compound selectors were added by Borning and Ingalls to support conflicting methods in the case of multiple inheritance (see later 2.3.4). The principle is to explicitly indicate the address of a method by using a selector whose prefix is a class and whose suffix is a selector (for instance Behavior.new or Point.+). The method of the given class will be copied down in the receiver's class and then applied :

```
!Behavior methodsFor: 'creating method dictionary'!
tryCopyingCodeFor: selector
        | classPart whichClass simpleSelector descr |
        selector isCompound ifFalse: [ ↑ #NotFound].
        classPart ←  selector classPart.
        simpleSelector ←  selector selectorPart.
        .....
        descr ←  (Smalltalk at: classPart) methodDescriptionAt: simpleSelector.
        self compileUnchecked: classPart , '.' , descr sourceCode.
        self insertClass: self selector: simpleSelector in: SelectorOfDirectedMethods.
        ↑ #OK!
```

2.3 Programming by Metaclasses

"The primary role of a metaclass in the Smalltalk-80 system is to provide the protocol for initializing class variables and for creating initialized instances of the metaclass' sole instance [17].

The goal of this section is to demonstrate that metaclasses are powerful enough to support other tasks than "initialization" roles.

2.3.1 The Counter example

This example illustrates how to act on a class behavior by defining methods at the metaclass level. Those methods are traditionally called class methods to indicate that they define the behavior of the class itself, contrary to instance methods which define the behavior of the class instances.

The class Counter inherits from Object, represents the structure of a specific Counter by its instance variable value. Counter also describes its behavior by two sets of instance methods organized in two protocols. The first set gives access to the state of a Counter: read returns the value of the instance variable while write: modifies this value with the new one received as argument. The second set describes the incr, decr and raz methods, each of them using one (or two) of the previous accessors :

```
Object subclass: #Counter
        instanceVariableNames: 'value '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Example1'!

!Counter methodsFor: 'standard operations'!
decr: anInteger
        self write: self read  - anInteger!
incr: anInteger
        self write: self read  + anInteger!
raz
        self write: 0! !
!Counter methodsFor: 'private'!
read
        ↑ value
write: anInteger
        value ←  anInteger! !
```

The metaclass `Counter class` was created implicitly with its sole instance `Counter`. By default both its instance variable and methodDictionary are empty. Consequently `new` sent to `Counter` is treated by looking first in `Object class`, then in `Class`, `ClassDescription` and finally in `Behavior`. As should be viewed by the inspector (1), `new` allocates a new Counter, whose instance-variable `value` is bound to `nil`. Before incrementing a new allocated Counter (4), the user has to initialize it explicitly (2,3) :

```
"1" Counter new inspect.
"2" aCounter1←  Counter new.
"3" aCounter1 write: 0.
"4" aCounter1 incr: 3.
```

Defining explicit methods at the `Counter class` level allows to compose the allocation and initialization processes. To initialize explicitly instance variables, Smalltalk programmers use `class methods`. For instance "`Point x: 2 y: 3` allocates a new `Point` whose instance variables x and y are respectively initialized to 2 and 3. Similarly, the method `value:` composes the standard allocator (`self new`) with the method `write:`. The following definition of `Counter class` handles equally the example method creating and inspecting a prototype of Counter initialized to 5 :

```
Counter class
      instanceVariableNames: ''!

!Counter class methodsFor: 'initialization'!
value: anInteger
       ↑ self new write: anInteger ! !
!Counter class methodsFor: 'example'!
example
      "Counter example"
      (self value: 5) inspect! !
```

2.3.2 The Generic Stack example

This example illustrates how to act on a class structure by defining instance variables at the metaclass level. There is a lack of terminology to denote these variables and a regrettable confusion introduced by the non-symetrical definition of class methods and class variables. The unfortunate choice of using the `class variables` terminology to express a global knowledge shared by all the instances of a same class hierarchy, kills the intuition to use a metaclass to control the structure of classes by adding instance variables.

