A Uniform Model for Object-Oriented Languages Using the Class Abstraction

Jean-Pierre BRIOT & Pierre COINTE

LITP - Université Paris-VI & Rank Xerox France
4, place Jussieu & 12, place de l'Iris
75005 Paris & 92071 La Défense
briot@litp.ibp.fr & pc@litp.ibp.fr

in Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI'87), Vol. 1, pages 40–43, August 1987.

Abstract

One of the main goals of object-oriented languages (OOLs) is to unify their universe: *"every entity of the language is an object."* The *class* concept, however, usually does not follow this wish: a class being not a real object, i.e., created from a class. The *metaclass* concept introduced in Smalltalk-80, attempts to give classes a first class citizenship but complicates the instantiation scheme without solving its fundamental limitations: the only partial specification of the class at the metaclass level, and the fixed number of meta-levels.

Some more recent approaches, as in Loops and then CommonLoops, tend towards a better unification, but reveal the same limitations. We propose to go one step further and present an ultimate new model for an optimal simplification of the *class/object* concept, while keeping the *class* abstraction. In this model, implemented by ObjVlisp[1], every object is an instance of a class, and a class is a true object fully specified at the meta-level. A *metaclass* is a true class inheriting from another one. Because ObjVlisp is built on a reflective architecture, the user has uniform access and control to all the levels of the language: class level, metaclass level... He can add new levels and the metaclass links can be created indefinitely.

Finally, we suggest to clarify the Smalltalk terminology with a new definition for *class variables*: the instance variables of a class - treated as an object - become the class variables of this class, explicitly expressed at the metaclass level.

[1]This research was supported by the "Object group" of the Greco de Programmation, CNRS, France.

I Class versus Object

This paper deals with the instantiation mechanism used by object-oriented languages which organizes objects along the class abstraction.

A. Class and Instantiation

"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' " [7].

Object-oriented programming is built from the *class* model, introduced in the language Simula-67. In Simula, a *class* is a way to describe an abstract data structure. Active objects may be dynamically created from the class description by instantiating the variables specified by the class. Such objects are called *instances* of the class.

B. Is a Class an Object?

"With respect to Simula, Smalltalk abandons static scoping, to gain flexibility in interactive use, and strong typing, allowing it to implement system introspection and to introduce the notion of metaclasses" [5].

The *object* paradigm is set, and is now distinct from abstract data types systems. The Smalltalk language has evolved from Smalltalk-72 [8] to Smalltalk-80 in order to give the user some control on classes. Smalltalk-76 [10] is the first language to consider a class as an object itself, i.e., instance of a special class, called a *metaclass*. However, besides a uniformity wish[2], a class is not a true object. Its structure is not fully specified at the meta-level but remains at the implementation level. Only the behavior of classes, i.e., the way they react to message passing, is specified at the meta-level, through this unique metaclass, called `Class`.

C. Smalltalk-80' Metaclasses are Not a Full Answer

A unique metaclass imposes a unique behavior to all classes. For this reason, Smalltalk-80 [9] introduced the *metaclass* concept to allow distinct behaviors for different classes. Each class now has its own metaclass. This facility is mainly used to (re)define the instantiation method, to initialize class variables, or to hold predefined examples. However, Smalltalk-80 metaclasses are *implicit* (they are implicitly created from the class description) and *virtual* (they cannot be used as true classes, e.g., being explicitly instantiated). Consequently metaclasses introduce such a conceptual gap in the understanding of Smalltalk-80 that Borning proposes to drop them:

"In our empirical studies, metaclasses were regarded as the most significant barrier to learnability by both students and teachers. We propose that they be eliminated. We have explored various alternatives to metaclasses, such as the use of prototypes. However, for DeltaTalk we simply propose that the language reverts to the situation in Smalltalk-76. Every class would be instance of class Class" [3].

We do not agree with Borning's view. It seems to us that metaclasses do add great expressive power to OOLs and that it is really worthwhile to find a correct definition for them. We need metaclasses to develop friendly open-ended systems [6] and we believe that the fundamental problem with Smalltalk remains the impossibility to explicitly specify classes as instances of true classes. In order to solve this deficiency and to simplify the Smalltalk-80 model we propose to unify classes and metaclasses.

[2] *"One way of stating the Smalltalk philosophy is to choose a small number of general principles and apply them uniformly"* [11].

II Unification

We claim that a class must be an object with a first class citizenship allowing greater clarity and expressive power.

A. Is Every Object a Class?

There are still classes and non-class objects in our model. The “non-class” objects are called *terminal instances*, they are fully instantiated and are not abstractions like classes.

B. How Many Object Types?

There are two kinds of objects (*classes* and *terminal instances*). There is no type distinction however. A class and a terminal instance only differ through their respective classes. For instance, a class will accept the `new` message to create an instance of itself, but a terminal instance will reject it.

