Metaclasses are First Class : the ObjVlisp Model

Pierre Cointe
Rank Xerox & LITP

RXF: DRBI, 12 Place de l'Iris - Cedex 38, 92071 La défense
LITP: Université Paris-6, 4 place Jussieu, Tour 55-65, 75223 Paris
Email: ...!seismo!inria!litp!pc cointe@inria.inria.fr.uucp

Abstract

This paper shows how an attempt at a uniform and reflective definition resulted in an open-ended system supporting ObjVlisp, which we use to simulate object-oriented language extensions.

We propose to unify Smalltalk classes and their terminal instances. This unification allows us to treat a class as a "first class citizen", to give a circular definition of the first metaclass, to access to the metaclass level and finally to control the instantiation link. Because each object is an instance of another one and because a metaclass is a real class inheriting from another one, the metaclass links can be created indefinitely.

This uniformity allows us to define the class variables at the metalevel thus suppressing the Smalltalk-80 ambiguity between class variables and instance variables: in our model the instance variables of a class are the class variables of its instances.

1 The Instantiation Mechanism

1.1 Classes & Metaclasses

We focus on the instantiation mechanism of object-oriented languages which organizes objects in taxonomies along the class abstraction. Let us recall that the class concept invented by Simula and reimplemented by Smalltalk-72 is used to express the behavior of a set of objects which share the same semantics operating on the same attributes. This approach considers a class as a mould, manufacturing pieces called its instances. Alternatives to the class model - allowing other organizations of knowledge - are well known, for instance Hewitt's actor model. An actor describes its own structure and exists without a class. Defining generator actors, using a copy mechanism [9] or designing a delegation mechanism [15] are other themes developed by O.O programming.

"One way of stating the Smalltalk philosophy is to choose a small number of general principles and apply them uniformly" [14].

In most common class-oriented languages, despite Krasner's uniformity principle, a class is not a REAL object. Some of them however, like Loops [1], Smalltalk-80 [11], CommonLoops [2] and CLOS [3] introduced the metaclass concept to provide greater abstraction by allowing the description of a class by another class.

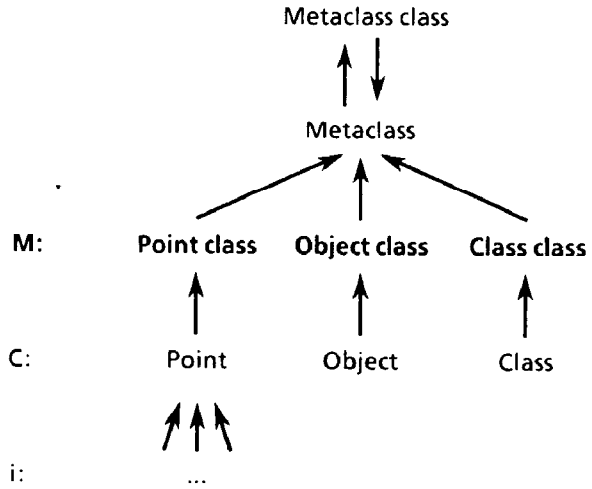
1.2 Smalltalk-80

"The primary role of a metaclass in the Smalltalk-80 system is to provide protocol for initializing class variables and for creating initialized instances of the metaclass's sole instance" (P. 287 [11]).

Smalltalk-80 uses the metaclass level facility to define the behavior of a class as the behavior of a regular object reacting to message passing (1). The role of a metaclass is to (re)define the instantiation method (3,4), to control the class variables initialization (2) or to explicitly explain the semantics of a class by predefined examples (5):

(1) Point class
(2) DeductibleHistory initialize
(3) a_point$_1$← Point new
(4) a_point$_2$← Point x: 20 y: 40
(5) Pen example

However, a Smalltalk metaclass is not a ordinary class but an anonymous one (accessible by the unary selector "class"), and which cannot be defined explicitly by users. This metaclass which supports the definition of the class instance variables and the class methods cannot exist without the class that is its only instance. Conversely, Smalltalk-80 associates a private metaclass to each class being created. Two classes cannot share the same metaclass. The next figure summarizes the Smalltalk-80 instantiation hierarchy :



Because every metaclass is automatically an instance of the `Metaclass` class, metaclasses are not true classes, the number of metalevels is fixed and the metalink cannot be created indefinitely [17].

From an inheritance point of view, the metaclass inheritance is also implicitly fixed by the system. The hierarchy of the metaclasses is parallel to the hierarchy of the classes. Because `Object` has no super class, the rule keeping a parallel hierarchy between classes and metaclasses, does not apply, and `Object class` is a subclass of the abstract class `Class`. Then, all Smalltalk metaclasses are subclasses of `Object class`, itself a subclass of `Class` :

```
Object ()
  Point (x y)
  Behavior (superclass methodDict format subclasses)
    ClassDescription (instanceVariables organization)
      Metaclass (thisClass)
      Class (name classPool sharedPools)
        <all metaclasses>
        Object class ()
          Point class ()
          Behavior class ()
          ...
```

Consequently, a metaclass cannot be defined ex nihilo as the subclass of a chosen class. This inheritance tree establishes the dichotomy between classes prototyped by the class `Class` and the metaclasses prototyped by the class `Metaclass` (even if they are both subclasses of `ClassDescription` and `Behavior`). These limitations, and the fact that a metaclass cannot exist without its class, introduce a first boundary between the implementor - who controls the metaclass level - and the user - who only has access to the class level. A second boundary is made apparent by the introduction of the `instance method` and `class method` terminologies (cf. the instance class SwitchView of the browser).

1.3 Loops

"For some special cases, the user may want to have more control over the creation of instances. For example, Loops itself uses different Lisp data types to represent classes and instances. The New message for classes is fielded by their metaclass, usually the object MetaClass. This section shows how to create a metaclass. Any metaclass should have Class has one of its super classes and MetaClass as its metaclass. The easiest way to create a new metaclass is to send a New message to MetaClass as follows : (← ($ Metaclass) New metaCLassName '(Class))" (P. 86 [1]).

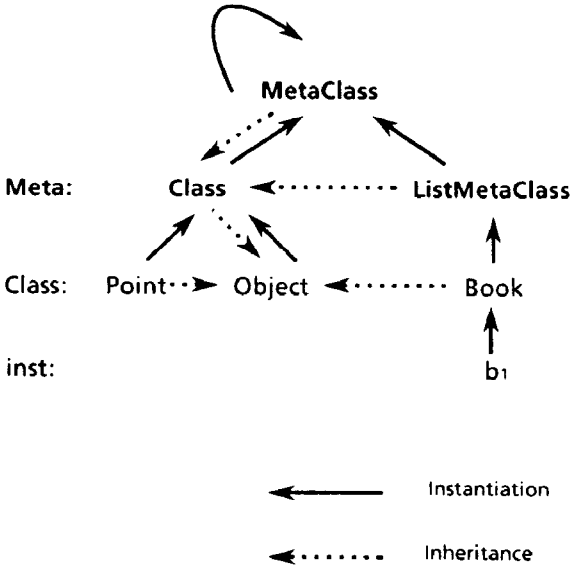
The Loops scheme for metalevels is close to the Smalltalk scheme. The basic idea is to introduce three levels corresponding to three kinds of object: instances, classes and metaclasses. This scheme is built on the "Golden Braid" `Object`, `Class` and `MetaClass` (cf. P. 113 [1]);

- `MetaClass` is the class which holds the default behavior for metaclasses as objects, it is the metaclass of all other metaclasses and its own metaclass. `MetaClass` holds the `New` method which creates class data-types.
- `Class` is the class which holds the default behavior for classes as objects. `Class` is the default metaclass for all classes. It holds the `New` method which creates instance data-types. Consequently, if `Class` is not the metaclass for a class, it must be on the supers list of that metaclass (which inherits its new method). According to this rule `Class` is a super of `Metaclass`.
- `Object` is the class which holds the default behavior for all instances. Consequently, `Object` is the root of the inheritance tree.

These three classes are used to create new metaclasses (1), new classes (2 3) and new instances (4 5) :

```
(1) (← ($ Metaclass) New 'ListMetaClass '(Class))
(2) (← ($ Class) New 'Point '(Object))
(3) (← ($ ListMetaClass) New 'Book)
(4) (← ($ Point) New 'a_point1)
(5) (← ($ Book) New 'b1)
```

The next figure summarizes the instantiation and subclass links provided by Loops (the black arrow means `instanceOf`, the shaded arrow means `subclassOf`) :



Unlike Smalltalk-80, a metaclass can be created explicitly as the subclass of another one but must be an instance of `MetaClass`. This last condition fixes the depth of the instantiation tree and leads the Loops implementors to use a non uniform representation for classes and terminal objects. On the other hand, the Loops manual does not express any circular definition of `MetaClass` as it would be suggested by its self-instantiation.

1.4 Unification

To suppress the gap between class and object, we propose a unification of the metaclass, class and object concepts. We claim that a class must be an object defined by a real class allowing greater clarity and expressive power.

The reverse question is "Is every object a class? ". The answer is no : some objects are only instances of a class and do not define a model. An instance of a Point class, e.g. an object $\texttt{a\_point}_1$, or an instance of the Number class, e.g. 3, are such non-instantiable objects. We call them terminal instances.

Thus we consider only one kind of object, without distinctions of structure or type between classes and terminal instances (non-class). 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 class. Otherwise it is a terminal instance. A metaclass is simply a class which instantiates 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 discrimination between metaclasses, classes and terminal instances[1] is only a consequence of inheritance and not a type distinction. We can distinguish between class and non-class objects, however the ObjVlisp model takes into account only one type of object.

This unification simplifies the instantiation and inheritance concepts, using them simultaneously : for example, a metaclass must be created as the subclass of another one (as an "ultimate" subclass of Class).

2 The ObjVlisp Model

Historically, the ObjVlisp model comes from our work on Smalltalk-76 [8]. Our wish is to present a synthesis, using operational semantics expressed in Lisp. We present here the reflective version which integrates the previous unification and gives a good solution to the problem of the *<class , instance>* dichotomy.

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

2.1 ObjVlisp in six Postulates

Following the classical presentation of Smalltalk-76 [13], six postulates fully describe the ObjVlisp model :

P1: An object represents a piece of knowledge and a set of capabilities :

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

P2: the only protocol to activate an object is message passing : a message specifies which procedure to apply (denoted by its name, the `selector`), and its arguments :

$$\boxed{(\text{send object } \texttt{selector } \text{Args}_2 \ldots \text{Args}_n)}$$

P3: every object belongs to a class that specifies its data (attributes called fields) 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, instantiated by another class, called its metaclass. Consequently (P3), to each class is associated a metaclass which describes its behavior as an object. The initial primitive metaclass is the class **Class**, built as its own instance.

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

P6: If the instance variables owned by an object define a local environment, there are also class variables defining a global environment shared by all the instances of a same class. These class variables are defined at the metaclass level according to the following equation :

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

2.2 Classes and objects

Structure of an object

The postulates P1 & P3 & P6 define an object as a "chunk" of knowledge and actions whose structure is defined by its class. More precisely:

Fields : are the set of variables defining the environment of the object;

a) **instance variables :** this first environment is organized as a "dictionary" split into two isomorphic parts :

1. the set of instance variables specified by the object's class,
2. 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; consequently an object cannot exist without its class. These two sets [2] are ordered - at creation time - by the inheritance rules defined on the superclasses components. In particular, to store the name of its class, each object holds as first instance variable one named `isit`. Each object holds also the `self` "pseudo instance variable" dynamically (at runtime) bound to the object itself when it receives a message. These two pseudo variables are respectively analogs to `isit` and `self` in Smalltalk-72 [10].

b) **class variables** : Smalltalk-80 class variables are accessible to both the class (via instance methods) and its metaclass (via class methods). Similarly, ObjVlisp instance variables of a class (defined as an object) are accessible to both its own methods and its metaclass methods. Consequently, the instance variables of a class are also the class variables of its instances, defining global environment at the metaclass level.

Methods : The methods define the procedures shared by all the instances of a class and owned by the class. To realize the unification between class and instance, we represent the method environment as a particular instance variable of its metaclass; the methods dictionary of a class is the value associated with a specific instance variable called methods. As a common object, a class is defined by its class and the values of the associated instance variables.

Structure of a class

As an object, a class owns also the `isit` instance variable inherited from Object (cf. 3.2). This variable is bound to the name of the metaclass when the class is created. Because a class is also a generator of objects , we have to introduce the minimal set of instance variables describing a class. Four explicit instance variables are owned by **Class** as the primitive metaclass:

1. `name` : the name of the class, which means that each class is a non-anonymous object,
2. `supers` : the list of the direct superclasses of the class,
3. `i_v` : the list of instance variables that the class specifies,
4. `methods` : the method-dictionary e.g. the list of methods held by the class expressed as a "P-list", with pairs <selector , λ-expression>.

[2] Each object is implemented as a pointer to an abstract structure (for example a list, a vector, a hash-table or a Lisp structure) which must be isomorphic to the list of instance variables held by its class :

object = #(class-name i_v_2* ... i_v_n*)
i_v [class-name] instantiate #(class-name i_v_2* ... i_v_n*)

Instantiation of a class

Unlike the Smalltalk-80 and Loops systems, ObjVlisp uses only one method to create an object which can be a terminal instance or a class.

Basicnew : this method is owned by the metaclass **Class** and uses the `makeInstance` primitive of the virtual machine [3] as expressed by the circular definition of **Class** (cf 3.1). This method implements only the allocation of the structure with the `nil` default-value for each instance variable :

(send Aclass 'basicnew) ⇒ #(Aclass nil ... nil)

New : to allocate a new object and to initialize its instance variables, ObjVlisp uses the `new` method, owned by **Class**. This `new` method has two effects: to allocate a new object and to give an initial value to each instance variable. To distinguish these two functions, `new` composes the `basicnew` method with one of the two `initialize` methods defined respectively in **Class** and **Object**.

Consequently, the instantiation semantic and syntax are totally uniform: the `new` message sent to a class always receives as arguments the values related to the instance variable specified by the receiver class and creates a new instance built on the class model. To allow more expressive power each argument of the `new` message must be prefixed by a keyword (for example :name for an instance variable called *name*) denoting the instance variable receiving the associated value:

(send Aclass 'new :i_v_2 i_v_2 ... :i_v_n i_v_n) ⇒
#(Aclass i_v_2* ... i_v_n*)

Examples : we define the class Point by instantiating the metaclass **Class**; the receiver (here **Class**) specifies the name of the model (the value of the implicit `isit` instance variable) and the values associated to the four instance variables of **Class** must be expressed :

```
(send Class 'new
  :name       'Point
  :supers     '(Object)
  :i_v        '(x y)
  :methods    '( x        (λ () x)
                 x:       (λ (nv) (setq x nv) self)
                 init     (λ () (setq x 40 y 12) self)
                 display  (λ ()
                            (format ()
                              (catenate "~" x "D")
                              "*") ) ) )
```

Then we create instances of Point, using the same `new` message[4] :

```
(setq a_point1 (send Point 'new :x 20 :y 30))
(setq a_point2 (send Point 'new :y 30 :x 20))
(setq a_point3 (send Point 'new))
(setq a_point4 (send Point 'new :x nil :y nil))
(setq a_point5 (send Point 'basicnew))
```

[3] The `makeInstance` function creates a new object when receiving as argument the name of its class :

```
(defun makeInstance (aclass)
  (tcons aclass (makelist (1- (length (send aclass 'i_v))) nil)))
```

The auto-quoted keywords suppress the order of instance variables values e.g. the two objects a_point_1 and a_point_2 are equal. Keywords may be also omitted, in this case the associated instance variable is bound to the `nil` default value (e.g. a_point_3, a_point_4 and a_point_5 are equal).

Obviously, the user can modify the behaviour of the `new` message by defining the `initialize` method at a subclass level. For instance, to create all the instances of Point with the `0` value for `x` and `y`, we will redefine in Point the `initialize` method : `(λ (i_v*) (setq x 0 y 0) self)`

3 From Uniformity to Reflection

Since giving complete control to the users means a complete transparency in the objects definitions, we adapt the reflective interpreter technique [16] to the construction of this model. ObjVlisp is supported by two graphs: the instantiation graph and the inheritance graph. The instantiation graph represents the `instanceOf` relationship (P3 & P4), and the inheritance graph the `subclassOf` link (P5). Class and Object are the respective roots of these two (acyclic) graphs: they are defined in ObjVlisp as follows.

3.1 Class: Instantiation

Class is the first object of the system. As the root of the instantiation graph, it defines the behavior for classes. Because the new primitive is fielded by Class it will recursively create all other objects. To prevent the infinite regress provided by the instantation link (a metaclass is a class which instantiates a class, a metametaclass is a class which instantiates a metaclass, a metametametaclass ...), Class must be its own instance which severely constrains its structure.

Reflective pattern matching of Class

To verify the previous statement, we have to guarantee that the instance variables specified by Class match the corresponding values also held by Class, as its own instance, which is easily obtained by :

isit	name	supers	i_v	methods
Class	Class	(Object)	(isit name supers i_v methods)	(new (λ..)..)

Notice that the value associated with the instance variable `i_v` is exactly the ordered set of instance variables `(isit, name, ... , methods)` itself, this reflective pattern matching illustrates the definition of Class as an object.

[4] This table shows the dictionary of values owned by each object :
? (send a_point_1 'i_values)
= #(Point 20 30)
? (send Point 'i_values)
= #(Class Point (Object) (isit x y) (x λ_1 x: λ_2 init λ_3 display λ_4))
? (send Class 'i_values)
= #(Class Class (Object) (isit name supers i_v methods) (new...))

To prepare the bootstrap: the Lisp skeleton

"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" [12].

Defining Class from itself necessitates specifying the bootstrap mechanism. We create manualy the skeleton of Class. If we represent objects as lists, we will use the skeleton :

```
(setq Class '(
  Class
  Class
  (Object)
  (isit name supers i_v methods)
  (
   new          (λ (self . i_values)
                  (send (makeInstance name)
                        'initialize i_values)
   initialize   (λ (self i_values)
                  (initIv self ... ...)
                  (setq i_v (herit-i-v supers i_v))
                  (setq methods (scan-methods ...))
                  (set name self)) ))
```

In fact, we only define the new and `initialize` methods supporting the self-instantiation of Class.

This bootstrapping process then creates the real Class object, by sending to the Class skeleton the appropriate new message. Note that the skeleton is destroyed by the circular (re)definition of Class.

The bootstrap: the Self Instantiation of Class

The Class definition establishes that Class is its own instance, is a subclass of Object, and uses the instance variables previously mentioned. These definitions and examples are given for Le_Lisp [7] :

```
(send Class 'new
  :name      'Class
  :supers    '(Object)
  :i_v       '(name supersi_v methods)
  :methods   '(
               new          (λ i_values
                              (send (send self 'basicnew)
                                    'initialize i_values))
               basicnew     (λ () (makeInstance name))
               initialize   (λ (i_values)
                              (run-super)
                              (setq i_v (herit-i-v supers i_v))
                              (setq methods ...))
                              (set name self))
                 ...                ...

               name         (λ () name)
               supers       (λ () supers)
               i_v          (λ () i_v)
               methodsDic   (λ () methods)
                 ...                ...
               understand   (λ (selector method)
                              (defmethod self selector method))
               selectors    (λ () (selectors methods)) ) )
```

The definitions of the methods shows that all instance variables are automatically bound to their values in a method body. Consequently, the λ-expressions associated with the `name`, `supers`, `i_v` and `methods` selectors are quite easy to express. Similarly, in the `new` method, `self` denotes the generator (here Class). The `initialize` method uses the `run-super` form to call the general allocator (`makeInstance`) defined at the Object level.

3.2 Object: Inheritance

Postulate (P5) introduces the inheritance mechanism (which concerns only classes). The ObjVlisp inheritance allows to connect together instance variables and methods of several classes but in two different ways:

- The inheritance of instance variables 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 the ":i_v" keyword used by the new message).

- On the other hand, method inheritance is dynamic and uses the virtual copy mechanism implemented by the linkage of classes in the inheritance graph which is supported by the `supers` instance variable. When the method lookup fails in the receiver class then the search continues in a depth-first/breadth-first way.

  This lookup call may be locally modified by the `run-super` form - same as in CommonLoops [2] and similar to the `super` construct of Smalltalk-80 - which overrides the current method. The lookup starts (statically) at the superclass(es) of the class containing the method.

Classes vs Terminal Instances: the initialize method

The inheritance mechanism of instance variables is applied only when creating classes. Thus we need to distinguish creation of classes and creation of terminal instances. As we pointed out already, a metaclass is a class which inherits from Class the `new` method and the (`name supers i_v methods`) instance variables[5].

The reflective definition of ObjVlisp allows to use only one allocator - the `basicnew` - and nevertheless to explicit the difference between class and terminal instance creations: the `initialize` method owned by Object treats the terminal instance, and the `initialize` method owned by Class implements the inheritance mechanism associated to instance variables at a class creation time.

Object the most common class

The second primitive class is Object, instance of Class. Object represents the most common class - the intersection of all classes - describing the most common behavior (for classes and terminal instances). It is created during the bootstrap mechanism, immediately before Class. The `isit` instance variable is statically inherited by all classes. Then `isit` provides the instanciation link (the umbilical cord) between a class and its instances.

```
(send Class 'new
  :name       'Object
  :supers     '()
  :i_v        '(isit)
  :methods    '(
                class        (λ () isit)
                initialize   (λ (i_values)
                               (initIv self ... ...) self)
                ?            (λ (i_var) (ref i_var self))
                ?←           (λ (i_var i_val)
                               (setf (ref i_var self) i_val))
                i_values     (λ () self)
                metaclass?   (λ () (memq 'supers i_v))
                class?       (λ ()
                               (send (i_v* isit) 'metaclass?))
                  ...          ...
                error        (λ msg '(λ bs ',msg)) ) )
```

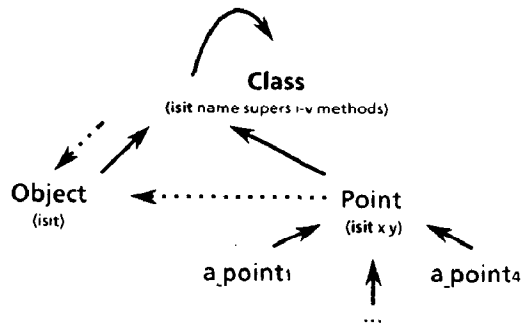
From this definition Object has no superclasses and each ObjVlisp object answers to the <*selector*> by <action>:

class	giving the name of its class
initialize	initializing the instance variables
?	returning the value of the field i_var
?←	writing i_var with the new value
i_values	returning the list of values of the i_v
metaclass?	testing if the object is a metaclass
class?	testing if the object is a class
error	implementing the standard treatment of error

Notice that the ? and ?← methods which access the value of any instance variable (read&write) respect their lexical scoping (and violate the encapsulation principle).

3.3 Architecture of the ObjVlisp model

We summarize the general structure of the ObjVlisp model by connecting together the `instantiation graph` and the `inheritance graph`. At the creation of the system there are only the Class and Object classes. The "naive" use of the system will keep the depth of the instantiation tree to three. See below for a similar example to Smalltalk-76 [13]; all classes are instances of Class :



[5] The *metaclass?* predicate defined in Object uses the `supers` instance variable to recognize metaclasses.

The rest of this paper establishes that creation of metaclasses brings substantial benefits. There is no longer any depth limitation of the instantiation tree, and the user can extend it as much as he wants to specify different metalevels of shared instance variables and methods.

4 From Reflection to Extensibility

4.1 Building new metaclasses

By combining the inheritance mechanism with the instantiation one we can create multiple metaclasses. **Ametaclass** is defined as a subclass of **Class** i.e. dynamically inherits the **new** primitive (to create objects) and it receives a copy of the basic instance variables defining a class (`name`, `i_v`, `supers` and `methods`), copy extended by the cv_i variables :

(send Class 'new	
:name	'Ametaclass
:i_v	'(cv_1 ... cv_n)
:supers	'(Class)
:methods	'(...))

Following this definition, the creation of Aclass needs the instantiation of every basic instance variables plus the instantiation of each new cv_i :

(send Ametaclass 'new	
:name	'Aclass
:i_v	'(iv_1 ... iv_n)
:supers	'(...)
:methods	'(...)
:cv_1	cv_1*
...	...
:cv_n	cv_n*)

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 us call it **MetaPoint**) specifies this common character :

(send Class 'new		
:name	'MetaPoint	
:supers	'(Class)	
:i_v	'(char)	
:methods	'())	
(send MetaPoint 'new		
:name	'DefaultPoint	
:supers	'(Object)	
:i_v	'(x y)	
:methods	'(*init*	(λ () (setq x 40 y 12) self)
	x	(λ () x)
	x:	(λ (nx) (setq x nx) self)
	display	(λ () (format () (catenate "~" x "D") char)))
:char	"*")	

MetaPoint is declared as a subclass of **Class** (thus it is a metaclass). It inherits the `name`, `supers`, `i_v` and `methods` instance variables from **Class** and adds to them the instance variable `char`. Consequently, DefaultPoint specifies the associated value of `char`, i.e. * by using the associated keyword. Now we could create such a point :

```
? (setq a_point (send DefaultPoint 'new :x 20))
= a DefaultPoint
? (send a_point 'display)
=          *
```

Class methods by Example

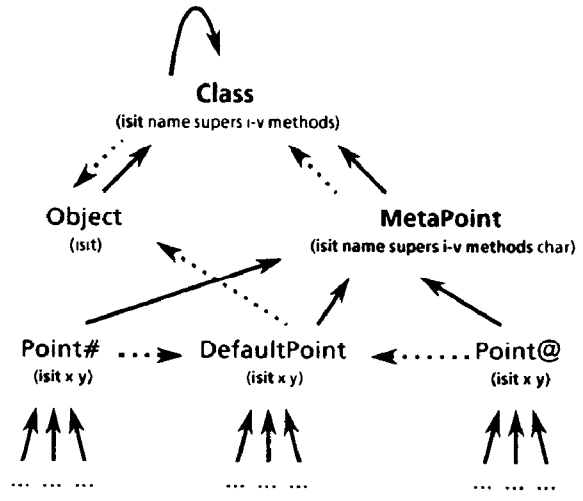
As for class variables, class methods are specified in the metaclass as ordinary methods. Suppose we want to define a new class method for DefaultPoint to create and initialize a new point. We simply define the `newinit` method of `MetaPoint` (assuming we define also an `init` method in the Point class, or at least in the Object class) :

(send Class 'new		
:name	'MetaPoint	
:supers	'(Class)	
:i_v	'(char)	
:methods	'(*newinit*	(λ () (send (send self 'new) 'init)))
	char	(λ () char)
	char:	(λ (newChar) (setq char newChar))))

- `newinit` creates a new instance, `(send self 'new)` then receives the `init` message.
- `char` gives access to the char variable. We have introduced this method to show that `char` is both accessible by the DefaultPoint `display` method and by the MetaPoint `char` method,
- `char:` allows the modification of the `char` class variable. For instance, the `(send DefaultPoint char: ''@'')` message provides the new @ display for all the instances of DefaultPoint.

4.2 Parametrization of a class

The DefaultPoint class is now parametrized by its display character and the **MetaPoint** metaclass represents this abstraction. Let us define two new classes, called Point# and Point@ with two different display characters. Obviously, they are defined as a subclass of DefaultPoint :



Notice that a same metaclass (here **MetaPoint**) can be used to instantiate several classes (Point#, Point@ ...) : there is no one specific metaclass associated to each class.

```
? (send MetaPoint 'new
        :name 'Point# :supers '(DefaultPoint) :char "#" )
= Point#
? (send 'MetaPoint 'new
        :name 'Point@ :supers '(DefaultPoint) :char "@" )
= Point@
? (send (send Point# 'new :x 1) 'display)
= #
? (send (send Point@ 'new :x 9) 'display)
=       @
```

Comparison with Smalltalk-80

We have pointed in [5] that the Smalltalk-80 terminology is not homogeneous. Smalltalk class variables are not the instance variables of the class defined as an object but a dictionary of variables shared between all the instances of a same 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.

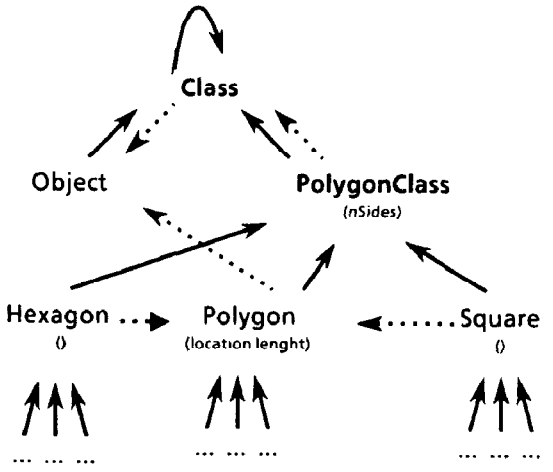
Nevertheless, if we use the instance variables of a metaclass, we can simulate the "MetaPoint" construction in Smalltalk-80. Obviously we need to give an explicit access to the char variable and we have to use a new different metaclass for each class of Point :

```
Object subclass: #DefaultPoint
  instanceVariableNames: 'x y'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Graphic-Primitives'.
DefaultPoint class instanceVariableNames: 'char'.
DefaultPoint class methodsFor: 'meta-iv access'
  initialize ↑char← "".

DefaultPoint subclass: #Point@
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Graphic-Primitives'.
Point@ class instanceVariableNames: ''.
Point@ class methodsFor: 'meta-iv access'
  initialize ↑char← "@".
```

The DefaultPoint example illustrates a general knowledge scheme (as does the Polygon example below). To generalize the solution, we have decided to extend the scope of the instance variables of a class to each of its instances. Unlike Smalltalk-80, our class variables are inherited but not shared by the subclasses.

Each polygon is defined by its `location` (the first vertex) and the `length` of any of its sides. To parametrize the number of sides - 4 for a square, 6 for a hexagon, undef for a polygon - we use the `nSides` class variable. To simplify the next figure, the inherited variables are not drawn :



4.3 Filiation link (Set) :

To use classes which remember all their instances, we define a new metaclass (**Set**), as a subclass of **Class** with the new **sons** instance variable pointing the list of instances. We just have to redefine the **new** method in **Set** to add the newly created instance at the end of the sons list :

```
(send Class 'new
  :name        'Set
  :supers      '(Class)
  :i_v         '(sons)
  :methods     '( sons      (λ () (cdr sons))
                  new       (λ i_values
                              (nconc sons
                                  (cons (run-super) ())))
                  mapsons   (λ (unaryS)
                              (mapc
                                (λ (rec) (send rec unaryS))
                                (send self 'sons))
                              unaryS) ) )

(send Set 'new
  :name        'Point
  :i_v         '(x y)
  :supers      '(Object)
  :methods     '( init      (λ () (setq x 40 y 12) self)
                  display   (λ () (print (format () ...)) self) )
  :sons        '(hook) )
```

The **sons** method gives access to the sons class variable and the **mapsons** method distributes an unary message (without arguments) to all the instances of a particular set. The next session shows the behavior of Point defined as an instance of **Set**

```
? (prog1 'ok (send Point 'new :x 10) (send Point 'new :x 20))
= ok
? (send Point 'sons)
= (#(Point 10 nil) #(Point 20 nil))
? (send Point 'mapsons 'display)
      *
            *
= display
```

This solution provides a uniform extension of the metaclass system and seems better than the Loops "class property hook" used by the **ListMetaClass** definition in [1] (P. 36).

4.4 MetaPoint as an instance of Set :

Now we can add a new metaclass level by defining **MetaPoint** as a subclass of **Set**, allowing DefaultPoint, Point# and Point@ to memorize their instances. Since a metaclass is a ordinary class, such an extension is easy to repeat and the metalinks can be created indefinitely.

5 Metaclasses are useful

"With respect to Simula, Smalltalk also abandons static scoping, to gain flexibility in interactive use, and strong typing, allowing it to implement system introspection and to introduce the notion of meta-classes [6]."

5.1 Metaclasses provide metatools to build open-ended architecture

"The metaclass determines the form of inheritance used by its classes and the representation of the instance of its classes. The metaclass mechanism can be used to provide particular forms of optimization or to tailor the Common Lisp Object System for particular uses (such as the implementation of other languages like Flavors, Smalltalk-80 and Loops)" [3].

From an implementor's point of view, metaclasses are very powerful because they provide hooks to extend or modify an existing kernel. For example, ObjVlisp uses the metaclass facilities to simulate other object-oriented systems. Metaclasses may control :

1. the inheritance strategy (simple, multiple, method wrapping [18]). To implement variations on inheritance schemes [19], we define at the metaclass level a method (or an instance variable) parametrizing the `lookup` method used by the `send` primitive,

2. the internal representation of objects by using different `makeInstance` primitives creating lists, vectors, hashtables or structures; each metaclass fielding a private `new` method,

3. the access to methods by implementing a caching technique. We associate with each class a private memory (the `cache` instance variable) memorizing the addresses of methods already called,

4. the access to instance variable values by distinguishing between private and public variables or by implementing active-values or demons.

5.2 Metaclasses remove the boundary between users and implementors

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 revert to the situation in Smalltalk-76. Every class would be instance of class Class" [4].

Obviously we disagree with the Borning's conclusion. We consider that metaclasses provide an explicit definition of the class system. They express the behavior of classes in a transparent way. Because they have ability to manipulate their own structures, they can implement system introspection. Consequently, metaclasses support a circular definition of the system reducing the boundary between users and implementors. But, to fully exploit this metalevel, the metaclass concept must be simple enough to be understood by the user. We believe this is not true in Smalltalk-80 but that the ObjVlisp uniform and reflective architecture has reached this goal.

6 Conclusions

6.1 Results

The ObjVlisp model's primary advantage is **uniformity**. There is now only one kind of object: a class is an object and a metaclass is a true class whose instances are classes. This allows a simplification and economy of concepts, which are thus more powerful and general. The second property is **reflection** which provides a language completely and uniformly accessible by the user. The system is self-described by the explicit definition of the root of the instantiation tree (**Class**) and the root of the inheritance tree (Object). The main results are that there is no limitation in the depth of the instantiation tree, the metalinks can be created indefinitely and class variables are defined at the metaclass level. Finally, **extensibility** permits various applications and modeling alternative semantics, for instance Thinglab composite objects and partwhole hierarchy or Smalltalk-80 dependencies [11].

6.2 New Improvements

A first version of this paper was presented at the *workshop on Meta-Level Architectures and Reflection* organized in Alghero [16]. In this new version, the difference between instance creation and class creation is explicitly defined at the ObjVlisp level through two distincts `initialize` methods, respectively owned by Object and `Class`. Thus we do not need to add an extra metalevel (the metaclass level of Loops or Smalltalk-80) [5] and the ObjVlisp instantiation kernel is really minimal.

6.3 Future work

We have used the ObjVlisp model to study the instantiation mechanism. We plan now to investigate three axes:

- experimentation in object-oriented methodologies by writing relevant examples in ObjVlisp other than those provided by the Smalltalk-80 image,

- development of an object kernel for EuLisp and IsoLisp. This work is very close to the CLOS approach [3] but we expect to use the ObjVlisp experience to propose a cleaner metaclass level,

- implementing the ObjVlisp metaclasses architecture in Smalltalk-80 by redefinition of the "kernel classes".

Acknowledgements

We thank Jean-Pierre Briot for its major contribution to the ObjVlisp model, Alain Deustch for its implementation of the tree-walker, Jean-François Perrot, Henry Lieberman, Kris Van Marcke, Glenn Krasner and Nicolas Graube for their helpful comments on this text.

The ObjVlisp project is part of the "O.O.P. Methodology" group of the GRECO de Programmation

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] Bobrow, D.G., DeMichiel L.G., Gabriel R.P., Keene S., Kiczales G., Moon D.A, Common Lisp Object System Specification, X3J13 (ANSI COMMON LISP), March 1987.

[4] Borning A., O'Shea, T., DeltaTalk: An Empirically and Aesthetical Motivated Simplification of the Smalltalk-80 Language, *ECOOP'87, to appear in Springer Verlag, Bezivin J. & Cointe P. ed.*, Paris, France, 15-17 June 1987.

[5] Briot, J-P., Cointe, P., A Uniform Model for Object-Oriented Languages Using the Class Abstraction, *IJCAI'87*, Milan, I, August 1987.

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

[7] Chailloux, J., Devin, M., Dupont, F., Hullot, J.M., Serpette, B., Vuillemin, J., LE LISP de l'INRIA, Version 15.2 (The manual), INRIA, Domaine de Voluceau, Rocquencourt 78153 le Chesnay, Mai 1986.

[8] Cointe, P., A VLISP Implementation of SMALLTALK-76, pp 89-102, *Integrated Interactive Computing Systems*, North-Holland, Degano, P. & Sandewall, E. editors, 1983.

[9] Cointe, P., Briot J.P., Serpette B., The FORMES language: a Musical Application of Object Oriented Concurrent Programming, pp 221-258 in Object Oriented Concurrent Programming, MIT Press, Cambridge, Mass A Yonezawa & M. Tokoro editors, May 1987.

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

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

[12] Hofstadter D.R., *GOEDEL, ESCHER, BACH: an Eternal Golden Braid*, The Harvester Press, John Spiers editor, Stanford Terrace, Hassocks, Sussex Publisher, 1979.

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

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

[15] Lieberman, H., Delegation and Inheritance, Two Modular mechanisms, Conf. Record of the 3rd Workshop on OOP, Centre Georges Pompidou, Paris, Bigre+Globule No 48, Bezivin J. and Cointe P. editors, January 1986.

[16] Maes, P., and al, Workshop on Meta-Level Architectures and Reflection, *to appear in North Holland, P. Maes & D. Nardi ed.*, Alghero, Italy, 27-30 October 1986.

[17] Maes, P., Computational Reflection, PhD thesis, Vrije Universiteit Brussel, AI-LAB Pleinlaan 2, B-1050 Brussels, Belgium, Mars 1987.

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

[19] Stefik, M., Bobrow, D.G., Object-Oriented Programming: Themes and Variations, The AI magazine, pp 40-62, Winter 1985.

Appendix

We give two alternatives implementations written in Le Lisp representing objects as lists. The first one dynamically binds the instance variables at the run-time (cf. the `send` form), when the second one pre-compiles the methods by using a tree-walker. The `smacrolet` form replaces each instance variable name by its access function (cf. the `ref` form). For instance, below are the definition of `x`, `x:` and `display` methods of Point after their textual expansion:

```
(λ () x)                          (λ () (ref 'x self))
(λ ()                             (λ ()
  (format () ... char))             (format () ... (ref 'char (class-of self))))
(λ (nx)                           (λ (nx)
  (setq x nx) self)                 (setf (ref 'x self) nx) self)
```

```
; Dynamic access to instance variables

(typecn #/: (typecn #/a))
(defsharp |:| ()
   (with ( (typecn #/: 'cpkgc)  (typecn #/# 'cpname) )
      (list (read)) ))
(defvar class 'noboot) (defvar object ()) (defvar self ())

(dmd type-of (name) '(car ,name))
(dmd class-of (obj) '(i-v* (type-of ,obj)))
(dmd metaclass-of (obj) '(i-v* (metatype-of ,obj)))
(dmd metatype-of (obj) '(type-of (class-of ,obj)))
(dmd name (name) '(cadr ,name))
(dmd supers (name) '(caddr ,name)))
(dmd i-v (name) '(cadddr ,name))
(dmd i-v* (x) '(symeval ,x))
(dmd keywords (name) '(cadddr (cdr ,name))))
(dmd methodDic (name) '(cadddr (cddr ,name))))
(dmd methods (name) '(cdr (plist-to-dico ,name)))
(dmd selectors (name) '(car (plist-to-dico ,name)))
(dmd attach (s l) '(rplac ,l ,s (cons (car ,l) (cdr ,l))))
(dmd methodfor (classe sel) '(getl (methodDic ,classe) ,sel))
(dmd rewrite (inst isit) '(cval-dico (i-v (i-v* ,isit)) ,inst))

(defun send (obj -selector- . -args-)
  (if (eq obj self) (apply (lookup -selector- (i-v* isit) obj) obj -args-)
      (letv (i-v (metaclass-of obj)) (class-of obj)
         (letv (i-v (class-of obj)) obj
             (protect
                (apply (lookup -selector- (i-v* isit) obj) obj -args-)
                (rewrite obj isit))))))
(dmd run-super ()
  '(apply ,(lookup -selector- (i-v* (car (supers (i-v* isit)))) self) self -args-))

(defun makeInstance (model)
    (tcons model (makelist (1- (length (i-v (i-v* model)))) nil)))
(defun initIv (structure slots slots*)
  (while slots (set (nextl slots) (nextl slots*)))
  structure)

(defun getl (l sel)
   (cond
      ((not (consp l)) ())
      ((eq (car l) sel) (cadr l))
      (t (getl (cddr l) sel)) ))
(defun plist-to-dico (sexp)
   (when (symbolp sexp) (setq sexp (plist sexp)))
   (let ((sel (ncons nil)) (meth (ncons nil)))
    (letn self ((qsel sel) (qmeth meth) (l sexp))
      (ifn l (cons (cdr sel) (cdr meth))
           (self (placdl qsel (nextl l)) (placdl qmeth (nextl l)) l)))))
(defun cval-dico (dic-var dic-val)
   (when dic-val
     (rplaca dic-val (i-v* (car dic-var)))
     (cval-dico (cdr dic-var) (cdr dic-val))))))
```

```
(defun lookup (sel isit obj)
   (tag backtrack (tag again (depth sel isit)) (send obj 'error 'lookup)))
(defun depth (key node)
  (let ((method (methodfor node key)))
    (cond
      (method (exit backtrack method))
      ((eq node object) (exit again 'backtrack))
      (t (breadth key (supers node))) )))
(defun breadth (key l-nodes)
  (while l-nodes  (tag again (depth key (i-v* (nextl l-nodes))))))

(defun scan-match (pat dat-plist)
   (mapcar (lambda (msg) (getl dat-plist msg)) pat))
(defun make-keywords (iv)
   (mapcar (lambda (k) (setq k (concat ': k)) (set k k)) iv))
(defun scan-selectors (dic-methods)
   (mapc (lambda (sel) (make-descriptor sel descriptor)) (selectors dic-methods)))
(defun scan-iv* (f v)
   (let ((lv v) (p ()))
     (while f (if v (nextl v) (newl p ()))
     (nextl f))
   (nconc lv p))))
(defun scan-method (lambda-form classe)
   (scan-parameters lambda-form (cadr lambda-form)))
(defun scan-parameters (l lpar)
   (cond
     ((null lpar) (rplaca (cdr l) '(self)))
     ((atomp lpar) (rplaca (cdr l) (cons 'self lpar)))
     (t (attach 'self lpar))))
(defun scan-methods (pl classe)
   (ifn (cddr pl) (when (cadr pl) (scan-method (cadr pl) classe))
        (scan-method (cadr pl) classe)
        (scan-methods (cddr pl) classe))
    pl)
(defun defmethod (class sel method)
   (scan-method method class)
   (let ((methodDic (methodDic class)))
     (attach method methodDic)
     (attach sel methodDic)))

(defun remove-duplicates (lst)
    (do ((l lst (cdr l))(res ()))((atom l) (nreverse res))
        (unless (member (car l) res) (newl res (car l)))))
(defun herit-i-v (supclass i-v)
   (remove-duplicates (append
      (apply 'append (mapcar (lambda (Ci) (i-v (i-v* Ci))) supclass))
      i-v)))

(make-keywords '(name supers i-v keywords methods))
(setq Class '#(Class Class (Object)
   (isit name supers i-v keywords methods)
   (:name :supers :i-v :keywords :methods)
   (new          (lambda (self . i-v*) (send (makeInstance name) 'initialize i-v*))
    initialize   (lambda (self iv*)
                   (initIv self (cdr (i-v (i-v* isit)))
                               (scan-match (keywords (i-v* isit)) iv*))
```

```
                (setq i-v (herit-i-v supers i-v))
                (setq methods (scan-methods methods self))
                (setq keywords (make-keywords (cdr i-v)))
                (rewrite self isit)
                (set name self) )) ))

(send Class 'new  :name 'Object
  :i-v      '(isit)
  :methods
  '(class      (lambda () isit)
    class?     (lambda () (send (i-v* isit) 'metaclass?))
    metaclass? (lambda () (memq 'supers i-v))
    ?          (lambda (iv) (i-v* iv))
    ?<-        (lambda (iv v) (set iv v))
    error      (lambda -msg- (print "selecteur inconnu  " sel " de la classe " isit)
    initialize (lambda (iv*)
                 (initIv self (cdr (i-v (i-v* isit)))
                   (scan-match (keywords (i-v* isit)) iv*))) ))

(send Class 'new :name 'Class
 :supers '(Object)
 :i-v    '(name supers i-v keywords methods)
 :methods
  '( basicnew      (lambda () (makeInstance name))
     new           (lambda i-v* (send (send self 'basicnew) 'initialize i-v*))
     initialize    (lambda (iv*)
                     (run-super)
                     (setq i-v (herit-i-v supers i-v))
                     (setq methods (scan-methods methods self))
                     (setq keywords (make-keywords (cdr i-v)))
                     (set name self))
     i-v           (lambda () i-v)
     subclassof    (lambda () supers)
     methodsDic    (lambda () methods)
     name          (lambda () name)
     selectors     (lambda () (selectors methods))
     methods       (lambda () (methods methods)) ))

; Static access to instance variables

(defun memdic (var l-var l-val)
  (when l-var
    (if (eq (car l-var) var) l-val
        (memdic var (cdr l-var) (cdr l-val)))))
(defun fnref (slot obj) (car (memdic slot (i-v (class-of obj)) obj)))
(defun setref (slot val obj)
  (rplaca (memdic slot (i-v (class-of obj)) obj) val)
  val)
(dmd ref (slot obj) '(car (memdic ,slot (i-v (class-of ,obj)) ,obj)))
(defun scan-method (l classe)
 (scan-parameters l (cadr l))
 (when (cddr l) (scan-body l (i-v classe) classe)))

(defun scan-body (l i-v classe)
  (let ((parameters (cadr l))
        (super-i-v (i-v (class-of classe))))
    (setf (cddr l)
          '((smacrolet
              ,(append
                 (mapcar #'(lambda (slot) (list slot slot))
                         parameters)
                 (mapcar #'(lambda (slot)'(,slot (lambda(var) '(ref ',var self))))
                         i-v)
                 (mapcar #'(lambda(slot)
                             '(,slot (lambda (var)'(ref ',var (class-of self)))))
                         super-i-v))
              ,@(cddr l))))
    l))

(send Class 'new :name 'Class ...)
(send Class 'new :name 'Object ...)

(defun send (obj -selector- . -args-)
   (apply (lookup -selector- (class-of obj) obj) obj -args-))
```

Safe Metaclass Programming

Noury M. N. Bouraqadi-Saâdani
Noury.Bouraqadi@emn.fr
École des Mines de Nantes
BP 20722
44307 Nantes - FRANCE

Thomas Ledoux*
Thomas.Ledoux@emn.fr
École des Mines de Nantes
BP 20722
44307 Nantes - FRANCE

Fred Rivard*
Fred_Rivard@oti.com
École des Mines & OTI Inc. Nantes
BP 20722
44307 Nantes - FRANCE

Abstract

In a system where classes are treated as first class objects, classes are defined as instances of other classes called *metaclasses*. An important benefit of using metaclasses is the ability to assign *properties* to classes (e.g. being abstract, being final, tracing particular messages, supporting multiple inheritance), independently from the base-level code. However, when both inheritance and instantiation are explicitly and simultaneously involved, communication between classes and their instances raises the *metaclass compatibility* issue. Some languages (such as SMALLTALK) address this issue but do not easily allow the assignment of specific properties to classes. In contrast, other languages (such as CLOS) allow the assignment of specific properties to classes but do not tackle the compatibility issue well.

In this paper, we describe a new model of meta-level organization, called *the compatibility model*, which overcomes this difficulty. It allows *safe metaclass programming* since it makes it possible to assign specific properties to classes while ensuring metaclass compatibility. Therefore, we can take advantage of the expressive power of metaclasses to build reliable software. We extend this compatibility model in order to enable safe reuse and composition of class specific properties. This extension is implemented in NEOCLASSTALK, a fully reflective SMALLTALK.

Keywords: Metaclasses, compatibility, class specific properties, class property propagation.

1 Introduction

It has been shown that programming with metaclasses is of great benefit [KAJ+93][Zim96] [BGL98]. An interesting use of metaclasses is the assignment of *specific properties* to classes. For example, a class can be abstract, have a unique instance, trace messages received by its instances, define pre-post conditions on its methods, forbid redefinition of some particular methods... These properties can be implemented using metaclasses, allowing thereby the customization of the classes behavior [LC96].

From an architectural point of view, using metaclasses organizes applications into abstraction levels. Each level describes and controls the level immediately below to which it is causally connected [Mae87]. Reified classes communicate with other objects including their own instances. Thus, classes can send messages to their instances and instances can send messages to their classes. Such message sending is named *inter-level communication* [MMC95].

However, careless inheritance at one level may break inter-level communication resulting in an issue called *the compatibility issue* [BSLR96]. We have identified two symmetrical kinds of compatibility issues. The first one is the *upward compatibility* issue, which was named *metaclass compatibility* by Nicolas Graube [Gra89], and the second one is the *downward compatibility* issue. Both kinds of compatibility issues are important impediments to metaclass programming that one should always be aware of.

⋆Funded by IBM Global Services - FRANCE.

*Since the 1^{st} July 1998: Object Technology International Inc. 2670 Queensview Drive, Ottawa, Ontario, CANADA K2B 8K1.

Currently, none of the existing languages dealing with metaclasses allow the assignment of specific properties to classes while ensuring compatibility. CLOS [KdRB91] allows one to assign any property to classes, but it does not ensure compatibility. On the other hand, both SOM [SOM93] and SMALLTALK [GR83] address the compatibility issue but they introduce a *class property propagation* problem. Indeed, a property assigned to a class is automatically propagated to its subclasses. Therefore, in SOM and SMALLTALK, a class cannot have a specific property. For example, when assigning the abstractness property to a given SMALLTALK class, subclasses become abstract too [BC89]. It follows that users face a dilemma: using a language that allows the assignment of specific class properties without ensuring compatibility, or using a language that ensures compatibility but suffers from the class property propagation problem.

In this paper, we present a model — *the compatibility model* — which allows safe metaclass programming, i.e. it makes it possible to assign specific properties to classes without compromising compatibility. In addition to ensuring compatibility, the compatibility model avoids class property propagation: a class can be assigned specific properties without any side-effect on its subclasses.

We implemented the compatibility model in NEOCLASSTALK, a SMALLTALK extension which introduces many features including explicit metaclasses [Riv96]. Our experiments [Led98][Riv97] showed that the compatibility model allows programmers to fully take advantage of the expressive power of metaclasses. This effort has resulted (i) in a tool that permits a programmer unfamiliar with metaclasses to transparently deal with class specific properties, and (ii) in an approach allowing reuse and composition of class properties.

This paper is organized as follows. Section 2 presents the compatibility issue. We give some examples to show its significance. Section 3 shows how existing programming languages address the compatibility issue, and how they deal with the property propagation problem. Section 4 describes our solution and illustrates it with an example. In section 5, we deal with reuse and composition of class specific properties within the compatibility model. Then, we sketch out the use of the compatibility model for both base-level and meta-level programmers. The last section contains a concluding summary.

2 Inter-level communication and compatibility

We define inter-level communication as any message sending between classes and their instances (see Figure 1). Indeed, class objects can interact with other objects by sending and receiving messages. In particular, an instance can send a message to its class and a class can send a message to some of its instances. We use SMALLTALK as an example to illustrate this issue[1].

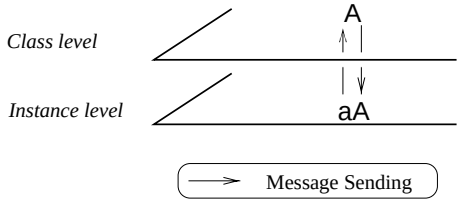


Figure 1: Inter-level communication

Two methods allow inter-level communication in SMALLTALK: new and class. When one of them is used, the involved objects belong to different levels of abstraction[2]:

- An object receiving the class message returns its class. Then, the class method makes it possible to go one level up. The following two instance methods — excerpted from Visual Works SMALLTALK — include message sending to the receiver's class.

```
⋆ message name is sent to the class:
Object≫printOn: aStream
  | title |
  title := self class name.
  ...
⋆ message daysInYear: is sent to the class:
Date≫daysInYear
  "Answer the number of days in the year
  represented by the receiver."
  ↑ self class daysInYear: self year
```

[1] We use the SMALLTALK syntax and terminology throughout this paper.

[2] Static measures we made over a Visual Works SMALLTALK image show that inter-level communication is very frequent. 25% of classes include instance methods referencing the class and 24% of metaclasses define methods referencing an instance.

- A class receiving the new message returns a new instance. Therefore, the new method makes it possible to go one level down. The following two class methods include message sending to the newly created instances.

```
⋆ message at:put: is sent to a new instance:
ArrayedCollection class≫with: anObject
  | newCollection |
  newCollection := self new: 1.
  newCollection at: 1 put: anObject.
  ↑newCollection
⋆ message on: is sent to a new instance:
Browser class≫openOn: anOrganizer
  self openOn: (self new on: anOrganizer) with-
TextState: nil
```

Thus, inter-level communication in SMALLTALK is materialized by sending the messages new and class. Other languages where classes are reified (such as CLOS and SOM) also allow similar message sending.

Since these inter-level communication messages are embedded in methods, they are inherited whenever methods are inherited. Ensuring *compatibility* means making sure that these methods will not induce any failure in subclasses, i.e. all sent messages will always be understood. We have identified two kinds of compatibility: *upward compatibility*[3] and *downward compatibility*.

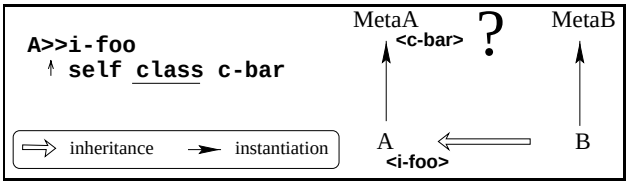


Figure 2: Compatibility need to be ensured at a higher level

2.1 Upward compatibility

Suppose A implements a method i-foo that sends the c-bar message to the class of the receiver (see Figure 2). B is a subclass of A. When i-foo is sent to an instance of B, the B class receives the c-bar message. In order to avoid any failure, B should understand the c-bar message (i.e. MetaB should implement or inherit a method c-bar).

[3] Nicolas Graube named this issue *metaclass compatibility* [Gra89].

Definition of upward compatibility:
Let B be a subclass of the class A, MetaB the metaclass of B, and MetaA the metaclass of A.

Upward compatibility is ensured for MetaB and MetaA iff: every possible message that does not lead to an error for any instance of A, will not lead to an error for any instance of B.

2.2 Downward compatibility

Suppose MetaA implements a method c-foo that sends the i-bar message to a newly created instance (see Figure 3). MetaB is created as a subclass of MetaA. When c-foo is sent to B (an instance of MetaB), B will create an instance which will receive the i-bar message. In order to avoid any failure, instances of B should understand the i-bar message (i.e. B should implement or inherit the i-bar method).

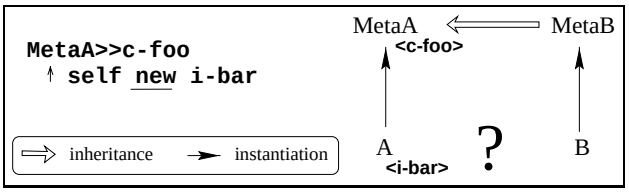


Figure 3: Compatibility need to be ensured at a lower level

Definition of downward compatibility:
Let MetaB be a subclass of the metaclass MetaA.

Downward compatibility is ensured for two classes B instance of MetaB and A instance of MetaA iff: every possible message that does not lead to an error for A, will not lead to an error for B.

3 Existing models

We will now show why none of the known models allow the assignment of specific properties to classes while ensuring compatibility.

3.1 CLOS

When (re)defining a class in CLOS, the validate-superclass generic function is called, before the direct superclasses are stored [KdRB91]. As a default, validate-superclass returns true if the meta-

class of the new class is the same as the metaclass of the superclass[4], i.e. classes and their subclasses must have the same metaclass. Therefore, incompatibilities are avoided but metaclass programming is very constrained.

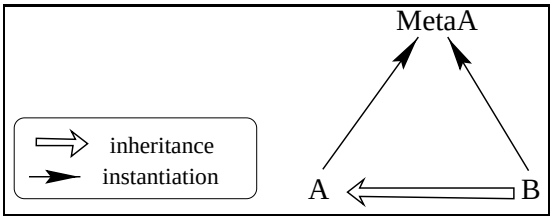


Figure 4: By default in CLOS, subclasses must share the same metaclass as their superclass

Figure 4 shows a hierarchy of two classes that illustrates the CLOS default compatibility management policy. Since class B inherits from A, B and A must have the same metaclass.

In order to allow the definition of classes with different behaviors, programmers usually redefine the validate-superclass method to make it always return true. Thus, CLOS programmers can have total freedom to use a specific metaclass for each class. So, they can assign specific properties to classes, but the trade-off is that they need to be always aware of the compatibility issue.

3.2 SOM

SOM is an IBM CORBA compliant product which is based on a metaclass architecture [DF94b]. The SOM kernel follows the OBJVLISP model [Coi87]. SOM metaclasses are explicit and can have many instances. Therefore, users have complete freedom to organize their metaclass hierarchies.

3.2.1 Compatibility issue in SOM

SOM encourages the definition and the use of explicit metaclasses by introducing a unique concept named *derived metaclasses* which deals with the *upward compatibility* issue [DF94a]. At compile-time, SOM automatically determines an appropriate metaclass that ensures upward compatibility. If needed, SOM automatically creates a new metaclass named a *derived metaclass* to ensure upward compatibility.

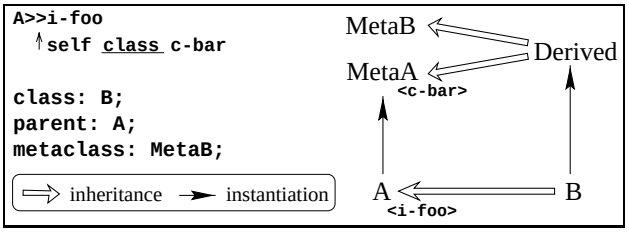


Figure 5: SOM ensures upward compatibility using derived metaclasses

Suppose that we want to create a class B, instance of MetaB and subclass of A. SOM will detect an upward compatibility problem, since MetaB does not inherit from the metaclass of A (MetaA). Therefore, SOM automatically creates a derived metaclass (Derived), using multiple inheritance in order to support all necessary class methods and variables[5]. Figure 5 shows the resulting construction. When an instance of B receives i-foo, it goes one level higher and sends c-bar to the B class. B understands the c-bar message since its metaclass (i.e. Derived) is a derived metaclass which inherits from both MetaB and MetaA.

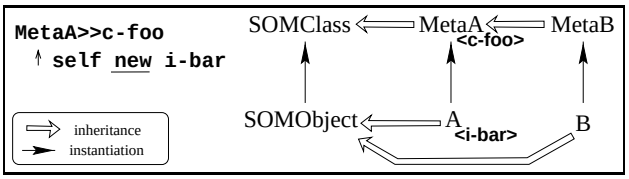


Figure 6: SOM downward compatibility failure example

SOM does not provide any policy or mechanism to handle *downward compatibility*. Suppose that MetaB is created as a subclass of MetaA (see Figure 6). The c-foo method which is inherited by MetaB sends the i-bar message to a new instance. When the B class receives the c-foo message, a runtime error will occur because its instances do not understand the i-bar message.

[4] In fact, it also returns true if the superclass is the class named t, or if the metaclass of one argument is standard-class and the metaclass of the other is funcallable-standard-class.

[5] The semantics of derived metaclasses guarantees that the declared metaclass takes precedence in the resolution of multiple inheritance ambiguities (i.e. MetaB before MetaA). Besides, it ensures the instance variables of the class are correctly initialized by the use of a complex mechanism.

3.2.2 Class property propagation in SOM

SOM does not allow the assignment of a property to a given class, without making its subclasses be assigned the same property. We name this defect *the class property propagation* problem. In the following example, we illustrate how derived metaclasses implicitly cause undesirable propagation of class properties.

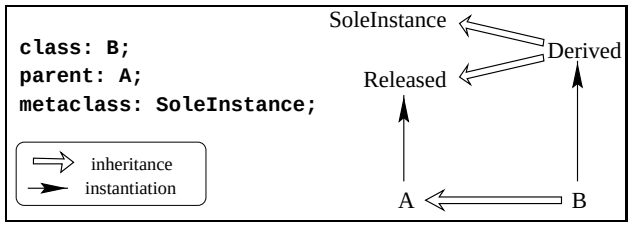


Figure 7: Class property propagation in SOM

Suppose that the A class of Figure 7 is a released class, i.e. it should not be modified any more. This property is useful in multi-programmer development environments for versionning purposes. In order to avoid any change, A is an instance of the Released metaclass. Let B a class that has a unique instance: B is an instance of the SoleInstance metaclass. But as B is a subclass of A, SOM creates B as instance of an automatically created derived metaclass which inherits from both SoleInstance and Released. Thus, as soon as B is created, it is automatically "locked" and acts like a released class. So, we cannot define any new method on it!

3.3 Smalltalk-80

In SMALLTALK, metaclasses are partially hidden and automatically created by the system. Each metaclass is non-sharable and strongly coupled with its sole instance. So, the metaclass hierarchy is parallel to the class hierarchy and is implicitly generated when classes are created.

3.3.1 Compatibility issue in Smalltalk-80

Using parallel inheritance hierarchies, the SMALLTALK model ensures both upward and downward compatibility. Indeed, any code dealing with `new` or `class` methods, is inherited and works properly.

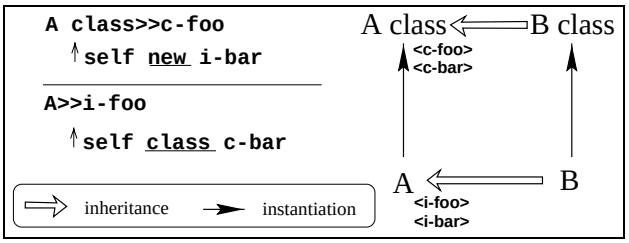


Figure 8: SMALLTALK ensures both upward and downward compatibilities

When one creates the B class, a subclass of A (see Figure 8), SMALLTALK automatically generates the metaclass of B ("B class"[6]), as a subclass of "A class", the metaclass of A. Suppose A implements a method i-foo that sends c-bar to the class of the receiver. If i-foo is sent to an instance of B, the B class receives the c-bar message. Thanks to the parallel hierarchies, the B class understands the c-bar message, and upward compatibility is ensured. In a similar manner, downward compatibility is ensured thanks to the parallel hierarchy.

3.3.2 Class property propagation in Smalltalk-80

Since metaclasses are automatically and implicitly managed by the system, SMALLTALK drastically reduces the opportunity to change class behaviors, making metaclass programming "anecdotal". As with SOM, SMALLTALK does not allow the assignment of a property to a class without propagating it to its subclasses.

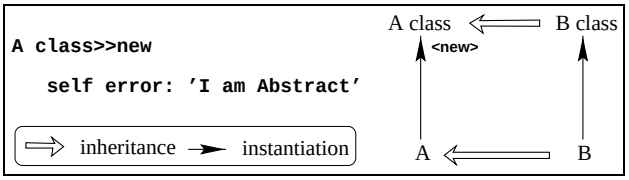


Figure 9: Class property propagation in SMALLTALK

In Figure 9, the A class is abstract since its subclasses must implement some methods to complete the instance behavior. B is a concrete class as it implements the whole set of these methods. Suppose

[6] The name of a SMALLTALK metaclass is the name of its unique instance postfixed by the word 'class'.

that we want to enforce the property of abstractness of A. In order to forbid instantiating A, we define the class method A class≫new which raises an error. Unfortunately, "B class" inherits the method new from "A class". As a result, attempting to create an instance of B raises an error[7]!

4 The compatibility model

Among the previous models, only the SMALLTALK one with its parallel hierarchies ensures full compatibility. However, it does not allow the assignment of specific properties to classes. On the other hand, only the CLOS model allows the assignment of specific properties to classes. Unfortunately, it does not ensure compatibility. We believe that these two goals can both be achieved by a new model which makes a clear separation between compatibility and class specific properties.

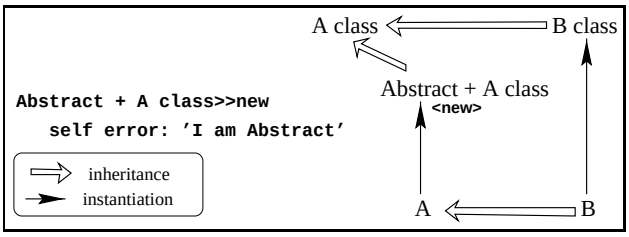


Figure 10: Avoiding the propagation of abstractness

We illustrate this idea of separation of concerns by refactoring the example of Figure 9. We create a new metaclass named "Abstract + A class" as a subclass of "A class" (see Figure 10). The A class is redefined as an instance of this new metaclass. As "Abstract + A class" redefines the new method to raise an error, A cannot have any instance. However, since "B class" is not a subclass of "Abstract + A class", the B class remains concrete. The generalization of this scheme is our solution, named *the compatibility model*.

In the remainder of this paper, names of metaclasses defining some class property are denoted with the concatenation of the property name, the + symbol and the superclass name. For example, "Abstract + A class" is a subclass of "A class" that defines the property of abstractness named Abstract.

4.1 Description of the compatibility model

The compatibility model extends the SMALLTALK model by separating two concerns: compatibility and specific class properties. A metaclass hierarchy parallel to the class hierarchy ensures both upward and downward compatibility like in SMALLTALK. An extra metaclass "layer" is introduced in order to *locally* assign any property to classes. Classes are instances of metaclasses belonging to this layer. So, the compatibility model is based on two "layers" of metaclasses, each one addressing a unique concern:

Compatibility concern: This issue is addressed by the metaclasses organized in a hierarchy parallel to the class hierarchy. We name such metaclasses: *compatibility metaclasses*. They define all the behavior that must be propagated to all (sub)classes. All class methods which send messages to instances should be defined in these metaclasses. Besides, all messages sent to classes by their instances should be defined in these metaclasses too.

Specific class properties concern: This issue is addressed by metaclasses that define the class specific properties. We name such metaclasses: *property metaclasses*. A class with a specific property is instance of a property metaclass which inherits from the corresponding compatibility metaclass. The property metaclass is not joined to other property metaclasses, since it defines a property specific to the class.

Figure 11 shows[8] the compatibility model applied to a hierarchy consisting of two classes: A and B. They are respectively instances of the "AProperty + AClass" and "BProperty + BClass" metaclasses. "AProperty + AClass" defines properties specific to class A, while "BProperty + BClass" defines properties specific to class B. As "AProperty + AClass" and "BProperty + BClass" are not joined

[7] This example is deliberately simple, and one could avoid this problem by redefining new in "B class". But, this solution is a kind of inheritance anomaly [MY93] that increases maintenance costs.

[8] Compatibility metaclasses are surrounded with a dashed line and property metaclasses are drawn inside a grey shape.

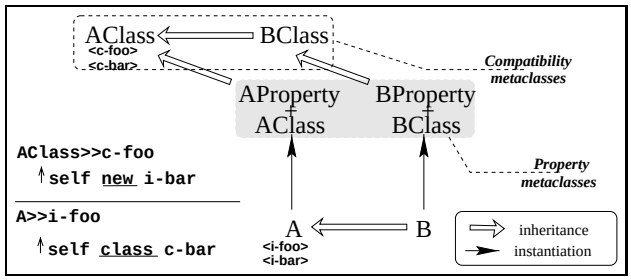


Figure 11: The compatibility model

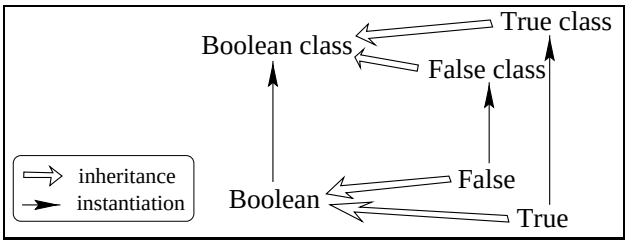


Figure 12: The Boolean hierarchy of SMALLTALK

by any link, class property propagation does not occur. Thus, A and B can have distinct properties.

Since "AProperty + AClass" and "BProperty + BClass" are subclasses of the AClass and BClass metaclasses, both upward and downward compatibility are guaranteed. Suppose that A defines two instance methods i-foo and i-bar. The i-foo method sends the c-bar message to the class of the receiver. The i-bar method is sent to a new instance by the c-foo method. Because the AClass and BClass metaclass hierarchy is parallel to the A and B class hierarchy, inter-level communication failure is avoided.

4.2 Example: Refactoring the Smalltalk-80 Boolean hierarchy

The Boolean hierarchy of SMALLTALK[9] is depicted in Figure 12. Boolean is an abstract class which defines a protocol shared by True and False. True and False are concrete classes that cannot have more than one instance. These properties (i.e. abstractness and having a sole instance) are implicit in SMALLTALK. Using the compatibility model, we refactor the Boolean hierarchy to emphasize them.

We first create "Boolean class", which is a compatibility metaclass. The second step consists of creating the property metaclass "Abstract + Boolean class", which enforces the Boolean class to be abstract. Finally, we build the Boolean class by instantiating the "Abstract + Boolean class" metaclass.

To refactor the False class, we first create the "False class" metaclass, as a subclass of "Boolean class" to ensure the compatibility. The second step consists of creating the property metaclass "SoleInstance + False class", which enforces the False class to have at most one instance. At last, we create the False class by instantiating the "SoleInstance + False class" metaclass. The True class is refactored in the same way. The result of rebuilding the whole hierarchy of Boolean is shown in Figure 13.

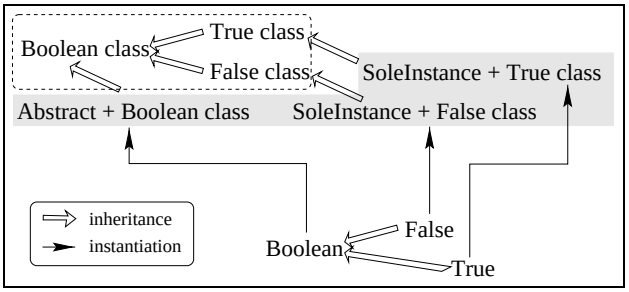


Figure 13: The Boolean hierarchy after refactoring

5 Reuse and composition within the compatibility model

We have experimented the compatibility model in NEOCLASSTALK[10] [Riv97], a fully reflective SMALLTALK. We quickly faced the need of *class property reuse and composition*. Indeed, unrelated classes belonging to different hierarchies can have the same properties, and a given class can have many properties.

In the previous section, both the True class and the False class have the same property: having a unique instance. Besides, we assigned only one property to each class of the Boolean hierarchy.

[9] We prefer this academic example to emphasize our ideas rather than a more complex example which should require a more detailed presentation.

[10] NEOCLASSTALK and all related papers can be downloaded from http://www.emn.fr/cs/object/tools/neoclasstalk/neoclasstalk.html

But, a class need to be assigned many properties. For example, the False class must not only have a unique instance, but it also should not be subclassed (such a class is final in JAVA terminology). So, we have to reuse and compose these class properties with respect to our compatibility model.

In this section, we propose an extension of our compatibility model that deals with reuse and composition of class properties. Any language where classes are treated as regular objects may integrate our extended compatibility model. NEOCLASSTALK has been used as a first experimentation platform.

5.1 Reuse of class properties

In SMALLTALK, since metaclasses behave in a different way than classes, they are defined as instances of a particular class, a *meta-metaclass*, called Metaclass. Metaclass defines the behavior of all metaclasses in SMALLTALK. For example, the name of a metaclass is the name of its sole instance postfixed by the word 'class'.

```
Metaclass≫name
  ↑thisClass name, ' class'
```

We take advantage of this concept of meta-metaclasses to reuse class properties. Since metaclasses implementing different properties have different behaviors, we need one meta-metaclass for each class property. Property metaclasses defining the same class property are instances of the same meta-metaclass.