The idea is to parameterize a Stack at the metaclass level by introducing an "instantiable" type allowing to simulate generic classes. `Stack` will be a generic class generating (by using the message `newType:` Stack of Integers, Stack of Points, Stack of Stacks ... :

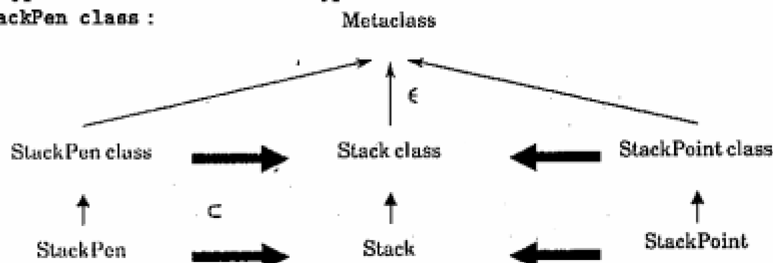
```
OrderedCollection variableSubclass: #Stack
      instanceVariableNames: ''
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Generic-classes'!

!Stack methodsFor: 'private'!
pop
      ↑ super removeFirst!
push: anObject
      self class type == anObject class
            ifTrue: [super addFirst: anObject]
            ifFalse: [self halt: 'wrong type']! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!
Stack class
      instanceVariableNames: 'type '!
!Stack class methodsFor: 'initialize'!
initialize
 self type: Object! !
!Stack class methodsFor: 'private'!
type
      ↑ type!
type: aClass
      type ←  aClass! !
!Stack class methodsFor: 'newType:'!
newType: aClass
      "Stack newType: #Point"
```

```
        | className classValue |
        className ←  (self name , aClass printString) asSymbol.
        self
             subclass: className
             instanceVariableNames: ''
             classVariableNames: ''
             poolDictionaries: ''
             category: self category.
        classValue ←  Smalltalk at: className.
        classValue class compile: 'initialize
self type: ' , aClass printString classified: #initialize.
        classValue initialize.
        ↑ className! !

Stack initialize!
```

To make Stack generic, Stack class needs a type instance variable. The methods type, type: and initialize allow respectively to read, write and initialize this instance variable.

Stack inherits from OrderedCollection, it renames the removeFirst and implements a new push: method. Before pushing a new element, the system checks its "class compatibility" with the type's value. Because the type of Stack is automatically initialized to Object, an instance of Stack will push only instances of Object, rejecting every kind of other objects.

The next step is to create new types of Stack, each of them inheriting from the previous Stack . The messages "Stack newType: Point" and "Stack newType: Pen" create the four classes: StackPoint, StackPoint class, StackPen and StackPen class :



StackPoint class inherits type from Stack class while StackPoint inherits push: and pop from Stack.

2.3.3 Adding PartWhole Hierarchy à la Thinglab

This third example generalizes the metaclass use. We act simultaneously on the structure and behavior of a set of classes defined within a same hierarchy. Our goal is to extend Smalltalk classes to handle the Thinglab part hierarchies. The principle is to generalize the instance variable concept by the partwhole concept implementing composite objects. A part is an instance variable whose default value is an instance of a given class.

For instance, we will define a Line as the composition of two parts: start and end, each of them constrained to be an instance Point. Similarly a Triangle will be implemented as the composition of three lines :

```
Thinglab subclass: #Line
      instanceVariableNames: ''
      partwhole: 'start Point end Point'
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Thinglab-Demo'
```

```
Thinglab subclass: #Triangle
      instanceVariableNames: ''
      partwhole: 'l1 Line l2 Line l3 Line'
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Thinglab-Demo'
```

When instantiating Line, the method new will instantiate automatically and recursively each part. Consequently l1 will be a Line whose start and end variables will be some Points.

To give access to the structure of the composite object, two accessors method (read & write) are automatically created for each part. Because we have chosen to use the same symbol to denote both the instance variable and the instance method, the methods start, start:, end, end: are generated in order to give access to the start and end parts of a given Line.

