Implementing the ObjVlisp model in Smalltalk-80 or making the Smalltalk-80 metaclasses explicit

Jean-Pierre Briot Pierre Cointe

Equipe Mixte Rank Xerox France - LITP,
Université Pierre et Marie Curie,
4 place Jussieu, 75005 Paris, France

Extension of existing language, Theory, Design, Methodology
briot/pc@rxf.univ-p6-7.fr.uucp

Abstract

This paper deals with an implementation of the ObjVlisp model in the Smalltalk-80 language.

We demonstrate that Smalltalk is extensible and flexible enough to support simultaneoulsy its standard metaclass architecture plus the ObjVlisp one.

We demonstrate also that the ObjVlisp model was enough general to be transposed directly in the Smalltalk world.

The result of this cross-fertilization is the "ObjVlisp" browser which allow to build and experiment a new pedagogical discourse in the field of class-oriented languages by making explicit the metaclass level. We develop this discourse as an answer to the Borning/O'Shea question about Smalltalk learnability.

To enhance this discourse we propose a first library of metaclasses which can be composed together to build a multi-levels instantiation hierarchy. Because metaclasses are true classes they can be also composed by using the multiple inheritance scheme provided by Ingalls and Borning for simple classes.

Finaly we introduce new problems such as which ergonomy for a (meta)class browser, and inheritance compatibility between classes which do not share the same metaclasses.

Keywords

Object, Class, Metaclass, ObjVlisp, Smalltalk-80, CLOS, kernel, uniformity, integration, compatibility, explicit, implicit flexibility, learnability,

1 Introduction

Uniformity is one of the main advantages of Object-Oriented Programming (OOP) [Goldberg and Robson 83]. Therefore in the sub-field of "class-oriented" languages, classes must be considered as first class objects [Cointe 88] and described by true and appropriate classes, called *metaclasses* [Briot and Cointe 87] [Bobrow and Kickzales 88].

The metaclass architecture provided in Smalltalk-80 voluntarily hides the metaclasses level from the user. A metaclass is automatically created by the system when the user defines a new class. This metaclass is not explicitly named and does not appear explicitly in the class editor (browser). If the programmer wants to understand the system, he needs to "go behind the browser" and discover the complexity of the "kernel classes".

The architecture he finds is complex, arbitrary and fixed. Consequently some people working in the field of "teaching object oriented programming" advocated that the presence of metaclasses complexifies unecessary the model and that they should be removed or *highlighted* [Borning and Oshea 87].

We do agree with the analysis but we disagree with the diagnosis and their remove solution. We have already advocated the needs for explicit metaclasses and we proposed in [Cointe 87] the ObjVlisp model which supports a simple, clean and minimal architecture for explicit metaclasses. The Common Lisp Object System (CLOS) has also been designed along such an architecture even if the description of slots and methods as objects extends and complexifies this model [Graube 88].

In this paper we demonstrate how to implement the ObjVlisp architecture in Smalltalk-80. One could wonder why such a proposal and about our motivation. This work is motivated by mainly two reasons :

- First, Smalltalk-80 is currently the most complete and flexible object-oriented programing system. A sophisticated environment is fully integrated into the system. Consequently Smalltalk-80 is one of the best current tools to prototype programming languages, systems and applications.

  On the other hand, ObjVlisp provides a simpler and cleaner model but there is no environment and no libraries. CLOS is a much richer model but there are currently few implementations and its programming environment is still under work and make the language still unusable.

  Some researchers in the field of knowledge representation are looking for a system like Smalltalk-80 but with some more explicit metaclass architecture in order to plug new mechanisms. We provide exactly this by integrating the ObjVlisp architecture into Smalltalk-80. The integration is complete, consequently one may reuse and combine the two styles of metaclass programming at once. Obvioulsy, the standard Smalltalk-80 environment appears not perfectly suited to support this new methodology. In order to support the explicit programming of metaclasses our second goal was to extend this standard environment to provide a browser for ObjVlisp (meta)classes.

- Second, the raw comparison we conducted between the Smalltalk-80 architecture and the ObjVlisp architecture was not perfectly fair, because the goals of the two systems are not equal. By integrating the ObjVlisp architecture into the Smalltalk-80 environment, we could reevaluate our comparative analysis of both architectures. Because the inheritance of metaclasses becomes explicit, some metaclass compability problems may appear. Also we could remark in the complete example of "typed stacks" which will be described that the programmer needs to elaborate the metaclasses needed from his low level wishes, i.e., form the final objects (bottom end instances) that he wishes to describe. Because of the class abstraction metaphor, the programmer needs to define classes *before* creating its concrete instances. When using extensively metaclasses on several levels, this leads the programmer to go up progressively several levels to define the `root` of his application architecture before going down by instantiating progressively down to bottom end instances. This unfortunately breaks down a little bit the interactive and incremental programming philosophy of Smalltalk-80.

The first section of this paper analyses the Smalltalk choice of strongly coupling the class and metaclass level, the second section deals with two altenative implementations (merging and grafting) of the ObjVlisp model, the third part develop the ObjVlisp browser, the fourth part defines a first libraries of metaclasses, the fifth part investigates how to compose these metaclasses by (multiple) inheritance and the last part discusses metaclasses as reusable components for software engeenering.

2 Power and Weaknesses of Smalltalk-80 Metaclasses

2.1 The Smalltalk Design

One of the key choice of the Smalltalk environment is to exhibit the metaclass level only when necessary. This choice has led to “simplify” the metaclass architecture by making the following assumptions :

1. when a class is explicitly created by the user the system automatically creates a private and implicit metaclass associated to this class,
2. this metaclass is an anonymous class but with true methods and true instance variables manipulable through the browser. The “implicit choice” conducts to non-sharable metaclasses and go against the general purpose of (meta)classes : to be instantiated as many time as needed.
3. because metaclasses are implicitly created, their inheritance rule obey to a default: “if a class is a subclass of another one, its metaclass is a subclass of the metaclass of the another one”.