When a property metaclass is created, the meta-metaclass initializes it with the definition of the corresponding class property. Thus, the code (instance variables, methods, ...) corresponding to the definition of the class property, is automatically generated. Reuse is achieved by creating property metaclasses defining the same class property as instances of the same meta-metaclass, i.e. they are initialized with the same class property definition (an example of such an initialization is given in section 5.4.2).

The root of the meta-metaclass hierarchy named PropertyMetaclass describes the default structure and behavior of property metaclasses. For example, the name of a property metaclass is built from the property name and the superclass name:

```
PropertyMetaclass≫name
  ↑self class name, '+', self superclass name
```

In the refactored Boolean hierarchy of section 4.2, both "SoleInstance + False class" and "SoleInstance + True class" define the property of having a unique instance. Reuse is achieved by defining both "SoleInstance + False class" and "SoleInstance + True class" as instances of SoleInstance, a subclass of PropertyMetaclass (see Figure 14).

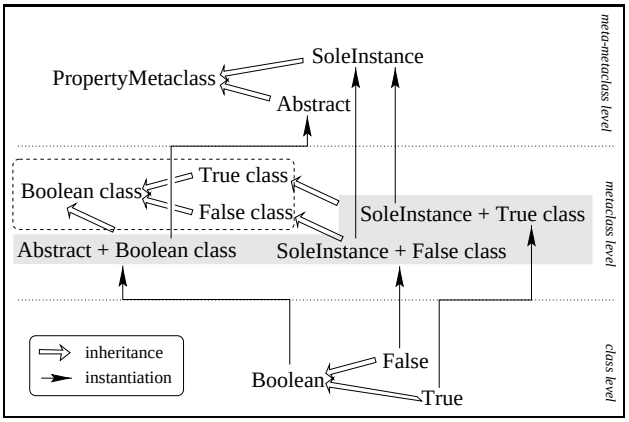


Figure 14: Reuse properties in the Boolean hierarchy

5.2 Composition of class properties

Since a given class can have many properties, the model must support the composition of class properties. We chose to use many property metaclasses organized in a single inheritance hierarchy, where each metaclass implements one specific class property.

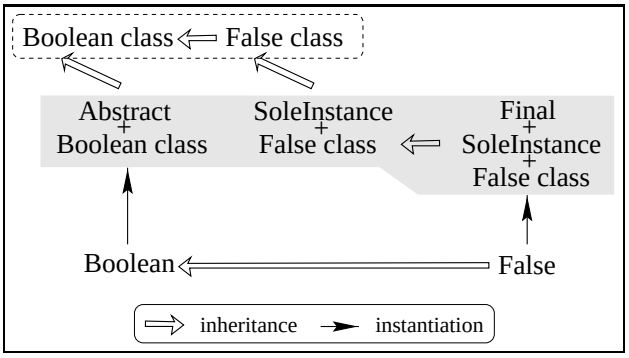


Figure 15: Assigning two properties to False

To illustrate this idea, we modify the instantiation link for the False class (see Figure 15). We define two property metaclasses, one for each property. The first property metaclass is "SoleInstance + False class", which inherits from the compatibility metaclass "False class". The second one is "Final + SoleInstance + False class", which is the class of False. It is defined as a subclass of "SoleInstance + False class". The resulting scheme respects the compatibility model: it allows the assignment of two specific properties to the False class and still ensures compatibility.

5.2.1 Conflict management

The solution of the property metaclasses composition issue is not trivial. Indeed, it is necessary to deal with conflicts that arise when composing different property metaclasses. When using inheritance to compose property metaclasses, two kinds of conflicts can arise: *name conflicts* and *value conflicts* [DHH+95].

Name conflicts happen when orthogonal property metaclasses define instance variables or methods which have the same name. Two property metaclasses are orthogonal when they define unrelated class properties. Name conflicts for both instance variables and methods are avoided by adapting the definition of a new property metaclass according to its superclasses. For example, although the two property metaclasses "SoleInstance + False class" and "SoleInstance + True class" define the same property for their respective instances (classes False and True), they may use different instance variable names or method names.

Value conflicts happen when non-orthogonal property metaclasses define methods which have the same name. Most of these conflicts are avoided by making the property metaclass hierarchy act as a cooperation chain, i.e. a property metaclass explicitly refer to the overridden methods defined in its superclasses[11]. Therefore, each property metaclass acts like a mixin [BC90].

[11] In NeoClasstalk, as in Smalltalk, this is achieved using the pseudo-variable **super**.

5.2.2 Example of cooperation between property metaclasses

Suppose that we want to assign two specific properties to the False class of Figure 16: (i) *tracing* all messages (Trace) and (ii) having *breakpoints* on particular methods (BreakPoint). These two properties deal with the message handling which is based in NeoClasstalk on the technique of the "method wrappers" described in [Duc98] and [BFJR98]. The executeMethod:receiver:arguments: method is the entry point to handle messages in NeoClasstalk, i.e. customizing executeMethod:receiver:arguments: allows a specialization of the message sending[12]. Thus, when the object false receives a message, the class False receives the message executeMethod:receiver:arguments:.

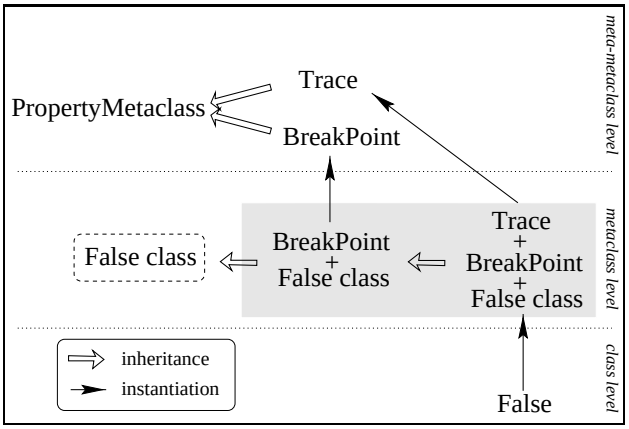


Figure 16: Composition of non-orthogonal properties

According to the inheritance hierarchy, (1) the trace is first done, then (2), by the use of **super**, the breakpoint is performed, and (3) a regular method application is finally executed (again called using **super**).

- (3) StandardClass≫executeMethod: method receiver: rec arguments: args
  ...
- (2) BreakPoint+False class≫executeMethod: method receiver: rec arguments: args
  method selector == stopSelector
  ifTrue: [self halt: 'Breakpoint for ', stopSelector].
  ↑super executeMethod: method receiver: rec arguments: args

[12] A default executeMethod:receiver:arguments: method is provided by StandardClass (the root of all metaclasses in NeoClasstalk) which just applies the method on the receiver with the arguments.

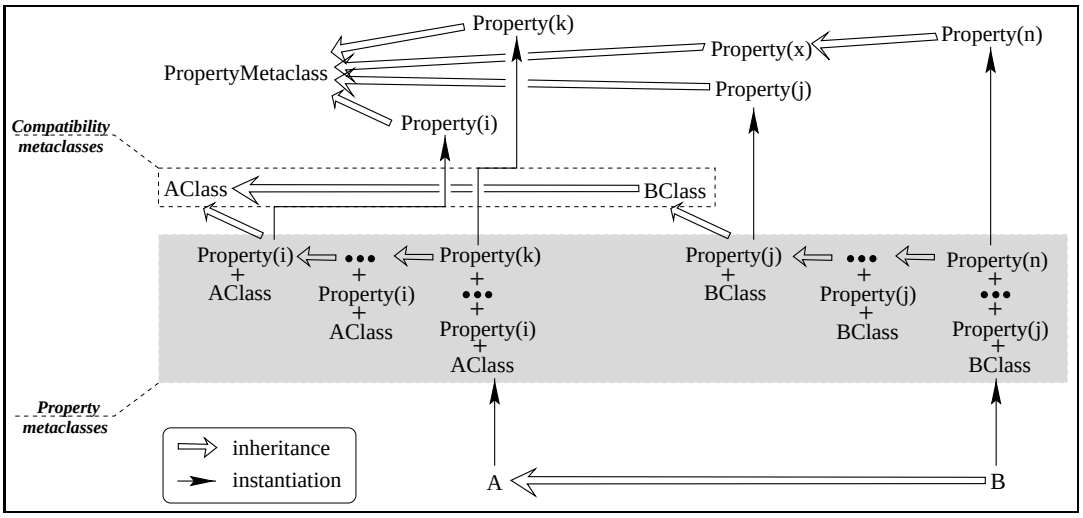


Figure 17: The Extended Compatibility Model

- (1) Trace+BreakPoint+False class≫executeMethod: method receiver: rec arguments: args

  self transcript show: method selector; cr.
  ↑super executeMethod: method receiver: rec arguments: args

5.3 The extended compatibility model

Generalizing previous examples allows us to define the *extended compatibility model* (see Figure 17) which enables reusing and composing class properties. Each property metaclass defines the instance variables and methods involved in a unique property. Property metaclasses specific to a given class are organized in a single hierarchy. The root of this hierarchy is a subclass of a compatibility metaclass[13]. Each property metaclass is an instance of a meta-metaclass which describes a specific class property, allowing its reuse.

Metaclass creation, composition and deletion are managed automatically with respect to the extended compatibility model. Each time a new class is created, a new compatibility metaclass is automatically created. This can be done in the same way that SMALLTALK builds its parallel metaclass hierarchy. The assignment of a property to this class results in the insertion of a new metaclass into its property metaclass hierarchy. This insertion is made in two steps[14]:

1. first, the new property metaclass becomes a subclass of the last metaclass of the property metaclass hierarchy;
2. then, the class becomes instance of this new property metaclass.

NEOCLASSTALK provides protocols for dynamically changing the class of an object (changeClass:) and the superclass of a class (superclass:) [Riv97]. Thus, the implementation of these two steps is immediate in NEOCLASSTALK, and is provided by the composeWithPropertiesOf: method.

```
PropertyMetaclass≫composeWithPropertiesOf: aClass
    self superclass: aClass class.
    aClass changeClass: self.
```

5.4 Programming within the extended compatibility model

We distinguish two kinds of programmers: (i) "base level programmers" who implement applications using the language and development tools, and (ii) "meta level programmers" for whom the language itself is the application.

[13] This single hierarchy may be compared to an explicit linearization of property metaclasses composed using multiple inheritance [DHHM94].

[14] The removal of a property metaclass is done in a symmetrical way.

5.4.1 Base Level Programming

To make our model easy to use for a "base-level programmer", the NeoClasstalk programming environment includes a tool that allows one to assign different properties to a given class using a Smalltalk-like browser (see Figure 18). These properties can be added and removed at run-time. The metaclass level is automatically built according to the selection of the "base-level programmer".

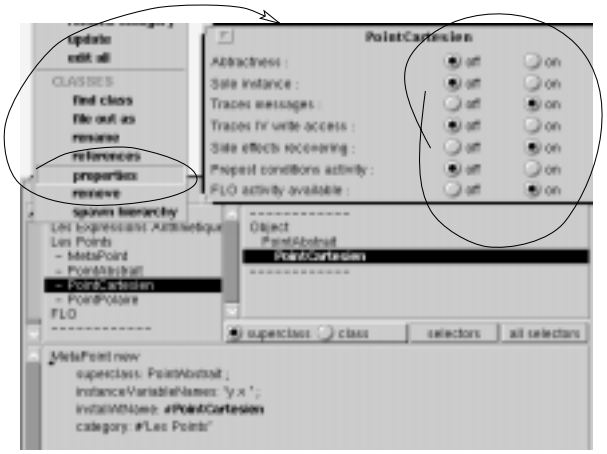


Figure 18: Specific properties assigned to a class using a browser

5.4.2 Meta Level Programming

In order to introduce new class properties, "meta-level programmers" must create a subclass of the PropertyMetaclass meta-metaclass. This new meta-metaclass stores the instance variables and the methods that should be defined by its instances (property metaclasses). When this new meta-metaclass is instantiated, the previous instance variables are added to the resulting property metaclass and the methods are compiled[15] at initialization time[16].

For example, the evaluation of the following expression creates a property metaclass — instance of the meta-metaclass Trace — that assigns the trace property to the True class.

```
Trace new composeWithPropertiesOf: True
```

In order to achieve the trace, messages must be captured and then logged in a text collector. Therefore, property metaclasses instances of Trace must define an instance variable (named transcript) corresponding to a text collector and a method that handles messages. Message handling is achieved using the executeMethod:receiver:arguments: method which source code was already presented in 5.2.2. These definitions are generated when the property metaclasses are initialized, i.e. using the initialize method of the Trace meta-metaclass:

```
Trace≫initialize
  super initialize.
  self instanceVariableNames:' transcript '.
  self generateExecuteMethodReceiverArguments.
```

6 Conclusion

Considering classes as first class objects organizes applications in different abstraction levels, which inevitably raises upward and downward compatibility issues. Existing solutions addressing the compatibility issues (such as Smalltalk) do not allow the assignment of specific properties to a given class without propagating them to its subclasses.

The *compatibility model* proposed in this paper addresses the compatibility issue and allows the assignment of specific properties to classes without propagating them to subclasses. This is achieved thanks to the separation of the two involved concerns: compatibility and class properties. Upward and downward compatibilities are ensured using the *compatibility metaclass* hierarchy that is parallel to the class hierarchy. The *property metaclasses*, allowing the assignment of specific properties to classes, are subclasses of these compatibility metaclasses. Therefore, we can take advantage of the expressive power of metaclasses to define, reuse and compose class properties in a environment which supports *safe metaclass programming*.

Class properties improve readability, reusability and quality of code by increasing *separation of concerns* [HL95] [Lie96] [KLM+97]. Indeed, they

[15] A faster solution consists of doing the compilation only once, resulting in proto-methods [Riv97]. Thus, when the property metaclass gets initialized, proto-methods are "copied" into the method dictionary of the property metaclass, allowing a fast instantiation of meta-metaclasses.

[16] This assumes that initialization is part of the creation process, which is true in almost every language. This is traditionnally achieved in Smalltalk by the redefinition of new into super new initialize [SKT96].

allow a better organization of class libraries and frameworks for designing reliable software. We are strongly convinced that our compatibility model enables separation of concerns based on the metaclass paradigm. Therefore, it promotes building reliable software which is easy to reuse and maintain.

Acknowledgments

The authors are grateful to Mathias Braux, Pierre Cointe, Stéphane Ducasse, Nick Edgar, Philippe Mulet, Jacques Noyé, Nicolas Revault, and Mario Südholt for their valuable comments and suggestions. Special thanks to the anonymous referees who provided detailed and thought-provoking comments.

References

[BC89] Jean-Pierre Briot and Pierre Cointe. Programming with Explicit Metaclasses in Smalltalk. In *Proceedings of OOPSLA'89*, pages 419–431, New Orleans, Louisiana, USA, October 1989. ACM.

[BC90] Gilad Bracha and William Cook. Mixin-based Inheritance. In *Proceedings of ECCOP/OOPSLA'90, Ottawa, Canada*, pages 303–311, October 1990.

[BFJR98] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. Wrappers to the Rescue. In *Proceedings of ECOOP'98*, July 1998.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and Distribution in Object Oriented Programming. *ACM Computer Surveys*, 1998. to appear.

[BSLR96] Noury Bouraqadi-Saâdani, Thomas Ledoux, and Fred Rivard. Metaclass Composability. In *ECOOP'96 workshop : "Composability Issues in Object Orientation"*, Linz, Austria, July 1996.

[Coi87] Pierre Cointe. Metaclasses are First Class: the ObjVlisp Model. In *Proceedings of OOPSLA'87*, pages 156–167, Orlando, Florida, USA, October 1987. ACM.

[DF94a] Scott Danforth and Ira R. Forman. Derived Metaclasses in SOM. In *Proceedings of TOOLS EUROPE'94*, pages 63–73, Versailles, France, 1994.

[DF94b] Scott Danforth and Ira R. Forman. Reflections on Metaclass Programming in SOM. In *Proceedings of OOPSLA'94*, pages 440–452, October 1994.

[DHH+95] Roland Ducournau, Michel Habib, Marianne Huchard, Marie-Laure Mugnier, and Amedeo Napoli. Le point sur l'héritage multiple. *Techniques et Sciences Informatique*, 14(3):309–345, 1995. (In french).

[DHHM94] Roland Ducournau, Michel Habib, Marianne Huchard, and Marie-Laure Mugnier. Proposal for a Monotonic Multiple Inheritance Linearization. In *Proceedings of OOPSLA'94*, pages 164–175, Portland, Oregon, October 1994.

[Duc98] Stéphane Ducasse. Evaluating Message Passing Control Techniques in Smalltalk. *Journal of Object-Oriented Programming*, 1998. to appear.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, The language and its implementation*. Addison Wesley, Readings, Massachusetts, 1983.

[Gra89] Nicolas Graube. Metaclass Compatibility. In *Proceedings of OOPSLA'89*, pages 305–315, New Orleans, Lousiana, October 1989.

[HL95] Walter L. Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.

[KAJ+93] Gregor Kiczales, J. Michael Ashley, Luis H. Rodriguez Jr., Amin Vahdat, and Daniel G. Bobrow. *"Object-Oriented Programming: The CLOS Perspective" edited by Andreas Pæpcke*, chapter Metaobject Protocols: Why We Want Them and What Else They Can Do, pages 101–118. The MIT Press, Cambridge, Massachusetts, 1993.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97*, number 1241 in LNCS, pages 220–242. Springer-Verlag, June 1997.

[LC96] Thomas Ledoux and Pierre Cointe. Explicit Metaclasses As a Tool for Improving the Design of Class Libraries. In *Proceedings of ISOTAS'96, LNCS 1049*, pages 38–55, Kanazawa, Japan, March 1996. Springer-Verlag.

[Led98] Thomas Ledoux. *Reflection and Distributed Systems : an Experiment with CORBA and Smalltalk*. PhD thesis, Université de Nantes, March 1998. (In french. Réflexion dans les systèmes répartis : application à CORBA et Smalltalk).

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[Mae87] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, pages 147–155, Orlando, Florida, 1987. ACM.

[MMC95] Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a Methodology for Explicit Composition of MetaObjects. In *Proceedings of OOPSLA'95*, pages 316–330, Austin, Texas, October 1995.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. *Research Directions in Concurrent Object-Oriented Programming*, chapter Analysis of inheritance anomaly in object-oriented concurrent programming languages. MIT Press, 1993.

[Riv96] Fred Rivard. A New Smalltalk Kernel Allowing Both Explicit and Implicit Metalclass Programming. OOPSLA'96, Workshop : Extending the Smalltalk Language, October 1996.

[Riv97] Fred Rivard. *Object Behavioral Evolution Within Class Based Reflective Languages*. PhD thesis, Université de Nantes, June 1997. (In french. Évolution du Comportement des Objets dans les Langages à Classes Réflexifs).

[SKT96] Suzanne Skublicks, Edward J. Klimas, and David A. Thomas. *Smalltalk with Style*. Prentice Hall, 1996.

[SOM93] IBM. *SOMobjects Developer Toolkit Users Guide release 2.0*, second edition, June 1993.

[Zim96] Chris Zimmermann, editor. *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.

Towards a New Model of Abstraction in Software Engineering

Gregor Kiczales

Published in proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures, 1992.

Appears in IMSA'92 Proceedings (Workshop on Reflection and Meta-level Architectures).

Towards a New Model of Abstraction in the Engineering of Software

Gregor Kiczales
Xerox Palo Alto Research Center*

We now come to the decisive step of mathematical abstraction: we forget about what the symbols stand for… [The mathematician] need not be idle; there are many operations he can carry out with these symbols, without ever having to look at the things they stand for.

Hermann Weyl, "The Mathematical Way of Thinking"

(This appears at the beginning of the *Building Abstractions With Data* chapter of "Structure and Interpretation of Computer Programs" by Harold Abelson and Gerald Jay Sussman.)

This is an abridged version of a longer paper in preparation. The eventual goal is to present, to those outside of the reflection and meta-level architectures community, the intuitions surrounding open implementations and the use of meta-level architectures, particularly metaobject protocols, to achieve them.

The view of abstraction on which software engineering is based does not support the reality of practice: it suggests that abstractions hide their implementation, whereas the evidence is that this is not generally possible. This discrepancy between our basic conceptual foundations and practice appears to be at the heart of a number of portability and complexity problems.

Work on metaobject protocols suggests a new view, in which abstractions do expose their implementations, but do so in a way that makes a principled division between the functionality they provide and the underlying implementation. By resolving the discrepancy with practice, this new view appears to lead to simpler programs. It also has the potential to resolve important outstanding problems surround reuse, software building blocks, and high-level programming languages.

Abstraction In Action

I want to start by talking about the current view of abstraction in software engineering: how we use it, what the principles are, what the terminology is and what it does for us. Rather than attempting any sort of formal definition, I will just use an example. I will talk about the implementation of a familiar system, using familiar terms of abstraction, with the goal of getting the terminology I am going to use out on the table.

Consider the display portion of a spreadsheet application. In practice, the implementation would be based on "layers of abstraction" as shown in Figure 1. The spreadsheet would be implemented on top of a window system, which would in turn be implemented on top of an operating system and so on down (not very far) to the machine.

The horizontal lines in the figure are commonly called "abstraction barriers," "abstractions" or "interfaces." Each provides useful functionality while hiding "implementation details" from the client above.[1] To the degree that an abstraction provides powerful, composable functionality, and is free of implementation issues, we call it "clean" or "elegant." In the particular case of the window system, the abstraction would provide the ability to make windows, arrange them on the screen, display in them, track the mouse etc. Issues such as how the windows are represented in memory and how the mouse is tracked would be hidden as implementation details.

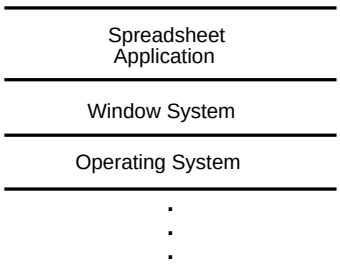


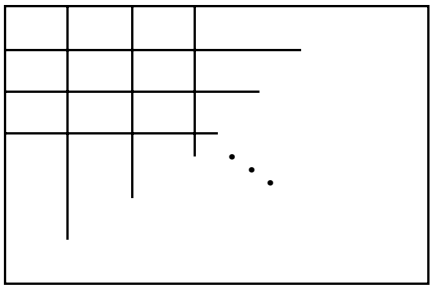
Figure 1: The layers of abstraction in the display portion of a spreadsheet application.

There seem to be (at least) three basic principles underlying our view of abstraction:

- The first, and most important, has to do with management of complexity. In this sense, abstraction is a primary concept in all engineering disciplines and is, in fact, a basic property of how people approach the world. We simply can't cope with the full complexity of what goes on around us, so we have to find models or approximations that capture the salient features we need to address at a given time, and gloss over issues not of immediate concern.
- Second, is a convention that a primary place to draw an abstraction boundary is between those aspects of a system's behavior that are particular to a particular implementation vs.

*3333 Coyote Hill Rd., Palo Alto, CA 94304; (415)812-4888; Gregor@parc.xerox.com.

[1] In this paper, the terms *client* and *application* are used to refer to a piece of software that makes use of some lower-level software; i.e. the spreadsheet is a client of the window system.



```
for i = 1 to 100
 for j = 1 to 100
  mkwindow(100, 100, i*100, j*100)
 end
end
```

Figure 2: A spreadsheet looks like a rectangular array of cells. The simplest way to implement it is to use one window for each cell.

those aspects of its behavior that common across all implementations.

- Third, is a sense that not only is the kind of abstraction boundary that arises from the second principle useful, it is in fact the *only* one it appropriate to give to clients. That is, we believe that issues of an interface's implementation are not of concern to, and should be completely hidden from, clients.

(Note that the first of these is so basic that it rarely, at least in our field, gets explicit attention. But arguably, what our informal notions of elegance, cleanliness, and orthogonality are about is the degree to which an abstraction includes those issues which are important without including any that are not.)

Layered on top of these three principles are our goals of portability, reusability and in fact the whole concept of system software. The idea has been that by taking commonly useful, "basement-level," functionality — memory allocators, file systems, window systems, databases, programming languages etc. — giving it a general-purpose interface, and isolating the client from the implementation, we could make it possible for a wide range of clients to use the abstraction without caring about the implementation. Portability stems in particular from isolating the client from implementation details; this makes it possible to have other implementations of the abstraction which the client code can be ported to. Reuse stems in particular from making the abstraction general-purpose; the more general it is, the wider a variety of clients that can use it.

In line with this story, it should be an easy matter to implement a spreadsheet on top of a clean, powerful window system. What is needed is just a rectangular array of cells; we need to be able to display and type in each cell independently; and we need to know when the mouse is clicked over a cell. Since this is exactly the functionality a window system provides, the simplest way to code the spreadsheet is to use one window for each cell. This takes advantage of the high-level window system abstraction to cleanly express what is desired, and makes maximal reuse of the existing window system code. A program written in this fashion is shown in Figure 2.

This is abstraction at its best. The code is simple, clear, and we can read it without having to know anything about the inner workings of the underlying implementation. Abstraction here is doing just what our small minds need: making it possible for us to think about important properties of our program — its behavior — without having to think about the entirety of the machinations the underlying hardware is having to perform to get it to run.

As wonderful as this may sound, few experienced programmers would be surprised if this code didn't quite work. That is, it might work, but its performance might be so bad as to render it, in any practical sense, worthless. This can happen if the window system implementation is not tuned for this kind of use. As part of writing the window system, the implementor is faced with a number of tradeoffs, in the face of which they must make decisions. No matter what they do, the window system will end up tuned for some applications and against others. In this case, the implementor might have assumed that 25 to 50 windows was a more typical number for an application to use than $10,000$. They might also have assumed that the typical configuration of windows would have an irregular, rather than highly-regularized, geometry. Implementation decisions based on these assumptions, once made, become locked away behind the abstraction barrier as implementation details.

We are all familiar with this sort of situation, and probably have a good sense of how we would respond. But, stepping back and looking through it carefully is fruitful. There are several points to notice: (i) While the simple program in Figure 2 may not perform adequately, its intended behavior is perfectly clear. In other words, the window system abstraction itself is adequate for expressing the behavior the client programmer is after. (ii) The fact that the implementation will fail to provide adequate performance is nowhere evident in the client code. That is, the window system abstraction is not, in and of itself, betraying these properties of the implementation. (It's also likely to be the case that this performance property can't be gleaned from reading the window system documentation.) (iii) So, predicting and/or understanding the performance properties of this program can only be done with knowledge of internal aspects of the window system implementation — the so-called "hidden implementation details." (iii) Finally, it is relatively easy to imagine an implementation of the window system in which this code would perform adequately. Moreover, such an implementation might not be all that different from the existing one.[2]

What is clear then is that there is a basic discrepancy between our existing view of abstraction and the reality of day-to-day programming. We say that we design clean, powerful abstractions that hide their implementation, and then use those abstractions, without thinking about their implementation, to build higher-level functionality. But, the reality is that the implementation cannot always be hidden, its performance characteristics can show through in important ways. In fact, the client programmer is well aware of them, and is limited by them just as they are by the abstraction itself.

Looking ahead, the idea underlying the new abstraction frame-

[2] The issue is whether a window is a large structure, which locally caches derived properties, or whether it is a small structure, which continually recomputes derived properties from its parent (i.e. does a window know its position, or does it have to ask its parent). In the latter approach, a great deal of memory could be saved on the cell windows. Each could be as small as a word, or even take up no storage at all in more radical architectures. In addition to saving memory, certain operations could be supported more efficiently. For example, to tell which cell the mouse was over, the main window could, because of the regular geometry of the cells, do simple arithmetic rather than using the more general mechanism of polling all the cell windows.

work will be to try and preserve what is good and essential about our existing abstraction framework — essentially the first two bulleted principles — while seeking to address the conflict between the third basic principle and the reality of practice. In doing this, the strategy will be to try and take advantage of the fact that very often, as in this example, our abstractions themselves are sufficiently expressive and our implementations may only be deficient in small ways. What we will end up doing is "opening up the implementation," but doing so in a principled way, so that the client doesn't have to be confronted with implementation issues all the time, and, moreover, can address some implementation issues without having to address them all.

Outline of the Paper

The rest of this paper expands this basic argument for open implementations. First, the consequences of the deficiency in our current abstraction framework are discussed, using both the window system example and an example from high-level programming languages. The application of metaobject protocol technology to these problems is discussed, and the new model of abstraction, drawn out from the intuitions underlying the metaobject protocol work, is presented. Given the new model, it is possible to identify a wide range of other work in the software engineering community which not only seems to confirm the intuition that the old model of abstraction is invalid but which in fact seems to be headed in the same direction as the framework presented in this paper. Finally there is a discussion of what future work might be required as part of continuing to develop this new abstraction framework.

The Origins of Complexity and Portability Problems

Cases like the spreadsheet application, where an abstraction itself is adequate for the client's needs but the implementation shows through and is in some way deficient are common. The machinations the client programmer is forced into by these situations make their code more complex and less portable. These machinations fall into two general categories: (i) Reimplementation of the required functionality, in the application itself, with more appropriate performance tradeoffs; and (ii) coding "between the lines."

Reimplementation of functionality is what would mostly likely happen in this case. The spreadsheet programmer would end up writing their own "little window system," that could draw boxes on the screen, display in them, and handle mouse events. Reimplementing this way would allow the the programmer to ensure that the performance properties met their particular needs. As suggested by Figure 3, reimplementing part of the underlying functionality this way increases the size of the application, and, therefore, the total amount of code the programmer must be responsible for.

In addition to making the application strictly larger, reimplementation of underlying functionality can also cause the rest of the application — the code that simply uses the reimplemented functionlity — to become more complex. This happens if for some reason the newly implemented functionality cannot be used as elegantly as the original underlying functionality. This in turn can happen if, for any reason, the programmer cannout manage to slide the new implementation in under the old interface.

Once the programmer is forced away from being able to use the old interface, and into the problem of designing one of their own, its quite likely they won't do as good a job. Simply put, the application programmer doesn't have the time (even if they do have the interest) to design the new interface as cleanly as might be nice.

(As an aside, its worth point out that even if the interface ends up being just as (or more) elegant, one of the primary purposes of high-level standardization — to be able to easily read each other's code — has been defeated.)

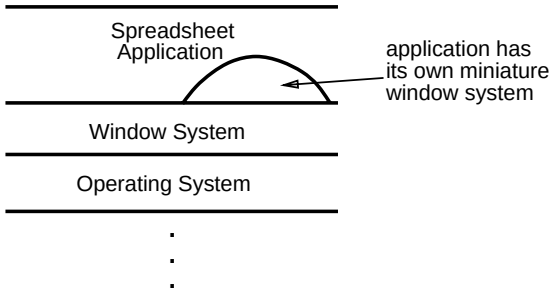


Figure 3: The spreadsheet application after being revised around the performance problems of the window system. The reimplementation of functionality which could not be reused from the window system appears as a 'hematoma' in the application. Each such hematoma increases the size of the application. In addition, the rest of the application can get more complex when it is rewritten to use the new functionality.

Coding between the lines is happens when the application programmer writes their code in a particularly contorted way in order to get better performance. A classic example is in the use of virtual memory. In a program that allocates a number of objects, there is often a order to allocating those objects that is "natural" to the program. But, if there get to be a lot of objects, and paging behavior becomes critical, people will often rewrite the application to "allocate the objects close to each other" and thereby get better performance. This is coding between the lines because although the documented virtual memory abstraction makes no mention about the physical locality of objects, the programmer manages to contort their code enough to "speak to" the inside of the implementation and get the performance they want.

When programmers are forced into these situations, their applications become unduly complex and, more importantly, even less portable. It is easiest to see how this happens by starting with a hypothetical prototype implementation, coded on a machine that was fast enough that the programmer was not forced into these sins, and then looking at what happens as the application is moved to a delivery platform. (In reality, code is usually "optimized" when it is first written, but this simpler case makes what happens more clear.)

The original implementation is simple, clear and makes the greatest re-use of the underlying abstractions (i.e. the simple spreadsheet implementation). But, when it comes time to move it to the delivery platform, a number of performance problems come up that must be solved. A wizard is brought in, and through tricks like those mentioned above, manages to improve the performance of the application. Effectively, the wizard *convolves* the original simple code with their knowledge of inner workings of the delivery platform.[3] (The term convolves is chosen to suggest that, as a result of the convolution, properties of the code which had been well localized become duplicated and spread out.) In the process, the code becomes more complex and *implicitly* conformant to the delivery platform. When it comes time to move it to another platform, the code is more dif-

[3] Note that putting it this way explains why the informal term "wizard" refers to someone who not only is good at working with a given abstraction (i.e. a window system), but who is also intimately familiar with the *inner workings* of the implementation. Simply put, the wizard is someone who specializes at doing what our traditional abstraction story says should never happen.

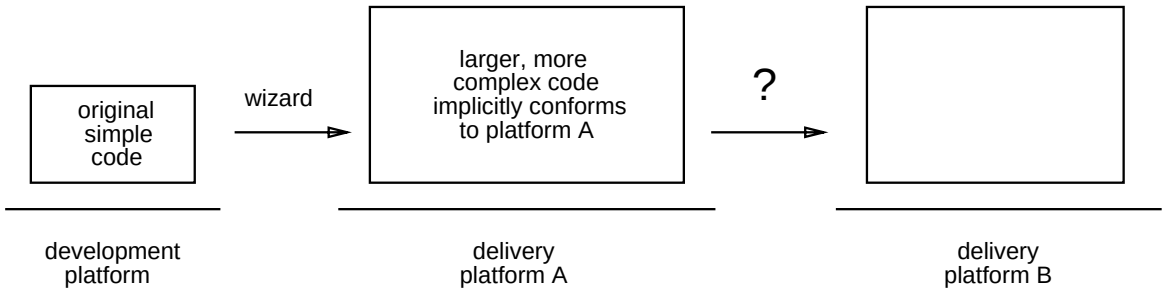


Figure 4: When an application is originally written on a fast machine, the code can start out being simple. To port the code to a delivery platform a wizard — someone who understands the inner workings of the delivery platform — is brought in to tune the code. The application gets larger and more complex, and above all it becomes implicitly adapted to the delivery platform. It is then even more difficult to move it to another platform.

ficult to work with, and because of the implicit conformance, it is difficult to tell just why things are the way they are. This is shown in Figure 4.

High-Level Languages

> I found a large number of programs perform poorly because of the language's tendency to hide "what is going on" with the misguided intention of "not bothering the programmer with details."
> *N. Wirth, "On the Design of Programming Languages," [Wir74]*

I want to look next at the domain of high-level programming languages, where the reflection and meta-level architectures community has done a lot of work to address these kinds of problems. First, I will show, using the Common Lisp Object System (CLOS) [Kee89, Ste90], how the same sorts of problems can come up. I will then show how those problems are addressed by the CLOS Metaobject Protocol (CLOS MOP) [BKK+86, Kic92, KdRB91]. From there, it will be possible to generalize and present the new model of abstraction.

Consider the following CLOS class definitions:

```
(defclass position ()
     (x y))

(defclass person ()
     (name age address ...))
```

The class `position` might be part of a graphics application, where the instances are used to represent the position of the mouse as it moves. The class defines two slots, `x` and `y`.[4] The behavior of the application is such that there will be a very large number of instances, both slots will be used in every instance and access to those slots should be as fast as possible.

The second definition, `person`, might come from a knowledge representation system, where the instances are being used as frames. In this case, the class defines a thousand slots, corresponding to the many properties of people which might be known. As with the class `position`, the behavior of the application means that a couple of things are known: there will be a very large number of instances; but in any given instance only a few slots will actually be used.

[4] *Slot* is the CLOS term for the fields of an instance.

Clearly, the ideal instance implementation strategy is different for the two classes. For `position`, an array-like strategy would be ideal; it provides compact storage of instances, and rapid access to the `x` and `y` slots. For `person`, a hash-table like strategy would be more appropriate, since it isn't worth allocating space for a slot until it is known that it will be used. This makes access slower, but it is a worthwhile tradeoff given a large number of instances.

What is most likely to be the case, in a run-of-the-mill CLOS implementation sans MOP,[5] is that the implementor will have chosen the array-like strategy. The prospective author of the `person` class will find themselves in a situation very much like that of the spreadsheet implementor above: While the CLOS language abstraction itself is perfectly adequate to express the behavior they desire, supposedly hidden properties of the implementation — the instance representation strategy — are critically getting in the way.

Metaobject Protocols

In this abridged version of the paper, this section is elided, since it would be redundant for IMSA'92 Workshop attendees.

For the eventual audience of this paper, the goal of this section will be to sketch the mechanics of metaobject protocols, and to show how, by careful design, a metaobject protocol can be used to allow the user to control critical aspects of the language implementation strategy, without overwhelming them with what truely are implementation details.

This section will also discuss, more briefly, how metaobjects protocols can be used to provide the user control over the semantics, or behavior of a language.

In addition to the CLOS Metaobject Protocol, other MOPs and reflective languages which might be discussed in this section include TELOS [Pad92], ABCL/R2 [MWY91], 3-KRS[Mae87], Anibus [Rod91, Rod92], Sartor [Ash92] and Ploy [Vah92].

A New Model of Abstraction

In the metaobject protocol approach, the client ends up writing two programs: a base-language program and an (optional) meta-language program. The base-language program expresses, the desired behavior of the client program, in terms of the functionality provided by the underlying system. The meta-language program can customize

[5] At this point all CLOS vendors I know of have plans to provide a metaobject protocol. So, a CLOS implementation sans MOP is more of a rhetorical tool than a reality.

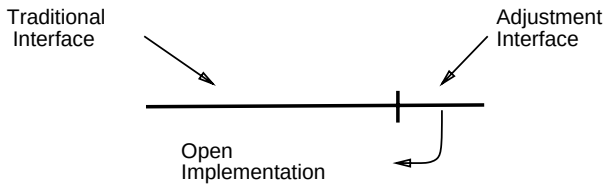


Figure 5: The dual-interface framework supports the notion of an open implementation. The client first writes a base-program, and then, if necessary, writes a meta-program to customize the underlying implementation to meet the base-program's needs. The curved arrow under the meta-level interface is intended to remind us that it provides access to what have traditionally been internal properties of the implementation.

particular aspects of the underlying system's implementation so that it better meets the needs of the base-language program.

What begins to emerge is a "dual-interface" picture something like that shown in Figure 5. A high-level system (i.e. CLOS) presents two coupled interfaces: base- and meta-level. The base-level interface looks like the traditional interface any such system would present. It provides access to the system's functionality in a way that the application programmer can make productive use of and which does not betray implementation issues. The client programmer can work with it without having to think about the underlying implementation details.

But, for those cases where the underlying implementation is not adequate, the client has a more reasonable recourse. The meta-level interface provides them with the control they need to step in and customize the implementation to better suit their needs. That is, by owning up to the fact that users needs access to implementation issues (i.e. instance implementation strategy), and providing an explicit interface for doing so, the metaobject protocol approach manages to retain what is good about the first two principles of abstraction.

It is much too early to attempt to provide a complete account of dual interface abstractions, how to design them, how to use them or what technologies can be used to support them. But, based on experience with metaobject protocols and other recent reflective and meta-level architectures, some basic comments can be made.

First off, it appears that the design of base-level interfaces can be done using existing skills. As mentioned above, we have become quite good at designing interfaces that do not themselves betray the implementation. We should be able to make base-level interfaces even more clean because we will now have a principled place to put implementation issues that the client must have access to — the meta-level interface.

Mastering the design of meta-level interfaces, and, importantly, the coupling between base- and meta-level interfaces is going to take a great deal more work. But we can enumerate four preliminary, and closely interrelated, design principles: scope control, conceptual separation, incrementality and robustness.

- *Scope control* means that when the programmer uses the meta-level interface to customize the implementation, they should be given appropriate control over the scope of the specialization. One can imagine various kinds of scope control. In the CLOS example above, the programmer wants to be able to say both that they only want to affect the instance representation strategy, and that only want certain classes (i.e. `person`) to be affected. Other classes, particularly classes that are part of other applications, should not be affected. The window system case is analogous; some windows should use the implementation tuned for spreadsheets whereas others should use the default implementation.

- *Conceptual separation* means that it should be possible to use the meta-level interface to customize particular aspects of the implementation without having to understand the entire meta-level interface. So, for example, the client programmer who wants to customize the instance implementation strategy shouldn't also have to be concerned with the method dispatch mechanism. This of course is difficult, since implementation issues can sometimes have surprisingly far-reaching effects. The challenge, as discussed in [LKRR92], is to come up with a sufficiently fine-grained model of the implementation.

- *Incrementality* means that the client who decides to customize some aspect of the implementation tradeoffs wants to do just that: *customize those properties*. They don't want to have to take total responsibility for the implementation and they don't want to end up having to write a whole new implementation from scratch. It must be possible for them to say just what it is they want to have be different, and then automatically reuse the rest of the implementation. This is the salient difference between the more recent reflective systems (CommonLoops, 3-KRS and beyond) and the original 3-Lisp work: by using object-oriented techniques, it has been possible to support the incremental definition of new implementations (interpreters, runtimes etc.) using subclass specialization. (More is said about object-oriented techniques later in the paper.)

- *Robustness*[6] simply means that bugs in a client's meta-program should have appropriately limited effect on the rest of the system. To date, much of the work in the reflection and metaobject protocols community has provided only limited robustness, either by checking the results of functional protocols, or absorbing it from the underlying runtime in imperative[7] protocols. But these approaches significantly restrict the power of the protocol. In more recent work, we are beginning to explore the use of more declarative protocols, combined with partial evaluation techniques to recover the performance loss [Ash92]. This remains a major open problem.

These four principles are not entirely orthogonal. Take for example, support for defining a new instance implementation strategy in the CLOS MOP. While it is easy to say that it does well on each of the first three, it is difficult to point to particular parts of the CLOS Metaobject Protocol design and say "Scope control comes from here and incrementality comes from here." Instead, they all seem to be intertwined; they all have to do with various kinds of "locality."

In fact, much of the recent work on reflective systems can be seen as experiments with locality. Group-wide reflection, one metaobject per object languages, metaobjects on a per-class basis, reifying the generic function rather than letting the class handle method dispatch — all of these provide different kinds of locality control [Coi87, IMWY91, IO91, Mae87, MWY91, WY90, WY91] (as well as many of the other papers appearing in this workshop). What is clear is that there is no one right or most elegant metaobject structure, each has relative costs and advantages, and we need to keep experimenting to learn about how to handle locality this way. (There is more to say about the subject of locality as the paper progresses.)

[6] This term is somewhat problematic, as it has particular technical meaning in some communities. Later in the paper, it will become clear that what is needed is a term that in some sense spans (at least) all of safety, reliability and security.

[7] In [KdRB91] we used the term *procedural* instead of *imperative*.

It is also possible to make a basic comment about the way the designer of a dual-interface abstraction — or any open implementation — works: iteratively. They start with a traditional abstraction (i.e. a window system or CLOS), and gradually add a meta-level interface as it becomes clear what kinds of ways a close implementation can cause problems for the users. Moreover, it isn't a good idea to try and make the first version of a new kind of system open in this sense. Opening the implementation critically depends on understanding not just one implementation the clients might want, but also the various kinds of variability around that point they might want. In this mode of working, user bug-reports and complaints about previous versions of the system take on an important value. We can look for places where users complained that they wanted to do X, but the implementation didn't support it; the idea is to add enough control in the meta-level interface to make it possible to customize the implementation enough to make X viable. (In fact, in work on the CLOS Metaobject Protocol, we spent a lot of time thinking about these kinds of bug reports.)

Another way of thinking about the design of meta-level interfaces can be found in a 1980 paper by Mary Shaw and Wm. Wulf [SW80], in which they present an interesting (and prescient) intuition about the situation: "Traditionally, the designers and implementors of programming languages have made a number of decisions about the nature and representation of language features that… are unnecessarily preemptive." By preemptive, they mean a decision, on the part of the implementor (or the language designer), that preempts the programmer from being able to use a language feature in a way that otherwise appears natural. (A specific example they give has to do with the choice of representation of arrays.) Their paper is focused primarily on programming language implementations, but the notion of preemption is a powerful one to work with when thinking about any kind of meta-level interface. It suggests that anytime we find ourselves saying "well, I'll implement this feature a particular way because I think *most* users will do X," we should immediately think about the other users, the ones whose options we are about to preempt, and how, using a meta-level interface, we might allow them to customize things so they can do other than X.

A Recap

At this point, it is possible to give a capsule summary of the argument so far:

> In practice, high-level abstractions often cannot hide their implementations — the performance characteristics show through, the user is aware of them, and would be well-served by being able to control them. This happens because making any concrete implementation of a high-level system requires coming to terms with a number of tradeoffs. It simply isn't possible to provide a single, fixed, closed implementation of such a system that is "good enough" that all prospective users will be happy with it. In other words, the third principle of abstraction presented above appears to be invalid, at least in actual practice.
>
> Work on metaobject protocols and other meta-level architectures suggests a new abstraction framework that better addresses the need for open implementations. Under this framework the abstraction presented by a system is divided into two parts: one that provides functionality in a traditional way and another that provides control over the internal implementation strategies supporting that functionality. This approach retains the first two principles of the old abstraction framework, dropping only the third.

Looking At Other Work

With this summarization in mind, it becomes possible to look for other areas where open implementations and dual interface abstractions could be particularly advantageous. In doing so, what we are trying to assess is how much of the argument presented above applies in domains other than high-level programming languages. Clearly we would expect the basic argument for open implementations to move across — after all, we started with a window-system not a programming language. On the other hand, we may or may not expect the concept of metaobject protocols (or at least our current notion of them) to move to memory systems or schedulers. And in between those two levels are the crucial intermediary notions of locality, reflection, meta, and object-oriented programming. By looking at other examples, we hope to get a better sense of the overall picture and where each of these important concepts fits in it.

We are looking for systems of more than modest functionality, yet where performance is an issue. The whole category of system software — operating systems, window systems, database systems, RPC mechanisms etc. — is a natural place to look. The abstractions have been well-honed over the years, there is tremendous understanding of the different kinds of implementation strategies that can be useful and, because these systems underlie everything else we build, the potential payoff of increased understanding of their nature is large.

It turns out that not only does work in these areas appear to support the basic argument for open implementations, but in fact there appears to be a lot of work already going on that is driving in similar directions.

Programming Languages

A number of programming language projects have discovered that attempting to give their users a black-box abstraction with a single fixed implementation does not work. In some sense, compiler pragmas were the first example of this — they can be thought of as open implementations with a "declarative" meta-level interface.

In Hermes [Hermes book], several of the built-in data structures come with a small collection of different implementations. This, like pragmas, is a step in the direction of open implementations — several implementations is after all more than one, and letting the user choose is a step in the direction of openness. But, it does not completely solve the problem because there is no reason to believe that some prospective users will not want an implementation that is different from any of the ones provided. The designers of Hermes are aware of this limitation, it is just that their concern for robustness (safety in particular) has so far prevented them from adopting the more powerful reflective or metaobject protocol techniques [Yemeni, private conversation]. One possibility might be to add an internal metaobject protocol, which the designers could use to quickly provide clients with newly requested implementations, but which would not be documented to normal users.

As discussed by Rodriguez [Rod92], the same sort of situation can be seen in languages for parallel programming. A key problem in this domain is that a compiler that attempts to automatically choose program's parallelization is often unable to do so optimally. Having recognized this problem, this community has developed architectures that allow the programmer to step in, in various ways, and direct the parallelization [Ber90, CiCL88, Hoa85, LR91, Luc87, YiC90]. These systems bear varying degrees of resemblance to explicit meta-level architectures, with one key difference being that

they have not (yet) adopted the use of object-oriented techniques to organize the meta-level.

At least one language has gone farther, to have what is clearly a metaobject protocol, the only difference being that they don't use the terminology we do. Joshua is a rule-based inference system developed at Symbolics [RSC87]. Because Joshua is such a high-level language, its default implementation can perform quite poorly on some examples. By allowing the client to step in and customize the inference mechanism to better suit the particular example, they sometimes get substantial performance improvements [Shrobe, private conversation].

Operating Systems

The operating system community long ago began to push up against the boundaries of the traditional black-box abstraction framework. Very early on, virtual memory systems provided limited meta-level interfaces that allowed clients to influence what page-replacement strategy was used (i.e. the Unix `madvise` facility). More recently, there has been a move, starting with systems like the Mach external pager, from the declarative approach to an approach more like that of metaobject protocols. Specifically, they are using object-oriented and imperative techniques to organize the meta-level.

Using this more powerful imperative approach, there has been similar work opening up thread packages and load-balancing mechanisms [ALL89]. In fact, people associated with this work have, more recently, been explicitly questioning the validity of the traditional closed-implementation notion of system software in many of the same ways discussed in this paper.[Anderson, talk at PARC] (Within the reflection community, there is of course the Muse work at Sony, which has been explicitly addressing these issues for some time [YTT89].)

In the operating system community, where there is a great deal of emphasis on reliability, the architectures have been interestingly different than in the metaobject protocol community. They have done a much better job of achieving robustness. The various efforts at reducing the size of the kernel are largely driven by a desire to make as much of the traditional operating system functionality user-replaceable. On the other hand, even though there is no apparent tradeoff between robustness and incrementality, they have done much less well at providing incrementality.

Other Systems

Looking at other kinds of systems software turns up similar kinds of work, although perhaps not as aggressively open as in the operating system community. There are interesting things to be said about databases, RPC mechanisms and document processing systems. In fact, the spreadsheet example presented in this paper was drawn from work at PARC which explicitly addressed the applicability of metaobject protocol ideas to the window system domain [Rao90, Rao91].

Future Work

Changing something as fundamental as our underlying conception of abstraction is not going to be a small task. All of our current design principles, conventions, tools, techniques, documentation principles, programming languages and more rest on the more fundamental notion of abstraction. This section provides a short sampling of what might need to be done, ranging from the relatively straightforward — assuring ourselves that the need for open implementations and a corresponding revision of our abstraction framework is in fact genuine — to the more far reaching — working out the ramifications of this revision, and what it will take to get it to work.

Much of what needs to be done involves looking at basic concepts in software engineering practice, to see how they depend on the old model of abstraction and how they might need to be revised. This includes issues like portability, software building blocks and top-down programming.

Complexity and Portability Revisited

A primary issue to be addressed has to do with what the consequences of the open implementation argument is for portability and complexity. One of the comments I often hear, when I talk about the metaobject protocol work, is that opening up implementations in this way will cause client code to be more complex and create portability problems. The goal is of course very much the opposite: to make code simpler and improve portability. But, these ideas makes people nervous; it is important that the meta-level architectures community be able to address their concerns carefully.

The criticism from skeptics is: (i) You are allowing the client to muck with implementation issues that used to be hidden. (ii) This will result in code that is more complex, and wedded to features specific to the implementation. (iii) This will make the code more difficult to work with and less portable.

The counterargument is: (i) Clients already are aware of the implementation issues, it is just that we have been trying to pretend that wasn't the case. That is the whole thrust of the first part of the paper. (ii) We believe that client code will be simpler, because it will be able to reuse more of the underlying functionality. There won't be hematomas and other complexities that currently result from performance problems in the library functionality. It is also important to understand that the meta-level interface is not implementation-specific. It applies to all implementations of the system. What is implementation-specific is the default implementation. So, the meta-program, since it is a customization of the default implementation, may end up depending on properties of the implementation for which it is written but: (a) programs already are implementation-specific; (b) in the new framework this dependence will be more explicit since it will be isolated to the meta-program; and (c) if there is less code to work with it will be easier to work with no matter what.

Higher-Level Building Blocks

The concept of open implementations has significant ramifications on our concepts of what kinds of building blocks it might be possible to work with in the future. Learning how to make clean, powerful open implementations should result in being able to build and work with higher-level building blocks, which should in turn result in simpler application programs. This expectation is based on the belief that what has kept us from being able to successfully develop very high level libraries has been our inability to provide (closed) implementations that pleased enough users.

The programming language domain is perhaps the place where it is most clear that a large part of what has kept us at a low-level is the closed implementation framework. High-level languages have enjoyed limited success in large part due to performance problems. We haven't been able to get good enough performance out of higher-level languages because we haven't been able to write compilers that are "smart enough" to satisfy all the users. But, the open implementation idea fundamentally acknowledges that if a language is more than modestly high-level, it simply isn't possible to build a closed compiler that is smart enough. We must instead open the compiler up so that the programmer, who knows a great deal about how they want their program to be compiled, can step in and help.

This restraining force on high-level languages is particularly evident in the earlier quote from Wirth. Essentially, his argument is

that since it isn't possible to properly implement high-level functionality (using a closed implementation), the language should be restricted to providing only low-level functionality. The question now is whether open implementations and the dual interface abstraction framework make it possible to make truly high-level languages with good performance. Experiments need to be done with a variety of such languages.

Top Down Programming vs. Reuse

In the previously mentioned paper by Shaw & Wulf they make the claim that top-down programming is fundamentally at odds with reusable code libraries and even the notion of system software. Their argument, as I understand it, is that a reusable library essentially blocks, at the abstraction boundary, the downward flow of design decisions, preventing those decisions from leaking into the library's implementation as we would like.

Their argument is essentially compatible with the one presented in this paper. From the dual interface abstraction point of view, the conflict is not between top-down programming and reusable code; it is between top down programming and *closed implementations* of reusable code. This leads to another way of thinking about open implementations, complementary to the dual interface model. The idea is that reusable code should be like a sponge: It provides basic functionality (the base-level interface), basic structure (the default implementation) but also allows the user to "pour in" important customizations from above to "firm it up."

Work needs to be done to go back and look at top-down programming and the conflict Shaw & Wulf mention to see how it informs the open implementation and dual interface abstraction frameworks.

Multiple Open Layers

This view of top-down programming makes it clear that opening an implementation only to the client immediately above is not enough. We need to do better than that; all layers need to be open to all layers above them. So, for example, when an application is written on top of a high-level language, which itself sits on top of a virtual memory system, the application code needs to be able to control not just how the language uses the memory it is allocated, but also how that virtual memory system allocates that memory.

Work needs to be done to develop this ability to push down, through multiple levels of abstraction this way.

Open Behavior

The discussion in this paper begins to provide an explanation of part of the problem metaobject protocols are solving — specifically, the need for open implementations. But a clear lesson from the metaobject protocol work is that users can also take productive advantage of being able to customize the *semantics* (or behavior) of systems they are building on top of.

Work needs to be done to integrate the need for open behavior, and the way that meta-level architectures provide it, into the argument presented in this paper and into any new abstraction framework that is developed.

Mastering Locality

The dual interface framework is similar to the way in which one might expect the conversation between the human provider and client of a system to talk. Much of the time they would just talk about the functionality that would be provided. At other times they would "go meta" and talk about how the functionality was going to be used and crucial performance issues.

And it is by making this analogy with the discussion between humans that we can get some insight into the problems that we will face in really trying to get this to work: very often, the concepts that are most natural to use at the meta-level cross-cut those provided at the base-level. What it seems we want to be able to do is to allow the user to use natural base-level concepts *and* natural meta-level concepts — as if they were the x and y axes of a plane — to get at just what it is in the implementation they want to affect. The problem is that the "points" in the plane spanned by these two axes are not necessarily easy to localize in an implementation.

Take, as an example, the user of a Lisp-like language who wants to control the tagging strategy for certain objects within a certain part of their program. It's quite natural for them to say something like: "Use immediate tagging for fixnums and positions, tag rectangles and lines in the pointer, and tag everything else in the actual object representations." But, it would be surprising to find an existing compiler in which making this change was easy, much less one that could be persuaded to have just part of a program work this way. (Getting such a compiler architecture is the thrust of the work reported in [LKRR92].)

We are, in essence, trying to find a way to provide two effective[8] views of a system through cross-cutting "localities." Getting this to work, in the general case, appears to be quite difficult; aside from crystallizing it as a problem, there isn't much to say about it at this time.

One strategy — the one that has been prevalent in existing meta-level architectures — is to make the problem easier by delaying the implementation of strategy selection until run-time or thereabouts. So, for example, the existing metaobject protocols address only those issues which do not need to be handled in a compile-time fashion. The various systems that address distribution, concurrency and real-time [other papers in this proceedings] are also addressing problems which are amenable to architectures with runtime dispatch.

An important point is that this problem, of having to handle two cross-cutting localities, isn't due to the dual-interface framework. It is a fundamental problem, it has always been there and it will always be there. The structure of complex systems is such that it is natural for people to make this jump from one locality to another, and we have to find a way to support that. All the dual-interface framework does is: (i) make it more clear that this problem needs to be solved, and (ii), give one particular organization to the relation between the two different localities. Of course, looking at the problem this way makes it clear that we may well want more than two, cross-cutting, effective interfaces to a system — the *dual* interface framework may quickly become the multi-interface framework.

Summary

It runs deep in our field that we consider ourselves to be based on mathematics. This leads us to try and take many of our basic notions from mathematics. The fact that Abelson and Sussman would quote Weyl the way they do is evidence of this.

But, while this appeal to mathematics for conceptual foundations may be attractive, it is, at least in the case of abstraction, risky. There is a deep difference between what we do and what mathematicians do. The "abstractions" we manipulate are not, in point of fact, abstract. They are backed by real pieces of code, running on real machines, consuming real energy and taking up real space. To attempt to completely ignore the underlying implementation is like

[8] *Effective* means essentially the same thing that "causally connected" did in Smith's earlier work.

trying to completely ignore the laws of physics; it may be tempting but it won't get us very far.

Instead, what is possible is to temporarily *set aside* concern for some (or even all) of the laws of physics. This is what the dual interface model does: In the base-level interface we set physics aside, and focus on what behavior we want to build; in the meta-level interface we respect physics by making sure that the underlying implementation efficiently supports what we are doing. Because the two are separate, we can work with one without the other, in accordance with the primary purpose of abstraction, which is to give us a handle on complexity. But, because the two are coupled, we have an effective handle on the underlying implementation when we need it. I like to call this kind of abstraction, in which we sometimes elide, but never ignore the underlying implementation "physically correct computing."

This is also like what the mechanical engineers call modeling, where they take multiple independent models of a system, each of which highlights certain properties and sets others aside. Of course a mechanical engineer's models aren't effective, and we would like ours to be — that is a fundamental difference in what we do and is why we can't borrow directly from them. But, it is the case that we are engineers not mathematicians. We would do better to look to other engineering disciplines, and not solely to mathematics, for our principles of abstraction.

This is, I think, the real contribution of the argument in this paper: Because we are engineers, not mathematicians, we must respect the laws of physics — we cannot hope to completely ignore the underlying implementation. The particular details of the dual interface model, the notion that two interfaces are enough, the role of object-oriented programming, the notion of meta; all of these are inherently approximate. What will remain, in the long term, is the intuition of physically correct computing and the requirement that we build open implementations.

Acknowledgments

I would like to thank Hal Abelson, J. Michael Ashley, Alan Bawden, Danny Bobrow, John Seely Brown, Jim des Rivières, Mike Dixon, John Lamping, Ramana Rao, Jonathan Rees, Luis Rodriguez, Erik Ruf, Brian Cantwell Smith, Marvin Theimer and Brent Welch for countless hours of discussion working out the ideas in this paper.

For their comments and feedback on earlier drafts of this paper itself, I would like to thank J. Michael Ashley, Danny Bobrow, Jim des Rivières, Mike Dixon, John Lamping and Luis Rodriguez.

References

[ALL89] T. Anderson, E. Lazowska, and H. Levy. The performance implications of thread management alternatives for shared memory multiprocessors. In *IEEE Transactions on Computers, 38(12)*, pages 1631–1644. IEEE, 1989.

[Ash92] J. Michael Ashley. Open compilers. To appear in forthcoming PARC Technical Report., August 1992.

[Ber90] Andrew Berlin. Partial evaluation applied to numerical computation. In *Lisp and Functional Programming Conference*, pages 139–150, 1990.

[BKK+86] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: Merging Lisp and object-oriented programming. In *OOPSLA '86 Conference Proceedings, Sigplan Notices* **21***(11)*. ACM, Nov 1986.

[CiCL88] Marina Chen, Young il Choo, and Jingke Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 2(2):171–207, October 1988.

[Coi87] Pierre Cointe. Metaclasses are first class: The ObjVlisp model. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Orlando, FL*, pages 156–167, 1987.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[IMWY91] Yuuji Ichisugi, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. An object-oriented concurrent reflective architecture for distributed computing environment. In *8th Conference Proceedings, Japan Society for Software Science and Technology*, September 1991. (in Japanese).

[IO91] Yutaka Ishikawa and Hideaki Okamura. A new reflective architecture: AL–1 approach. In *Proceedings of the OOPSLA Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, 1991.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, 1989.

[Kic92] Gregor Kiczales. Metaobject protocols — why we want them and what else they can do. In Andreas Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1992.

[LKRR92] John Lamping, Gregor Kiczales, Luis H. Rodriguez Jr., and Erik Ruf. An architecture for an open compiler. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. Also to appear in forthcoming PARC Technical Report.

[LR91] Monica S. Lam and Martin C. Rinard. Coarse-grain parallel programming in Jade. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, 1991.

[Luc87] John M. Lucassen. Types and effects: Towards the integration of functional and imperative programming. Technical Report MIT/LCS/TR-408, MIT, August 1987.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 147–155, 1987.

[MWY91] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *European Conference on Object Oriented Programming*, pages 231–250, 1991.

[Pad92] *The EuLisp Definition*, April 1992. Draft.

[Rao90] Ramana Rao. Implementational reflection in Silica. In *Informal Proceedings of ECOOP/OOPSLA '90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[Rao91] Ramana Rao. Implementational reflection in Silica. In Pierre America, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*, pages 251–267. Springer-Verlag, 1991.

[Rod91] Luis H. Rodriguez Jr. Coarse-grained parallelism using metaobject protocols. Master's thesis, Massachusetts Institute of Technology, 1991.

[Rod92] Luis H. Rodriguez Jr. Towards a better understanding of compile-time mops for parallelizing compilers. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. Also to appear in forthcoming PARC Technical Report.

[RSC87] Steve Rowley, Howard Shrobe, and Robert Cassels. Joshua: Uniform access to heterogeneous knowledge structures or Why Joshua is better than conniving or planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 48–58, 1987.

[Ste90] Guy L. Steele. *Common Lisp: The Language (second edition)*. Digital Press, 1990.

[SW80] Mary Shaw and Wm. A. Wulf. Towards relaxing assumptions in languages and their implementations. In *SIGPLAN Notices 15, 3*, pages 45–51, 1980.

[Vah92] Amin Vahdat. The design of a metaobject protocol controlling the behavior of a scheme interpreter. To appear in forthcoming PARC Technical Report., August 1992.

[Wir74] Niklaus Wirth. On the design of programming languages. In *Information Processing 74*, 1974.

[WY90] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In *Informal Proceedings of ECOOP/OOPSLA '90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990. (Extended Abstract of [WY91]).

[WY91] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands, May, 1990*, number 489 in Lecture Notes in Computer Science, pages 405–425. Springer Verlag, 1991.

[YiC90] J. Allen Yang and Young il Choo. Meta-crystal – a metalanguage for parallel-program optimization. Technical Report YALEU/DCS/TR-786, Yale University, April 1990.

[YTT89] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, July 1989. (also available as a technical report SCSL-TR-89-001, Sony Computer Science Laboratory Inc.).

Metaobject protocols: Why we want them and what else they can do

Gregor Kiczales, J.Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow

Published in A. Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 101 — 118. The MIT Press, Cambridge, MA, 1993.

Metaobject Protocols
Why We Want Them
and
What Else They Can Do

Appears in
Object Oriented Programming: The CLOS Perspective

Gregor Kiczales, J. Michael Ashley, Luis Rodriguez,
Amin Vahdat and Daniel G. Bobrow

Originally conceived as a neat idea that could help solve problems in the design and implementation of CLOS, the metaobject protocol framework now appears to have applicability to a wide range of problems that come up in high-level languages. This chapter sketches this wider potential, by drawing an analogy to ordinary language design, by presenting some early design principles, and by presenting an overview of three new metaobject protcols we have designed that, respectively, control the semantics of Scheme, the compilation of Scheme, and the static parallelization of Scheme programs.

Introduction

The CLOS Metaobject Protocol (MOP) was motivated by the tension between, what at the time, seemed like two conflicting desires. The first was to have a relatively small but powerful language for doing object-oriented programming in Lisp. The second was to satisfy what seemed to be a large number of user demands, including: compatibility with previous languages, performance comparable to (or better than) previous implementations and extensibility to allow further experimentation with object-oriented concepts (see Chapter 2 for examples of directions in which object-oriented techniques might be pushed). The goal in developing the MOP was to allow a simple language to be extensible enough that all these demands could be met.[1]

Traditionally, languages have been designed to be viewed as black box abstractions; end programmers have no control over the semantics or implementatation of these abstractions. The CLOS

[1]Some might argue that the base CLOS language is not so simple. What is more interesting is to consider how much of what is in that language could be dropped given the current or an improved metaobject protocol, now that we have a better handle on that technology. Among those features that would be prime candidates for elision would be: method combination, the `:argument-precedence-order` option, methods on individuals (`eql` specializers) and class variables. More aggressive proposals might drop aspects of the initialization protocol like slot-filling initargs and default initargs. Even more aggressive proposals might drop multiple inheritance and multi-methods.

MOP on the other hand, "opens up" the CLOS abstraction, and its implementation to the programmer. The programmer can, for example, adjust aspects of the implementation strategy such as instance representation, or aspects of the language semantics such as multiple inheritance behavior. The design of the CLOS MOP is such that this opening up does not expose the programmer to arbitrary details of the implementation, nor does it tie the implementor's hand unecessarily — only the *essential structure* of the implementation is exposed.

In more recent work, we have pushed the metaobject protocol idea in new directions, and we now believe that idea is not limited to its specific incarnation in CLOS, or to object-oriented languages, or to languages with a "large runtime" or even to Lisp-like languages. Instead, we believe that providing an open implementation can be advantageous in a wide range of high-level languages and that metaobject protocol technology is a powerful tool for providing that power to the programmer.

The purpose of this chapter is to summarize what has been learned to date about MOP technology, and suggest directions it might be pursued in the near future. In the first two sections, we start with the CLOS MOP, first summarizing the motivation for it, and then giving a brief overview of how one might think about its development. We then present a general framework for thinking about MOP design, together with some early principles for design cleanliness in a MOP. Finally, we present an overview of three new MOPS we have designed for Scheme, showing that a MOP can be used to handle a range of issues other than those in the CLOS case.

Motivating Examples

In this section, we present two simple problems that arise in languages like CLOS without a MOP, as a way of summarizing the motivation for the CLOS MOP. We also generalize from these examples, as a way of suggesting what other kinds of problems an open language implementation might be used to solve. The first example has to do with performance. In it, the programmer has written a program which CLOS expresses quite well, but finds (perhaps to their surprise) that the underlying language implementation doesn't perform adequately. In the next section we return to this example to show how the CLOS MOP addresses this problem by opening up part of the implementation strategy to the programmer. The second example, is a case where the programmer can be well-served by being able to adjust some part of the language semantics to better suit their needs.

A Performance Problem

Consider the two CLOS class definitions shown in Figure 1. The class `position` might be part of a graphics application, where the instances are used to represent the position of the mouse as it moves. The class defines two slots `x` and `y`. In this case, the behavior of the program is such that there will be a very large number of instances, both slots will be used in every instance and access to those slots should be as fast as possible.

The second definition, `person`, might come from a knowledge representation system, where the instances are being used as frames to represent different individuals. The thousand slots defined in the class correspond to a thousand properties of a person that might be known. In this application, the behavior is such that although there will be a very large number of instances, in any given instance only a few slots will actually be used. Furthermore, access to these properties will rarely be in the inner loop of a computation.

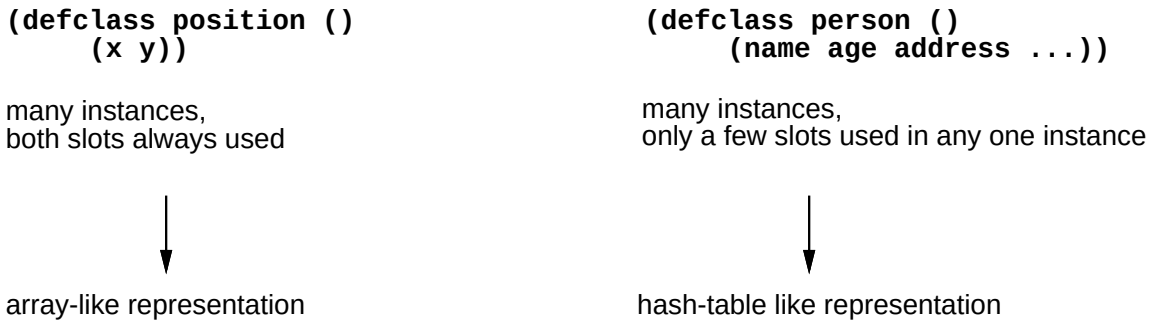


Figure 1: Two sample CLOS class definitions. Ideally, each class would like a different underlying instance implementation strategy. In a traditional (sans MOP) CLOS implementation, one or the other of these classes (probably `person`) will have bad performance.

Clearly, the ideal instance implementation strategy is different for the two classes. For `position`, an array-like strategy would be ideal; it provides compact storage of instances, and rapid access to the `x` and `y` slots. For `person`, a hash-table like strategy would be more appropriate, since it isn't worth allocating space for a slot until it is known that it will be used. This makes access slower, but it is a worthwhile tradeoff given a large number of instances.

The likely default implementation in most object-oriented languages is the array-like strategy. This serves the author of the `position` class quite well, but author of the `person` will not be so happy. Even though the CLOS language abstraction serves to express their program quite clearly, (supposedly) hidden properties of the implementation will impair performance.

This particular problem is not an isolated incident, it is an instance of a common, and much more general problem. As a programming language becomes higher and higher level, its implementation in terms of the underlying machine involves more and more tradeoffs, on the part of the implementor, about what cases to optimize at the expense of what other cases. That is, the fact that a typical object-oriented language implementation will do well for `position` and poorly for `person` is no accident — the designer of the implementation made (what we hope was a concious) decision to tune their implementation this way. What a properly designed open language implementation does is allow the end-programmer to go back, and "re-make" one of those tradeoffs to better suit their needs. A more complete discussion of this kind of performance problem, and the ways in which open language implementations based on MOPs can help address it is given in [1].

The Desire to Customize Behavior

As an example of how a programmer can derive benefit from being able to adjust the language's semantics, consider the case of a programmer who is porting a large body of Flavors [2] or Loops [3] code to CLOS. In most respects, Flavors, Loops and CLOS are sufficiently similar that the task of porting the code is (relatively) straightforward. Each has classes, instances with slots (or instance variables), methods, generic functions (or messages), and all the other basics of object-oriented behavior. It appears that a largely syntactic transformation can be used.

But there is one critical difference between the languages: while they both support multiple

inheritance, they use different rules to order the priority of superclasses. This means that a simple mapping of Flavors programs into CLOS can fail because of asymmetries in the inheritance of methods and slots. Given a traditional CLOS implementation, the programmer would have to either rewrite the code appropriately, or (perish the thought) implement Flavors for themselves from scratch.

This too is an instance of a more general and more common problem. Again, as language become higher and higher level, and their expressive power becomes more and more focused, the ability to cleanly integrate something outside of the language's scope becomes more and more limited. An open language implementation that provides control over the language permits the programmer to shift the language focus somewhat, so that it becomes a better vehicle for expressing what they have in mind.

This example also reflects the ability of MOP-based open language implementations to help keep languges smaller and simpler. That is, by having a metaobject protocol, the base language need not provide direct support for functionality which only some users will want—that can be left to the user to provide for themselves using the metaobject protocol.

Simple Metaobject Protocols

The first example above was brought to our attention by users who complained about the performance of their KR programs. To fix this, we needed to find a way to give them control over the part of the implementation that decides the instance representation strategy. To make that strategy replaceable, we need to put all the parts of the runtime that are based on it under the control of generic functions: the code that allocates instances—which needs to know how much space to allocate; and the code that accesses slots—which needs to know where to go to get or put a slot value.