The composition of accessors-methods allows to read-or-write any piece of the part hierachies. For instance, if aTriangle is instantiated from Triangle, the composition: "aTriangle line1 start x" returns the value of the part x of the part start of the part line1. Nevertheless, as discussed by [2], the object receiving the composition of messages is not the object producing the answer. To give the control to the composite object itself, we re-use the

"compound selector" idea. The principle is to express at the selector level, the explicit path giving access to the part of the composite structure. For instance the compound message: "aTriangle line1.start.x: 404" will modify the x part of the start part of the line1 part BUT under the control of aTriangle. The partwhole implementation requires the following steps :

1. extending the class structure by adding the slot parts and providing the associated accessor methods,
2. specializing the class behavior, and precisely the new allocator which must now automatically instantiate the part hierarchies,
3. modifying the instance behavior in order to accept compound messages and recognize them as a path of parts.

Extending the standard structure of classes The structure and behavior of a Thinglab class is described by the metaclass Thinglab class. Because Smalltalk classes and metaclasses are mutually dependent, the Thinglab hierarchy must be "grafted" at the Object class level. The inheritance rules of metaclasses and the fact that Thinglab class must be a subclass-of Object class constrains Thinglab to be a subclass-of Object :

```
Object ()
     Thinglab ()
          Line (start end)
          Triangle (line1 line2 line3)
               IsoceleTriangle ()
     Behavior (superclass methodDict format subclasses)
          ClassDescription (instanceVariables organization)
               Metaclass (thisClass )
               Class (name classPool sharedPools)
                    Object class ()
                         Thinglab class (parts)
                              Line class ()
                              Triangle class ()
                                   IsoceleTriangle class ()
```

A Thinglab class (for instance Line or Triangle) will be defined as an undirect subclass of Thinglab, thus its metaclass will be an undirect subclass of Thinglab class. In fact the class Thinglab is (almost) empty and is used only to develop the Thinglab hierarchy. The effective implementation of partwhole hierarchies is described at the metaclass level :

```
Object subclass: #Thinglab
     instanceVariableNames: ''
     classVariableNames: ''
     poolDictionaries: ''
     category: 'ThingLab'!

!Thinglab methodsFor: 'initialization'!
partsInitialize
      ↑ nil! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!
Thinglab class
     instanceVariableNames: 'parts '!

!Thinglab class methodsFor: 'private'!
parts
      ↑ parts!
parts: stringOfParts
     parts ← stringOfParts! !
```

Each Thinglab class owns the instance variable parts inherited from Thinglab class and recognizes the two associated "accessor methods". To treat the partwhole declaration, a new method for Thinglab "sub-class" creation must be provided by Thinglab class :

```
!Thinglab class methodsFor: 'subclass creation'!
subclass: t instanceVariableNames: f partwhole: p classVariableNames: d
          poolDictionaries: s category: cat
     "Scanner new scanFieldNames: 'p1 Point p2 Point'
          --> #('p1' 'Point' 'p2' 'Point' )"
     | scannedParts stringOfParts aMethodString arrayOfParts j aThinglabClass |
     scannedParts ← Scanner new scanFieldNames: p.
```

```
        stringOfParts ←  ''. j ←  1.
        arrayOfParts ←  Array new: scannedParts size // 2.
        aMethodString ←  'partsInitialize \' withCRs.
        (1 to: scannedParts size by: 2)
              do:
                    [:i |
                    stringOfParts ←  stringOfParts , (scannedParts at: i) , ' '.
                    aMethodString ←  aMethodString , (scannedParts at: i) , '←  ' ,
    (scannedParts at: i + 1) , ' new.\' withCRs.
                    arrayOfParts at: j put: (scannedParts at: i).
                    j ←  j + 1].
        aMethodString ←  aMethodString , '\ super partsInitialize' withCRs.
        aThinglabClass← ((super
              name: t
              inEnvironment: Smalltalk
              subclassOf: self
              instanceVariableNames: f , stringOfParts
              variable: false
              words: true
              pointers: true
              classVariableNames: d
              poolDictionaries: s
              category: cat
              comment: nil
              changed: false)
              parts: p).
        aThinglabClass makePartsAccessors: arrayOfParts.
        aThinglabClass compile: aMethodString classified: #initialization.
        ↑ aThinglabClass!
    makePartsAccessors: arrayOfParts
        | aStringR aStringW |
        arrayOfParts == nil ifTrue: [↑ self].
        arrayOfParts do:
              [:iv |
              aStringR ←  (iv , '\ ↑ ' , iv) withCRs.
              aStringW ←  (iv , ': value \' , iv , ' ←  value') withCRs.
              self compile: aStringR classified: #private.
              self compile: aStringW classified: #private]! !
```