These choices aimed at strongly coupling each class with a private metaclass.

At the implementation level the “coupling choice” has lead to distinguish between classes and metaclasses. The structure and behavior of a class is defined by `Class` while the structure and behavior of a metaclass is defined by `Metaclass`.

Classes `ClassDescription` and `Behavior` are introduced to explicit the common structure and behavior of both classes and metaclasses.

All metaclasses inherit from `Object class` which is the root of the metaclass hierarchy. `Object class` is itself a direct subclass of `Class`. Finaly Smallalk-80 proposes the following inheritance hierarchy for (meta)classes :

```
Object
~~~~~~Behavior
~~~~~~~~~ClassDescription
~~~~~~~~~~~~Class
~~~~~~~~~~~~Metaclass
~~~~~~~~~~~~~~~Object class ()
~~~~~~~~~~~~~~~~~~<all metaclasses>
```

As developed in [Cointe 88] a (meta)class describes both the structure and the behavior of its instances. The structure is defined by a set of instance variables while the behavior is defined by a set of methods. The indentation used by the previous figure expresses the subclass relationship. In the tow next figures the classes prefixe either their instance-variables (using ()) either their methods (using). Obviously inheritance of instance-variables is static while the inheritance of methods is dynamic.

Structure

```
Object ()
~~~~~~Behavior (superclass methodDict format subclasses)
~~~~~~~~~ClassDescription (instanceVariables organization)
~~~~~~~~~~~~Class (name classPool sharedPool)
~~~~~~~~~~~~Metaclass (thisClass)
~~~~~~CartesienComplex (x y)
~~~~~~~~~~~~~~~Object class ()
~~~~~~~~~~~~~~~~~~Behavior class (...)

~~~~~~~~~~~~~~~~~~CartesienComplex class ()
```

Behavior

```
Object <...>
~~~~~~Behavior <new new:  ...>
~~~~~~~~~ClassDescription < ...>
~~~~~~~~~~~~Class <...>
~~~~~~~~~~~~Metaclass <new  ...>
~~~~~~CartesienComplex <setX:setY: ...>
~~~~~~~~~~~~~~~Object class <...>
~~~~~~~~~~~~~~~~~~Behavior class <...>

~~~~~~~~~~~~~~~~~~CartesienComplex class <x:y: ...>
```

2.2 Object Allocation

As explained in [Cointe 88] the standard allocator `new` which is defined in `Behavior` is redefined in `Metaclass` in order to constrain a metaclass to own one and only one class. The "backward" pointer between the metaclass and its sole instance beeing realized by the instance-variable `thisClass` : [1]

The main comment about this architecture is that Smalltalk provides a unified allocator for its differents kind of objects (instances, classes and metaclasses) but does not support an explicit initialization method à la CLOS and à la ObjVlisp.

On one hand, simple (terminal) objects are created with the default-value "nil" for each instance variables. On the other hand, classes (abstract objects) must be created with explicit values in order to fill a common template. The consequence is a non-uniform syntaxe to create classes and instances. Terminal instances are created by sending the message `new` to their classes while classes are defined by using the "mysterious" *subclass:instanceVariablesNames:... :category* selector which is used by the browser to create both a class and its private metaclass. In fact this selector call twice the basic allocators `new`, one to allocate the metaclass, one to allocate the class.

2.3 Controling objects initialization

The advantage of coupling a class with its private metaclass is to make easier the definition of an explicit method to compose the creation and the explicit initialization for instances. The principe is to specialize the standard new by defining a private (class)method hold by the metaclass.

For example if the class `CartesienComplex` uses the two instance-variables `x` and `y` to define the structure of its instances :

1. an instance method (`setX:setY:`) is defined to initialize (write) these instances variables
2. a class method (`x:y:`) is defined to compose the standard allocator with the initializator,
3. an instance is explicitly created and initialized by sending the class method to the class (`CartesianComplex x: 1 y: 0`) :

```
Object ()
~~~~~~CartesienComplex (x y)
```

[1]

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

```
~~~~~~~~~Object class ()
~~~~~~~~~~~~CartesienComplex class ()

Object <...>
~~~~~~CartesienComplex <setX:setY:>

~~~~~~~~~Object class <...>
~~~~~~~~~~~~CartesienComplex class <x:y:>
```

```
!CartesianComplex  methodsFor: 'private'!
setX: aX setY: aY
      x - aX . y - aY

!CartesianComplex class methodsFor: 'creation'!
x: aX y: aY
      ^ super new setX: aX setY aY
```

If the rule of "parallel inheritance" between classes and metaclasses works pretty well to control the initialization process, this same rule can conduct to unacceptable (unsolvable) problems when one want to act upon the allocator itself.

2.4 The Abstract Classes counter-example

The "abstract class" example shows how some natural description does not work in Smalltalk-80. The term *abstract class* is introduced to name classes which are defined by the programmer in the inheritance hierarchy to share most definitions. Such a class is not intended to own instances, only its subclasses will specialize and be able to model concrete instances.

