# The OBJVLISP Model: Definition of a Uniform, Reflexive and Extensible Object Oriented Language

Jean-Pierre BRIOT* Pierre COINTE†

## Abstract

This paper presents the ObjVlisp model designed to experiment and synthesize the expanding activities of the Object Oriented world. The goal is to specify a minimal kernel whose semantic is perfectly uniform, then to modelize other Object semantics by extending this concentrate kernel.
We propose a unification of the metaclass, class and object concepts which allows an optimal uniformity. This unification is obtained as a reflexive definition of the language. We rapidly describe some ObjVlisp variations establishing the malleability of the system. The kernel is built on a portable virtual machine defined as a set of Lisp primitives which preserve the Lisp evaluator and maintain the spirit of dynamic (or lexical) scoping.

**Keywords:**

object, class, metaclass, uniformity, instance variable, classe variable, message, instantiation, CLASS, inheritance, OBJECT, Lisp, Smalltalk, ObjVlisp, virtual machine, dynamic scoping, lexical scoping, portability, reflexive, extensible.

## 1 Uniformity

*"One way of stating the Smalltalk philosophy is to choose a small number of general principles and apply them uniformly"* [Kra83].

### 1.1 Is a class an object?

In most common Object Oriented Languages, besides Krasner's uniformity "wish", a class is not a REAL object. Some of them however, like Smalltalk-80 [GR83] and Loops [BS83], introduced the new *metaclass* concept to provide a greater abstraction. To definitively suppress the gap between *class* and *object*, we propose a unification of the *metaclass*, *class* and *object* concepts.
We claim [Bri85] that a class must be an object, allowing greater clarity and expressive power. As a consequence, we no longer need to introduce the *class variable* and *class method* concepts of Smalltalk which are not real variables and methods.

### 1.2 Class and terminal instance

The reverse question is *"is every object a class?"*, whose answer is "no". [1] Some objects are "only" instances of a class, and don't define a model. An instance of a POINT class, e.g. an object *a_point*, or an instance of the NUMBER class, e.g. 3, are such objects. We call them *terminal instances*.

### 1.3 How many object types?

We consider only one kind of objects, without structure or nature distinction between generators (*classes*) and non generators (*terminal instances*). In fact, they only differ by their capacity to react to the instantiation message. If the class of an object owns the primitive instantiation method (*new* selector, owned by the primitive class **CLASS**) or inherits it, this object is a generator, otherwise a terminal instance.

### 1.4 Metaclasses are usual classes

A metaclass (or a metagenerator) is simply a class which generates other classes. Every class declared as a subclass of the metaclass **CLASS** inherits its *new* method, and becomes a metaclass. Therefore the introduction of the metaclass concept is unnecessary and **"the distinction between metaclasses, classes and terminal instances [2] is only a inheritance consequence and not a type distinction. There is now only one type of objects in the model."**
This unification induces a simplification of the instantiation and inheritance concepts but imposes to use them simultaneously: for example a metaclass is created as the subclass of another one (as an "ultimate" subclass of **CLASS**).

*Tokyo Institute of Technology, Dept. of Information Science, Ookayama, Meguro-ku, Tokyo 152, & LITP, 4 place Jussieu, 75005 Paris.

†Centre Mondial Informatique, 22 Avenue Matignon, 75008 Paris, electronic mail: ...!mcvax!inria!cmirh!pc & LITP 4 place Jussieu, 75005 Paris.

[1] In the *actor* model, an *actor* describes its own structure and exists without a class. Defining generator actors [Yon86] or using a copy mechanism [Lie86] [CBS86] are alternatives to the class model.

[2] To easily distinguish them, we use upper-case for classes, upper-bold for metaclasses, and lower-case for terminal instances.

# 2 The ObjVlisp Model

Historically the $\boxed{\text{ObjVlisp}}$ model comes from our work on Smalltalk-76 [Coi83], and our wish to present a synthesis in an operational semantic expressed in Lisp [Coi84b]. We present here the new reflexive version which exactly integrates the previous unification and gives a fine solution to the problem of <*class* , *instance*> dichotomy.