C. Metaclasses are True Classes

A class of classes is called a *metaclass*, it specifies the structure and the behavior of classes. The first primitive metaclass in the language is called `CLASS` and owns the `new` primitive method for instantiation. Any class declared as a subclass of `CLASS` inherits its ability to specify and control classes (e.g., the `new` method), and thus becomes a metaclass. Consequently, metaclasses are true classes and also true objects.

D. Creation of a Class

This unification induces a simplification of the instantiation and inheritance concepts but imposes that they be used simultaneously. We can create objects with inheritance (classes and metaclasses) or without inheritance (terminal instances). For instance a metaclass is created as the subclass of another one (as an “ultimate” subclass of `CLASS`).

The distinction between metaclasses, classes and terminal instances is only a consequence of inheritance and not a type distinction. There is now only one type of objects in the model.

III The Unified Model

A. Structure of an Object

A class describes the structure of a (potential) set of objects through an ordered collection of *instance variables*. The first instance variable - called `isit` - is automatically *inherited* from the `OBJECT` class and refers to the name of the class of the object (each object is the instance of a class).

As an example, the `POINT` class describing 2D points specifies the following instance variables : `<isit x y>`

An instance of this `POINT` class, e.g., the point 10@20, owns the values associated to the instance variables specified by its class: `<POINT 10 20>`

A set of procedures (called *methods*), usable by any of its instances, is also specified by the class as we see below.

B. Structure of a Class

If we want to define a class, we need to know the instance variables describing a class. They are specific to our model: `<isit name supers i_v methods>`

Because it is convenient to have named classes, `name` denotes the name of the class.

The list of the names of the direct superclasses from which the class will inherit is denoted by `supers`.

The list of instance variables that the class specifies is denoted by `i_v`.

The set of methods held by the class expressed in a P-list way[3], with pairs <*selector-name* . *λ-expression*>, is denoted by `methods`.

The `isit` instance variable belongs to the `OBJECT` class, the most general class in the model. In contrast to other usual instance variables, the corresponding value is automatically supplied when creating the object.

We can now describe the structure and the behavior of a class through this set of instance variables: a class has at last become a real object.

C. The POINT Example

To illustrate our model, we present its Lisp implementation called ObjVlisp. We define `POINT` as a subclass of the `OBJECT` class by instantiating the first metaclass `CLASS`. A CommonLisp syntax is used for specifying the values associated to the instance variables of the class-receiver :

```
(send CLASS 'new
      :name       'POINT
      :supers     '(OBJECT)
      :i_v        '(x y)
      :methods    '(x  (λ () x)
                    x: (λ (nx) (setq x nx)) ))
```

Then we create an instance of POINT, using the same `new` message : `(send POINT 'new :x 10 :y 20)`

IV Reflection

`CLASS` is the first primitive object. It will recursively create all other objects. `CLASS` needs to be an instance of some class as any object of the model. To prevent an infinite regression (we need the class of `CLASS`, and the class of this class...) the usual technique in OOLs is to model this infinite construct through recursion. We circularize the instantiation tree by adding a loop at its top.

The simplest way is to set this loop at the `CLASS` level, i.e., by declaring `CLASS` as its own instance (and class). (We will discuss the consequences of this choice in paragraph **C.**)

A. Self Pattern Matching of CLASS

The previous statement severely constrains the structure of `CLASS`. The instance variables specified by `CLASS` must match the corresponding values held by `CLASS` itself, as its own instance. Below are the instance variables and the associated values :

isit	name	supers	i_v	methods
CLASS	CLASS	(OBJECT)	(isit name supers i_v methods)	(new (λ ..))

Note that the value associated to the instance variable `i_v` is exactly the list of instance variables itself. This self pattern-matching illustrates the circular definition of `CLASS`.

B. The Golden Braid

In order to implement our model in a reflective way, we need a "bootstrap" [6]. We first create, on the Lisp level, a skeleton of `CLASS` owning the `new` method. Then we create, on the ObjVlisp level, the class `OBJECT`, root of the inheritance tree :

```
(send CLASS 'new
      :name       'OBJECT
```

[3]Note that for the efficiency purpose, the `methods` instance variable may represent any structure (hash-code table, package...) allowing an efficient method lookup. The P-list format was chosen in this paper for its simplicity.

```
:supers       '()
:i_v          '(isit)
:methods      '(class (λ () isit) ..))
```

Then we redefine CLASS by a self-instantiation using the values presented above :

```
(send CLASS 'new
      :name         'CLASS
      :supers       '(OBJECT)
      :i_v          '(name supers i_v methods)
      :methods      '(new (λ i_v*
                            (make-instance ...))  ))
```

After the bootstraping process, the system owns only the CLASS and OBJECT classes and the instantiation tree is exactly like the Smalltalk-76 one. But, as demonstrated in [4], the uniformity and the explicit definition of the objects CLASS and OBJECT open an immense variety of possibilities.