Three generic functions in the protocol suffice: `allocate-instance`, `get-value` and `set-value`.[2] We require that the runtime, whenever it needs to create an instance or access a slot, do so by calling these generic functions. The simple metaobject protocol now looks like:

Run Time Action	Implemented By Call To
instance allocation	(`allocate-instance` *class*)
slot read or write	(`get-value` *class instance slot-name*)
	(`set-value` *class instance slot-name new-value*)

Note that these generic functions all receive the metaobject—the class in question—as their first argument. This is the case even for `get-value` and `set-value`, which could conceivably determine it from the object. They must receive it as an argument to make it possible to define methods specialized to the class metaobject class.

Given this protocol, the user can write meta-code—an extension to the CLOS implementation that provides a new kind of class—to support instances with a hash-table representation. As shown

[2] In this paper, where we are trying to briefly summarize the MOP approach, we have allowed ourselves some leeway from the details of the real CLOS MOP. To avoid confusion, we have used different names for protocol presented here which differs from the real MOP.

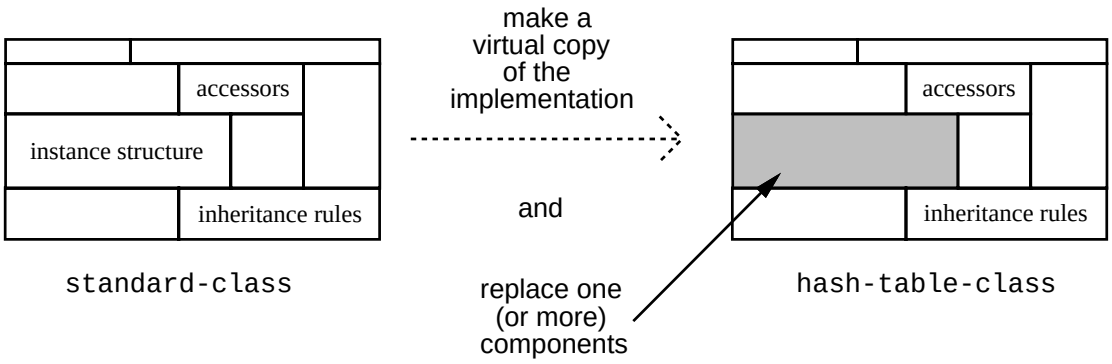


Figure 2: The standard implementation of classes has an internal structure, defined by the metaobject protocol, consisting of a number of components, including the instance representation strategy. Defining a new kind of class (i.e. `hash-table-class`) is a two-step operation: (i) Defining the subclass is like making a virtual copy of the standard implementation. (ii) Defining specialized methods (i.e. on `allocate-instance`, `get-value`, and `set-value`) is like replacing an internal component in that copy.

in Figure 2, the use of object-oriented techniques in the MOP makes this convenient. The code looks something like:

```
(defclass hash-table-class (standard-class) ())

(defmethod allocate-instance ((c hash-table-class))
  ...allocate a small hash table to store the slots...)

(defmethod get-value ((c hash-table-class) instance slot-name)
  ...get the slot value out of the hash table...)

(defmethod set-value ((c hash-table-class) instance slot-name new-value)
  ...store the slot value in the hash table...)
```

Then, in their base program, programmers can request that the metaobject for specific classes they define be instances of `hash-table-class` rather than `standard-class`. This is done by marking the definition of those classes using the `:metaclass` option.

```
(defclass person ()
     (name age address...)
  (:metaclass hash-table-class))
```

Recovering Performance

While the approach outlined above does provide the programmer a great deal of power, the fact that it introduces meta-level generic function calls into basic operations of the language runtime means it incurs a significant performance penalty. In the CLOS case, it is clear that we can't require the implementation of slot access, which should inherently be a two memory read operation, to require a generic function call at the meta-level.[3] Our current approach to this problem is to incrementally redesign the protocol so that we "pull the meta-level generic function dispatches out of the critical runtime paths," although we hope to explore the use of automatic techniques, such as partial evaluation, to help with this kind of problem.

Again turning to the CLOS MOP, the protocol for generic function invocation and method dispatch provides an excellent example of how this can be done. According to the semantics of CLOS, when a generic function is called, the following actions are performed: (i) The class of each of the required arguments is determined. (ii) Using those classes, the sorted set of applicable methods is determined. (iii) The applicable methods are combined into an effective method. (iv) That effective method is run, receiving as arguments all the original arguments to the generic function.

Under a simple protocol in the previous style, we might introduce generic functions for the second and third of these steps. The nature of this protocol can be seen in the following piece of code, which would be the runtime code required to implement generic invocation under this protocol. Note that, for simplicity, we are ignoring optional, keyword and rest arguments, and also ignoring all error checking. (Uppercase is used to mark calls to generic functions in the metaobject protocol.)

```
(defun apply-generic-function (gf args)
  (let* ((classes (mapcar #'class-of args))
         (methods (COMPUTE-METHODS gf classes))
         (effective (COMPUTE-EFFECTIVE-METHOD gf methods))
         (compiled (compile-effective-method effective)))
    (call-effective-method compiled args)))
```

Again, as with slot access, this protocol is powerful, but the requirement that meta-level generic functions be called as part of the invocation of each base level generic function means that performance will be unacceptable.

The optimization to the protocol is based on two observations: First, by far and away most of the work is in the generic functions `compute-methods` and `compute-effective-method`. Second, that by placing only modest restrictions on these generic functions, we can allow the implementation to cache their results. The restriction is simply that any method on these generic functions be *functional* in nature; that is, that given the same arguments it must return the same results. (The real MOP uses a somewhat more elaborate set of rules, but the basic principle is the same.)

The essence of the resulting protocol, together with its implementation, can be seen in the following pseudo-code for the part of the runtime that applies a generic function to arguments. This code shows that in the common path through the runtime—the cache hit—there are no meta-level generic function calls. This means that in steady state, no performance penalty is incurred for

[3] Because of the particular rules for slot inheritance in CLOS, a typical implementation, which supports incremental development, cannot get down to just a single memory read. See Chapters 13 and 14 for a discussion of these implementation issues.

having the MOP [4]. Nevertheless, the protocol does allow the programmer to customize method lookup and combination rules, and thereby achieve a wide variety of alternate language behavior.

```
(defun apply-generic-function (gf args)
  (let* ((classes (mapcar #'class-of args))
         (cached (cache-lookup gf (class-of first))))
    (if cached
        (call-effective-method cached args)
        (let* ((methods (COMPUTE-METHODS gf classes))
               (effective (COMPUTE-EFFECTIVE-METHOD gf methods))
               (compiled (compile-effective-method effective)))
          (fill-cache gf classes compiled)
          (call-effective-method compiled args)))))
```

Optimization of procedural protocols, such as `get-value` and `set-value` is more complex but is based on the same general approach. As with method dispatch, some of the protocol is "pulled back" to earlier times in the run-time image, like cache-filling time. Other aspects of the protocol are actually run as part of compiling method bodies. This is covered in greater detail in [5]. Pulling the protocol back as far as compile-time also shows up in the Scheme compiler MOPs discussed later.

Methodology

One way of thinking about MOP design is by analogy to programming language design. As shown in the left half of Figure 3, a good language designer works by first having in mind — or, better yet, "down on paper" — examples of the kinds of programs they want their users to be able to write. The language then follows from those examples as the designer works to be able to express them cleanly. Of course the process is far more iterative and ad-hoc, but the basic point is that language designers are, in effect, working with two different levels of design process at the same time: the level of designing particular programs in terms of a given language, and the level of designing the language to support the lower-level design processes.

MOP design is similar, with the addition of yet one more level of design process. Again, the fundamental driving force is a sense of the kinds of programs users should be able to write. But, in this case the designer is not thinking about a single language that might be used to express those programs, but rather a whole range of languages. So, in addition to the two previous levels — the programmer designing a program in terms of a single language and the language designer designing a single language to support that lower level — there is a third level, designing a MOP to support a range of language designers. This is depicted in the right half of Figure 3.

In this kind of design process, one important observation is that user complaints about previous languages and implementations take on tremendous value. Read carefully, they can provide important clues as to what flexibility programmers might want from the MOP. It is also important to note that, like language design, MOP design is inherently iterative; it is difficult to guess just what the users will want the first time out. This is certainly true of the CLOS MOP. The use of object-oriented techniques to organize a meta-level architecture first appeared in [6] and [7]. The former was a suggestive implementation for a MOP for Common Lisp. It took a prototype

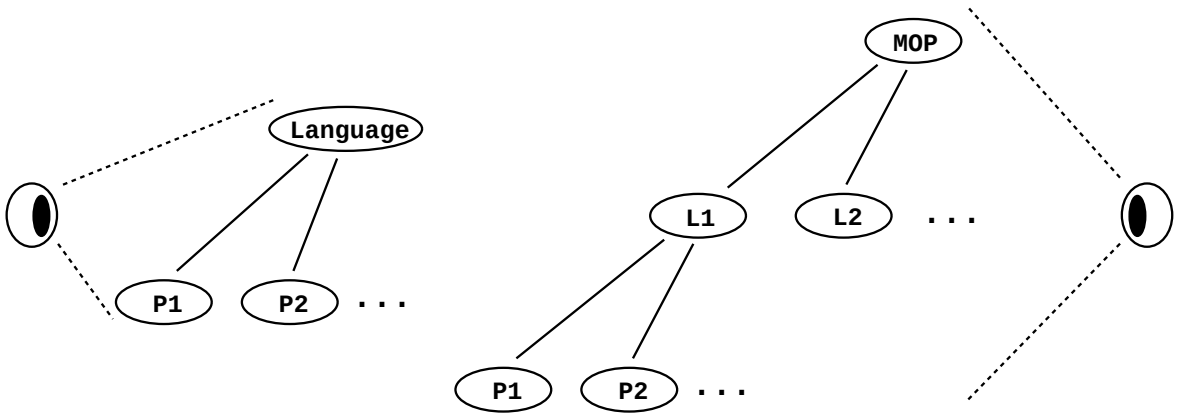


Figure 3: Language design and MOP design. On the left, the language designer envisages a range of programs the language should support elegantly, and designs a language accordingly. On the right, the MOP designer thinks about an even wider range of programs, and a *range* of languages, and designs the MOP accordingly.

implementation [4], and the feedback from a large community over five years to refine it. Moreover, the documentation of this kind of protocol brings up a number of subtle and unsolved issues in object-oriented specification [8].

In this sort of design process, the designer makes recourse to inuitive, aesthetic principles like "elegance" and "concision." That is, when desiging a programming language, one wants to make the resulting programs "simple" and "elegant." When designing a MOP, one wants to make not only the base programs simple and elegant, but also the meta-programs that define language extensions simple and elegant.[4]

0.1 Locality in MOP Design

This notion of elegance is inherently informal, and there is a great deal of work to be done before it can be reduced to standard practice, but there are some early design principles we have found it productive to think in terms of. There are based on a common aesthetic principle in computer science, the notion of localization of concerns and effect. Following are five coarse notions of locality in metaobject protocols. These are neither sharp nor orthogonal, but they do serve to talk about intuitive notions that come up in MOP design.

- *Feature Locality* – The metaobject protocol should provide access to individual features of the base language. So, in CLOS for example, a programmer should be able to customize slot inheritance without having to take over all aspects of inheritance. In a MOP for Scheme, the programmer should be able to get at binding discipline without dealing with order of evaluation.

[4]It is arguably more important for the base programs to be elegant than the meta programs, since there are far more of them, and we anticipate they will be written by less sophisticated programmers.

- *Textual Locality* – The programmer should be able to indicate, using convenient reference to their base program, what behavior they would like to be different. So for example, if the programmer is changing slot inheritance, they should be able to conveniently mark what classes in their program use the new inheritance.

- *Object Locality* – The programmer should be able to affect the implementation on a per-object basis. So, for example, in a MOP for Scheme, the programmer should be able to refer to individual closures, or all the closures resulting from a particular `lambda`.

- *Strategy Locality* – The programmer should be able to affect individual parts of the implementation strategy. So, for example, it should be possible to affect instance representation strategy without affecting method dispatch strategy.

- *Implementation Locality* – Extension of an implemenation ought to take code proportional to the size of the change being contemplated. A simple customization ought to be an *incremental* change. Programmers don't want to have to take total responsibility for the entire implementation; they don't want to write a whole new implementation from scratch. A reasonably good default implementation must be provided, and the programmer should be able to describe their extension as an incremental deviation from that default.

Applicability to Other Languages

While the metaobject protocol mechanism depends on having an object-oriented meta-language, there is no requirement that the base language be object-oriented. In this section we discuss the application of the approach to Scheme [9], using three metaobject protocols we have developed at PARC. There are two important structural differences between these MOPs and the CLOS MOP: First, the base language (Scheme) is not object-oriented, so the MOP is controlling issues of Lisp implementation and semantics, such as function calling convention and variable binding semantics, that are outside the purview of the CLOS MOP. Second, the meta-language used to implement the MOP is CLOS, not Scheme or an object-oriented extension to Scheme. That is, these systems are not meta-circular and they do not have towers.

Ploy

The first of these protocols, called Ploy, is a simple interpreter for Scheme. As an interpreter, Ploy's main use is as a testbed for working with programmer extensions of base language semantics. We are, in essence, using it to drive the iterative process of discovering *what* aspects of Scheme semantics to put under control of the protocol, before we attempt to develop a fully-featured, high-performance protocol.

In the Ploy MOP there are three distinct categories of metaobject: (i) Those that represent the program being interpreted — nodes in the program graph. (ii) Those that represent internal runtime data structures of the interpreter — environments and bindings. (iii) Those that represent Scheme values that are visible in the base level program — pairs, numbers, procedures and the like. The default metaobject class graph provided by Ploy is shown in Figure 4.

The interpreter works in the natural way: a documented set of generic functions evaluates the program, building environment structure as it goes. The most general of these generic functions

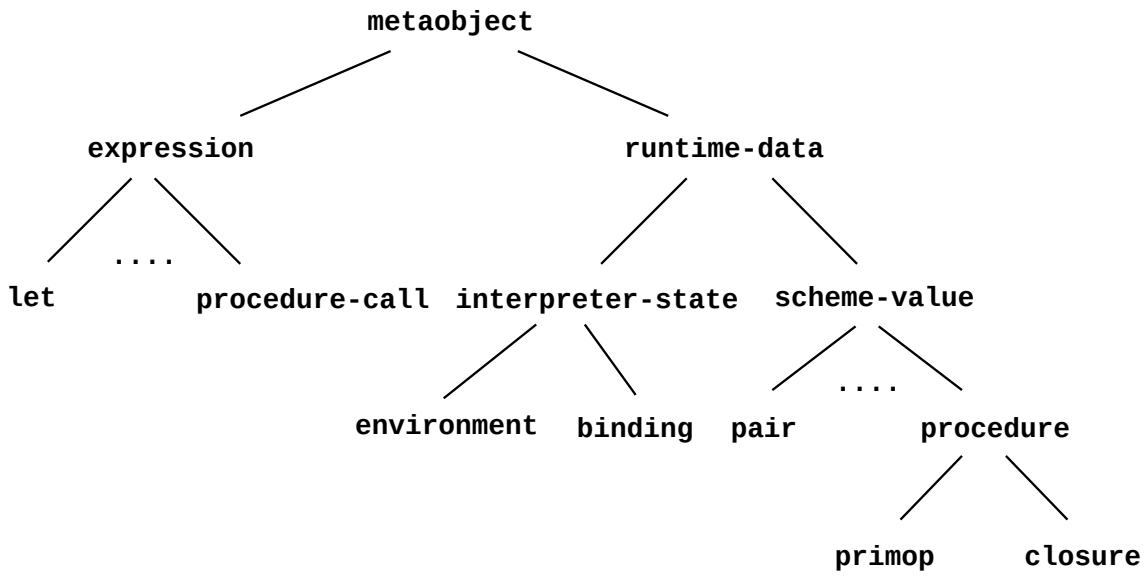


Figure 4: An overview of the default classes in Ploy, showing the three basic categories of metaobject: `program-element`, `interpreter-state`, and `scheme-value`.

is `eval`. In addition, to make user extension simpler, the protocol is also *layered*. That is, there are subsidiary generic functions that have more specific purposes and are thereby easier to define methods for. For example, there are special generic functions that manipulate binding metaobjects: creating them, and reading and writing their values. Figure 5 illustrates the layers in the Ploy protocol.

The current protocol is small, with only 9 generic functions. But, using it, we have been able to implement a number of traditional extensions to Scheme semantics, including: normal order evaluation, partial closures [10], control over order of evaluation, values that trace their path through the program, monitored variables and fully dynamic variables. The implementation of each of these extensions is simple and small, ranging from 8 to no more than 30 lines of code. The protocol localizes effect well enough that all these extensions can be loaded at the same time, and they can all be used in the same program.

In this paper, we demonstrate the use of the Ploy MOP using a toy example, monitored variables. More elaborate examples, including the implementation of partial closures and normal order evaluation, can be found in [11].

A monitored variable binding is one that prints out a message whenever its value is read or set. The Ploy MOP supports this kind of extension by requiring that access to the value of a binding go through the documented generic functions `read-value` and `write-value`. This protocol makes the implementation of monitored bindings quite straightforward. The core of the code is the definition of a new class of binding metaobject and appropriate methods on the reading and writing generic functions. (Note that because the meta-level code is in CLOS, not Scheme, it is easier to distinguish it from base-level code. This is one of the reasons we have chosen this approach.)

```
(defclass monitored-binding (binding) ())
```

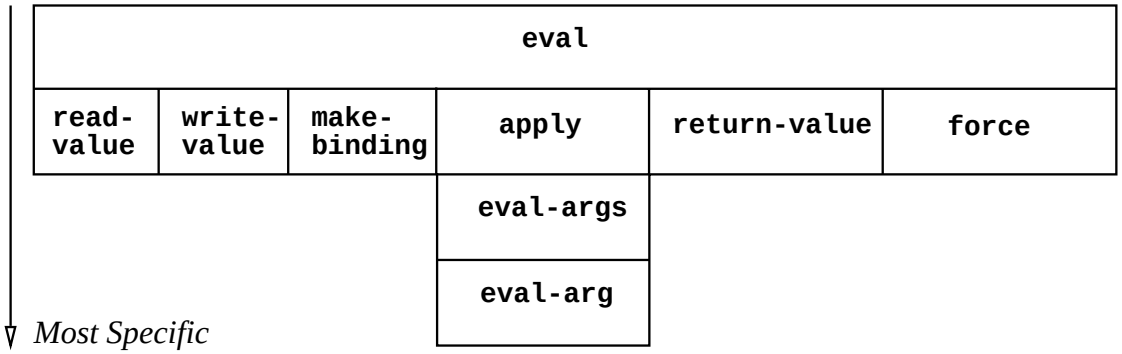


Figure 5: The calling structure of the Ploy protocol, indicating its layering.

```
(defmethod read-value :before ((b monitored-binding))
  (format t "Reading the value of ~S." b))

(defmethod write-value :before ((b monitored-binding) new)
  (format t "Setting the value of ~S to ~S." b new))
```

Now we must arrange for all the binding metaobjects corresponding to a particular variable to be of this new class. The Ploy MOP allows the user to control, using special syntax in the base-level program, the class of program element metaobjects; we must therefore arrange for a special class of variable program element metaobject to produce the special class of binding metaobject. To support this "chaining" from program element metaobject to interpreter state metaobject, the protocol provides the `binding-class` generic function. (This kind of chaining is common in our MOPs for Scheme, and more will be said about it in the next section.) The rest of the code for the extension looks like:

```
(defclass monitored-variable (variable) ())

(defmethod binding-class ((f monitored-variable))
  (find-class 'monitored-binding))
```

Then, using a special syntax with curly braces, the programmer requests that the program element metaobject for `x` be of class `monitored-variable` rather than the default `variable`.

```
(let (({monitored-variable}x 1)
      (y 2))
```

within the body of the let, access
to x prints out a message, but
access to y, or any variables free
in the let, are unaffected

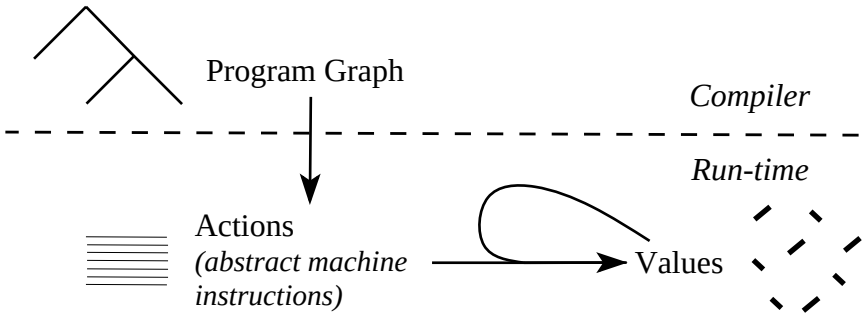


Figure 6: The Sartor architecture. The compiler protocol translates the source program graph into actions, or abstract machine instructions. The runtime protocol executes those actions, producing and operating on runtime data objects.

)

Sartor

Sartor is a compiler that compiles Scheme programs to abstract object code for a virtual machine. The architecture is illustrated in Figure 6. As indicated in the figure, an internal representation of the source program, a *program graph*, is compiled to a program in a high-level target language of *actions*. When this program is run, it creates and manipulates a variety of run-time data including both Scheme values such as pairs and closures, and internal values such as activation records.

Sartor is intended to be a testbed for experimenting with the use of MOP techniques to allow a user to control compilation strategies. As with Ploy, our initial goal has been to study *what* aspects of a compiler's behavior it makes sense to expose, rather than to handle actual code generation. In this sense, Sartor resembles the initial slot access protocol presented above. It introduces meta-level control at the cost of "runtime" method dispacth overhead. Once we have a better sense of what to protocolize, we will focus on restructuring the architecture to generate high-performance code.

Sartor in fact has two metaobject protocols: The compiler MOP operates on program graph metaobjects, producing action metaobjects. The compiler MOP makes decisions about order of evaluation and how the environment structure will be organized. This allows programmer extensions to control such issues as whether a free variable should be stored in a closure, or passed in by all callers to the closure.

The runtime MOP operates on actions, internal runtime data structures, and Scheme values. It controls the representation of runtime structures data, including such issues as how an activation record should be implemented, or how a closure's environment "tuple" should be represented in memory.

As an example of the protocol and how it might be customized, consider the implementation of activation records. The sequence of actions for performing a procedure call invokes the `construct-activation-record` generic function on the closure being called. By default, the record

constructed allocates storage for itself in the heap. But, the programmer can define a new class of activation record, that allocates itself on the stack. Or perhaps one that stores some of the arguments in registers.

The "metaobject chaining" issue mentioned in the previous section can be seen here quite clearly. In this case, the programmer is going to want a special kind of activation record to be created at runtime, but in order to do this, they are going to want to mark a particular `lambda` node (or perhaps a call site) in the source program. From that `lambda` node, there is a chain to the action that creates the closure at runtime, to the closure itself, and finally to the activation record. This kind of chaining structure is so common in our Scheme MOPs that we are now developing a new kind of object-oriented language specifically to support it.

A more detailed discussion of Sartor, together with examples of specializing both its compiler and runtime MOPs are presented in [12].

Anibus

We have also developed a MOP-based parallelizing Scheme compiler, called Anibus. Anibus is our first metaobject protocol to operate entirely at compile-time; there is no runtime object-oriented dispatch.

In Anibus, the programmer's model of the base language is Scheme, together with a set of parallelization directives, or marks, that provide high-level direction about what parallelization strategy the compiler should use. This general architecture is common in such compilers, as it provides the programmer with a clean separation between the algorithm their program implements and its parallelization.[13, 14, 15, 16]

But, a compiler with just this base model suffers from the same kind of performance problems discussed with CLOS above. Some programs, because of particular properties of their behavior, will require a parallelization strategy other than one of those provided by default. To address this, Anibus provides a metaobject protocol that allows the programer to define new kinds of marks.

The Anibus architecture is similar to Ploy and Sartor in that it operates on program graph metaobjects. The protocol consists of a small number of generic functions, each concerned with a separate aspect of parallelization, such as distribution of data, distribution of computation and synchronization. As the compilation proceeds, these generic functions gradually rewrite the program into a language with more primitive distribution and synchronization constructs.[5] As in Ploy and Sartor, a mark on the source program causes the corresponding program element metaobject to be of that class, and thereby affects what methods will be applicable to that metaobject.

Defining a new parallelization strategy is done simply by defining a new mark class, together with appropriate methods on the protocol generic functions. Very often, this can be done by subclassing one of the existing marks, and only one or two method definitions are required.

A complete description of Anibus, including examples of using it to define several alternative parallelization marks can be found in [17]. A discussion of how the functionality provided by Anibus differs from more traditional parallelizing compilers can be found in [18].

[5] The lower level language is also a variant of Scheme. It is then compiled by a compiler for the target architecture.

Conclusion

This chapter has shown that the idea underlying the CLOS Metaobject Protocol — to provide an open language implementation using object-oriented and reflective techniques to organize a meta-level architecture — is far more general than its incarnation in CLOS, or object-oriented programming, or even Lisp-like languages.

The original intuition behind this idea is that the very thing that makes high-level languages great — that they hide implementation details from the programmer — is also their greatest liability, since it is the inability of the programmer to be able to control those details that can result in poor performance.

One traditional approach to resolving this dilemma has been to hide in the programming language only those implementation details which can be automatically optimized — that is, to keep the language lower level than might otherwise be desirable. This philosophy, which is most clearly evident in languages designed for systems programming [19], is reflected in the following quote [20]:

> I found a large number of programs perform poorly because of the language's tendency to hide "what is going on" with the misguided intention of "not bothering the programmer with details." *N. Wirth, "On the Design of Programming Languages," Information Processing 74, pp. 386-393.*

Another approach has been to explicitly provide the user with declarative extra-lingual mechanisms to advise the implementation (i.e. compiler pragmas) about how some part of the program should be implemented.[6] As discussed in the section on Anibus, this approach can suffer from the problem that specific users may want to instruct the compiler in ways not supported by the supplied pragmas.

Metaobject protocols provide an alternative framework that opens the language implementation up to user "intervention." The major difference is that they are imperative in nature, and as a result are much more powerful. The metaobject protocol approach also distinguishes itself from previous approaches in that it allows the programmer to alter the semantics of the language. While this latter may seem controversial, we have found, in our work with the CLOS MOP, that properly used, programmers can derive tremendous benefit and program clarity from being able to customize the language semantics.

Acknowledgements

Work on the CLOS Metaobject Protocol was done jointly with Jim des Rivières. The development of more general ideas about the need for open language implementations and metaobject protocols has benefited tremendously from discussions with Hal Abelson, Mike Dixon, John Lamping and Brian Smith.

This work has also benefited from the contributions and feedback of the entire CLOS Metaobject Protocol user community. Their willingness to experiment with our rapidly changing prototype implementation (PCL) allowed us to develop the CLOS MOP and these ideas much more quickly than would otherwise have been possible.

[6] We call these extra-lingual because they do not in general affect the program semantics.

References

[1] Gregor Kiczales. Towards open implementations – a new model of abstraction in software engineering. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. Also to appear in forthcoming PARC Technical Report.

[2] D. Moon. Object-oriented programming with Flavors. In *OOPSLA '86 Conference Proceedings, Sigplan Notices* **21***(11)*. ACM, Nov 1986.

[3] Daniel G. Bobrow and Mark Stefik. The Loops manual. Technical report, Xerox PARC, 1983.

[4] Gregor J. Kiczales and Luis H. Rodriguez Jr. Efficient method dispatch in PCL. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 99–105, 1990.

[5] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[6] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: Merging Lisp and object-oriented programming. In *OOPSLA '86 Conference Proceedings, Sigplan Notices* **21***(11)*. ACM, Nov 1986.

[7] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 147–155, 1987.

[8] Gregor Kiczales and John Lamping. Issues in the design and documentation of class libraries. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 1992. To Appear.

[9] IEEE Std 1178-1990. *Ieee Standard for the Scheme Programming Language.* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[10] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. Technical report, Indiana University Computer Science Department, Aug 1992. Forthcoming Technical Report.

[11] Amin Vahdat. The design of a metaobject protocol controlling the behavior of a scheme interpreter. To appear in forthcoming PARC Technical Report., August 1992.

[12] J. Michael Ashley. Open compilers. To appear in forthcoming PARC Technical Report., August 1992.

[13] Rajive Bagrodia and Sharad Mathur. Efficient implementation of high-level parallel programs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 142–151, 1991.

[14] William Weihl, Eric Brewer, Adrian Colbrook, Chrysanthos Dellarocas, Wilson Hsieh, Anthony Joseph, Carl Waldspurger, and Paul Wang. Prelude: A system for portable parallel software. Technical Report MIT/LCS/TR-519, MIT, October 1991.

[15] J. Allen Yang and Young il Choo. Meta-crystal – a metalanguage for parallel-program optimization. Technical Report YALEU/DCS/TR-786, Yale University, April 1990.

[16] Monica S. Lam and Martin C. Rinard. Coarse-grain parallel programming in Jade. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, 1991.

[17] Luis H. Rodriguez Jr. Coarse-grained parallelism using metaobject protocols. Master's thesis, Massachusetts Institute of Technology, 1991.

[18] Luis H. Rodriguez Jr. Towards a better understanding of compile-time mops for parallelizing compilers. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. Also to appear in forthcoming PARC Technical Report.

[19] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, Englewoods Cliff, New Jersey, 1991.

[20] Niklaus Wirth. On the design of programming languages. In *Information Processing 74*, 1974.

User-Level Language Crafting
Introducing the CLOS Metaobject Protocol

Andreas Paepcke

3.1 Introduction

The idea of open and modular systems is becoming more and more popular in the areas of networking and operating systems. In the former, services like packet transfer may be implemented in different ways without affecting the rest of the system [1]. In operating systems, attempts are made to open functions such as memory paging up to change [2]. CLOS carries this idea into the realm of language design which has traditionally been almost as closed as database implementations.

There are many reasons why language implementations should be open. One important reason is the ever increasing complexity of software development. Its management requires correspondingly more sophisticated tools which must obtain detailed language-internal information, such as class structure or information about methods. Traditionally designed languages often require implementation-specific modifications to compilers or run-time environments which are non-portable because that information is otherwise not obtainable. As the cost of software development rises, such inefficiencies in the creation of integrated environments become less and less tolerable.

The basic idea of the CLOS design is to specify a model for the language implementation and to standardize it. The inner workings of the implementation thereby become manipulable in a controlled manner. This internal model is called the CLOS Metaobject Protocol (MOP)[1].

The goal of this chapter is to explain the basic idea, the important principles and some design issues behind this part of the CLOS language. We make the reader understand why the approach is important and how it works. The material should be sufficient to provide intuition for deciding when the use of the Metaobject Protocol would be appropriate for some given application and how to go about its design.

This chapter shows selected highlights and is not a replacement for an eventual study of the specification in part two of [3], although it should make its consumption easier. We have tried to avoid the complexity caused by a formal specification without sacrificing important information on the material we cover. Part one of [3] is a detailed explanation of the principles of protocol-based design, while this is an *introduction* to the CLOS MOP.

Section 3.2 explains what the Metaobject Protocol is about, what it is trying to do and why it is interesting. Most of this material is kept at an abstract level and does not require deep knowledge of CLOS particulars.

Sections 3.4 and 3.5 are much more concrete. They present selected details of the MOP using an example that is introduced in section 3.3. These sections do assume knowledge of CLOS as explained in [4, 5].

The main body of the chapter closes with pointers to related work and the conclusion. Appendices provide some material about the MOP which is useful for the deeply interested reader but which are too detailed to include in the text.

3.2 The Metaobject Protocol

Beyond trying to be a powerful language in general, CLOS has two additional, unusual goals:

- Allowing users and external programs to *inspect* the internals of CLOS environments.
- Allowing external programs to *extend* the CLOS language itself without modifying existing implementation code and without affecting other, existing programs.

The first of these goals is particularly relevant for the construction of browsers that aid in software development, such as class hierarchy layout displays, and for the implementation of other system analysis tools, such as debuggers. Inspecting the internals includes, for example, the ability to programmatically determine the class structure of a program — without scanning and parsing source code, or to find out which methods are specialized on some class.

The ability to extend or modify the language is necessary to enable experimentation and adjustments to CLOS behavior which may be required to satisfy new applications or system environments. This might include control over how slots are accessed or how instances are made. Enabling non-instrusive modifications can significantly increase return on the investment of designing and building a language because it can be made applicable to a wider range of consumers.

Let us take a first-level look at how these objectives are addressed in CLOS. The approach is at this level quite applicable to designs of other systems that share these goals.

3.2.1 Design Premise and Challenges

When we study the basics of CLOS internals with focus on their openness and flexibility, it is convenient to separate static from dynamic aspects. This partitioning roughly reflects the two goals of the CLOS internal design we listed above and provides a way of organizing the material in our mind.

The static part of the CLOS design may be called its *metalevel architecture*. It describes the components of the system, its structural and procedural building blocks and how they are put together. Examples of major building blocks are the manifestations of classes, slots or methods in the language's implementation.

The dynamic part is described in terms of *protocols* which prescribe the manipulations of the building blocks that must be performed to effect the behavior of the language at run-time. For each 'behavior pattern' of the language, one or more protocols specify how the building blocks must change and interact. Example: everything that is supposed to happen in a CLOS system when a new class is defined is governed by the class *initialization* and *finalization* protocols. They specify the language-internal building blocks and run-time activities that together effect the definition process.

Thus we distinguish between the language itself and a 'metalevel' where its concepts are described abstractly and then implemented. This metalevel world has collectively become known as the CLOS *Metaobject Protocol* and is the focus of this chapter.

Every reasonably designed system has the characteristics we described so far: an external specification of behavior and an internal, hopefully modular model and its implementation. The step CLOS is attempting to take beyond this is to *export* the internal model, to standardize it and to make it part of the final product itself. This final step is what gives this language the desired flexibility and which makes it go beyond many other systems.

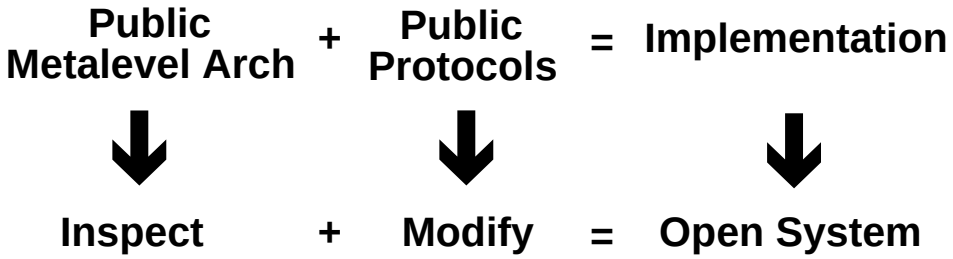


Figure 3.1: The Top-Level CLOS Design Premise

The means to modify CLOS have become **part of the language** and are therefore made as portable as the language itself.

Figure 3.1 tries to illustrate how the metalevel architecture representing statics, and the protocols representing dynamics together make up the CLOS design. The figure also suggests that the existence of these two explicit, standardized components enables us to inspect internals and to modify behavior, which in turn implies that CLOS is an open system — an unusual trait for a language.

All this sounds rather obvious and straight-forward. But designing such a system is difficult. The challenge begins when the proper break-down into building blocks must be decided. This break-down determines how cleanly the eventual implementations will be able to reflect the internal model. It can also determine how far modifications to one building block need to propagate through the system to other building blocks. These are, of course, crucial issues because the organized manipulation of the internal model are the way of modifying the implementation. Clean relationships between them are therefore important.

An even more difficult challenge than finding the proper break-down for the internal model is to find the level of detail to which protocols must be specified. An incorrect level not merely causes inconvenience, but it can lead to system failure. If too much detail is specified, implementations do not have enough room to introduce necessary optimizations. If too little is specified, it becomes unclear where and how modifications must be properly introduced to effect some desired change in behavior. This can lead to a loss in portability of the modifications.

When trying to find the proper balance for standardization questions like these, designers face the dilemma that sample applications are needed to find where the system's degrees of freedom should be placed. Obtaining a significant number of such applications, however, almost requires the a priori existence of the standard. Building an open system like this is therefore generally much more time consuming and frustrating than the construction of a more traditional design[2]. But the payoff is considerable.

3.2.2 Implementation of the Design Premise

We have so far spoken of 'building blocks' and 'behavior' in the abstract. What are these in the concrete case of CLOS? It is a very convenient characteristic of the 'CLOS-producing metalevel world' that it is itself written in CLOS. This unity of language is called *meta-circularity*. The language is itself a CLOS program which is manipulated through techniques of object-oriented programming. This is accomplished through appropriate bootstrapping facilities which do not need to concern us here. The ability to modify the language's implementation without leaving the realm of the language is called *reflection*.

Unity of language makes the life of the metalevel manipulator much easier. Instead of needing to learn a new configuration or implementation language, we can freely move between 'regular programming' and metalevel programming without having to switch languages and our way of thinking.

In particular, the metalevel architecture is defined and implemented as a CLOS class hierarchy. Instances of these classes implement elements of the CLOS object model at run-time. CLOS classes or methods, for instance, are themselves instances of classes at the metalevel. We will introduce these classes in section 3.4. Extensions to this *static* part of the CLOS implementation are made by subclassing the classes at the metalevel.

The *dynamics* of CLOS are captured in a set of generic functions and methods specialized on these classes. The protocols describe the main activities of these generic functions and explain which of them must be invoked to implement the behavior patterns of the language. Extensions and modifications of the dynamic part of CLOS are therefore usually implemented by defining new methods on existing system generic functions.

This uniformity of the metalevel and CLOS-level worlds does have the potential of causing confusion in that we must keep track of whether we are, for instance, talking about classes a regular programmer would define, or classes at the metalevel, which are pieces of the CLOS implementation.

The term *metaobject class* is used to denote a class at the metalevel. Instances of these classes are called *metaobjects*. A metalevel instance that implements a CLOS generic function or a CLOS programmer-level class is therefore a metaobject.

3.2.3 A More Detailed View

Let us pull together what we know so far about the CLOS design and its metalevel world and add some new pieces.

Figure 3.2 shows how we could view the system. A regular user of CLOS would be at the bottom of the figure 'looking up'. Regular programmers do not modify the language itself. They create metaobjects through definitional macros, such as the familiar `defclass`, `defmethod` or `defgeneric`. They unwittingly use these metaobjects by such activities as creating instances of classes and invoking generic functions.

Metalevel programmers perform the same activities, but they also handle metaobjects more consciously. In particular, they use mechanisms such as `find-class`, `find-method` or `symbol-function` which take a name and return an associated metaobject. `Find-class`, for example, takes the name of a programmer-level class and returns the metaobject that implements it.

Metalevel programmers also work with the static and dynamic parts of the language implementation by subclassing and by adding methods to system generic functions.

At the center of figure 3.2's upper portion we see the snapshot of a run-time collection of metaobjects which implement some running CLOS program. They are surrounded by the major design components which control them: the metaobject class hierarchy defining the static setup of the metalevel world. The exported, standardized system generic functions and methods which provide the implementation of the dynamic aspects and the protocol component which controls what the dynamic component does.

The next section explains some restrictions that are imposed on manipulations of the internal model.

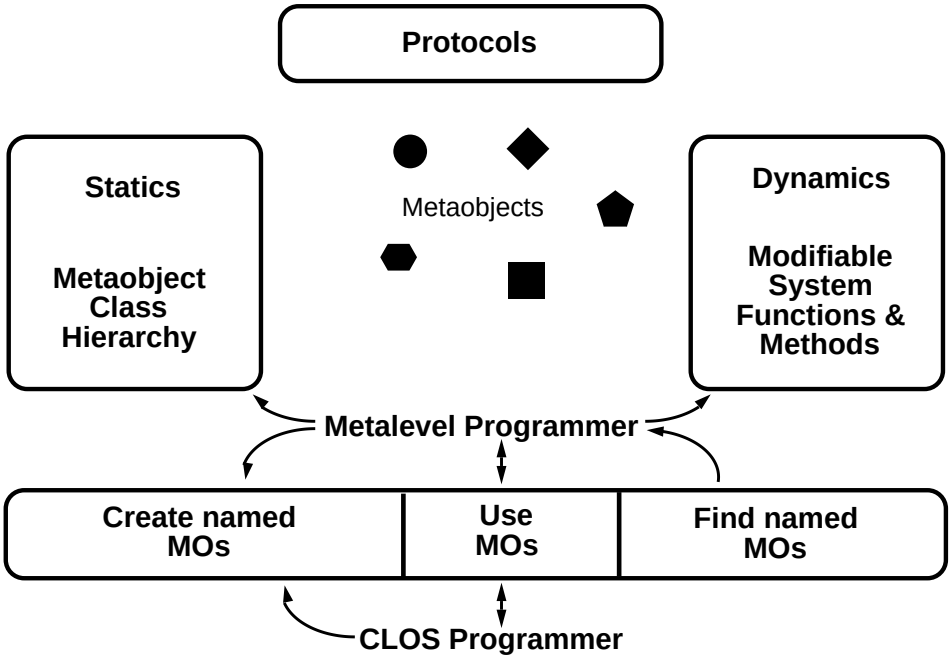


Figure 3.2: The Overall CLOS Design

3.2.4 Curbing Chaos

We have seen that the main tools of the metalevel programmer are subclassing and the definition of methods on exported system generic functions. Indiscriminate use of these tools can prevent a system from functioning properly. The problem lies in the fact that programmers build modules under the assumption that the language they work with is immutable. If the loading of one module changes the language, other modules can fail unless special care is taken.

The MOP does not include enforced safeguards against conflicts arising from metalevel manipulations. Instead, there are rules regarding these activities which are intended to ensure that extensions made at the metalevel are portable and do not destroy the system for other programs running in the same environment. One reason for such extreme openness is that radical modifications do have their place. One example has been the reduction of CLOS to a very small, fast, low-functionality delivery kernel after the completion of program development [6]. In general, however, programs will need to be portable, which means that they will need the ability to coexist with other, independently produced programs. This includes metalevel programs.

The rules regarding metalevel work all have the same purpose: To ensure that new behavior does not change *existing* system behavior that is relied upon by others. Appendix A contains a list of these rules.

Before we begin to introduce details of the Metaobject Protocol, we describe the skeleton of an application which we will use throughout the subsequent sections to illustrate how all the facilities can be put to use.

3.3 An Example Problem

As an example for the use of the Metaobject Protocol let us imagine that we want to add persistence to the objects in CLOS programs [7, 8].

We assume that objects may be either transient or persistent. The state of each persistent object is stored in a database and retrieved from there as needed. We make objects persistent by sending them the message `make-persistent`. This will cause the database to be prepared to receive the object's state and will then transfer the state there.

Objects may be cached, which means that their state is withdrawn from the database and stored in memory until it is explicitly returned to the database. Whether a persistent object is cached or not, it is always possible to send messages to it as if it were transient. There is to be no semantic difference between these object states, other than the persistence of values. If a slot of an uncached, persistent object is read, the slot value is retrieved from the database and returned as if it had been stored in memory. Slot updates are propagated to the database.

For reasons of efficiency and for some other technical reasons, it is desirable to allow individual slots to be transient. The value of a transient slot is not placed in the database but is always memory-resident, even if the object as a whole is made persistent. The programmer may declare individual slots to be transient when the class is being defined. In cases where some slot is provided by more than one superclass, we assert that transience is legal for the slot only if *all* superclasses have declared it to be transient. Otherwise it must be made persistent.

One tricky problem is caused by class redefinition, which CLOS makes easy to accomplish: we must create some appropriate schema in the underlying database which corresponds to the class hierarchy of the program that will generate the persistent instances. If this hierarchy changes, the schema will have to evolve as well. We will not cover how this can be accomplished in the database — that is a research issue in itself. We will merely point out how we can use the MOP to cause schema evolution to be initiated when necessary.

Given this problem description, how must we change the behavior of standard CLOS to accommodate a solution:

- We must be able to programmatically examine classes so that we can build appropriate schemas in the underlying database.
- The definition and redefinition of classes must be trapped to allow schema creation and evolution to be triggered.
- We need to manipulate the class inheritance.
- A new slot option must be introduced into the language to allow slots to be declared transient.
- Information about which slots are transient must be stored somewhere in the runtime system.
- Without the programmer being aware, internal information must be kept with each instance that is created. An important such piece of information is whether that particular instance is currently persistent or not.
- Additional information must be kept with each class. This might include information about how the database must be accessed or special caching policies for instances of that class.

- Slot access must be intercepted to implement faulting to the database.

Even a cursory glance at this list of requirements shows that these are significant modifications to any language and cannot be accomplished by working outside the language implementation. Our strategy will be to define a *class* metaobject class called `persistent-metalevel-class`. When a programmer defines a class whose instances are to have the potential of being persistent, she specifies that `persistent-metalevel-class` is to provide for that class' implementation.

We will define a programmer-level class `persistence-root-class` which provides some methods for persistent objects, such as `cache` and `make-persistent`. We will have `persistent-metalevel-class` take care of mixing that class into persistent user classes transparently.

Clearly, a full-scale persistent object system will need to do more than what we describe in this skeleton, but it turns out that this subset covers the language incisions that are necessary for such systems. It is therefore well suited to illustrate what we have to say about the details of the Metaobject Protocol.

In the following section we go into the details of the MOP's structural parts.

3.4 Metalevel Statics

We explained above that the structural part of the Metaobject Protocol reflects a breakdown of CLOS into basic concepts which is itself reflected in the metalevel class hierarchy. It is, of course, important to understand this hierarchy, as it is the key to making structural modifications and to accomplishing inspection of program internals.

The main building blocks are:

1. *Classes*
2. *Slots*
3. *Methods*
4. *Generic Functions*
5. *Method combination*

Each of these is represented by a class subtree at the metalevel whose terminals are the sources of the corresponding metaobjects.

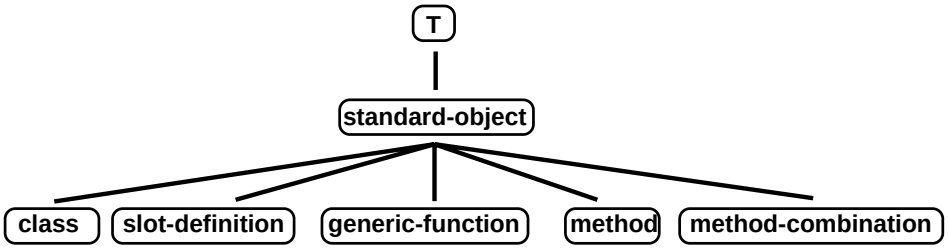


Figure 3.3: The Top-Level MOP Class Hierarchy

Figure 3.3 summarizes this.

In this section we will take several of these building blocks in turn and will explain their structural properties. Please note that we will not show the complete subtrees of a typical CLOS implementation. We try to extract the subclasses most likely to be of general interest to avoid confusion. It should not be necessary to understand more of the hierarchy.

Remember that the interface to the metalevel world provides us with powerful ways of finding out about the structural properties covered here. We can use `find-class <class-name-symbol>` to obtain instances of any of the *class* metaobjects we talk about. Using `describe` on those will reveal much useful information. Browsing the implementation in this way is indeed a very good way of getting acquainted with the system.

3.4.1 The *Class* Metaobject Class

The most frequently inspected and modified building block is the CLOS **class** since many important methods are defined on it and it contains a large amount of information useful for debugging and program maintenance. As a rule of thumb, if desired information is usually specified in a `defclass`, the resulting *class* metaobject is the place to find that information later on[3]. Standard CLOS comes with several *class* metaobject classes built in.

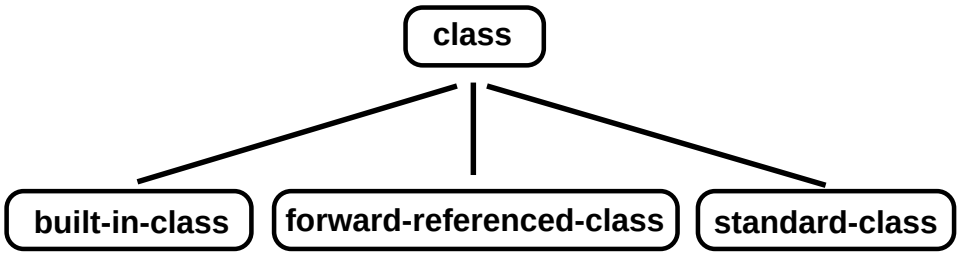


Figure 3.4: The *Class* Metaobject Class Subtree

Figure 3.4 shows some of these. The most important is `standard-class` since its instances are the metaobjects which by default implement the classes a programmer defines with the `defclass` macro. Most new metaclasses a user might want to write will be subclasses of `standard-class` and we concentrate on it here. But since most metalevel work tends to cause programmers to come across some of the others in passing, we mention their role briefly:

Instances of `built-in-class` implement classes that are not specified using `defclass` but are pre-constructed by CLOS implementations. Examples are classes that are made to correspond to standard CommonLisp types. `Built-in-class` metaobjects have various special properties, like the fact that they may not be redefined.

The `forward-referenced-class` is used when a programmer defines a class whose superclasses are not yet defined. In that case a metaobject of class `forward-referenced-class` is created to act as a 'place holder' until the superclass is defined later on.

The following information is kept in a `standard-class` metaobject. It is easy to see the correspondence between what a `defclass` specification contains and the information listed here. Indeed, the class metaobject is where most of the `defclass` entries end up. This information is available and we list the published reader function names for each of the items in parentheses.

As an example for the use of this information, assume the existence of a programmer-level class `train`. We could find its direct superclasses through:

```
(class-direct-superclasses (find-class 'train))
```

- The slots of the class are kept as a list of *slot* metaobjects. Reader `class-slots` returns all slots, including the inherited ones, `class-direct-slots` returns just the ones defined for this class explicitly.
- The super- and subclasses are stored as a list of *class* metaobjects (`class-direct-superclasses` and `class-direct-subclasses`).
- The class precedence list is recorded as a list of *class* metaobjects (`class-precedence-list`).
- The default initialization arguments for the class are kept. Reader `class-default-initargs` returns all initargs, including the ones inherited from superclasses while reader `class-direct-default-initargs` returns only the ones specified for the respective class directly.
- Information on whether the class has already been finalized is also available (This will be false if, for example, there were undefined superclasses at the time the *class* metaobject was created.) (`class-finalized-p`).

We can now introduce the first of the modifications our persistent object example requires: the storage of additional information in *class* metaobjects. We define a new metaclass:

```
(defclass persistent-metalevel-class (standard-class)
  ((checked-schema-congruence-p :initform NIL
                                :reader class-checked-schema-congruence-p)
   ))
```

It adds a new slot to *class* metaobjects which allows us to record whether we have checked that the structure of the class conforms with any schema we might have built earlier in the database to hold persistent objects of this class.

Now we can define our first persistent programmer-level class:

```
(defclass hypertext-node ()
  ((contents  :initform "" :accessor contents)
   (in-links  :initform NIL)
   (out-links :initform NIL))
  (:metaclass persistent-metalevel-class)
  )
```

This is a good time to make sure that easy-to-arise confusion between the metalevel and the regular CLOS level is avoided: at this point we have a programmer-level CLOS class called `hypertext-node` which contains the three slots `contents`, `in-links` and `out-links`. This class is all a regular CLOS programmer ever works with. If we now move into the metalevel world, we find out that this class is in reality a metaobject which is an instance of the *class* metaobject class called `persistent-metalevel-class`. Since that inherits from `standard-class`, it presumably has some slots we have no access to (the

reader functions listed earlier provide all the information we are supposed to have). But we have added the additional slot for the schema congruence check whose value is available to us. This slot is therefore part of the metaobject, **not** part of any future programmer-level instances of `hypertext-node`.

With this clarified, let us get a hold of the class metaobject and find out some details about it (system responses are indented):

```
(setf hypertext-class-metaobject (find-class 'hypertext-node))

(class-direct-slots hypertext-class-metaobject)
  (#<Standard-Slot-Definition CONTENTS>
   #<Standard-Slot-Definition IN-LINKS>
   #<Standard-Slot-Definition OUT-LINKS>)

(class-precedence-list hypertext-class-metaobject)
  (#<Persistent-Metalevel-Class HYPERTEXT-NODE>
   #<Standard-Class STANDARD-OBJECT>
   #<Standard-Class T>)

(class-checked-schema-congruence-p hypertext-class-metaobject)
  NIL
```

We can also begin to add some behavior to our new metaclass which allows us to build a database relation based on the slots of the class and to record in the database some information about the class itself. We assume that we have a database object `*database*`. This object may be used for calls to generic functions that manipulate a database consisting of tables. We assume further that the database contains a special table called "master-class-table" which we initialized earlier and in which class-related information is stored. Tables can be searched by key and we can add and delete rows:

```
(defmethod create-schema ((class persistent-metalevel-class))
  (create-table *database* (class-name class) (class-slots class)))

(defmethod store-class-structure ((class persistent-metalevel-class))
  (unless (find-entry *database* 'master-class-table (class-name class))
    (add-row *database*
             'master-class-table
             (class-name class)
             (class-slots class)
             (class-precedence-list class))))
```

3.4.2 The *Slot-definition* Metaobject Class

Slots in CLOS and other object-oriented languages are more than a physical place to store a value. Issues of typing, initialization and accessability must be remembered and managed. This is why the second major building block of the Metaobject Protocol is the `slot-definition` metaobject. We use the `class-slots` generic function on the *class* metaobject to get a hold of `slot-definition` metaobjects. Recall that this returns a list of the class' slots.

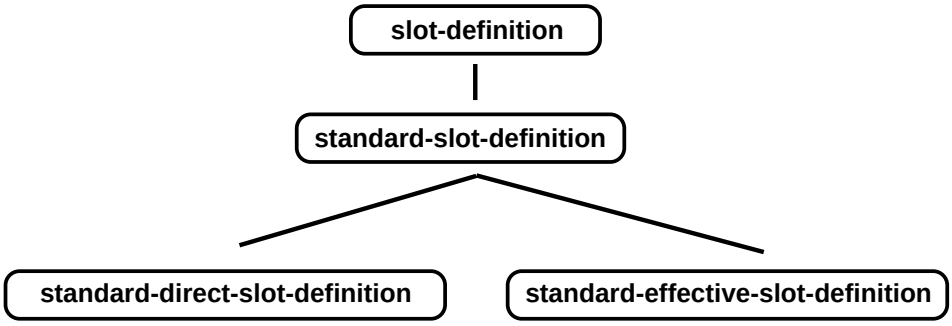


Figure 3.5: The *Slot-definition* Metaobject Class Subtree

Figure 3.5 shows part of the relevant metaobject class subtree. We see that there are two main branches: `standard-direct-slot-definition` and `standard-effective-slot-definition`. Instances of the first hold the 'raw', 'untreated' slot-related information from the class definition form, while instances of the second hold information that reflects the actual, run-time properties of the slots after the CLOS inheritance rules have been applied. If, for instance, a slot is defined with the `:initarg` slot option[4]set to a value different from the same option in a slot it shadows, the `standard-direct-slot-definition` will show the child's initialization argument, while the `standard-effective-slot-definition` will show a list of the initialization arguments containing both the child's and the parent's specification.

Recall that we can extract a list of direct slot definitions and effective slot definitions from a *class* metaobject class by using the two accessors `class-direct-slots` and `class-slots` respectively. Once we have a slot definition metaobject in hand, we can extract the following information:

- The slot name, type and allocation may be obtained through `slot-definition-name`, `slot-definition-type` and `slot-definition-allocation`, respectively.

- The initialization form that was supplied in the `defclass` may be retrieved from a `slot-definition` by means of `slot-definition-initform`. If such an initform has been supplied, the initialization process of the class will also have provided a function with no arguments which returns the initform value. Thus, if a slot was defined with the `:initform` option `(+ 1 2)`, the method `slot-definition-initform` will return `(+ 1 2)`, while `slot-definition-initfunction` returns something like: `#<Interpreted-Function (LAMBDA NIL (+ 1 2)) 1238467>`. The form `(funcall (slot-definition-initfunction <slot-definition-metaobject>))` returns 3.

- The methods `slot-definition-initargs`, `slot-definition-readers` and `slot-definition-writers` return lists of the slot initialization argument(s) and reader/writer function specifier(s), respectively.

Note that the slot *value* is **not** stored in the slot definition metaobjects. Remember that there is only one such slot definition metaobject per slot per class. Since every instance has its own value for the slot, such an implementation would be incorrect.

Our persistent object system will augment the internal representation of slots by adding information on whether a slot is transient:

```
(defclass persistent-standard-direct-slot-definition
    (standard-direct-slot-definition)
  ((transientp :initform NIL :reader slot-definition-transient-p)))
(defclass persistent-standard-effective-slot-definition
    (standard-effective-slot-definition)
  ((transientp :initform NIL :reader slot-definition-transient-p)))
```

In section 3.5 we will see how the MOP may be influenced to use these classes instead of their parents when constructing one of our persistent classes.

3.4.3 The *Method* Metaobject Class

Methods are the next building block of the Metaobject Protocol. The metaobjects that implement them hold all the information associated with methods. This includes the information specified in the defining `defmethod`. We can get a hold of *method* metaobjects by using `find-method` as follows:

```
(find-method <generic-function-meta-object>
             <list-of-qualifier-keywords>
             <list-of-class-metaobjects>)
```

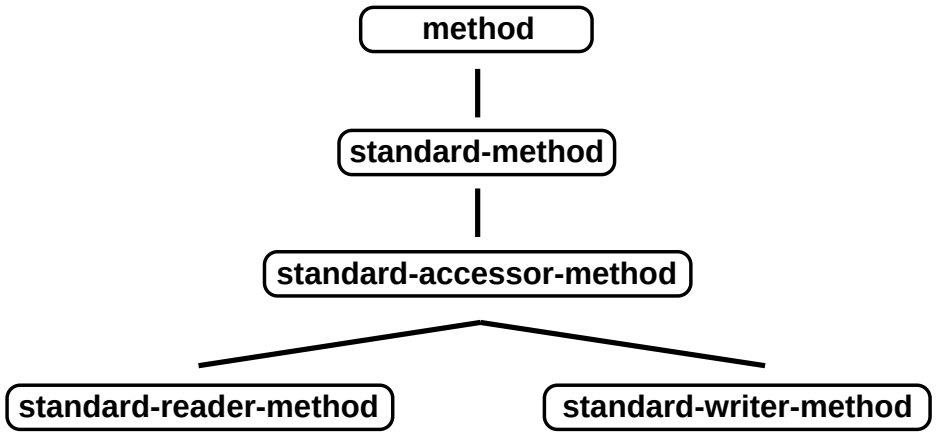


Figure 3.6: The *Method* Metaobject Class Subtree

Figure 3.6 shows part of the relevant metaobject class subtree. In order to illustrate the kind of information we can extract from method objects, let us define two hypothetical methods for the persistent hypertext class defined earlier on:

```
(defmethod linking ((source-node hypertext-node)
                    (destination-node hypertext-node))
  (push destination-node (slot-value source-node 'out-links))
  (push source-node (slot-value destination-node 'in-links)))
```

The following `:before` method allows us to observe the linking together of nodes at run-time:

```
(defmethod linking :before ((source-node hypertext-node)
                            (destination-node hypertext-node))
           (format t "Creating link from ~S to ~S.~%"
                   source-node destination-node))
```

Here is how we can obtain the method metaobjects that implement these two methods:

```
(let ((linking-gen-func (symbol-function 'linking)))
  (setq *primary-method* (find-method linking-gen-func
                                      nil
                                      (list (find-class 'hypertext-node)
                                            (find-class 'hypertext-node))))
  (setq *before-method* (find-method linking-gen-func
                                     '(:before)
                                     (list (find-class 'hypertext-node)
                                           (find-class 'hypertext-node)))))
```

Let us see some of what we can find out about these two methods:

- The generic function a method is currently associated with is returned by `method-generic-function` as a generic function metaobject.
- We can find the lambda list and the list of specializers of a method by using `method-lambda-list` and `method-specializers`. Both return lists. The first is a list of the argument names without any of the classes they are specialized to. The second is a list of *class* metaobjects. For both of our methods these would be:

  ```
  (SOURCE-NODE DESTINATION-NODE)
  (#<Persistent-Metalevel-Class HYPERTEXT-NODE>
   #<Persistent-Metalevel-Class HYPERTEXT-NODE>)
  ```

- The qualifiers of a method, finally, are obtained through `method-qualifiers`. This returns a list of qualifier specifications as they are used in the `defmethod` macro. Our primary method would return `NIL`, the `:before` method would return `(:before)`.

This concludes our look at the static part of the Metaobject Protocol. The information presented should be sufficient to extract a large amount of interesting information from the run-time environment of a CLOS program. In the next section we turn to the dynamics of the Protocol.

3.5 Metalevel Dynamics

When we want to go beyond inspection to modifying the behavior of the language, we will often modify the static part of the MOP by subclassing. Most of the time we will then need to modify parts of the dynamics as well. Many times this will involve initializing new information we keep in our metalevel subclasses. Sometimes there will be other run-time work to be taken care of as well. The goal of this section is to explain the sequences of events that take place to effect some of the major behavior patterns of CLOS. This information should be sufficient to locate where to 'hook in' to change these patterns.

As explained in section 3.2, the dynamics of the MOP are captured in a set of protocols. Here is a list of some major ones:

- The class initialization and class finalization protocols control what happens when a new class is defined.
- The instance initialization protocol describes what goes on when a new instance is created and readied for use.
- The dependent maintenance protocol helps in maintaining relationships among metaobjects. Examples are classes and their subclasses, or generic functions and their methods.
- The method lookup protocol determines how the correct method is found when a generic function is invoked.
- The instance structure protocol attempts to formalize just enough of the implementation of instance access to allow for organized modification, while leaving sufficient freedom for implementors.

We will begin with a tour through the process of defining a new class. The creation and initialization of *slot* metaobjects is part of this process. This will be followed by a description of how slot access works[5].

A theme common to most of metalevel CLOS initialization is that a user's definitional macros, such as `defclass` are checked for errors. Then the information supplied is brought into a canonical form that makes further processing easier. Once this has been accomplished, metalevel functions are invoked to create metaobjects and to initialize them.

After the error checking and canonicalization, these processes are the same whether they were initiated by the execution of a defining macro, or whether they were begun by programs. There is, for instance, a specific point in the processing of a class definition where programs wishing to define a new class would begin. We will point out those programmatic entry points into the metalevel machinery as we encounter them.

3.5.1 Class Definition

When CLOS classes are first defined, their superclasses need not necessarily have been defined yet. *Class* metaobjects must therefore be created and initialized as far as possible, without necessarily knowing all necessary details. Later, once all information is available, the inheritance of the class is *finalized.* The following two subsections explain how this happens.

3.5.1.1 Initialization

Figure 3.7 gives a simplified overview of what happens during class initialization. A full overview is available in appendix B. This figure, and similar ones later on in the text, list the various activities that must take place during the course of the protocol. A series of indented subactivities below an entry shows the steps necessary to accomplish that entry. For example, (re)initialization of the *class* metaobject involves the superclass compatibility check and the other subactivities at the same level of indentation. Successive levels of indentation thus represent increasing levels of detail.

Generic functions listed below an entry in parentheses are responsible for accomplishing that entry's task, often with help from the functions listed further down the list. When appropriate, we point out in the figures the functions through which programs may initiate protocols that are normally initiated through macros, such as `defclass`. The figures serve

two purposes: to give a quick overview of a protocol and to show the 'hooks' available to effect changes.

DEFCLASS
1 Syntax error checking
2 Canonicalize information
3 Obtain *class* metaobject
(ensure-class, ⟵ **Programmatic entry**
ensure-class-using-class) ⟵ **Programmatic entry**
3.1 Find or make instance of proper *class* metaobject class
(make-instance, the :metaclass option) ⟵ **Programmatic entry**
3.2 (Re)initialize the *class* metaobject
((re)initialize-instance)
3.2.1 Check compatibility with superclasses
(validate-superclass)
3.2.2 Determine proper *slot-definition* metaobject class
(direct-slot-definition-class)
3.2.3 Create and initialize the *slot-definition* metaobjects
(make-instance, initialize-instance)
3.2.4 Maintain the 'subclasses' lists of superclasses
(add-direct-subclass,remove-direct-subclass)
3.2.5 Initiate inheritance finalization, if appropriate
(finalize-inheritance)

Figure 3.7: Summary of the Class Initialization Protocol

Let us touch on the main pieces of the class initialization process a step at a time.

DEFCLASS Expansion. The goal of the `defclass` macro expansion is to produce a call to the function `ensure-class` which will create the actual *class* metaobject. It is also used for redefining existing classes.

```
ensure-class <name> &key :environment
                    &allow-other-keys
```

Note that this is a regular function, not a generic one because when it is called we have no instance whose class we would specialize to. To illustrate the processing from `defclass` to `ensure-class`, consider the following subclass of our hypertext node:

```
(defclass monitored-hypertext-node (hypertext-node)
  ((access-count   :initform 0 :accessor access-count)
   (security-level :reader security-level))
  (:metaclass persistent-metalevel-class))
```

Here is roughly what we will end up with when this macro is expanded:

```
(ensure-class 'monitored-hypertext-node
              ':direct-superclasses '(hypertext-node)
              ':direct-slots (list (list ':name 'access-count
                                         ':initform '0
                                         ':initfunction #'(lambda () 0)
                                         ':readers '(access-count)
                                         ':writers '((setf access-count)))
                                   (list ':name 'security-level
                                         ':readers '(security-level)))
              ':metaclass 'persistent-metalevel-class)
```

We see that all but the name information from the `defclass` form is passed to `ensure-class` through keyword arguments. The specification of slots deserves special attention. It is the result of step 2 in figure 3.7 and takes the form of a list of *canonicalized slot specifications*. Each of these is itself a list of keyword-value pairs which will be used as keyword arguments when making *slot-definition* metaobjects later on.

This technique of preparing information into a form that can be used directly as an initialization argument later on is a common canonicalization method in the MOP and we will see other examples of it.

The `:initform` entry relays the form that was specified in the class definition. The `:initfunction` entry is a function that, when called, will return the proper initial value.

The next step in the initialization process happens in the generic function `ensure-class-using-class` which is the workhorse of `ensure-class` and is specialized to a particular *class* metaobject class or to `null`.

```
ensure-class-using-class <class> <name> &key :metaclass
                                              :direct-superclasses
                                              :environment
                                         &allow-other-keys
```

It is called either with a *class* metaobject bound to `<class>`, indicating that we wish to *redefine* a class, or with `NIL`, indicating that we are to create a new one.

`Make-instance` is used to create new *class* metaobjects and the regular CLOS instance initialization procedures are used to get the class ready for use: `initialize-instance` takes care of fixing up a new class, `reinitialize-instance` handles existing classes that are to be redefined. Here is what the class initialization protocol calls for when defining a new *class* metaobject.

Superclass Compatibility Check. The first job is to convert the superclass names from the `defclass` form into *class* metaobjects and to make sure there is no clash. This can happen, for instance, when the class being defined and one of its superclasses are of different metalevel classes. The compatibility check is done by the generic function:

```
validate-superclass <class-metaobject> <superclass-metaobject>
```

When constructing a new *class* metaobject class, the designer must decide whether a programmer-level class implemented by his new metalevel class and inheriting from a super that is implemented by a different metalevel class would lead to inconsistencies.

Unless we define a method on `validate-superclass`, the following will lead to an error because the proper `:metaclass` option was not specified and the system therefore defaulted to using `standard-class`:

```
(defclass simple-hypertext-node (hypertext-node)
  ((slot1)))
```

If we were sure that our new metaclass followed a protocol compatible with `standard-class`, we would provide:

```
(defmethod validate-superclass
           ((class persistent-metalevel-class)
            (superclass standard-class))
  t)
```

This would make the above class definition work. Note that incompatibilities can be a pervasive problem because they prevent the user from inheriting existing superclasses which are not under his control. If, for instance, someone else had provided an interesting 'text display' class facility that we want to reuse by mixing it in with `hypertext-nodes`, we must either certify compatibility in a `validate-superclass` method, or that provider must be asked to change his class to use the `:metaclass persistent-metalevel-class` option in his class definition.

Slot Definitions. Next in the process of class definition is the creation of an appropriate *slot-definition* metaobject for each slot which contains the 'untreated' information specified in the `defclass` definition. Recall that 'untreated' means that slot conflicts with inherited attributes have not been resolved yet. Once these metaobjects exist, dealing with the slots in the later stages of class initialization and finalization will be more convenient. The generic function `direct-slot-definition-class` is called with the *class* metaobject and the canonicalized slot definitions to find out which *slot-definition* metaobject class should be instantiated to implement each slot. This allows the slot implementation to be controlled either by the *class* metaobject class or by new slot options an implementor might introduce.

The choice of slot implementation is something we need to take care of in our persistence example. Recall that we defined a new `persistent-standard-direct-slot-definition` metaobject class and we must make sure that it is used, instead of `standard-direct-slot-definition`:

```
(defmethod direct-slot-definition-class
    ((class persistent-metalevel-class) initargs)
  (declare (ignore initargs))
  (find-class 'persistent-standard-direct-slot-definition))
```

This will ensure that all slots in persistent classes will be implemented with *our* slot metaobjects. The initargs contain the information about the slot that was provided in the `defclass` form, such as `:initform`, `:allocation` or `:type`. This information may be needed by some methods on this generic function to make their decision, though we do not require it for our purposes here.

Creating Slot Definitions. When `make-instance` is used to create a direct slot definition, all the slot options from the `defclass` form are passed in as initialization arguments. The `standard-direct-slot-definition` metaobject classes therefore have initialization arguments corresponding to each legal slot option. These arguments are then processed and installed in the direct slot definition metaobject as we have seen in section 3.4.

We will need to make some changes to introduce our new `:transient` slot option into the system. As things stand, a class definition, such as:

```
(defclass foo ()
  ((slot1 :transient t)))
```

would produce an error, such as:

```
>>Error: Invalid initialization argument :TRANSIENT for class
         STANDARD-DIRECT-SLOT-DEFINITION
```

In order to allow this new option, we modify our definition for *slot* metaobjects introduced in section 3.4.2 to include an `:initarg` option:

```
(defclass persistent-standard-direct-slot-definition
   (standard-direct-slot-definition)
  ((transientp :initform NIL
       :initarg :transient
       :reader slot-definition-transient-p)))
(defclass persistent-standard-effective-slot-definition
   (standard-effective-slot-definition)
  ((transientp :initform NIL
       :initarg :transient
       :reader slot-definition-transient-p)))
```

After the appropriate direct slot definition metaobject has been created and initialized for each slot specified in the `defclass`, the list is kept with the metaobject so that `class-direct-slots` can retrieve and return it.

Maintaining Class Hierarchy Pointers. Recall that we are to be able to ask for all direct super- and subclasses of any class. Since we validated and recorded the superclasses of our new class as part of this initialization process earlier, we know how the information for the former is obtained. But something must still be done to maintain the information for the latter. This is done through the generic functions:

```
add-direct-subclass <superclass-metaobject> <class-metaobject>
remove-direct-subclass <superclass-metaobject> <class-metaobject>
```

When a class is first defined, a call is made to `add-direct-subclass` for each of the new class' supers. In case of reinitialization, a combination of both functions is used to ensure that all class 'downpointers' are correct.

With this the initialization process of the new *class* metaobject is complete. At some point between now and the time the first instance is made, the final, inheritance-related issues must be resolved.

3.5.1.2 Inheritance Finalization

The class finalization protocol is responsible for controlling everything that has to do with a class' inheritance.

Figure 3.8 shows what needs to happen during the finalization of a class. A full overview is included in appendix B.

Let us go through the protocol a step at a time.