A simple example of such kind of programming is the library we could propose for complex numbers. Two models for representation are useful for complex numbers, cartesian and polar coordinates. Therefore we need two classes, respectively `CartesianComplex` and `PolarComplex` to implement them. The abstract class `Complex` factorizes the common behaviors, for instance computing arithmetics:

```
Object ()
~~~~~~Complex ()
~~~~~~~~~CartesienComplex (x y)
~~~~~~~~~PolarComplex (ro teta)
```

To express the fact that an abstract class MAY not generate instances, we would redefine the `new` instantiation method to raise an error[2]

In order to reuse this behavior in other abstract classes, we may define a new class, named `Abstract`. For example we redefine the class `Complex` as a subclass of `Abstract`:

```
Object <...>
~~~Abstract <>
~~~~~~Complex <>
~~~~~~~~~CartesienComplex <>
~~~~~~~~~PolarComplex <>

~~~~~~~~~~~~Object class <>
~~~~~~~~~~~~~~~Abstract class <new>
~~~~~~~~~~~~~~~~~~Complex class <>
```

[2]!Abstract class methodsFor: 'creation'!
new
self error: 'no instance for abstract classes'! !

```
~~~~~~~~~~~~~~~~~~~~~CartesienComplex class <>
~~~~~~~~~~~~~~~~~~~~~PolarComplex class <>
```

This works because the "error `new`" method of `Abstract class` is inherited by `Complex class`. UNFORTUNATELY, because `CartesianComplex` and `PolarComplex` inherit from `Complex`, their metaclass will inherit the `new` method too. Consequently they become abstract classes, and it will be impossible to create any complex number!

The problem is that inheritance of metaclasses is implicit and that we control instantiation through inheritance in Smalltalk-80. We would like to control explicitely their inheritance by explicitly changing the inheritance rule of `CartesianComplex class` and `PolarComplex class` in order to shadow the "abstractness" inherited from `Complex class`. [3]

In fact the good solution to the abstract problem will be the following one :

- to create a new metaclass `AbstractClass`,
- to create a new abstract class `Complex`, instance of `AbstractClass`,
- to create two standard classes `CartesienComplex` and `PolarComplex` instances of `Class` :

We can express this solution in a Smalltalk syntax :

```
Class newName: #AbstractClass
      superclass: Class
      instanceVariableNames: ''
      category: 'Metaclass-Library'!

!AbstractClass methodsFor: 'creation'!
new
      self error: 'no instance, I am an abstract class'!


AbstractClass newName: #Complex
      superclass: Object
      instanceVariableNames: ''
      category: 'Numeric-Complex'!


Class newName: #CartesianComplex
      superclass: Complex
      instanceVariableNames: 'x y '
      category: 'Numeric-Complex'!

Class newName: #PolarComplex
      superclass: Complex
      instanceVariableNames: 'rho theta '
      category: 'Numeric-Complex'!
```

The advantages of this sheme are the following:

1. a new class is created explicitly by sending a message to a previous metaclass,
2. a new metaclass is created as a true class BUT defined as a subclass of a previous metaclass

The rest of this paper will develop how to make possible this scheme in Smalltalk-80

[3]One way to proceed is to explicitly change the inheritance link of the metaclasses by updating the associated instance variable named *superclass* :

CartesianComplex class superclass: Object class.
PolarComplex class superclass: Object class

3 Implementing the ObjVlisp Model in Smalltalk-80

The complete solution to these problems has already been advocated in previous research papers [Briot and Cointe 87] [Cointe 87] [Cointe 88] Creation of classes must be explicitly stated as instances of some other classes called metaclass. ObjVlisp is a system which proposes such an architecture. It is also minimal in the sense of beeing self-defined by only two classes which are: the root of the instantiation tree (`Class`) and the root of the inheritance tree (`Object`).

An ObjVlisp metaclass is only one class which can access to a standard allocator by using the inheritance path. `Class` as the holder of the allocator is the first metaclass, a new metaclass is created as a subclass of a previous one.

3.1 ObjVlisp & Clos vs Smalltalk

For these two systems *Class* holds the standard allocator (*basicNew*) and the *new* method which compose this allocator with the initializator. The standard initializator for terminal instances is defined in Object while the standard initializator for (meta)classes is redefined in *Class* as a specialization of the previous one :

```
Object <initialize>
~~~~~~Class <basicNew new initialize>
```

Therefore we will now discuss how to introduce the ObjVlisp model into Smalltalk-80. This introduction can be done in two different ways:

1. to have the minimal solution and to get the higher combination between both architectures, we identify (map, merge) the ObjVlisp kernel with the corresponding Smalltalk-80 classes, that is reuse Smalltalk classes `Class` and `Object`.
2. to graft ObjVlisp without reusing (and extending) `Class`, defining the first ObjVlisp metaclass (let say `ObjVlisp`) as a subclass of `ClassDescription`

3.2 Merging the ObjVlisp Architecture into Smalltalk-80

`Class class` beeing a subclass of `Class` the instance method `newName:...:category:` is inherited and consequently can be used both to allocate a new class or new metaclass. This inheritance link between `Class` and its metaclass `Class class` can be compared with the self instantiation of `Class` provides by the ObjVlisp model.

```
Object ()
~~~~~~Behavior <new ...>
~~~~~~~~~ClassDescription <definition ...>
~~~~~~~~~~~~Metaclass <new definition ...>
~~~~~~~~~~~~Class <newName:superclass:instanceVariableNames:category: ...>
~~~~~~~~~~~~~~~Object class <...>
~~~~~~~~~~~~~~~~~~Behavior class <...>
~~~~~~~~~~~~~~~~~~~~~ClassDescription class <...>
~~~~~~~~~~~~~~~~~~~~~~~~Class class <...>
```

In order to create (allocate and initialize) classes we introduce a new instantiation message which is a simplification of the usual `subclass:...:category:` message[4].

We name this method `newName:superclass:instanceVariableNames:category:`. to make explicit the inheritance link, the structure and the category associated to the class[5].

Conforming to ObjVlisp, the message is now sent to the metaclass, i.e., the class which is the generator (descriptor) of the new object (class), and no more to the subclass as in standard

[4] `subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:`

[5] As advocated by ObjVlisp, “class and pool” variables are not yet used

Smalltalk-80. Remark that this is more conform to the semantics of generation of a new object as an instance of its class.

The complete implementation of the method includes a dispatch along the type of variables of the subclass (named or indexed variables). The strategy is analog to the standard instantiation message `subclass:...:category:` in Smalltalk-80. All the branches of the dispatch join together when calling the "private" instantiation method called `newName:environment:...:category:`. This method is analog to the standard Smalltalk method `name:inEnvironment:...:changed:`. Note that to simplify the undesrtanding of this method, we removed part of the code which manages the environment (syntax, changes ...), we replaced it by two comments.