C. Unicity versus Genericity

The model we present is optimal in its simplicity and generality. The unicity of the `new` method reflects the unique type of objects: real instances of classes. On the other hand, because of inheritance there are two kinds of object creations. Thus the `new` method is not fully generic.

When creating an object, inheritance applies to classes but not to terminal instances. As a consequence, the `make-instance` primitive needs to discriminate between classes and terminal instances. In order to explicit these two ways of creating objects (with or without inheritance) and gain genericity of the `new` method, we may use another primitive object in the model, called METACLASS, owning the `new` method with inheritance (creating classes). Then CLASS will own another `new` method without inheritance (creating terminal instances). This second alternative is similar to the Loops kernel [1], but augmented with the full specification of classes at the meta-level. The disadvantage is the increased complexity necessary to gain genericity for the `new` method.

Our complete solution is by splitting the `new` method owned by CLASS in two methods: `basic_new`, owned by CLASS, which allocates the instance of a class, and a generic `initialize` method:

```
(send (send self 'basic_new) 'initialize i_v*)
```

By sending the `initialize` message to the newly created instance, we move the metaclass/class generator discrimination to the class/instance created object discrimination. The basic `initialize` method is owned by OBJECT and fills the slots with the values specified by the keywords. The `initialize` method owned by CLASS calls the former one and implements the inheritance of instance variables.

D. Indefinite Meta-Levels

We may extend the system with the same tools that were used to create it: instantiation and inheritance. For instance, we can specify and control two new levels of metaclasses by defining parametrized sets of points. We first define the SET metaclass which is a generator of sets, i.e., classes memorizing the list of their instances (`listOfInstances`). Then we create the MPOINT metaclass, as a subclass of SET and with a display character parameter (`c`). Its instance, the POINT@ class, is a set of @ points. (We may easily define other sets of points POINT#, POINT+... using the technique described in next section) :

```
(send CLASS 'new
     :name 'SET :supers '(CLASS) :i_v '(listOfInstances))
(send CLASS 'new
     :name 'MPOINT :supers '(SET) :i_v '(c))
(send MPOINT 'new
     :name 'POINT@ :supers '(OBJECT) :i_v '(x y) :c ''@'')
```

V Class Variable

We propose an alternative to the Smalltalk *class variables*. The principle is to extend the scope of the instance variables of a class to each of its instances. Then the class variables are defined at the metaclass level as simple instance variables of the class treated as an object.

A. The Polygon Problem

Let us develop the Polygon construction to illustrate this idea. The problem is to represent the (regular) Polygon abstraction and the Square and Hexagon sub-abstractions. The methodology of Smalltalk-80 leads to use the class hierarchy to define, first the `Polygon` class, then its `Square` and `Hexagon` subclasses.

B. The Smalltalk-80 Solution

Each polygon will be defined by its location (the first vertex) and the length of any of its sides. Consequently, `location` and `length` will be defined as the instance variables of `Polygon` treated as a class. The problem then, is to parametrize the number of sides: `4` for a square, `6` for a hexagon, `undef` for a polygon. If we define `nSides` as a class variable of `Polygon`, `nSides` will be inherited by `Triangle` and `Square` because they are sub-classes of `Polygon`.

But in Smalltalk the inheritance of class variables does not follow the inheritance of instance variables. Class variables are used to share knowledge between instances of a class hierarchy. For example, if the `new` method is redefined to add the newly created instance of a class inside a `Collection`'s class variable, the instances of its subclasses will also be memorized. In the same way, each square or hexagon would share the same number of sides!

Because class and metaclass hierarchies are parallel, the unique solution is to define `nSides` as an instance variable of `Polygon` treated as an object, i.e., at the "`Polygon class`" level. Nevertheless, to access the value of `nSides` from an instance of Polygon (or Square and Hexagon) we have to explicit at the metaclass level two "read-write" methods controlling this metavariable[4]. To define "6 sided" polygons we will use the transmission :

[4]Here are these two methods held by `Polygon class` :

```
Polygon class>>nSides
    ↑nSides
Polygon class>>initialize: numberOfSides
    ↑nSides← numberOfSides
```

Below is the display method drawing every class of polygons. Note the composed transmission `self class nSides` :

```
Polygon>>display
   | aPen |
   aPen← Pen new place: location.
   (self class nSides) timesRepeat:
         [aPen go: length ; turn: 360 // (self class nSides)]
```

```
Hexagon initialize: 6
```