FINALIZE-INHERITANCE ←— **Programmatic entry**

1 Compute the class precedence list
(compute-class-precedence-list)

2 Resolve conflicts among inherited slots with the same name

2.1 Determine proper effective slot definition metaobject class
(effective-slot-definition-class)

2.2 Create the effective slot definition metaobjects
(make-instance)

2.3 Initialize the effective slot definitions
(initialize-instance, compute-effective-slot-definition)

Figure 3.8: Summary of the Class Finalization Protocol

The Class Precedence List. The generic function:

```
compute-class-precedence-list <class-metaobject>
```

computes the linearized list of class metaobjects that are in the hierarchy above the class being finalized. The default methods do this according to the rules of official CLOS. We will make a small change here that causes all persistent classes to inherit a class which provides some persistence-related methods, such as `cached?`, `persistent?`, `make-persistent` and so on. We first define that class:

```
(defclass persistence-root-class ()
  ((persistent? :initform T)
   (cached? :initform NIL))
  (:metaclass persistent-metalevel-class))
```

We see that this service class also introduces some slots that are used for house keeping. This is our way of adding system information to **each instance** of our persistent world. Now let us 'sneak' this class into the class precedence list of every persistent class. We do this right when a class is defined.

The `member-if` statement in the following method looks at each superclass in turn and finds out whether any of them is a persistent class. If yes, that super already provides the service class through inheritance and we do nothing special. Otherwise we add our service class to the list of direct superclasses. The `apply` is necessary to make the keyword manipulation work:

```
(defmethod initialize-instance :around
  ((class persistent-metalevel-class)
   &rest all-keys
   &key direct-superclasses)
  (let ((root-class (find-class 'persistence-root-class))
        (pobjs-mc (find-class 'persistent-metalevel-class)))
    (if (member-if
         #'(lambda (super)
             (eq (class-of super) pobjs-mc)) direct-superclasses)
```

```
    (call-next-method)
    (apply #'call-next-method
           class
           :direct-superclasses (append direct-superclasses
                                        (list root-class))
           all-keys))))
```

The next major step in the class finalization is the coalescence of slots: The system needs to find the slots that are defined in multiple classes and must resolve any conflicts that arise in the details of their specifications, such as required value type or initialization.

Resolving Slot Inheritance Conflicts. The first entry point to the slot coalescence activity is:

```
compute-slots <class-metaobject>
```

Its final outcome is a list of `effective-slot-definition` metaobjects, each of which contains all information about one coalesced slot. `Compute-slots` first collects groups of all like-named direct slot definitions from the superclasses and then repeatedly calls the generic function:

```
compute-effective-slot-definition <class-metaobject>
                                  <slot-name>
                                  <direct-slot-definitions>
```

There is one call to this function for each group of conflicting slots. Each time, a single `effective-slot-definition` metaobject is created and returned. As explained earlier, the complete list of these is available through `class-slots` when the process is finished.

As an example, consider a class and its superclass which both provide a slot named 'contents'. The class initialization procedures of the two classes would each have produced one `direct-slot-definition` metaobject which would be kept with the respective class metaobject. During finalization of the subclass, `compute-slots` would construct a list of these two `direct-slot-definition` metaobjects and would call `compute-effective-slot-definition` with that list. The result would be a single `effective-slot-definition` metaobject that records the 'net' properties of the slot for instances of the subclass.

Analogous to the mechanism that allowed `ensure-class-using-class` to create proper direct slot definition metaobjects, the generic function `effective-slot-definition-class` is used to determine which metaobject class should be used for effective slot definition metaobjects. Recall that we defined `persistent-standard-effective-slot-definition` earlier on and we need to ensure that the system uses this class instead of the default:

```
(defmethod effective-slot-definition-class
    ((class persistent-metalevel-class) initargs)
  (declare (ignore initargs))
  (find-class 'persistent-standard-effective-slot-definition))
```

Now we need to ensure that our inheritance rules regarding slot transience will be enforced: A slot will be treated as transient only if all classes in the inheritance chain that define a slot with that name agree that it should be transient. Otherwise the slot will be persistent.

```
(defmethod compute-effective-slot-definition :around
           ((class persistent-metalevel-class)
            slot-name
            direct-slot-definitions)
  ;; Let default system do its work first:
  (let ((slotd (call-next-method)))
    (setf (slot-value slotd 'transientp)
      (every #'slot-definition-transient-p direct-slot-definitions))
    slotd))
```

This example also illustrates how *class* metaobject class incompatibilities discussed in section 3.5.1.1 may introduce subtle problems: our persistence example was written to use `persistent-standard-direct-slot-definitions` for the slots of `persistent-metalevel-class`. All its *slot* metaobjects therefore have a method `slot-definition-transient-p` defined for them. If all classes involved in the inheritance used our metaobject class, the code above would therefore work. If, on the other hand, some of the supers were not using `:metaclass persistent-metalevel-class`, some `direct-slot-definitions` would not have `slot-definition-transient-p` defined on them and the code would fail. To ensure compatibility, we would have to define a default method on `slot-definition-transient-p` that returned `nil`.

We have one more problem to solve in the context of our persistent object system: Whenever a class is defined or redefined to change the number of slots, we must create or modify a corresponding piece of database schema. This can happen through changes to the class itself or through modifications of one of its superclasses. We can handle this conveniently by using the 'chokepoint' introduced in this section:

```
(defmethod finalize-inheritance :after
  ((class persistent-metalevel-class))
  (maintain-schema class))

(defmethod maintain-schema ((class persistent-metalevel-class))
  (if (schema-exists-p class)
    (rework-database-schema class)
    (progn
      (create-schema class)
      (store-class-structure class))))
```

This concludes our look at the creation and redefinition of classes. A full treatment of method definition and invocation would overload this introductory text, but we include the protocol outlines in appendix B.

3.5.2 Slot Access

The last piece of CLOS dynamics we will consider here is the setting and retrieving of slots. When making modifications in this area, the implementor should keep a small checklist of issues in mind:

- The different built-in CLOS slot allocations must be considered (e.g. `:instance` vs. `:class` allocation).
- There is a group of slot access related built-in generic functions that must be kept synchronized: Changes to one could require changes in the other. We will point to examples below.

All the 'official', programmer-level slot traffic goes through the `slot-value` function. This will not generally be true for code generated automatically for reader or writer methods. The entry point for such code is the generic function `slot-value-using-class` and its `setf` dual which are the main point for slot access modifications:

```
slot-value-using-class <class-metaobject>
                       <instance>
                       <effective-slot-definition-metaobject>
(setf slot-value-using-class) <new-value>
                              <class-metaobject>
                              <instance>
                              <effective-slot-definition-metaobject>
```

Figure 3.9 shows the protocol for accessing slots.

SLOT-VALUE-USING-CLASS ←— **Programmatic entry**

1 Check for existence of slot
(`slot-exists-p`, `slot-missing`)

2 Check for slot being bound
(`slot-boundp-using-class`, `slot-unbound`)

3 Retrieve the value

Figure 3.9: Summary of the Slot Reading Protocol

Apart from the generic functions listed in the figure, `slot-makunbound-using-class` should be considered if changes are made to the slot access process.

It is an error to attempt access to a non-existent slot. The Metaobject Protocol allows metalevel programmers to control what happens when this condition is encountered. That enables the programmer to react in a way that makes sense in his modified CLOS context. This control is exercised by defining methods on:

```
slot-missing <class-metaobject> <instance> <slot-name> <operation>
             &optional new-value
```

The `operation` parameter is one of the symbols `slot-value`, `setf`, `slot-bound` or `slot-makunbound`. These can be used to provide a helpful error message.

We need to intercept slot access for our persistent objects to work correctly. The main problem is that we must fault to the database if the object is persistent and not currently cached. In all other cases, we will defer to the built-in way of accessing slots[6].

This brings up a subtle problem that exemplifies the potential dangers of metalevel programming: recall that we record with each instance whether it is persistent and whether it is cached. We did this by causing persistent classes to inherit from `persistence-root-class`, which adds the slots `persistent?` and `cached?`. In order to find out whether an instance is cached or persistent, we therefore need to perform a slot access. Since we must do this to accomplish slot access in the first place, there will be infinite recursion whenever a slot is read, unless we take special precautions. We take care of this in the following code for reading a slot for persistent classes[7]:

```
(defmethod slot-value-using-class :around
  ((class persistent-metalevel-class)
   object
   (slotd persistent-standard-effective-slot-definition))
  (let (
        (slot-name (slot-definition-name slotd))
        (persistent?-slotd
         (find-if #'(lambda (slotd)
                      (eq (slot-definition-name slotd) 'persistent?))
                  (class-slots class)))
        (cached?-slotd
         (find-if #'(lambda (slotd)
                      (eq (slot-definition-name slotd) 'cached?))
                  (class-slots class))))
    (if (and (not (eq slot-name 'persistent?))
             (not (eq slot-name 'cached?))
             (slot-value-using-class class object persistent?-slotd)
             (not (slot-value-using-class class object cached?-slotd))
             (not (slot-definition-transient-p slotd)))
        (slot-value-from-database class slotd)
        (call-next-method))))
```

This concludes our summary of the Metaobject Protocol dynamics. We have seen that each protocol attempts to specify just enough detail about some piece of the CLOS operation to allow controlled modifications to be made. We have covered the process around creating and initializing new classes and the access to slots. Let us now move on to putting the approach into perspective with earlier work.

3.6 Related Work

The concept of making languages extensible concentrated initially on syntactic extension and the creation of new types [9]. Opening languages up for deep semantic changes is a more recent development. This requires the kind of architectural considerations introduced in this chapter.

The idea of making seemingly fundamental components of systems in reality be elements of a meta-level 'world' has been explored in various earlier systems.

Like CLOS, Smalltalk [10] includes the notion of metaclasses. But the concept, though equal in name, is quite different in the two languages: Each Smalltalk class is an instance of exactly one metaclass which in turn may only have that one class as its instance. A class thereby acts like 'regular', program-level objects in the sense that it responds to messages whose effects are determined by its metaclass. In particular, the metaclass controls the initialization of class variables and also manufactures the class' instances. But in contrast to CLOS, the programmer cannot modify metaclasses and use object-oriented programming at the metalevel to produce special effects.

ObjVlisp [11], which is very similar to CLOS [12], has worked on introducing a full metalevel class mechanism into Smalltalk-80 [13]. This has led to a kind of 'metaclass workbench' called *Classtalk* which helps with the construction of metaclass libraries and provides a metaclass browser.

An interesting angle to metalevel architectures is added by [14] which shows how the principle can be used in the construction of operating systems.

There is a rapidly accumulating body of literature about CLOS and its uses. The first, second and third "CLOS Users and Implementors Workshops" of 1988-1990 are good sources for information on a wide spectrum of CLOS aspects. Another report on the use of the Metaobject Protocol can be found in chapter five of this volume.

3.7 Conclusion

This chapter has attempted to introduce the CLOS programmer to the world that lies beyond the confines of the language proper. This world is defined and controlled by the Metaobject Protocol which makes the mechanisms for changing CLOS part of the language definition and thereby renders it portable.

We have introduced the basic notions of this 'metalevel world', giving the reader enough understanding to appreciate the concepts and to read the somewhat more formal specification for more in-depth information.

We believe that the tendency of making systems open should extend beyond areas like networking to the realm of language implementation, operating systems and databases. The CLOS Metaobject Protocol approach is an important step in this direction. Experience during its development has shown that it is difficult to find the correct balance between standardized degrees of freedom and the needs for optimization, between openness and safety, between flexibility and portability. Writing modular systems is more difficult initially than building monoliths. Making systems be open and portable is an additional dimension which requires additional care and sophistication. The payoff, however, is worth the investment because the system covers much more ground than it could with more conventional approaches.

A word of caution is in order at this point. Metalevel programming is still systems programming. Increased power bears with it additional dangers. Research is needed to understand which design rules and conventions can be added to the object-oriented programming style to introduce the necessary measure of safety. We have introduced the rules that were developed for controlling the use of the Metaobject Protocol. More experience is needed to find out whether these rules are necessary and sufficient. More probing still is needed to understand whether they have any universal applicability.

Designing protocols is another area that needs further investigation. The current specification of the Metaobject Protocol is only slightly more formal than our presentation in this chapter. Are there good formal ways of specifying behavior at the right level of detail? Are there indeed formal or informal ways of finding the correct level of detail in the first place? What definitely has to be specified to ensure portability of modifications and what must be left open to allow for optimizations? Understanding the nature of protocol design would go a long way towards making the idea of the CLOS design approach applicable to systems other than languages, a goal that seems intriguing after seeing the CLOS Metaobject Protocol as a datapoint.

3.8 Acknowledgements

This work benefited greatly from detailed and insightful suggestions by Daniel G. Bobrow and Gregor Kiczales. Robin Jeffries contributed through comments on earlier drafts of this chapter.

3.9 Appendix A: Rules for Metalevel Extensions

Different rules apply for implementors of the system and programmers wishing to create portable code which manipulates the metalevel. We do not address restrictions for implementors here but concentrate on the ones applying to portable programs. The following rules all have the same underlying reason: To ensure that new behavior does not modify *existing* system behavior that is relied upon by others:

- For a metalevel program to be portable it must not redefine existing metaobject classes, generic functions, methods or method combinations which are explicitly specified by the MOP.

  In syntactic terms this implies that every new metalevel method must have at least one specializer in its parameters which is not one of the built-in metaobject classes. This means that writing a :before, :after or :around method which specializes only on existing metaobject classes can render a program non-portable. Violating this rule could inadvertently destroy a method provided by the system, or it could cause unexpected side effects for programs using the default implementation. The programmer must produce his own metaobject class and specialize on it.

  Allowing the destructive modification of the existing stock of behavior could also lead to a kind of race condition in which two programs make a modification to the same piece of behavior. The order in which the programs are loaded would then determine the final behavior, which is unacceptable.

- Unless explicitly forbidden by the underlying generic function, it is always legal to *extend* the behavior of an existing method by writing a new one which specializes to relevant subclasses as explained above. But the arrangement must ensure that the original, less specific method will be called. For standard CLOS this means that the new method must be a :before or :after method, or that it is a primary or :around method which calls `call-next-method`. This ensures that any new behavior is *added* to the default behavior, as opposed to replacing it.

- Only if a generic function explicitly allows it, may methods be overridden, that is replaced completely by primary or :around methods that do not use `call-next-method` in their body.

  Note that MOP generic functions often come in 'groups' which must be kept consistent. When overriding one, consistency with the others must be ensured. One example is the group `add-dependent`, `remove-dependent` and `map-dependent`. These groupings are not always explicitly defined in the MOP.

3.10 Appendix B: Protocol Overviews

In order to make the MOP specification easier to follow we include here the full summaries of various protocols that were shortened in the text or are not covered there at all. We explained in section 3.5.1.1 how these figures are to be read.

Class Definition Protocol:
DEFCLASS
1 Syntax error checking
2 Canonicalize information
3 Obtain *class* metaobject
(`ensure-class`, ←— **Programmatic entry**
`ensure-class-using-class`) ←— **Programmatic entry**
3.1 Find or make instance of proper *class* metaobject class
(`make-instance`, the `:metaclass` option) ←— **Programmatic entry**
3.2 (Re)initialize the *class* metaobject
(`(re)initialize-instance`)
3.2.1 Default unsupplied keyword arguments/error checking
3.2.2 Check compatibility with superclasses
(`validate-superclass`)
3.2.3 Associate superclasses with this new class metaobject
3.2.4 Determine proper *slot-definition* metaobject class
(`direct-slot-definition-class`)
3.2.5 Create and initialize the *slot-definition* metaobjects
(`make-instance, initialize-instance`)
3.2.6 Associate them with this new class metaobject
3.2.7 Check `default-initargs`
3.2.8 Maintain the 'subclasses' lists of superclasses
(`add-direct-subclass,remove-direct-subclass`)
3.2.9 Initiate inheritance finalization, if appropriate
(`finalize-inheritance`)
3.2.10 Create reader/writer methods
3.2.11 Associate them with this new class metaobject

Slot Reading Protocol:
SLOT-VALUE-USING-CLASS ←— **Programmatic entry**
1 Check for existence of slot
(`slot-exists-p, slot-missing`)
2 Check for slot being bound
(`slot-boundp-using-class, slot-unbound`)
3 Retrieve the value

Class Finalization Protocol:
FINALIZE-INHERITANCE ←— **Programmatic entry**
1 Compute the class precedence list
(`compute-class-precedence-list`)
2 Resolve conflicts among inherited slots with the same name
2.1 Determine proper effective slot definition metaobject class
(`effective-slot-definition-class`)
2.2 Create the effective slot definition metaobjects
(`make-instance`)
2.3 Initialize the effective slot definitions
(`initialize-instance, compute-effective-slot-definition`)
2.4 Associate them with the class metaobject
3 Enable/Disable slot access optimizations
(`slot-definition-elide-access-method-p`)

Method Lookup Protocol:
Generic Function Call
1 Invoke the generic function's discriminating function
1.1 Find out which methods are applicable for the given arguments
(`compute-applicable-methods-using-classes, compute-applicable-methods`)
1.2 Combine the methods into one piece of code
(`compute-effective-method`)
1.2 Run the combined methods
(`method-function-applier`)

Method Definition Protocol:
DEFMETHOD
1 Syntax error checking
2 Obtain target *generic function* metaobject
(`ensure-generic-function`, ←— **Programmatic entry**
`ensure-generic-function-using-class`) ←— **Programmatic entry**
2.1 Find or make instance of proper *generic function* metaobject class
(`make-instance`, `:generic-function-class` **from** `defgeneric` **form**)
2.2 (Re)initialize the *generic function* metaobject
(`(re)initialize-instance`)
2.2.1 Default unsupplied keyword arguments/error checking
2.2.2 Check lambda list congruence with existing methods
2.2.3 Check argument precedence order spec against lambda list
2.2.4 (Re)define any old 'initial methods'
2.2.5 Recompute the generic function's discriminating function
(`compute-discriminating-function`)
3 Build method function
(`make-method-lambda`)
4 Obtain *method* metaobject
4.1 Make instance of proper *method* metaobject class
(`make-instance`, `generic-function-method-class`)
4.2 Initialize the *method* metaobject
(`initialize-instance`)
4.2.1 Default unsupplied keyword arguments/error checking
5 Add the method to the generic function
(`add-method`)
5.1 Add method to the generic function's method set
5.2 Recompute the generic function's discriminating function
(`compute-discriminating-function`)
5.3 Update discriminating function
5.4 Maintain mapping from specializers to methods
(`add-direct-method`)

Bibliography

[1] International Organization for Standardization. Basic reference model for open systems interconnection, 1984.

[2] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proc. 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*. Computer Society Press, 1987.

[3] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[4] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. Technical Report 88-002R, X3J13 Standards Committee, 1988. (Also published in SIGPLAN Notices, Vol. 23, special issue, Sept. 1988, and in Guy Steele: Common Lisp, The Language, 2nd ed., Digital Press, 1990.).

[5] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley Publishing Company, 1989.

[6] James Bennett, John Dawes, and Reed Hastings. Cleaning CLOS applications with the MOP. In Gregor Kiczales, editor, *Proceedings of the Second CLOS Users and Implementors Workshop*, 1989.

[7] Andreas Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, Springer Verlag, 1988.

[8] Andreas Paepcke. PCLOS: A Critical Review. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1989.

[9] ECL programmer's manual. Center for Research in Computing Technology, Harvard University, TR-23-74, December 1974.

[10] Adele Goldberg and David Robinson. *Smalltalk-80: The Language and its Implementation*. Addison Wesley, 1983.

[11] Pierre Cointe. Metaclasses are first class: The ObjVlisp model. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*. Association of Computing Machinery, 1987.

[12] P. Cointe and N. Graube. Programming with metaclasses in CLOS. In *Proceedings of the First CLOS Users and Implementors Workshop*, 1988.

[13] Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1989.

[14] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In *Proceedings of the European Conference on Object-Oriented Programming*, 1989.

Notes:

[1]The MOP is not part of the official CLOS standard at this time. Its current state is documented in part two of [3].

[2]CLOS was developed using a reference implementation (PCL) which was distributed, critiqued and improved many times before commercial implementations began to emerge.

[3]Notable exception: details about slots are kept in another kind of metaobject which is covered in section 3.4.2 but which is also accessed indirectly through *class* metaobjects.

[4]Recall that an *initarg* is a name that may be associated with a slot and that may later be used in calls to `make-instance` to specify an initial value for that slot.

[5]Interesting protocols we do not cover here include the definition process for generic functions and methods and the addition of methods to generic functions.

[6]This assumes that we make cached objects look like regular CLOS objects. This is actually a very useful way of dealing with caching.

[7]For efficiency, the `persistent?` and `cached?` *slot definition* metaobjects should not be searched for during every slot access as is done by the `find-if` calls in the example. They would be cached in a real system.

Biography:

Andreas Paepcke has been with Hewlett-Packard Laboratories since 1982, working on a wide range of projects, including an infrared network for terminals and workstations, the integration of telephone service into workstation environments, transparent persistence of CLOS objects on a variety of databases and access to information services through object-oriented views. Mr. Paepcke received his BS and MS degrees from Harvard University and his Ph.D. in Computer Science from the University of Karlsruhe, Germany.

Open Implementation Design Guidelines

Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar,
Gail Murphy

Published in proceedings of the 19th International Conference on Software Engineering. ACM Press, 1997.

Open Implementation Design Guidelines

Gregor Kiczales, John Lamping,
Cristina Videira Lopes,
Chris Maeda, Anurag Mendhekar
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 U.S.A
gregor@parc.xerox.com

Gail Murphy
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver B.C. Canada V6T 1Z4
murphy@cs.ubc.ca

ABSTRACT
Designing reusable software modules can be extremely difficult. The design must be balanced between being general enough to address the needs of a wide range of clients and being focused enough to truly satisfy the requirements of each specific client. One area where it can be particularly difficult to strike this balance is in the implementation strategy of the module. The problem is that general-purpose implementation strategies, tuned for a wide range of clients, aren't necessarily optimal for each specific client—this is especially an issue for modules that are intended to be reusable and yet provide high-performance.

An examination of existing software systems shows that an increasingly important technique for handling this problem is to design the module's interface in such a way that the client can assist or participate in the selection of the module's implementation strategy. We call this approach *open implementation*.

When designing the interface to a module that allows its clients some control over its implementation strategy, it is important to retain, as much as possible, the advantages of traditional closed implementation modules. This paper explores issues in the design of interfaces to open implementation modules. We identify key design choices, and present guidelines for deciding which choices are likely to work best in particular situations.

Keywords
open implementation, software design, software reuse

INTRODUCTION
Software has traditionally been constructed according to the principle that a module should expose its functionality but hide its implementation. This principle, informally known as black-box abstraction, is a basic tenet of software design, underlying our approaches to portability, reuse, and many other important issues in computing.

Black-box abstraction has many attractive qualities—amortized development costs, localization of change, etc. Exposing only the functionality of a module in its interface, however, can sometimes lead to performance difficulties when the module gets reused. It has been observed that in such cases, clients "code around" the problem either by re-implementing an appropriate version of the module or by using existing modules in contorted ways [5, 6]. In either case, many of the goals that motivated creating the module in the first place are not actually realized.

Many recent systems address this problem by having modules that allow client control of their implementation strategy [7, 8, 9, 10, 11, 12,]. We say that these modules have open implementations.

The open implementation approach works by somewhat shifting the black-box guidelines for module design. Whereas black-box modules hide all aspects of their implementation, open implementation modules allow clients some control over selection of their implementation strategy, while still hiding many true details of their implementation. In doing this, open implementation module designs strive for an appropriate balance between

preserving the kind of opacity black-box modules have, and providing the kind of performance tailorability some clients require.

A number of existing systems have open implementation style interfaces, but thus far, there has been no systematic study of open implementation design, and as a result, designers of these systems have had little or no general guidance to assist them. This paper addresses this need by examining a series of specific modules with open implementations, including designs taken from published systems and toy designs that illustrate specific issues. The designs serve to illustrate important concepts, guidelines, and tradeoffs. They also provide concrete instances to study and use as idioms in future designs.

This paper is specifically focused on the design of interfaces to modules with an open implementation. While the implementation techniques that support these interfaces are crucial, they are beyond the scope of this paper.[1] Neither does this paper focus on the general motivation for open implementation—that can be found in [13, 14, 6, 15, 16]—instead we operate from the premise that some modules can benefit from the open implementation approach, and focus on issues in the design of their interfaces.

A BASE CASE

Before we begin an exploration of open implementation interface designs, it is necessary to provide a basis for the terms *module* and *interface*. We use these terms in a similar fashion to [17] where a module represents a work assignment, and an interface is the set of assumptions a client programmer using the module may make about its behavior.[2] The modules subject to an open implementation are conceived in the same manner as any other module, namely by the application of the information hiding principle [18]. According to this principle, modules are selected to localize and hide design decisions.

The following interface design for a simple set module will be used as an illustrative example throughout the paper. This black-box interface presents only the functionality of the set module and hides all implementation issues behind the interface. It will serve as a comparison point for subsequent open implementation designs for interfaces to set modules. We are using the set module throughout to help make the differences between the designs more clear. But not all of the designs we present will be appropriate for a module as simple as this. These will noted explicitly.

Set Module Interface Design A

This is the simple "black-box" design. It has the usual procedures for creating sets, adding and removing elements from sets, and mapping over the elements of a set. The calling interface to the module might look something like:

```
makeSet()
insert(item, set)
delete(item, set)
isIn(item, set)
map(function, state, set)[3]
```

Interface design A is attractive in its simplicity. In addition, it adheres to the five characteristics of quality interface designs outlined in [19]. That is, the interface is *consistent* (e.g., the set parameter is consistently passed as the last argument), *essential* (e.g., each service is offered in only one way), *general* (e.g., a set may be used for only insertions, or both insertions and deletions), *minimal* (e.g., each function provides one operation), and *opaque* (e.g., the interface hides the "secret" around which the module has been defined).

It is, however, inherently difficult to develop an implementation of this interface that will please a large range of prospective clients. This difficulty arises because determining the best implementation strategy for a set depends on knowing what is going to be done with it. How many elements will it have? How often will new elements be inserted? Will existing elements be deleted? How often? How often will the other set operations be called? All of these factors are important in determining how to implement a set. This is why there are so many different implementation strategies for sets. The libg++ library [20], for example, has eleven variants of set, including linked lists, B-trees and hash tables, to name a few. But with design A, the set module implementor has little basis for selecting which implementation strategy to use—the interface makes it difficult for the set module to know what a specific client's usage pattern will be. This is, in short, an appropriate case for an open implementation design.

[1] Many of the implementation techniques are straightforward, and will be apparent simply from looking at the interface design. Others are more subtle, and involve recently developed techniques in language and system implementation [1, 2, 4]. There is, as yet, no unified presentation of these techniques; a separate paper describing this is in preparation.

[2] In this paper, we are concerned with guidelines on the selection and form of the interface to an open implementation module. Issues related to the specification of an interface are outside the scope of this work.

[3] The `map` procedure calls the function on every element of the set, passing it both the element and the state block. This design makes it possible to "simulate a closure."

SEPARATION OF USE FROM IMPLEMENTATION STRATEGY CONTROL

The following design addresses the difficulty of developing a reusable implementation of design A by providing clients limited control over the selection of the module's implementation strategy.

Set Module Interface Design B

In this design, the interface is the same as in design A, except that now **`makeSet`** can optionally be called with an argument that describes the client's pattern of use. The intent is that the set module implementation can examine this description and select an appropriate specialized implementation strategy tuned for that pattern of use. The optional usage parameter is a string in a simple declarative language that supports the encoding of information such as the size of the set and the relative frequency with which the various operations are called.

```
makeSet(usage)
makeSet()
insert(item, set)
delete(item, set)
isIn(item, set)
map(function, state, set)
```

The following example calls to **`makeSet`** show how the usage parameter works:

```
makeSet("n=10000,
         insert=lo,
         delete=lo,
         isIn=hi")
makeSet("n=5,
         insert=hi,
         delete=hi")
```

In this design, the opacity criteria have been relaxed somewhat from design A. Whereas design A kept the implementation entirely "secret," design B admits to clients that selecting the implementation strategy is an important issue, and that understanding how the set will be used can help in that selection. But note that most of the secrets remain hidden. The client does not know what the actual implementation strategies are, and they certainly do not know any of the details about how those strategies are implemented.

We begin with a few simple observations about this new interface design:

- It is only a small change from interface design A. The **`makeSet`** procedure now accepts an optional argument; all the other procedures are unchanged.
- The client's use of the new functionality is optional. It is still possible to call **`makeSet`** with no arguments, which will leave the set module free to choose a default general-purpose implementation strategy, much as it would have in design A.
- The client's use of the new functionality has an inherently well-bounded effect. The implementation strategy control associated with a given call to **`makeSet`** affects only the sets created by that call. This makes it possible for some sets to use the new functionality and others not, and for different sets that use the new functionality to do so in different ways to get different implementation strategies.
- The new part of the interface can be seen as being relatively orthogonal to the original interface. The new part supports client control of implementation strategy, whereas the old part supports actually using sets.

The last observation means that set module interface design B effectively splits client code into two kinds: most of the client code simply uses the set module's functionality, while the parameter to **`makeSet`** is involved in controlling the set module's implementation strategy.

This important property is in fact the subject of the first design guideline—***open implementation module interfaces should support a clear separation between client code that uses the module's functionality (use code) and client code that controls the module's implementation strategy (ISC code) .***

A clear separation between client use code and ISC code is important because it helps to preserve the advantages of black-box modules. It helps the client programmer selectively focus their attention on either the way their code uses the module's functionality, or the way their code controls the module's implementation strategy. When focusing on the use code, the client programmer is effectively working with a black-box interface to the module.

Design B does a good job in this respect; the client programmer simply has to selectively ignore the paramater passed to **`makeSet`** in order to focus on use code. It would even be easy to build an automatic tool that could hide the ISC code when the programmer wanted to ignore it.

In working with this guideline, what is most important is the effective separation the client programmer has to work with, as manifested in their code. This goal can be supported by use/ISC separation in the interface, but it is separation in the client code that is the real benefit.

In addition to having a clear separation between client use and ISC code, ***open implementation module interfaces should be designed to make the ISC code optional, make the ISC code easy to disable, and support alternative ISC codes for one piece of use code.*** These additional guidelines provide further support for the development of clients

of open implementation modules. They enable clients to first be developed with a focus on getting the functionality right, by leaving out ISC code. They assist performance debugging, by selectively turning parts of the ISC code on and off. They facilitate porting, by allowing different ISC code for different environments. They support division of expertise, since use code can be written by a person (or group) with one expertise and ISC code can later be written by a person (or group) with another expertise.

One example of a system with clear use/ISC code separation is High-Performance Fortran (HPF) [21], a Fortran extension intended to support efficient data parallel programming. One of HPF's principal components is a set of declarations that allows programmers to assist the compiler (and the runtime system) in determining strategies for distributing arrays across multiple processors. In our terminology, these declarations are ISC code. Clear use/ISC separation is achieved by embedding the declarations into what would be comments in a Fortran-90 program. An example of the use of this mechanism is:

```
REAL A(1000,1000), B(998,998)
!HPF$ ALIGN B(I,J) WITH A(I+1,J+1)
```

where the first line is use code that declares two large arrays and the second line is ISC code saying how to lay out the elements of the arrays with respect to each other.

Scoring the HPF interface design against the use/ISC separation guidelines:

- The use/ISC code separation is clear—the ISC code can easily be ignored by the client programmer or hidden by a tool.
- The ISC code is optional—either HPF or Fortran-90 compilers will compile an HPF program without the ISC code.
- The ISC code is easy to disable—a very simple tool can strip it out of a program before passing that program on to the compiler.
- HPF doesn't directly support multiple ISC codes for one use code, but it is easy to build a tool that does do so, for example by further extending the syntax to mark each line of ISC code with the platform for which it is intended, and then using a pre-processor to strip out inappropriate lines before passing the code off to the HPF compiler.

These properties translate into direct benefits to HPF programmers. Programs can be developed focusing on just the use code. The ISC code can be added later during tuning, possibly by different programmers. Even after the ISC code has been added, the use code is internally complete and executable on its own, so that evolution can be accomplished by first adjusting and testing the use code, and then making any needed adjustments to the ISC code.

```
s1=makeSet("n=1000")
for i = 1 to 700 do
   insert(s1, i+i)
   count(s1)

s2=makeSet("n=5")
insert(s2, 5)
insert(s2, 6)
count(s2)

function count(s) {
   map(.., .., s)
}
```

Figure 1: Scope control in Design B

An example that doesn't do quite as good a job on use/ISC separation is the libg++ library [20], a large library of C++ classes and other building blocks, that includes a set module with an open implementation. But in this design, ISC code is mandatory at set construction, requiring client programmers to always think about the set module's implementation strategy, even in the many cases where a general-purpose strategy would be sufficient. The result is that too many of the benefits of the black-box interface are lost. This also means there is no way to tell from reading the client code whether a particular piece of ISC code was well thought out, or was merely intended to be a default. This makes the code harder to reason about and maintain. The work described in [10] improves on the libg++ design in several ways, one of which is to provide a more clear use/ISC separation.

SCOPE CONTROL

An important observation about design B is that any given piece of ISC code affects the implementation of only some sets—just those sets created by the `makeSet` the ISC code appears in. This important point is the focus of the next design guideline—***open implementation module interfaces should be designed to allow the scope of influence of ISC code to be controlled in a way that is both natural and sufficiently fine-grained.***

Like use/ISC separation, the motivation for this guideline is to help the client programmer understand their program, in this case by making it easier for them to reason about the effect of the ISC code they write. The programmer's reasoning is directly facilitated when the scope of influence of ISC code is natural and fine-grained.

Design B does a good job of meeting this guideline. The ISC code on a specific call to `makeSet` affects only those sets returned from that call (and all the set operations on them). It is natural for the client programmer to think in

terms of sets created by a given call to **makeSet**. This granularity is sufficiently fine grained for the programmer to reason easily about the effect of any piece of ISC code.

Figure 1 shows the effect of design B's scope control from the client programmer's perspective. It shows a number of lines of use code, and two pieces of ISC code, the strings **"n=1000"** and **"n=5"**. The dashed lines indicate what parts of the use code are in the scope of influence of each piece of ISC code. Note that the **count** function, and the call to **map** inside it are in both scopes, since it can be passed sets with either kind of implementation.

Choosing the Scope Control

While the importance of natural and fine-grained ISC code scope control is easy to state, designing an appropriate scope control for an interface can be a subtle problem. Coming up with the design involves considering how and why the client is going to want to control the implementation strategy, and making sure that the design gives clients a fine-enough granularity to work with, without being overly difficult to implement or use. This section presents some alternative scope controls, to illustrate some of the considerations that come into play.

As an alternative scope control for design B, consider a design where the client could only control the implementation strategy on a per-application basis. This might be done with a declaration associated with the makefile for the application, that affected all the sets used by that application. This scope control would not be fine-enough grained, because it is reasonable to expect that an application will want to use sets more than once, and do so in different ways, and thus want different implementations strategies. This alternative design would thus be not much more useful than a closed implementation of sets.

As another example consider file systems that allow the client to control their pre-fetching and caching strategy [22]. These systems tend to provide this control on a per stream basis.[4] A per-file basis would be too coarse a granularity, because it would cause problems if two different clients opened the same file but wanted different implementation strategies. Similarly, ISC scope control on a per-process basis would be too coarse, since it is reasonable to expect that a system running in one process might want to open different streams with different implementation strategies.

While it is important to have sufficiently fine-grained scope control, there is a tension in that the more fine-grained it gets, the harder it can be both to use and to implement. For example, if a file system allowed the client to control the pre-fetching strategy on a per-byte basis—every call to **readByte** could control the pre-fetching that happened with that call—it would undoubtedly be more powerful than on a per-stream basis, but it could be more cumbersome to use and difficult to implement. (Implementation technology capable of supporting such a design does exist however [3].)

There are, however, cases where very coarse ISC scope control has proven useful. Consider for example the BLAS libraries [23] for matrix routines. There are different library implementations customized for different hardware architectures. The library is linked in when execution starts, and affects *all* the matrix arithmetic in the application, but in this case that is an appropriate granularity.

In summary, natural and fine-grained scope control complements clear use/ISC separation. A clear use/ISC separation divides the client code into use code and ISC code. Natural and fine-grained ISC code scope control partitions the client code into parts depending on what ISC code affects them.

SUBJECT MATTER

While design B does address the original need for client control of implementation strategy, the way in which it does so has a few potential weaknesses:

- If a client programmer mis-describes the behavior of their program they may wind up with an implementation strategy that is worse than the default.
- Even if the client programmer properly describes the behavior of their program, they have no guarantee that they will get an implementation strategy that is optimal for their purposes. An implementation of design B might not include an implementation strategy that is optimal for every usage profile a client might describe in a call to **makeSet**.

In essence, design B allows the client to say more about its behavior, but leaves the client unsure about the effect this will have on the module's implementation strategy. Addressing this uncertainty is the motivation for the next design.

Set Module Interface Design C

This design for the set module interface is identical to design B except for the optional argument to **makeSet**. In this design the client programmer has the option to explicitly specify one of a fixed list of implementation strategies for the new set. The fixed list is: **BTree, LinkedList, HashTable**.

```
makeSet(strategy)
makeSet()
insert(item, set)
```

[4] By stream we mean the result of opening the file, that is a handle to the file that can be used to read/or write bytes.

Subject Matter	Client ISC Code	Example
client program's behavior	`n=10000,insert=hi,delete=lo,isIn=hi`	Design B
performance requirements the module must meet at its interface	`bandwith=10000`	Network Quality of Service [24]
module implementation strategy	`HashTable`	Design C

Table 1: Subject matter and Style of ISC Code

```
delete(item, set)
isIn(item, set)
map(function, state, set)
```

Two example calls to **`makeSet`** are:

```
makeSet("LinkedList")
makeSet("HashTable")
```

First we note some of the ways that design C is similar to design B:

- It has similar use/ISC separation, i.e. a parameter of a procedure in the use interface.
- It has similar scope control, i.e. a given piece of ISC code affects only operations on sets returned by that call to **`makeSet`**.

But designs B and C differ in an important respect, having to do with the nature of the ISC code in clients of each. To capture this difference, we introduce a concept called the ***ISC code subject matter*** of an open implementation module's interface design. We use this term to refer to the explicit subject of the ISC code.

In design B, the ISC code subject matter is the client program's behavior. In design C it is the module's implementation strategy. This distinction may appear somewhat subtle, since, after all, both designs allow the client to affect the module's implementation strategy. And pieces of ISC code from designs B and C can have the same intent, even though they have different subject matter, i.e. **`"n=1000, insert=lo, delete=lo, isIn=hi"`** and **`"HashTable"`**. The difference is in what the ISC code is *explicitly* about: the client program's behavior in design B vs. is the module's implementation strategy in design C.

There is a third important possibility for ISC code subject matter—performance requirements the module must meet at its interface. While this subject matter may not be appropriate for the interface to a set module, it is useful in other cases.[5] One example of open implementation modules with this ISC code subject matter is network protocol interfaces that allow clients to request a particular quality of service [24]. Such guarantees are critical for applications, such as audio- and video-conferencing, that send real-time data streams over a network.

The three possibilities for ISC code subject matter are summarized in Table 1.

Tradeoffs

Choosing the ISC code subject matter is a key decision in the design of the interface to an open implementation module. The ISC code subject matter has a significant effect on how easy the module will be to design, specify and implement, as well as how well it will work for its clients.

Making the ISC code subject matter be the client's behavior feels like it should be easier for the client programmer, since all they have to do is figure out the behavior of their program and let the module do the rest. But this isn't always the case. It can often be easier for a client programmer to simply name a well-known implementation strategy that they know will be appropriate. Further, this can give the client programmer more certainty that their ISC code will have the effect they desire. This is why the libg++ set library has module implementation strategy as its subject matter, not client program behavior. (It is more like design C than design B.)

On the other hand, having the ISC code subject matter be the module's implementation strategy opens the door to

[5] The libg++ set library uses the module's implementation strategy as its ISC code subject matter. (It is like design C in that sense.) But, the documentation of the different strategies (**`XPSets`**, **`OXPSets`**, **`SLSets`** etc.) itself includes a description of each strategy's order of complexity (i.e. [a O(n)], [f O(n)], [d O(n)]… for **`XPSets`**), so it describes itself in terms of performance properties at the module's interface.

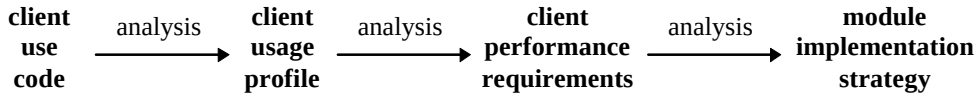


Figure 2: Analysis steps in the process of selecting implementation strategy

potential problems if the client programmer chooses an inappropriate strategy. We are all familiar with the fact that good C compilers ignore register declarations because programmers almost always use them incorrectly. So the interface designer should only make this choice for ISC code subject matter when there is a reasonable chance that the client programmer will be able to choose correctly.

And, while having the subject matter be the performance requirements at the interface seems like a happy compromise, it is not always the best choice either. There are many cases where it is easier for the client programmer to speak in terms of one of the other subject matters.

One rule of thumb for selecting ISC code subject matter is based on seeing the process of selecting implementation strategy as a series of analysis steps: Given the client use code, how does it use the interface? Given a client with that usage pattern, what performance properties does it require? Given those performance requirements, what implementation strategy will best satisfy them? This process is illustrated in Figure 2.

Seeing the process that way, the guideline is: ***Pick the first subject matter along the process of Figure 2 for which all of the following criteria hold:***

- ***It is possible to build an automatic mechanism that completes the chain of reasoning from that point onwards to get an optimal implementation strategy.***
- ***It is easy to design an interface to express the subject matter at that point.***
- ***It would be easy for the client programmer to use that interface to express that subject matter. This includes both figuring out what to say and how to say it.***

Note that this guideline also provides a way of knowing when not to use an open implementation. An open implementation is not needed when all of the steps of the above inference process can be handled automatically to arrive at an optimal implementation strategy.

One example of an appropriate choice of ISC code subject matter is the `inline` declaration found in many programming languages, including C and Common Lisp. This declaration allows the programmer to name an implementation strategy for procedure calling. It comes at the end of the inference process above, and so the programmer has a clear sense of what its effect will be.

A corresponding example of inappropriate choice of ISC code subject matter is the speed/space/safety declarations found in Common Lisp [25]. These declarations don't have a clear subject matter; it isn't clear where they fall in the inference process above, and programmers don't have a clear sense of what their effect will be.

Implementation Details Must be Hidden

Design C further relaxes the original secrets around which Design A was defined. Now, the existence of a fixed set of implementation strategies is no longer secret. But notice that the true details of each strategies implementation is still hidden. There is still plenty of information hiding across the interface between the client and the implementation. This can be stated in a design guideline: ***Open omplementation module interfaces should be designed to pass only essential implementation strategy information.*** The three subject matters are different ways of encoding the essential information.

STYLE OF THE ISC CODE

While design C addresses the lack of guarantees in design B, both designs are limited to whatever set of implementation strategies is provided by the module. This makes them both vulnerable to the implementation not being flexible enough for a wider range of clients. This motivates yet another design.

Set Module Interface Design D

In this design, the use interface is exactly the same as in design C. But this design not only allows client programmers to choose from a fixed set of default implementation strategy, but also allows them to provide entirely new implementation strategies for the set module. The client provides these strategies in the form of an entirely new implementation of the set functionality, packaged up as a subclass of the class **`Set`**. (In this paper we use the mechanism of object-oriented programming to capture this kind of design, but other mechanisms like callbacks or dispatching procedures could be used just as well.)

[6] If there is one implementation strategy that is appropriate for all clients, there is no need for an open implementation.

The following example illustrates the use of interface design D:

In use file

```
makeSet("mySet")
```

In ISC file

```
class mySet (Set) {
  method insert…
  method delete…
  method isIn…
  method map…}
```

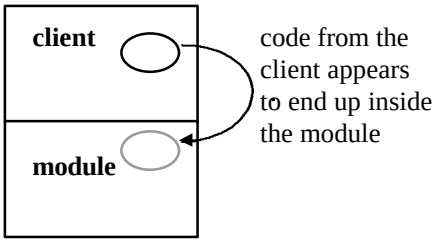


Figure 3: Effects of ISC code in design D.

Design D is similar to design C in many ways:

- It has the same scope control.
- It has similar use/ISC code separation. The key difference in design D is that client ISC code includes not only the code inside the arguments to **`makeSet`**, but also the code that defines any new implementation strategies for sets.
- The ISC code subject matter in this design is the implementation strategy of the module. But in this design, the ISC code takes two different forms. The part inside the arguments to **`makeSet`** is just like in design C, but the part that defines new subclasses of Set is different.

To capture this difference between the declarative ISC code in designs B and C and the programmatic ISC code that in design D, we introduce a new concept, the ***style of the ISC code***.

Declarative style ISC code is simple, but its power is limited to the forms of declarations supported by the interface. This limitation can be problematic when a client has needs that fall out of the purview of these declarations. An interface that supports *programmatic* ISC code addresses this limitation by allowing the client to write ISC code in the form of a small program.

In design D, the set primitives **`insert`**, **`delete`**, **`isIn`** etc. will invoke the client's programmatic ISC code when one of the client-defined implementations is requested. Errors in this ISC code will cause errors seen by the use code. So, unlike the situation in the earlier designs, ISC code has the potential of breaking the use functionality of the interface.

The programmatic style of interface thus can lead to less robust designs. For this reason, it should only be used in cases, such as this one, where otherwise the client would be forced to "code around" the performance deficiency of the module. The use of programmatic ISC code puts a premium on having the right scope control, so as to restrict the consequences of bad programmatic ISC code to those places where it is requested. So, for example, if a buggy backing store is given to the Mach external pager, the whole operating system does not come crashing down. Only the process requesting that backing store is affected.

THE DESIGN SPACE

Figure 4, on the next page, summarizes these four design approaches. It illustrates the progressively deeper involvement of the client in the implementation in the successive styles. The right style to use for a particular module is the one that lets client get as involved in implementation strategy as they need to, without having to get more involved.

Layering

Not only can different clients of a module need different implementation strategies, different clients of a module may also be better served by different interface design styles. Fortunately, this can be accommodated.

Notice that interface design D subsumes both design C and design A. That is, a client of design D has three choices regarding control of the set module's implementation strategy:

1. They can specify no ISC code and get the default implementation strategy.
2. They can choose from the list of the built-in strategies.
3. They can provide a new strategy.

We say that design D is a *layered* interface design.[7] In this design the client can get into the implementation strategy selection process at three different levels. In fact, the first two levels of the above layering have been implicitly present since design B, stemming from the guideline that ISC code should be optional.

Many existing open implementation modules have layering in this sense. The file system mentioned above is one

[7] Layered interface designs refer to the structure of the interface, not to the underlying software structure. A layered interface design might or might not be implemented by a layered software architecture.

Interface Style and *example*	**How Strategy is Selected**	**Tradeoffs**	**When it is Appropriate**
Style A – No implementation strategy control interface	Module selects implementation strategies by observing client's use of the Black-Box Interface.	Same as Black-Box Abstraction.	One implementation strategy will satisfy all clients. Or the module can determine a good strategy by itself.
Style B – Client provides declarative information about its usage pattern. *"sequential file scan."*	Module selects strategy by matching usage pattern information from client to the best available strategy.	Client provided information about its usage pattern doesn't constrain the implementation. Difficult for client to know how it is influencing module strategy.	It is easy to choose an effective implementation strategy if the client behavior is known.
Style C – Client specifies the implementation strategy the module should use. *"LRU cache management"*	Module adopts the strategy specified by client.	Easy to specify exact strategy. However, client might be uninformed or wrong about best strategy to use.	There are a few candidate implementation strategies, but it is difficult to choose among them automatically.
Style D – Client provides the implementation strategy to use. *an object that implements a custom strategy on top of the cache management protocol*	Module adopts the strategy provided by client.	Easy to specify exact strategy. However, designing module to support replaceable strategies might be difficult. For client, building a new strategy implementation might be expensive.	It is not feasible for the module to implement all implementation strategies that clients might need.

Figure 4: open implementation interface styles.

example, that closely parallels design D. The client can do nothing, in which case they get a default pre-fetching policy, or they can choose from a small set of built-in policies, or they can write programmatic ISC code to define a new policy.

A layered interface design aims at exploiting a version of the 90/10 rule. The idea is that 90% of the clients can use the default strategy, the remaining 10% will need to write some ISC code. 90% of that 10% can select from among the built-in strategies, and only the final 1% (but probably a very important 1%) have to provide an entirely new strategy.

Layering is not an end in itself, but a technique to address what might otherwise seem like an irresolvable trade-off. In particular, layering is a way to design an interface that has the robustness and ease of use of declarative ISC code, while at the same time having the power of programmatic ISC code. The guideline is: ***When there is a simple interface that can describe strategies that will satisfy a significant fraction of clients, but it is impractical to accommodate all important strategies in that interface, then the interfaces should be layered.***

OTHER DESIGNS

The range of design approaches presented here are suitable for a large class of open implementations. But there is no room here to cover all the approaches. Two notable omissions are: an approach, particularly used in some open operating systems, that allows incremental definition of new strategies; approaches for allocating shared resources.

These other approaches will be explored in future work.

CONCLUSION

Open implementation is appropriate for reusable modules that have clients with a wide range of different performance requirements. Open implementation is based on reworking the opacity guidelines for traditional black-box modules. In open implementation, modules allow their clients to participate in their implementation strategy, but still hide many aspects of their implementation details. Open implementation requires new design guidelines to augment the existing ones for black-box modules. This paper provides an initial set of such guidelines and issues having to do with:

- Clear use/ISC client code separation
- Natural and fine-grained ISC code scope control

- Selection of appropriate ISC code subject matter
- Selection of appropriate ISC code style
- Incrementality in the ISC interface
- Use of layering to balance ease of use and power

ACKNOWLEDGMENTS

We would like to thank the people who have contributed directly to this paper: Art Lee, Rob DeLine, John Irwin, Jean-Marc Loingtier, and Marvin Theimer.

BIBLIOGRAPHY

1. Kiczales, G., J.d. Riveres, and D.G. Bobrow, *The Art of the Metaobject Protocol*. 1991: MIT Press.
2. Chambers, C. and D. Ungar. *Making Pure Object-Oriented Languages Practical*. in *OOPSLA '91 Proceedings; SIGPLAN Notices*. 1991. Phoenix, AZ.
3. Pu, C. and H. Massalin, *An Overview of The Synthesis Operating System*. 1989: Columbia University.
4. Chiba, S. *A Metaobject Protocol for C++*. in *OOPSLA '95 Conference Proceedings Object-Oriented Programming Systems, Languages, and Applications*. 1995. Austin: ACM Press.
5. Stonebraker, M., *Operating System Support for Database Management*. Communications of the ACM, 1981. **24**(7): p. 412-418.
6. Kiczales, G. *Towards a New Model of Abstraction in Software Engineering*. in *Proceedings of the International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture*. 1992. Tokyo, Japan.
7. Young, M.W., *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.* Vol. Technical report CMU-CS-89-202. 1989: Carnegie Mellon University, Computer Science Department.
8. Hamilton, G. and P. Kougiouris, *The Spring Nucleus: A Microkernel for Objects*. 1993: Sun Microsystems Laboratories, Inc.
9. Yokote, Y. *The Apertos Reflective Operating System: The Concept and its Implementation*. in *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*. 1992.
10. Lortz, V.B. and K.G. Shin. *Combining Contracts and Exemplar-Based Programming for Class Hiding and Customization*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1994. Portland, Oregon: ACM Press.
11. Maeda, C. and B.N. Bershad. *Service without Servers*. in *Fourth Workshop on Workstation Operating Systems*. 1993: IEEE Computer Society Technical Committee on Operating Systems and Application Environments, IEEE Computer Society Press.
12. Anderson, T.E. and others, *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* ACM Transactions on Computer Systems, 1992. **10**(1): p. 53-79.
13. Shaw, M. and W.A. Wulf, *Towards Relaxing Assumptions in Languages and Their Implementations.* SIGPLAN Notices, 1980. **15**(3): p. 45-61.
14. Heninger Britton, K., R.A. Parker, and D.L. Parnas. *A Procedure for Designing Abstract Interfaces for Device Interface Modules*. in *5th International Conference on Software Engineering*. 1981: IEEE Computer Society Press.
15. Kiczales, G., *Beyond the Black Box: Open Implementation.* IEEE Software, 1996. **13**(1): p. 8--11.
16. *Open Implementation Home Page*, Xerox Palo AltoResearch Center, http://www.parc.xerox.com/oi.
17. Parnas, D.L. and P.C. Clements, *A Rational Design Process: How and Why to Fake It.* IEEE Transactions on Software Engineering and Methodology, 1986. **SE-12**(2): p. 251--257.
18. Parnas, D.L., *On the Criteria to be Used in Decomposing Systems into Modules.* Communications of the ACM, 1972. **15**(12): p. 1053-1058.
19. Hoffman, D., *On Criteria For Module Interfaces.* IEEE Transactions on Software Engineering and Methodology, 1990. **16**(5): p. 537--542.
20. Gnu, *Lib G++ Documentation*, http://www.delorie.com/gnu/docs.
21. Steele Jr., G.L., *High Performance Fortran: Status Report.* ACM SIGPlan Notices, 1993. **28**(1).
22. Patterson, R.H. and et al., *A Status Report on Research in Transparent Informed Prefetching*, in *ACM Operating Systems Review*. 1993. p. 21-34.
23. Dongarra, J.J., *et al.*, *An Extended Set of Fortran Basic Linear Algebra Subprograms.* ACM Transactions on Mathematical Software, 1988. **14**: p. 1--17.
24. Zhang, L., *et al.*, *RSVP: A New Resource ReSerVation Protocol.* IEEE Network, 1993(September).
25. Steele Jr., G.L., *Common Lisp the Language*. Second ed. 1990: Digital Press. 1029.

Smalltalk: a Reflective Language

Fred Rivard
Laboratoire Jules Verne
Ecole des Mines de Nantes & Object Technology International Inc.
France
rivard@info.emn.fr

Abstract

As in the LISP tradition, SMALLTALK is almost entirely written in itself. It offers important advantages such as large portability, dynamicity, a fully unified world, graphical user interface builders, connection to databases, powerful development tools, etc. In this paper we discuss the trait that underlies all these features: REFLECTION. We quote one of its definitions and in the first part of this paper go through the different reflective aspects of SMALLTALK. We expand five major aspects in detail: meta-operations, the classes/metaclasses model, the reified compiler, message sending and the behavioral representation through the reification of the executive stack frame of each process. We illustrate their use with significant applications, based both on our industrial and research experiences. In the second part of the paper, we introduce and fully develop *pre/post conditions* in SMALLTALK, dealing with extensions of the model, the compiler, and the development environment.

1 Introduction

SMALLTALK derives its success largely from being not only a language but also an operating system and a development environment as well as producing applications which are extremely portable on multiple platforms. The most important aspect about the language is that, in the LISP tradition, it is almost entirely written in itself. This property makes it an open system that is easily extendable. The implementation of SMALLTALK [Par94b][1] itself is structured as an object-oriented program, expressed in SMALLTALK and organized around meta-level objects representing the classes, methods, lexical closures, processes, compilers, and even the stack frames. SMALLTALK belongs to the field of languages that deals with *reflection*.

"*Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation : **introspection** and **intercession**. Introspection is the ability of a program to observe and therefore reason about its own state. Intercession is the ability of a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called **reification***" [DBW93].

Even if the precise point at which a language with reflective facilities becomes a reflective language is not well defined (and is an interesting issue that merits examination by the reflective community as a whole), SMALLTALK has one of the most complete sets of reflective facilities of any language in widespread use. Although SMALLTALK is not fully reflective due to the pragmatic reason of *efficiency* [GR83], its reflective facilities can provide much of the power of full reflection [FJ89]. This characteristic is responsible for most of its advantages over other industrial object-oriented languages, such as C++ and ADA95.

1.1 Following the Lisp tradition

What probably accounts for a large part of the success of the early LISP interpreters and their different derived dialects, is the great ease with which one can describe and build programs in terms of simple objects such as lists. Taking the trivial example of the addition of two numbers, the program can be described as

```
(cons '+ '(1 2))
```

Thus, one can consider programs as regular data and may use them as such. Futhermore, the program can reason about itself. The idea follows that a program could see

[1] **In this paper, Smalltalk designates the version VisualWorks 2.0 of ParcPlace.**

itself as data, and thus modify itself.

Although SMALLTALK seems to be a little bit more complicated than LISP at first glance, it has kept LISP's approach towards code, regarding and manipulating it as regular data. Taking the creation of simple objects such as points as an illustration, the external representation of a point matches exactly the program that creates it.

`1@2` represents a point where the `x` value is `1` and the `y` value is `2`. Moreover, the execution of this representation, viewed as an expression, returns exactly the point object `1@2`. The internal representation can also be accessed. An object may have a textual representation of its internal state using the message `storeString`, which returns a sequence of characters that is an expression whose evaluation creates an object similar to itself. Thus `(1@2) storeString` returns the string `'Point x: 1 y: 2'`. Explicitly calling the regular evaluator using `Compiler evaluate: '...aString...'`, the evaluation of this next string returns `true`:

```
(1@2) = (Compiler evaluate:
          ((1@2) storeString ))
                                ⇒true
```

Classes, which are complex objects, also have a textual representation.

```
ArithmeticValue subclass:  #Point
        instanceVariableNames:  'x y '
        classVariableNames:  ''
        poolDictionaries:  ''
        category:  'Graphics-Geometry'
```

The above text matches the definition of the `Point` class, which can be obtained by sending the `definition` method to the reified object that represents the `Point` class. Thus the evaluation of a class definition returns an object (a class) that returns exactly the same string when asked for its `definition`.

The SMALLTALK code is stored in what is called a *method*, which corresponds (approximately) to a named LISP lambda-expression. As for classes, a textual representation may be obtained just by sending introspective messages. `[:x | x+1]` is equivalent to the `(lambda (x) (+ x 1))` LISP expression. It is represented by an object from which one can ask for its external textual representation. In order to get their external textual representation, methods and lexical closures, denoted under the vocable *block*, use their internal representation, which mainly comprises bytecodes, as well as a decompiler (which is reified, too). A special tool (`CompiledCodeInspector`) makes the access to this source representation very user friendly, using the mouse and a click on a field.

Therefore, following the LISP tradition, a SMALLTALK program may reason about itself regarding and manipulating the different objects that represent it (textually or internally).

1.2 Meta-Objects

"First, the basic elements of the programming language - classes, methods and generic functions - are made accessible as objects. Because these objects represent fragments of a program, they are given the special name of ***metaobjects****. Second, individual decisions about the behavior of the language are encoded in a protocol operating on these metaobjects -* ***a metaobject protocol****. Third, for each kind of metaobject, a default class is created, which lays down the behavior of the default language in the form of methods in the protocol."* [KdRB91]

Ordinary objects are used to model the real world. *Meta-objects* describe these ordinary objects. As a consequence, meta-objects mostly describe SMALLTALK entities. We quote non-exhaustively major meta-object classes (classified by subject):

1. ***Structure***:
   `Behavior, ClassDescription, Class, Metaclass, ClassBuilder`
2. ***Semantics***:
   `Parser, Compiler, Decompiler, ProgamNode, ProgramNodeBuilder, CodeStream`
3. ***Behavior***:
   `CompiledMethod, CompiledBlock, Message, Signal, Exception`
4. ***Control State***:
   `Context, BlockContext, Process, BlockClosure, ProcessorScheduler`
5. ***Resources***:
   `ObjectMemory, MemoryPolicy, WeakArray`
6. ***Naming***:
   `SystemDictionary, NameScope, PoolDictionary`
7. ***Libraries***:
   `MethodDictionary, ClassOrganizer, SystemOrganizer`
8. ***Environment***:
   `Browser, Inspector, Debugger`

The methods associated with these classes formalize what can be considered as the SMALLTALK MOP.

1.3 Paper Organization

This paper is divided in two parts: the first part is a survey of the reflective capabilities of the language, and the second is an illustrative example of those capabilities. After having presented meta-operations and their use, we focus on the most important reflective subjects: *structure*, *behavior*, *semantic* and *control state*. We describe the involved meta-objects and their classes. We quote significant applications using such objects. As an illustration of reflective manipulations, we introduce *pre/post conditions* in SMALLTALK, dealing with (small) extensions of the model, the compiler and the development environment. We conclude with the current propensity of SMALLTALK to include more and more reflection in recent releases, which we consider as a sign of adaptability to new software engineering challenges.

2 Reflective aspects survey

Rather then going through a complete enumeration of all the reflective facilities of SMALLTALK, we concentrate on the most important ones:

1. **Meta-Operation**: regular objects as metaobjects,
2. **Structure**: classes as regular objects,
3. **Semantics**: compilers as regular objects,
4. **Message Sending**: messages as regular objects (when errors occur),
5. **Control State**: processes as regular objects.

2.1 Meta-Operations

Meta-operations are operations that provide information about an object as opposed to information directly contained by the object. [...] They permit things to be done that are not normally possible (page 195 of [LP90]).

2.1.1 Model

Major meta-operations are defined in the root of the inheritance tree, the class `Object` as methods for:

- **addressing the internal object structure**
  - `Object>>instVarAt:(put:)`[2]
    reads (writes) an instance variable using an index instead of the name of the instance variable,
- **addressing the object meta representation**
  - `Object>>class`
    returns the class of the receiver,
  - `Object>>changeClassToThatOf:`
    changes the class of an object, and thus its behavior. But a heavy restriction of this method is that both classes must define the same format, i.e., describe the same physical structure for their instances,
- **addressing the object identity**
  - `Object>>#allOwners`
    returns an array of all objects referencing the receiver,
  - `Object>>#identityHash`
    returns an integer ranged in 0..16383. It is used to implement dictionary classes[3] which provide efficient access to the objects of a collection using keys,
  - `Object>>#become:`
    swaps references between two objects (the receiver and the argument).

These meta operations consider an object as a meta-object, but an object understands ordinary methods too, such as `printString` or `inspect`. While some classes define only meta-objects (`Class`, `Compiler`, ...), other classes define instances that can be qualified as meta-objects depending on the context in which they are used (`Object`, `Array`(cf 2.3), ...). Therefore, stamping labels on classes based on their meta(or not) instances cannot always be reduced to a dichotomy of choices.

2.1.2 Usage

Introspection is the essence of *reflection*, and so the first applications using structural reflective facilities are tools used to introspect the SMALLTALK system: the `Inspector` class and its subclasses.

An inspector enables the user to look at the structure of an object, and to modify its instance variable values, using `Object>>#instVarAt:(put:)` methods. The inspector uses the inspected object class (`Object>>#class`) to get its instance variable names

[2] `NameOfClass>>selector`: this syntax expresses that the `#selector` method is implemented by the `NameOfClass` class.

[3] `Dictionary`, `IdentityDictionay` classes.

(`Behavior>>#allInstVarNames`) and the index of the instance variables. Notice that these methods allow the programmer to break the encapsulation of an object, and this must only be used in pertinent contexts.

```
(3@4) x                          ⇒ 3
(3@4) instVarAt: 1               ⇒ 3
(3@4) instVarAt: 1 put: 5        ⇒ 5@4
(3@4) class instSize             ⇒ 2
(3@4) class allInstVarNames      ⇒ ('x' 'y')
```

A hierarchy of inspectors is available, allowing specialized inspection on particular objects, such as collections, dictionaries, etc.

```
Inspector
  ChangeSetInspector
  CompiledCodeInspector
  ContextInspector
  DictionaryInspector
  SequenceableCollectionInspector
    OrderedCollectionInspector
```

2.2 Structure

Structural reflection implies the ability of the language to provide a complete reification both of the program currently being executed as well as of its abstract data type[DM95]. SMALLTALK as a unified language only manipulates objects. Each object is an instance of a class that describes both the behavior and the structure of its instances. A class named `Object` defines the basic behavior of every object of the system, such as accessing the class of an object.

2.2.1 Model

Classes as regular objects are described by other (regular) classes called metaclasses[4]. A metaclass has a single instance (except metaclasses involved in the kernel of SMALLTALK). It establishes a couple class/metaclass schema. Inheritance on metaclasses follows the one at the class level (cf Figure 1), defining the SMALLTALK metaclass composition rule. This schema is known as the SMALLTALK-80 schema, and states how metaclasses are composed. It may induce class hierarchy conflicts [Gra89], but for everyday development, the pragmatic SMALLTALK choice suits most needs. Metaclass display is the concatenation of the global name of its sole instance (a class), and the *class* string. As an example, the metaclass of the class *`Object`* is the *`Object class`* metaclass.

The behavior of classes and metaclasses are described by two (meta)classes respectively named `Class` and `Metaclass`. In order for classes to behave as classes, `Object class` inherits from `Class`. In particular the `new` method, enabling object creation, is accessible. This property is often given as the definition of a class. All metaclasses are instances of `Metaclass`, and in particular the `Metaclass class` is also an instance of `Metaclass`, stopping de facto an instantiation of infinite regression. Two abstract classes named `Behavior` and `ClassDescription` regroup the common behavior between metaclasses and classes (for example `new` is defined on `Behavior`).

Finally the class/metaclass kernel of SMALLTALK is self-described with only five classes:

- `Object`
  provides default behavior common to all objects,
- `Behavior`
  defines the minimal behavior for classes, especially their physical representation, which is known by the SMALLTALK virtual machine,
- `ClassDescription`
  implements common behavior for `Class` and `Metaclass` such as category organization for methods, named instance variables, and a save (`fileOut`) mechanism,
- `Class`
  describes regular class behavior,
- `Metaclass`
  describes regular metaclass behavior.

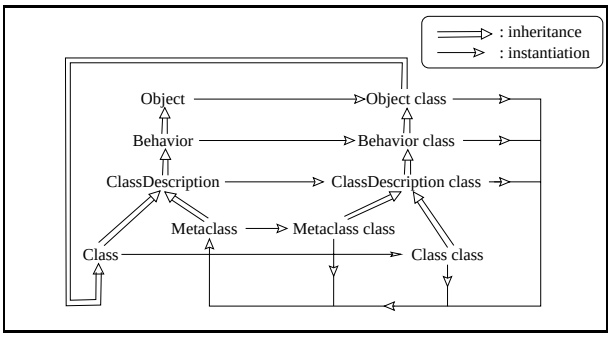


Figure 1: SMALLTALK *class/metaclass kernel.*

The SMALLTALK-80 kernel has pragmatic origins, resulting from several years of intensive development using simpler models that chronologically were SMALLTALK-72 [KG76] and SMALLTALK-76 [Ing78]. In order to keep

[4] Metaclass definition: classes whose instances are classes themselves.

an "easy to use" model, a tool named `ClassBuilder` hides the apparent complexity of the kernel from the end-user. A class creation (and its associated metaclass creation) is fully managed by the tool, which is called by the class creation protocol[5]. It also automatically manages class redefinition, guaranteeing system consistency in terms of object structures and preventing name conflicts, especially instance variable name conflicts. When a class definition changes, existing instances must be structurally modified in order to match the definition of their new class. Instead of modifying an existing object, the `ClassBuilder` creates a new one with the correct structure (i.e., from the new class that replaces the old one). It then fills this new object with the values of the old one. The `ClassBuilder` uses the `become:` primitive (cf 2.1.1) to proceed with the strutural modifications, by replacing[6] the old objects with the new ones throughout the entire system.

Methods are held by classes in an instance variable `methodDict`, whose value is an instance of the `MethodDictionary` class. It enables access to the SMALLTALK
code. It also allows methods to be dynamically added at runtime (`ClassDescription>>compile:classified:`). The `ClassOrganizer` class provides an organization of methods according to their purpose in protocols and every class holds such an organization in the instance variable `organization`. Classes themselves are grouped into categories according to their purpose. `Smalltalk organization` represents the organization of classes. It is an instance of the `SystemOrganizer` class which is a subclass of the `ClassOrganizer` class.

2.2.2 Usage

An ordinary use of the self-expressed kernel is to extend it in order to match new application domains. Our next pre/post conditions example (cf 3) is such an extension. As another typical example, CLASSTALK [Coi90] proposes an experimental platform (an extension of SMALLTALK) to study explicit metaclass programming. But even in the language, reification is of great benefit allowing introspection using dedicated tools: `Browser`. It manipulates classes and metaclasses as regular objects. Thus, it can investigate their definitions `ClassDefinition>>#definition` and their inheritance links, following the reified `superclass/subclasses` instance variables.

[5] `subclass:instanceVariableNames:classVariableNames: poolDictionnaries:category:`

[6] These are actually pointer manipulations

The `Browser` organizes the user external interface according to the information held by the different reified organizations (cf Figure 2):

- A list pane showing the categories, using `Smalltalk organization`,
- A list pane showing class names,
- A list pane showing the protocols of a selected class,
- A list pane showing the selectors of a selected protocol,
- A text pane for method edition, class definition edition, class comment, ...

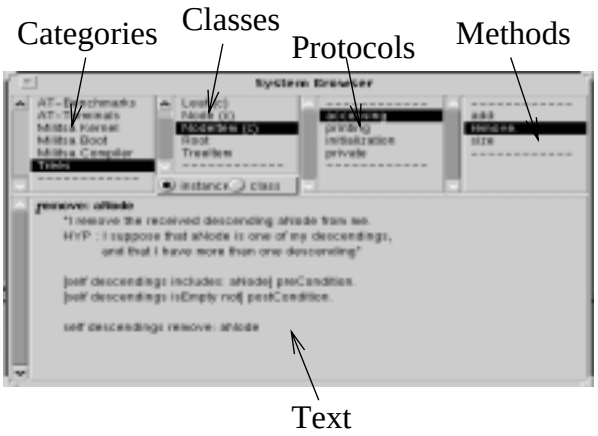


Figure 2 :SMALLTALK *browser with the different panes.*

The reification of classes allows the language to provide essential efficient utilities such as *implementors* (look into all classes for methods matching a given name), *senders* (look into all methods for the ones performing a given sending message) and *messages* (look for implementors of a message present in a given method).

```
Point selectors
    ⇒ IdentitySet( #x #y #transpose ...)
Point compiledMethodAt:  #+
    ⇒ Point>>+
Point findSelector:  #class
    ⇒ #( Object Object>>class)
Point superclass
    ⇒ ArithmeticValue
Point compilerClass
    ⇒ Compiler
```

2.3 Semantics

One of the salient features of SMALLTALK is the fully reified compilation process. Since any compiler implicitly gives the semantics of the language it compiles, and

because SMALLTALK has in itself, as regular objects, its own compiler, the SMALLTALK semantics is fully controllable. Therefore one may extend the current language semantics providing new compile-time features by extending/modifying current compilers.

This approach must be compared to the one of compile-time MOP [LKRR92], which breaks the compilation process into small independent fully redesignable pieces. SMALLTALK compilation uses the existing SMALLTALK code for its own needs, and is designed as a regular OO program which is causally connected to the language. Thus, using current OO technology, one can extend the current compilation process. Next we describe what can be considered as the first compile-time MOP. But the heavy interaction between what is part of the compiler and what is not sometimes makes the use of this compile-time MOP difficult. Therefore the authors of [HJ95] proposes a more parametrized compiler. This big interconnection between the compilation phase and the SMALLTALK language as a whole is demonstrated by the next small example, which discusses the order of argument evaluation of a message send. The compilation process uses the regular `do:` method from the `SequenceableCollection` class, allowing the treatment of each element of a collection in a left to right order. Therefore, it defines a left to right semantics for the argument evaluation order. In that, the `SequenceableCollection` class can be seen as a part of the compilation process because it defines the semantics of the argument evaluation order. Notice that the array that is used to hold the arguments of a message at compile time is therefore a meta-object(cf 1.2) but other arrays would not necessarily be meta-objects.

2.3.1 Model

The two separated parts of the compilation process, parsing and code generation, are described by class hierarchies. We first describe them, and then proceed with their order of execution for compiling method source.

- `Parser`: it produces a parse tree whose nodes are `ProgramNode`. The SMALLTALK syntax is concise, as it only requires method definition. A method is described by a keyword associated with argument names[7] followed by an optional temporaries list and an optional expressions list. Expressions are assignment, message sending and instance variable access. The parser/compiler also defines pseudo-variables (`self, super, thisContext`) and syntactical objects (`true`, `false`, `nil`, `#( ...anArray...)`, `[...a block closure... ]`),

[7] The pattern may be omitted for evaluation.