```
!Class methodsFor: 'ObjVlisp'!

newName: className superclass: superClass instanceVariableNames: ivnString category: categoryString
      ^self
            newName: className
            environment: Smalltalk
            superclass: superClass
            otherSupers: nil
            instanceVariableNames: ivnString
            variable: false
            words: true
            pointers: true
            category: categoryString!

newName: className environment: env superclass: superClass otherSupers: otherSuperClassesArray instanceVariableNames: ivnString variable: v words: w pointers: p category: categoryString

      | newClass wasPresent oldClass |
      "Here takes place syntax checking and redefinition management."
      "Allocation of the new class"
      newClass _ self new.
      "Initialization of the new class"
      newClass
            superclass: superClass
            methodDict: MethodDictionary new
            format: -8192
            name: className
            organization: ClassOrganizer new
            instVarNames: (Scanner new scanFieldNames: ivnString)
            classPool: nil
            sharedPools: ObjVlispTestPoolSet.
      "(1)" otherSuperClassesArray isNil
               ifFalse: [newClass otherSuperclasses: otherSuperClassesArray].
      newClass
            format: newClass allInstVarNames size
            variable: v
            words: w
            pointers: p.
      env declare: className from: Undeclared;
          at: className put: newClass.
      ObjVlispOrganization classify: newClass name
                           under: categoryString asSymbol.
      "Here takes place hierarchy updating and change management."
      "(2)" otherSuperClassesArray isNil
```

```
                ifFalse: [newClass copyMethods].
        ^newClass!

hasMultipleSuperclasses
        ^false! !

!ClassDescription methodsFor: 'ObjVlisp'!

isObjVlispClass
        ^self class isKindOf: Metaclass! !
```

The class created appear now clearly as an instance of the receiver of the message. Instantiation of bottom end objects and classes is now uniform as in ObjVlisp.

One may remark that one of the parameter of the message, prefixed by the keyword `otherSupers:`, represents other superclasses. It is not used now, but will be used when we will introduce multiple inheritance. Thus we don't have to duplicate most of the code when introducing multiple inheritance into our extension, as opposed to Smalltalk-80 which had to. (Multiple inheritance was introduced as an extension into Smalltalk-80 after the design of Smalltalk-80, whereas we could designe our kernel more modular by anticipating on the extension:-)

The expressions commented "(1)" and "(2)" are evaluated in case of multiple inheritance. (1) assigns an array of the remaining superclasses to the class created. (2) calls the management of multiple inheritance as for the extension of Smalltalk-80. This will recompile the methods which need to and generate conflicting methods if needed.

The global variable `ObjVlispOrganization` refers to an organizer which will organize the browser specific to classes generated this way, as explained in section **??**.

If we want to discriminate between ObjVlisp classes and standard Smalltalk-80 classes, the minimal way is to check if the metaclass of the class is a standard Smalltalk-80 metaclass (instance of `Metaclass` or its subclass `MetaclassForMultipleInheritance`), or any kind of other class. Note that the kernel and its current prototype environment does not need it.

3.3 Grafting ObjVlisp

The class `ObjVlisp` is defined as a subclass-of `ClassDescription`. From a structural point of view an ObjVlisp class will be named but will not implement shared variables such as classPool and sharedPool:

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

From a behavioral point of view, and in order to create a class and a metaclass in a same way, the instantiation method `newName:...:category` must be defined both at the `ObjVlisp` level and duplicated at the `ObjVlisp class` level :

```
Object ()
~~~~~~Behavior <new new: ...>
~~~~~~~~~ClassDescription <...>
~~~~~~~~~~~~Metaclass <new ...>
~~~~~~~~~~~~Class <subclass:...:category ...>
~~~~~~~~~~~~~~~ObjVlisp <newName:...:category: ...>
```

```
~~~~~~~~~~~~~~~Object class <...>
~~~~~~~~~~~~~~~~~~ObjVlisp class <newName:...:category: ...>
```

4 The ObjVlisp Browser

```
Object <...>
~~~Behavior <...>
~~~~~~ClassDescription <definition>
~~~~~~~~~Class <definition>
~~~~~~~~~MetaClass <definition>
~~~~~~~~~~~~Object class <definition>

Browser <...>
~~~ObjVlispBrowser <text>

BrowserView class <>
~~~~~~ObjVlispBrowserView class <openOn:>
```

The Smalltalk-80 standard browser appears confusing for the programmer when browsing on ObjVlisp-like defined classes. The instance/class switch if set to class does not show any more the usual metaclass of the class but the real generator of the class. Moreover the template and the definition printed in the browser do not reflect the definition. If evaluated, the definition of the class already created won't lead to the same result, but will redefine the class in the standard Smaltalk-80 way while forgetting about the explicit metaclass.

Therefore we propose now a minimal browser dedicated to our kernel. The first problem we have to answer is how to distinguish between standard classes and ObjVlisp-like defined classes. In order to display their definition in a different way in the browser or when saving a definition (fileOut), we need to discriminate between classes. Smalltalk-80 does this because there are two `definition` methods. One belongs to class `ClassDescription` and the other one belongs to class `Metaclass`. The first one is used to display simple classes, the other one to display metaclasses. Unfortunately for us, our choice for minimality of the kernel imposes ObjVlisp classes to be instances of `Class` or one of its subclasses. If we redefine the method `definition` in `Class` all metaclasses will inherit this new definition (because they all are subclasses of `Class`), consequently the change will be for every class except metaclasses.

We want a solution both minimal and which does not change the underlying Smalltalk-80. We propose to redefine the method `definition` in class `Class` as proposed earlier. We also recopy the standard definition from `ClassDescription` onto `Object class` in order to bypass the lookup for `definition` through `Class`.

Note that the method checks if there are several superclasses in order to adjust the printing of the inheritance part. The method `nonTrivial` is redefined to signal the definition saving management (fileOut) that there is no metaclass to print.

The first remaining class method initializes the system and reset the main menu to propose the specific ObjVlisp browser. The last one redefines the template for ObjVlisp classes. It will be called by the specific ObjVlisp browser.

The code for the browser is a simplification of the standard browser. The instance/class switch disappears. The organization of ObjVlisp classes is specific (`ObjVlispOrganization`).

5 Library of Metaclasses

[Cointe and Graube 88]

We will now introduce some basic metaclasses defined from the ObjVlisp kernel. They will be used as bricks to define more complex metaclasses as developed in the next section.

- AbstractClass generating non-instantiable classes
- MemoClass generating classes memorizing their instances by using a backward pointer
- AutoInitClass generating classes initializing explicitly their instances at their creation time
- TypedClass generating parameterized classes à la ¡Eiffel
- AutoAccessClass generating classes with instance-methods to access (read-wite) to each instance variables.

5.1 AbstractClass

It is now easy to define the metaclass named `AbstractClass`:

```
Class newName: #AbstractClass
      superclass: Class
      instanceVariableNames: ''
      category: 'Metaclass-Library'!

!AbstractClass methodsFor: 'creation'!

new
      self error: 'no instance, I am an abstract class'!
new: n
      self error: 'no instance, I am an abstract class'! !
```

In order to also trap the instantiation message for classes with indexed variables (it will be used in the example of stacks, discussed in section **??**). We will omit these double definitions in the following of the paper.

5.2 AutoInitClass

Every Smalltalk-80 programmer has at least once redefined the method `new` in order to get an automatic initialization of instances. To avoid duplicating this code in a lot of methods, we define it in a specific metaclass, named `AutoInitClass`:

```
Class newName: #AutoInitClass
      superclass: Class
      instanceVariableNames: ''
      category: 'Metaclass-Library'!


!AutoInitClass methodsFor: 'creation'!

new
      ^super new init'! !
```

Every instance of this metaclass will automatically send the message `init` to its newly created instances.

5.3 MemoClass

The metaclass `AutoInitClass` may be used to generate classes but also metaclasses as we will see. Suppose now that we want classes which memorize the collection of all their instances. (It already exists in Smalltalk-80, through the implementation support, a method called `allInstances`). An instance variable, named `instances`, will keep all instances. It needs being initialized before starting creating new instances. Therefore the most practical is to ensure an automatic initialization. Therefore we define this new metaclass, named `MemoClass`, as an instance of previous metaclass `AutoInitClass`:

```
AutoInitClass newName: #MemoClass
      superclass: Class
      instanceVariableNames: 'instances '
      category: 'Metaclass-Library'!


!MemoClass methodsFor: 'init'!

init
      instances _ OrderedCollection new'! !

!MemoClass methodsFor: 'creation'!

new
      ^instances add: super new'! !
```

This construction is summarized by figure ??.

Note that this construction is impossible in Smalltalk-80, because the number of levels of instantiation is fixed for the user (only 3 levels), whereas in ObjVlisp, it is not limited.

As an example, we may now memorize the cartesian complex numbers, by defining a new class, named: `MemoCartesianComplex`, as an instance of `MemoClass`:

```
MemoClass newName: #MemoCartesianComplex
      superclass: CartesianComplex
      instanceVariableNames: ''
      category: 'Numeric-Complex'!
```

5.4 TypedClass

In order to parametrize classes, we introduce the metaclass, named `TypedClass`. It will define a new instance variable, named `type`:

Becausewe want to have a combined creation-initialization message we need to get one more class level in order to define the `newNamed:...:type:category:` message. This message will allow to give the type of a class at its creation time. Now we may not implicitly use class methods anymore, but we need to define explicitly the metaclass, that we will name

MetaTypedClass, under the usual ObjVlisp conventions. Then TypedClass is defined as an instance of MetaTypedClass:

```
Class newName: #MetaTypedClass
      superclass: Class
      instanceVariableNames: ''
      category: 'Metaclass-Library'!

!MetaTypedClass methodsFor: 'creation'!

newNamed: name subclass: subclass instanceVariableNames: ivNames type: type category: category
      ^(self newNamed: name subclass: subclass instanceVariableNames: ivNames category: category) type: type! !


MetaTypedClass newName: #TypedClass
      superclass: Class
      instanceVariableNames: 'type '
      category: 'Metaclass-Library'!

!TypedClass methodsFor: 'accessing'!

type
      ^type!

type: aClass
      type _ aClass! !

TypedClass newName: #Stack
      superclass: OrderedCollection
      instanceVariableNames: ''
      type: Object
      category: 'Metaclass-Library'!
```

5.5 AutoAccessClass

Some really repetitive programming in Smalltalk-80 is the definition of accessor methods ([Goldberg and Robson 83] page ??). They usually are named as the instance variables they access to. We propose now a metaclass which describe how to generate automatically such accessors. A method named makeIvAccessOn: will generate accessor methods onto the instance variables specified as parameter (an array of the strings of the names of the variables). The presence of this parameter keeps the programmer free to decide which variables will become public. We name the new metaclass which defines these methods AccessClass.

```
Class newName: #AccessClass
      superclass: Class
      instanceVariableNames: ''
      category: 'Metaclass-Library'!

!AccessClass methodsFor: 'access generation'!

makeIvAccessOn: ivNameArray
      ivNameArray do: [:ivNameString |
            self compile: ivNameString , '\ ^' withCRs , ivNameString
                 classified: #accessing;
```

```
                 compile: ivNameString , ': value\
' withCRs , ivNameString , ' _ value'
                 classified: #accessing]! !


