A Practical use of Metaclasses

François Pachet

LAFORIA
Université Paris VI
Tour 45-46
4, Place Jussieu
75252 Paris Cedex 05
fdp@litp.ibp.fr

Juillet 1989

abstract

In Object-Oriented Programming languages, metaclasses, when they exist and when the programmer have access to them, are mostly used for implementation sake [5] [6] or to ensure the uniformity of the underlying model [2] [3] [4]. They are hardly ever used for design purposes, i.e. for describing specific behaviors of an application's classes, apart from basic initialization behaviors. They are rather seen as a mean of language description or as a tool for hypothetic demanding programmers who would want to pervert the system's kernel to fit their extravagant needs.
We introduce and comment here two examples taken from a real-world application, part of the SAGESSE project[1] and discuss their various possible implementations in Smalltalk. We show how the implementations using metaclasses are "better" than those simply using the class hierarchy and instance methods.
We finally draw some conclusions about Object Oriented design methodology and the good use of metaclasses.

1 Introduction

The definition of metaclasses derives very simply from the definition of classes (i.e. the description of the instances structure and behavior) and the principle of uniformity (i.e every object is an instance of a class) : Thus, a metaclass is simply a class whose instances are classes.

In Smalltalk, classes are defined by subclassing as opposed to instantiation. Metaclasses are thus created automatically by the system. This is a limitation, since all metaclasses are instances of the Abstract class Metaclass, and the environment does not provide any documented means of changing this, although it has been shown that minor kernel additions could provide explicit creation of metaclasses [3].
Nevertheless, metaclasses are accessible, and it is possible to specify structures (variable names) and behavior (methods) for them.
Browsing the Smalltalk system's class descriptions and methods, we find two main uses of metaclasses, apart from general class descriptions, as found in Behavior subclasses:

- Class initialization :
  methods found in metaclasses allow initializations of constant tables, class variables or global variables (controllers initializing menus, Character defining ascii constants) .[2]
- Instance creation methods[3]:
  A great amount of classes redefine their instance creation methods, generally in order to specialize initializations (i.e. View, Point, metaclasses ...).

In particular no system metaclass has instance variable : their specificity is defined only by methods.
This is clearly a mistake in a number of cases. One of the most obvious is the way menus are defined in controllers, namely StandardSystemController : the yellow and blue button menus are

[1] A Simulation System for decision-making in crisis situation, for the French Ministère de l'Intérieur

[2] It is interesting to note that the Smalltalk methods that implement the fileOut mechanism automatically insert initialization messages to the metaclasses that implement an *initialize* method. This tends to show that these methods have only a side-effect role in an application description.

[3] The only serious reason for a class to have a name is the fact that one has to create instances out of it by sending it nominatively a message

defined as clas variables, thus accessible by al sub-instances of the class. When a subclass redefines those menus, it has to declare two additional class variables, and change the initialization message used at instance creation time. Using class instance variables would have been more coherent (the actual menus would not have been inherited by the subclasses), more readable (redefining carelessly those menus in a subclass would mess up all existing controllers), and less space consuming.
Using two real-world examples, we advocate here a more general use of system description using metaclasses, to achieve more coherence and readability.

2 First example : A declarative solution for remanent objects

In our application, we were faced with the problem of saving objects that represent networks (in our case : a region with all kinds of nodes (cities, villages) and links (roads, motorways) between them). In such cases, the default mechanism provided by the Smalltalk environment (namely the method `storeOn:` [1]) would not work since networks are thoroughly recursive objects :

- as soon as the network is not acyclic, there are nodes pointing towards nodes that point to themselves.
- the links between the nodes are objects themselves (instances of various subclasses of Link such as Road, Lane, Railway ...), so a node will point to a link that points back to itself.

There are solutions to this [11] which consist in writing a method `storeOn:using:` (and conversely `readStructureFrom:using:`) that simply keeps track of the various objects encountered, in order to avoid recursion, and to preserve shared objects [2].
The method `storeOn:using:` is an instance method of `Object`, and the method `readStructureFrom:using` is a method of `Object class`. Subclasses redefine those methods for specific behavior (e.g. `Number`).

This solution works but has two major drawbacks :

1 - It saves possibly unwanted objects, since it saves the whole network of objects pointed by the saved objects.
2 - At reconstruction time, specific initialization methods could be required. The current reconstruction consists in an allocation (creation of an instance of the class with a `basicNew` (or `basicNew:`)) followed by assignements of instance variables to the saved objects (with `instVarAt:put:` or `basicAt:Put:`). It may lead to strange results when applied to complex objects such as views.

In fact it assumes that the structure that must be saved is the actual structure defined by the class (i.e. all instance variables, including the superclasses's ones).
The idea is then to have the possibility of defining, for a given class, which instance variables will

---

[1] We should say the methods *storeOn:* since *storeOn:* is redefined by most of the system classes. By using the singular, we certainly mean a kind of abstraction of these methods ?