## 2.1 ObjVlisp in five Postulates

Following the classical presentation of Smalltalk-76 [Ing78], five postulates fully describe the new $\boxed{\text{ObjVlisp}}$ model:

**P1:** every entity of the language is an *object*. An object represents a piece of knowledge and a set of potentialities:

$$\boxed{object \;=\; <\ data\ ,\ procedures\ >}$$

**P2:** the only control structure is *message passing*: the protocol to activate an object.
A message specifies which procedure to apply (denoted by its name, the *selector*), and the arguments to pass:

$$\boxed{(\text{send}\ object\ selector\ Args_1\ \ldots\ Args_n)}$$

**P3:** every object belongs to a *class* that specifies its data (attributes called *fields* or *instance variables*) and its behavior (procedures called *methods*). Objects will be dynamically generated from this model, they are called *instances* of the class. Following Plato, all instances of a class have same structure and shape, but differ through the values of their common *instance variables*.

**P4:** a class is also an object, generated from a class, called a *metaclass* (because describing a class). Consequently (P3), to each class is associated a metaclass. The initial primitive metaclass is the class **CLASS**.

**P5:** a class can be defined as a subclass of one (or many) other class(es). This subclassing mechanism allows inheritance of fields and methods, and is called *inheritance*. The class OBJECT represents the most common behavior shared by all objects.

## 2.2 Classes and objects

### 2.2.1 Structure of an object

The postulates P1 & P3 define an object as a "chunk" of knowledge and actions whose structure is defined by its class. More precisely:

**fields:** This environment (also called a dictionary) is split in two parts:
a) the set of *instance variables* specified by the object's class,
b) the set of associated values.
The set of instance variables belongs to the class, and is shared by all its instances. The set of values is owned by each instance. An object cannot exist without its class.

Each object implicitly holds three "pseudo-fields"; **self** and **isit** [3] are bound to the object itself and its class, **metaisit** denotes the metaclass of the object and is characteristic [4] of the $\boxed{\text{ObjVlisp}}$ model.

**methods:** The methods define the procedures shared by all the instances of a class and owned by it.

To realize the unification between class and instance we represent this method environment as a particular field of a class; the methods dictionary of a class is the value associated to a specific instance variable called *methods*. As a common object, a class is defined by its (meta)class and the values of the associated instance variables.

### 2.2.2 Structure of a class

We present now the three instance variables describing a class. They are owned by **CLASS** as the primitive metaclass:

***supers :*** the list of super-classes of the class,

***i_variables :*** the list of instance variables that the class specifies,

***methods :*** the list of methods held by the class organized in a P-list way, with pairs <*selector* , *lambda-expression*>.

### 2.2.3 Instantiation of a class

A class is instantiated by sending it a message of selector *new*. The *new* message received by a class provides the generation of a new object built on the class model. For example, we define the class POINT by instantiating the metaclass **CLASS**; we specify the name of the object (value of self) and the three values associated to the **CLASS**'s instance variables:

```
; (send 'CLASS 'new <name> <supers> <i_variables> <methods>)
(send 'CLASS 'new 'POINT '(OBJECT) '(x y)
   '( draw   (lambda () (tty-print x y "*")) ) )
```

[3] respectively analogs to SELF and ISIT in Smalltalk-72 [GK76]

[4] We use it to simulate the *class variables* even if we have the double equality: $\boxed{\textbf{metaisit} = (\text{class-of}\ \textbf{isit}) = (\text{send}\ \textbf{isit}\ \text{'is})}$

Then we create an instance of POINT, using the same *new* message: [5]

```
; (send 'POINT 'new <name> <x> <y>)
(send 'POINT 'new 'a_point 40 15)
(send 'POINT 'new 'another_point 24 12)
```

The semantic and syntax of the *new* method are totally uniform. The *new* message always receives as arguments: the name of the instance to generate, and the values related to the instance variables specified by the receiver-class.
Unlike the SMALLTALK-80 system, ObjVlisp uses only one method (named *new*) to create an object (by using the **make-object** primitive of the virtual machine). This method is owned by the metaclass **CLASS** as expressed by its reflexive definition.

# 3 Reflexivity

Because we need a complete transparency in the objects definitions to give more complete control to the users, we adapt the reflexive interpreter technic [dRS84] to the construction of our model. ObjVlisp is supported by two graphs: the *instantiation graph* and the *inheritance graph*. The instantiation graph represents the "*instance of*" relation (P3 & P4), and the inheritance graph the "*subclass of*" one (P5). **CLASS** and OBJECT are the roots of these two graphs: they are defined in ObjVlisp .