AutoInitClass newName: #AutoAccessClass
      superclass: AccessClass
      instanceVariableNames: ''
      category: 'Metaclass-Library'!

!AutoAccessClass methodsFor: 'init'!

init
      self makeIvAccessOn: instanceVariables! !
```

Now we would like that for accessors generation to be automatic for some classes, i.e., done as an initialization when defining such a class. Therfore we define a new metaclass, named `AutoAccessClass`, which will be an instance of `AutoInitClass`. We just need to define the `init` method to call `makeIvAccessOn:` on all instance variables defined by the class:

Unfortunately this construction does not work. The method `newNamed:superclass:instanceVariableNames:category:` (owned by `Class`) calls the method `new` (owned by `AutoInitClass`) which calls the previous `new` (owned by `Behavior`) and then applies `init` to the newly class created. Unfortunately the class is created but not initialized yet. The instance variable `instanceVariableNames` is still undefined (value `nil`), thus there is no accessor generated, assumed that there is no instance variable. In order the automatic initialization to work not only for simple instances, but also for classes, we need to redefine the method for creating classes, named `newNamed:subclass:instanceVariableNames:category:`, in `AutoInitClass`:

```
!AutoInitClass methodsFor: 'creation'!

newNamed: name subclass: subclass instanceVariableNames: ivNames category: category
      ^(self newNamed: name subclass: subclass instanceVariableNames: ivNames category: category) init! !
```

6 Further Composition of Metaclasses using Multiple Inheritance

We introduced the current library of metaclasses that the programmer may use. In non trivial cases, this is useful to combine several such metaclasses, for instance to describe classes which memorize and initialize their instances by combining `MemoClass` and `AutoInitClass`. Therefore we need multiple inheritance. We need to define an interface of ObjVlisp with the standard extension of multiple inheritance in Smalltalk-80 to reuse it. In order to keep the kernel of Objtalk minimal and modular, we define the interface with multiple inheritance facility as an extension of our kernel, i.e., a set of metaclasses. We define a new metaclass, named `MultipleInheritanceClass`. This metaclass will define a new instance variable, named `otherSuperclasses`, to hold the superclasses except the first one. They will be kept in an array. (Note that the extension for multiple inheritance in standard Smalltalk-80 defines a new metaclass, named `MetaclassForMultipleInheritance`, which holds an instance variable named `otherSupers`. The strategy chosen in Smalltalk-80 is that the metaclass of the class keeps the remaining superclasses. This works because there is only one metaclass for each class. But this is a bit strange. Our implementation is more consistent, because the class itself keeps the remaining superclasses).

In order to define a new message for creating such classes and specifying its superclasses, we need to define the metaclass of `MultipleInheritanceClass`, that we will name `MetaMultipleInheritanceClass`:

```
Class newName: #MetaMultipleInheritanceClass
      superclass: Class
      instanceVariableNames: ''
      category: 'Multiple-Inheritance'!


!MetaMultipleInheritanceClass methodsFor: 'creation'!

newName: className superclasses: superclassNames instanceVariableNames: ivnString category: categoryString
      | superclasses firstSuperclass |
      superclasses _ Class getSuperclasses: superclassNames.
      firstSuperclass _ superclassNames first.
      ^self
            newName: className
            environment: Smalltalk
            superclass: firstSuperclass
            otherSupers: (superclassNames copyFrom: 2 to: superclassNames size)
            instanceVariableNames: ivnString
            variable: false
            words: true
            pointers: true
            category: categoryString! !
```

In fact there is a dispatch along the type of variables (named or indexed) which is not shown here. This method is analog to the method `newNamed:superclasses:instanceVariableNames:category:` owned by `Class`. In fact there is also a dispatch along the type of variables (named or indexed) which is not shown here. Like in the standard extension for multiple inheritance, the first superclass is used by the standard implementation, the recompilation of some of the methods belonging to the other superclasses will occur after creating the class.

Now we may define the metaclass `MultipleInheritanceClass` as instance of `MetaMultipleInheritanceClass`. We define it also as a subclass of `MetaMultipleInheritanceClass` in order for being able to later generate other metaclasses for multiple inheritance, as shown in **??**.

```
MetaMultipleInheritanceClass newName: #MultipleInheritanceClass
      superclass: MetaMultipleInheritanceClass
      instanceVariableNames: 'otherSuperclasses '
      category: 'Metaclass-Library'!

!MultipleInheritanceClass methodsFor: 'accessing class hierarchy'!

hasMultipleSuperclasses
      ^true!

otherSuperclasses
      ^otherSuperclasses!