- `ProgramNodeBuilder`: programNode generators. They are used by parsers to construct the nodes of the syntax tree. Builders allow the complete disconnection of the (recursive descent) parsing mechanism from its result (the nodes),
- `ProgramNode`: syntactic nodes built by `programNodeBuilders`. They hold the code generation methods `emitEffect:` and `emitValue`. The next hierarchy presents the classes that formalize the SMALLTALK syntactical rules.

ProgramNode
MethodNode
ParameterNode
StatementNode
ReturnNode
ValueNode
ArithmeticLoopNode
AssignmentNode
CascadeNode
ConditionalNode
LeafNode
BlockNode
LiteralNode
VariableNode
LoopNode
SequenceNode
SimpleMessageNode
MessageNode

The `MessageNode` class represents message sending. It implements a tiny macro expansion mechanism at code generation time. The `MacroSelectors` dictionary holds selectors that need expansion[8] and their associated transformation symbols. In order to proceed to its code generation, a `messageNode` first tries to expand itself. It then proceeds to the regular code generation of its expansion, or to the generation of itself if no expansion has occurred. As an example, an `and:` message send is transformed using `transformAnd` into a conditional.

- `CodeStream`: byteCode accumulators during code generating. They hold the compilation context in the form of a chain of environments. A `codeStream` is the argument that is passed to both `emitEffect:` and `emitValue:` methods while the (recusive descent) code generation occurs. The result of the code generation is a `CompiledMethod`,
- `CompiledMethod`: it holds (in the instance variable `bytes`) the array that represents the byteCodes: opcodes defined by the `DefineOpcodePool` class, which

[8] `timesRepeat:`, `ifTrue:`, `ifFalse:`, `and:`, `or:`, `whileFalse:`, `whileTrue:`, `repeat`, ...

defines a set of opcodes of a usual stack-based machine, with a special instruction for message sending. These opcodes are understood by the Virtual Machine (VM). As a matter of fact, when a method is executed for the first time, the VM translates the SMALLTALK bytecodes into codes of the underlying machine. These new native codes are then executed each time the method is used. Changing platforms makes methods return to their initial creation state (i.e., native code generation occurs again at first call). The `CompiledMethod` class is a variable class[9], i.e., instances have a part (called the variable part) that behaves as an array. The literals of a method such as literal arrays and string, are buffered into this variable part. According to VM code limitations, the literal collection size of a method may not be greater than 256 (`ByteCodeStream class>>literalLimitSignal`)[10].
A `CompiledMethod` may return its source, using the `#getSourceForUserIfNone:` method, which asks the `SourceFileManager default` for the corresponding source. If no source is available, a `Decompiler` decompiles the method byteCodes and pretty prints the result,

- `NameScope`: they are linked together in order to build the chain representing the compilation context, also called *the symbol table* in other language compilers. The code generation occurs in a compilation context, which is currently associated with a given class, and its superclasses. When `Object` is reached, the dictionary `Smalltalk` is taken as the repository of system globals. Compilation makes the assumption that the receiver is from the class (or subclasses) to which the method currently being compiled will be added. This is not always true, as when using the `become:` method, for example (cf 2.1.1),

- `Compiler`: they are in charge of the scheduling of the *parsing* and *code generation* phases. Parsers are associated with compilers through the `preferredParserClass` method which returns the parser class needed to parse the text to be compiled.

- `CompilerErrorHandler`: they manage error notifications during code generation. Error management is disconnected from the compilation process, allowing a change of policies. Thus subclasses are provided such as `InteractiveCompilerErrorHandler`, `NonInteractiveCompilerErrorHandler`, `SilentCompilerErrorHandler`. The default behavior is to use an `interactiveCompilerErrorHandler` when compiling from a `browser` and a `nonInteractiveCompilerErrorHandler` when reading source from an external file (`fileIn` action). An `InteractiveCompilerErrorHandler` provides a speller when a new symbol is encountered (`newSelector`), warns the user when a temporary is used before it is initialized (`readBeforeWritten`), watches out for undeclared objects such as temporaries and class variables (`undeclared`), and proposes appropriate corrections to the user (`declareGlobal:from:`, `declareTemp:from:`, `declareUndeclared:from:`),

- `Decompiler`: they are translators of `CompiledMethods` into parse trees (`ProgramNode`). `Decompilers` use a `ProgramNodeBuilder` to produce the parse tree from byteCodes. It allows the complete disconnection of the byteCodes interpretation from the result (usually `ProgramNodes` when using standard `ProgramNodeBuilder`).

All of these classes are part of the compilation process. In order to introduce new semantics into SMALLTALK, one can extend these classes and the associated process that compiles code. We next describe what steps this compilation process follows:

1. While compiling a new method on a class, the class is asked what compiler should be used in order to perform the compilation. This is done through the `Behavior>>compilerClass` method. It returns a compiler class appropriate for the source methods of this class (the default is `Compiler`),

2. The compiler is then asked for its default parser (`preferedParserClass`) in order to proceed with the source analysis,

3. The parser scans the source-stream, picking out SMALLTALK syntactic tokens. According to the token produced by the `scanToken` method, it recursively descends into the rules of grammar (`constant`, `expression`, `primaryExpression`, `temporaries`, `statementsArgs:temps:`, `argument`, `pattern`, `method:context:`, ...methods). Each time a syntactic element is completely defined, the builder is asked to create it. In regular SMALLTALK, `ProgramNodeBuilder` returns `ProgramNode`. The result of the parsing is the root node (a `MethodNode`) of the tree that expresses all the syntactic entities of the method,

[9] `variableSubclass:instanceVariableNames:.....`
[10] This limitation must be taken into account while dealing with large automatically generated methods.

4. The compiler builds a `codeStream`, which is initialized according to the class of the method that is being currently compiled. It builds the different `NameScope`s, linking them together,

5. The syntactic tree is asked for code generation. The root `methodNode` receives the `emitEffect:` method. It recursively asks each node of the tree to generate its respective byteCodes into the `codeStream`,

6. The `codeStream` builds a `CompiledMethod`, according to the bytecodes it has buffered. If there are inner blocks (`BlockClosure`) in the method, which need this method filled in as the outer method, the `codeStream` proceeds to do it.

These steps are summarized in the `translate:noPattern:ifFail:needSourceMap:handler:` method[11]:

```
SmalltalkCompiler>>translate:aStream noPattern:...
  "< 1 >...parsing..."
  methodNode := class parserClass new
        parse:  aStream
        builder:  ProgramNodeBuilder new ...
  "< 2 >...code generation..."
  codeStream := self newCodeStream.
  methodNode emitEffect: codeStream.
  method :=
        codeStream makeMethod: methodNode.
  ↑method
```

2.3.2 Usage

Extending the proposed semantics by intervening in the two phases of compilation allows new semantics to be implemented that suit the domain of the application to be modeled as well as possible. The open ended compiler allows modification of itself in order to get improvements needed to face new user requirements, such as a new breakpoint mechanism [HJ95]. The introduction of new methods into the language can be easily performed by subclassing `MessageNode`, in order to propose new message sending semantics. The code generation of this new node will be different, inserting its own semantics. In our experience there are five major methods that are frequently used to add new semantics:

(i) extension of the parser
(ii) extension of the node construction
(iii) modification of the obtained parse tree
(iv) extension of the code generation phase
(v) extension of the compilation environment

Our next pre/post conditions introduction (cf 3) uses a modification of the parse tree (iii). As another example, we provide an efficient implementation of asynchronous message sending for ACTALK [Bri89] (cf 2.4.2), dealing with node construction extension (ii) [Riv95].

Within ACTALK, the user has two message send semantics at his disposal: the regular SMALLTALK one, and an asynchronous one. An asynchronous message send is syntactically declared using the `'a.'` prefix[12].

```
anActor a.message
```

The distinction between the two semantics can be made by a syntactic analysis. Thus, the idea is to intercept the `messageNode` creation made by `aNodeBuilder` (`mewMessageReceiver:selector:arguments:`). We introduce a new class, `ActalkProgramNodeBuilder`, subclassing the regular `ProgramNodeBuilder`. When the new nodeBuilder creates a `messageNode`, it analyzes the selector of the message. If it starts with the `'a.'` prefix, then the `ActalkProgramNodeBuilder` returns `aMessageNode` of which the selector is the one that queues (at runtime) the asynchronous message into the received messages queue of the actor (`addMessage:arguments:`). Thus, for the '`anActor a.message`' expression, the builder returns the next `messageNode`[13]:

```
aMessageNode
  selector : #addMessage:arguments:
  receiver : anActor
  arguments: #( message, #() )
```

Notice that this transformation can be assimilated to a macro-expansion of all `'a.'` prefixed message sends.

More generally, used in association with the kernel extension, compilation reflection allows one to build new languages [RC94]. It allows SMALLTALK to execute source code whose semantics is different from the default one. A large industrial example is given by OBJECT5 [Sie94] [14]. It is a strongly typed hybrid language based both on the actor and class paradigms, dedicated to Programmable-Logical-Controllers. Although it has 3 different message sending semantics (2 are asynchronous), it is entirely executed in SMALLTALK, without an OBJECT5 interpreter being written. This eliminates an always penalizing software stratum. Types have been introduced extending the

[11] We simplified the code for clearer understanding

[12] The `Actor` class provides the behavior for such an actor-object.

[13] See A.1 for the full source of the `newMessageReceiver:selector:arguments:` method of the `ActalkProgramNodeBuilder` class.

[14] a PLC OO framework for Siemens; 20 year/man; currently used in batch or continuous processes.

class/metaclass kernel (`TypedClass` subclass of `Class`) in order to provide typed information (method signature, instance variable types, ...). New syntactical nodes have been introduced, and new compilers, too. Finally the SMALLTALK VM executes this new language as it used to execute regular SMALLTALK. Contrary to the (latent) reproach of the lack of efficiency of reflective systems, here reflection brought an outstanding gain of efficiency.

2.4 Message Sending

2.4.1 Model

The unique control structure of SMALLTALK is message sending. It is composed of two phases:

1. *lookup*: a search for the method to apply according to the receiver of the message sending,
2. *apply*: an application of the found method.

The lookup happens at execution time and uses class information. Although it is not described in the language for reasons of efficiency, the necessary information is accessible and modifiable from the language. All the information lies in classes:

- the dictionary of methods (`methodDict` instance variable: pair (`aSymbol`, `aCompiledMethod`))
- the inheritance link (`superclass` instance variable),
- caches, allowing optimization of the hardwired algorithm. Caches are not reified, but can be reinitialized using primitives (`Behavior>>#flushVMMethod-Cache`).

Messages are not currently reified using instances of the `Message` class except when the lookup fails. In that last particular case, the `#doesNotUnderstand:` method is sent by the VM to the original receiver with a reified message given as the argument.

```
2 zork                                  results in
2 doesNotUnderstand: aMessage           with
aMessage selector      ⇒ #zork          and
aMessage arguments     ⇒ #()
```

An explicit message send may be called using the `perform:` primitive[15]. A lookup result is a `CompiledMethod`(cf 2.3.1), a regular object. The `valueWithReceiver:arguments:` primitive allows the application of `aCompiledMethod` with an array of arguments.

[15] The general form is `perform:withArguments:`.

```
-regular message send:
    5 factorial                         ⇒ 120
-explicit message send using a symbol:
    5 perform: #factorial               ⇒ 120
-application of a CompiledMethod:
    (Integer>>#factorial)
            valueWithReceiver: 5
            arguments: #()              ⇒ 120
```

Accesses to overwritten behavior are qualified by sending a message to the pseudo variable `super`. The lookup semantics of such a message is slightly different from the default lookup, since it starts from the from the superclass of the class which implements the method that executes the `super`. As a matter of fact, the class from whose superclass the lookup starts is accessible within the `compiledMethod` variable part[16] (cf 2.3.1). This class is pushed into the variable part at compile time (`CodeStream>>sendSuper:numArgs:`).

To sum up lookup, SMALLTALK provides two different entry points:

- one that starts the lookup from the class of the receiver,
- one that starts the lookup from the superclass of a class stored in the `compiledMethod` variable part.

Notice that as message sending is the only control structure, an extension of the method semantics provides an extension of the message sending semantics.

2.4.2 Usage

Everything is expressed in terms of sending messages. There is no need for special keywords or special forms, as in BASIC, ADA'95 or C++, etc. As an example, a class declaration is made by sending the `subclass:instanceVariableNames:classVariableNames:-poolDictionary:category:` message with correct arguments. `Browsers` use this facility (cf 2.2.2).

An evaluation is expressed in terms of a default method. Then it is mostly evaluated (using `#valueWithReceiver:arguments:`) with `nil` as the default receiver. The result is either discarded (doIt action), inspected through the sending of the `inspect` message (inspectIt action), or pretty-printed through the sending of the message `printString` (printIt action).

The management of the lookup failure allows the building of a catch-up mechanism by specializa-

[16] using `Object>>at:` and `Object>>at:put:` methods.

tion of the doesNotUnderstand: method, as in the encapsulator paradigm [Pas86], and in the implementation of asynchronous messages for ACTALK [Bri89]. In particular, #valueWithReceiver:arguments: and #perform: methods can be used. More generally, #valueWithReceiver:arguments: enables one to dispense with the use of the default lookup and to implement (in cooperation with the Compiler) new lookup algorithms, such as multiple inheritance. This last approach is an efficient alternative to the use of the doesNotUnderstand:method (cf 2.3.2).

As an example of the use of the doesNotUnderstand method, we describe the implementation of lazy evaluation in SMALLTALK[17].

```
aLazyObject := [ ...aBlock ...] lazyValue.
```

A lazy object represents an execution that may not be required. It does not start execution until at least one message has been received. aLazyObject is used as the regular object that would have resulted from the evaluation of the code inside the block ([...aBlock ...]). Thus it receives messages, such as color if it represents a Car.

```
nil subclass: #Lazy
   instanceVariableNames: 'result done args '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Kernel-Processes'
```

As the Lazy class is a subclass of nil, every message send causes the invocation of the doesNotUnderstand method.

Lazy

```
doesNotUnderstand: aMessage
  done
  ifFalse: [ result:=
              result valueWithArguments: args.
            done := true].
  ↑result perform: aMessage selector
          withArguments: aMessage arguments
```

When it receives its first message, the lazy object forces the evaluation of the block. Therefore it computes the real object, which was previously in a lazy state (i.e., uncomputed). It is buffered for other message sends. An explicit message send, using perform:withArguments:, allows the regular execution scheme to continue.

A classical use of super is the initialization of newly-created objects. When adding a subclass, both new and inherited initializations must be carried out. Thus, the initialize method of the subclass usually looks like:

```
Subclass>>initialize
   super initialize.
   self localInitialization
```

[17] Thanks to Mario Wolczko.

2.5 Control State

The SMALLTALK system is based on reified processes, and more generally on the objects needed to build a multiprocess system. Processes manage time scheduling (timingPriority), event inputs such as keyboard/mouse (lowIOPriority), and regular user evaluations (userBackgroundPriority, userSchedulingPriority, userInterruptPriority).

2.5.1 Model

Processor, the sole instance of the ProcessorScheduler class, coordinates the use of the physical processor by all processes requiring service. It defines a preemptive semantics between processes having different priorities. Processor yield gives processes that have the same priority of the one currently running a chance to run. Semaphore class provides synchronized communication between processes (using wait signal methods). Real time scheduling is provided by the Delay class. It represents a real-time delay in the execution of aProcess. The process that executes a delay is suspended for an amount of (real) time represented by the resumption time of the delay.

The BlockClosure class represents lexical closures. It freezes a piece of code (along with its environment) so that it may be evaluated later on. Blocks can have temporaries and arguments. The general syntactic form is [:arg1 ...:argN| tmp1 ...tmpM | expr1 ...exprP]. Block evaluation is provided by primitives named value, value:, valueWithArguments: depending of the number of arguments. SMALLTALK uses lots of blocks, as in the SequenceableCollection>>do: method for example:

```
do: aBlock
    "Evaluate aBlock with each of the receiver's
    elements as the argument."

    1 to: self size do: [:i | aBlock value: (self at: i)]
```

Process creation is based on blocks; the body of a process is the body of the block. The BlockClosure>>fork method creates a process. As blocks may share an environment, independent processes uses this facility to share common objects. A process may be suspended, resumed or killed (using respectively suspend, resume or terminate methods). The

`interruptWith:` method forces the process that receives it to interrupt whatever it is doing and to evaluate the received block, passed as the argument. The `ProcessorScheduler>>yield` method is a tiny but good illustrative example[18]:

```
yield
    "Give other Processes at the current priority
    a chance to run."
    | semaphore |
    semaphore := Semaphore new.
    [semaphore signal] fork.
    semaphore wait
```

The currently running process (the one that executes this code) creates a new `semaphore`. It proceeds to the creation of a new process (`[...] fork`) that is pushed into the list of the processes that may run (at the same priority). The current running process then suspends itself while it executes the `wait` primitive. The VM then takes the next available process and makes it run. The small created process, which shares the semaphore with the previously running process, will run in its turn. Its only action before dying is to unblock the previously running process using the `signal` primitive on the common `semaphore`.

The most remarkable reflective facility of SMALLTALK is the reification of any process runtime stack, through a chain of linked stack frames, called *contexts* [Par94a]. The pseudo-variable `thisContext` returns the present context of the currently running process. It is an instance of the `MethodContext` class, or the `BlockContext` class.

A context mainly knows (Figure 3):

- the context (`sender`) which has "created" it via the application of a method (cf `valueWithReceiver: arguments:`), or the evaluation of a `BlockClosure` using `#valueWithArguments:` (or `#value #value:` ...),
- the method (`aCompiledMethod` held by a class) currently being executed,
- an instruction pointer, remembering the operand that is actually being executed in the method,
- the receiver of the message, and the arguments. Note that the receiver is an instance of the `BlockClosure` class for `BlockContext`.

[18] We have simplified the code for clearer understanding.

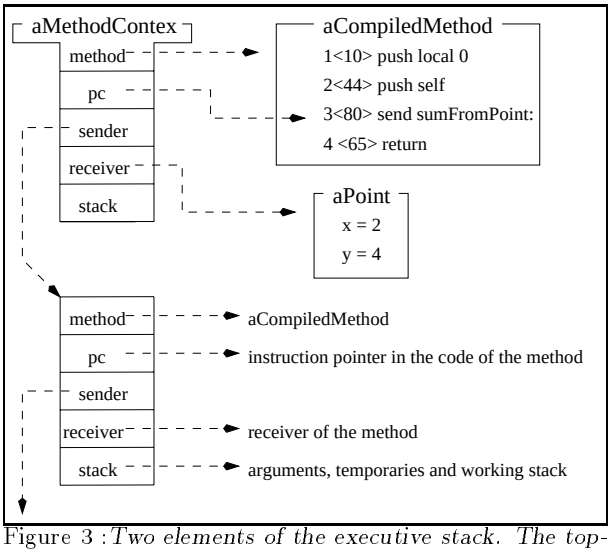


Figure 3 : *Two elements of the executive stack. The topmost* `MethodContext` *represents* `thisContext`.

2.5.2 Usage

SMALLTALK's extreme power of expression allows programs to fully control its own execution, using regular objects such as `Context`: this is intercession.

Therefore, a first application of this execution control is the implementation of the exception handler mechanism into SMALLTALK, which modifies the "regular" execution scheme. The `Exception` class reifies objects which manipulate the executive stack in order to handle errors (`return`, `reject`, `restart`). Exceptions are raised through the stack, and are caught by handlers defined by the `handle:do:` message, in order to take appropriate actions on errors. This implementation may itself be extended or replaced in order to propose an alternative to the error handling system of SMALLTALK [Don90].

A second very important application of the reification of the runtime stack is the `Debugger` tool (see Figure 3), which can:

- consult any context of the entire executive stack,
- look at what part of the selected context is being executed,
- inspect the receiver of the message of the selected context,
- inspect arguments and temporaries of the selected context,
- proceed to a "step by step" execution (`send`, `step`),
- modify any context by recompilation of its method, and continue the execution with this new code.

3 Reflective Extension: Addition of Pre/Post Conditions

Having described the most important reflective facilities, we illustrate their use with a small but complete realization. Dealing with extensions of the model, the compiler, and the development environment, we introduce pre/post conditions on regular SMALLTALK methods. This is a typical way of using the general reflection of the language: add new constructions and extend current facilities in order to provide a language that suits the actual application domain as well as possible. Pre/post conditions fall under the category of software engineering tools.

Applications are not stable during both development and coding phases. Therefore it is essential to provide mechanisms in order to check both the properties of and the assumptions made on methods. Pre/post conditions are devoted to this role. A number of languages, following Flavors [Moo86], implement *before/after methods* (SOM [DFM94], CLOS,...). One of their uses can be the implementation of pre/post conditions on methods. But because before/after methods rely on a complex composition mechanism and because they are assigned to a selector (name of methods) instead of the methods themselves (regular objects in SMALLTALK (cf 2.3.1)), we use another implementation. It better suits their roles as described by: *"The pre-condition expresses the properties that must be checked when the method is called. The post-condition certifies those properties that the method guarantees when it returns."* [Mey90]. When the development is over and the software is about to be released, correct method use makes pre/post conditions no longer useful. They should be removed in order to provide software clean from any development topics. This is how we use pre/post conditions. Our goal is to provide pre/post conditions in SMALLTALK that respect the dynamic and convivial tradition of the language. Specifications are summarized as follows:

- **dynamic behavior** : SMALLTALK users are used to dealing with dynamicity, like adding an instance variable anywhere in a hierarchy of classes. Dynamicity for pre/post conditions means being able to swap from a state where they are *active* to another one where they do not interfere at all with the code,
- **hierarchy independence**: the SMALLTALK model deeply connects a class to its metaclass (cf 2.2.1), of which it is the sole instance. In respect to this model we propose the activation (or deactivation) of pre/post conditions on the class/metaclass couple, but only locally. The activity of conditions on an `A` class does not propagate to `A`'s subclasses,
- **syntactic convention**: instead of extending the syntax with a new special character such as the temporaries delimiter (|), we use a convention. It is an often-used scheme in SMALLTALK, as for example with the `private` protocol, which states that methods from this protocol are supposed to be for private purposes [GR83]. Notice that an extension of the method semantics (using the reified compiler chain (cf 2.3.1)) can provide such privacy,
- **return semantics compatibility**: the return semantics (the ↑ symbol) may require the popping of many contexts. We assume that an active post-condition will be evaluated even when returns occur in the body of methods (or in a block evaluation which closes a return),
- **flexibility**: the code of both pre- and post-conditions may access the method context, especially parameters and temporaries,
- **convivial interface** (cf Figure 4): The interface modifications must be as small as possible. The user can:
  - look at the source of the pre/post conditions associated with a method while browsing the method source (without other manipulations),
  - know through his favorite development tool (`browser`), whether or not conditions are active just by looking at the class name display (class pane of the browser),
  - change the activity of the conditions of a class using a popup menu, as in SMALLTALK's usage.

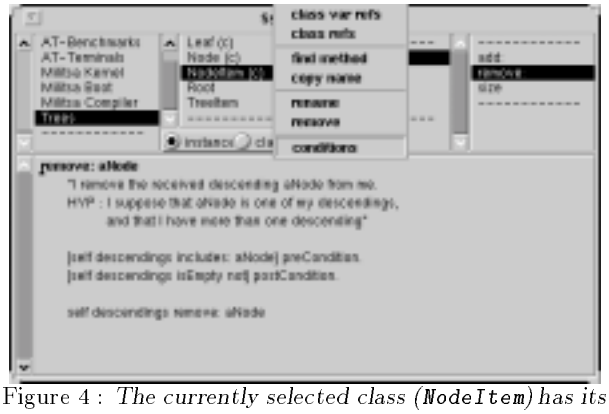


Figure 4 : *The currently selected class (`NodeItem`) has its conditions activeness set (cf (c)). The associated conditions codes is executed at runtime. The figure also shows the menu (`conditions`) that permits the change from active to non-active conditions (and vice versa).*

Next we present the convention used to write conditions (one or both conditions may be omitted):

```
selector
  "comment"
  | temporaries |

  [..blockPreCondition..]  preCondition.
  [..blockPostCondition..]  postCondition.
  expr1 ....  exprN
```

This syntactic representation offers several advantages:

- no "parasitic" methods are introduced, whose semantics would have been derived from their selectors, such as the creation of qualified methods as it is done with method combinations described in [Coi90]. As a matter of fact, this last solution suffers major drawbacks: these qualified methods pollute the interface of the class, and there is no way to prohibit their use as regular methods in another context,
- Using a block to represent a condition allows full access to the method context. It would have been quite difficult to manipulate such a method context with conditions outside the method itself (both temporaries and arguments access would have been hard to realize, for example).

3.1 Model Extension

When not active, pre/post conditions should absolutely not interfere at execution time. This is the most important specification of our method pre/post conditions. This point is crucial. It means that at execution time, we do not allow ourselves to test to see if the conditions are active. Therefore, the test must be done at compile time:

- if conditions are active, then the code needed for their execution is generated at compile time,
- if conditions are not active, then the conditions are ignored and only the regular method body is generated.

Thus, we need two different compilation phases. Changing from active to non-active conditions (and vice versa) is expressed in terms of having a quick recompilation of the class interface[19].

We next describe our solution based on the introduction of a subclass of `Metaclass`[20].

[19] This recompilation does not interfere with the source management.

[20] Conceptually, our extension can be assimilated to the introduction of a new metaclass in a system allowing explicit metaclasses programming, such as ObjVLisp [Coi87].

Considering that the behavior related to conditions activity is both on the class and its metaclass, and that it should not interfere with the inheritance, we put the activity notion on `Metaclass`, and on a newly created subclass named `MetaclassWithControl`. This new metaclass manages behavior according to development topics such as pre/post conditions. The `compilerClass` method (cf 2.3.1) returns the class whose instances (a compiler) are used to compile the methods of a given class. Thus the default `compilerClass` method is conceptually raised one meta level from that of `Behavior` to that of `Metaclass` and `MetaclassWithControl` (cf Smalltalk kernel 2.2.1).

- `Behavior>>compilerClass` returns the compilerClass of the metaclass (i.e., calls one of the next two `compilerClass` methods) (cf A.1),
- `Metaclass>>compilerClass` returns the default compiler that does not take conditions into account (and just forgets their associated codes),
- `MetaclassWithControl>>compilerClass` returns the compiler that deals with conditions codes.

Thus (cf Figure 5),

- the metaclass of a class whose conditions are active is an instance of `MetaclassWithControl`,
- the metaclass of a class whose conditions are not active is an instance of `Metaclass`.

Changing from active conditions to non-active ones is done by dynamically changing the class [Riv96] of the metaclass from `MetaclassWithControl` to `Metaclass` (and vice versa) using the `changeClassToThatOf:` method (cf 1.2).

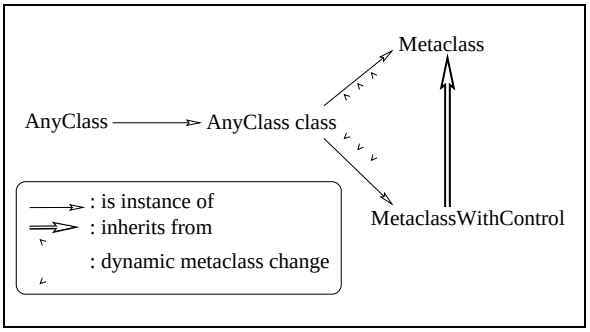


Figure 5 : *The metaclass class changes its class dynamically.*

This solution has many advantages:

- as expected, it allows a class to behave in a certain way, without interfering with inheritance. Indeed, a dynamically added `compilerClass` method (cf 2.3.1) on an `A class` metaclass would have been inherited by all `A class` subclasses. Thus `A` and all its subclasses would have a connected behavior, which is not within our specification. This is due to the parallel inheritance trees provided by both the class and metaclass levels (cf 2.2.1: SMALLTALK model).

- no development topics lie hidden in classes (neither in their definition nor in their interface). This must be contrasted with a solution that would have added an instance variable to the `Class` class definition, in order to remember the activity at runtime. The default `compilerClass` would have to test this instance variable in order to answer the correct compiler. Compared to ours, this last solution is very expensive both in terms of class definition impact and space. Moreover it implies another problem: when an application is released, all its classes have a "development" instance variable always positioned to the same boolean value. It is not reasonable to produce such a class structure. A recompilation of the `Class` class before release is not possible either, because it would no longer be possible to have both released applications and applications in the development stage. In any case, it does not agree with the specification that when not active, conditions should not interfere in any way with regular SMALLTALK.

Finally, notice that this model extension illustrates the great extensiveness of the SMALLTALK kernel. Indeed, if active conditions are put on `Metaclass`, its class (`Metaclass class` (cf 2.2.1)) is an instance of `MetaclassWithControl`, instead of `Metaclass`, which was the kernel "trick" to stop the infinite instantiation regression. Moreover a new loop in the instantiation link appears when `MetaclassWithControl` has its conditions activity set to true. This demonstrates that even the very deepest part of the SMALLTALK kernel (cf 2.2.1) can easily be extended, without causing the whole system to fail.

3.2 Environment Extension

Our choice of syntactic convention allows the method context to be accessible from condition codes. From an interface point of view, the user looks at its method and associated condition sources at the same time. Practical experience shows the advantage of this convivial representation. It is combined with an immediate view of the activity of the class conditions: when a class has active conditions, the name of the class is suffixed by the `(c)` string (cf Figure 4).

As we have extended the model in order to add a new metaclass description to deal with development topics, browsers should also take into account this new description. Standard SMALLTALK browsers, as global introspection class tools, assume that class semantics are fixed. Thus, in order to take new class semantics into account, we modifiy the class interface by adding a cooperation between classes and browsers [RM93]: a browser does not simply ask for the name of the class, but for its `browsingName`. With this message, a class fully controls what a browser shows. `MetaclassWithControl>>browsingName` adds the '`(c)`' string suffix to the name (`classOnControlString` method).

3.3 Compiler Extension

Having designed the structural part of the model and shown its implication in terms of interface extension, we now need to extend the compilation in order to manage the needed codes for active pre/post conditions.

Our solution is based on manipulation of the parse tree, which is generated by the SMALLTALK parser. We need:

- to position the pre-condition (if one exists) as the first statement of the method. We also add the test that raises an exception if the pre-condition evaluation does not return `true` at execution time,

- to position the post-condition (if one exists) as the last statement of the method. As with the pre-condition, we add the test that raises an exception if the post-condition evaluation does not return `true`. As returns may occur (in the method itself or wrap within a blockClosure received as an argument), it could cause the post-condition to not be evaluated. We wrap the entire method using `valueNowOrOnUnwindDo:` which allows execution of the post-condition regardless of what happens.

Next we give an equivalent syntactic form of what could be the code if we were to *decompile* the parse tree after its reshaping:

```
selector
   ''comment''
   | temporaries |

   [[..blockPreCondition..]  value ifFalse:
        [ParserWithControl preConditionSignal
                                raiseRequest].
    expr1 ....  exprN ] valueNowOrOnUnwindDo:[
    [..blockPostCondition..]  value ifTrue:
        [ParserWithControl postConditionSignal
                                raiseRequest]]
```

As we need a new compiler when pre/post conditions are active, the `CompilerWithControl` class is introduced as a subclass of the standard `Compiler` class. We subclass the `Parser` class with `ParserWithControl` class, which is associated with the new `CompilerWithControl` class through a redefinition of its `preferredParserClass` method (cf 2.3.1). We next describe the steps that produce a method and its conditions:

1. the method is parsed as a regular SMALLTALK method. A parse tree is obtained as a result (cf 2.3.1) of the first step of the compilation process,
2. the parser, `aParserWithControl`, reshapes the resultant parse tree to get the previously described transformation. During the transformation, new `ProgramNode`s are created, using the parser builder, `aProgramNodeBuilder` (cf code A.1 `ParserWithControl>>compilePreCondition`).
3. the parse tree generates regular SMALLTALK code.

The regular parser (an instance of the`Parser` class) removes pre/post condition codes, if any.

3.4 Benchmarks

The major goal of this extension is to provide code *free* from any tests when pre/post conditions are not active. Thus, if not active, conditions do not affect the runtime performance at all. When active their code is executed according to the code wrapped around the conditions, which of course takes time.

We make two significant benchmarks on the compilation process:

1. we compare the time taken to compile a method which is free from any conditions both (i) without our extension, and (ii) using our extension with conditions activity set to true. The compilation time increases on average by less than 2% from (i) to (ii), which allows a comfortable use of the extension,
2. we compare the compilation time of (i) a method that has active conditions using our extension and (ii) the equivalent code hand written by the user. (i) is on average 9% quicker than (ii). This results mainly from the fact that the source to parse is smaller when writing conditions using our conditions extension.

4 Conclusion

We have described the current reflective facilities of SMALLTALK. We have presented the most important current aspects: meta-operations, the class/metaclass model, semantics control through the reified compiler, message sending and behavioral representation through the reification of the runtime stack processes. We have fully described an example of reflective use with the introduction of pre/post conditions into SMALLTALK.

As it evolves, SMALLTALK tends to become more and more reflective. In particular we can quote the reification of the dependent link (`DependencyTransformer` class), and the definition of a parser generator (`ParserCompiler` class), written in itself. REFLECTION is the heart of SMALLTALK. It gives the language its great expressive power. Because the language possesses the ability to naturally adapt itself to new application domains, it may be considered as a truly perennial language.

Acknowledgments

I wish to thank all the reviewers for their comments. Thanks to Pierre Cointe who helped me in the organization of the paper. Special thanks to Jacques Malenfant who spent time on the elaboration of the final version of the paper.

References

[Bri89] Jean Pierre Briot. Actalk : A testbed for Classifying and Designing Actor Languages in Smalltalk-80. In *Proceedings of ECOOP'89, Nottingham*, July 1989.

[Coi87] Pierre Cointe. Meta-classes are First Class: the ObjVlisp Model. In *Proceedings of OOPSLA'87*, pages 156–167, Orlando, Florida, December 1987. ACM Sigplan Notices.

[Coi90] Pierre Cointe. The ClassTalk System: a Laboratory to Study Reflection in Smalltalk. In *Informal Proceedings of the First Workshop on Reflection and Meta-Level Architectures in Object-Oriented Programming, OOPSLA/ECOOP'90*, October 1990.

[DBW93] R.G. Gabriel and D.G. Bobrow and J.L. White. *CLOS in Context - The Shape of the Design Space. In Object Oriented Programming - The CLOS perspective.* MIT Press, 1993.

[DFM94] Scott Danforth, Ira R. Forman, and Hari Madduri. Composition of Before/After Metaclasses in SOM. In *Proceedings of OOPSLA'94*, Portland, Oregon, October 1994.

[DM95] Francois-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming : a Short Comparative Study. In *Workshop of IJCAI'95 : On Reflection and Meta-Level Architecture and their Application in AI*, pages 29–38, August 1995.

[Don90] Christophe Dony. Exception Handling and Object-Oriented Programming: towards a synthesis. In *Proceedings of OOPSLA/ECOOP'90*, pages 322–330, 1990.

[FJ89] Brian Foote and Ralph E. Johnson. Reflective Facilities in Smalltalk-80. In *Proceedings of OOPSLA'89, ACM Sigplan Notices*, volume 24, pages 327–335, October 1989.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, The language and its implementation.* Addison Wesley, Readings, Massachusetts, 1983.

[Gra89] Nicolas Graube. Metaclass Compatibility. In *Proceedings of OOPSLA'89, ACM Sigplan Notices*, volume 24, pages 305–315, October 1989.

[HJ95] Bob Hunkle and Ralph E. Johnson. Deep in the Heart of Smalltalk. In *The Smalltalk Report* , july 1995.

[Ing78] D.H.H. Ingalls. The Smalltalk-76 Programming System Design and Implementation. In *5th POPL*, pages 9–17. Tuscon, Arizona, 1978.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* Cambridge, MIT Press, 1991.

[KG76] A. Kay and A. Goldberg. Smalltalk-72 Instruction Manual / SSL-76-6. Technical report, Xerox Parc, Palo Alto, California, 1976.

[LKRR92] J. Lamping, G. Kiczales, L. Rodriguez, and E. Ruf. An Architecture for an Open Compiler. In A. Yonezawa and B. C. Smith, editors, *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture*, pages 95–106, 1992.

[LP90] Wilf R. Lalonde and John R Pugh. *Inside Smaltalk (volume 1).* Prentice-Hall International Editions, Englewood Cliffs, New Jersey, 1990.

[Mey90] Bertrand Meyer. *Conception et Programmation par Objets - version franç aise.* iia - InterEditions tirage 1991, France, 1990.

[Moo86] David A. Moon. Object-Oriented Programming with Flavors. In *Proceedings of OOPSLA'86*, pages 1–8, Portland, Oregon, September 1986. ACM Sigplan Notices.

[Par94a] ParcPlace. `spaceDescription` method of the `ObjectMemory class` metaclass. Description of the `StackSpace` in VisualWorks2.0. Technical report, ParcPlace System, Inc, August 1994.

[Par94b] ParcPlace Systems, Inc, Sunnyvale. *VisualWorks Release 2.0 of 4 August 1994*, 1994.

[Pas86] G.A. Pascoe. Encapsulators: A New Software Paradigm in Smalltalk-80. In *Proceedings of OOPSLA'86, ACM Sigplan Notices*, pages 341–346, November 1986.

[RC94] Fred Rivard and Pierre Cointe. From Envy-Classtalk to Ada9x - Final Progress Report. Technical report, OTI-EMN, December 1994.

[Riv95] Fred Rivard. Extension du compilateur Smalltalk, Application à la paramétrisation de l'envoi de message. In *Actes des Journées Francophones des Langages Applicatifs, JFLA'95.* INRIA - collection didactique, January 1995.

[Riv96] Fred Rivard. Dynamic Instance-Class Link. In *Submission to OOPSLA'96*, February 1996.

[RM93] F. Rousseau and J. Malenfant. Browsing in Explicit Metaclass Languages : an Essay in Reflective Programming Environments. In *Informal Proceedings of the Third Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA'93*, October 1993.

[Sie94] Siemens. Simatic Object 5 Offline. Technical report, Siemens, 1994.

A.1 Code

We give here some major methods for the addition of pre/post conditions into SMALLTALK semantics. The full development can be loaded using ftp at `ftp.emn.fr` under `/pub/rivard/Smalltalk/visualworks2.0/prepost.st`. (We provide a version for visualworks1.0 in `/pub/rivard/Smalltalk/visualworks1.0/prepost.st`.)

Browser
swapControls
"Changing the class of the metaclass to get some compilation controls or vice versa"

```
| metaClass |
className isNil ifTrue:[↑1234].
self changeRequest ifFalse:[↑1234].
metaClass := self nonMetaClass class.
Cursor wait showWhile:[ metaclass swapControl].
className := metaClass browsingName.
self changed: #className
```

Metaclass
swapControls
"I get some compilation and execution controls "

```
self toMetaclassWithControl
```

MetaclassWithControl
swapControls
"I don't want compilation and execution controls any more"

```
self toMetaclass
```

Behavior
compilerClass
"Answer a compiler class to source methods of this class"

```
↑ self class compilerClass
```

Metaclass
compilerClass
"Answer a compiler class to source methods of this class"

```
↑ Compiler
```

MetaclassWithControl
compilerClass
"Answer a compiler class to source methods of this class"

```
↑ CompilerWithControl
```

ClassDescription
browsingName
"Answer an appropriate browsing name."

```
↑ self class browsingName
```

Metaclass
browsingName
"Answer an appropriate browsing name."

```
↑ self soleInstance name
```

MetaclassWithControl
browsingName
"Answer an appropriate browsing name."

```
↑ (super browsingName ,
    self class classOnControlString
        ) asSymbol
```

Parser
compilePrePostCondition
"Just forget about the pre- and post-conditions"

ParserWithControl
compilePrePostCondition
" compile the pre and post condition if they are valid"

```
preCondition isNil
ifFalse:[self compilePreCondition].
postCondition isNil
ifFalse:[self compilePostCondition].
```

ParserWithControl

```
compilePreCondition
    "Adding the preCondition to the beginning of the current statements.
    The tree to produce is something like
    [...the preCondition...] value
    ifFalse:[ CompilerWithControl preConditionSignal raiseRequest]"

    | block statement |
    preCondition := builder     newMessageReceiver: preCondition receiver
                                selector: #value
                                arguments: #() .
    "------error block construction---------"
    block := builder   newMessageReceiver:
                            (builder      newMessageReceiver: ( builder newVariableName: 'CompilerWithControl')
                                          selector: #preConditionSignal
                                          arguments: #())
                       selector: #raiseRequest
                       arguments:#().
    "------preConditionBlock construction------"
    statement := builder     newMessageReceiver: preCondition
                             selector: #ifFalse:
                             arguments: (Array with: ( builder  newBlockArguments: #()
                                                                body: (builder newSequenceTemporaries: #()
                                                                             statements: (Array with: block)))).
    parseNode body statements addFirst: statement.
```

ActalkProgramNodeBuilder

```
newMessageReceiver: rcvr selector: sel arguments: args
  "a.selector are asynchronous messages"
  | ws |
  sel isCompound
  ifFalse: [↑super newMessageReceiver: rcvr selector: sel arguments: args].
  "optimizing for less than 5 arguments. 99.1 purcent of the symbols "
  sel numArgs <= 5
  ifTrue:[ | tab |
    (tab := OrderedCollection with: (self newLiteralValue: sel selectorPart)) addAll: args.
    ws := (String new: 100) writeStream.
    ws nextPutAll: 'addMessage:'.
    sel numArgs timesRepeat:[ws nextPutAll: 'with:'].
    ↑self   newMessageReceiver: rcvr
            selector: ws contents asSymbol
            arguments: tab asArray].
  ws := (String new: 100) writeStream.
  args size timesRepeat:[ ws nextPutAll: 'with:'].
  ↑self   newMessageReceiver: rcvr
          selector: #addMessage:arguments:
          arguments: (Array  with: (self newLiteralValue: sel selectorPart)
                             with: (self newMessageReceiver: (self
                                          newVariableName:'Array' )
                                          selector: ws contents asSymbol
                                          arguments: args))
```

Evaluating Message Passing Control Techniques in Smalltalk

*

Stéphane Ducasse
Software Composition Group, Universität Bern
ducasse@iam.unibe.ch
http://www.iam.unibe.ch/∼ducasse/

Appeared in JOOP (Journal of Object-Oriented Programming) June 1999

Abstract

In a language like Smalltalk in which objects communicate only via message passing, message passing control is a fundamental tool for the analysis of object behavior (trace, spying) or for the definition of new semantics (asynchronous messages, proxy,...). Different techniques exist, from the well known approach based on the specialization of the doesNotUnderstand: method to the exploitation the method lookup algorithm done by the virtual machine. Until now no comparison between these techniques has been made. In this article we compare the different techniques taking into account the reflective aspects used, the scope, the limit and the cost of the control.

Keywords: message passing control, instance specialization, doesNotUnderstand:, error handling, method compilation, anonymous class, minimal object

1 Message Passing Control: A need

Message passing control is the corner stone of a broad range of applications from application analysis (trace[BH90, PWG93], interaction diagrams, class affinity graphs) to the introduction of new language features (multiple inheritance[BI82], interfaces [Sch96], distributed systems[GGM95, Ben87, McC87], active objects [Bri89]...). CLOS is one of the rare languages that made the effort to *explicitly* provide message passing control at the meta-level via its MOP [KdRB91, Bec95]. In Smalltalk, message passing control is not explicitly provided. However, its reflective capabilities allows one to define message passing control using various techniques: The best-known is based on the definition of so called *minimal objects* and the specialization of the doesNotUnderstand: method [Pas86, Lal90, PWG93]. Some other techniques exist like the definition of method wrappers [Bra96] or anonymous classes[McA95].

Up to now, no comparison between these techniques has been made that evaluates their applicability, benefits and drawbacks. This is a problem because each solution possesses good and bad points and often people apply a technique without checking all the consequences of their choice.

In this article we compare these techniques taking into account the reflective aspects used, the controlled objects, the integration of the control into the programming environment, the limit and the cost of the control. We start by giving an overview of the different applications of message passing control in section 1.1. We define the criteria to compare the different techniques. For the sake of understanding, we summarize the reflective facilities of Smalltalk on which such techniques are based. We then present each main technique in detail: error handling specialization in section 2, exploiting the VM method lookup in section 3, and modification of the compiled method in section 4. Finally we conclude with a discussion of message passing control in other languages.

*This research is supported by the Swiss National Science Foundation, grant 2000-46947.96. This article is an extension of the article [Duc97]

1.1 Message Passing Control Applications in Smalltalk

Applications[1] which use message passing control can be roughly sorted into three main categories. The first is *application analysis and introspection* that is based on the development of tools that display interaction diagrams, class affinity graphs, graphic traces [BH90, PWG93, Bra96, Mic96]. The second category is *Smalltalk language extension*. In such a case message passing control allows one to define new features from within the language itself: Garf [GGM95], Distributed Smalltalk [Ben87] or [McC87] introduce object distribution in a transparent manner. Language features like multiple inheritance [BI82], backtracking facilities [LG88], instance-based programming [Bec93b, Bec93a, Hop94], Java interfaces [Sch96] or inter-objects connections [DBFP95] have been introduced. Futures [Pas86, Lal90] or atomic messages [FJ89, McA95] are also based on message passing control capabilities. The third category is the *definition of new object models*, introducing concurrent aspects such as active objects (Actalk [Bri89]) and synchronization between asynchronous messages (Concurrent Smalltalk [YT87]). Other work proposes new object models like the composition filter model [ABV92] or CodA that is a meta-object protocol that controls all the activities of distributed objects [McA95].

1.2 Selected Reflective Features of Smalltalk

Even if Smalltalk is a reflective language [GR89, FJ89, Riv96], it is not possible to change all its aspects. Indeed, the virtual machine (VM) defines the way the objects are represented in memory, and how messages are handled. As message passing control implementations have to use the reflective facilities offered by the VM, we now summarize them.

The Smalltalk dialects referenced are: VisualWorks (previously named ObjectWorks from ParcPlace newly ObjectShare), IBM Smalltalk (integrated into the VisualAge environment of IBM) and VisualSmalltalk (previously Smalltalk/V then Parts of Digitalk). Note that the examples will be presented using VisualWorks and that we will discuss the other solutions when there are significant differences.

[1] Due to the space limitation we limited this short overview to the use of message passing control in Smalltalk.

Reification and Dynamic Creation. In Smalltalk, classes and methods are objects and are described by classes. It is not only possible, as in Java [Fla97], to access to the information that represents such entities but also to modify and dynamically create instances of these classes.

In VisualWorks, classes are dynamically created by invoking the method subclass:instanceVariableNames:classVariableNames:poolDictionaries:category: of the class Class. It is possible to access and modify the inheritance link, the method dictionary and the methods defined in method dictionary of a class (methods superclass, superclass:, methodDictionary:, compiledMethodAt: of the class Behavior in VisualWorks).

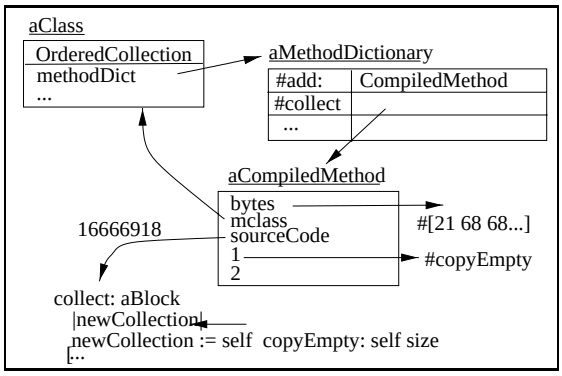


Figure 1: Relationship between class, method dictionary and compiled method in VisualWorks. The collect: method of the OrderedCollection class. The instance variable sourceCode holds an index that is used by the source manager to retrieve the source code for the method.

In VisualWorks, methods are instances of the CompiledMethod class. They can be created by invoking the method compile:notifying: of class Behavior. As shown in figure 1, they are stored in the class method dictionary. A compiled method defines information to access its source code (sourceCode), its compiled byte codes (bytes), the class that compiled it (mclass) and a variable part called the *literal frame* of the method that contains Smalltalk literal objects, such as the symbols, arrays, numbers, byte-arrays and blocks defined in the method.

Note that the source code of a method is stored separately from its byte codes and that a method only needs its byte codes to be executed. The method source can be changed without changing the executable byte codes of the method. Moreover, a compiled method is similar to a Lisp lambda-expression because it does not know its selector. To

know the name of a method (its selector) the class for which it was compiled is asked. A compiled method can be executed without being defined in a method dictionary.

Finally, it is possible to invoke a given method without first doing dynamic dispatch (methods valueWithReceiver:arguments: of class CompiledMethod in VisualWorks and executeWithReceiver:andArguments in IBM Smalltalk). Note that this last functionality did not exist in the first implementations of Smalltalk. This recent addition explains why only a few implementations are based on this possibility.

Moreover, the method perform:with: defined on the class Object allows one to explicitly send a message to any object in the system. anObject perform: #zork with:12 sends to anObject the message whose selector is zork and argument 12.

Changing Reference. The become: primitive allows one to change object references. After invoking a become: b all the pointers that pointed on a point to b and conversely. Note that the semantics of this primitive depends on the Smalltalk implementations: it is symmetric in VisualWorks and asymmetric in IBM Smalltalk.

Changing of Class. An object can dynamically change its class, from a *source* class to a *target* class. This change can be perceived as pointer swap when the two classes possess the same instance structure. In VisualWorks the method changeClassToThatOf: takes as argument an object whereas in IBM Smalltalk the method fixClassTo: takes a class. The implementors of VisualWorks are then sure that the target class is an instantiable class without having to test this at the VM level. The change of class is only possible if the format of the source and target classes are compatible. The format of one class describes the memory layout of its instances (methods format, setFormat: defined on the Behavior class in VisualWorks, and instanceShape, instanceShape: defined on the class Class in IBM Smalltalk).

Message Reification and Error Handling Specialization. When an object receives an unknown message, the Smalltalk virtual machine sends the doesNotUnderstand: message to this object with the reification of the message leading to this error. On the class Object the method doesNotUnderstand: raises an exception which, if it is not trapped (unhandled exception), opens the debugger. This method can be specialized to support message passing control as will be shown in 2.

The reification of the message is done by the VM by creating an instance of the class Message. For example, the message 3 zork: 4 leads to the invocation of 3 doesNotUnderstand: aMessage for which aMessage possesses the following information:

```
aMessage selector   -> #zork:
aMessage arguments  -> #(4)
```

A deontological remark. Some of the functionalities presented above and used in the techniques to be described are qualified as *private* in the Smalltalk versions and therefore are subject to change. It is common use and good style not to use such private methods. However, the internal aspects of the presented techniques imply their use. We stress that if such methods would had been really private some interesting techniques would have been simply impossible.

1.3 Three Main Techniques

First of all message passing control is not limited to the definition of auxiliary methods executed before and after the controlled method. Indeed, a full message control should be able to modify the original arguments, to change the semantics of the message as in remote-calls or even to refuse the execution of a method [DBFP95].

We identified 6 different techniques to implement message passing control. However, some of them are difficult to reproduce or lead to unportable code. That's why we briefly present and sort these techniques before describing the selected ones.

1. Source code modification. One way to control message passing is to instrument the code via source code modification and recompilation. In case of implementing a control similating CLOS-like before and after methods, a controlled method setX:setY: could look as follows after source code modification. As the object responsible for the message passing control is not necessarily the receiver itself, we use an ellipsis to represent it. For example, in the case of meta-object approaches [McA95], the receiver is not its own controller.

```
setX: t1 setY: t2
 ...before
 Original source code
 ...after
```

Note that one might try to use the method aBlock valueNowOrOnUnwindDo: anotherBlock that

allows one to trap the return out of a method. This method evaluates aBlock (the receiver) and when this block exits, it evaluates anotherBlock. However, this is not appropriate, because, as we stated earlier, the execution of the controlled method can be delegated to the message passing control and not limited to additional actions like before and after method executions. Note that to simplify the presentation we will present controlling method body in case of a control simulating before and after CLOS-like methods and we will discuss how this can extended to full control.

The main drawbacks of this technique are: All controlled methods have to be reparsed and recompiled. Moreover, another recompilation is needed to reinstall the original method. This technique is not applicable in deployed or stripped images in which scanners and compilers have been removed.

2. Byte code extension. Smalltalk is based on a byte-code interpreter [GR89, IKM+97], so it is possible to add new byte-code in order to introduce new message passing semantics, like in the Concurrent Smalltalk approach [YT87]. However as the resulting interpreter is no longer standard and the applications are no longer portable, we do not discuss this technique.

3. Byte code modification. Another way to control message passing is to directly insert new byte-code representing the control into the compiled method byte-codes [MB85]. However, implementing this technique is far from simple. More important, it heavily relies on knowledge of the byte code instructions used by the virtual machines. These codes are not standardized and can change.

4. Specialization of error handling. The idea is to encapsulate controlled objects into so called *minimal objects* that do not understand messages and to specialize the doesNotUnderstand: method [Pas86, Bri89, PWG93] (see section 2).

5. Exploiting the VM method lookup implementation. This is realized by explicit subclassing or by the introduction of anonymous classes in the instantiation chain [FJ89, McA95, Mic96, Riv97], or by the definition of a method dictionary array in VisualSmalltalk [Bec93b, Bec93a, Pel96] (see section 3).

6. Method substitution. The idea is to change the compiled method associated to the selector in class method dictionary [BH90, Bra96, Riv97] (see section 4).

The three last techniques can be implemented from within the language itself at a reasonable level of abstraction and, they are portable. That's why we only will present and compare them in detail in the following sections.

Note that we take into account only those techniques that introduce a control of *standard* message passing from the language itself. The key point here is that we want to control objects already defined in the Smalltalk language. Therefore we exclude approaches based on meta-interpreters that define their own explicit message sending [Coi90].

Remark. Message reification allows a particular interpretation of the message semantics such as asynchronous messages [Fer89]. However, message reification on its own does not allow one to control specific objects [Fer89, DBFP95]. Moreover, as mentioned by Adele Goldberg in [GR89] message reification has only been introduced in Smalltalk for error handling due to efficiency reasons. Nevertheless, the combination of message reification and instance-based control techniques offers a wide range of possibilities. For example, in CodA message passing control is implemented using the technique 5, but the message reification provided by the technique 4 is also used for the various message semantics offered in CodA [McA95].

1.4 Some Comparison Criteria

To compare the techniques on a common basis we propose the following comparison criteria.

Control granularity. Sometimes it is necessary to only control one specific message sent to one specific object. In other cases, all the messages sent to a set of objects should be controlled (note that objects can share the same message passing control definition without belonging to the same class).

So a control can be applied to all the instances of one class in a similar manner, or only to certain instances, or only one instance. We call the first possibility a *class-based control*, the second a *group-based control* and the third one an *instance-based control*.

Moreover, we qualify a control as *global* if *all* the messages sent to an instance are controlled, as *class-based* if all the methods of the class of the object are controlled and as *selective* if it is possible to only control certain specific messages.

Environment Integration. Since Smalltalk implementations offer rich programming environments, we also consider the impact of the techniques

on the proposed tools. It is important to know if the browsers and their functionality (senders, implementors, messages, class references, instance variable references,...) continue to work after applying the message passing technique.

Efficiency. To compare the execution costs we consider that the code executed during the control, such as a display in a trace, to be constant for all the techniques. The cost takes into account only the mechanism used to control the invoked method. Moreover, we evaluate if the process requires methods to be recompiled during the installation of and during the reinstallation of the original methods.

Definition Cost. Finally we should mention if the proposed solution is easy to implement or if it needs quite complex mechanisms.

Glossary. *Controlling* entities (classes and methods) are those that implement the message passing control. *Original* entities are those that are normally executed in absence of control.

2 Error Handling Specialization

As presented in 1.2, when an object receives an unknown message the method doesNotUnderstand: is invoked. The technique consists of defining *minimal objects* that will encapsulate the object being controlled. A minimal object is an object for which *ideally* each message provokes an error. Note that to be viable in the Smalltalk environment such an object should possess a minimal set of methods that do not lead to an error. We use the become: primitive to substitute the object to be controlled by a minimal object that encapsulates it.

The figure 2 illustrates the message passing control: (1) the original message is sent, (2) the VM invokes the method doesNotUnderstand: and (3) the original method is executed.

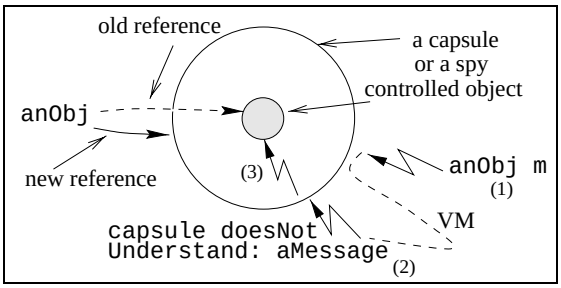


Figure 2: Installation of minimal objects and message passing control by generation and control of errors.

Note that the use of the become: primitive is only necessary when one needs to control *existing* objects of the Smalltalk library [Pas86, Lal90, PWG93, GGM95]. In [Ben87, McC87], the goal is not to control predefined objects but to define controllable objects, so the reference exchange is not necessary: messages are controlled because they are simply unknown for the object. Note that for this particular case the methods inherited from Object class should be recompiled to include control and substitute primitives calls by controllable methods [McC87].

2.1 Minimal Object

The creation of a *minimal object* [Bri89, PWG93], also named *capsule* or *encapsulator*, is based on the creation of a class that does not inherit from Object class. Doing so all the messages sent to an instance of such class invoke the doesNotUnderstand: method and then are controlled. The code to invoke the original method can be the following one:

```
MinimalObject>>doesNotUnderstand: aMessage
 ..."control specific actions"
 originalObject perform: aMessage selector
              withArguments: aMessage arguments
 ...
```

The creation of classes that inherit from nil (the unique instance of the UndefinedObject class whose value means referring to nowhere) does not lead to the desired solution. Indeed Smalltalk allows the creation of new root inheritance classes. To do so, the class creation protocol is redefined on the class UndefinedObject to permit the creation of class that does not inherit from any other class. However, to integrate such classes in the Smalltalk environment, Smalltalk defines a specialized version of the doesNotUnderstand: method that automatically and lazily copies the methods from the Object class. We then obtain an incremental copy of Object class.

The right technique to create a minimal object is the following: (1) creation of a subclass of Object, (2) assignment of the superclass link to nil and (3) definition of the minimal behavior by copying the needed methods from Object. Here follows the code taken from Actalk [Bri89].

```
MinimalObject class>>initialize
 superclass := nil.
 #(doesNotUnderstand: error: ~~ isNil =
   == printString printOn: class inspect basicInspect
   basicAt: basicSize instVarAt: instVarAt:put:)
         do: [:selector | self recompile: selector from: Object]
```

2.2 Problems

This approach implies three main problems identified by [PWG93].

The self problem. The variable self is a pseudo-variable with which objects refers to themselves without using explicit pointers. Messages that an object sends to itself are not redirected to the minimal object and thus not controlled. Moreover, this problem appears not only when an object sends messages to itself. In fact a message can only be controlled if: (1) the message is not sent by the object itself and (2) the reference from the sender of the message to the receiver of the message (the original object) was not installed via a reference to self [PWG93]. The authors of Spies [PWG93] proposed a delicate and costly solution based on the dynamic analysis of the execution stack to detect if the messages sent should or should not be controlled.

Class Control. Control of classes is impossible because classes can not be swapped by objects of different nature. The ClassBuilder uses become: when a class is incrementally defined but the swap is done between two classes.

Minimal Object. As already mentioned a minimal object should define a minimal set of methods such as class, isKindOf,=, ==, instanceVarAt:, myDependents... This leads to the problem of the interpretation by the minimal object of messages that were initially destined for the controlled object. The problem is double because not only is the message executed by the minimal object but the controlled object does not receive the message.

Pascoe proposed a heavy solution that consists in fully duplicating the inheritance hierarchy and to prefix all the methods destined for minimal objects with an E [Pas86]. Even if such a solution works well, it is heavy to set up and uses lots of memory.

2.3 Discussion

This approach proposes an *instance-based control* with a *global* granularity: all the methods are controlled. Contrary to other approaches that presuppose the knowledge of the messages that should be controlled, this approach is the only one to offer the ability to control *all* the sent messages. It is not mandatory to know in advance the potentially controllable messages.

In addition to the above mentioned problems this approach is not efficient as shown in 5.1. Indeed, the control is based on the error of the lookup of the method associated with the message. Thus each control needs one additional lookup and a double traversal of the execution stack due to exception handling. Moreover, each control implies a message instance creation.

This approach is simple to implement when one does not attempt to solve all its inherent problems such as those linked to the identity of the object.

3 Exploiting of the VM Method Lookup Algorithm

In object-oriented programming, the standard approach for specializing behavior is subclassing. In Smalltalk, when an object receives a message, the lookup of the method starts in object class and follows the inheritance link. The super variable allows one to invoke overridden methods. Its semantics is to start the lookup in the superclass of the class in which the method was found.

Controlling sent messages is possible by interposing between the object and its original class a new class that specializes the looked up methods. This can be achieved by an explicit traditional subclassing (see figure 3) or an implicit subclassing based on anonymous classes associated to each instances and a class change (see figure 4).

Common Principle. This approach is composed by three aspects: (1) creation of the controlling class that will be interposed between the object and its original class, (2) definition of controlling methods in that class and (3) class change (see in 1.2). Controlling methods should have the same selectors as the original methods.

3.1 Explicit Subclassing

The interposed class is created by invoking the class creation definition method. Moreover, an original method can be invoked by the controlling method by use of the super variable.

The newly created class can be inserted using superclass: into the class hierarchy, so the subclasses can benefit from the control of the methods. To support a control of all the instances of the class, the

reference to the original class in the system dictionary class should be changed to refer the subclass.

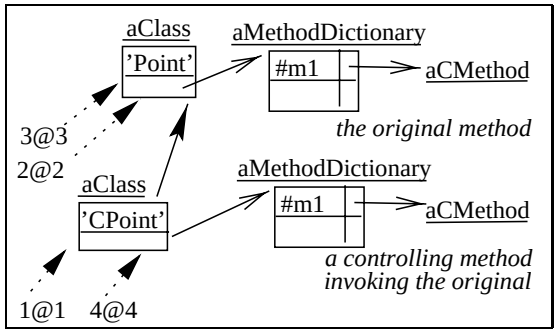


Figure 3: Explicit subclassing to control message passing. The CPoint class defines its own method dictionary containing controlled methods. Thus, 1@1 and 4@4 are controlled whereas 3@3 and 2@3 are not controlled.

Discussion. The control offered by this approach is a *group-based* or *class-based* control and possesses a *selective* granularity. Note that it could be possible to create as many classes as controlled instances but this would result in a proliferation of explicit classes.

The control is removed by another class change (see in 1.2). The execution cost is equal to the cost of a method execution. The main drawback of this solution is the creation of an explicit class, so this solution is not transparent from the point of view of the controlled objects.

3.2 Implicit Subclassing

Another solution is to interpose an anonymous class between the object and its class and to define controlling methods local to this specific object as shown by Fig. 4.

The following steps define the control installation:

1. Create an anonymous class, nCl, instance of Behavior[2] in VisualWorks or instance of Class in IBM Smalltalk.

2. Copy the class instance description (format) from the class to nCl and assign the inheritance link of nCl to the original class of the object.

3. Change the class of the instance to refer to nCl.

[2] According to McAffer, Peter Deutsch mentioned that the class Behavior had been originally designed to allow such implementations [McA95] p. 68.

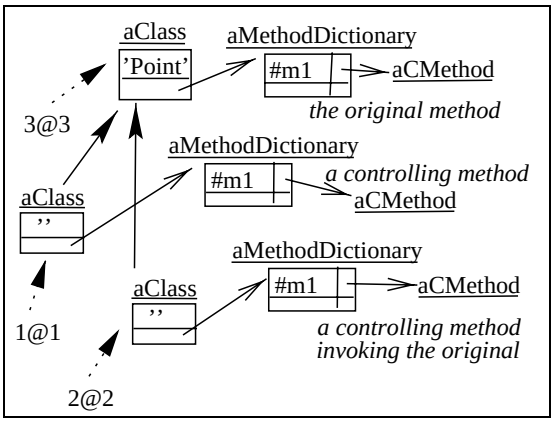


Figure 4: Implicit subclassing using anonymous classes to provide instance-based control message passing in VisualWorks.

4. Compile in nCl the methods that should be controlled.

VisualWorks Implementation. A possible installation of the control is illustrated in the following example method. The line number corresponds to the previous mentioned steps.

```
Object>>specialize
    |nCl|
(1) nCl := Behavior new
(2)         setInstanceFormat: self class format;
(2)         superclass: self class;
            methodDictionary: MethodDictionary new.
(3) self changeClassToThatOf: nCl basicNew
```

The fourth step is implemented by invoking the method compile:notifying: of the class Behavior with a string representing the controlling method. Such a method source code can be automatically generated. In the case of a control implementing before and after CLOS-like methods, the controlling method for the method named setX:setY: could look like:

```
anAnonymousClass>>setX: t1 setY: t2
... before
   super setX: t1 setY: t2
... after
```

IBM Smalltalk Implementation. Joseph Pelrine in [Pel96] describes a similar implementation:

```
Object>>specialize
    | nCl |
(1) nCl := Class new
(2)       superclass: self class;
(2)       instanceShape: self class instanceShape
(2)           instVarNames: self class instVarNames;
          setMethodDictionary: MethodDictionary new.
(3) self fixClassTo: class
```

Integration and semantics of class**.** A good integration into the programming environment redefines locally in the anonymous class the class method. Without that the control cannot be transparent: a user could ask for the *original* class and obtain the *anonymous* class. This method can be compiled on the anonymous class as shown in the following method. Note that an access to the anonymous class is also compiled. basicCompile: is a method that invokes in a protected manner the compile:notifying: method defined in superclasses of the original class.

```
AnonymousClass>>installEssentialMethods
  self basicCompile: 'class ^ super class superclass'.
  self basicCompile: 'isControlled ^ true'.
  self basicCompile: 'anonymousClass ^ super class'
```

Invocation of the original method. The original method could be invoked from within the controlling method defined in the anonymous class. An obvious solution is to directly invoke the method using the super variable. However such a solution is only possible if the control is done by the receiver via the anonymous class implementation and not by another object like in CodA [McA95] or in FLO [DBFP95].

A possible solution in that case is to define the call to the original method via a block ([super selector args]) that will be activated later by a value method. This solution is costly because this kind of block closure cannot be optimized by the compiler. Another solution is to refer to the compiled method instance in the controlling method using the same trick as in MethodWrapper (see 4.2) and invoke directly the method (valueWithReceiver:arguments:).

When the control is done by another object (like a meta-object), the following code can be automatically generated for the original method with selector setX:setY:. Here the meta-object defines a method control:call:withArgs: that effectively does the control.

```
anAnonymousClass>>setX: t1 setY: t2
  ^ self meta control: #setX:setY:
          call: [super setX: t1 setY: t2]
          withArgs: (Array with: t1 with: t2)
```

3.3 The VisualSmalltalk Solution

Contrary to VisualWorks and IBM Smalltalk, in which each object refers to its class that has a method dictionary, in VisualSmalltalk, each object refers to an array of method dictionaries. Such an array can be shared amongst all the instances of a class. Each method dictionary possesses an instance variable called class referring to the class to which it belongs as shown in 5. The method dictionaries are sorted from the class to its superclasses. This different implementation allows one to control message passing by using the VM method lookup [Bec93b, Bec93a, Pel96] as shown in fig. 6.

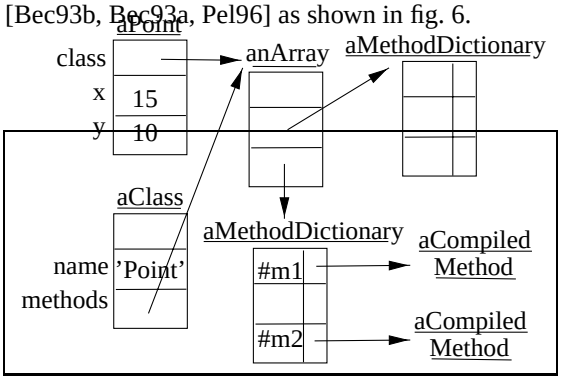


Figure 5: Relationship between instances, classes, method dictionaries and compiled methods in VisualSmalltalk: 15@10 an instance of Point does not refer to its class directly. It refers to an array of method dictionaries to which the class Point also refers to as method dictionary.

In VisualSmalltalk controlling a message sent to a specific instance is done by the following steps: (1) creation of a copy of method dictionary array of the object, (2) in the first place of this array addition of a new method dictionary and (3) definition of the controlling methods in this method dictionary.

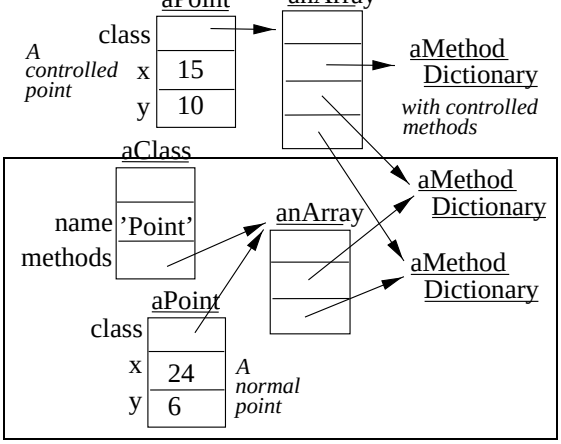


Figure 6: Instance specialization in VisualSmalltalk: 15@10 is controlled whereas 24@6 is not.

```
Object>>isSpecialized
  ^self methodDictionaryArray
   == self class methodDictionaries

Object>>specialize
  self isSpecialized ifTrue:[^self].
  self addBehavior: MethodDictionary new.

Object>>specialize: aString
  |assoc|
  self specialize.
  assoc := Compiler compile: aString in: self class.
  self methodDictionaryArray first add: assoc
```

The argument aString represents the source of a controlling method.

3.4 General Discussion

The technique based on anonymous classes is briefly mentioned in [FJ89], that qualified such classes as *lightweight* classes, and in CodA [McA95]. McAffer uses this technique to implement meta-objects and to control message passing. Ernest Micklei proposed a similar approach [Mic96]. However the meta-class is also controlled and his approach is more complex. NeoClasstalk uses this technique coupled with a method code change to implement dynamic specialization [Riv97] (see in 4.3).

These approaches support both *instance-based* control and *selective* control. Note that they can also support *class-based*, or *group-based* control by sharing the anonymous class amongst the controlled objects. Moreover, when all the instances of a given class have to share the same control, the method allInstances can be used to access to the instances of the original class.

These approaches are at the same time flexible and efficient as shown in 5.1. The lookup and execution of methods defined by the VM are used at their optimum. As the control is not based on method lookup failure, the cost is only one additional method execution. However, these techniques can only control methods that are known in advance to be controlled.

The implementation of these approaches is relatively simple and adaptable in the various dialects. However, an error during the installation can irreparably break the system. Indeed method dictionaries and format of the instances are crucial information for the VM. Moreover, method compilation is not necessary to install the control because the controlling methods can be copied and installed from predefined method skeletons (see 4.2). Therefore these techniques have a good installation speed and can be applied on deployed applications.

Finally, as a last important point, these methods do not raise the problem of object identity because the receiver of a controlled message is the object itself (see in 2.2).

4 Method Substitution

In Smalltalk, the methods defined in a class are stored in a method dictionary associated with the class. Such a dictionary associates each method selector (a symbol) with an instance of class CompiledMethod as shown in fig. 1.

As shown in figure 7, changing the compiled method associated with a selector supports message passing control. TRACER [BH90] and MethodWrappers [Bra96] use this technique. NeoClasstalk [Riv97] generalizes it. The original method can be simply stocked in the method dictionary associated with another symbol as in TRACER or it can be encapsulated in the controlling method like in MethodWrappers.