`subclass:...:partwhole:...:category:` generalizes the standard `subclass:...:category:` held by `Class` by analysing the partwhole declarations. The couple (part,class) are scanned to :

1. concatenate the string of parts to the string of instance variables,
2. compile for each part its two accessor methods (this task is done by `makePartAccessors` (cf. p 289 of [17])),
3. compile a `partsInitialize` method instantiating each part to an instance of its associated class,

Looking at the `Line` definition, `start` and `end` will be added to the set of instance variables. The methods whose definitions follow will automatically be created :

```
!Line methodsFor: 'private'!
start
        ↑ start
start: value
        start ←  value! !
!Line methodsFor: 'initialization'!
partsInitialize
        start ←  Point new.
        end ←  Point new.
        super partsInitialize! !
```

Extending the standard behavior of classes To realize the automatic instantiation of part hierarchies we have to compose the basic allocator (`super new`) with the initialization of parts (`partsInitialize`) :

```
!Thinglab class methodsFor: 'creating'!
new
        ↑ super new partsInitialize! !
```

Implementing access to parts Accesses to parts use compound selectors expliciting the path. The associated methods are compiled at the first access following the two transformations:

```
compositeObject part1.restOfPath →  compositeObject part1 restOfPath
compositeObject part1.restOfPath: arg1 →  compositeObject part1 restOfPath: arg1
```

For instance the first transmission: `aTriangle line1.start.x: 404` generates the method `line1.start.x:` :

```
!Triangle methodsFor: 'messagesToParts'!
line1.start.x: arg1
      self line1 start.x: arg1! !
```

which can in its turn generate the `start.x:` method ... Receiving for the first time the compound selector provides an error which is trapped by a redefinition at the metaclass level of `tryCopyingCodeFor:`. This method proceeds the path transformation, compiles recursively the "missing" methods and perfoms again the previous message.

```
!Thinglab class methodsFor: 'error-handling'!
tryCopyingCodeFor: selector
      | path selector1 args body |
      selector isCompound ifFalse: [↑ #NotFound].
      path ←  selector classPart.
      selector1 ←  selector selectorPart.
      args ←  ''. body ←  '\ ↑ ' withCRs.
      (selector1 includes: $:)
            ifTrue:
                  [selector1 ←  selector1 , ' arg1'. args ←  ' arg1' .
                   body ←  '\ 'withCRs].
      self compile: selector , args , body , 'self ' , path , ' ' , selector1
           classified: #messagesToParts.
      ↑ #OK!
```

2.3.4 Adding Multiple Inheritance

The changes required to add multiple inheritance to Smalltalk 80 are only a few pages of Smalltalk code. ... There are few other programming environments in wich such a fundamental extension could be made easily [7].