[2] an object having two instance variables x and y, whose values are the *same* object, say a Point, is not circular, but the evaluation of the result of the standard storeOn: method sent to this object will create a object whose x and y will point to physically *different*, though identical, instances of Points.

have to be saved, and how the some others will be reconstructed.
We show in the next three sections three possible implementations of this feature, respectively based on a class method, an instance method and finally a metaclass structure.

2.1 a class method implementation

The first solution that comes to mind is to simply write a Class method, say `goodFields`, that returns the list of the instance variable names liable to be saved. A default method in `Object` class (or `Class`, or `Behavior`) would return the set of all instance variable names :

```
!Object class methodsFor:
'saving'!
goodFields
        ^allInstVarNames
```

And any class needing to specify the fields to be saved would redefine this method.
For example, for a class named `Network` holding the following instance variables :

```
Network

instanceVariableNames:
'nodes links listOfProcesses
currentState'
plus superclasses variables
names
```

We could decide to save only the topologic fields i.e. nodes and links

```
!Network class  methodsFor:
'saving'!
goodFields
        ^#(nodes links)
```

This solution works, but is not satisfactory, since it lies on the Smalltalk specificity i.e.: all metaclasses have only one instance. Using this particularity, it is possible to specify class methods specific to the metaclass's sole instance. In a more general framework, where metaclasses could have several instances, this methodology would not hold any longer.

2.2 a solution based on an instance method.

One may then want just to implement the method as an instance method, since the saving method is itself an instance method. But this is not satisfactory either, since the class then has no natural way of knowing which of its instance variables are saveable or not.
Clearly, the information provided by `goodFields` is an information about the *structure* of the class (as well as the instance variable names), and thus must be specified by the metaclass. But this information must be *specific* to the class.
The only coherent implementation is thus described by the next section.

2.3 a metaclass instance variable.

`goodFields` must be an instance variable of the metaclass and thus its value will be specific to the class. In order to accomplish this, we define a root class, say `RemanClass`, whose metaclass (`RemanClass class`) defines an instance variable `goodField`:

```
RemanClass class

instanceVariableNames:
'goodFields'
```

At class creation time, the value of `goodFields` will be specifed as a parameter of an appropriate subclass creation method, for example :

```
!RemanClass class
methodsFor: 'sub class
creation'!

subclass:           s
instanceVariableNames: i
goodFields:         g
```

```
classVariableNames:    c
poolDictionaries:      p
category: c
|newClass|
newClass := super
subclass: s
instanceVariableNames: i
classVariableNames:    c
poolDictionaries:      p
category: c.
newClass goodFields:
(Scanner new scanFieldNames:
g).
^newClass
```

Access methods for the new variable will have to be defined as follows :

```
!RemanClass class
methodsFor: 'access'!

goodFields: g
        goodFields := g

goodFields
        ^goodFields
```

Then, a *default behavior* would have to be defined in Object class as in the first solutions :

```
!Object class methodsFor:
'remanence'!
goodFields
        ^allInstVarNames
```

The correct implementation is thus acheived by using a metaclass structure because it prevents from basing the implementation on the Smalltalk sole-instance metaclass mechanism and locates class structure information at the class level.

3 Second Example : Implementing a save/restore facility for object's states.

We propose here to describe a feature allowing objects to dynamically save and recover their states (basically the values of their fields).

Here the idea is to implement a save/restore mechanism in order to have a backtraking facility on the state of objects.

This is very useful in number of cases. Two different uses can be done of this feature, corresponding to distinct semantics :

> **local use** (bactrack on one instance): in optimization computations for instance, one may want to know the "hypothetic" state of a particular object after some computation has been made, without actually change its "real" state.
>
> **global use** (backtrack on all instances of a class) : it is very useful to be able to globally save/retrieve values for all instances of given classes, in order to simulate the notion of environment, especially in simulation systems[1].

Since we are dealing with Object-Oriented languages, which highly advocate encapsulation, the best place to put the saved data is in the objects themselves: objects will encapsulate their data ("real values" of the instance variables) as well as their old states.
To do so, we simply create extra instance variables which will point to stacks, for which usual push/pop operations are possible. Once again each class will have to specify which instance variables need to be stacked in a way or another.

3.1 the straightforward solution

A root class SimulObject is created to avoid redefinition of the class Object.

[1] In SAGESSE, we needed to visualize the application after decisions had been taken by the user, and then be able to come back to previous states.