[5] This table shows the half-dictionary of values owned by each object:

```
? (send 'a_point 'i_values)
= (40 15)
? (send 'POINT 'i_values)
= ((OBJECT) (x y) (draw ..))
? (send 'CLASS 'i_values)
= ((OBJECT) (supers i_variables methods) (new ..))
```

## 3.1 CLASS: Instantiation

**CLASS** is the first object of the system, as the root of the instantiation graph, it will recursively generate all other objects. **CLASS** is also its own instance and we have the equality:

class [**CLASS**] = **CLASS**

### 3.1.1 Self pattern-matching of CLASS

To verify the previous equality we have to guarantee that the instance variables specified by **CLASS** match the corresponding values also held by **CLASS**, as its own instance:

| | | | |
|---|---|---|---|
| *i_variables* : | supers | **i_variables** | methods |
| *i_values* : | (OBJECT) | **(supers i_variables methods)** | (new (lambda$_1$ ..) ... menu (lambda$_n$ ..)) |

The value associated to the instance variable *i_variables* is exactly the list of instance variables itself, this self pattern-matching illustrates the definition of **CLASS** as an object.

### 3.1.2 Boot-strap

*"A natural and fundamental question to ask, on learning of these incredibly interlocking pieces of software and hardware is: "How did they ever get started in the first place?". It is truly a baffling thing"* [Hof79].

Defining **CLASS** from itself necessitates to precise the boot-strap mechanism. We create the skeleton of **CLASS** using the **make-object** primitive. We just need to introduce the *new* method which supports the self-instantiation of **CLASS**:

```
; (make-object <metamodel> <model> <instance> <i_variables> <i_values>)
(make-object 'CLASS 'CLASS 'CLASS
  '( supers        i_variables                   methods )
  '( (OBJECT)   (supers i_variables methods)   (new (lambda (name . i_values)
                                                  (make-object isit self name i_variables i_values))) ) )
```

This bootstrap prepared, we can define ObjVlisp in ObjVlisp :

### 3.1.3 Self-instantiation of CLASS

```
; (send 'CLASS 'new <name> <supers> <i_variables> <methods>)
(send 'CLASS 'new 'CLASS '(OBJECT) '(supers i_variables methods)
  '( new          (lambda (name . i_values) (make-object isit self name i_variables i_values))
     supers       (lambda () supers)
     i_variables  (lambda () i_variables)
     methods      (lambda () methods)
       ...             ...
     methodfor    (lambda (selector) (methodfor selector methods))
     change       (lambda (selector method) (changemethod selector method methods))
     understand   (lambda (selector method) (defmethod selector method methods))
     menu         (lambda () (selectors methods))  ) )
```

This definition is given for a dynamically scoped Lisp (Le_Lisp [CDH84]) as other ones and examples. Then all the instance variables are automatically bound to their values in a method body. Consequently the lambda-methods associated to the *supers*, *i_variables* and *methods* selectors are quite easy to express.

Similarly, in the *new* method, *isit* denotes the metagenerator (here **CLASS**), *self* the generator (again **CLASS**), and *i_variables* the variables to instantiate with i_values. They are used as free variables in the **make-object** call.

## 3.2 OBJECT: Inheritance

(P5) precises the inheritance mechanism which concerns only classes. Inheritance allows to connect together fields and methods of several classes but in two differents ways.

The inheritance of fields is static and done once at creation time.
When defining a class, its instance variables are calculated as the union of a copy of the instance variables owned by the superclasses with the instance variables specified at creation (the value associated to *i_variables* in the *new* message).

On the contrary, the methods inheritance is dynamic and uses the virtual copy mechanism realized by the linkage of classes in the inheritance graph.
If the lookup of a method fails in the receiver class, then the search continues in a depth/breadth way. This graph is supported by the *supers* instance variable.