C. The ObjVlisp Solution

The "Polygon scheme" is believed to be quite general when applying the class abstraction to the knowledge representation field. To capture it, we have decided to define the ObjVlisp class variables at the metaclass level, realizing a "global environment" shared by all the instances of a class and following the inheritance rules of instance variables. Unlike Smalltalk-80, our class variables are inherited but not shared by the subclasses. Consequently, here is our alternative version of the Polygon example :

```
(send CLASS 'new
      :name          'METAPOLYGON
      :supers        '(CLASS)
      :i_v           '(nSides) )

(send METAPOLYGON 'new
      :name          'POLYGON
      :supers        '(OBJECT)
      :i_v           '(location length)
      :methods       '(display (λ () ...)  ...)
      :nSides        'undef) )
```

`METAPOLYGON` is a subclass of `CLASS`, thus it is a metaclass. The creation of `POLYGON` explicits the instantiation of the class variable `nSides`. Then we can define new classes of polygons, with distinct values of `nSides`, by defining them as inheriting from `POLYGON`. As an example the `SQUARE` and `HEXAGON` objects are the classes of "4 sided" and "6 sided" polygons :

```
(send METAPOLYGON 'new
     :name 'SQUARE :supers '(POLYGON) :nSides 4)
(send METAPOLYGON 'new
     :name 'HEXAGON :supers '(POLYGON) :nSides 6)
(send SQUARE 'new
     :location (send POINT 'new :x 100 :y 200)
     :length 20)
```

Every object has access to its own environment as well as the environment of its class. Consequently `nSides` is bound at two levels: class and instance methods[5]. The previous instance of `SQUARE` has access to the bindings of the instance variables of its class with the associated values it owns, i.e.: `(isit . SQUARE)(location . 100@200)` `(length . 20)`. But it has also access to the bindings of the instance variables of its metaclass with the associated values that `SQUARE` owns, thus gaining the value of the (meta)instance variable: `(nSides . 4)`.

D. Towards a New Terminology

The problem with the terminology developed by Smalltalk-80 is the non-symmetry between the instance and the class levels. We agree with the *instance method* and *class*

[5]To illustrate this point, here is the definition of the `display` method held by `POLYGON`. `nSides` is now bound in this instance method :

```
(λ () (let ((aPen (send (send Pen 'new) 'place: location)))
  (repeat nSides
    (send aPen 'go: length) (send aPen 'turn: (/ 360 nSides)))))
```

method definitions because they respectively express the behaviors of the instances and then the behavior of a class as an object. On the other hand, we are confused by the *class variable* definition which does not define the field of a class as an object but defines a knowledge shared by all its instances.

We propose to keep the term *class variable* BUT to use it for a different meaning. A class variable becomes an instance variable of the class treated as an object. To rename the Smalltalk term *class variable*, we suggest the term *shared variable*.

VI Conclusion

We have presented here a new model for object-oriented programming. This model unifies *class* and *object* concepts. A class is now a true object, fully specified at its meta-level. The primitive metaclass of the model, called `CLASS`, is described and created in a circular way, as an instance of itself. This class is the root of the instantiation tree whose depth is potentially infinite and the user has now an uniform control on every meta-level. We concluded by proposing a new definition of the *class variable* concept compatible with the *class methods*.

Acknowledgments We thank Jean-Francois Perrot, Kris Van Marcke and Henry Lieberman for their helpful comments.

References

References

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

[2] 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.

[3] Borning A., O'Shea, A., DeltaTalk: An Empirically and Aesthetical Motivated Simplification of the Smalltalk-80 Language, *ECOOP'87, to appear in Springer Verlag, P. Cointe & H. Lieberman ed.,* Paris, France, 15-17 June 1987.

[4] Briot, J-P., Cointe, P., The ObjVlisp Model: Definition of a Uniform, Reflexive and Extensible Object-Oriented Language, *ECAI'86*, pp. 270-277, Brighton, UK, 21-25 June 1986.

[5] Cardelli, L., A Semantics of Multiple Inheritance, *Bell Laboratories*, Murray Hill NJ, USA, 1984.

[6] Cointe, P., The ObjVlisp Kernel: A Reflective Lisp Architecture to Define a Uniform Object-Oriented System, *Proc. of the Workshop on Meta-Level Architectures and Reflection, to appear in North Holland, P. Maes & D. Nardi ed.*, Alghero, Italy, 27-30 October 1986.

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

[8] Goldberg, A., Kay, A., Smalltalk-72 Instruction Manual, *Research Report SSL 76-6*, Xerox PARC, Palo Alto CA, USA, March 1976.

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

[10] Ingalls, D.H., The Smalltalk-76 Programming System Design and Implementation, *5th ACM Symposium on POPL*, pp. 9-15, Tucson AZ, USA, January 1978.

[11] Krasner, G., Smalltalk-80 - Bits of History - Words of Advice, Addison-Wesley, Reading MA, USA, 1983.

Contents