otherSuperclasses: classArray
      otherSuperclasses _ classArray!
```

```
superclasses
      "Answer with an array of all the receiver's superclasses."
      superclass == nil ifTrue: [^#()].
      self hasMultipleSuperclasses
            ifTrue: [^(Array with: superclass) , otherSuperclasses].
      ^Array with: superclass! !
```

The methods it defines are trivial. The method `superclasses` redefines the standard method with same name owned by class `Behavior` in order to look for the other superclasses inside the class itself and not inside its metaclass.

7 Multi-levels Instantiation Hierarchy: A developed Examples

We will now describe a complete example to summarize many of the library metaclasses already described. In this example we intend to define typed stacks, i.e., stacks whose parameter of the `push:` method is typechecked. To ease the definition and to show how we may reuse classes, we suppose that a class named `Stack` has been already defined (for instance as a sublass of `Array` and by adding an index to it).

We want to be able to express different types of stacks, for instance `IntegerStack`, `StringStack`.... The class `TypedStack` will define the common behavior. In order to parametrize the type of the stacks, `TypedStack` should be an instance of metaclass `TypedClass`.

We could then define classes `IntegerStack`, `StringStack` ... as subclasses of `TypedStack`. Therefore `TypedStack` will be an abstract class. This means that `TypedStack` should be an instance of metaclass `AbstractClass`.

Our last requirement is the automatic initialization of typed stacks. If we assume that there is a method named `init` defined in class `Stack` and which initalizes a stack (for instance by setting the index with value 0), this method should be called when creating a stack. Therefore classes `IntegerStack`, `StringStack` ...should be instances of the metaclass `AutoInitClass`. Note that `TypedStack` needs not because it won't have instances, consquently to initialize them is out of context.

If we summarize our requirements, `TypedStack` should be instance of both `TypedClass` and `AbstractClass`. The concrete classes `IntegerStack`, `StringStack` ...should be instances of `TypedClass` and `AutoInitClass`.

Because a class (and any kind of object) cannot be instance of several classes, we will define two new metaclasses, `AbstractTypedClass` as a subclass of `AbstractClass` and `TypedClass`, and `AutoInitTypedClass` as a subclass of `AutoInitClass` and `TypedClass`. They both must be instances of `MultipleInheritanceClass` because they have multiple superclasses. Remark that the ordering of names is not significant in terms of specification (`TypedAbstractClass` would be equivalent), and is just a choice for clarity. However the implementation of multiple inheritance in Smalltalk-80 is not symetric (the first superclass is privileged). Consequently in order to ease the resolution of conflicting methods, the choice for ordering the superclasses may be significant.

Our last requirement will be to be able to use the creation/initialization message `newNamed:subclass:instanceVariableNames:type:category:` in order to specify the type of a class stacks when creating it, for instance to associate the class `IntegerStack` with the class `Integer` as its type. Therefore the metaclass `AutoInitTypedClass` should be instance of the metaclass `MetaTypedClass`. Remark that on the other hand the metaclass `AbstractTypedClass` does not have necessarily to, because the value of the type is not significant for its instance the abstract class `TypedStack`.

In summary, `AutoInitTypedClass` should be instance of both `MultipleInheritanceClass` and `MetaTypedClass`. Therefore, and as earlier, we will create a new metaclass, subclass of these two metaclasses and that we name MultipleInheritanceMetaTypedClass:

```
MultipleInheritanceClass newName: #MultipleInheritanceMetaTypedClass
      superclasses: 'MultipleInheritanceClass MetaTypedClass '
      instanceVariableNames: ''
      category: 'Metaclass-Combination'!

!MultipleInheritanceMetaTypedClass methodsFor: 'conflicting inherited methods'!

hasMultipleSuperclasses
      ^super hasMultipleSuperclasses!

superclasses
      ^super superclasses! !
```

Note that because we are using multiple inheritance, we need to solve explicitly the conflicts. The two conflicting methods, hasMultipleSuperclasses and hasMultipleSuperclasses, should be found in MultipleInheritanceClass. to use the pseudo variable super is sufficient here because we use the fact that MultipleInheritanceClass is the first s¡uperclass.

We will now define the metaclasses TypedStack:

```
MultipleInheritanceClass newName: #AbstractTypedClass
      superclasses: 'AbstractClass TypedClass '
      instanceVariableNames: ''
      category: 'Metaclass-Combination'!


!AbstractTypedClass methodsFor: 'conflicting inherited methods'!

new
      ^super new!

new: n
      ^super new: n! !
```

Again the conflicting methods are redirected onto the first superclass, because an abstract class should forbid instantiation.

We now define AutoInitTypedClass, this later one needs to be an instance of MultipleInheritanceMetaTypedClass, as stated above:

```
MultipleInheritanceMetaTypedClass newName: #AutoInitTypedClass
      superclasses: 'AutoInitClass TypedClass '
      instanceVariableNames: ''
      category: 'Metaclass-Combination'!


!AutoInitTypedClass methodsFor: 'conflicting inherited methods'!

new
      ^super new!

new: n
      ^super new: n!

newName: className superclass: superClass instanceVariableNames: ivnString category: categoryString
      ^super newName: className superclass: superClass instanceVariableNames: ivnString category: categoryString! !
```

Now we may define at first the abstract class `TypedStack`:

```
AbstractTypedClass newName: #TypedStack
      superclass: Stack
      instanceVariableNames: ''
      category: 'Stack-Collection'!


!TypedStack methodsFor: 'accessing'!

type
      ^self class type! !

!TypedStack methodsFor: 'operations'!

push: x
      (x isKindOf: self type)
            ifTrue:  [super push: x]
            ifFalse: [self error: 'wrong type']! !
```

The method `push:` is being redefined by checking the type of the class. Therefore we need to define the method `type` in order to access (through the method `type` owned by `MetaTypedClass`) the type owned by the class.

We are now ready to define some concrete subclasses:

```
TypedAutoInitClass newName: #IntegerStack
      superclass: TypedStack
      instanceVariableNames: ''
      type: Integer
      category: 'Stack-Collection'!


TypedAutoInitClass newName: #StringStack
      superclass: TypedStack
      instanceVariableNames: ''
      type: String
      category: 'Stack-Collection'!
```

An integer stack may be created by instantiating `IntegerStack`. One may remark that there are 7 levels of instantiation between this terminal instance and initial metaclass `Class`. The construction is summarized by figure **??**.

8 Problems introduced by explicit metaclasses

ObjVlisp is now available in Lisp, Smalltalk and Prolog [Malenfant, Lapalme and Vaucher 89].

8.1 Conception

8.2 Compatibility

We could further complexify this example by adding to typed stacks the ability of memorizing their instances. By inheriting both from `AutoInitClass` and `MemoClass`, we encounter a combination problem. This time choosing the right `new` method to solve the conflict is not enough, we need a real combination of the two behaviors. Unfortunately such ability of method combination is not avalaible in the extension for multiple inheritance of Smalltalk-80 that we are reusing.

9 Further Work

Experiment and make experiment :-) Extend the environment. One thing to do: design a instantiation tree browser, analog to the inheritance hierarchy. Using Smalltalk as a new platform to graft sub-systems

(Meta)class compatibility

10 Annex