As pointed by Borning, Smalltalk-80 was open-ended enough to support an extension realizing multiple inheritance for classes. This extension was supported by the metaclass architecture without stepping out of the language. We have not enough room to give the full implementation in details but we can indicate the general scheme.
From the structural point of view, a new kind of metaclass was added defined by `MetaclassForMultipleInheritance` :

```
Object ()
      Behavior (superclass methodDict format subclasses)
            ClassDescription (instanceVariables organization)
                  Metaclass (thisClass )
                        MetaclassForMultipleInheritance (otherSuperclasses)
```

Metaclasses allowing multiple inheritance for their instance inherit from standard metaclasses: they memorize the first of their superclasses (or supers) within the inherited `superclass` and the rest within the new `otherSuperclasses`.
Notice that classes using multiple supers have an undirect knowledge of their supers via the metaclass indirection. For instance the `Digimeter` class holds `Instrument` in its superclass, while `LCD` is held by `Digimeter class` in its `otherSupers`.

```
Class subclass: #DigiMeter'
      superclasses: 'Instrument LCD'
      instanceVariableNames 'x y'
      classVariableNames: ''
      category: 'Multiple Inheritance'
```

From the behavioral point of view the `subclass:superclasses:...:category:` method held by `Class class` has to call the new `name:...:and:...:changed` method which is a specialization of the standard `name:...:changed:` method held by `Metaclass`. Conflicting methods are detected by the system, the user having to solve explicitly conflicts by using compound selectors.

Nevertheless this extension towards multiple inheritance complicates the instantiation tree (by adding 2 levels) and does not operate in a uniform way. The creation of a class using multiple inheritance does not follow the general template: the message is sent to `Class` and not yet to the "super". There is a dissymetry between the first super (known by the class) and the other supers (known by the metaclass). Finally multiple inheritance of instance variables is not well defined at the metaclass level.

3 ObjVlisp

ObjVlisp was built to understand the metaclass concept and to provide an alternative to the Smalltalk solution. In fact we agreed with Borning about the complexity of the Smalltalk metaclasses but rather than rejecting them [9] we preferred to explicit them fully as true classes.

3.1 The ObjVlisp Postulates

The ObjVlisp postulates were already published in [10] and [12]. We detail here the postulates which differ from Smalltalk.

1. ObjVlisp unifies the class and metaclass levels. A metaclass is a true class inheriting its structure and behavior from a previous metaclass. Consequently, the one to one relation, between a Smalltalk metaclass and its private instance is broken: a metaclass can own potentially an infinite set of instances and several classes can share the same metaclass (cf. 3.3.3).
2. Because ObjVlisp implements multiple inheritance for classes, multiple inheritance for metaclasses is obtained for free.
3. The depth of the instantiation tree is potentially infinite and the regression of metaclasses solved by the self-instantiation of the first metaclass: `Class`.
4. To clarify the `class variable` terminology, ObjVlisp implements shared environments by using the instance variables of the metaclasses. These variables are accessible both at the class and at the instance levels (cf. 3.3.2) but are shared only by the instances of a same class (and not by the instances of a same class hierarchy).

3.2 The ObjVlisp architecture

The basic architecture is supported by only two classes: `Class` the root of the instantiation tree and `Object` the root of the inheritance lattice. The first class `Class` describes the sole `make-instance` method (Smalltalk `new`) which composes the `allocate-instance` method owned by `Class` and the `initialize-instance` methods owned both by `Class` and `Object`. Because `Class` is its own instance, the values of `Class` instance variables have to match the structure of `Class` itself including its list of instance variables.

To expose a flavor of the ObjVlisp bootstrap we give below the circular definitions of `Object` and `Class` :

a) `Class` skeleton is built manually with its slots and the three methods `allocate-instance`, `initialize-instance` and `make-instance` :

```
(setq Class
     '(Class Class (Object)
        (isit name supers i-v methods)
        (make-instance (λ ..) allocate-instance (λ ..) initialize-instance (λ ..)) ) )
```

b) `Object` is created by instantiation of the `Class` skeleton :

```
(send Class 'make-instance
      :name       'Object
      :supers     '()
      :i-v        '(isit)
      :methods    '(class-of             (λ () isit)
                     initialize-instance  (λ (initargs) (initialize-slots self ...) self)
) )
```

c) The real object `Class` is created by self-instantiation of the `Class` skeleton (which is destroyed) :

```
(send Class 'make-instance
      :name       'Class
      :supers     '(Object)
```

```
        :i-v         '(name supers i-v methods)
        :methods     '(make-instance (λ (&rest initargs)
                                       (send (send self 'allocate-instance)
                                             'initialize-instance initargs))
                       allocate-instance    (λ () (make-instance name))
                       initialize-instance (λ (initargs) (run-super) ...) ) )
```