4.1 Hidden Methods

Another technique to control message passing is to associate a new selector (Xm1 in Fig. 7) with the original method and to associate a controlled method with the original method selector (m1) in the method dictionary. In case of before and after CLOS-like methods a controlling method could be schematically as:

```
aClass>>setX: t1 setY: t2
  ...before...
  self XsetX: t1 setY: t2
  ... after....
```

As compiled methods do not refer explicitly to their selector, it is not necessary to recompile the methods when they are associated with different selectors. Moreover, the installation of the controlled methods can be done by copying method skeletons

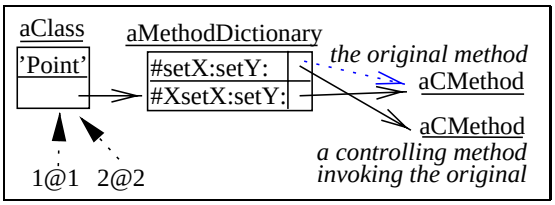


Figure 7: Addition of a new selector that refers to the controlled method and association.

and changing some method information: if we compare two controlling methods, the only difference is that they send different selectors to invoke their original methods. The selector that is used for such an invocation can be easily changed by replacing it in the method's literal frame. Therefore, to install a controlling method from a skeleton one only needs to change the selector, to set up the mclass instance variable to refer to the class (see 1.2) and to change the source code to refer to the source code of the original method.

4.2 MethodWrappers

The previous solution has the serious drawback of introducing new selector-method associations in the method dictionary and to polluting the interface of the controlled object class. Although it is unlikely that a user will invoke a hidden method, this solution is not good when inspecting the system. MethodWrappers is a clever approach that does not stock the original methods in the method dictionary of the controlled objects class but in the compiled methods themselves [Bra96]. Instead of creating a new association selector-compiled method, the original method is substituted by a method that encapsulates the original one – the wrapper has a reference to the original method as shown in Fig. 8.

4.2.1 Definition.

The following code describes the class MethodWrapper subclass of CompiledMethod. The instance variable clientMethod refers to the original method and selector represents the original method selector.

```
CompiledMethod variableSubclass #MethodWrapper
  instanceVariableNames: 'clientMethod selector'
  classVariableNames: ''
  poolDictionaries:''
  category: 'Method Wrappers'
```

As shown by the control of the method color of the class Point below, the class method on:inClass: returns a wrapped method that can further be installed on a compiled method by invoking the method install.

```
(MethodWrapper on: #color inClass: Point) install
```

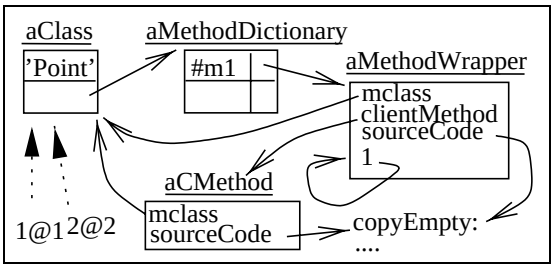


Figure 8: After installation: the original method is encapsulated into a method wrapper.

MethodWrapper class also specializes the method valueWithReceiver:arguments to introduce message passing control as follows. Note that in such a case the control is limited to before and after method executions implemented by helper method beforeMethod and afterMethod.

```
WrapperMethod>>valueWithReceiver: anObject arguments: args
  self beforeMethod.
  ^ [clientMethod valueWithReceiver: object
                arguments: args]
    valueNowOrOnUnwindDo: [self afterMethod]
```

A controlling method definition ensures that the method valueWithReceiver:arguments: is called. The following method source, that is automatically generated, shows how the arguments are managed.

```
aClass>>originalSelector: t1
 | t2 |
(t2 := Array new: 1) at: 1 put: t1.
^#() valueWithReceiver: self arguments: t2
```

When a message is sent to an object, it is necessary to invoke certain methods on the method wrapper itself (like valueWithReceiver:arguments in the previous code). But Smalltalk does not offer a pseudo-variable to refer to the current invoked method. Instead of using the thisContext pseudo-variable that costly reifies the method execution context, the author of MethodWrappers modifies the literals of the method wrapper. He uses the #() literal object in the previous code to reserve place to put a reference during the installation to the method wrapper itself. Note that using the self pseudo-variable in the source code of the prototype shown above was

not the right solution because self represents the object on which the method was invoked and not the method itself.

As in the hidden method approach, MethodWrappers do not need to be compiled to be installed. The controlling method can be copied from a method skeleton having the same number of arguments. Then, the mclass instance variable, the literal and the clientMethod should be set. Moreover, to be fully and transparently integrated in the Smalltalk environment, the source code of the controlling method references the source code of controlled one as shown in Fig. 8.

4.3 NeoClasstalk

NeoClasstalk is a new implementation of Smalltalk that introduces explicit meta-classes [Riv97]. NeoClasstalk allows the definition of class properties such as method trace, instance variable access trace and pre- and post- conditions. These properties are based on a *controlled* modification of method source code. It proposes a framework for the composition of the different control policies. A metaprogrammer can specify a part of the method source code that will be automatically compiled in the controlled methods.

The NeoClasstalk implementation uses similar techniques to MethodWrapper (prototype and literal modification) but gives the control to the class. Moreover, NeoClasstalk uses a dynamic change of class based on the definition of anonymous classes (as shown in 3.2).

Control Definition. In NeoClasstalk the execution of a method is invoked by the method execute:receiver:arguments: defined on the class AbstractClass. The *definition* (source code) of this method is defined by the method generateBodyOn: of the class TemporalComposition.

Let us suppose that we want to define a message passing control that realizes a trace of the invoked methods. To do so, we define a new class TraceAllMessages (subclass of TemporalComposition) and we specialize the method generateApplyBodyOn: that controls a part of the method source code generation of execute:receiver:arguments:. The following code shows the addition of the textual definition (source code part) of a trace to the normal method definition. The last line ensures that the normal behavior of the method will be added in this definition.

```
TraceAllMessages>>generateApplyBodyOn: aStream
  aStream nextPutAll: ' | window |
    window := self transcript.
              cm printNameOn: window.
              window cr; endEntry.'.
  super generatedApplyBodyOn: aStream
```

To control the class Point one should invoke the temporalComposition: method as follows:.

```
TraceAllMessages new temporalComposition: Point.
```

TraceAllMessages new creates an implicit class with method wrappers. temporalComposition: Point changes the class of the class Point so that it will be instance of the class TraceAllMessages.

A part of the Framework. As shown below, the method applyMethod defined on the class TemporalComposition specifies the definition of the source code of the method execute:receiver:arguments:. A part of this definition is under the responsibility of the method generateApplyBodyOn:. The method applyMethod ensures a semantic context of the generated method such as the insurance that the original method will be invoked (as shown by the message super execute:... below).

```
TemporalComposition>>applyMethod
"rec is the receiver, args are the arguments
 of the method, cm is the currently reified method"

| ws |
ws := (String new: 100) writeStream.
ws nextPutAll: 'execute: cm receiver: rec arguments: args';crtab;
  nextPutAll: "system generated method";cr;crtab.
self generateBodyOn:ws. "<- the method to override"
^ ws contents
```

```
TemporalComposition>>generateApplyBodyOn: aStr

  aStr crtab;
      nextPutAll: '^ super execute: cm receiver: rec arguments: args'
```

Note that by changing the method generateApplyBodyOn: it is also possible to change the complete semantics of the control.

4.4 Discussion

These techniques possess a *class-based* control and a *selective* granularity. Indeed all the instances of a class are controlled without the ability to select

them. The control execution cost is the cost of a method execution.

The first solution based on the definition of new association selector/method in the method dictionary polluted the interface of the objects. This problem does not appear with the other approaches. NeoClasstalk takes in charge the recompilation of the methods and proposes a well defined context for the definition and the composition of the method control. However, its solution is complex, and this complexity is not due to the concepts used as the automatic recompilation, but by the framework definition based on explicit meta-classes[3]. Contrary to the other approaches the reproduction of the mechanism is difficult.

Finally, contrary to the approach based on identity change, the main advantage of message passing control by means of anonymous classes (see in 3.2) or method wrappers (MethodWrappers and NeoClasstalk) is that the tools defined in the browsers such as (implementors, senders...) continue to fully function.

5 Summary and Conclusion

Before presenting how other object-oriented languages support message passing control, we summarize and compare the techniques.

5.1 Overview

The following table gives a quick overview of the presented techniques in terms of the criteria defined in 1.4. We present here only the main or default characterics. For a deeper analysis, the reader should refer to the previous discussions. The *entity* column refers to the granularity of the control that states which entities can be controlled, the *message* criteria shows if all or some messages can be controlled, the last criteria establishes if the solution is well integrated in the Smalltalk environment in terms of browser functionality (senders and implementors) and transparency from the user point of view.

[3]Note that NeoClasstalk proposes tools for selecting class properties that simplifies the life of the lambda programmer.

Technique	*entity*	*message*	*integration*
Error handling	instance-based	global	average
Explicit Subclassing	group-based	selective	average
Anonymous Class	instance-based	selective	good
Hidden Methods	class-based	selective	bad
Method Wrapper	class-based	selective	good
NeoClasstalk	class-based	class	good

The next table compares the different approaches for the runtime overhead. These tests were performed on a Power Mac 7100/166 with 24MB memory using Visualworks2.5. The results are the mean over five series of 10000 calls with 0,1,2 and 3 arguments. Moreover, during our numerous tests such results show some variability, therefore we consider that a difference up to 10 milliseconds is not really significant.

Technique	0	1	2	3
Explicit Subclassing	40.0	40.0	46.6	39.8
Anonymous Class	40.0	40.2	43.2	43.2
Hidden Methods	40.0	43.2	43.2	43.4
Method Wrapper	200	233	243.4	250
Inlined Method Wrapper	100	126	140	153
Error handling	213.4	229	233.4	240

As we can expect, the comparison shows that the techniques based on the explicit and implicit subclassing (anonymous classes) have the same overhead. Moreover, these two techniques have the same overhead than the technique based on hidden methods. It shows that the lookup of the method via super in the two first approaches is equivalent to the lookup via self in the hidden method approach. This is not surprising in presence of method cache mechanisms performed by the Virtual Machine. This comparison shows that the technique based on error handling is five times slower. The method wrapper approach has the same cost. This situation comes from the fact that method wrappers must create arrays for their arguments and that in our tests we do not remove the call of the valueNowOrOnUnwindDo: method.

As an experiment, we change the Method Wrapper's implementation, the controlling method continued to call the method valueWithReceiver:arguments: but we remove the call to the method valueNowOrOnUnwindDo:. The results,

named Inlined Method Wrapper are two times faster than the normal Method Wrapper. Moreover, this approach could be optimized by inlining in the call inside the controlling method body instead of calling the method valueWithReceiver:arguments: defined on the class MethodWrapper.

5.2 Message Passing Control in Other Languages.

CLOS is the object system integrated into Common Lisp. It is one of the few class based languages to offer the ability to define instance specific methods using the *eql specializer*[Kee89]. Moreover CLOS is also one of the rare languages to provide a meta object protocol (MOP) in which message passing control is an entry point [KdRB91].

In CLOS the message passing concept is replaced by the generic function[4] The CLOS MOP allows one to control all the aspects of the generic function application: the application of the generic function (compute-discriminatingfunction), the application of the effective method (compute-effective-method-function) or the application of a single method composing the effective method (compute-method-function).

In the prototype based languages, Moostrap allows a message passing control based on the definition of a reflective protocol: object meta-object is responsible for the method lookup and application [MC93].

In the realm of less flexible languages, the definition of OpenC++ -that can be perceived in its last version as an open compiler [Chi95] - shows the interest for a control of message passing. More recently, the definition of MetaJava offers the ability to control message passing in Java [Gol97]. In this implementation anonymous classes called *shadow classes* are interposed between the instance and its original class (see in 3.2). However, in the new version called MetaXa, the interpreter is extended by the introduction of new byte-codes. As a direct consequence MetaXa's applications are no longer portable.

Java in its newest version 1.1 reified certain aspects of the language such as the classes, the methods and the instance variables (see Core Reflection API [Fla97]). However, this reification is *only* introspective reflection. Indeed, the classes Field, Method and Constructor are declared as final. This implies that they cannot be specialized. Moreover, only the Java VM can create new instances of these classes. Only the value of the instance variables can be modified and the methods can be invoked using the handle() method. Such an approach was necessary to offer tools comparable to the Smalltalk browsers in Java. However, this reification is not causally connected to the language. There is no possibility to modify the methods or the classes from within the language itself. This means the reflective facilities are not really adapted to extend or modify the language.

5.3 Conclusion

This comparison highlights that the most commonly used technique based on the specialization of the doesNotUnderstand: method is not the best one. As a first explanation of this situation, one should note that the ability to directly execute a method has only lately been introduced in the interpreters (methods valueWithReceiver:arguments: on CompiledMethod class in VisualWorks and executeWithReceiver:andArguments: in IBM Smalltalk). Moreover, this comparison shows that the techniques based on VM lookup method or method wrappers should be considered by more programmers than it is currently the case.

The reflective aspects of Smalltalk and their causal connection to the language itself offer strong advantages for the language extensions or modifications [5]. We illustrate them by showing how message passing control is possible by different approaches. This study shows the power offered by languages like CLOS or Smalltalk that provide reflective facilities that are not limited to introspective reflection like in the new version of Java (1.1).

Acknowledgments. The author would like to thank J. Brant, P. Cointe, M. Fornarino, J. McAffer, E. Micklei, O. Nierstrasz, F. Pachet, J. Pelrine, M. Rieger and F. Rivard.

References

[ABV92] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *ECOOP'92, LNCS 615*, pp 372–395, 1992.

[Bec93a] K. Beck. Instance specific behavior: Digitalk implementation and the deep meaning of it all. *Smalltalk Report*, 2(7), May 1993.

[4] A generic function is a group of methods. During the application of a generic function, methods from that group are selected to constitute an effective method application. This is the effective method that is executed.

[5] A reflective aspect of a language is said causal if any change in the reified aspect immediately influences the represented aspect and conversely.

[Bec93b] K. Beck. Instance specific behavior: How and why. *Smalltalk Report*, 2(6), Mar 1993.

[Bec95] K. Beck. A modest meta proposal. *Smalltalk Report*, July/August 1995.

[Ben87] J. K. Bennett. The Design and Implementation of Distributed Smalltalk. In *OOPSLA'87*, pp 318–330, 1987.

[BH90] H.-D. Böcker and J. Herczeg. What tracers are made of. In *OOPSLA/ECOOP'90*, pp 89–99, 1990.

[BI82] A. H. Borning and D. H. Ingalls. Mutiple Inheritance in Smalltalk-80. In *Proc. of NCAI AAAI*, pp 234–237, 1982.

[Bra96] J. Brant. Method Wrappers. http://st-www.cs.uiuc.edu/users/brant/Applications/MethodWrappers.html, 1996.

[Bri89] J. Briot. Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. In *ECOOP'89*, pp 109–129, 1989.

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *OOPSLA'95*, pp 285–299, 1995.

[Coi90] P. Cointe. The ClassTalk System: A Laboratory to Study Reflection in Smalltalk. In *OOPSLA/ECOOP'90 Workshop on Reflection and Metalevel Architectures*, 1990.

[DBFP95] S. Ducasse, M. Blay-Fornarino, and A. Pinna. A Reflective Model for First Class Dependencies. In *OOPSLA'95*, pp 265–280, 1995.

[Duc97] S. Ducasse. Des techniques de contrôle de l'envoi de message en smalltalk. *L'Objet*, 3(4), 1997. Numero Special Smalltalk.

[Fer89] J. Ferber. Computational reflection in class based object oriented languages. In *OOPSLA'89*, pp 317–326, 1989.

[FJ89] B. Foote and R. E. Johnson. Reflective facilities in Smalltalk-80. In *OOPSLA'89*, pp 327–336, 1989.

[Fla97] D. Flanagan. *Java in a Nutshell*. O'Reilly, 2nd edition, 1997.

[GGM95] B. Garbinato, R. Guerraoui, and K. Mazouni. Implementation of the GARF replicated objects plateform. *Distributed Systems Engineering Journal*, Mar. 1995.

[Gol97] M. Golm. Design and Implementation of a Meta Architecture for Java. Master's thesis, IMMD at F.A. University, Erlangen-Nuernberg, 1997.

[GR89] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its implementation*. Addison-Wesley, 1989. ISBN: 0-201-13688-0.

[Hop94] T. Hopkins. Instance-Based Programming in Smalltalk. Esug Tutorial, 1994.

[IKM+97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In *OOPSLA '97*, 1997.

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kee89] S. E. Keene. *Object-Oriented Programming in Common-Lisp*. Addison-Wesley, 1989.

[Lal90] W. Lalonde. *Inside Smalltalk (volume two)*. Prentice Hall, 1990.

[LG88] W. R. LaLonde and M. V. Gulik. Building a Backtracking Facility in Smalltalk Without Kernel Support. In *Proceedings of OOPSLA'88*, pp 105–122, 1988.

[MB85] S. L. Messick and K. Beck. Active Variables in Smalltalk-80. Cr-85-09, Tektronix, Computer Research Lab., 1985.

[MC93] P. Mulet and P. Cointe. Definition of a reflective kernel for a prototype-based langage. In *ISOTAS'93, LNCS 742*, pp 128–144, 1993.

[McA95] J. McAffer. *A Meta-Level Architecture for Prototyping Object Systems*. PhD thesis, University of Tokyo, 1995.

[McC87] P. L. McCullough. Transparent Forwarding: First steps. In *OOPSLA'87*, pp 331–341, 1987.

[Mic96] E. Micklei. Spying messages to objects. Esug Tutorial, 1996.

[Pas86] G. A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *OOPSLA'86*, pp 341–346, 1986.

[Pel96] J. Pelrine. Meta-level programming in smalltalk. Esug Tutorial, 1996.

[PWG93] F. Pachet, F. Wolinski, and S. Giroux. Spying as an Object-Oriented Programming Paradigm. In *TOOLS EUROPE'93*, pp 109–118, 1993.

[Riv96] F. Rivard. Smalltalk : a Reflective Language. In *REFLECTION'96*, pp 21–38, 1996.

[Riv97] F. Rivard. Evolution du comportement des objets dans les langages à classes réflexifs, 1997. Ecole des Mines de Nantes, Thèse de l'Université de Nantes.

[Sch96] B. Schaeffer. Smalltalk: Elegance and Efficiency. Ecoop Tutorial, 1996.

[YT87] Y. Yokote and M. Tokoro. Experience and Evolution of Concurrent Smalltalk. In *OOPSLA'87*, pp 406–415, 1987.

Designing an Extensible Distributed Language with a Meta-Level Architecture

Shigeru Chiba* Takashi Masuda
Department of Information Science, The University of Tokyo
E-mail: {chiba,masuda}@is.s.u-tokyo.ac.jp

In Proceedings of 7th European Conference on Object-Oriented Programming (ECOOP'93), Kaiserslautern, July 1993, LNCS 707, pp.482–501

Abstract

This paper presents a methodology for designing extensible languages for distributed computing. As a sample product of this methodology, which is based on a meta-level (or reflective) technique, this paper describes a variant of C++ called *Open C++*, in which the programmer can alter the implementation of method calls to obtain new language functionalities suitable for the programmer's applications. This paper also presents a framework called *Object Communities*, which is used to help obtain various functionalities for distributed computing on top of Open C++. Because the overhead due to the meta level computation is negligible in distributed computing, this methodology is applicable to practical programming.

1 Introduction

Languages for distributed computing have been designed mostly to provide a general functionality that can be used in a broad range of application domains. Designers of these languages have developed numerous language primitives or functionalities, such as Ada's rendezvous [26], the remote procedure call [2], and Orca's shared data-object [1]. Each of these functionalities has its own most suitable domain of applications, so a language that has a single one of these functionalities will be small and simple but will not be suitable for some applications. It is, on the other hand, possible to design a language that has many or all such functionalities, but such a language would be large and awkward.

The goal of this paper is to demonstrate another approach, which is to make a language extensible. By this approach, we have been able to design a language that is, at the same time, simple, elegant, and applicable in a wide range of domains. A programmer can tailor the language to exploit various functionalities. Language extensibility has long been an important issue, and Kiczales et al., for example, have recently discussed the designing of extensible class libraries [11]. A typical approach to supporting various functionalities within a single language is to provide a set of reusable code, called a library program, that implements functionalities that are not supported

*JSPS (Japan Society for the Promotion of Science) Fellow-DC

by the language alone. Although functionalities implemented by this approach may show lower performance than ones implemented by altering the language system such as the compiler, this approach is broadly employed because sufficient performance is usually obtained by this approach in practice. The library-program approach, however, is limited in that it cannot implement a functionality that deals with non-first-class entities of the language.

This paper proposes methodology using an object-oriented meta-level technique in designing of an extensible language for distributed computing. To demonstrate the use of this methodology, we present *Open C++*, which is a C++ [23] variant including a simple *metaobject protocol* (MOP) [10]. In Open C++ the implementation of a method call (or in the object-oriented terminology, message passing) is made open-ended by that MOP. To obtain a new functionality that fits the application, the programmer can easily extend the implementation within Open C++ itself. Performance overheads are one of major issues in meta-level techniques, but they are not critical in domains such as distributed computing, which Open C++ deals with. The seriousness of the overheads depends on the inherent cost of functionalities achieved with the meta-level technique. Since the overhead of Open C++ is negligible in comparison with the implemented functionalities, we believe that our approach is — like the library-program approach, which is useful in spite of its relative slowness — applicable to actual problems.

As with other systems using meta-level techniques, an extension of Open C++ is described in meta code (meta-level program). Although meta code is usually written only by a system specialist because MOP would be often complicated and extension was not frequent, we expect normal programmers (who are not "wizards") to write meta code in Open C++ whenever a new functionality is required for their applications. The Open C++ MOP is therefore designed to provide an abstraction that encapsulates implementation details unnecessary to the extension of a method call. To facilitate extension by normal programmers, this paper also provides a framework, called *Object Communities*, that includes some basic functionalities for extending a method call for distributed computing. With this framework, normal programmers can easily obtain various functionalities for distributed computing on top of Open C++.

2 Open C++: A Simple MOP for C++

In most imperative languages for distributed computing, procedure calls (or in the object-oriented terminology, method calls) are extended to support remote communication across a network. Those extended method calls provide not only a functionality invoking a procedure (or a method) at a remote machine, but also a functionality synchronizing multiple threads of control. In Ada [26] and Concurrent C [5], for example, a statement syntactically similar to a procedure call is used for executing a rendezvous, and a procedure call is extended to block the sender thread until the receiver is ready. In ConcurrentSmalltalk [28], a method call of Smalltalk-80 [6] is extended to be synchronous or asynchronous: an *asynchronous method call* lets a sender thread continue its execution without blocking, whereas a *synchronous method call* blocks the sender thread until the receiver thread finishes a requested task.

By using a meta-level or so-called reflection technique [21], Open C++ offers normal programmers the ability to extend a method call. Normal programmers can modify the implementation of a method call within a user program to obtain various functionalities for remote communication.

The implementation of a method call is exposed to programmers as a *metaobject* [15], which is an abstract model of that implementation and conceals implementation details unnecessary to the extension. A metaobject is almost the same as a normal object, but its behavior corresponds to the actual execution of the method call. An object at the *base level* has its metaobject at the *meta level*, and the execution of its methods is controlled by the metaobject. If a method of the object is invoked, the specific method of the metaobject, instead of a default implementation embedded in the compiler, is used to execute the invoked method. Since a metaobject is defined in C++, the programmer can alter the implementation of a method call by defining another metaobject and then substituting it. Our approach does not require rebuilding the compiler but is done within a user program.

2.1 Base-Level Directives

Open C++ provides a very simple MOP (metaobject protocol*) to make a method call extensible. The objects controlled by metaobjects are called *reflective objects*. Because control by a metaobject imposes some performance and memory overhead in Open C++, the programmer can specify whether or not an object is reflective. A nonreflective object is compiled to be a normal C++ object, which has no metaobject, so that it is executed without overhead. To distinguish between reflective and nonreflective objects, a reflective object is identified by a different class name. If the class of an object that may be reflective is `X`, then a reflective object is `refl_X` and a nonreflective object is still `X`. In the current implementation, the class `refl_X` is a subclass of `X`.

To create a reflective object, the class of the object and its metaobject must be declared with special directives, which are C++ comments that start with `//MOP`. Note that even if a program includes the directives of Open C++, that program is still a valid C++ program. The declaration of a reflective object takes the form

```
//MOP reflect class X : M;
```

This declaration means that an object of the class `refl_X` is a reflective object controlled by a metaobject of the class `M`. Note that it never means that the classes `X` or `refl_X` are subclasses of `M`. The class `M` is a normal C++ class except that it must inherit from the class `MetaObj`. To extend its implementation of a method call, a metaobject can be a reflective object that is controlled by a meta-metaobject. Open C++ allows such an ascending tower of metaobjects.

The methods of a reflective object are divided into two groups, depending on whether the invocation of the method is controlled by its metaobject. The methods controlled by the metaobject are called `reflect` methods, and although `reflect` methods are invoked in an extended manner, the other methods are invoked according to the plain C++ method call semantics. The following is an example of specifying a `reflect` method.

```
class X {
public:
    X();
//MOP reflect:
    int func(int);
```

*A metaobject protocol is a meta-protocol organized using object-oriented techniques. Here a meta-protocol is a protocol about the behavior and implementation of another protocol, such as interface and functionality.

```
private:
    int p;
};
```

The methods following the directive "`//MOP reflect:`" are specified as `reflect` methods. Here, for example, `func()` is a `reflect` method. Such methods may have a category name to enable their metaobject to recognize a role of the methods. A metaobject may alter the execution of a method call according to the category name. Consider the following example: The method `update()` has a category name "`write`".

```
class Y {
public:
    ...
//MOP reflect(write):
    int update(int);

//MOP reflect(metamethod):
    void Meta_operation();
    ...
```

The category name "`metamethod`" has a special meaning: it is used to call meta-methods of a metaobject from the base level across the boundary of the levels. Calling a `reflect` method in this category is regarded as calling a meta-method that has the same *method* name. The `reflect` methods having the category name "`metamethod`" themselves are never executed.

2.2 Metaobject Protocol

When a `reflect` method is called, its execution is controlled by its metaobject. A metaobject is defined in C++, and its class must inherit from the base class `MetaObj`, which mainly defines the following two methods.

- `void Meta_MethodCall(Id method, Id category, ArgPac& args,`
  `ArgPac& reply);`

  This method implements a method call at the base level. It is invoked if a `reflect` method is called.

- `void Meta_HandleMethodCall(Id method, ArgPac& args,`
  `ArgPac& reply);`

  This method is used to actually execute a `reflect` method.

To alter the implementation of a method call, the programmer defines a subclass of `MetaObj` in which those methods are redefined so that the metaobject acts in the intended way.

Suppose that a `reflect` method `f()` is called. If the method `f()` is called, then the method `Meta_MethodCall()` is instead invoked at the meta level. The first argument of the method `Meta_MethodCall()` is bound to the integer identifier of the called method `f()` (the type `Id` represents integers), and the second argument represents the category name of the method `f()`. The actual arguments of the method call to `f()` are passed as the third argument, `args`. Note that within a metaobject, the actual argument list of a method call is a first-class entity because the

third argument, `args`, is a normal C++ object whose class is `ArgPac`. The argument `args` has the same interface as a stack so that the programmer can access any actual argument stored in `args`. The programmer can also transfer the argument `args` to another metaobject that may reside on a different machine. Converting the actual arguments to an `ArgPac`-class object corresponds to the *reifying* process, which is impossible in C++ alone without support of the Open C++ compiler.

The method `Meta_MethodCall()` carries out certain computation and stores the result into the fourth argument, `reply`. The stored result is returned as a return value to the caller that calls the `reflect` method `f()`. The method `Meta_MethodCall()` usually uses the method `Meta_HandleMethodCall()` to compute the result value. This method takes a method identifier and an actual argument list, and it returns the result value of the specified method. This method allows any `reflect` method to be executed at any time. In the example above, the metaobject can execute another `reflect` method as well as `f()` to compute the result value.

To illustrate the Open C++ MOP, consider a simple example in which this metaobject prints a message before executing a `reflect` method called at the base level:

```
class VerboseMetaObj : public MetaObj {
public:
    void Meta_MethodCall(Id method, Id category,
                         ArgPac& args, ArgPac& reply){
        printf("***reflect method %s() was called.\n",
               Meta_GetMethodName(method));
        Meta_HandleMethodCall(method, args, reply);
    };
};
```

If a metaobject of the class `VerboseMetaObj` is specified, a message is printed on the console every time a `reflect` method is called. The method `Meta_MethodCall()` specifies that this metaobject prints the name of the called method before actually executing that method. Note that if we eliminate the line "`printf(...);`" from this method, the implementation of a method call by this metaobject becomes the same as the implementation in plain C++. Figure 1 shows how a metaobject of the class `VerboseMetaObj` controls a method call. The metaobject controls an object of the class `refl_X` (as previously shown, a reflective object of the class `X`). When a `reflect` method `func()` of that object is called, the metaobject traps that method call and executes the method `func()` according to the method `Meta_MethodCall()`.

Converting the actual arguments to an `ArgPac`-class object is similar to the marshaling/unmarshaling process in remote procedure calls. In the current implementation, the class `refl_X` (which the Open C++ compiler generates) redefines a `reflect` method so that the method carries out such conversion and then invokes the method `Meta_MethodCall()` of its metaobject. The current Open C++ compiler converts some atomic types (integers, pointers, etc.) implicitly but does not class types (i.e., objects). The class types that can be an argument of a `reflect` method must have some specific methods for the conversion. A similar limitation also appears in the marshaling/unmarshaling process because the efficiency of converting complex data, such as an object, often depends on the program semantics. Such conversion should be under programmer's control [8]. Open C++, however, provides a convenient library to implement the methods for the conversion, and it also provides some predefined classes that facilitate to use a character string etc. as an argument of a `reflect` method. Thereby, the limitation on argument types of `reflect`

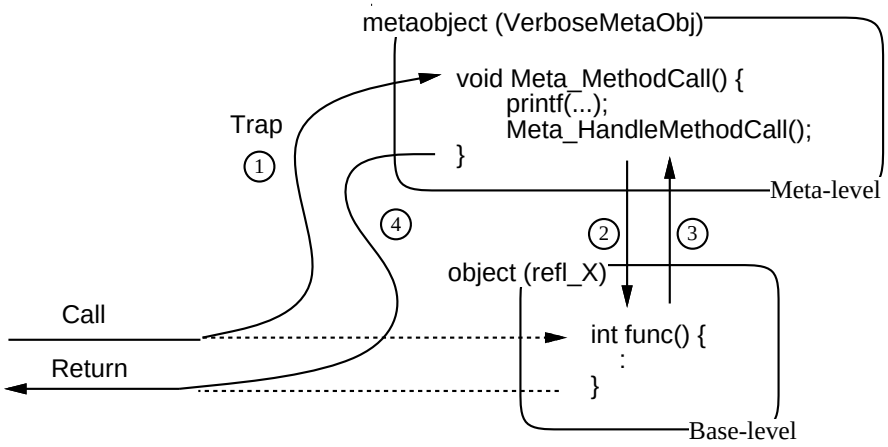


Figure 1: Metaobject protocol of Open C++

methods is not awkward.

2.3 Why Meta? Pros and Cons

Open C++ does not expose the implementation of a method call directly, but through an abstract interface. Although the original implementation of a method call, which is embedded in the compiler, is described in assembly code, the programmer who attempts to extend the method call describes a new implementation of C++ methods such as `Meta_MethodCall()` instead of assembly code. Because of the description through the abstract interface, the programmer need not consider such details of the implementation as a stack image and the number of arguments. The programmer can thus concentrate on matters strongly relevant to the extension.

This feature of Open C++ is due to the meta-level technique that Open C++ uses. When a method of an object is invoked, the computation of the method call is *reified* to be entities available in a C++ program, and operations on these entities are *reflected* in the actual computation. This is a difference from "pseudo-open" systems, which directly expose their internal structure to be extensible. Smalltalk-80, for example, provides the whole source code of its runtime system. Thus in a sense, it is an open-ended system because user programmers can freely modify classes of kernel objects to extend the system behavior. This feature of Smalltalk-80 may be a kind of reflection[†]. Such modification of kernel objects, however, can easily lead the system into collapse because the programmer deals with the complicated kernel code directly, without an abstract interface.

On the other hand, the reifying process implies that the performance of Open C++ degenerates. The cost of reifying and reflecting is not negligible compared with the original implementation fully described in assembly code. This is because the reifying process bridges the wide gap between the assembly level and the C++ level. The higher the abstract interface Open C++ provides for extension, the bigger the performance degeneration of the reifying process will be. This degenera-

[†]Peter Deutsch pointed this out at the BOF session in the '92 workshop on reflection and meta-level architecture.

tion is negligible, however, when Open C++ is used for distributed computing. The method call extended for distributed computing is so slow that the performance degeneration becomes relatively insignificant. This issue is discussed in detail in Section 5.

Another benefit of Open C++ is that meta code defines the extension independently of each object so that meta code has high reusability. The same meta code can be used to extend method calls to different objects. Because meta code is organized according to the metaobject protocol, furthermore, part of it is also reusable by class inheritance.

Open C++ improves the expressive power of a class library, which is also a technique for supporting various functionalities within a single language. If a functionality like remote method calls is implemented solely by means of class libraries, the translation of an argument list into a network message becomes responsibility to the programmer. This is because the class library alone cannot deal with any entities except these available at the base level, and an argument list is available not at the base level but at the meta level. On the other hand, Open C++ enables a class library to deal with an entity available at the meta level through a metaobject. For example, it can use a metaobject for transferring an argument list to a different machine and can execute a remote method.

3 Object Communities — An Additional MOP for Distributed Computing

Because a method call is a good basis of functionalities for distributed computing, various functionalities can be implemented on top of Open C++. Most imperative languages include a method call statement, and it has been used to implement a lot of existing functionalities for distributed computing. A method call can be extended to support not only a remote method call but also asynchronous message passing and message broadcasting. It can also be extended to be a synchronization mechanism such as a rendezvous or a distributed semaphore.

To obtain a functionality suitable for the application, normal programmers should themselves describe meta code to extend a method call. Although previous systems usually expected meta code to be written only by a specialist, the simple MOP of Open C++ makes meta programming possible for programmers with little knowledge as well as for specialists. The MOP of Open C++, however, does not in itself support distributed computing; it only provides a platform on top of which a functionality for distributed computing is implemented. This section proposes a framework, called *Object Communities*, that facilitates to implement such a functionality on top of Open C++. This framework is a class library of metaobjects and includes facilities that are commonly used to extend a method call. *Object Communities* add a layered protocol onto the MOP of Open C++. It provides the classes of metaobjects that implement some typical functionalities for distributed computing so that programmers can obtain functionalities tailored to their applications by redefining some methods of those classes.

3.1 Background Problem

Object Communities are designed to be a framework for implementing various application-specific functionalities for distributed computing, such as distributed shared data, distributed transactions,

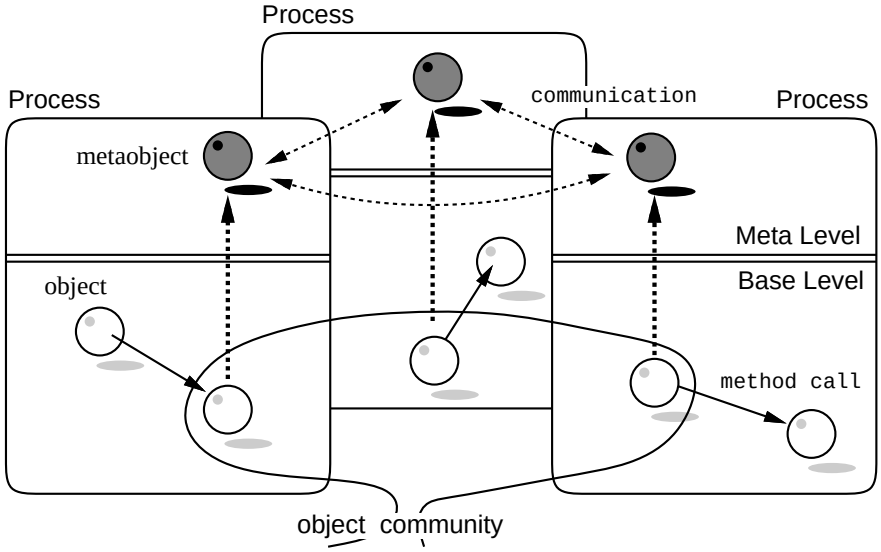


Figure 2: An object community

remote procedure calls. Such a framework must provide a facility managing computation distributed to multiple processes on different machines. A simple client-server framework based on remote procedure calls is not sufficient as such a framework because although it can request computation to another process, it cannot synchronize computation between processes.

The simple client-server framework, for example, cannot in an easily understandable way implement the functionality required by groupware[4] (or multiuser applications), which supports collaborative work by multiple users. The essential feature of groupware is that an application program consists of multiple *autonomous* processes that are responsible for interaction with each user. Those processes interfere with each other because the users manipulate shared entities, such as shared documents and pictures, and their actions are therefore restricted by the actions of other users. The processes may also notify each other when shared entities are updated and they can request computation, such as redrawing the displays, in order to keep consistent images of the entities on the displays. To do these things, the application needs a functionality that makes it possible to block the execution of other processes as well as to request computation to other processes.

3.2 Overview of Object Communities

The fundamental functionality of *Object Communities* is the management of a group of objects distributed in different machines. Such a group is called *an object community* (Figure 2). We assume that each object belongs to a single process that has its own address space separated from others and communicates with other processes across a network. A process is invoked explicitly by the user, and it performs cooperatively with other processes in the same application.

Each object of an object community acts in a manner that depends on behavior of other

objects of that object community. The method calls to the objects are executed cooperatively by the metaobjects so that the objects provide a certain functionality for distributed computing. Note that although a group of objects as a whole provides some functionality, the definition of the objects does not include any distributed concepts: these appear only in the definition of the metaobjects. The functionality provided is implemented at the meta level, and the base-level programmer has only to know how a method call is extended at the base level. *Object Communities* provide a clear separation between distributed computation and the substantial computation executed in the application.

In *Object Communities*, a metaobject has the following additional abilities.

- *Concurrency Control*. A metaobject controls the internal concurrency of its object. It can ignore and delay execution of a called method of the object until some condition is satisfied. A metaobject can also execute multiple methods of its object concurrently. And a metaobject can execute a method of its object when other metaobjects request that the method be executed.

- *Communication*. A metaobject has two means of communicating with other metaobjects of the same object community: a remote method call and message broadcasting. A metaobject can call a remote method of other metaobjects. This is done in a manner similar to that of a local method call. The caller metaobject is blocked until a reply is returned from the called object. A metaobject can also send a message to all metaobjects of the same object community. Because broadcast messages are serialized by the underlying system, all metaobjects receive the messages in the same order. A broadcast message is also delivered to the metaobject that sent the message.

Although a metaobject controls the internal concurrency of its object, there is with few exceptions no internal concurrency of the metaobject by default. The methods of a metaobject are executed sequentially, so the behavior of a metaobject is easily understandable. If internal concurrency of a metaobject is necessary, it must be controlled by an explicitly specified meta-metaobject.

3.3 MOP of Object Communities

To append the *Concurrency Control* and *Communication* abilities, *Object Communities* provide the class `OcCoreMetaObj`, which is a subclass of `MetaObj`, and the other classes of metaobjects that implement functionalities based on *Object Communities* must inherit from this subclass.

The class `OcCoreMetaObj` defines some methods for manipulating an object community, for network communication, for controlling concurrency, and so on. The following methods are to manipulate an object community.

- `Meta_CreateOc(...)` creates an object community.
- `Meta_DestroyOc(...)` destroys an existing object community.
- `Meta_Join(...)` lets an object join a specified object community.
- `Meta_Leave(...)` lets an object leave a specified object community.

An object community is treated if it were a communication channel. An object can join or leave an object community at any time, but the object community remains even if no object belongs to it. It exists until it is destroyed explicitly. To give initial information to a metaobject that joins an object community, the underlying system holds an initializing message for each object community. This message, which can be dynamically updated by a metaobject, is passed to a metaobject when its object joins to an object community.

The class `OcCoreMetaObj` defines three methods for communication with other metaobjects.

- `Meta_EventNotify(...)` broadcasts a message to the other metaobjects of the same object community.
- `Meta_Query(...)` calls a method of other metaobjects in a manner like that of the remote procedure call. The metaobject is blocked until a reply message arrives.
- `Meta_WaitForEvent(...)` blocks a metaobject until it is ready to receive a broadcast message. A metaobject can use this method to wait for a message broadcast by itself.

A message sent with the first two methods must be a pair consisting of an method identifier (`Id`) and an actual argument list (`ArgPac`). By sending a message, a metaobject requests other metaobjects to execute a method of their object so that the methods are executed cooperatively.

The behavior of a metaobject receiving a message is defined by the following methods. The class `OcCoreMetaObj` only declares these methods; their bodies are defined in its subclasses to alter the behavior of each metaobject.

- `Meta_EventCallbackBody(...)` is executed when a broadcast message is received.
- `Meta_SelfEventCallbackBody(...)` is executed when a broadcast message that the metaobject itself sent is received.
- `Meta_ReplyQueryBody(...)` is a method exported to other metaobjects. This method can be called by other metaobjects with the method `Meta_Query()`.

Although basically there is no internal concurrency of a metaobject, these three methods may be executed concurrently when the metaobject is blocked by either the method `Meta_Query()` or `Meta_WaitForEvent()`. This exception is necessary to prevent a deadlock.

The current implementation of *Object Communities* does not provide a preemptive scheduler. The programmer must therefore voluntarily cause a context switch at short intervals. The class `OcCoreMetaObj` defines methods like `WakeupTaskSv()` and `RecvMessage()` to cause a context switch. Note that implementing a preemptive scheduler is possible, and that a preemptive scheduler can, in fact, be obtained if a timer-signal handler is available. The reason that a non-preemptive scheduler is selected is to prevent the internal concurrency of a metaobject that has no meta-metaobject. The methods of a metaobject are executed atomically; they are not preempted.

4 Examples of Method-Call Extension

Many functionalities for distributed computing can be implemented as a group of objects on different machines. Since *Object Communities* provide the ability to manage a group of objects,

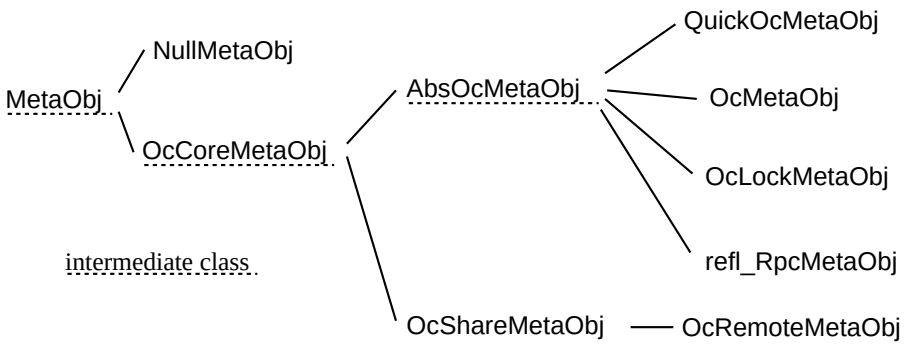


Figure 3: Class hierarchy of metaobjects

such a functionality is implemented on top of Open C++ by defining a subclass of the class `OcCoreMetaObj`. In fact, *Object Communities* already include some subclasses of `OcCoreMetaObj`, which implement various functionalities for distributed computing. Figure 3 illustrates the class hierarchy of metaobjects provided by *Object Communities* in default.

The class `NullMetaObj` is irrelevant to *Object Communities*: it implements a method call that is done in the original manner of C++ method calls. The other subclasses correspond to various functionalities. They implement distributed shared data, transactions, and remote procedure calls. They also implement remote object pointers with which an object can transparently call a method of an object on a different machine. The implementation of remote object pointers exploits other programming techniques such as "smart pointers" [23] so that remote object pointers are naturally available in C++. Furthermore, another subclass implements persistent objects by using the ability of Open C++ to deal with instance variables of an object by the metaobject. Because of space limitation, the details of this ability are not given here; we will present them in another paper.

Here we explain two of the subclasses of *Object Communities*: distributed shared data and transactions. Distributed shared data are implemented by the class `OcMetaObj`. The shared data are replicated and held by the objects that belong to the same object community. The metaobjects control those objects to hold consistent values of the shared data. Suppose that the shared data is an integer and is represented as a variable `p` of the class `SharedData` at the base level. To update the variable `p`, the class `SharedData` has a method `Update()`. If the variable `p` is updated, this method redraws a graphical display according to the new value of `p`:

```
class SharedData {
    ...
public:
//MOP reflect:
    void Update(int new_p) { p = new_p; RedrawDisplay(); }
private:
    void RedrawDisplay();
    int  p;      // inaccessible from the outside of the object
};

//MOP reflect class SharedData : OcMetaObj;
```

An object of the class `SharedData` can be a reflective object, and the class of the metaobject is `OcMetaObj`. The method `Update()` is a `reflect` method. If an object of the class `refl_SharedData` is created, the metaobject makes the object join the specified object community. Then the variable `p` of the object is maintained by the metaobject to hold the same value as the values of `p` of the other objects of the same object community. If the method `Update()` is called, the metaobject requests the other metaobjects to use the same argument `new_p`, and execute the same method of their objects. Thus if the method `Update()` of an object of the object community is called, then the methods of all the objects are executed and the values of `p` are updated keeping consistency. Note that the definition of the class `SharedData` does not include any code concerning distributed computation; such code is in the definition of the metaobject. Methods of the metaobject are defined as follows:

```
void OcMetaObj::Meta_MethodCall(Id method_id, Id category,
                                ArgPac& args, ArgPac& reply){
    // notifying others of a method call
    Id event = Meta_EventNotify(method_id, args);
    // waiting until that notification is serialized
    Meta_WaitForEvent(event, args);
    // executing the called method actually
    Meta_HandleMethodCall(method_id, args, reply);
}

void OcMetaObj::Meta_EventCallbackBody(Id method_id,
                                ArgPac& args, ArgPac& reply){
    //  if other metaobjects report a method call,
    //  the metaobject executes the called method.
    Meta_HandleMethodCall(method_id, args, reply);
}
```

The consistency between the values of `p` is guaranteed even if two metaobjects attempt to execute the method `Update()`. This is because the notifications by those metaobjects are serialized so that every metaobject receives the notifications in the same order.

Since there is no restriction in terms of the definition of the method `Update()`, the programmer can define any action that is executed whenever the shared data are updated. This kind of processing cannot be adequately treated by other mechanisms for distributed shared data, such as distributed shared memory [13], because they do not support a functionality that invokes user-defined actions on each machine that shares the data.

Although in the example above other metaobjects are notified of a method call immediately, some mechanisms for distributed shared data improve performance by using an algorithm in which the notification is delayed [24]. Such an algorithm is also available in Open C++ if the programmer defines a subclass of `OcMetaObj` to implement it. When implementing such an algorithm, it is necessary to distinguish methods that modify the shared data from methods that simply read the data. Category names of `reflect` methods are useful for this. For example,

```
class SharedData2 {
    ...
public:
//MOP reflect(write):
    void Update(int new_p) { p = new_p; RedrawDisplay(); }
```

```
//MOP reflect(read):
    int Get() { return p; }
    ...
};
```

The category names let the metaobject identify the method `Update()` as a "write" method, and the method `Get()` as a "read" one.

Next we show another subclass of *Object Communities*. The class `OcLockMetaObj` of metaobjects implements atomic transactions. Although the concept of atomic transactions includes recoverability (a transaction causes no side-effect if it fails), the class `OcLockMetaObj` does not support recoverability. It only guarantees atomicity; the sequence of the operations in a transaction is executed continuously. The method `Meta_MethodCall()` of the class `OcLockMetaObj` is as follows.

```
void OcLockMetaObj::Meta_MethodCall(Id method_id, Id category,
                                    ArgPac& args, ArgPac& reply){
    while(locked)
        Meta_WaitForEvent();     // block until a lock is released.

    // the following is the same as the method of OcMetaObj
    Id event = Meta_EventNotify(method_id, args);
    Meta_WaitForEvent(event, args);
    Meta_HandleMethodCall(method_id, args, reply);
}
```

The metaobject of the class `OcLockMetaObj` delays the method execution while the execution is locked. To begin a transaction, the programmer calls a method of the metaobject, which locks method execution with a broadcast message. Receiving the message, the other metaobjects of the same object community stop method execution until that metaobject releases the lock. The variable `lock` indicates whether execution is locked or unlocked. It is maintained by messages between metaobjects.

5 Overheads due to having a Meta Level

Efficient implementation of meta-level techniques is a major research topic. Because execution of a reflective object in Open C++ is partly interpreted by a metaobject, its execution is slower than that of a nonreflective object. This section briefly shows the result of measurements in terms of the execution speed.

The current Open C++ compiler is a preprocessor of the C++ compiler. Because no modification is added to the C++ compiler, an Open C++ program is translated into a plain C++ code. Calling a `reflect` method thus imposes some overhead that by some standards is not small.[‡] We show the result of performance measurements of method calls.

Table 1 lists latency time for three kinds of null method calls. These values were measured on a SPARC station 2 (SunOS 4.1.1), and the compiler was Sun C++ 2.1. The latency was measured for different numbers of arguments. The type of arguments was `int` except for the data of the rightmost column, for which the type was `double`. Although the 0-argument method does not

[‡]The initial version of the Open C++ compiler showed that a `reflect` method call was 100 times slower than a `virtual` method call of C++.

Table 1: Average Latency (μsec.) of a null method call

number of arguments	0	1	5	5 × `double`
C++ function	0.3	0.6	1.3	2.1
C++ `virtual` method	0.8	1.0	1.8	2.2
`reflect` method	1.8	6.3	13.8	21.7
`reflect`/`virtual` ratio	2.3	6.3	7.7	9.9

SPARC 40 MHz (28.5 MIPS) and Sun C++ 2.1

return anything, the other methods return an `int` value. A method that takes 5 `double` arguments returns a `double` value. The three kinds of null method calls are a C++ function, a `virtual` method, and a `reflect` method. The first two are supported by both C++ and Open C++, whereas the last is available only in Open C++. A C++ function call is to call a method of an object pointed to by a variable. This takes a form like `ptr->func()`. A `virtual` method call is to call a method of an object whose class is unknown at compile time; a method name is dynamically bound to a method body. A `reflect` method call is one controlled by a metaobject of the class `NullMetaObj`, which implements a method call so that its behavior is the same as that of a C++ function call.

The last line of the table shows the ratio of the latency of a `reflect` method call to that of a `virtual` method call. This ratio increases with the number of arguments because the overhead of a `reflect` method call is mainly due to the reifying process of the argument list of the method call. Arguments are copied to an `ArgPac` class object separately when the `reflect` method is called. The overhead for this copying increases in proportion to the number of arguments. Since the 0-argument method takes no argument, its overhead is smaller than that of the other methods.

The result of these measurements shows that a `reflect` method call is 6 to 8 times slower than a `virtual` method call. Although this overhead seems important, it is actually negligible if Open C++ is used for distributed computing, since the network latency time is between several hundred microseconds and several milliseconds. The overhead is also reduced by a proper designing of the applications. In carefully designed applications, distributed computation is localized in a small number of objects, which would be reflective, and the other objects are executed without overhead since Open C++ allows to specify whether or not an object is reflective. We believe that meta-level techniques are already applicable to practical programming if the programmer selects a domain in which the overhead is negligible in comparison with the overhead for performance of a functionality implemented with the meta-level technique.

Furthermore, from the viewpoint of distributed computing, the overhead of Open C++ is due to the cost of the marshaling/unmarshaling process, in which transferred data are converted into a network message. Because this process commonly appears in distributed computing, the overhead of Open C++ is almost equivalent to that of other approaches such as Sun's RPC [25]. When Sun's RPC library is used, each conversion of an `int` argument takes a few microseconds because that library is a general one, and a few nested function calls are needed whenever a converting routine

(an XDR routine) is called.

If the increased overhead of a meta-level technique is limited to within a factor of 10, then the advantage of that meta-level technique is worthwhile. In the concurrent language ABCL/R2 [17], for example, the execution that involves a meta-level operation is 6 or 7 times slower than a normal execution [16]. As in Open C++, the programmer can select whether or not an object is controlled by a metaobject. As a result, ABCL/R2 improves the execution speed of a program by a meta-level technique.

6 Related Work

C++ provides some meta-level operations. The macro set of handling a variable argument list can be considered to provide a few restricted meta-level operations. It allows the programmer to traverse an argument list whose length and element types are variable, as if the argument list were a first-class entity. Operator overloading is also a meta-level operation because it enables the replacement of predefined operators, such as `+` and `->`, with user-defined procedures. No meta-level information is available in a overloading procedure, however, because operator overloading is not implemented by using the concept of reflection.

The stub generator [2] of remote procedure calls, such as Sun's `rpcgen` [25], has a functionality similar to that of the Open C++ compiler. It reads the description file of a remote procedure and then generates a stub routine, which is a utility routine for calling the remote procedure. Unlike the Open C++ compiler, however, the stub generator does not expose the inside of a stub routine, so the programmer cannot alter the implementation of a stub routine in a well-organized manner. The FOG compiler [7] provides the ability of extending a generated code. It allows to use in C++ a fragmented object (FO), which is a distributed object. In the FOG compiler, the programmer can specify a communication protocol of a remote procedure call.

Meta-level (or reflection) techniques have been applied in various domains and they are still an active area of research. CLOS MOP [10] is the first try to apply the meta-level techniques to a practical language. It provides an extensible implementation of CLOS [22]: all specifications of CLOS are modifiable. The mechanism for method lookup, for example, is extensible by a metaobject. There are several reflective language systems other than CLOS MOP. ABCL/R2 applies a meta-level technique to parallel computation, and RbCl [9] tries to minimize the run-time kernel that is not extensible. AL1/D [18] provides multiple abstract models for each aspect of the implementation, and this is effective when many aspects of the implementation are exposed. The programmer can alter each aspect independently, without considering other aspects.

Meta-level techniques are also beginning to be used for commercial systems. The Meta-Information-Protocol (MIP) [3] used in some commercial systems, is a mechanism for accessing the type information of a C++ object at run time. It represents type information by a metaobject so that typesafe downcast is available in C++. Because a metaobject in the MIP exposes internal information but a change of the metaobject does not influence behavior of an object, the overheads of the MIP is obviously small with respect to execution speed compared with Open C++. Meta-level techniques are also used for developing systems other than languages, such as an operating system and a window system. Apertos [27] is an operating system completely based on a meta-level technique, and Silica [19] is a window system with which the programmer can alter how the

system draws an image on a window, how the relationship of windows is maintained, and so on. The Choices operating system uses a meta-level technique to implement its kernel and subsystems [14]. Using macros and programmer conventions, Choices exploits a meta-level technique within the confines of plain C++.

Some researchers try to reduce the cost associated with having the meta level. CLOS MOP, for example, has no costs beyond these of plain CLOS. This is achieved by careful protocol design and by implementation devices in which, for example, calls to the meta-level functions are partially evaluated. Because the execution mechanism of CLOS has inherent complexity and costs, the cost due to the meta level can be recovered by those techniques. On the other hand, C++ is designed so that the program is directly translated into efficient assembly code. The C++ method call, for example, is compiled into a few machine instructions. The techniques used for CLOS MOP are therefore insufficient to implement Open C++ MOP without overhead.

Anibus [20] and Intrigue [12] support "compile-time" MOPs to reduce the cost due to the meta level. They are Lisp compilers that are extensible according to MOP. The "compile-time" MOPs modify the compilers to compile a program in a different scheme. Because a meta code replaces an internal code of the compilers instead of a compiled code, this approach, like that of CLOS MOP, does not generate overheads. In this approach, however, meta code must describe not how an object behaves, but how the compiler generates compiled code that makes an object behave according to the programmer's intention. Although this approach has no overhead, its meta code is less straightforward than those in CLOS MOP and Open C++.

7 Conclusion

This paper described Open C++ in order to demonstrate a methodology for designing extensible languages for distributed computing. Open C++ is designed on the basis of an object-oriented meta-level (or reflection) technique so that the implementation of a method call is made open-ended. The programmer can alter the implementation of a method call according to a simple metaobject protocol (MOP), and obtain on top of Open C++ a new language functionality for distributed computing. Open C++ MOP is made so simple and easily understandable that programmers who are not familiar with the meta system can implement a new functionality effortlessly on top of Open C++. The MOP exposes the implementation of a method call with some abstraction. Open C++ also provides *Object Communities*, which is a framework that facilitates meta-level programming for implementing a functionality for distributed computing.

Open C++ clearly separates distributed computation from the other computation that is more substantial to the programmer. Computation concerning communication and synchronization notions appears only at the meta level, and need not be considered by the programmer writing a program at the base level. This feature of Open C++ makes a program more understandable and easier to describe.

The overhead associated with Open C++ MOP is negligible when Open C++ is used for distributed computing, since even though it is not small, it is negligible in comparison with network latency time. How much performance the system using the MOP must achieve depends on the operations controlled by the MOP. Although meta-level techniques are still difficult to implement efficiently, they are already applicable to practical programming if the domain is selected properly.

Unlike CLOS MOP, Open C++ introduces a meta-level technique into a compiler-based language. Because Open C++ must bridge an abstraction gap between C++ and an assembly language, its design considered implementation issues that the design of CLOS MOP did not. It restricts the extensible part of the language specifications in order to reduce the cost associated with the meta level. The entities that the MOP reifies are only those necessary for distributed computing. To apply Open C++ in application domains such as parallel computing as well as distributed computing, however, the overhead due to extensibility needs to be further reduced.

Acknowledgments

We thank Satoshi Matsuoka for his suggestions on clarifying and organizing this work. We also thank Gregor Kiczales, Hidehiko Masuhara, and Frank Buschmann for their helpful comments on earlier drafts of this paper.

References

[1] Bal, H. E., M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A Language For Parallel Programming of Distributed Systems," *IEEE Trans. Softw. Eng.*, vol. 18, no. 3, pp. 190–205, 1992.

[2] Birrell, A. D. and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, vol. 2, no. 1, pp. 39–59, 1984.

[3] Buschmann, F., K. Kiefer, F. Paulisch, and M. Stal, "The Meta-Information-Protocol: Run-Time Type Information for C++," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 82–87, 1992.

[4] Ellis, C., S. Gibbs, and G. Rein, "Groupware –Some Issues and Experiences," *Commun. of the ACM*, vol. 34, no. 1, pp. 38–58, 1991.

[5] Gehani, N. and W. Roome, "Concurrent C," *Software–Practice and Experience*, vol. 16, no. 9, pp. 821–844, 1986.

[6] Goldberg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[7] Gourhant, Y. and M. Shapiro, "FOG/C++: a Fragmented-Object Generator," in *Proc. of USENIX C++ Conference*, pp. 63–74, 1990.

[8] Herlihy, M. and B. Liskov, "A Value Transmission Method for Abstract Data Types," *ACM Trans. Prog. Lang. Syst.*, vol. 4, no. 4, pp. 527–551, 1982.

[9] Ichisugi, Y., S. Matsuoka, and A. Yonezawa, "RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 24–35, 1992.

[10] Kiczales, G., J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[11] Kiczales, G. and J. Lamping, "Issues in the Design and Specification of Class Libraries," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 435–451, 1992.

[12] Lamping, J., G. Kiczales, L. Rodriguez, and E. Ruf, "An Architecture for an Open Compiler," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 95–106, 1992.

[13] Li, K., *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Dept. of Computer Science, Yale Univ., 1986.

[14] Madany, P., P. Kougiouris, N. Islam, and R. H. Campbell, "Practical Examples of Reification and Reflection in C++," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 76–81, 1992.

[15] Maes, P., "Concepts and Experiments in Computational Reflection," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 147–155, 1987.

[16] Masuhara, H., S. Matsuoka, T. Watanabe, and A. Yonezawa, "Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 127–144, 1992.

[17] Matsuoka, S., T. Watanabe, and A. Yonezawa, "Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming," in *Proc. of European Conf. on Object-Oriented Programming '91*, no. 512 in LNCS, pp. 231–250, Springer-Verlag, 1991.

[18] Okamura, H., Y. Ishikawa, and M. Tokoro, "AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 36–47, 1992.

[19] Rao, R., "Implementational Reflection in Silica," in *Proc. of European Conf. on Object-Oriented Programming '91*, no. 512 in LNCS, pp. 251–267, Springer-Verlag, 1991.

[20] Rodriguez Jr., L. H., "Coarse-Grained Parallelism Using Metaobject Protocols," Techincal Report SSL-91-61, XEROX PARC, Palo Alto, CA, 1991.

[21] Smith, B. C., "Reflection and Semantics in Lisp," in *Proc. of ACM Symp. on Principles of Programming Languages*, pp. 23–35, 1984.

[22] Steele, G., *Common Lisp: The Language*. Digital Press, 2nd ed., 1990.

[23] Stroustrup, B., *The C++ Programming Language*. Addison-Wesley, 2nd ed., 1991.

[24] Stumm, M. and S. Zhou, "Algorithms Implementing Distributed Shared Memory," *IEEE Computer*, vol. 23, no. 5, pp. 54–64, 1990.

[25] Sun Microsystems, *Network Programming Guide*. Sun Microsystems, Inc., 1990.

[26] U.S. Dept. of Defense, *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A, 1983.

[27] Yokote, Y., "The Apertos Reflective Operating System: The Concept and Its Implementation," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 414–434, 1992.

[28] Yokote, Y. and M. Tokoro, "The Design and Implementation of ConcurrentSmalltalk," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 331–340, 1986.

A Metaobject Protocol for C++

Shigeru Chiba

Xerox PARC & University of Tokyo
chiba@parc.xerox.com chiba@is.s.u-tokyo.ac.jp

Abstract

This paper presents a metaobject protocol (*MOP*) for C++. This MOP was designed to bring the power of meta-programming to C++ programmers. It avoids penalties on runtime performance by adopting a new meta-architecture in which the metaobjects control the compilation of programs instead of being active during program execution. This allows the MOP to be used to implement libraries of efficient, transparent language extensions.

1 Introduction

A metaobject protocol (MOP) is an object-oriented interface for programmers to customize the behavior and implementation of programming languages and other system software. The usefulness of this kind of customizability has been argued elsewhere[11, 9, 10], and interesting MOPs have been included in languages such as Lisp[20], ABCL[21], and, to a lesser degree, Smalltalk[6]. The goal of our work is to bring the power of meta-programming to the more mainstream language C++, while respecting their performance concerns in that community.

This paper proposes a new MOP for C++, called OpenC++ Version 2. Like previous MOPs, it allows programmers to implement customized language extensions such as persistent or distributed objects, or customized compiler optimizations such as inlining of matrix arithmetic. These can be implemented as libraries[1] and then used repeatedly. Unlike previous MOPs, our proposal incurs *zero* runtime speed or space overhead compared to ordinary C++.

To make this possible, our MOP works by providing control over the *compilation* of programs rather than over the runtime environment in which they execute. Specifically, our MOP provides control over the compilation of the following key aspects of C++: class definition, member access, virtual function invocation, and object creation. This feature means that the design of our MOP is *inherently* efficient, as opposed to MOPs, such as the CLOS MOP, where only sophisticated implementation techniques enable efficient execution.

Our MOP has been developed by a synthesis and re-engineering of a number of ideas in this field: we took our basic protocol structure from the CLOS MOP [11], we took the basic structure of a compile-time MOP from Anibus and Intrigue [18, 13], and we took some ideas for the basic structure of a MOP for C++ from the meta-information protocol (MIP) [2] and our previous work on Open C++ Version 1 [4].

This paper is a status report on the development of OpenC++ Version 2. Our MOP has been prototyped in Scheme and a number of examples are running using the prototype. For simplicity, however, we use C++ notation in this paper. In the rest of the paper, we first discuss what we want our

Appeared in OOPSLA'95 Proceedings pp.285–299

[1] We use the term *library* to mean a collection of reusable code such as functions, data types, and constants, written by the language users within the description capability of the language.

MOP to support, then we present the basic architecture of the MOP, and we illustrate its suitability for real-world programming.

2 What We Want to Enable

The motivation for our C++ MOP is to enable programmers to easily write libraries that provide language extensions transparently and efficiently. This section illustrates what we want to enable with the MOP.

Suppose we want to implement persistent objects as a C++ library. In terms of transparency, the goal is to allow the library users to specify that some of their classes should produce persistent objects simply by adding an annotation to ordinary class definitions.

```
persistent class Node {
public:
  Node*  next;
  double value;
};
```

The program that deals with persistent objects should look like:

```
Node* get_next_of_next(Node* p)
{
  Node* q = p->next;
  return q->next;
}
```

The key point is that adding or removing the simple annotation `persistent` should be all that is required to change this from defining persistent objects or transient ones.

Unfortunately, such a simple annotation is quite difficult to implement in C++. For example, one way to try to do this in C++ is to develop a class library that provides a class `PersistentObject` from which other classes can inherit if they want to be persistent. In such a scheme, the hope is that the definition of the class `Node` would look like:

```
class Node : public PersistentObject {
public:
  Node*  next;
  double value;
};
```

But, the inheritance mechanism does not provide enough access to implement persistent objects by itself; the library user will also have to edit their programs to correctly use that functionality. To implement persistence, references to persistent objects must be detected at runtime. If this is not done in hardware, the software will need to be edited to look something like:

```
Node* get_next_of_next(Node* p)
{
  Node* q = (p->Load(), p->next);
  return (q->Load(), q->next);
}
```

Because the class `PersistentObject` cannot control its subclasses' member accesses, the library user will have to call the member function `Load()` before every access to the object. `Load()` is a member function supplied by the class `PersistentObject` to load objects from disk on demand.

The situation may not appear so bad in this simple example, but tracking down and editing all such accesses can be quite difficult. The library implementor must describe the need for editing in a document, and the user must carefully read and follow those instructions. More importantly, changing code back and forth from persistent to transient is extremely labor-intensive.

Our MOP provides the ability to implement a persistent object library so that persistence can be selected with only a simple edit to the class definition. It enables not only annotations for language extensions but also ones for compiler optimizations.

From the pragmatic viewpoint, the design criteria of such a MOP are high performance and arbitrary customizability. For the former, the MOP should not include any runtime overhead or preclude optimization by the current C++ compiler. For the latter, the MOP should provide the ability to implement common C++ extensions such as persistent C++ or distributed C++.

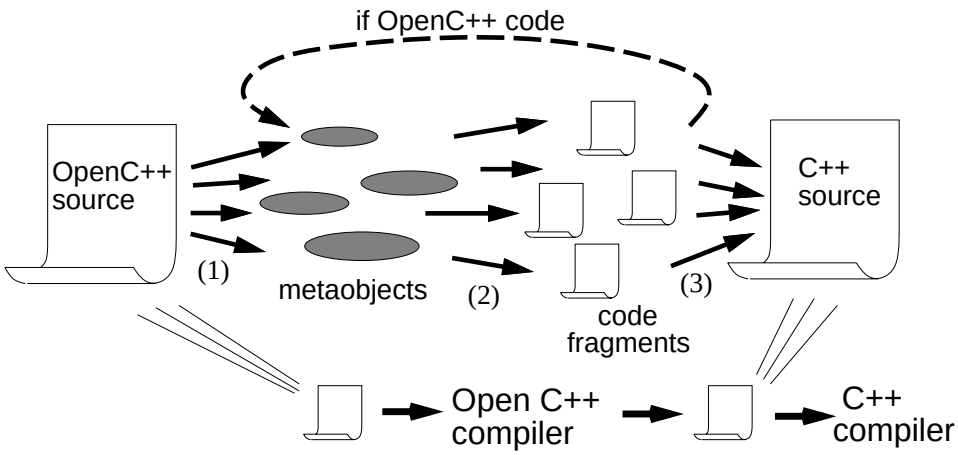


Figure 1: The Protocol Structure

3 The Basic Architecture

The basic architecture of the OpenC++ MOP is similar to that of the CLOS MOP in that metaobjects represent language entities visible to the programmer. There are class metaobjects and function metaobjects. The behavior of the program is controlled by those metaobjects.

3.1 Compile-time and Runtime

A distinguishing feature of the OpenC++ MOP as compared to the CLOS MOP is that the OpenC++ MOP clearly separates the compile-time environment and the runtime environment. Ordinary objects exist only at runtime, and metaobjects exist only at compile-time.[2]

Since the metaobjects exist only at compile-time, the way they alter the behavior of the objects is by controlling the compilation of the program. The metaobjects appropriately translate top-level definitions of the program, and, if necessary, append supplementary runtime functions, types, and data to the translated code.

This means that our MOP inherently implies no penalty in runtime space or speed. On the other hand, in the CLOS MOP, key aspects of the object system are executed through runtime method invocation of the metaobjects. The CLOS MOP hence requires sophisticated implementation tricks to achieve good runtime performance.

3.2 Basic Protocol Structure

The OpenC++ MOP controls source-to-source translation from OpenC++ (extended C++) to C++. First of all, the source code of the OpenC++ program is parsed and divided into top-level definitions for classes and (member-)functions.[3] Then a metaobject is constructed for each such definition. The metaobject then translates the top-level definition into appropriate ordinary C++ (or C) code. The translated code is then collected and assembled into contiguous source code. Figure 1 shows this protocol structure.

To see how this works, we now walk through an example of how the MOP compiles a small program, specifically the two definitions shown below:

```
class Point {
public:
  void MoveTo(int, int);
  int x, y;
};
```

[2] A simple programmer extension that we will show later can allow some of the functionality of runtime metaobjects.

[3] The definition of global variables is also a top-level definition. But in this paper, we ignore it for simplicity.

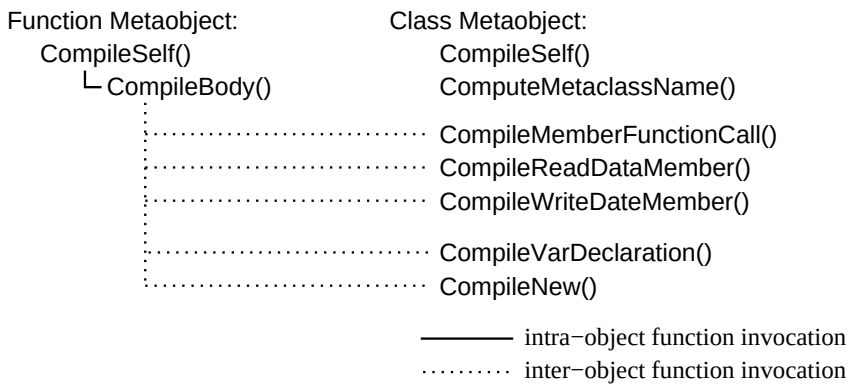


Figure 2: Overview of the Protocol

```
void Point::MoveTo(int new_x,
                   int new_y)
{
  x = new_x; y = new_y;
}
```

In phase 1, after parsing, the MOP constructs two metaobjects, one for the class `Point` and one for the member function `MoveTo()`. By default, the class metaobject is an instance of the class `Class`. It contains information given by the class definition such as its name, base classes, members, etc. The function metaobject is an instance of the class `Function` by default. It contains similar information from the function definition, such as its name, parameters, and the parse tree of the function body. Since this is a member function, it also has a pointer to the class metaobject that supplies this member function.

In phase 2, the metaobjects are called upon to generate appropriate code fragments to substitute for the original code fragments of their definitions. To do that, the member function `CompileSelf()` of each metaobject is invoked.

In response to `CompileSelf()`, class metaobjects generate an ordinary C++ definition — in the form of parse tree — for their class. The default is just to emit the original definition. Similarly, function metaobjects generate an ordinary C++ definition — also in the form of a parse tree. Again, the default is to just emit the original definition.

To translate the function body, function metaobjects follow a layered sub-protocol, walking the parse tree of the body step by step, asking appropriate class metaobjects to translate each fragment of the body. This is done by passing the parse tree of the code fragment to the class metaobject, and getting the parse tree of the translated fragment back. This layered protocol is shown in Figure 2. Note that since the function metaobject compiles the function body by making queries of the class metaobjects, the compilation of the program is mainly the responsibility of the class metaobjects. The default action of the class metaobjects is to just return the given parse tree without any translation.