Two methods are implemented in the root class SimulObject , which will save/restore the values of the specified variables as follow :

```
!SimulObject methodsFor:
'backtracking'!

save
self class simulVariables
do:
[:s|
(self instVarAt: (self class
allInstVarNames indexOf:
('stack',s))) push: (self
instVarAt: (self class
allInstVarNames indexOf: s)]
```

and similarly :

```
restore
self simulVariables do:
[:s|
(self instVarAt: (self
class allInstVarNames
indexOf: s)) push:
(self instVarAt: (self
class allInstVarNames
indexOf:('stack',s))]
```

One could also implement class methods for global use:

```
!SimulObject class
methodsFor: 'global
backtracking'!

saveAll
self allInstances do: [:x| x
save]

restoreAll
self allInstances do: [:x| x
restore]
```

Here again, the only place to put the information simulVariable is in an instance variable of the metaclass SimulObject class.

```
SimulObject class

instanceVariableNames:
'simulVariables'
```

Then a new sublass creation method is implemented in the metaclass SimulObject class :

```
!SimulObject class
methodsFor: 'sub class
creation'!

subclass: s
instanceVariableNames: i
simulVariableNames: s
classVariableNames: c
poolDictionaries: p
category: c
|string stackVariables
simulVariables|
string := i,s.
simulVariables := (Scanner
new scanFieldNames: s).
stackVariables :=
simulVariables
collect: [sv| stackVar :=
'stackOf',sv.
string :=
string,' ',stackVar.
stackVar].
^(self subclass: s
instanceVariableNames:
string classVariableNames: c
poolDictionaries: p
category: c) simulVariables:
simulVariables;yourself
```

This way, any class defined as a subclass of SimulObject can specify at creation time a set of instance variables for which corresponding stack variables will be created[1].

Example :

```
SimulObject subclass: #Foo

instanceVariableNames: 'x y'
 simulVariableNames: 'a b'
 classVariableNames: ''
 poolDictionary: ''
 category: 'test'

Foo allInstVarNames -> (x y
a b stackOfa stackOfb)
```

Assuming access methods are defined :

```
(Foo new) a: 1; save; a: 2;
save; a: 3; restore;
restore; a -> 1
```

[1] to be fully integrated in the environment, the methods definition and copyForValidation must be also changed to handle the extra class instance variable

This solution would work in Smalltalk, but is not satisfactory because the definition of this ability to declare and save/restore some instance variables is spread over both the class (save/restore methods) and the metaclass (creation method, class instance variable access method).

3.2 the solution using (almost) only metaclasses

A root class SimulObject is created, not in order to be able to depict some new structure or method, but because (in Smalltalk) one cannot create metaclasses by instantiation.

The two save/restore methods are implemented in the root metaclass SimulObject class, as follow :

```
!SimulObject class methodsFor: 'instance backtracking'!

save: x
self simulVariables do:
[:s| (x instVarAt: (self allInstVarNames indexOf: ('stack',s)))
      push: (x instVarAt: (self allInstVarNames indexOf: s))]
```

and similarly :

```
restore: x
self simulVariables do:
[:s| (x instVarAt: (self allInstVarNames indexOf: s))
      push: (x instVarAt: (self allInstVarNames indexOf: ('stack',s))]
```

and the generic methods :

```
saveAll
self allInstances do: [:x| self save: x]

restoreAll
self allInstances do: [:x| self restore: x]
```

The metaclass SimulObject class defines an instance variable and a creation method as in the first solution. But now, only the metaclass SimulObject class has been defined and the class SimulObject is only here in order to access its metaclass.
The only remaining instance methods are the save/restore for instances, which will only call class methods :

```
!SimulObject methodsFor: 'instance backtracking'!

save
  self class save: self

restore
  self class restore: self
```

Thus, once again, the correct implementation is acheived by using appropriate metaclass structure and behavior, and by clearing the class level.

3.3 towards better solutions

An other solution would consist in using variable subclasses. The room allocated for indexed element would contain the stacks, making them less "visible" than extra instance variables.
The implementation would then consist in changing the subclass creation method in order to create variable subclasses instead of standard subclasses, and then to change the save/restore class methods, so that they store and retreive values to and from the indexed locations of the instance.

The ideal solution would consist in defining some specific allocation method (basicNew) that would allocate extra room for the stacks, with specific behavior to access them. The task is definitely not easy within the actual Smalltalk environment, because of the difficulty to control metaclass creation, and because allocation methods are primitives, hidden to the user.

4 Conclusion

Other works in Object Oriented languages show that metaclasses are helpful as soon as the system description is not trivial (defining tokens behavior in Rete network [8], implementing part-whole hierarchy [10]).
Works on Object-Oriented methodology have demonstrated the need for careful and coherent programming, emphasising on the notions of encapsulation [12], reusability [9], safety, or portability.
Metaclasses are never mentionned as a means of ensuring code readability, reusability or portability.
Metaclasses play exactly the same role regarding classes, that classes play towards their instances : if they are absent or not used, then this role has to be played by the class themselves, in an acrobatic fashion. Programming without them is most of the time possible, and even advocated [1], but, as soon as the classes have specific behaviors, leads to programming techniques that make classes looking like ad hoc combinations of methods rather than clean abstractions of realities.