3.3 Programming by Metaclasses with ObjVlisp

3.3.1 The Counter example

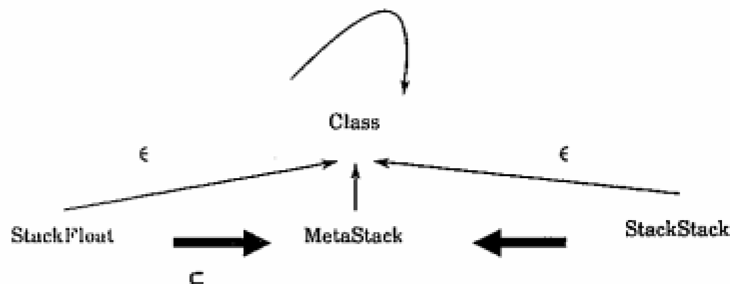
This first example shows the ObjVlisp version of the Counter. Contrary to Smalltalk the metaclass `CounterClass` is explicitly created as an instance and as a subclass of `Class` and BEFORE creating the class `Counter`. Then `Counter` itself is created as an instance of `CounterClass` :

```
(send Class 'new
      :name      'CounterClass
      :supers    '(Class)
      :methods   '(value: (λ (anInteger) (send (send self 'new) 'write: anInteger))) )

(send CounterClass 'new
      :name      'Counter
      :supers    '(Object)
      :i-v       '(value)
      :methods   '(write: (λ (anInteger) (setq value anInteger) self)
                   read   (λ () value)
                   raz    (λ () (send self 'write: 0))
                   incr:  (λ (incr) (send self 'write: (+ (send self 'read) incr)))
                   decr:  (λ (decr) (send self 'write: (- (send self 'read) decr))) ))

(send Counter 'new value: 3)
```

We observe that at the syntactical level, the creation of a metaclass, a class or a terminal instance is the same. In all cases we send the message new to the class and we use keywords (whose names are derived from the instance variable one) to express the initial value of the instance variables.

3.3.2 The Stack example

This second example gives the ObjVlisp solution to the "Generic Stack problem". We use here the possibility to share the same metaclass `MetaStack` between all classes representing different kinds of stacks: `StackFloat`, `StackStack` ... :



At the code level we can observe two other differences: the value of the instance variable type of `MetaStack` is directly accessible from the instance method push: of class `Stack`. At each class creation type is explicitly initialized :

```
(send Class 'new
      :name      'MetaStack
      :supers    '(Class)
      :i-v       '(type)
      :methods   '(type   (λ () type)
                   type:  (λ (aType) (setq type aType)) ))
```

```
(send MetaStack  'new
       :name      'Stack
       :supers    '(Object)
       :i-v       '(aList)
       :methods   '(initialize-instance
                              (λ () (setq aList '(**bottom**)))
                   pop        (λ () (prog1 (car aList) (setq aList (cdr aList))))
                   addFirst: (λ (qq) (setq aList (cons qq aList)))
                   push:      (λ (entity) (when (eq (type-of entity) type)
                                    (send self 'addFirst: entity))) )
       :type     'Object)

(send MetaStack 'new :name 'StackFloat :supers '(Stack) :type 'Float)
(send MetaStack 'new :name 'StackStack :supers '(Stack) :type 'Stack)
```

4 The CLOS Postulates and metaclass Architecture

"The metaobject kernel of the CommonLisp Object System (CLOS) comprises the classes and methods that define and implement the behavior of the system. Since CLOS is an object-oriented program itself, exposing this kernel allows people to make useful integrated extensions to CLOS without changing the behavior of the system for ordinary programs, and without unwarranted loss of efficiency [6]".