In the example of `MoveTo()`, since `x` is a data member of the class `Point`, it means that when `x=new_x` is encountered during the tree traversal, the function metaobject invokes the member function `CompileWriteDataMember()` of the class metaobject `Point`. That member function translates the parse tree that corresponds to the expression `x=new_x`, and returns the translated parse tree. By default, the translated expression is also `x=new_x`. The original subtree is then replaced with the returned subtree. Similarly, if the function metaobject encounters a variable declaration such as "`Point* p`", it invokes the member function `CompileVarDeclaration()` of the class metaobject `Point` to translate the subtree of the variable declaration.

In phase 3, the parse trees generated by the individual metaobjects are then collected and converted into C++ source text. This source text is then compiled by the C++ compiler. Since the MOP is implemented as a separate preprocessor of the C++ compiler, the converted C++ text is in the form of character file. So the C++ compiler must parse the contents of the file again. We could avoid this overhead, however, if we integrated the MOP into the C++ compiler.

A conceptually significant point is that the metaobjects are permitted to generate arbitrary code fragments, so the generated fragments may contain OpenC++ code. If that happens, the translation is recursively applied to each generated fragment until it becomes ordinary C++ code. Phases 1 and 2 are repeated on each fragment so that all code fragments become C++ code before starting phase 3.

3.3 A Simple Programmer Customization

Now we show a simple example of what programmers can do with this MOP. Suppose we want to specialize class metaobjects to implement a mechanism for getting class information such as data member names at runtime. In essence, the idea is to implement a subset of the functionality of the Meta-Information Protocol[2]. At runtime, for every class, this mechanism will automatically and transparently make available a record that contains the class information.

Any C++ customization implemented as a MOP-based library is usually divided into two parts: *compile-time code* and *runtime code*. The former is a class library of metaobjects, and the latter is a set of runtime support routines. Given the two, the programmer can then write a *user program* that simply uses the C++ customization (see Figure 3).

First we show the user program the library user wants to be able to write:

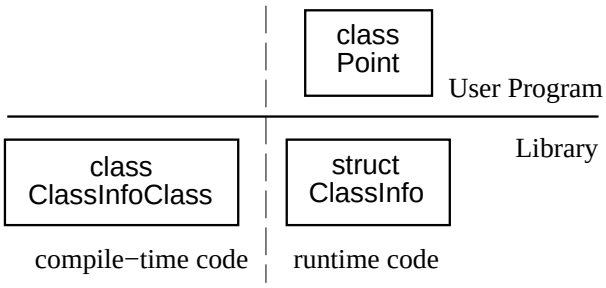


Figure 3: MOP-based library

```
metaclass Point : ClassInfoClass;
class Point { ... };
  ...
int   i = 0;
char* name;
while((name=ClassDataMemberNames(Point)
            [i++]) != NULL)
  printf("Point's data member: %s\n",
         name);
```

This program prints the names of the data members of the class `Point`. The first statement specifies the *metaclass* of the class `Point`. It declares that the class metaobject for `Point` should be an instance of the class `ClassInfoClass`. This annotation directs the MOP to produce runtime class information for the class `Point`. The user can then access it through the runtime functions supplied by the library. In the code above, `ClassDataMemberNames()` is one such runtime access function.

To enable such a user program, the library implementor must write the appropriate compile-time and runtime library, as shown below. First, we show the runtime code. It includes the definition of `ClassInfo`, a record type for class information such as the class name and the data members. It also includes the function `ClassDataMemberNames()` to access `ClassInfo`.[4]

```
struct ClassInfo {
  char*  class_name;
  char** data_member_names;
};
```

[4] That function is implemented as a macro because C++ does not deal with a symbol name as first-class data. `##` is a macro operator for concatenating two symbol names.

```
#define ClassDataMemberNames(name)\
      ((info_##name).data_member_names)
  ...
```

Next, we show the compile-time code. Its objective is to control the compilation so that the runtime support routines work appropriately with the user's program. In this example, compilation of the definition of the class `Point` must translate it into this:

```
class Point { ... };    // not changed
ClassInfo info_Point = {
  "Point",
  {"x", "y", NULL}
};
```

The second line makes the record `info_Point`, which contains the class information of `Point`. To do such translation, we define a subclass of `Class` for the compile-time library.

```
class ClassInfoClass : public Class {
public:
  Expression CompileSelf();
};

Expression
ClassInfoClass::CompileSelf()
{
  Expression code = MakeParseTree(
    "ClassInfo info_%s = {\"%s\", {",
    name, name);

  for(int i = 0;
      i < n_data_members;
      ++i)
    code->Append(
      MakeParseTree(
        "\"%s\",",
        data_member_names[i]));

  code->Append(
    MakeParseTree("NULL}};"));

  return Class::CompileSelf()
         ->Append(code);
}
```

The member function `CompileSelf()` simply produces a parse tree that corresponds to a C++ statement that makes the record data such as `info_Point`. It first constructs the statement in the form of a character string, then converts it to a parse tree by using a utility function `MakeParseTree()`, which receives a string and replaces `%s` symbols with other sub-strings in a similar way to C language's `printf()` function. Then it converts the string to a parse tree. The result of `CompileSelf()` is the concatenation of the produced parse tree and the result of the base class' `CompileSelf()`. It is substituted for the original definition of the class `Point` in the user's program.

4 Suitability for the Real World

A MOP itself is a mechanism to implement something necessary. This section presents how the OpenC++ MOP is utilized for practical programming.

The OpenC++ MOP can be viewed as a tool for implementing libraries that efficiently and transparently provide useful facilities for the programmer. In other words, the OpenC++ MOP is primarily a mechanism for library implementors. The benefit to library users is not the MOP itself but the transparency and efficiency of the library implemented with the MOP.

4.1 Persistent-Object Library (revisited)

Using the OpenC++ MOP, the persistent objects described in Section 2 can be transparently provided for library users. A new metaclass is used to encapsulate the implementation of the extension from the user-defined classes. Such a new metaclass is developed through the three steps seen before: 1) decide what the user program should look like, 2) figure out what it should be translated to, and 3) write appropriate compile-time and runtime code to perform and support the translation. We show the persistent-object library along those three steps.

First, the program the library user writes should look as follows:

```
metaclass Node : PersistentClass;
class Node {
public:
  Node*  next;
  double value;
};

Node* get_next_of_next(Node* p)
{
  Node* q = p->next;
  return q->next;
}
```

That is, the library user can obtain persistent objects simply by specifying the metaclass `PersistentClass`.[5] There is no need for the user program to explicitly invoke `Load()` as in the C++ code in Section 2.

The next step is to figure out how to translate the user program. The library implementor must decide what should be supplied as the runtime code of the library, what should be generated for each user class and should be directly embedded in the translated code. In this example, the function `Load()` is supplied as a runtime support routine, while invocation of `Load()` is embedded in the translated code by the metaobjects. The user program should be translated to:

```
Node* get_next_of_next(Node* p)
{
  Node* q = (Load(p), p->next);
  return (Load(q), q->next);
}
```

Italic letters indicate code inserted by the metaobjects. The metaobjects translate the user program so that it appropriately invokes runtime support routines. Unlike in Section 2, the inserted code is `Load(p)` instead of `p->Load()`. This syntactical change is because `Load()` is an ordinary function. `Load()` is not supplied as a member function any more since the class `Node` does not inherit from any base class.

Finally, the library implementor writes the compile-time code of the library. For the translation specified at the previous step, the class `PersistentClass` is defined as a subclass of `Class`. It redefines the member function `CompileReadDataMember()` and `CompileWriteDataMember()` so that the member accesses like `p->next` are appropriately translated. The definition of `CompileReadDataMember()` is as follows:

```
Expression
PersistentClass::CompileReadDataMember(
                    Environment env,
                    String member_name,
                    String variable_name)
{
  return MakeParseTree(
            "(Load(%s), %e)",
            member_name,
            Class::CompileReadDataMember(
                      ...));
}
```

The `CompileReadDataMember()` supplied by the base class `Class` returns the parse tree that corresponds to the original code, in this example, `p->next`.

More Realistic Implementation

The implementation shown above is only part of the persistent library. In a typical implementation, when a persistent object is loaded onto memory, pointers that the object contains must be translated to point to correct memory addresses because the actual layout of objects changes every session. This translation, which is often called *pointer swizzling*, is needed even if references to persistent objects are detected by a virtual memory system, which is more sophisticated implementation.

To perform pointer swizzling, the runtime function `Load()` has to know which data fields of persistent objects hold pointer values. So the library must record the type of an object and its type definitions when the object is constructed at runtime. Recording the type definitions can be implemented in a similar way to the Meta-Information Protocol shown in Section 3.3.

To record the type of the object, for example, an expression `new Node()` should be translated into:

[5] Our MOP makes it possible to put even more syntactic sugar on this so the programmer can simply write: `persistent class Node {...};`. This is discussed in Section 5.

```
(Node*)RecordObjectType(new Node(),
                        "Node")
```

The runtime support function `RecordObjectType()` receives a pointer to the constructed object and the type of that object, records the type, and returns the received pointer.

The definition of the member function `CompileNew()` that performs that translation is as follows:

```
Expression
PersistentClass::CompileNew(
                  Environment env,
                  Expression arguments,
                  Expression keywords)
{
  char* pat =
    "(%s*)RecordObjectType(%e, \"%s\")";
  return MakeParseTree(
           pat,
           name,
           Class::CompileNew(env,
                             arguments,
                             keywords),
           name);
}
```

4.2 Matrix Library

The next example is a matrix library that can be implemented more efficiently if our MOP is exploited. A new metaclass is used to optimize the implementation of a specific class, which is also provided by the library.

This example is based on C++'s mechanism of overloading operators that allows us to implement a matrix library with which matrices are available in an arithmetic expression. For example, the programmer can write:

```
Matrix a, b, c, d;
  ...
a = b + c + d;
```

Although matrix arithmetic is transparently provided, the performance of this library is quite bad because each `+` operation is done through a function call and thus the intermediate value is passed as a `Matrix` object between the function calls. The following inlining is clearly better.

```
Matrix a, b, c, d;
  ...
for(i = 0; i < number_of_rows; ++i)
  for(j = 0;
      j < number_of_columns;
      ++j)
  {
    a.element[i][j] = b.element[i][j]
                    + c.element[i][j]
                    + d.element[i][j];
  }
```

Directing such an inlining scheme to the compiler is not possible within the confines of C++. Expecting the C++ compiler to automatically detect all such kinds of optimization is not realistic [1].

Our MOP enables the programmer to write a library of such customized optimizations. It allows the library implementor to provide the class `Matrix` with a metaclass for the inlining by using knowledge of implementation details of the class `Matrix`. The metaclass `MatrixClass` will have this member function:

```
Expression
MatrixClass::CompileMemberFunctionCall(
               Environment env,
               String member_name,
               String variable_name,
               Expression arguments)
{
  if(member_name == "="){
    Expression assigned_expr
               = arguments->GetFirst();
    if(inlining is applicable)
      return MakeParseTree(
               "for(i = 0; ...", ...);
  }

  // Otherwise, do the default translation.
  return
    Class::CompileMemberFunctionCall(
              env, member_name, ...);
}
```

Because C++ regards an overloaded operator as a member function call, the member function `CompileMemberFunctionCall()` shown above is invoked to translate the expression `a = b + c + d` into the inlined form. On this invocation, the operator `=` is interpreted as a member function call on the object `a`. The sub-expression

`b + c + d` is an argument of the `=` operator, so the parse tree of that sub-expression is passed to `CompileMemberFunctionCall()` as the parameter `arguments`. The invoked member function checks the passed parse tree, and then, if the inlining is applicable, it actually applies the inlining to the expression; it generates a parse tree that corresponds to the inlined expression. Otherwise it would have delegated the translation to the base class.

To determine whether the inlining is applicable, the compile-time code must traverse the passed parse tree and see whether the shape of the tree matches a particular pattern. For example, if the parse tree represents repetition of an identifier and the symbol `+`, then the inlining is applicable.

4.3 Customizing Implementation of Objects

Some C++ compilers, such as GNU C++ 2.5.8 for SPARC and Sun C++ 2.0, do not allocate objects in registers. For example, if we compile the following program:

```
class Vector {
public:
  double x, y;
};

double xpos[1000], ypos[1000];
Vector v;
  ...
for(int i = 0; i < 1000; ++i){
  xpos[i] += v.x;
  ypos[i] += v.y;
}
```

In the `for` loop, the compiled code loads the values of `v.x` and `v.y` into registers for every iteration. This is obviously redundant since the values should be loaded only once before the iteration starts.

An experienced programmer can remove those redundant load instructions by editing the `for` statement as follows:

```
double s = v.x, t = v.y;
for(int i = 0; i < 1000; ++i){
  xpos[i] += s;
  ypos[i] += t;
}
```

Since `double` variables `s` and `t` are allocated on registers, the compiled code does not include redundant load instructions and runs faster.

Although this kind of technique is popular in C and C++ programming, the performance of the compiled code depends on implementation of compilers. The technique shown above works for particular compilers, but it may not for other compilers or even other versions of those compilers.

Our MOP provides the ability to do such optimization in a more sophisticated way. Programmers can separate description for optimization from the rest of the program. The program would look like:

```
metaclass Vector LightWeightClass;
class Vector {
public:
  double x, y;
};

double xpos[1000], ypos[1000];
Vector v;
  ...
for(int i = 0; i < 1000; ++i){
  xpos[i] += v.x;
  ypos[i] += v.y;
}
```

The metaclass `LightWeightClass` specializes implementation of instances of the class `Vector`.[6] Since the implementation scheme is separately described from the program above, programmers can independently customize it to fit their compilers.

To do the optimization shown first, the metaclass `LightWeightClass` redefines the member functions to translate the program into:

```
double xpos[1000], ypos[1000];
double x_of_v, y_of_v;
  ...
```

[6] Our MOP enables more smart annotation. As discussed later, programmers can switch implementation by simply adding or removing an annotation `lightweight` to variable declaration. For example, if they write:

```
lightweight Vector v;
```

the variable `v` is implemented in the lightweight way. Otherwise, it is done in the ordinary way.

```
for(int i = 0; i < 1000; ++i){
  xpos[i] += x_of_v;
  ypos[i] += y_of_v;
}
```

Now the `Vector` object `v` is implemented with two distinct `double` variables.

The definition of the member function `CompileVarDeclaration()` is as below. It translates variable declarations.

```
Expression
LightWeightClass::CompileVarDeclaration(
                    Environment env,
                    String variable_name)
{
  return MakeParseTree(
           "double x_of_%s, y_of_%s",
           variable_name,
           variable_name);
}
```

Also, the member function `CompileDotReadDataMember()` is redefined to translate data-member accesses.

```
Expression
LightWeightClass
::CompileDotReadDataMember(
    Environment env,
    String member_name,
    String variable_name)
{
  return MakeParseTree("%s_of_%s",
                       member_name,
                       variable_name);
}
```

4.4 Selecting a Concrete Class at Compile Time

The last example is a mechanism to select the most appropriate concrete class for a given abstract class at compile time. With this mechanism, programmers do not have to directly instantiate a specific concrete class. Instead, they can instantiate an abstract class with an annotation about the requirement for the implementation of the instance. The effective concrete class is selected by the compiler.

A class like `Set` can be implemented with different data structures and algorithms. Since appropriate implementation depends on user programs, a typical implementation scheme is to define several subclasses each of which corresponds to a different implementation scheme, such as `LinkedListSet`, `ArraySet`, and `SortedSet`. But in the typical implementation scheme, selecting the appropriate concrete class happens at runtime, thereby incurring a performance overhead.

The mechanism we show below is similar to that in [14], but it automatically selects the most appropriate subclass at compile time. The user just annotates requirements for each instantiation of a class, then the compiler selects a subclass to mostly satisfy the requirements. For example, the user program of the `Set` library will look like:

```
Set* s = new Set("size<=1000;sorted");
```

The requirement is specified as the first initialization parameter to the class `Set`.[7] The compiler selects a subclass of `Set` that matches the specified requirement, and translates the code above into like:

```
Set* s = new SortedArraySet();
```

`SortedArraySet` is a subclass of `Set`, which the compiler selects for that particular user program.

To enable that translation, the library implementor defines a metaclass `SetClass` of the class `Set`. It redefines the member function `CompileNew()` as follows:

```
Expression
SetClass::CompileNew(Environment env,
                     Expression arguments,
                     Expression keywords)
{
  Expression requirement
               = arguments->GetFirst();
  String class_name = SelectSubclass(
                        requirement);
  return MakeParseTree("new %s()",
                       class_name);
}
```

[7] If we use our MOP's capability to extend the syntax, we can separate the requirement from the initialization parameter. The program could be:

`new require("size<=1000;sorted") Set()`.

```
String SetClass::SelectSubclass(
                  Expression requirement)
{
  ...
}
```

The member function `SelectSubclass()` interpretes the requirement, which is passed as a character string, and returns the name of the selected subclass for that requirement.

Although the code shown above selects a subclass at compile time, it is also possible to postpone the final decision to runtime. For example, the user may want to use runtime information for the requirement. In this case, the metaobject translates a `new` expression into an expression that calls an appropriate runtime support function, which selects a subclass and returns its instance. This implementation is also available without the MOP, but if we use the MOP, interpretation of the requirement can be done at compile-time. For example, the metaobject may translate the requirement from a string to an appropriate data structure for the runtime support function to efficiently handle it.

5 Other Issues

This section briefly surveys several other issues on the OpenC++ MOP.

Inheritance

An interesting issue is whether a subclass should inherit the metaclass of its base class. This is important because selecting the metaclass is the main mechanism for directing what MOP-based customization programmers use. The OpenC++ MOP provides customizability on this issue as well.

Our MOP selects the metaclass of a given class `X` with the following algorithm:

1. If the class `X` has a base class, then call `ComputeMetaclassName()` on the metaobject for that base class, and select its resulting value for the metaclass of `X`. By default, this member function simply returns the same metaclass as that of the base class.
2. If the metaclass of `X` is explicitly specified by the programmer with the `metaclass` declaration, then select that metaclass.
3. Otherwise, select the default metaclass `Class`.

By the first rule, subclasses inherit the metaclass from their base class. This inheritance policy can be customized by programmers, however. They can redefine the member function `ComputeMetaclassName()` to customize the policy.

In the case where a class has more than one base class and the class metaobjects for them give different metaclasses, we simply raise a compilation error. Other researchers have proposed automatic derivation of a mixed-in metaclass in this case [5], but applying that idea to the OpenC++ MOP is not straightforward because combining the same member function of two metaclasses is not always possible.

Syntax extension

The OpenC++ MOP provides limited ability to extend the language syntax. The programmer can register new keywords, which can appear only in certain limited places: the modifiers of type names, class names, and the `new` operator. For example, the following code is available.

```
distributed class Point { ... };
lightweight Vector v;
p = require("sorted") new Set;
```

`distributed`, `lightweight` and `require` are registered keywords. These keywords are passed to a metaobject when a code fragment is translated. The metaobject can use those keywords to decide how to translate that code fragment. The keywords may be followed by some *meta* arguments. For example, `"sorted"` is an argument of the keyword `require`. The arguments are simply passed to the metaobject together.

Otherwise, the keyword registered by the programmer must appear as the member name in a member access expression. In this case, the parser recognizes the whole member access expression as a user-defined statement. For example,

```
Matrix big_matrix = ...;
big_matrix->foreach(row == 1) {
  element = 1.0;
}
```

`foreach` is a keyword. It must be followed by an arbitrary expression `(row == 1)`, and a statement { `element = 1.0` }. The expression may be a list of formal arguments, or it may take the `for` statement's style, which is (<*expression*>; <*expression*>; <*expression*>). The OpenC++ MOP itself does not specify any meaning to the interpretation of that `foreach` statement. The interpretation is responsibility of the class metaobject for `Matrix`, which may translate that statement to an appropriate statement of C++ as macros do in Lisp. The translated code may look like:

```
for(int row = 0;
    row < number_of_rows;
    ++row)
  for(int col = 0;
      col < number_of_columns;
      ++col)
    if(row == 1){
      big_matrix->element[row][col]
        = 1.0;
    }
```

Although this code is not efficient, the library implementor can write compile-time code to translate the statement into more efficient code if the condition such as `(row == 1)` matches a particular pattern. For example, the compile-time code may check whether the parse tree of the given condition represents a sequence of the identifier `row`, the symbol `==`, and an integer. If so, the compile-time code can remove the `for` loop on `row`.

Protocol Overheads

Since programs translated by the OpenC++ MOP invoke runtime support functions, the MOP may initially seem to involve runtime penalties. But this kind of penalty is not due to the MOP itself. It is due to the implementation scheme of the library. For example, in the persistent-object library, the translated program invokes a function `Load()` every member access. But the performance penalty

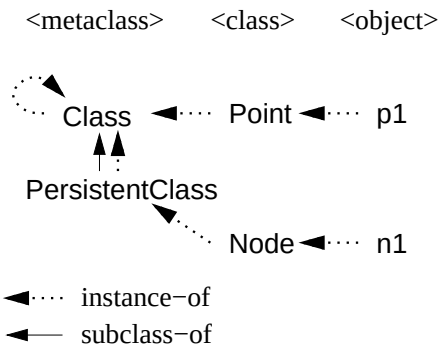


Figure 4: Metaclass, Class, and Object

with this function invocation is inherent in the implementation scheme we chose for the library. In fact, this function invocation is needed even if we do not use the MOP, as we saw in Section 2.

Although the OpenC++ MOP does not involve runtime penalties, it involves compile-time penalties since it moves meta-level computation from runtime to compile-time. However, at least regarding to the default metaobjects, the compile-time penalties can be reduced by elaborate implementation. The phase 2 and 3 of the protocol structure, which we showed in Section 3.2, are fused into very simple computation since the member function `CompileSelf()` on the default class metaobjects is an identity function that generates the same source code as the input.

Meta Circularity

The design of the OpenC++ MOP is, like other MOPs, conceptually meta-circular. There are no substantial differences between metaclasses and classes. A metaclass is simply a class that instantiates other classes (i.e. metaobjects for the classes). The relationship between a class and a metaclass is equivalent to the class-instance relationship (Figure 4). Thus when a program includes definitions of metaclasses, the MOP also constructs class metaobjects for those metaclasses. The constructed metaobjects control compilation of the metaclasses.

The OpenC++ MOP, however, avoids the appar-

ent infinite regress of this meta circularity in a way similar to that of other meta-circular MOPs. To compile a class, its metaclass must be first compiled, before compiling that metaclass, its metaclass must be compiled, and so on. But such a chain of compilation is not infinite because the metaclass `Class` is the root of any class-metaclass chain and it is the metaclass of itself. The only question we have to answer is how the MOP compiles `Class` for bootstrapping. The answer is simple. `Class` is directly compiled by the C++ compiler because the compilation specialized by `Class` is equivalent to the compilation done by the C++ compiler.

6 Related Work

In previous work, we have proposed another MOP for C++, called OpenC++ Version 1 [3]. That MOP had the ability to transparently implement language extensions for distributed computing as libraries on top of ordinary C++. But, because that MOP was based on a meta-architecture in which the metaobjects exist at runtime, it implied runtime overheads. Also, that MOP provided only limited ability to control program behavior, covering only member access and object creation. The degree of transparency of libraries written with that MOP was not enough for real-world programming.

A number of ideas for the OpenC++ MOP came from previous work. The idea of a compile-time MOP is due to Anibus and Intrigue [18, 13]. These are compile-time MOPs for controlling a Scheme compiler. In those MOPs, the metaobjects are not just language entities, but also represent global information such as the results of flow analysis. Our basic protocol architecture is due to the CLOS MOP[11]. The major difference is that the CLOS MOP's metaobjects are runtime ones and thus the CLOS MOP requires relatively large runtime environment if it is directly applied to C++. The idea of a meta-interface of the early-stage of compilation was also proposed in MPC++[8].

Like the CLOS MOP and OpenC++ Version 1, a number of systems [15, 7, 17, 22] adopt runtime metaobjects, which represent underlying mechanisms such as the language interpreter and the OS kernel, and are responsible for runtime behavior of the system. Since the runtime metaobjects allow users to change various decision policies of the system, such as scheduling and migration, the users can tune the system performance to fit their needs. A drawback of the runtime metaobjects is runtime overhead. A few ideas have been proposed on this problem [12, 16, 19]. For example, inlining and partial evaluation are effective techniques to reduce the overhead. It is difficult to recover the whole overhead of a runtime meta architecture, however.

7 Current Status

We are in the process of developing OpenC++ Version 2. Our methodology is to first develop a simplified version of the target system (i.e. C++), then design and test a MOP for that simplified system, and finally port the developed MOP back to the target system. We have thus developed a C++-like object system, called S++, on top of Scheme, and designed the MOP presented in this paper for S++. A number of examples similar to these presented here have been implemented to test the S++ MOP, and we have repeatedly re-designed the MOP based on the results of the tests. We are currently porting our MOP back to C++.

8 Conclusion

This paper describes a metaobject protocol for C++. It was developed to bring the power of metaprogramming to a more mainstream language. This MOP differs from most existing MOPs in that the metaobjects exist exclusively at compile-time — they control the compilation of programs to alter the behavior of the basic language constructs of C++. This feature means that this MOP involves no runtime speed or space overheads.

This paper also illustrates how the customizability provided by our C++ MOP can be used to implement language extensions efficiently and transparently as libraries. Currently, many language ex-

tensions such as persistence or distribution end up being re-implemented for each application because the existing language mechanisms are insufficient for customizing existing code to fit each application. Our C++ MOP enables us to implement those extensions as libraries, making such extensions easier to develop and maintain and thus more reusable.

Acknowledgments

The basic idea of the OpenC++ MOP was produced through discussions with Gregor Kiczales. The author thanks John Lamping, Ellen Siegel, and Chris Maeda for their comments on early drafts of this paper. The author's work was partially supported by Japan Society for the Promotion of Science.

References

[1] Angus, I. G., "Applications Demand Class-Specific Optimizations: The C++ Compiler Can Do More," *Scientific Programming*, vol. 2, no. 4, pp. 123–131, 1993.

[2] Buschmann, F., K. Kiefer, F. Paulisch, and M. Stal, "The Meta-Information-Protocol: Run-Time Type Information for C++," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 82–87, 1992.

[3] Chiba, S., "Open C++ Programmer's Guide," Technical Report 93-3, Dept. of Information Science, Univ. of Tokyo, Tokyo, Japan, 1993.

[4] Chiba, S. and T. Masuda, "Designing an Extensible Distributed Language with a Meta-Level Architecture," in *Proc. of the 7th European Conference on Object-Oriented Programming*, LNCS 707, pp. 482–501, Springer-Verlag, 1993.

[5] Danforth, S. and I. R. Forman, "Reflections on Metaclass Programming in SOM," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 440–452, 1994.

[6] Goldberg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[7] Honda, Y. and M. Tokoro, "Soft Real-Time Programming through Reflection," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 12–23, 1992.

[8] Ishikawa, Y., "Meta-Level Architecture for Extendable C++," Technical Report 94024, Real World Computing Partnership, Japan, 1994.

[9] Kiczales, G., "Towards a New Model of Abstraction in Software Engineering," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 1–11, 1992.

[10] G. Kiczales, ed., *Workshop on Open Implementation'94*, internet publication (`http://www.parc.xerox.com/PARC/spl/eca/oi/workshop-94`), Oct. 1994.

[11] Kiczales, G., J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[12] Kiczales, G. J. and L. H. Rodriguez Jr., "Efficient Method Dispatch in PCL," in *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 99–105, 1990.

[13] Lamping, J., G. Kiczales, L. Rodriguez, and E. Ruf, "An Architecture for an Open Compiler," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 95–106, 1992.

[14] Lortz, V. B. and K. G. Shin, "Combining Contracts and Exemplar-Based Programming for

Class Hiding and Customization," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 453–467, 1994.

[15] Maes, P., "Concepts and Experiments in Computational Reflection," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 147–155, 1987.

[16] Masuhara, H., S. Matsuoka, T. Watanabe, and A. Yonezawa, "Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 127–144, 1992.

[17] Okamura, H., Y. Ishikawa, and M. Tokoro, "AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 36–47, 1992.

[18] Rodriguez Jr., L. H., "Coarse-Grained Parallelism Using Metaobject Protocols," Techincal Report SSL-91-61, XEROX PARC, Palo Alto, CA, 1991.

[19] Ruf, E., "Partial Evaluation in Reflective System Implementation," in *Proc. of OOPSLA'93 Workshop on Reflection and Metalevel Architectures*, 1993.

[20] Steele, G., *Common Lisp: The Language*. Digital Press, 2nd ed., 1990.

[21] Watanabe, T. and A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 306–315, 1988.

[22] Yokote, Y., "The Apertos Reflective Operating System: The Concept and Its Implementation," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 414–434, 1992.

Load-Time Structural Reflection in Java

Shigeru Chiba

Institute of Information Science and Electronics
University of Tsukuba
and Japan Science and Technology Corp.
chiba@is.tsukuba.ac.jp, chiba@acm.org

Abstract. The standard reflection API of Java provides the ability to introspect a program but not to alter program behavior. This paper presents an extension to the reflection API for addressing this limitation. Unlike other extensions enabling behavioral reflection, our extension called *Javassist* enables structural reflection in Java. For using a standard Java virtual machine (JVM) and avoiding a performance problem, Javassist allows structural reflection only before a class is loaded into the JVM. However, Javassist still covers various applications including a language extension emulating behavioral reflection. This paper also presents the design principles of Javassist, which distinguish Javassist from related work.

1 Introduction

Java is a programming language supporting reflection. The reflective ability of Java is called the reflection API. However, it is almost restricted to introspection, which is the ability to introspect data structures used in a program such as a class. The Java's ability to alter program behavior is very limited; it only allows a program to instantiate a class, to get/set a field value, and to invoke a method through the API.

To address the limitations of the Java reflection API, several extensions have been proposed. Most of these extensions enable behavioral reflection, which is the ability to intercept an operation such as method invocation and alter the behavior of that operation. If an operation is intercepted, the runtime systems of those extensions call a method on a *metaobject* for notifying it of that event. The programmer can define their own version of metaobject so that the metaobject executes the intercepted operation with customized semantics, which implement a language extension for a specific application domain such as fault tolerance [9].

However, behavioral reflection only provides the ability to alter the behavior of operations in a program but not provides the ability to alter data structures used in the program, which are statically fixed at compile time (or, in languages like Lisp, when they are first defined). The latter ability called structural reflection allows a program to change, for example, the definition of a class, a function, and a record on demand. Some kinds of language extensions require this ability for implementation and thus they cannot be implemented with a straightforward

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 313–336, 2000.

program using behavioral reflection; complex programming tricks are often needed.

To simply implement these language extensions, this paper presents *Javassist*, which is a class library for enabling structural reflection in Java. Since portability is important in Java, we designed a new architecture for structural reflection, which can be implemented without modifying an existing runtime system or compiler. Javassist is a Java implementation of that architecture. An essential idea of this architecture is that structural reflection is performed by bytecode transformation at compile-time or load time. Javassist does not allow structural reflection after a compiled program is loaded into the JVM. Another feature of our architecture is that it provides source-level abstraction: the users of Javassist do not have to have a deep understanding of the Java bytecode. Our architecture can also execute structural reflection faster than the compile-time metaobject protocol used by OpenC++ [3] and OpenJava [20].

In the rest of this paper, we first overview previous extensions enabling behavioral reflection in Java and point out limitations of those extensions. Then we present the design of Javassist in Section 3 and show typical applications of Javassist in Section 4. In Section 5, we compare our architecture with related work. Section 6 is conclusion.

2 Extensions to the Reflection Ability of Java

The Java reflection API dose not provide the full reflective capability. It does not enable alteration of program behavior but it only supports introspection, which is the ability to introspect data structures, for example, inspecting a class definition. This design decision was acceptable because implementing the full capability was difficult without a decline in runtime performance. An implementation technique using partial evaluation has been proposed [17,2] but the feasibility of this technique in Java has not been clear.

However, several extensions to the Java reflection API have been proposed. To avoid performance degradation, most of these extensions enable restricted behavioral reflection. They only allow alteration of the behavior of specific kinds of operations such as method calls, field accesses, and object creation. The programmers can select some of those operations and alter their behavior. The compilers or the runtime systems of those extensions insert *hooks* in programs so that the execution of the selected operations is intercepted. If these operations are intercepted, the runtime system calls a method on an object (called *a metaobject*) associated with the operations or the target objects. The execution of the intercepted operation is implemented by that method. The programmers can define their own version of metaobject for implementing new behavior of the intercepted operations.

The runtime overheads due to this restricted behavioral reflection are low since only the execution of the intercepted operations involves a performance penalty and the rest of the program runs without any overheads. Especially, if hooks for the interception are statically inserted in a program during compila-

tion, the runtime overheads are even lowered. To statically insert hooks, Reflective Java [22] performs source-to-source translation before compilation and Kava [21] performs bytecode-level transformation when a program is loaded into the JVM. MetaXa [16,11] internally performs bytecode-level transformation with a customized JVM. It uses a customized just-in-time compiler (JIT) for improving the execution speed of the inserted hooks. This hook-insertion technique is well known and has been applied to other languages such as C++ [4].

Although the restricted behavioral reflection is useful for implementing various language extensions, there are some kinds of extensions that cannot be intuitively implemented with that kind of reflection. An example of these extensions is binary code adaptation (BCA) [13], which is a mechanism for altering a class definition in binary form to conform changes of the definitions of other classes. Suppose that we write a program using a class library obtained from a third party. For example, our class Calendar implements an interface Writable included in that class library:

```
class Calendar implements Writable {
  public void write(PrintStream s) { ... }
}
```

The class Calendar implements method `write()` declared in the interface Writable.

Then, suppose that the third party gives us a new version of their class library, in which the interface Writable is renamed into Printable and it declares a new method `print()`. To make our program conform this new class library, we must edit the definitions of all our classes implementing Writable, including Calendar:

```
class Calendar implements Printable {
  public void write(PrintStream s) { ... }
  public void print() { write(System.out); }
}
```

The interface of Calendar is changed into Printable and method `print()` is added.

BCA automates this adaptation; it automatically alters class definitions in binary form according to a configuration file specifying how to alter them. Note that the method body of `print()` is identical among all the updated classes since `print()` can be implemented with the functionality already provided by `write()` for the old version. If that configuration file is supplied by the library developer, we can run our program without concern about evolution of the class library.

Unfortunately, implementing BCA with behavioral reflection is not intuitive or straightforward. Since behavioral reflection cannot directly provide the ability to alter data structures such as a class definition or construct a new data structure, these reflective computation must be indirectly implemented. For example, the implementation of BCA with behavioral reflection defines a metaobject indirectly performing the adaptation specified by a given configuration file. For the above example, this metaobject is made to be associated with Calendar and it

watches method calls on Calendar objects. If the method `print()` is called, the metaobject intercepts that method call and executes the computation corresponding to `print()` instead of the Calendar object. The metaobject also intercepts runtime type checking so that the JVM recognizes Calendar as a subtype of Printable. Recall that Java is a statically typed language and the original Calendar is a subtype of Writable.

The ability to alter data structures used in a program is called structural reflection, which has not been directly supported by previous systems. Although a number of language extensions are more easily implemented with structural reflection than with behavioral reflection, the previous systems have not been addressing those extensions. They have been too much focused on language extensions that can be implemented by altering the behavior of method calls and so on.

3 Javassist

To simply implement language extensions like BCA shown in the previous section, we developed Javassist, which is our extension to the Java reflection API and enables structural reflection instead of behavioral one. Javassist is based on our new architecture for structural reflection, which can be implemented without modifying an existing runtime system or a compiler.

3.1 Implementations of Structural Reflection

Structural reflection is the ability to allow a program to alter the definitions of data structures such as classes and methods. It has been provided by several languages such as Smalltalk [10], ObjVlisp [6], and CLOS [14]. These languages implement structural reflection with support mechanisms embedded in runtime systems. Since the runtime systems contain internal data representing the definitions of data structures such as a class, the support mechanisms allow a program to directly read and change those internal data and thereby execute structural reflection on the correspondent data structures.

We could not accept this implementation technique for Javassist since it needs to modify a standard JVM but portability is important in Java. Furthermore, a naive application of this technique to Java would cause serious performance degradation of the JVM because this technique makes it difficult for runtime systems to employ optimization techniques based on static information of executed programs. Since a program may be altered at runtime, efficient dynamic recompilation is required for redoing optimization on demand. For example, method inlining is difficult to perform. If an inlined method is altered at runtime with structural reflection, all the inlined code must be updated. To do this, the runtime system must record where the code is inlined. This will spend a large amount of memory space. Another example is the "v-table" technique used for typical C++ implementations [8]. This technique statically constructs method dispatch tables so that invoked methods are quickly selected with a constant

offset in the tables. If a new method is added to a class at runtime, then the dispatch tables may be updated and all offsets in the tables may be recomputed. Since the dynamic recompilation technique has been used so far for gradually optimizing "hot spots" of compiled code at runtime [12], it has been assuming that a program is never changed at runtime. Effectiveness of dynamic recompilation without this assumption is an open question.

Another problem is correctness of types. Since Java is a statically typed language, a variable of type X must be bound to an object of X or a subclass Y of X. If a program can freely access and change the internal data of the JVM, it may dynamically change the super class of Y from X to another class. This change causes a type error for the binding between a variable of type X and an object of Y. To address this problem, extra runtime type checks or restrictions on the range of structural reflection are needed.

3.2 Load-Time Structural Reflection

To avoid the problems mentioned above, we designed a new architecture for structural reflection; it does not need to modify an existing runtime system or a compiler. On the other hand, it enables structural reflection only before a program is loaded into a runtime system, that is, at load time. Javassist is a class library enabling structural reflection based on this architecture. In Java, the bytecode obtained by compilation of a program is stored in *class files*, each of which corresponds to a distinct class. Javassist performs structural reflection by translating alterations by structural reflection into equivalent bytecode transformation of the class files. After the transformation, the modified class files are loaded into the JVM and then no alterations are allowed after that. Thereby, Javassist can be used with a standard JVM, which may use various optimization techniques.

Javassist is used with a user class loader. Java allows programs to define their own versions of class loader, which fetch a class file from a not-standard resource such as a network. A typical definition of the class loader is as follows:

```
class MyLoader extends ClassLoader {
  public Class loadClass(String name) {
    byte[] bytecode = readClassFile(name);
    return resolveClass(defineClass(bytecode));
  }

  private byte[] readClassFile(String name) {
    // read a class file from a resource.
  }
}
```

The methods `defineClass()` and `resolveClass()` are inherited from ClassLoader. They request the JVM to load a class constructed from the bytecode given as an array of `byte`. The returned value is a Class object representing the loaded class. Once a class X is manually loaded with an instance of MyLoader, all classes referenced by that class X are loaded through that class loader. The

JVM automatically calls `loadClass()` on that class loader for loading them on demand.

Javassist helps `readClassFile()` shown above obtain the bytecode of a requested class. It can be regarded as a class library for reading bytecode from a class file and altering it. However, unlike similar class libraries such as the JavaClass API [7] and JOIE [5], Javassist provides source-level abstraction so that it can be used without knowledge of bytecode or the data format of the class file. Also, Javassist was designed to make it difficult to wrongly produce a class file rejected by the bytecode verifier of the JVM.

3.3 The Javassist API

We below present the overview of the Javassist API.

Reification and Reflection: The first step of the use of Javassist is to create a CtClass (compile-time class) object representing the bytecode of a class loaded into the JVM. This step is for reifying the class to make it accessible from a program. If `stream` is an InputStream for reading a class file (from a local disk, memory, a network, etc.), then:

```
CtClass c = new CtClass(stream);
```

creates a new CtClass object representing the bytecode of the class read from the class file, which contains enough symbolic information to reify the class. Also, the constructor of CtClass can receive a String class name instead of an InputStream. If a String class name is given, Javassist searches a class path and finds an InputStream for reading a class file.

One can call various methods on the CtClass object for introspecting and altering the class definition. Changes of the class definition are reflected on the bytecode represented by that object. To obtain the bytecode for loading the altered class into the JVM, method `toBytecode()` is called on that object:

```
byte[] bytecode = c.toBytecode();
```

Loading the obtained bytecode into the JVM is regarded as the step for reflecting the CtClass object on the base level. Javassist provides several other methods for this step. For example, method `compile()` writes bytecode to a given output stream such as a local file and a network. Method `load()` directly loads the class into the JVM with a class loader provided by Javassist. It returns a Class object representing the loaded class. Recall that Class is included in the Java reflection API while CtClass is in Javassist.

Note that Javassist does not provide any framework for specifying how and what classes are processed with Javassist. The programmer of the class loader has freedom with respect to this framework. For example, the class loader may process classes with Javassist only if they are specified by a configuration file read at the beginning. It may process them according to a *hard-coded* algorithm.

Table 1. Methods in CtClass for introspection

Method	Description
`String getName()`	gets the class name
`int getModifiers()`	gets the class modifiers such as `public`
`boolean isInterface()`	determines whether this object represents a class or an interface
`CtClass getSuperclass()`	gets the super class
`CtClass[] getInterfaces()`	gets the interfaces
`CtField[] getDeclaredFields()`	gets the fields declared in the class
`CtMethod[] getDeclaredConstructors()`	gets the constructors declared in the class
`CtMethod[] getDeclaredMethods()`	gets the methods declared in the class

Javassist allows a user class loader to define a new class from scratch without reading any class file. This is useful if a program needs to dynamically define a new class on demand. To do this, a CtClass object must be created as follows:

```
CtClass c2 = new CtNewClass();
```

The created object `c2` represents an empty class that has no methods or fields although methods and fields can be added to the class later through the Javassist API shown below. If `toBytecode()` is called on this object, then it returns the bytecode corresponding to that empty class.

Introspection: Javassist provides several methods for introspecting the class represented by a CtClass object. This part of the Javassist API is compatible with the Java reflection API except that Javassist does not provide methods for creating an instance or invoking a method because these methods are meaningless at load time. Table 1 lists selected methods for introspection.

CtClass objects returned by `getSuperclass()` and `getInterfaces()` are constructed from class files found on a class path. They represent the original class definitions and thus accept only introspection but not alteration. To alter a class, another CtClass object must be explicitly created with the `new` operator. Modifications to this object have no effect on the CtClass object returned by `getSuperclass()` or `getInterfaces()`. For example, suppose that a class C inherits from a class S. If a CtClass object for S is created with `new` and a method `m()` is added to that object, this modification is not reflected on the object returned by `getSuperclass()` on a CtClass object for C. The class C inherits `m()` from S only if the CtClass object created with `new` is converted into bytecode and loaded into the JVM.

The information about fields and methods is provided by objects separate from the CtClass object; it is provided by CtField objects obtained by `getDeclaredFields()` and CtMethod objects obtained by `getDeclaredMethods()`, respectively. The information about a constructor is also provided by a CtMethod object. Table 2 lists methods in CtField and CtMethod for introspection.

Table 2. Methods in CtField and CtMethod for introspection

Method	in CtField	Description
`String`	`getName()`	gets the field name
`CtClass`	`getDeclaringClass()`	get the class declaring the field
`int`	`getModifiers()`	gets the field modifiers such as `public`
`CtClass`	`getType()`	get the field type
Method	in CtMethod	Description
`String`	`getName()`	gets the method name
`CtClass`	`getDeclaringClass()`	get the class declaring the method
`int`	`getModifiers()`	gets the method modifiers such as `public`
`CtClass[]`	`getParameterTypes()`	gets the types of the parameters
`CtClass[]`	`getExceptionTypes()`	gets the types of the exceptions that the method may throw
`boolean`	`isConstructor()`	returns `true` if the method is a constructor
`boolean`	`isClassInitializer()`	returns `true` if the method is a class initializer

Table 3. Methods for alteration

Method	in CtClass	Description
`void`	`bePublic()`	make the class `public`
`void`	`beAbstract()`	make the class `abstract`
`void`	`notFinal()`	remove the `final` modifier from the class
`void`	`setName(String name)`	change the class name
`void`	`setSuperclass(CtClass c)`	change the super class
`void`	`setInterfaces(CtClass[] i)`	change the interfaces
`void`	`addConstructor(...)`	add a new constructor
`void`	`addDefaultConstructor()`	add the default constructor
`void`	`addAbstractMethod(...)`	add a new `abstract` method
`void`	`addMethod(...)`	add a new method
`void`	`addWrapper(...)`	add a new wrapped method
`void`	`addField(...)`	add a new field
Method	in CtField	Description
`void`	`bePublic()`	make the field `public`
Method	in CtMethod	Description
`void`	`bePublic()`	make the method `public`
`void`	`instrument(...)`	modify a method body
`void`	`setBody(...)`	substitute a method body
`void`	`setWrapper(...)`	substitute a method body

Alteration: A difference between Javassist and the standard Java reflection API is that Javassist provides methods for altering class definitions. Several methods for alteration are defined in CtClass (Table 3). These methods are categorized into methods for changing class modifiers, methods for changing class hierarchy, and methods for adding a new member. They were carefully selected to satisfy our design goals.

Our design goals are three. (1) The first goal is to provide source-level abstraction for programmers. Javassist was designed so that programmers can use it without knowledge of the Java bytecode. (2) The second goal is to execute structural reflection as efficiently as possible. (3) The last goal is to help programs perform structural reflection in a safe manner in terms of types.

As for the first goal, the most significant design decision was how programmers specify a method body. Suppose that a new method is added to a class. If a sequence of bytecode is used for specifying the body of that method, the programmers would get great flexibility but have to learn details of bytecode. To achieve the first goal, Javassist allows to copy a method body from another existing method although this design decision restricts the flexibility of the added method. The copied bytecode sequence is adjusted to fit the destination method. For example, the bytecode for accessing a member through the `this` variable contains a symbolic reference to the type of `this`. This reference is replaced with one to the class declaring the destination method.

Despite the well-known quasi-equivalence between Java source code and bytecode, the correspondence between source-level and bytecode-level alterations are not straightforward. Hiding the gap between the two levels from programmers is also a part of the first goal.

For example, `setName()` renames a class but it also substitutes the new name for all occurrences of the old name in the definition of that class, including method signatures and bodies. Modifying a single constant-pool item never performs this substitution. If a constructor calls another constructor in the same class (if it executes `this()`), then the bytecode of the former constructor is modified since the bytecode contains a symbolic reference to the name of the class declaring the latter constructor. This reference must be modified to indicate the new name.

`setSuperclass()` performs similar substitution. If it is called, all occurrences of the old super class name is replaced with a new name and all constructors are modified so that they call a constructor in the new super class. However, there is an exception to this substitution. If the name of the original super class is java.lang.Object (the root of the class hierarchy), `setSuperclass()` does not perform the substitution except it modifies constructors. This is because java.lang.Object is often used for representing any class. For example, although `addElement()` in java.util.Vector takes a parameter of class java.lang.Object, which is the super class of java.util.Vector, this never means that `addElement()` takes an instance of the super class.

The second design goal is to reduce overheads due to class loading with Javassist. Since we will use Javassist for implementing a mobile-agent system, in which Javassist inserts security-check code into bytecode, Javassist must transform bytecode received through a network as efficiently as possible. Mobile agents frequently move among hosts and thus we cannot ignore the loading time of the bytecode implementing the mobile agents.

Our design decision on how programmers specify a method body was influenced by the second goal as well as the first one. Javassist does not use source code

for specifying the body of an added method. If source code is used, it must be compiled *on the fly* when a class is loaded into the JVM. A naive implementation of this source-code approach would produce a complete class definition including the added method at source level and then compile it with a Java compiler such as `javac`. As we show later, however, this implementation implies serious performance penalties. To achieve practical efficiency, we need a special compiler that can quickly compile only a method body. We did not adopt the source-code approach because of limitations of our resources. Instead, Javassist allows to copy a pre-compiled method body from a class to another. This approach does not imply overheads due to source-code compilation at load time.

The third design goal is to prevent programs to wrongly produce a class including type incorrectness. To achieve this goal, Javassist allows only limited kinds of alteration of class definitions. In general, reflective systems should impose some restrictions on structural reflection so that programs do not falsely collapse themselves with reflection. Suppose that a reflective system allows to remove a field from a class at runtime. If there are already instances of that class, is it appropriate that the system simply discards the value of the removed field of those instances?

Since erroneous bytecode produced with Javassist is rejected by the bytecode verifier, it can never damage the JVM. However, restricting the reflective capability of Javassist is still necessary because it is often awkward to correct a program producing erroneous bytecode. For this reason, Javassist does not provide methods for removing a method or a field from a class because they cause type incorrectness if there is a method accessing the removed method or field. Javassist also imposes restrictions on the class passed to `setSuperclass()`, which is a method for changing a super class. The new super class must be a subclass of the original super class since there may be methods that implicitly cast an instance of that class to the original super class. Of course, the new super class must not be `final`. Furthermore, Javassist does not provide a method for changing the parameters of a method. Programmers are recommended to add a new method with the same name but with different parameters.

Adding a new member: Javassist provides methods for adding a new method to a class. To avoid the abstraction and performance problems mentioned above, `addMethod()` receives a `CtMethod` object, which specifies a method body. The signature of `addMethod()` is as shown below:

```
void addMethod(CtMethod m, String name, ClassMap map)
```

`name` specifies the name of the added method. The method body is copied from a given method `m`. Since a method body is copied from an existing compiled method, no source-code compilation is needed at load time or no raw bytecode is given to `addMethod()`. Programmers can describe a method body in Java and compile it in advance. Javassist reads the bytecode of the compiled method and adds it to another class. This improves execution performance of Javassist since a compiler is not run at load time.

When a method body is copied, some class names appearing in the body can be replaced according to a hash table `map`.[1] For example, programmers can declare a class XVector:

```
public class XVector extends java.util.Vector {
  public void add(X e) {
    super.addElement(e);
  }
}
```

and copy the method `add()` into a class StringVector:

```
CtMethod m = /* method add() in XVector */;
CtClass c = /* class StringVector */;
ClassMap map = new ClassMap();
map.put("X", "java.lang.String");
c.addMethod(m, "addString", map);
```

The class name java.lang.String is substituted for all occurrences of the class name X in `add()`. The added method is as follows:

```
public void addString(java.lang.String e) {
  super.addElement(e);
}
```

Javassist provides another method `addWrapper()` for adding a new method. It allows more generic description of a method body:

```
void addWrapper(int modifiers, CtClass returnType, String name,
                CtClass[] parameters, CtClass[] exceptions,
                CtMethod body, ConstParameter constParam)
```

The first five parameters specify the modifiers, the return type, the method name, the parameter types, and the exceptions that the method may throw. The body of the added method is copied from the method specified by `body`. No matter what the signature of the added method is, the method specified by `body` must have the following signature:

```
Object m(Object[] args,  value-type constValue)
```

To fill the gap between this signature and the signature of the added method, `addWrapper()` implicitly wraps the copied method body in *glue* code, which constructs an array of actual parameters passed to the added method and assigns it to `args` before executing the copied method body. The glue code also sets `constValue` to a constant value specified by `constParam` passed to `addWrapper()`. In the current version of Javassist, an integer value or a String

[1] At least, `addMethod()` replaces all occurrences of the name of the class declaring the copied method. Even if that class name does not appear at source level, the corresponding bytecode may include references to it.

object can be specified for the constant value. For example, this constant value can be used to pass the name of the added method.

The value returned by the copied method body is an Object object. The glue code also converts it into a value of the type specified by `returnType`. Then it returns the converted value to the caller to the added method. If type conversion fails, then an exception is thrown. Although methods added by `addWrapper()` involve runtime overheads due to type conversion, a single method body can be used as a template of multiple methods receiving a different number of parameters. Examples of the use of `addWrapper()` are shown in Section 4.

Javassist also provides a method for adding a new field to a class:

```
void addField(int modifiers, CtClass type, String fieldname,
              String accessor, FieldInitializer init)
```

If `accessor` is not `null`, this method also adds an accessor method, which returns the value of the added field. The name of the accessor is specified by `accessor`. Moreover, the last parameter `init` specifies the initial value of the added field. The initial value is either one of parameters passed to a constructor, a newly created object, or the result of a call to a static method.

Altering a method body: Although Javassist does not allow to remove a method from a class, it provides methods for changing a method body. `setBody()` and `setWrapper()` in CtMethod substitute a given method body for an original body:

```
void setBody(CtMethod m, ClassMap map)
void setWrapper(CtMethod m, ConstParameter param)
```

They correspond to `addMethod()` and `addWrapper()` respectively. `setBody()` copies a method body from a given method `m`. Some class names appearing in the body are replaced with different names according to `map`. `setWrapper()` also copies a method body from `m` but it wraps the copied body in glue code. The signature of `m` must be:

```
Object m(Object[] args,  value-type constValue)
```

Javassist also provides a method for modifying expressions in a method body. `instrument()` in CtMethod performs this modification:

```
void instrument(CodeConverter converter)
```

The parameter `converter` specifies how to instrument a method body. The CodeConverter object can perform various kinds of instrumentation. Table 4 lists methods provided by the current implementation of Javassist. They direct a CodeConverter object to replace a specific kind of expressions with *hooks*, which invoke static methods for executing the expressions in a customized manner. The idea of CodeConverter came from C++'s operator overloading. CodeConverter was

Table 4. Methods in CodeConverter

Method	Description
`void redirectFieldAccess()`	change a field-access expression to access a different field.
`void replaceNew()`	replace a `new` expression with a `static` method call.
`void replaceFieldRead()`	replace a field-read expression with a `static` method call.
`void replaceFieldWrite()`	replace a field-write expression with a `static` method call.

designed for safely altering the behavior of operators such as `new` and `.` (dot) independently of the context.

For example, expressions for instantiating a specific class can be replaced with expressions for calling a static method. Suppose that variables `xclass` and `yclass` represent class X and Y, respectively. Then a program:

```
CtMethod m = ... ;
CodeConverter conv = new CodeConverter();
conv.replaceNew(xclass, yclass, "create");
m.instrument(conv);
```

instruments the body of the method represented by the CtMethod object `m`. All expressions for instantiating the class X such as:

```
new X(3, 4);
```

are translated into expressions for calling a static method `create()` declared in the class Y:

```
Y.create(3, 4);
```

The parameters to the `new` expression are passed to the static method.

Reflective class loader: The class loader provided by Javassist allows a loaded program to control the class loading by that class loader. If a program is loaded by Javassist's class loader L and it includes a class C, then it can intercept the loading of C by L to self-reflectively modify the bytecode of C (Figure 1). For avoiding infinite recursion, while the loading of a class is intercepted, further interception is prohibited. The `load()` method in CtClass requires that a program is loaded by Javassist's class loader although the other methods work without Javassist's class loader.

Java's standard class loader never allows this self-reflective class loading for security reasons. If it is allowed, a program may change some `private` fields to `public` ones at load time for reading hidden values. Furthermore, in Java, if a program creates a class loader and loads a class C with that class loader, the loaded class is regarded as a different one from the class denoted by the name C appearing in that program. The latter class is loaded by the class loader that loaded the program.

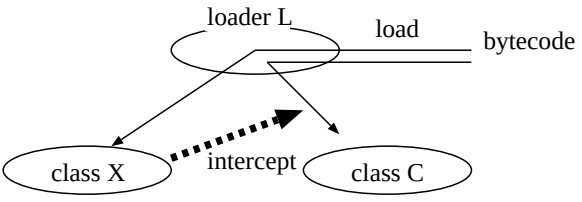


Fig. 1. Javassist's class loader allows self-reflective class loading

Using Javassist without a class loader: Javassist can be used without a user class loader. There are three kinds of usage of Javassist: with a user class loader, with a web server, and off line.

For security reasons, an applet is usually prohibited from using a user class loader. However, we can write an applet working with Javassist if we use a web server as a replacement of a user class loader. Since classes used in an applet are loaded from a web server into the JVM of a web browser, we can customize the web server so that it runs Javassist for processing the classes before sending them to the web browser. Javassist includes a simple web server written in Java as a basis for such customization. We can extend it to perform structural reflection with Javassist. The program of the customized web server would be as follows:

```
for (;;) {
  receive an http request from a web browser.
  CtClass c = new CtClass(the requested class);
  do structural reflection on c if needed.
  byte[] bytecode = c.toBytecode();
  send the bytecode to the web browser.
}
```

Before sending a requested class to a web browser, it performs structural reflection on the class according to the algorithm, for example, given as a configuration file.

Another usage of Javassist is "off line". We can perform structural reflection on a class and overwrite the original class file of that class with the bytecode obtained as the result. The altered class can be later loaded into the JVM without a user class loader. The following is an example of the off-line use of Javassist:

```
CtClass c = new CtClass("Rectangle");
do structural reflection on c if needed.
c.compile();      // writes bytecode on the original class file.
```

This program performs structural reflection on class Rectangle and overwrites the class file of that class with the bytecode obtained by `c.toBytecode()`.

4 Examples

This section shows three applications of Javassist. We illustrate that Javassist can be used to implement non-trivial alteration required by these applications despite the level of the abstraction.

4.1 Binary Code Adaptation

The mechanism of binary code adaptation (BCA) [13] automatically alters class definitions according to a file written by the users, called a delta file:

```
delta class implements Writable {
  rename Writable Printable;
  add public void print() { write(System.out); }
}
```

This delta file specifies adaptation that we mentioned in Section 2.

If Javassist is used, the implementor of BCA has only to write a parser of delta file and a user class loader performing adaptation with Javassist. For example, the parser translates the delta file shown above into the Java program shown below:

```
class Exemplar implements Printable {
  public void write(PrintStream s) { /* dummy */ }
  public void print() { write(System.out); }
}

class Adaptor {
  public void adapt(CtClass c) {
    CtMethod printM = /* method print() in Exemplar */;
    CtClass[] interfaces = c.getInterfaces();
    for (int i = 0; i < interfaces.length; ++i)
      if (interfaces[i].getName().equals("Writable")) {
        interfaces[i] = CtClass.forName("Printable");
        c.setInterfaces(interfaces);
        c.addMethod(printM, new ClassMap());
        return;
      }
  }
}
```

The class Exemplar is compiled together with Adapter in advance so that `adapt()` can obtain a `CtMethod` object representing `print()`. `adapt()` uses the reification and introspection API of Javassist for obtaining it. It first constructs a `CtClass` object representing Exemplar and then obtains the `CtMethod` object by `getDeclaredMethods()` in CtClass. The class file for Exemplar is automatically found by Javassist on the class path used for loading Adapter.

The user class loader calls `adapt()` in Adaptor whenever a class is loaded into the JVM. It creates a CtClass object representing the loaded class and calls `adapt()` with that object. The method `adapt()` performs adaptation if the

loaded class implements Writable. Then the user class loader converts the CtClass object into bytecode and loads into the JVM.

Note that this implementation is more intuitive than the implementation with behavioral reflection. Moreover, it is simpler than the implementation without reflection since the implementor does not have to care about low-level bytecode transformation. If the users of BCA can directly write the classes Exemplar and Adaptor instead of a delta file, then the implementation would be much simpler since we do not need the parser of delta file.

4.2 Behavioral Reflection

Behavioral reflection enabled by MetaXa [16,11] and Kava [21] can be implemented with an approximately 750-line program (including comments) using Javassist. A key idea of their implementations is to insert *hooks* in a program when a class is loaded into the JVM. We below see an overview of a user class loader performing this insertion with Javassist.

Let a metaobject be an instance of MyMetaobject, which is a subclass of Metaobject:

```
public class MyMetaobject extends Metaobject {
  public Object trapMethodcall(String methodName, Object[] args) {
    /* called if a method call is intercepted. */ }
  public Object trapFieldRead(String fieldName) {
    /* called if the value of a field is read. */ }
  public void trapFieldWrite(String fieldName, Object value) {
    /* called if a field is set. */ }
}
```

If field accesses and method calls on an instance of C:

```
public class C {
  public int m(int x) { return x + f; }
  public int f;
}
```

are intercepted by the metaobject, then the user class loader alters the definition of the class C into the following:[2]

```
public class C implements Metalevel {
  public int m(int x) { /* notify a metaobject */ }
  public int f;
  private Metaobject _metaobject = new MyMetaobject(this);
  public Metaobject _getMetaobject() { return _metaobject; }
  public int orig_m(int x) { return x + f; }
  public static int read_f(Object target) {
      /* notify a metaobject */ }
  public static void write_f(Object target, int value) {
      /* notify a metaobject */ }
}
```

where the interface Metalevel declares the method `_getMetaobject()`.

[2] For simplicity, this implementation ignores `static` members although extending the implementation for handling `static` members is possible within the ability of Javassist.

```
class Exemplar {
  private Metaobject _metaobject;

  public Object trap(Object[] args, String methodName) {
    return _metaobject.trapMethodcall(methodName, args);
  }

  public static Object trapRead(Object[] args, String name) {
    Metalevel target = (Metalevel)args[0];
    return target._getMetaobject().trapFieldRead(name);
  }

  public static Object trapWrite(Object[] args, String name) {
    Metalevel target = (Metalevel)args[0];
    Object value = args[1];
    target._getMetaobject().trapFieldWrite(name, value);
  }
}
```

Fig. 2. Class Exemplar

This alteration can be performed within the ability of Javassist. The interface Metalevel is added by `setInterfaces()` in CtClass. The field `_metaobject` and the accessor `_getMetaobject()` are added by `addField()` in CtClass.

For intercepting method calls, the user class loader first makes a copy of every method in C by calling `addMethod()` in CtClass. For example, it adds `orig_m()`[3] as a copy of `m()`. Then it replaces the body of every method in C with a copy of the body of the method `trap()` in Exemplar (see Figure 2). This modification is performed by `setWrapper()` in CtMethod. The gap between the signatures of `m()` and `trap()` is filled by `setWrapper()`. The substituted method body notifies a metaobject of interception. The first parameter `args` is a list of actual parameters and the second one `name` is the name of the copy of the original method such as `"orig_m"`. These two parameters are used for the metaobject to invoke the original method through the Java reflection API.

For intercepting field accesses, the user class loader instruments the bodies of methods in all classes. All accesses to a field `f` in C are translated into calls to a `static` method `read_f()` or `write_f()`. This instrumentation is performed by `instrument()` in CtMethod and `replaceFieldRead()` and `replaceFieldWrite()` in CodeConverter. The methods `read_f()` and `write_f()` notify a metaobject of the accesses. They are added by `addWrapper()` in CtClass as copies of `trapRead()` and `trapWrite()` in Exemplar. The gap between the signatures of `read_f()` (or `write_f()`) and `trapRead()` (or `trapWrite()`) is filled by `addWrapper()`. For example, actual parameters to `read_f()` are converted into the first parameter `args` to `trapRead()`. The second parameter `name` to `trapRead()` is the name of the accessed field such as `"f"`.

[3] If a method name is overloaded, a copy of each method must be given a different name such as `orig_m1()`, `orig_m2()`, ...

4.3 Remote Method Invocation

Generating stub code for remote method invocation is another application of Javassist. A Java program cannot directly call a method on a remote object on a different computer. It needs the Java RMI tools generating stub code, which translates a method call into lower-level network data transfer such as TCP/IP communication. However, the Java RMI tools are compile-time ones; a program must be processed by the RMI compiler, which generates and saves stub code on a local disk. Also, a program using the Java RMI must be subject to a protocol (i.e. API) specified by the Java RMI.

Javassist allows programmers to develop their own version of the RMI tools, which specify a customized protocol and produce stub code at either compile-time or even runtime. Suppose that an applet needs to call a method on a Counter object on a web server written in Java. For remote method invocation, the applet needs stub code defining a proxy object of the Counter object, which has the same set of methods as the Counter object. If the Counter object has a method `setCount()`, the proxy object also has a method `setCount()` with the same signature. However, the method on the proxy object serializes given parameters and sends them to the web server, where `setCount()` is invoked on the Counter object with the received parameters.

This stub code can be generated at runtime with Javassist at the server side and it can be sent on demand to the applet side. The applet programmer can easily write the applet without concern about low-level network programming. The stub code for accessing the Counter object is as follows:

```
public class ProxyCounter {
  private RmiStream rmi;
  public ProxyCounter(int objectRef) {
    rmi = new RmiStream(objectRef);
  }
  public int setCount(int value) { /* remote method invocation */ }
}
```

An instance of ProxyCounter is a proxy object. An RmiStream object handles low-level network communication. The class RmiStream is provided by a runtime support library.

ProxyCounter can be defined within the confines of Javassist. The field `rmi` is added by `addField()` in CtClass and the initialization of `rmi` in a constructor can be specified by a FieldInitializer object passed to `addField()`.

The method `setCount()` is added by `addWrapper()` in CtClass as a copy of the method `invoke()` in Exemplar shown below:

```
class Exemplar {
  private RmiStream rmi;
  Object invoke(Object[] args, String methodName) {
    return rmi.rpc(methodName, args);
  }
}
```

The gap between the signatures of `setCount()` and `invoke()` is filled by `addWrapper()`. If `setCount()` is called, the actual parameter `value` is converted into an array of Object and assigned to `args`. `methodName` is set to a method name `"setCount"`[4]. Then `rpc()` is called on the RmiStream object for serializing the given parameters and sends them to the web server. Note that the parameters can be serialized within the ability of the standard Java if they are converted into an array of Object.

Stub code generation is another example, which is not straightforward to implement with behavioral reflection. In a typical implementation with behavioral reflection, a proxy object is an instance of the class Counter although all method calls on the proxy object are intercepted by a metaobject and forwarded to a remote object; the class ProxyCounter is not produced. Therefore, if the proxy object is created, a constructor declared in Counter is called and may cause fatal side-effects since the class Counter is defined as a class at the server side but the proxy object is not at that side.

5 Related Work

Reflection in Java: MetaXa [16,11] and Kava [21] enable behavioral reflection in Java whereas Javassist enables structural reflection. They are suitable for implementing different kinds of language extensions. However, Javassist indirectly covers applications of MetaXa and Kava since a class loader providing functionality equivalent to MetaXa and Kava can be implemented with Javassist as we showed in Section 4.2.

Although Kava performs bytecode transformation of class files before the JVM loads them as Javassist does, they only insert hooks for interception in bytecode but do not run metaobjects at that time. They enable reflection at runtime and their ability is not structural reflection but the restricted behavioral reflection.

The Java reflection API was recently extended in the JDK 1.3 beta to partially enable behavioral reflection [19]. The new API allows a program to dynamically define a proxy class implementing given interfaces. An instance of this proxy class delegates all method invocations to another object through a type-independent interface.

Javassist is not the first system enabling structural reflection in Java. For example, Kirby et al proposed a system enabling structural reflection (they called it linguistic reflection) in Java although their system only allows to dynamically define a new class but not to alter a given class definition at load time [15]. With their system, a Java program can produce a source file of a new class, compile it with an external compiler such as `javac`, and load the compiled class with a user class loader. They reported that their system could be used for defining a class optimized for a given runtime condition.

[4] If a method name is overloaded, this should be `setCount1`, `setCount2`, ... for distinction.

Compile-time metaobject protocol: The compile-time metaobject protocol [3] is another architecture enabling structural reflection without modifying an existing runtime system. OpenJava [20] is a Java implementation of this architecture. As Javassist does, it restricts structural reflection within the time before a class is loaded into the JVM although it was designed mainly for off-line use at compile time. However, OpenJava is source-code basis although Javassist is bytecode basis; OpenJava reads source code for creating an object representing a class, a method, or a field. Alteration to the object is translated into corresponding transformation of the source code. The bytecode for the altered class is obtained by compiling the modified source code. Since OpenJava is source-code basis, it can deal with syntax extensions within a framework of structural reflection. For example, one can extend the syntax of class declaration and make it possible to add an annotation to a class declaration.

On the other hand, the source-code basis means that OpenJava needs the source file of every processed class whereas Javassist needs only a class file (compiled binary). This is a disadvantage because source files are not always available if the class is provided by a third party. OpenJava also involves a performance overhead due to handling source code; the source file of every class must be parsed for reification and compiled for reflection. Although this overhead is compensation for the capability for fine-grained transformation of source code (including syntax extension), it is not negligible if OpenJava is used by a class loader for altering a loaded class. Some kinds of applications such as a mobile agent system do not need fine-grained transformation but fast class loading.

Although the implementations of OpenJava or Javassist have not been tuned up, the performance difference between OpenJava and Javassist is notable with respect to reification and reflection. If a class loader can be implemented with either OpenJava or Javassist, Javassist achieves shorter loading time. To show this performance difference, we compared Javassist and OpenJava with two small applications. We implemented BCA [5] and behavioral reflection presented in Section 4 with both Javassist and OpenJava and we measured the time needed for altering a given class with each implementation. For fair comparison, the implementations with Javassist write modified class files back on a local disk.

Table 5 lists the results. The execution time is the average of five continuous repetitions, which do not include the first repetition. Since a program is gradually loaded into the JVM during the first repetition, the first one is tremendously slow. For compiling a modified source file, OpenJava uses a compiler provided by the Sun JDK for Solaris. However, it never uses the `javac` command since it starts the compiler in a separate process; instead, it directly runs the compiler (`sun.tools.javac`) on the same JVM.

Although the sizes of the programs implementing the applications are almost equal between Javassist and OpenJava, Javassist processed a class more than ten times faster than OpenJava. Note that the execution time by Javassist is shorter than the time needed only for compiling a modified source file. This is because

[5] Of course, the implementation of BCA with OpenJava does not modify a class file in binary form. It emulates equivalent adaptation at source-code level.

Table 5. Performance comparison between Javassist and OpenJava

		execution time (msec)	program size (lines)	original source (lines)	original class file (bytes)	modified class file (bytes)
BCA	Javassist	42	26	24	372	551
	OpenJava	543 (172†)	17	24		548
Reflection	Javassist	142	205	35	946	3932
	OpenJava	4108 (302†)	247	35		2244

Sun JDK 1.2.2 (HotSpot VM 1.0.1), UltraSPARC II 300MHz
†compilation time by `sun.tools.javac` (Java compiler).

Javassist can move compilation penalties to an earlier stage. Even a method body is not compiled while Javassist is running; it is pre-compiled in advance and the resulting bytecode is directly copied to a target class at run time.

Bytecode translators: Bytecode translators such as JOIE [5] and the JavaClass API [7] provide a functionality similar to Javassist. They enable a Java program to alter a class definition at load time. However, they are toolkits for directly dealing with bytecode, that is, the raw data structure of a class file. For example, classes included in JOIE are ClassInfo, Code, and Instruction. They show that JOIE was designed for experienced programmers who have a deep understanding of the Java bytecode and want to implement complex transformation. On the other hand, Javassist was designed to be easy to use; it does not require programmers to have knowledge of the Java bytecode but instead it provides source-level abstraction for manipulating bytecode in a relatively safe manner. Although a range of instrumentation of a method body is restricted, we showed that Javassist can be used to implement non-trivial applications. Javassist can be regarded as a front end for easily and safely using a bytecode translator like JOIE; it is not a replacement of the bytecode translators.

Using bytecode instrumentation for implementing a reflective facility is a known technique in Smalltalk [1]. A uniqueness of Javassist against this is the design of the API providing source-level abstraction. The Javassist API was carefully designed to avoid wrongly producing a class definition containing type incorrectness.

Others: OpenJIT [18] is a just-in-time compiler that allows a Java program to control how bytecode are compiled into native code. It provides better flexibility than Javassist with respect to instrumenting a method body while OpenJIT does not allow to add a new method or field to a class. However, using OpenJIT is more difficult than using Javassist because OpenJIT requires programmers to have knowledge of both the Java bytecode and native code. Although OpenJIT can be used without knowledge of the Java bytecode if programmers use a me-

chanism of OpenJIT for translating bytecode into a parse tree of an equivalent Java program, overheads due to that translation has not been reported.

The idea of enabling reflection only at load time for avoiding performance problems is found in the CLOS MOP [14]. For example, the CLOS MOP allows a program to alter the algorithm of determining the super classes of a given class but the super classes are statically determined when the class is loaded; the program cannot dynamically change the super classes at runtime.