We advocate here a use of metaclasses similar to the uses of classes : abstract definition of entities having structures **and** behavior.

This correct use of metaclasses should avoid two pitfalls :

1 - Replacement of structures by ad hoc behavior.

The problem holds for classes as well as for metaclasses, as we can see on a classical example :
Suppose the class Human, for which one wants to define the property of being mortal.

The simplest way to do so is by defining a method :

```
!Human methodsFor:
'testing'!
mortal
  ^true
```

the coherent way would be to define mortal as a part of the structure of Human, and to have methods to cater for the value of mortal, (e.g. initialize):

```
Human
  instanceVariableNames
:'mortal'
```

```
!Human methodsFor:
'initialize'!
initialize
  mortal :=true
```

and simply an access method to answer the frightening question :

```
!Human methodsFor:
'testing'!
mortal
  ^mortal
```

It is clear here that Smalltalk (as most of the Object Oriented languages) provides means of turning around "structured" definitions by using methods and inheritance. But this is in a way a trick : in the first solution, subclasses of Human would not really inherit from the property of being mortal (or not), but would inherit from the behavior defined by the method mortal.

2 - Mixing class and metaclass levels.

Inheritance in Smalltalk, hides the need for metaclasses careful definition, because of the parallel inheritance of metaclasses, but defining a particular behavior for a class should be independant of the class itself, in order to be really reusable.
Let C be a class and M its metaclass. The wanted behavior is defined both by C and M behaviors as in the first solution of the last example.

Then, suppose one wants to have a class C2 having similar behavior than C.
One just needs to create it as a subclass of C, as both C and M structures and behavior will be inherited.
In a more general language (CLOS, Loops, Classtalk), where the user has control over both classes superclasses, *and* metaclasses, then the behavior one wants to inherit from has to be defined only in the metaclass, and not simulated by means of ad hoc instance methods, otherwise, C2 would be forced to inherit from C, which is a big constraint.
This is also particulary important with regards to readability and portability : finding the specification of a particular class's behavior is easy if it is defined entirely at the metaclass level. It is a lot more difficult to do so if this specification is spread between the class (and its subclasses) and the associated metaclasses.

The two previous examples make use of the ability of metaclasses to describe classes structures (instance variables) and behavior (methods) in a coherent way.
Besides from giving a more coherent implementation of the notions, this use is more declarative and more readable : the only place to put specifications of classes is their metaclasses.
The Smalltalk environment is not really made for this (the user has no control on the creation of metaclasses, and the use of class instance variable is not documented) but minor additions to the kernel allow extensive and free use of metaclasses [3].
Although other problems may arise when intensive and careless use of metaclasses is made [7], they give a powerful and coherent means for specifying high-level descriptions in a readable and portable way.

Aknowledgment

I whish to thank J.F. Perrot for his support and N. Graubé for his fruitful remarks.

References

[1] Borning A., O'Shea T.
Deltatalk: An Empirically and Aesthetically motivated Simplification of the Smalltalk-80 Language.
Proceedings of ECOOP'87.

[2] Briot, J-P., Cointe P.
A Uniform model for Object-Oriented Languages Using the Class Abstraction. Proceedings of IJCAI 87, pp 40-43, Milan, Italy, August 1987.

[3] Briot, J-P., Cointe P.
Programming with ObjVlisp Metaclasses in Smalltalk-80.
Proceedings of OOPSLA '89

[4] Cointe P.
Metaclasses are first class: the ObjVlisp model, OOPLSA '87, Orlando, Florida, USA October 87.

[5] 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 88.

[6] Graube N.
Reflexive Architecture: From ObjVlisp to CLOS, ECOOP '88, Lecture notes in Computer science, Vol. 322, pp 110-127, Oslo, Norway, August 88.

[7] Graube N.
Metaclass compatibility.
Proceedings of OOPSLA '89

[8] Laursen
Opus : a Smalltalk production system.
Proceedings of OOPSLA '86.

[9] Lieberherr K., Holand I., Riel A.
Object-Oriented Programming : An Objective Sense of Style.
Proceedings of the OOPSLA '88

[10] Malenfant J., & al.
ObjVProlog : Metaclasses in Logic.
Proceedings of the ECOOP '89

[11] Vegdahl, Steven R.
Moving Structures between images. Proceedings of the OOPSLA '86

[12] Wirfs-Brock A., Wilkerson B.
Variable limit reusability *in* journal of Object-Oriented programming, pp 34-40. May/June 1989, Vol 2, N° 1. SIGS Publication, New York.