### 3.2.1 Classes vs Terminal Instances

The inheritance mechanism of fields is applied only when creating classes. Thus we need to discriminate creation of classes and creation of terminal instances. We have the equivalence:

*create a class* <=> *creator is a metaclass*

As we pointed out already, a class inheriting from **CLASS** is a metaclass and inherits statically the *supers*, *i_variables* and *methods* instance variables. The predicate-method *metaclass?*, presented below, implements this test. It is used by the **make-object** primitive, thus no inheritance will occur when creating a terminal instance.

### 3.2.2 OBJECT the most common class

The second primitive class is OBJECT, instance of **CLASS**. OBJECT is usually the default specified superclass (e.g. **CLASS** is a subclass of OBJECT), so it represents the most common class (intersection of all classes), describing the most common behavior (for classes and terminal instances).

```
; (send 'CLASS 'new <name> <supers> <i_variables> <methods>)
(send 'CLASS 'new 'OBJECT () ()
 '( is           (lambda () isit)
    ?            (lambda (i_var) (eval i_var))
    ?<-          (lambda (i_var i_val) (set i_var i_val))
    i_values     (lambda () (i_values self))
    metaclass?   (lambda () (memq 'supers i_variables))
    class?       (lambda () (send isit 'metaclass?))
     ...           ...
    print        (lambda () (pretty self))
    error        (lambda msg `(lambda bs ',msg)) ) )
```

From this definition, each $\boxed{\text{ObjVlisp}}$ object answers to the <*selector*> by <*action*>:

| | |
|---|---|
| *is* | giving the name of its class |
| *?* | returning the value of the field i_var |
| *?<-* | writing i_var with the new value |
| *i_values* | returning the values of the fields |
| *metaclass?* | testing if the object is a metaclass |
| *class?* | testing if the object is a class |
| *print* | printing an external representation |
| *error* | precising the standard error treatment |

Notice that the ? and ?<- methods which access to the instance variables are conform to the dynamic binding of Le_Lisp .

## 3.3 Self-Extensions

### 3.3.1 Class variables by Example

Let us return to the POINT class, previously defined. Now we would like the constant character "⋆" to be a class variable shared by all the points of a same class. We redefine the POINT class as before, but metaclass of which (let's call it **METAPOINT**) specifies this common character:

```
; (send 'CLASS 'new <name> <supers> <i_variables> <methods>)
(send 'CLASS 'new 'METAPOINT '(CLASS) '(char) ())

; (send 'METAPOINT 'new <name> <supers> <i_variables>
    <methods>
    <char>)
(send 'METAPOINT 'new 'POINT '(OBJECT) '(x y)
   '( draw    (lambda () (tty-print x y char)) )
   "⋆" )
```

**METAPOINT** is declared as a subclass of **CLASS** (thus it is a metaclass). It inherits the instance variables *supers*, *i_variables* and *methods* from **CLASS** and adds them the instance variable *char*. Consequently, POINT specifies the associate value of *char*, i.e. "⋆".
Now we could create such a point:

```
? (send 'POINT 'new 'a_point 20 10)
= a_point
? (send 'a_point 'draw)
            ⋆
= ⋆
```

**Parametrization of a class:** The POINT class is now parametrized by the "display" character and the **METAPOINT** metaclass represents this abstraction. Let's define two new classes, called POINT# and POINT@ with two other different display characters. Obviously, they are defined as a subclass of POINT:

```
? (send 'METAPOINT 'new 'POINT# '(POINT) () () "#")
= POINT#
? (send 'METAPOINT 'new 'POINT@ '(POINT) () () "@")
= POINT@
? (send (send 'POINT# 'new 'a_point# 1 2) 'draw)
 #
= #
? (send (send 'POINT@ 'new 'a_point@ 3 4) 'draw)
  @
= @
```

Such a simple and intuitive construction is IMPOSSIBLE in Smalltalk, because *class variables* are implemented like *pool variables* [GR83], and are not inheritable.
With the ObjVlisp model we have the identification:

class variable [*an_object*] =
instance variable [*an_object*'s class]

### 3.3.2 Architecture of the ObjVlisp model

We summarize the general structure of the ObjVlisp model by connecting together the *instantiation graph* and the *inheritance graph* to explain the "POINT construction":

Below an object, appears its half-dictionary of values; they correspond to the instance variables of its class, this list (in bold) being itself a value corresponding to the *i_variables* instance variable owned by **CLASS**, and inherited by **METAPOINT**.
There is no longer any depth limitation of the instantiation graph. We can extend it as much as we want to specify different levels of shared knowledge and methods.