```
!Class methodsFor: 'ObjVlisp'!

definition
      | aStream |
      aStream _ WriteStream on: (String new: 300).
      aStream nextPutAll: self class name printString.
      aStream nextPutAll: ' newName: '.
      self name storeOn: aStream.
      aStream cr; tab.
      self hasMultipleSuperclasses
            ifTrue: [aStream nextPutAll: 'superclasses: ';
                     store: self superclassesString]
      ifFalse: [aStream nextPutAll: 'superclass: ';
                nextPutAll: self superclass name printString].
      aStream cr; tab; nextPutAll: 'instanceVariableNames: '.
      aStream store: self instanceVariablesString.
      aStream cr; tab; nextPutAll: 'category: '.
      (ObjVlispOrganization categoryOfElement: self name) asString storeOn: aStream.
      ^aStream contents!


nonTrivial
      ^false! !


!Class class methodsFor: 'ObjVlisp'!

initializeObjVlisp
      "Class initializeObjVlisp"
      Smalltalk at: #ObjVlispOrganization put: SystemOrganizer new.
      ScreenController compile: 'openObjVlispBrowser\
ObjVlispBrowserView openOn: ObjVlispOrganization' withCRs classified: 'menu messages';
            addMenuLabelString: '\objvlisp browser' withCRs messageArray: #(openObjVlispBrowser);
            allInstancesDo: [:c | c initializeYellowButtonMenu]!

objVlispTemplate: category
      ^'Class newName: #NameOfClass
      superclass: SubClass
      instanceVariableNames: ''instVarName1 instVarName2''
      category: ''' , category , ''''! !


Browser subclass: #ObjVlispBrowser
      instanceVariableNames: ''
```

```
      classVariableNames: ''
      poolDictionaries: ''
      category: 'ObjVlisp-Browser'!


!ObjVlispBrowser methodsFor: 'text'!

text
      | text |
      textMode == #classDefinition ifTrue:
      [className == nil
            ifTrue: [^ (Class objVlispTemplate: category) asText]
            ifFalse: [^ self selectedClass definition asText]].
      textMode == #methodDefinition
            ifTrue: [selector == nil
                        ifTrue: [^ self selectedClass sourceCodeTemplate asText]
                        ifFalse: [^ (self selectedClass sourceCodeAt: selector) asText
      makeSelectorBoldIn: self selectedClass]].
      textMode == #category ifTrue:
      [^ 'category to add' asText].
      textMode == #categories ifTrue:
      [^ organization printString asText].
      textMode == #protocol ifTrue:
      [^ 'protocol to add' asText].
      textMode == #protocols ifTrue:
      [^ self selectedClass organization printString asText].
      textMode == #comment ifTrue:
      [text _ self selectedClass comment asText.
      text isEmpty ifFalse: [^ text].
      self selectedClass isMeta ifTrue: [^'Select the browser switch "instance" to see the comment' asText].
      ^ 'This class has no comment' asText].
      textMode == #hierarchy ifTrue:
      [^ self selectedClass printHierarchy asText].
      ^ Text new! !


BrowserView subclass: #ObjVlispBrowserView
      instanceVariableNames: ''
      classVariableNames: ''
      poolDictionaries: ''
      category: 'ObjVlisp-Browser'!


!ObjVlispBrowserView class methodsFor: 'instanceCreation'!

openOn: anOrganizer
      | topView aBrowser topY bottomY  |
      aBrowser _ ObjVlispBrowser new on: anOrganizer.
      topY _ 0.35.
      "change this to re-proportion system browser"
      bottomY _ 1 - topY.
      (topView _ self model: aBrowser label: 'ObjVlisp is winning again' minimumSize: 400 @ 250)
      addCategoryView: (0 @ 0 extent: 0.25 @ topY) on: aBrowser readOnly: false;
```

```
addClassView: (0.25 @ 0 extent: 0.25 @ topY) on: aBrowser readOnly: false;
addProtocolView: (0.5 @ 0 extent: 0.25 @ topY) on: aBrowser readOnly: false;
addSelectorView: (0.75 @ 0 extent: 0.25 @ topY) on: aBrowser readOnly: false;
addTextView: (0 @ topY extent: 1.0 @ bottomY) on: aBrowser initialSelection: nil.
topView icon: (Icon constantNamed: #default).
topView controller open! !
```

References

[Borning and Oshea 87]

[Bobrow and Kickzales 88] 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, Snow-Bird Utah, USA, July 1988.

[Briot and Cointe 87] 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

[Cointe 87] P. Cointe, Metaclasses are First Class: the ObjVlisp Model, in [OOPSLA 87], pages 156-167.

[Cointe and Graube 88] Cointe, P., Graube, N., Programming with Metaclasses in CLOS Proceedings of the First CLOS Users and Implementors Workshop, Xerox Parc, Palo Alto, California, USA October 1988.

[Graube 88]

[Cointe 88] P. Cointe, A Tutorial Introduction to Metaclass Architectures as Provided by Class Oriented Languages, International Conference on Fifth Generation Computer Systems (FGCS'88), Vol. 2, pages 592-608, Tokyo, Japan, 28 November - 2 December 1988.

[Goldberg and Robson 83] A. Goldberg and D. Robson, Smalltalk-80: the Language and its Implementation, *Series in Computer Science*, Addison Wesley, Reading MA, USA, 1983.

[Ingalls and Borning 82] D.H.H. Ingalls and A.H. Borning, Multiple Inheritance in Smalltalk-80, Proceedings of the National Conference on Artificial Intelligence, AAAI, pages 234-237, August 1982.

[Malenfant, Lapalme and Vaucher 89] J. Malenfant, G. Lapalme and J. Vaucher ObjVProlog; Metaclass in Logic Ecoop-89, to appear, July 1989.

Contents