CLOS [14] is the object-oriented paradigm for CommonLisp merging the features of CommonLoops [5] and NewFlavors [22]. This paradigm was designed by Xerox, Symbolics, Lucid, HP and TI people in order to provide a unified object system for the CommonLisp world. CLOS was accepted as the standard for CommonLisp by the Ansi X3J13 committee. [26] gives a complete description of the CLOS specification but let us recall the more significant features :

- unified syntax derived from CommonLisp,
- multiple inheritance scheme using a linearization algorithm,
- generic functions generalizing function call and message passing,
- method combination,
- class updating allowing class redefinition and instance updates,
- integration of the class system into CommonLisp type system
- a meta object protocol.

Obviously, from a reflective point of view, the last item is the most important. It means that the structure and behavior of entities such as `methods`, `slots`, `generic functions`, `method combinations`, `types` and `structures` are described at the language level by pre-defined classes. In fact the CLOS metaclass architecture is very close to that of ObjVlisp and starts with the `standard-class` of which almost all classes in the kernel are instances, including `standard-class` itself. [11], [19], [1] discuss how to modify the ObjVlisp metaclass kernel in order to support an implementation of the CLOS metaobject protocol.

5 Conclusion

The abstract of this paper represents what we have planned to write. We have not developed the Pattie Maes approach aiming at describing computational reflection for the object-oriented world [21]. In fact the topic was too vast and we only developed the metaclass architecture provided by Smalltalk. We were surprised by its potentiality and we believe that Smalltalk metaclasses deserve to be better known. We were really surprised by the harmony of the Smalltalk interpreter, and convinced that the reification process allowing to manipulate the context, the stack and the message at the language level is inherent to the Smalltalk system.

We have demonstrated the Smalltalk ability for automatic code generation but we have not developed the expression of control structures in Smalltalk itself by using the block construction.

In the field of reflection, we are convinced of the relationship between Lisp (Scheme) and Smalltalk implementations technics and we suggest to realize a 3-Smalltalk as the Smalltalk equivalent to 3-LISP.

We have quickly reviewed the ObjVlisp and CLOS metaclasses, but contrary to the Smalltalk community, ObjVlisp and CLOS programmers are convinced of the necessity to use metaclasses.

Finally, we could observe that languages such as Eiffel, C++ and Beta which are not providing the manipulation of the abstraction lose the benefit of the reflective process as developed by the Lisp (Scheme) and Smalltalk traditions.

References

[1] Attardi, G., Boscotrecase, M.R., Diomedi, S., DC-LOS: an integration of CLOS and the ObjVlisp model, *Technical Note MADS TN-87.3 (ESPRIT Project P440)*, Viareggio, Italy, December 87

[2] Blake. E., Cook. S., On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk, *ECOOP'87 proceedings*, Springer Verlag: Lecture Notes in Computer Science, V 276, pp 41-50, Paris, France, June 1987

[3] Bobrow, D.G., Stefik, M., The LOOPS Manual, Xerox PARC, Palo Alto CA, USA, December 1983.

[4] Bobrow, D.G., Stefik, M., Object-Oriented Programming: Themes and Variations, *The AI Magazine*, pp 40-62, Winter 85.

[5] Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., Zdybel, F., CommonLoops: Merging Lisp and Object-Oriented Programming, *OOPSLA'86*, Special Issue of SIGPLAN Notices, Vol. 21, No 11, pp. 17-29, Portland OR, USA, November 1986.

[6] Bobrow, D.G., Kiczales G., The Common Lisp Object System Metaobject Kernel A Status Report, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pp 309-315, SnowBird Utah, USA, July 1988.