### 3.3.3 Class methods by Example

As for class variables, *class methods* are specified in the metaclass as usual methods. Suppose we want to define a new *class method* for POINT to create and initialize a new point. We simply define the *newinit* method of **METAPOINT** (assuming we define also an *init* method in the POINT class, or at least in the OBJECT class):

```
(send 'METAPOINT 'understand 'newinit
  '(lambda (name)
     (send (send self 'new name) 'init)) )
```

### 3.3.4 Filiation link

To use classes registering all their instances, we define a new metaclass **SET**, as a subclass of **CLASS** with a new instance variable *sons* pointing the list of instances. We just have to redefine the **CLASS**' *new* method to add the newly instance at the end of the *sons* list:

```
(send 'CLASS 'new 'SET '(CLASS) '(sons)
  '( new        (lambda (name . i_values)
                   (nconc sons (cons name ()))
                   (run-super))
     mapsons    (lambda (selector)
                   (mapc
                      (lambda (an_object)
                         (send an_object selector))
                      sons)) ) )
```

The *mapsons* method distributes an unary message (without argument) to all the instances of a particular set. The "(run-super)" form (same as in CommonLoops [BKK*85]) recalls the current message, but the lookup for the method will begin in the superclasses of the current class.

# 4 Implementation

ObjVlisp means OBJect extension of a Virtual LISP and is implemented along the virtual machine technic allowing a quick installation on any Lisp (dynamically or lexically scoped). The classical difficulty when defining such a machine is to choose general primitives available on major aimed systems without privileging particular constructions, even if they are powerful for a given target. Consequently we have decided to preserve the <eval , apply> engine, and to use classical Lisp data structures such as atom, list, structure or environment. The Le_Lisp instantiation of this virtually machine is totally described in [Coi84a], and we are just finishing its Scheme and

CommonLisp mergings.
To synthesize our previous presentation we conclude this paper with Le_Lisp & Scheme definitions of an object:

**ObjLe_Lisp :** an object is implemented as a Lisp atom whose functional value (**send** is equal to the *funcall* function) is a self-referenced closure:

```
(defun an_object (selector . args)
-1-   (let ( (self 'an_object) (isit 'A_CLASS)
-1-          (metaisit 'A_METACLASS) )
-2-     (letdic (i_variables metaisit) (i_values isit)
-2-       (letdic (i_variables isit) (i_values self)
-3-         (protect (apply (lookup selector isit) args)
-3-           (rewrite self isit metaisit))))))
```

1. The pseudo instance variables are dynamically bound by the "let" function. [6] Notice that those variables allow a factorization of the script of an object, the parts **-2-** to **-3-** defining a completely parametrized (and shared) piece of code,

2. in the spirit of a dynamic Lisp, the activation of an object is done in its fields' context. This double environment is dynamically set by the two "letdic" (let dictionary) functions:
"(letdic (i_variables metaisit) (i_values isit)" sets the class environment,
and "(letdic (i_variables isit) (i_values self)" the object's one,

3. when the object is disactivated, the updating of the double closure is automatically done with the function "rewrite". Consequently, the usual Lisp affectation functions (set, setq, nextl, incr ...) can be used to assign the instance (and class) variables.

**ObjScheme:** an object is now anonymous and represented as a *compound-procedure* grouping together the environment of the instance variables (i_dico) and a lambda-expression:

```
; i_dico = < (metaisit . ?) (isit . ?) (i_variable_1 . ?) ... (i_variable_n . ? ) >
(define (self selector . args)
-1-   (apply (lookup selector (access isit i_dico))
-2-          (cons* self i_dico args)))
```

1. the "i_dico" is built as a Scheme environment by the "make-environment" primitive.

2. each method takes "i_dico" as argument allowing the explicit access to each field.

[6] We propose (cf. ObjScheme) to replace these "let" pseudo-variables bindings by real instance variables, defining OBJECT with *isit* and *metaisit* (facultative) as instance variables. Objects can also be anonymous by using structures in place of atoms (notice that naming classes is useful for documentation and debugging, therefore **CLASS** will own the new *name* instance variable).

# 5 Improvements and Possibilities

The $\boxed{\text{ObjVlisp}}$ model first advantage is UNIFORMITY. There is now only one kind of objects. This allows a simplification and reduction of concepts, which are thus more powerful and general. The second property is REFLEXIVITY which provides a language totally and uniformly accessible by the user. Finally, EXTENSIBILITY authorizes various applications and semantics modelizations. For example the study of inheritance strategies of such systems like Flavors [MW80], Smalltalk-80 [GR83] or Loops [BS83] are simulated by defining new metaclasses [Coi84a].

*The* $\boxed{\textit{ObjVlisp}}$ *project is part of the "O.O.P. Methodology, GRECO de Programmation" Research Group.*

# References


[BKK*85] G. Bobrow, K. Kahn, G. Kiczakes, L. Masinter, M. Stefik and F. Zdybel: COMMONLOOPS: Merging Common Lisp and Object-Oriented Programming (IJCAI85 draft), 16 August 1985.

[Bri85] J.P. Briot: *Les Métaclasses dans les langages Orientés Objets.* Vol. 2, p. 755-764, AFCET/INRIA, Grenoble, 27-29 Novembre 1985.

[BS83] D. Bobrow and M. Stefik: *The LOOPS Manual.* Xerox PARC, December 1983.

[CBS86] P. Cointe, J. P. Briot, and B. Serpette: The FORMES Language: a Musical Application of Object Oriented Concurrent Programming. In *Object Oriented Concurrent Programming*, A. Yonezawa & M. Tokoro ed., MIT Press, Cambridge, Mass., May 1986.

[CDH84] J. Chailloux, M. Devin, and J.M. Hullot: LeLisp a Portable and Efficient Lisp System. In *Conference Record of the 1984 ACM symposium on LISP and Functional Programming*, p. 113-123, ACM, Austin, Texas, 5-8 August 1984.

[Coi83] P. Cointe: A Vlisp implementation of Smalltalk-76. In *Integrated Interactive Computing Systems*, P. Degano & E. Sandewall ed., p. 89-102, North-Holland, Amsterdam, 1983.

[Coi84a] P. Cointe: *Implémentation et interprétation des langages objets, application aux langages Formes, ObjVlisp et Smalltalk (thèse d'Etat).* Technical Report 85-55, LITP/IRCAM, 17 Décembre 1984.

[Coi84b] P. Cointe: Une extension de Vlisp par les objets. *Science of Computer Programming*, Vol. 4, No 161, p. 291-322, 1984.

[dRS84] J. des Rivières and B. C. Smith: The implementation of proceduraly reflective languages. In *Conference Record of the 1984 ACM symposium on LISP and Functional Programming*, p. 331-347, ACM, Austin, Texas, 5-8 August 1984.

[GK76] A. Goldberg and A. Kay: *SMALLTALK-72 Instruction Manual.* Technical Report, Xerox PARC, Palo Alto, CA, March 1976.

[GR83] A. Goldberg and D. Robson: *SMALLTALK-80 The Language and its Implementation.* Addison-Wesley, Reading, 1983.

[Hof79] D. R. Hofstadter: *GOEDEL, ESCHER, BACH: an Eternal Golden Braid.* The Harvester Press, Stanford Terrace Publisher, 1979.

[Ing78] D. H. Ingalls: *The Smalltalk-76 Programming System Design and Implementation.* Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages, Tucson, January 1978.

[Kra83] G. Krasner: *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, Reading, 1983.

[Lie86] H. Lieberman: Delegation and Inheritance, two modular mechanisms. In *Conf. Record of the 3rd Workshop on OOP, AFCET/IRCAM*, J. Bezivin & P. Cointe ed., Bigre+Globule No 48, Centre Georges Pompidou, Paris, January 1986.

[MW80] D. A. Moon and D. Weinreb: *Flavors: Message Passing In The Lisp Machine.* Technical Report, MIT, Cambridge, Mass., November 1980.

[Yon86] A. Yonezawa: An approach to Object Oriented Concurrent Programming - A language ABCL. In *Conf. Record of the 3rd Workshop on OOP, AFCET/IRCAM*, J. Bezivin & P. Cointe ed., Bigre+Globule No 48, Centre Georges Pompidou, Paris, January 1986.

# Contents