[7] Borning A. H., Ingalls D. H., Multiple Inheritance in Smalltalk-80 *Proceedings of the AAAI-82* pp. 234-237 Pittsburgh, USA, August 1982.

[8] Borning A. H., Classes versus Prototypes in Object-Oriented Languages *Proceedings of the IEEE Fall joint conference* pp. 36-40 Dallas, TX, USA, November 1986

[9] Borning A. H., Deltatalk: An Empirically an Aesthetically Motivated Simplification of the Smalltalk-80 Language *ECOOP'87 proceedings*, Springer Verlag: Lecture Notes in Computer Science, V 276, pp 1-11, Paris, France, June 1987

[10] Briot, J.P., Cointe, P., A Uniform Model for Object-Oriented Languages Using The Class Abstraction *IJCAI'87*, Volume 1, pp 40-43 Milano, ITALIA, August 23-28 87

[11] Cointe, P., Towards the design of a CLOS Metaobject Kernel: ObjVlisp as a first layer *First International Workshop on LISP Evolution and Standardization (IWoLES 88)*, pp 33-40 Afcet, Afnor, LITP and Inria, Paris, France, February 1988

[12] Cointe, P., Metaclasses are First Class: the ObjVlisp model, *OOPSLA'87*, Special Issue of SIGPLAN Notices, Vol. 22, No 12, pp. 156-167, Orlando, Floride, USA October 87

[13] Dahl, O., Myhrhaug, B., Nygaard, K., Simula-67 Common Base Language, *SIMULA information, S-22*, Norvegian Computing Center, Oslo, Norway, October 1970.

[14] DeMichiel L.G., Gabriel. R.P., The Common Lisp Object System: An Overview, *ECOOP'87 proceedings*, Springer Verlag: Lecture Notes in Computer Science, V 276, pp 151-170, Paris, France, June 1987

[15] DesRivieres, J.C., Smith B.C, The Implementation of Procedurally Reflective Languages, *Conference record of the third ACM Lisp and Functional Programming conference*, pp 331-347, Austin Texas, USA, July 1984.

[16] Ferber J., Briot J.P, Design of a Concurrent Language for Distributed Artificial Intelligence *FGCS'88 proceedings*, Tokyo, Japan, December 1988

[17] Goldberg, A., Robson, D., Smalltalk-80 - The Language and its Implementation, Addison-Wesley, Reading MA, USA, 1983.

[18] ParcPlace Systems, Smalltalk-80 sources

[19] Graube, N., Reflexive architecture: from ObjVlisp to CLOS *Ecoop-88 proceedings*, Lecture Notes in Computer Science, Springer Verlag, pp 110-127 Gjessing. S.,Nygaard. K. Editors, Oslo, NORWAY, August 88.

[20] Ingalls D. H., The Smalltalk-76 Programming System Design and Implementation, *Conference record of the fifth ACM symposyum on Principle of Programming Languages*, Tucson, USA, January 23-25, 1978.

[21] Maes, P., Concepts and Experiments in Computational Reflection, *OOPSLA'87*, Special Issue of SIGPLAN Notices, Vol. 22, No 12, pp. 147-155, Orlando, Floride, USA October 87

[22] Moon, D., Object-Oriented Programming with Flavors, *OOPSLA'86*, Special Issue of SIGPLAN Notices, Vol. 21, No 11, pp. 1-16, Portland OR, USA, November 1986.

[23] Queinnec, C., Cointe P., An Open Ended Data Representation Model for Eu-Lisp *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pp 298-308 SnowBird Utah, USA, July 1988.

[24] Van Marcke K., The Use and Implementationof the Representation Language KRS *Ph Thesis, Vrije Universiteit Brussel* April 88

[25] Watanabe T., Yonezawa A. Reflection in an Object-Oriented Concurrent Language OOPSLA'88 San Diego, California, USA, 25-29 September 89

[26] X3J13, Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S.E., Kickzales, G. and Moon, D.A., *X3J13 standards committee documents 88-002 ans 88-003*, March 15, 88