Some readers may think that Javassist is very similar to BCA. However, Javassist was designed for a wider range of applications than BCA, which is specialized for on-line class adaptation. BCA only allows to modify a given class but not to dynamically define a new class from scratch. On the other hand, BCA allows programmers to describe the algorithm of adaptation in declarative form.

6 Conclusion

This paper presented Javassist, which is an extension to the Java reflection API. Unlike other extensions, it enables structural reflection in Java; it allows a program to alter a given class definition and to dynamically define a new class. A number of language extensions are more easily implemented with structural reflection than with behavioral reflection.

For avoiding portability and performance problems, the design of Javassist is based on our new architecture for structural reflection. Javassist performs structural reflection by instrumenting bytecode of a loaded class. Therefore, it can be used with a standard JVM and compiler although structural reflection is allowed only before a class is loaded into the JVM, that is, at load time. Since a standard JVM is used, the classes processed by Javassist are subject to the bytecode verifier and the SecurityManager of Java. Javassist never breaks security guarantees given by Java.

The followings are important features of Javassist:

- Javassist is portable. It is implemented in only Java without native methods and it runs with a standard JVM. It does not need a platform-dependent class library. Portability is significant in Java programming.
- Javassist provides source-level abstraction for manipulating bytecode in a safe manner while bytecode translators, such as JOIE [5] and the JavaClass API [7], provide no higher-level abstraction. The users of Javassist do not have to have a deep understanding of the Java bytecode or to be careful for avoiding wrongly making an invalid class rejected by the bytecode verifier.
- Javassist never needs source code whereas OpenJava [20], which is another system for structural reflection with source-level abstraction, does. Since OpenJava performs structural reflection by transforming source code, it must parse and compile source code for reifying and reflecting a class. Thus a class loader using Javassist can load a class faster than one using OpenJava. However, OpenJava enables fine-grained manipulation of class definitions so that the resulting definitions may be smaller and more efficient than ones by Javassist.

The architecture that we designed for Javassist can be applied to other object-oriented languages if a compiled binary program includes enough symbolic information to construct a class object. However, the API must be individually designed for each language so that it allows a program to alter class definitions in a safe manner with respect to the semantics of that language.

Acknowledgment. The author thanks Michiaki Tatsubori, who wrote programs using OpenJava for the experiment in Section 5. He also thanks Hidehiko Masuhara for his comments on an early draft of this paper, and the anonymous reviewers for their helpful comments.

References

1. Brant, J., B. Foote, R. E. Johnson, and D. Roberts, "Wrappers to the Rescure," in *ECOOP'98 - Object Oriented Programming*, LNCS 1445, pp. 396–417, Springer, 1998.
2. Braux, M. and J. Noyé, "Towards Partially Evaluating Reflection in Java," in *Proc. of Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, SIGPLAN Notices vol. 34, no. 11, pp. 2–11, ACM, 1999.
3. Chiba, S., "A Metaobject Protocol for C++," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, SIGPLAN Notices vol. 30, no. 10, pp. 285–299, ACM, 1995.
4. Chiba, S. and T. Masuda, "Designing an Extensible Distributed Language with a Meta-Level Architecture," in *Proc. of the 7th European Conference on Object-Oriented Programming*, LNCS 707, pp. 482–501, Springer-Verlag, 1993.
5. Cohen, G. A., J. S. Chase, and D. L. Kaminsky, "Automatic Program Transformation with JOIE," in *USENIX Annual Technical Conference '98*, 1998.
6. Cointe, P., "Metaclasses are first class: The ObjVlisp model," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 156–167, 1987.
7. Dahm, M., "Byte Code Engineering with the JavaClass API," Techincal Report B-17-98, Institut für Informatik, Freie Universität Berlin, January 1999.
8. Ellis, M. and B. Stroustrup, eds., *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.
9. Fabre, J. C. and T. Pérennou, "A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 78–95, 1998.
10. Goldberg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
11. Golm, M. and J. Kleinöder, "Jumping to the Meta Level, Behavioral Reflection Can Be Fast and Flexible," in *Proc. of Reflection '99*, LNCS 1616, pp. 22–39, Springer, 1999.
12. Hölzle, U. and D. Ungar, "A Third Generation Self Implementation: Reconciling Responsiveness with Performance," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, SIGPLAN Notices vol. 29, no. 10, pp. 229–243, 1994.
13. Keller, R. and U. Hölzle, "Binary Component Adaptation," in *ECOOP'98 - Object Oriented Programming*, LNCS 1445, pp. 307–329, Springer, 1998.

14. Kiczales, G., J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol*. The MIT Press, 1991.
15. Kirby, G., R. Morrison, and D. Stemple, "Linguistic Reflection in Java," *Software – Practice and Experience*, vol. 28, no. 10, pp. 1045–1077, 1998.
16. Kleinöder, J. and M. Golm, "MetaJava: An Efficient Run-Time Meta Architecture for Java," in *Proc. of the International Workshop on Object Orientation in Operating Systems (IWOOS'96)*, IEEE, 1996.
17. Masuhara, H. and A. Yonezawa, "Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Languages," in *ECOOP'98 - Object Oriented Programming*, LNCS 1445, pp. 418–439, Springer, 1998.
18. Ogawa, H., K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and F. Kimura, "OpenJIT : An Open-Ended, Reflective JIT Compiler Framework for Java," in *Proc. of ECOOP'2000*, Springer Verlag, 2000. To appear.
19. Sun Microsystems, "Java^{TM} 2 SDK Documentation." version 1.3 (beta release), 1999.
20. Tatsubori, M., S. Chiba, M.-O. Killijian, and K. Itano, "OpenJava: A Class-based Macro System for Java," in *Reflection and Software Engineering* (W. Cazzola, R. J. Stroud, and F. Tisato, eds.), LNCS 1826, Springer Verlag, 2000.
21. Welch, I. and R. Stroud, "From Dalang to Kava — The Evolution of a Reflective Java Extension," in *Proc. of Reflection '99*, LNCS 1616, pp. 2–21, Springer, 1999.
22. Wu, Z., "Reflective Java and A Reflective-Component-Based Transaction Architecture," in *Proc. of OOPSLA'98 Workshop on Reflective Programming in C++ and Java* (J.-C. Fabre and S. Chiba, eds.), 1998.

Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment

Jean-Pierre BRIOT

Equipe Mixte LITP - RXF,
Université Pierre et Marie Curie,
4 place Jussieu, 75005 Paris, France
briot@litp.univ-p6-7.fr.uucp

Abstract

This paper describes a system for designing and classifying actor languages. This system, named *Actalk*, which stands for *act*ors in Small*talk*-80, is based on some minimal kernel introducing actors into Smalltalk-80. The Actalk kernel extends passive and sequential objects activated by synchronous message passing into active and concurrent actors communicating by asynchronous message passing. This defines a sub-world of actors embedded into the Smalltalk-80 programming language and environment, which is left unchanged. The Actalk kernel is composed of only two Smalltalk-80 classes. Simulating current actor languages or designing new ones is achieved by defining subclasses of these two kernel classes. Consequently all such extensions are implicitly classified by the inheritance mechanism and unified into the Actalk environment. We are currently extending the standard Smalltalk-80 programming environment to design a specific one dedicated to Actalk concurrent actors.

In this paper, the motivations and the goals which led to design the Actalk system are first discussed. Then the structure and implementation of the kernel of Actalk is detailed. The very good integration of this kernel into the Smalltalk-80 programming language and environment is demonstrated through some examples. Interests and limits of combining objects with actors are then discussed. In the last part of the paper, we demonstrate the expressive power of the Actalk kernel by extending it to simulate the two main actor computation models and programming languages, namely, the Actor model of computation, and the Abcl/1 programming language. Related and further work is summarized before concluding this paper.

Keywords

object, Smalltalk-80, concurrency, ConcurrentSmalltalk, actor, kernel, integration, modularity, combination, Act*, Abcl/1, classification, environment

1 INTRODUCTION

This paper is concerned with the design and experiment of Object-Based Concurrent Programming (OBCP) languages. We believe that the so-called *actor* family of languages is among the most promising approaches. The theory of actors was invented by Carl Hewitt [Hewitt 77]. Plasma was the first language designed along this philosophy. It was followed by Act1 [Lieberman 81], Act2 [Theriault 83], and the latest and most achieved prototype: Pract/Acore [Manning 87], based on the latest computation model called the *Actor computation model* defined by Gul Agha [Agha 86]. Besides this stream of prototypes designed around Carl Hewitt at the AI Lab of MIT, the actor metaphor gave rise to other OBCP proposals based on the original actors, although changing some aspects of the model. All these *actor*, or *actor-based* languages keep the foundation of active and concurrent objects communicating asynchronously. One of the most representative element is the Abcl/1 language proposed by Akinori Yonezawa [Yonezawa et al. 86].

We remarked that, from the user point of view, it is not always easy to analyze and compare all these proposals. Comparing various semantics needs to abstract from various syntax and implementation supports. Moreover some of the prototypes are not usable outside of some labs or very specific machines. From the implementor point of view, experience (including our personal one) of designing actor languages shows that usually too many prototype implementations are thrown away before clarifying the design. Making a prototype modular (and improving its efficiency) is usually achieved *afterwards* and not in the early implementation effort. The reuse of previous prototypes or even other ones would greatly improve the design task. This goal of modularity is endorsed by Object-Oriented Programming (OOP). This led us recently to design some modular system/environment for actor languages based on OOP, in order to satisfy these needs. We will now discuss how we designed it.

2 GOALS AND DESIGN DECISIONS

To design a system for integrating various actor languages in a single environment, we had the following goals in mind:

1. *uniformity and modularity*
   We wanted to unify various actor languages into some common environment and to be able to analyze and define them step by step. Therefore we chose the object-oriented paradigm as a basis for matching these two first goals. We decided to introduce actors into traditional OOP, by defining a sub-world of actors embedded into the world of objects, and without changing the underlying object system.

2. *minimality and extensibility*
   We wanted some minimal kernel expressing the most general semantics of actor languages, and to further extend it in order to simulate various existing languages and to design new ones. Therefore we chose a minimal architecture

in the spirit of ObjVlisp [Briot and Cointe 87,Cointe 87] which is based on a minimal kernel, and then further uniformly extended. We use inheritance to classify the various actor models.

3. *an integrated environment*
We didn't want to restrict our system to some semantic model and raw implementation, but also to provide a full environment for pragmatic experiment with actor-oriented programming. Therefore we chose Smalltalk-80 because it is the most achieved and flexible OOP system with a fully integrated programming environment. Because of the integration of our actors into the Smalltalk-80 model and environment, the designer of actor languages could automatically use the standard Smalltalk-80 programming environment. Furthermore, this environment could be extended in order to support the specific concurrency aspects of actors.

By choosing Smalltalk-80 we could also provide a minimal implementation of the system, because all entities needed to build actors: objects, classes and messages, and to express concurrency: processes and semaphores, are provided by Smalltalk-80. The resulting system is named *Actalk* (which stands for *act*ors in Small*talk*-80), as a spiritual offspring of *ObjVlisp* (*obj*ects in *v*irtual *Lisp*). Actalk is an integrated environment, embedded into Smalltalk-80, used as a testbed to classify and design actor languages. Actalk does not change the underlying Smalltalk-80 system, but rather merges into it. Because of the optimal integration of actors within objects, Actalk may also be used as a basis for studying relation and combination between objects and actors. We are currently using Actalk in graduate courses to introduce and teach actor-oriented programming to object-oriented programmers.

3 FROM SMALLTALK-80 OBJECTS TO ACTORS

We will now shortly introduce the model of actors, and how we embed them into Smalltalk-80. We will focus on *why* and *how* to introduce them into a standard OOP model, namely Smalltalk-80, which is extended towards concurrency.

In Smalltalk-80, as in standard OOP, objects are activated by message passing. Objects are passive because they undergo the request of activation by the sender of the message. They have no activity of their own. Sending a message represents the transfer of activity from one object to another one, before going back to the sender whose activation is suspended.

To achieve concurrency, Smalltalk-80 introduces multiple activations, called processes. Then multiple messages may activate concurrently several objects. But if a single object receives concurrently two messages, the two activations may compete to control the object. If both methods assign concurrently a same variable of the object, the resulting state cannot be predicted and is inconsistent. The problem is that an object is still passive, therefore it should be protected against multiple simultaneous activations.

The simplest and most pragmatic solution to this problem is to ensure the single activation condition, also called *mutual exclusion*. Such a technique is described in Smalltalk-80 by [Pascoe 86]. A further solution ensures this condition by providing its own activity to the object. This changes the model of computation from passive into active objects. An object will now possess its own internal activity, and will no more depend on external activations through message passing. The object becomes *active* and *autonomous*. It gained the ability to decide on its own when to perform a message and has the power to complete it.

Because the receiver has its own activity to perform the message, the sender does not need any more to suspend its activity to transfer it to the receiver. If no reply is needed, the sender may resume its computation immediately after sending the message. Message passing becomes unidirectional and asynchronous. Thus, sending a message reduces to its delivery to the receiver. The receiver may process it any time after the delivery, therefore messages should be buffered in a *mailbox* associated to the receiver, before being processed.

The initial model of passive objects synchronously activated by messages has been extended towards a model of concurrent active objects communicating asynchronously by passive messages. We will call such entities *actors*. An actor will encapsulate a standard Smalltalk-80 object to make it active and to change the semantics of message passing. An actor is composed of a mailbox where the incoming messages will be buffered, and the active object that will process them, which we will call its *behavior*.

4 IMPLEMENTATION OF THE ACTALK KERNEL

4.1 Definition of the Minimal Kernel

We will now define the most minimal and general kernel for our Actalk system. We will call it the *Actalk kernel*. Two classes define its semantics. The class `Actor` defines the semantics of actors. The class `ActorBehavior` defines the semantics of behaviors (of actors).

The class `Actor` simply defines an actor through its two components, the mailbox or queue which will contain the incoming messages, and the behavior which will process them. The implementation of asynchronous message passing led us to define a third class, presented in section 4.4, in order to clearly separate the implementation technique required from the semantics of the kernel.

Designing the class `ActorBehavior` needs careful attention. We want to define the most general semantics of how a behavior processes messages. The Actor computation model of Gul Agha states that a behavior processes only one message and has to specify which behavior will process next message. This generalizes the usual assignment of instance variables in OOP languages. The Actor computation model is minimal and general enough to express any other kind of computation model, as discussed in [Agha 86]. However we do not choose it as the candidate for the kernel, because we

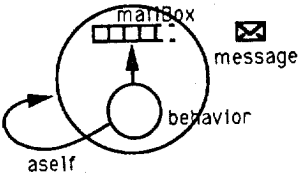


Figure 1: Implementation of an actor in Actalk.

want an extension of the Smalltalk-80 language with minimal implementation and maximal integration within Smalltalk-80. We would like the programmer to specify behaviors of actors as he would specify behaviors of standard Smalltalk-80 objects. Therefore we must keep usual variable assignment to specify state changes. Because there could be state changes during the processing of a message, messages have to be *serialized* [Hewitt and Atkinson 79], i.e., processed one at a time.

We believe that the basic semantics of behaviors we choose is minimal *and* general enough. It expresses the default semantics of the activity of behaviors, and it can be extended or redefined by the extensions of the kernel. For instance the Actor computation model of Gul Agha is expressed as an immediate extension of the kernel (one method is redefined and another one is added), as shown in section 7.1.

Figure 1 gives a representation of the implementation of an actor. All Smalltalk-80 source for Actalk will be printed in the standard *file Out* format, where the underline character _ means assignment.

4.2 The class Actor

The class `Actor` implements actors as encapsulators built around standard Smalltalk-80 objects. Its superclass is the class `MinimalObject` which will be used for the implementation of asynchronous message passing and is described in section 4.4. It specifies two instance variables:

`mailbox` denotes the queue of messages, an instance of the standard class `SharedQueue`,

`behavior` denotes the behavior which processes the messages, an instance of a subclass of the class `ActorBehavior` described in next section.

```
MinimalObject subclass: #Actor
   instanceVariableNames: 'mailBox behavior '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Kernel'!
```

```
!Actor methodsFor: 'initialization'!

initialize
   mailBox _ SharedQueue new!

initializeBehavior: aBehavior
   behavior _ aBehavior.
   behavior initializeAself: self! !

!Actor methodsFor: 'access to instance variables'!

mailBox
   ^mailBox! !

!Actor methodsFor: 'message passing'!

asynchronousSend: aMessage
   mailBox nextPut: aMessage! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

!Actor class methodsFor: 'instance creation and initialization'!

behavior: aBehavior
   ^self new initializeBehavior: aBehavior!

new
   ^super new initialize! !
```

The `behavior:` class method creates an actor and initializes its behavior (`initializeBehavior:` instance method). The `new` class method is redefined to initialize the mailbox of the actor (`initialize` instance method). We designed this decomposition in two distinct initialization protocols in order to improve modularity and reuse of the kernel when extending this basic implementation (but this won't be shown in this paper). The `asynchronousSend:` instance method implements asynchronous message passing to an actor by enqueueing the corresponding message onto its mailbox.

4.3 The class ActorBehavior

The class `ActorBehavior` implements the general behavior of an actor. Users will describe behaviors of actors by defining classes of behaviors as subclasses of `ActorBehavior`. The class `ActorBehavior` defines one instance variable: `aself`, which denotes the actor currently using this behavior. `aself` allows an actor to send (asynchronous) recursive messages to itself. The difference with standard pseudo variable `self` will be explained in section 4.4.

A background process will be created with the actor to implement the activity and autonomy of the behavior. This process is infinite. As stated in the definition of the kernel, the behavior will keep dequeueing the next message from the mailbox and perform it.

```
Object subclass: #ActorBehavior
   instanceVariableNames: 'aself '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Kernel'!

!ActorBehavior methodsFor: 'initialization'!

initializeAself: anActor
   aself _ anActor.
   self setProcess!

setProcess
   [[true] whileTrue: [self acceptNextMessage]] fork! !

!ActorBehavior methodsFor: 'message acceptance'!

acceptNextMessage
   self acceptMessage: aself mailBox next!

acceptMessage: aMessage
   self performMessage: aMessage! !

!ActorBehavior methodsFor: 'actor creation'!

actor
   ^Actor behavior: self! !
```

The `initializeAself:` instance method initializes the self-reference (`aself`) of the actor and starts the activity of the behavior (`setProcess` method). The `acceptNextMessage` and `acceptMessage:` methods accept and perform the next message in the mailbox. The process is suspended until the mailbox is not empty by the semaphore which synchronizes reading data from the shared queue [Goldberg and Robson 83, page 262]. The `actor` method creates an actor from a passive object which is used as its behavior. It is the only method that the end user has to know about the kernel, i.e., the public interface, as shown in section 5.1.

Note that the Actalk kernel is stated in just 2 classes and 11 small methods. The number of methods could have been further reduced, but as already stated in previous section, we chose this modular decomposition to easily describe further extensions. For the same kind of reason, the implementation primitive `performMessage:` method is defined in class `Object`:

```
!Object methodsFor: 'message passing'!

performMessage: aMessage
   ^self perform: aMessage selector withArguments: aMessage arguments! !
```

4.4 Transparent Asynchronous Message Passing

Local Redefinition of Message Passing Semantics Now we would like to integrate harmoniously our new message passing model (`asynchronousSend:` method) between actors with the current message passing syntax of Smalltalk-80. We use a technique initiated in Smalltalk-80 in order to change the syntax (to introduce compound selectors for multiple inheritance [Ingalls and Borning 82]) or the semantics (to encapsulate objects [Pascoe 86]) of message passing through redefinition of error semantics. A message is sent to an object which, on purpose, does not recognize the selector. This error is trapped by redefinition of the standard error method (`doesNotUnderstand:`) which then executes some specific strategy. Local redefinition of the `doesNotUnderstand:` method ensures the locality of changes. Thus we redefine it in the class `Actor` in order to redefine the semantics of message passing *locally* to actors:

```
!Actor methodsFor: 'message passing'!

doesNotUnderstand: aMessage
   self asynchronousSend: aMessage! !
```

A message sent to an actor will get an asynchronous semantics whereas the same message sent to some standard Smalltalk-80 object will keep the standard synchronous semantics. Note that the value returned by asynchronous transmission is not significant. (Actually the value of the expression is the receiver of the message, due to the Smalltalk-80 convention.)

When specifying the behavior of an actor which sends recursive messages, the programmer may choose between *pseudo*-variables `aself` or `self`. Sending to `aself` uses Actalk asynchronous message passing. The message is put in the mailbox of the actor, and the behavior will process it later. Sending to `self` relies on standard Smalltalk-80 synchronous message passing. This implies immediate and "internal" processing of the message by the behavior.

Implementation By using this error redefinition technique we assume that an actor won't recognize the selector of the message it receives because we *do* want to trigger an error, in order to put the message in the mailbox. (Note that, in contrast, the "Actalk implementation methods" defined in class `Actor` will be directly processed.) But methods defined in class `Object` are inherited by every class, consequently such messages sent to an actor won't fail as expected.

[Pascoe 86] and [McCullough 87] discuss various implementation strategies to ensure the assumption of unrecognized messages. Our simplified solution is to define another root of the inheritance tree, besides the class `Object`, in order to bypass it. Actually the method dictionary of this new class should not be fully empty, because a minimal set of system methods is needed to trap errors, print, compare, and inspect their instances. Otherwise the Smalltalk-80 interpreter and environment would not be able to

manage properly such objects. Unfortunately the standard Smalltalk-80 environment disallows the user to define a class without superclass. Therefore the implementation trick is to define the new class, named `MinimalObject`, at first as a subclass of `Object`. Then its initial inheritance link is automatically removed (`superclass _ nil`), and a minimal set of system methods belonging to `Object` is copied (`recompile:from:`) onto it.

```
Object subclass: #MinimalObject
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Kernel-Encapsulator'!

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

!MinimalObject class methodsFor: 'initialization'!

initialize
   superclass _ nil.
   #(doesNotUnderstand: error:   "" isNil   = ==   printString printOn: class
     inspect basicInspect basicAt: basicSize instVarAt: instVarAt:put:)
        do: [:selector | self recompile: selector from: Object]! !

MinimalObject initialize!
```

5 A FIRST EXAMPLE: THE COUNTER

Our first example will be one of the most simple and paradigmatic examples of object-oriented programming: the counter. This will show how well Actalk actors are integrated into the Smalltalk-80 language. The class `Counter` will describe the behavior of counter actors, i.e., which behave as counters.

5.1 Definition of the Counter

The class `Counter` defines two instance methods to reset (`reset`), and increment (`incr`) the instance variable `contents` of a counter. In the actor terminology, these two methods constitute the *script* of the actor.

```
ActorBehavior subclass: #Counter
   instanceVariableNames: 'contents
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Examples'!

!Counter methodsFor: 'script'!

incr
   contents _ contents + 1!
```

```
reset
   contents _ 0! !
```

We defined this class of behaviors of actors-counters, exactly as we usually define the class of objects-counters. `Counter` must be defined as a subclass of class `ActorBehavior`. We may create an instance of `Counter` as some usual Smalltalk-80 counter object postfixed with the selector `actor`, and send (implicitly asynchronous) messages to this newly created actor:

```
| aCounter |
aCounter _ Counter new actor.
aCounter reset; incr; incr
```

Notice that the selector `actor` is the only special keyword to create actors. Definition and message passing are transparent within the Smalltalk-80 language into which actors are embedded. Some difference in programming style will be when returning values, as we will see in next section.

5.2 Concept of Reply Destination: the Printer Example

Now suppose that we want to consult the contents of the counter and display it for instance. But, due to the asynchronous nature of message passing to actors, we cannot rely any more on the returned value of a message as in standard Smalltalk.

The intuitive idea is to simulate bidirectional transmission by a second unidirectional message as reply, i.e., to specify within the message the actor to which the reply will be returned. Such an actor is called a *customer* [Agha 86], or *reply destination* [Yonezawa et al. 86]. Reply destinations are also used to implement *continuations* which is one of the main concept of programming with actors [Hewitt 77], but won't be addressed in this paper. We will specify a reply destination when consulting the contents of a counter with the following method:

```
!Counter methodsFor: 'script'!

consultAndReplyTo: replyDestination
   replyDestination reply: contents! !
```

We assume that every actor (or even object, see section 7.2) used as a reply destination handles the selector `reply:`, convention for replying the value. For instance, some standard Actalk actor which is bound to the global variable `Print`, displays values in the Smalltalk-80 Transcript window. Its behavior is defined by the class `Printer`:

```
ActorBehavior subclass: #Printer
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Examples'!

!Printer methodsFor: 'script'!

reply: value
   Transcript show: '> ' , value printString; cr! !
```

Following is an example of use:

```
Counter new actor
   reset; incr; incr; consultAndReplyTo: Print
```

which displays:

```
> 2
```

6 SYMBIOSIS BETWEEN OBJECTS AND ACTORS

Historically, classes and objects were proposed by Simula and Smalltalk to describe abstract and concrete concepts. Actors were proposed by Plasma to describe control structures and concurrency. Actalk appears as a proposal to combine both, i.e., extend classes and objects towards concurrency, or/and give a structure and an environment to describe actors. Because the Actalk sub-world of actors is fully integrated into the Smalltalk-80 language, actors may send messages to objects and vice versa. Thus the two programming styles may be combined.

6.1 Safety of Combination

One of the motivation for introducing actors in the object world of Smalltalk-80 was to automatically solve the inconsistencies between objects and processes (discussed in section 3). However some unrestricted combination between objects and actors may see these problems reoccur. If several actors happen to share a single passive object, the situation will be equivalent to processes sharing an object.

One drastic and ultimate way to solve the problems is to remove passive objects and to make every Smalltalk-80 object become an actor. ConcurrentSmalltalk-II [Yokote and Tokoro 87] walked a step in this direction and reduced some of the problems by changing parts of the Smalltalk-80 system. We did not choose this way because we did not want to change Smalltalk-80 in any way.

Another way that we chose is to provide safety rules. Some of the rules are ensured by the Actalk implementation and user interface (for instance, the only way to create an actor). But some methodological rules are also necessary as compromises between too strict rules forbidding any reuse of standard objects and too weak rules leading to havoc. For instance, Smalltalk-80 classes may be safely approximated to constant objects. Consequently they may stand concurrent activations. (In the actor terminology, a Smalltalk-80 constant object is similar to an *unserialized* actor.)

6.2 Extending the Smalltalk-80 Environment towards Actors

Because Actalk actors are well integrated into the Smalltalk-80 system, they automatically benefit from the standard Smalltalk-80 programming environment, which is a great help when designing languages and applications. A further goal is to extend this standard environment to support the specificity of actors. A first step is to extend the Smalltalk-80 MVC model for interface design towards actors. The prototype basic extension of MVC that we designed allows to control representations (views) of an actor during its activation. Another challenge is to extend the current Smalltalk-80 debugger towards a specific debugger for actors. A first prototype has already been implemented. It relies on extended messages which contain the context of the sender to reconstruct the appropriate chain of contexts.

7 EXTENDING THE KERNEL TO SIMULATE VARIOUS ACTOR LANGUAGES

Now we will sketch some extensions to our actor kernel in order to simulate some of the most representative OBCP computation models and programming languages based on the actor concept. Such simulations are not concerned about a complete reimplementation of some programming language environment, but to express its most essential and specific characteristics. These extensions will use inheritance to refine the semantics of the Actalk kernel. The first example will express Agha's Actor computation model as a subclass of `ActorBehavior`, whereas the second example will express Yonezawa's Abcl/1 model as a subclass of `Actor`. This shows the merits of modularity for our kernel. Because the kernel and its extensions are related by inheritance, one could easily compare them. Inheritance helps not only to classify various actor models, but also to clearly relate and to reuse their various implementations.

Figure 2 shows the hierarchy of the classes of the Actalk kernel, augmented by some of its current extensions, and the example of the counter. Note that the class `ActorBehavior` and all its extensions are *abstract classes*, i.e., don't have instances. These classes only give the semantics of how to compute messages, not the semantics of messages themselves. Only the application classes, like `Counter`, will generate actual instances, the behaviors. Actors will be instances of class `Actor` and its subclasses. The classes `ExtendedActor` and `ExtendedActorBehavior` introduce a generic control of actor events (i.e., receiving a message, computing it...) into Actalk. They are currently used to modularly change the semantics of the underlying Smalltalk-80

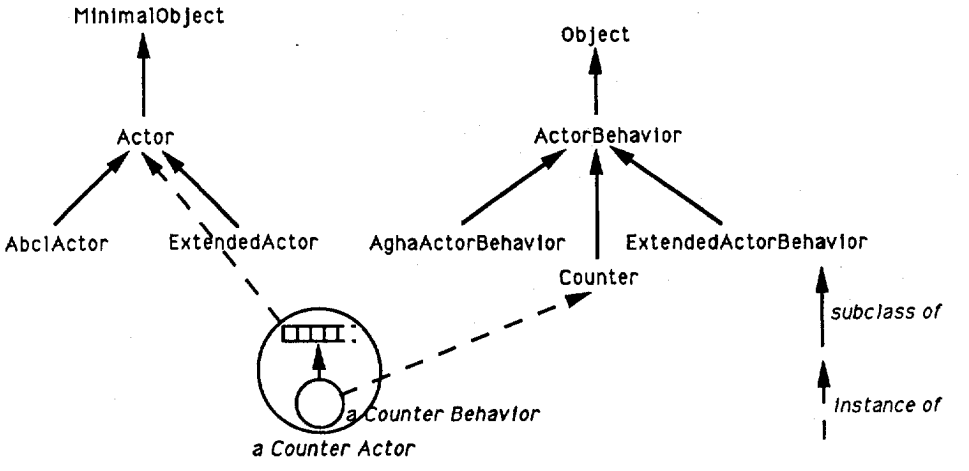


Figure 2: Architecture of the Actalk system (kernel and few extensions).

scheduler of processes, and design actor event driven representations (views) of actors (through a combination with our Actalk/MVC interface). Their complete description is found in [Briot 88].

7.1 The Actor Model of Computation

The Concept of Behavior Replacement The Actor model of computation, as exposed in [Agha 86], replaces state change (assignment) with a much higher level concept: *behavior change*. When performing a message, the current behavior of an actor will specify its *replacement behavior*, i.e., how it will perform next incoming message. A behavior will now accept only one message. The replacement behavior in turn, on accepting the next message, will specify its own replacement behavior. This leads to a causally connected ordered chain of behaviors isomorphic to the queue of messages. The two important points to highlight are the *absence of assignment* and the *separation between the successive behaviors*. As a consequence they may execute concurrently.

Implementation To change accordingly the semantics of behaviors, we introduce a new class, named `AghaActorBehavior`, as a subclass of `ActorBehavior`:

```
ActorBehavior subclass: #AghaActorBehavior
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Extensions-Agha'!
```

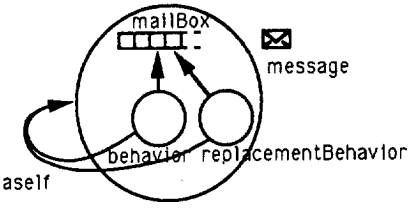


Figure 3: The concept of behavior replacement.

```
!AghaActorBehavior methodsFor: 'initialization'!

setProcess
   [self acceptNextMessage] fork! !

!AghaActorBehavior methodsFor: 'behavior replacement'!

replace: replacementBehavior
   aself initializeBehavior: replacementBehavior! !
```

Only two instance methods define our extension. The `setProcess` method is redefined, and now accepts only one message in the message queue. We introduce a new method, named `replace:`, whose semantic is to specify the replacement behavior (and to initialize it).

This redefinition slightly changes the role of the instance variable named `behavior` and defined in class `Actor`. It now represents the *current behavior* of an actor. When performing a behavior replacement this variable will be reassigned (by the `initializeBehavior:` method of `Actor`) to the *replacement behavior*. The current behavior won't be touched, thus it will complete its current computation. It will then be garbage-collected by the system because it is no longer referenced by the actor and the process is also terminated. Figure 3 shows this new model of actors.

Example We easily redefine the class of counters (defined in section 5.1) as a subclass of `AghaActorBehavior` and name it `AghaCounter`. Previous assignment will be replaced by the specification of a replacement behavior. (We suppose the existence of the `contents:` class method to create and initialize a new counter.) Remark that the `consultAndReplyTo:` method, although not changing state, needs to specify a replacement behavior (equal to the current one) in order to process next incoming message.

```
AghaActorBehavior subclass: #AghaCounter
   instanceVariableNames: 'contents '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Extensions-Agha-Examples'!
```

```
!AghaCounter methodsFor: 'script'!

consultAndReplyTo: replyDestination
   self replace: self.
   replyDestination reply: contents!

incr
   self replace: (AghaCounter contents: contents + 1)!

reset
   self replace: (AghaCounter contents: 0)! !
```

Further extensions of this initial implementation of the Actor model have been easily obtained with Actalk, e.g., various strategies for optimization, as for instance proposed in the Pract/Acore system [Manning 87], and introduction of the concept of *future*. This concept will also be dealt with, by other means, in the next section.

7.2 The Abcl/1 Model of Computation

Principles The Abcl/1 language (which stands for *A*ctor-*b*ased *c*oncurrent *l*anguage) [Yonezawa et al. 86], although based on the actor philosophy, chose a more pragmatic approach. The language is not intended to be self-contained, but supports hybrid computation. The actor-oriented model of computation may combine with more traditional programming languages which could be used for expressing parts of the behaviors of actors. The main characteristic of Abcl/1 is to propose three distinct types of communication protocols between actors, called *types of message passing*, at the user level:

past is the asynchronous type of message passing, equivalent to the one we designed in the Actalk kernel. The action of sending the message is already completed (in the *past*) as soon as specified, and the sender may immediately resume its computation.

now is a synchronous type of message passing. The sender wants the reply *now* and will wait for it.

future is an eager type of message passing. The place where the reply (or possibly several successive replies) will be eventually delivered in the *future* is specified at the time of sending. Consequently the sender may start manipulating the (*future*) reply before getting its actual value.

Moreover, these three types of message passing are consistent. The same message is sent but the reply destination depends on the type. The reply destination is implicit for the *now* and *future* types. (This will be illustrated in section 7.2.) Consequently the receiver handles uniformly the three types of messages, and only the semantics of replying will change according to the various reply destinations.

Actually, there are some more (four) major properties in order to fully define Abcl/1. Due to space limitation, they won't be discussed here, but they have also been simulated by extending further the following extension.

Principles of Implementation To simulate Abcl/1 into Actalk, we will follow the pragmatic philosophy of Abcl/1. We will define the *now* and *future* types of message passing as returning immediately some standard Smalltalk-80 object on which the sender will synchronize to get the value(s) once computed. We will call this object a *future* object (because it acts in place of the future actual value). It should behave as a queue buffering the successive replies. To implement it, we will use the standard class SharedQueue, already used to implement mailboxes. We just need to rename the assignment message and define a message to consult the first value by suspending until the queue is not empty. The class MAFuture (which stands for *m*ultiple *a*ssignment *future*) implements such *future* objects:

```
SharedQueue subclass: #MAFuture
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Extensions-Abcl'!

!MAFuture methodsFor: 'assignment'!

reply: aValue
   self nextPut: aValue! !

!MAFuture methodsFor: 'consultation'!

value
   | firstValue |
   readSynch wait; signal.
   accessProtect critical: [firstValue _ contentsArray at: readPosition].
   ^firstValue! !
```

The reply: method ensures our reply selector convention (defined in section 5.2) by renaming the nextPut: assignment method. The value method consults the first element of the queue (without removing it). The semaphore controlling the non emptiness of the queue is checked then reset (in order to be able to read the value once again). (This definition is minimal but relies on the implementation of standard class SharedQueue.)

Two more useful methods, next and isEmpty, are inherited from class SharedQueue. The next method returns (and removes) the first element in the queue. But the value and next methods need to wait if the queue is empty. Therefore the isEmpty method provides checking the emptiness of the queue. This is useful for an actor not to be "glued" onto a future object not yet ready, but on the contrary to do some other computation for a while. These two inherited methods respectively simulate the next-value and ready? standard Abcl/1 constructs to access future objects.

The Future Type of Message Passing We may now easily define the Abcl/1 *future* type of message passing. It creates a new future object, sends the original message in the *past* type with the future object as the reply destination, and immediately returns this future object as the value of the transmission (through the underlying standard synchronous message passing level). When the receiver will finish computing the reply to this message, it will reply the returned value to the future object. Then the queue will get assigned with a first element, and will be available for consultation.

The *future* type of message passing will reduce to the *past* type with a future object as reply destination. In order to always and easily know which argument of the message specifies the reply destination, we assume that a reply destination is *always* specified as the *last argument* of a message. Consequently the newly created future object will always replace the initial last argument of the message before sending the message in the *past* type. The initial argument is not significant, but will be used for discriminating the 3 types of message passing as shown in section 7.2. We now define the class `AbclActor` as a subclass of class `Actor` to add this new *future* type of message passing:

```
Actor subclass: #AbclActor
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Extensions-Abcl'!

!AbclActor methodsFor: 'message passing'!

futureCall: aMessage
   | aFuture |
   aFuture _ NAFuture new.
   aMessage arguments at: aMessage arguments size put: aFuture.
   self asynchronousSend: aMessage.
   ^aFuture! !
```

From Eager to Synchronous Communication The *now* (synchronous) type reduces immediately to the *future* type of message passing plus the explicit consultation of (waiting for) the first value of the future object:

```
!AbclActor methodsFor: 'message passing'!

nowCall: aMessage
   ^(self futureCall: aMessage) value! !
```

This reduction of the *now* type to the *future* type, itself reduced to the *past* type, is similar to the reduction semantics proposed in [Yonezawa et al. 86, pages 263–264]. Like in Abcl/1, the implicit reply destination is a future object, first class object, which may be passed along or delegated to other actors. Note however that in Actalk the future object is a standard Smalltalk-80 object and not an actor like in Abcl/1. This

does not limit our simulation in practice however, because of the symbiosis between objects and actors in Actalk.

Combination of Message Types into the Abcl/1 Model The *now* and *future* types of message passing will now be combined with the *past* type already integrated (in section 4.4) into standard Smalltalk-80 syntax. We extend the error redefinition technique by using a symbol to specify the two new types of message passing. The last (`replyTo:`) parameter of the message is used for this purpose. Moreover the value of this parameter is not significant when sending a message in the *now* or *future* types, because their reply destination is implicit. This parameter will be replaced during the reduction process, as seen in section 7.2, by the real reply destination, a newly created future object.

Consequently we will redefine further the `doesNotUnderstand:` method in order to discriminate the type of message passing. We first check if there is at least one argument to the message, and in such a case then check the last one:

```
!AbclActor methodsFor: 'message passing'!

doesNotUnderstand: aMessage
   "aMessage arguments isEmpty
      ifTrue: [self asynchronousSend: aMessage]
      ifFalse: [aMessage arguments last == #future
                  ifTrue: [self futureCall: aMessage]
                  ifFalse: [aMessage arguments last == #now
                              ifTrue: [self nowCall: aMessage]
                              ifFalse: [self asynchronousSend: aMessage]]]! !
```

The three types of messages passing are summarized in this example of consulting a counter:

`aCounter consultAndReplyTo: Print`
: will be sent in the *past* type and include an explicit reply destination (`Print`). The contents of the counter will be displayed on the Transcript, as already explained in section 5.2.

`aCounter consultAndReplyTo: #now`
: will be sent in the *now* type. The value of the expression is the contents of the counter.

`aCounter consultAndReplyTo: #future`
: will be sent in the *future* type. The value of the expression is a future object (instance of class `MAFuture`), referencing the contents of the counter.

One needs to send the message `value` to this future object in order to get the real value. Note that we also developed other implementations of future objects where the consultation of the real value is implicit.

8 RELATED AND FURTHER WORK

Our work can be compared with similar activities which have not been already referred in the paper. [Bézivin 88] shares a similar goal on studying concurrency within Smalltalk-80, however his study is much more general, whereas ours is only devoted to study the actor paradigm for computation by providing a testbed dedicated to it. The ConcurrentSmalltalk [Yokote and Tokoro 87] and Actra [Lalonde et al. 86] projects went further than Actalk in terms of combination of objects and actors. But to achieve it, they had to slightly change both the Smalltalk-80 virtual machine and the semantics of original actors, whereas our objective was to preserve the underlying environment and to simulate various actor systems into it. We are not concerned either in designing an efficient extension of Smalltalk-80 to concurrency, but we provide a platform for specification and experiment with actor languages which takes benefit and reuse of the standard Smalltalk-80 programming environment.

We now expect to explore many fields with this unified tool. We will attempt simulating more actor languages as extensions of the Actalk kernel. For instance, Actalk is currently being used as a designing tool for the Mering-IV project [Ferber and Briot 88]. We also plan the design of a higher level language, analog to Acore [Manning 87], with a compiler generating Actalk kernel code. A group of students is currently working on a general framework based on Smalltalk-80 MVC to visualize and control Actalk actors. Other prospective experiments with the Actalk platform include: modeling communication protocols, modeling strategies for allocation of actors and tasks, and compiling production rules into concurrent daemons implemented by Actalk actors [Voyer 89].

9 CONCLUSION

In this paper we discussed the design of a system, named Actalk, based on Smalltalk-80 and providing an environment to compare and design various actor languages and implementation strategies. The kernel of Actalk introduces actors into the current Smalltalk-80 system. Its implementation was completely described. A methodology for combining traditional Smalltalk-80 programming and actor-oriented programming was discussed. The extension of the current Smalltalk-80 programming environment towards actors was also sketched. The minimal Actalk kernel has been successfully extended in several directions to simulate various actor languages. We described the complete implementation in Actalk of the Actor model of computation and the communication protocols of the Abcl/1 programming language.

Because of space limitation, many topics were just sketched in this paper. They are extensively discussed in the current preliminary report on Actalk [Briot 88] which includes all code for Actalk kernel, extensions and examples.

We would like to express our thanks to Jean-Francois Perrot and the reviewers for suggesting improvements of the paper.

References

[Agha 86] G. Agha, Actors: a Model of Concurrent Computation in Distributed Systems, *Series in Artificial Intelligence*, MIT Press, Cambridge MA, USA, 1986.

[Bézivin 88] J. Bézivin, Langages à Objets et Programmation Concurrente: quelques Expérimentations avec Smalltalk-80, *Actes des Journées Afcet-Groplan Langages et Algorithmes*, Bigre+Globule, No 59, pages 176-187, Irisa, Rennes, France, April 1988.

[Briot 88] J.-P. Briot, From Objects to Actors: Study of a Limited Symbiosis in Smalltalk-80, *Research Report LITP 88-58 RXF*, Université Pierre et Marie Curie, Paris, France, September 1988.

[Briot and Cointe 87] J.-P. Briot and P. Cointe, Definition of a Uniform, Reflexive and Extensible Object-Oriented Language, *European Conference on Artificial Intelligence (ECAI'86)*, Advances in Artificial Intelligence-II, North-Holland, Amsterdam, Netherlands, pages 225-232, 1987.

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

[Ferber and Briot 88] J. Ferber and J.-P. Briot, Design of a Concurrent Language for Distributed Artificial Intelligence, *International Conference on Fifth Generation Computer Systems (FGCS'88)*, Vol. 2, pages 755-762, Icot, Tokyo, Japan, November-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.

[Hewitt 77] C.E. Hewitt, Viewing Control Structures as Patterns of Passing Messages, *Journal of Artificial Intelligence*, Vol. 8 No 3, pages 323-364, 1977.

[Hewitt and Atkinson 79] C.E. Hewitt and R. Atkinson, Specification and Proof Techniques for Serializers, *IEEE Transactions on Software Engineering*, Vol. SE-5, No 1, January 1979.

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

[Lalonde et al. 86] W.R. Lalonde, D.A. Thomas and J.R. Pugh, Actors in a Smalltalk Multiprocessor: a Case for Limited Parallelism, *Technical Report SCS-TR-91*, School of Computer Science, Carleton University, Ottawa, Canada, May 1986.

[Lieberman 81] H. Lieberman, Concurrent Object-Oriented Programming in Act1 in [OOCP 87], pages 9-36.

[Manning 87] C.R. Manning, Acore: The Design of a Core Actor Language and its Compiler, *Revised Master Thesis*, EE and CS dept., MIT, Cambridge MA, USA, 15 May 1987.

[McCullough 87] P.L. McCullough, Transparent Forwarding: First Steps, in [OOPSLA 87], pages 331-341.

[OOCP 87] Object-Oriented Concurrent Programming, edited by A. Yonezawa and M. Tokoro, *Computer Systems Series*, MIT Press, Cambridge MA, USA, 1987.

[OOPSLA 86] Conference on Object-Oriented Programming Systems, Languages and Applications, Special Issue of SIGPLAN Notices, ACM, Vol. 21, No 11, November 1986.

[OOPSLA 87] Conference on Object-Oriented Programming Systems, Languages and Applications, Special Issue of SIGPLAN Notices, ACM, Vol. 22, No 12, December 1987.

[Pascoe 86] G.A. Pascoe, Encapsulators: A New Software Paradigm in Smalltalk-80, in [OOPSLA 86], pages 341-346.

[Theriault 83] D. Theriault, Issues in the Design and Implementation of Act2, *Technical Report No 728*, AI Lab, MIT, Cambridge MA, USA, June 1983.

[Voyer 89] R. Voyer, Implémentation d'Architectures Efficaces pour la Représentation des Connaissances. Application aux Langages Loopsiris et Oks, *Thèse d'Université*, Université Pierre et Marie Curie, Paris, France, February 1989.

[Yokote and Tokoro 87] Y. Yokote and M. Tokoro, Experience and Evolution of ConcurrentSmalltalk, in [OOPSLA 87], pages 406-415.

[Yonezawa et al. 86] A. Yonezawa, J.-P. Briot and E. Shibayama, Object-Oriented Concurrent Programming in ABCL/1, in [OOPSLA 86], pages 258-268.

Meta-level Programming with CodA

Jeff McAffer

Department of Information Science
The University of Tokyo
and
Object Technology International
jeff@acm.org

Abstract. Meta-levels are complex pieces of software with diverse demands in both the computation and interaction domains. Common techniques using just code to express behaviour fail to clearly assign responsibility for a particular behaviour's definition or to provide support for the reuse or integration of existing behaviour descriptions. The techniques of fine-grained decomposition of meta-level behaviour into objects and their subsequent composition into object models provides a framework for creating, reusing and integrating complex object behaviours. Using such a framework, we show that users can develop and integrate quite different object models while retaining a high degree of abstraction and fostering meta-level component reuse.

1 Introduction

Meta-levels are potentially complex pieces of software. They have diverse requirements both for computation and for interaction. Building open meta-level architectures is particularly challenging because of the diversity of behaviours we may wish to describe while maintaining a uniform base-level view of object behaviour.

Many of the current architectures are open but in a restricted sense. They reify various aspects of a particular language or object model and provide infrastructure for change within the scope of that domain. If this is called a 'top-down' approach, we have taken a 'bottom-up' approach. Rather than starting with and then opening a particular object model, we start by describing various notions or views of generic object behaviour and then provide infrastructure for composing these behaviours into specific object models.

This is somewhat related to a basic concept of object-oriented software engineering — decompose a specific problem into generic components and then compose the pieces to solve the problem. From this we get both a solution to our problem and a set of reusable components. In addition, the fine-grained decomposition approach fosters a clear separation between, and definition of, the various components.

Using these techniques at the meta-level, we have developed CodA, a meta-level architecture for describing a wide range of object behaviour models. CodA can be thought of as a generic object engine framework in which users define,

on a per-object or even per-use basis, how object's behave computationally. To demonstrate this we present three object models from different computing domains, specifically; concurrency, distribution, and communication.

The abstraction of behaviours into objects encourages reuse and simplifies the combination of object behaviours. While in CodA, object model combination is still a somewhat manual process, we propose that the architecture inherently provides for; the easier identification of points of conflict, isolation of the effects of changing a component, increased reusability of components derived from the combination of object models and easier management of the behaviour space. The addition of some simple annotations for describing component *properties* further enhances these qualities.

The remainder of the paper is organized as follows. The next section details the CodA architecture and in particular, our approach to meta-level factoring. Sections 3, 4 and 5 detail three object models implemented in CodA and section 6 discusses the combination of object models. We relate CodA to other work throughout the paper and summarize these relations in section 7. A further section concludes the paper and points out some directions for future work. Appended to the paper are the standard interface and default implementations for many of the meta-components discussed.

2 The CodA Meta-Architecture

The key concept in the design of the CodA meta-level architecture is the 'decomposition' or 'factoring' of the meta-level into fine-grained objects or *meta-components*. When programming base-level object-oriented systems we typically factor out behaviour, create objects for each factor and then compose these objects into applications. The only difference here is that our 'applications' are *object models* which describe the operation of objects. That is, the meta-level is just an application whose domain happens to be the behaviour[1] of objects.

In factoring the meta-level we developed a relatively generic model of object execution. The meta-level is defined as playing a number of *roles* in the description of base-level object behaviour. Each role is filled by a meta-component and corresponds to some behaviour such as; object execution (both mechanisms and resources), message passing, message to method mapping and object state maintenance. Roles may be filled by many different components and components can sometimes fill several roles. An object's behaviour is changed by explicitly redefining components or by extending the set of roles. Examples in later sections highlight this process.

The CodA meta-level architecture is largely run-time oriented. It does not provide integral support for language constructs like classes which are required for the static description of object behaviour. Rather, these constructs are borrowed from whatever language is used to implement CodA. For example, in the

[1] We use the term *behaviour* to denote *how* an object acts as opposed to *what* it does and so by nature, *behaviour* is a meta-level concept.

Smalltalk implementation, Smalltalk metaclasses are used to define a nice interface for particular object models (i.e., sets of meta-components) which are shared by instances. These interfaces determine which components can be modified and what configurations the meta-level can take. CodA is perhaps 'lower-level' than other systems but this approach allows us to gain a certain measure of language independence while retaining the potential of the architecture.

This is in contrast to the meta-level facilities found in systems like CLOS[7] and ClassTalk[4, 5]. The focus of these systems is different in that they intend to open or extend the functionality of particular language facilities or constructs. As such, they deal with somewhat more static issues and it is natural that their capabilities and constructs be language specific.

CodA differs in two other related ways; granularity and decomposition into objects. A common approach to factoring object execution behaviour (e.g., message sending or method execution) is to create public interface methods on a small number of meta-level objects [8, 6, 7]). That is, object behaviour is decomposed into code at the meta-level. Changes to a behaviour are made by modifying its related interface methods.

Unfortunately, code is inherently more difficult to deal with than objects. Method objects have little support or infrastructure for interaction, change or extension. Without this, describing complex interactions between several behaviours (i.e, groups of interface methods) is confusing. Behaviours are not encapsulated into atomic units and overall responsibility for a behaviour is not clear. Unanticipated behaviours have no clear home for their description and/or state.

Decomposition into objects gives us a higher-level view of behaviour. Objects abstract code, define points of interaction and ease integration. They remove us from the details of implementation code. The boom in micro-kernel operating systems is also witness to these ideas. Rather than creating one large, all encompassing kernel with many functions, we define a small and simple infrastructure and build/use the OS components as needed.

Forming meta-levels by composition has some advantages over techniques such as multiple inheritance (e.g., ClassTalk[4, 5]). The issue here is similar to the "parameterization vs. subclassing" issue — Should we create a generic object which we parameterize by plugging together pieces, or do we create a class hierarchy covering all possible parameter configurations? Unfortunately, the number of possible configurations is a combinatorial function with explosive potential. Just the small number of behaviours we will discuss in this paper translates into hundreds of possible combinations. Apart from the class/metaclass name-space explosion (which can be partially covered up using anonymous metaclasses), we also encounter a method name-space problem where methods from metaclasses which describe different behaviours, collide.

To put this design discussion in more concrete terms, below we give descriptions of the major elements in the CodA object system. In addition to the meta-level infrastructure, there is the set of seven components each of which describes some behaviour in the basic execution model.

2.1 The Meta-level

CodA specifies that every object has a conceptual meta-level. The meta-level is not a single object but rather a *set* of meta-components each of which describes some aspect of base-level object behaviour. The implementation of the meta-level (i.e., these sets of components) is not defined except to say that a particular set, and thus a particular meta-level, can optionally be: *fixed* and allow no changes, *changeable* in that existing components can be replaced or *extensible* by allowing new roles and components to be added to the meta-level.

Meta-level programmers need some way of shifting to the meta-level and of accessing an object's meta-components. This can be done using language constructs as in ABCL/R2's ↑ (up arrow) [8] or by making explicit meta-level objects which are accessed via normal message passing (e.g., CLOS). In CodA we adopt the technique best suited to the base-level language. For example, in the Smalltalk implementation an object's meta-level as a whole is 'represented' by the result of sending meta to some object (e.g., anObject meta). The meta is used for both shifting and accessing. All messages sent to a meta are executed at the meta-level and metas 'broker' (i.e., provide access to) meta-components.

In the implementation, the meta may take many different forms. It may store internally the meta-components it brokers, it may simply fetch them from somewhere or it may create new ones for each request. This is transparent to the user/programmer. When asked, a meta will return the actual component which fills the specified role. Similarly, a meta can be requested to use a particular component for a particular role.

2.2 Meta-level Components

As a basis, CodA defines a default set of seven components which are present at the meta-level of all objects; Send, Accept, Queue, Receive, Protocol, Execution and State. While they do not cover every possible aspect of object behaviour, the set is extensible and the standard seven cover the behaviours essential to common object models. Readers should note that these meta-components are *logically* distinct. In reality, one entity may fill multiple roles. For example, the Queue and Receive components may, in a particular case, be implemented as one physical object. Meta-components can also be shared between objects.

In our discussion below, details such as message selectors are taken from the Smalltalk implementation of CodA. A simplified specification of these meta-component's execution interface is provided in an appendix.

Send. A Send's main role is to manage the potentially complex series of interactions between message sender and receiver ensuring proper transmission and synchronization. This includes protocol negotiation, synchronization and resource management. For example, when sending a synchronous message, the sender's Send must inform the receiver that a completion signal (e.g., reply) is required, how and to where the signal is to be transmitted, and also block the

sender until the completion signal is received. The PortedObject model discussed below contains a Send which diverges substantially from the default model.

An object's Send is accessed using anObject meta send{:}. It invokes the message receiver's Accept.

Accept. Accepts define the receiver side of the message passing protocol negotiation and synchronization. They are also responsible for determining if a message is valid and how it should be handled (e.g., queued, processed immediately). Note that *accepting* a message is different from *receiving* a message. Acceptance concerns the interaction between the sender and receiver while receiving is the internal act, by the receiver, of picking the message for processing.

An object's Accept is accessed using anObject meta accept{:}. It is invoked by the message sender's Send and invokes the message receiver's Queue.

Queue. Queuing is the main mechanism of decoupling the execution of message senders and message receivers. Messages which have been accepted but cannot yet be processed must be queued. Once queued, the message's sender can be released to continue executing if the message's protocol allows. There are a great variety of possible queuing policies using a variety of factors to determine in which queue a message should be stored (e.g., by sender or type) and the message's place in that queue (e.g., FIFO, priority).

An object's Queue is accessed using anObject meta queue{:}. It is invoked by the message receiver's Accept and by an object's Receive.

Receive. As noted above, receiving and accepting are different operations. Receiving refers to the actual fetching of the next message for execution. In other words, while Accepts are concerned with how objects synchronize and interact with each other (i.e., inter-object synchronization), Receives deal with intra-object synchronization. When a Receive is asked for the next message to process, it may consider many different physical queues and consult various constraint specifications before determining the next appropriate message. The PortedObject model discussed in section 5 details an example of such a situation.

Note that many architectures implicitly combine the operations of Accept, Queue and Receive into the same object with quite a narrow interface. As such, the implementation or integration of a new scheme for one of these behaviours necessarily impinges on the others. Making them explicit and concrete simplifies the construction of complex behaviours.

An object's Receive is accessed using anObject meta receive{:}. It is invoked by objects when they are looking for the next message to process and invokes the object's Queue.

Protocol. A message, having been received, is translated into a method for execution. This is the primary responsibility of an object's Protocol. The most common mapping is an exact message selector to method name match where

methods are examined according to some inheritance scheme. Protocols define both the selection criteria (e.g., exact match) and the search scheme (e.g., single/multiple inheritance). In more complex cases, Protocols may maintain multiple method tables and determine which to use based on some aspect of the base-level or system state.

An object's Protocol is accessed using anObject meta protocol{:}. It is invoked by objects when they need to map a message to a method to execute. That is, typically from an object's Execution.

Execution. For an object to execute methods, it must interact with some system resources (e.g., virtual machines, processes). Executions describe how this interaction occurs. By manipulating its Execution, a programmer can control where and when an object runs as well as its overall importance (e.g., priority) and independence.

Having an explicit execution model also enables methods to be somewhat more abstract and to be executed in different ways depending on the situation. For example, if we are debugging an object, we may wish to execute its methods on a special debugging virtual machine or interpreter whereas normally methods are executed as native machine code. It is the Execution's role is to determine how to execute methods and then execute them.

An object's Execution is accessed using anObject meta execution{:}. Executions are generally invoked by either the Accept or Queue (in the passive case) or by the explicit or implicit invocation of a receive operation (in the active case).

State. Though not directly involved in execution, state is an essential part of an object. The role of a State is to organize and maintain object state. It defines both what slots the object has as well as how the data in those slots is stored. The State *does not* actually hold the data. It simply knows how it can be accessed.

An object's State is accessed using anObject meta state{:}. States are invoked whenever one of the object's slots is accessed.

2.3 Example Meta-level

In Figure 1 we depict the events, meta-components and interactions involved in the sending of a message M from object A to object B (as indicated by the heavy dashed arrow). The shaded areas contain meta-components. Each light arrow is an interaction event (dashed for A's execution thread, solid for B's). The heavy solid arrows indicate the base/meta relationship and go from base-level to meta-level. We have labeled only those meta-components relevant to this particular interaction.

We see that A sends M by interacting with it's Send (1). The Send then transfers M to B's Accept (2) which queues it with the Queue (3). At some point, B will execute a receive operation which invokes the Receive (4) and fetches the next message from the Queue (5). The message is mapped to a method by the Protocol (6) and finally, the message is processed by executing the found method (7). In this way, every aspect of basic execution is reified.

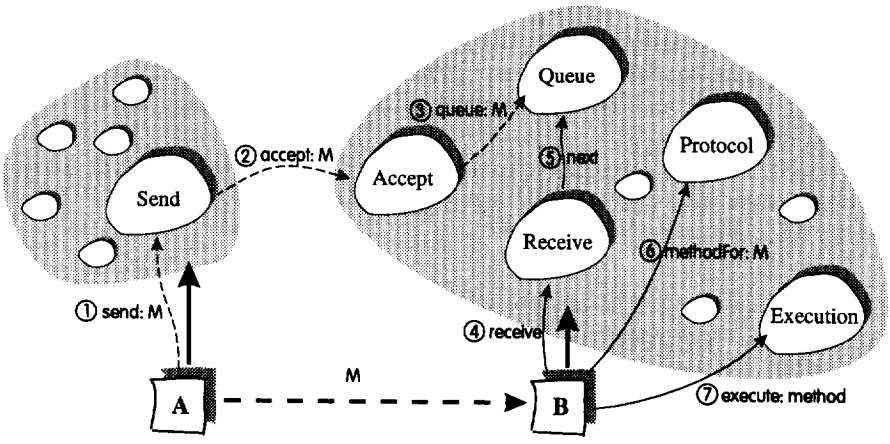


Fig. 1. Sample meta-level configuration and interaction

2.4 Implementation

CodA has been implemented in Smalltalk and much of the remaining discussion draws from that experience. Because CodA deals largely with execution rather than language issues, we have been able to fully integrate it into Smalltalk. For each Smalltalk object we transparently and lazily add the meta-level infrastructure and default meta-components which describe the standard Smalltalk object model. In this implementation, a standard CodA object behaves just like a standard Smalltalk object. Having opened the implementation of a Smalltalk object, we can adjust its behaviour as needed. Objects generally retain their base-level semantics but gain some additional behaviour such as concurrency or distribution.

Using the CodA framework as described above, we have created a library of components with which we have built a number of object models. The following sections describe the design and implementation of three models; ConcurrentObject, DistributedObject and PortedObject as demonstrations of the CodA concepts and design. An appendix contains example implementations of the default behaviours for some meta-components. Our discussion is set in terms of changes to these behaviours.

3 Concurrent Objects

Passive objects are reactive in that they simply respond to external stimulus or input and 'borrow' processing resources from message senders. In the ConcurrentObject model, objects have their own internal *activity* and processing

resources (threads). This behaviour is described by a ConcurrentExecution component which fills the Execution role.

A ConcurrentExecution's idling execution behaviour (i.e., what objects do when they are not driven by user code) is similar in intent to that of Actors as seen in [1, 3, 10]. While formal Actors redefine their execution behaviour after every execution, in practice the replacement behaviour is the same; receive and process a message. Our basic activity model is similar; an endless loop, receiving and processing messages. For passive objects, the activity loop is implicit in the runtime system. For ConcurrentObjects, the loop runs explicitly in the threads associated with an object's ConcurrentExecution. The following is an example of such a loop for an object base.

```
| message result |
[true] whileTrue: [
    message := base meta receive receiveFor: base.
    result := self process: message for: base.
    base meta send reply: result to: message for: base]
```

When a message arrives, a passive object's Queue actually calls the object's Execution and directly triggers the processing of the message. That is, there is no queuing, only immediate processing. The Receive is never called explicitly as objects are always implicitly receiving incoming messages. Adding explicit thread(s) and an activity to an object both invokes its Receive and raises the possibility the sender and receiver of a message may be disjoint with respect to execution threads.

In addition to the activity loop, ConcurrentObjects change the Queue to ensure that messages are actually queued rather than passed on to the Execution. StandardQueue (shown below) is an example of such a Queue. It maintains an internal queue structure on which it implements the Queue interface. The actual queuing model used (e.g., FIFO) depends entirely on how we want incoming messages to be ordered (i.e., the object's queuing policy). This is specified by the user in the creation of the StandardQueue.

```
StandardQueue>>nextFor: base
    ^queue next

StandardQueue>>enqueue: message for: base
    queue add: message
```

The ConcurrentObject model does not need to define new message sending mechanisms as the default Send components already include the notions of synchronous, asynchronous and future messages. In the default, passive object case, synchronous sending is the default, future messages represent a promise to compute similar to closures or blocks, and asynchronous messages are mapped to synchronous messages where the result is ignored.

These ideas are included in the default behaviour for two reasons; they are useful in normal object behaviour description and they are relevant to system

parallelism, not object concurrency. For example, a distributed system can contain no ConcurrentObjects but still require asynchronous sends.

4 Distributed Objects

The DistributedObject model is a somewhat larger change to object behaviour. In the model, objects live in *spaces*. An object's Execution and State can live in different spaces and can be independently copied, replicated or moved between spaces. Inter-space messaging fits naturally into the normal object model through the use of RemoteReferences or Proxies [14]. The model contains a sophisticated, uniform mechanism for describing how objects are transmitted from space to space (marshaling). The model is equally applicable to passive and active objects and is built largely out of new components/roles (e.g., Marshaling, Replication and Migration) and infrastructure objects (e.g., RemoteReference and Space). Here we present a few of these new structures. More detailed coverage of the DistributedObject model can be found in [9].

Spaces are places in which objects exist (store their state) and execute. A Space is known to every other Space and can be addressed (i.e., sent messages) directly from anywhere in the distributed machine. They manage the mapping between global object ids and local representatives (e.g., RemoteReferences, replica).

A RemoteReference is a local representative or *Proxy*[14] for some remote object. Locally they are just like any other object. They can be stored in instance slots, assigned to variables, passed as arguments, etc. When they are sent a message, the simplest RemoteReference just forwards it to the space containing the real object — the *target*. More sophisticated RemoteReferences process some messages locally while forwarding others to the target.

RemoteReferences are themselves implemented using modified CodA meta-components. According to the CodA execution model, when a message is sent to an object, the sender's Send and the receiver's Accept interact to effect the message transfer. In the DistributedObject case, these meta-components are in different spaces. Local to the sender, the receiver is a RemoteReference and the receiver's Accept is an intelligent RemoteReference to the target's Accept. Rather than performing the normal accept operation, the local Accept *marshals* the message into a stream of bytes and transmits it to the remote space. Once there, it is reconstructed and accepted by the target's Accept. In this way, the DistributedObject model is uniformly applied to all objects in the system, even those at the meta-level.

4.1 Replication

The basic idea of replication is that an object's state can exist in multiple Spaces at the same time. Furthermore, through the use of some distributed consistency schemes, we can maintain the proper semantics of our programs. Schemes for

replication range from simple one-off copying (not technically considered replication here) to fully coherent replication. Since this is an entirely new behaviour for objects, it is a new role (Replication) for the meta-level and requires new meta-components for its implementation (The addition of new roles is covered in Section 6). As we will see, the role of the Replication is quite independent of the object's execution and the actual structure of an object's state variables.

To demonstrate replication we develop the partial replication scenario shown in Figure 2. The figure shows two objects, original (in Space 0) and replica (in Space 1). Though not shown, original is actually a 2D N-Body [2] problem solver which calculates the forces exerted by, and movements of, a collection of bodies or *particles* in a 2D plane. N-Body solvers arrange a set of particles in a Quad tree structure according to their physical location and then process each particle individually. Overall, processing consists of a couple tree scans and iterations over the collection of particles.

To distribute this algorithm we divide the particles into subsets which are worked by different solvers, one per Space. The sets however, are not entirely independent since all particles potentially exert forces on all others. As the tree is the central data structure for relating particles to one another, it must be globally known and unique. The solver is a prime candidate for partial replication.

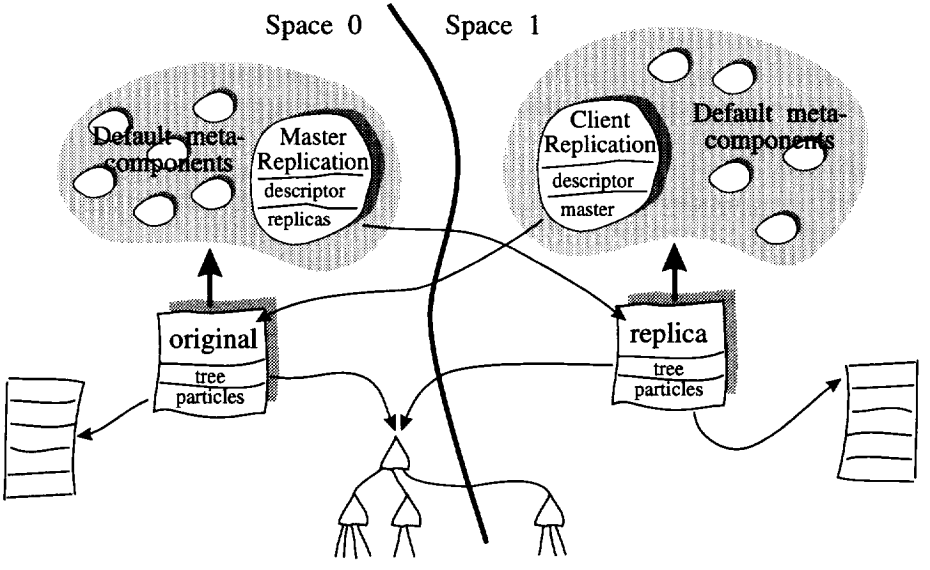


Fig. 2. Distributed object layout

As shown in Figure 2, original, the solver, has two slots; particles and tree. replica is a partial replica of original where the tree slot is consistency managed

and the particles slot is not. Every replica in the system shares the same tree but has an independent particle set. The replication of original is done in a series of six steps. Figure 3 shows the required code while the discussion below explains each step.

```
1)  original meta replication asMasterUsing: #('tree') for: original.
2)  original meta replication replicateIn: (Spaces at: 1) for: original

MasterReplication>>replicateIn: space for: base
3)  space replicate: base using: descriptor for: base.
6)  replicas add: space

Space>>replicate: copy using: descriptor for: master
4)  copy meta replication asClientOf: master using: descriptor for: copy.
5)  master become: copy
```

Fig. 3. The making of a replica

1. Ensure that the original's Replication is compatible with the behaviour described by the MasterReplication component. It should be able to detect state changes in the appropriate slots and maintain a list of replicas. The first argument is a marshaling descriptor which specifies how the slots of original are to be copied to the remote space and as a result, how original is to be replicated. Simply giving a slot name indicates that the slot is to be replicated using whatever marshaling technique is appropriate at the time (i.e., the default).
2. Invoke the replication operation and specify which spaces are to receive replica. In keeping with our example, only Space 1 is specified.
3. Copy the relevant slots of original to all of the specified Spaces. The replicate:using:for: message has three arguments. Though the first and third appear redundant, they are not — they are marshaled differently. The first argument is marshaled according to the specification in descriptor while the third is marshaled as a reference. This difference is critical for the next two steps. When the message gets to the remote space, the first and third arguments will no longer be identical. The first will be a copy of base while the third will be a reference to base. Note that though marshaling descriptor specification is a simple addition to the messaging syntax, the details are omitted from this example to improve clarity.
4. Make the remote copy into a replication *client* of original. This is similar to step 1 and executes in the remote Space which will contain replica. copy's meta-level is modified such that all state changes are delegated to master and the Replication knows the identity of its master for future reference.

5. Convert any preexisting remote references to original to be local references to replica. Remote spaces may contain references to master prior to replication. To maintain a consistent view of the world, these remote references should be changed into local references to the newly created replica.
6. Invoke consistency management on the replicated slots of original by adding the Space to the list of consistency controlled replica locations.

In step 1 we hooked the relevant state change operations for original. Note that we do not require a new State component. The existing component's meta-level is manipulated to hook state accesses. This both isolates replication from representation and reduces the possibility of object model conflict. When original's replicated state is changed, its Replication's update:with:for method (shown below) is invoked by the hook. The method simply broadcasts the change in slot to all of original's clients. Typically this would be done by multi-cast messaging for efficiency though here we specify an iterative approach for clarity.

```
MasterReplication>>update: slot with: value for: base
    replicas do: [:space | | rep |
        rep := (base in: space).
        rep meta replication update: slot with: value for: rep]
```

In this example we have shown a relatively lax model of consistency. To implement *strict consistency* requires only the addition of a two phase update protocol between masters and clients and the hooking or delegation of both read and write state accesses on masters and clients rather than just writes. Both of these changes are straightforward and are done using existing meta-level structures and mechanisms.

4.2 Summary

The DistributedObject model, and this example in particular, highlight a central theme to our work — the addition to, or modification of, an object's computational behaviour without changing its base-level code. The result is the ability to use standard class libraries in many different environments. For the N-Body application, the original uniprocessor sequential version required a code change only where a new tree node was created and we wished to explicitly direct its location. Other changes were done to take advantage of the newly introduced concurrency but were not essential.

In terms of distributed object-oriented computation, our model also highlights something which we feel is essential — the ability to talk about distribution on a *per-use* basis rather than just on a per-class or even per-object basis. In the N-Body application (replica creation step 3) we demonstrate the need for use-based marshaling. Also, while our distribution scheme calls for partial replication of the solver objects, someone else's might use a different scheme. Using this architecture, all they need program is the meta-level of individual objects as they are used.

5 Ported Objects

PortedObjects are objects which communicate and behave in a dataflow-like way. They have *ports* or channels over which data flows and when data is available for processing, processing is done. This style of behaviour is interesting in a number of areas. People working in concurrency formalisms like the π-calculus [11] have found channels and ports to be useful in specifying object communication. Most popular data visualization and analysis systems like AVS [16], IRIS/Explorer [15] and parallel system analysis tools like Pablo [13] have dataflow or analysis graph architectures. The model is interesting to us in both regards.

Ported behaviour should be as transparent as possible to the base-level code. For example, in an analysis system we developed, we used a set of generic objects which describe various analysis operations (e.g., filters, collectors, expert systems, DSP processors) and added a set of meta-components which gave these *analysis objects* ported behaviour. The idea was that users (analysts) could then build their own analysis tools by simply connecting existing analysis objects to form the desired analysis graphs.

The addition of porting is done by identifying the parameters and results of each analysis object. Each parameter or result is made into a port on the surface of the object. Users program with PortedObjects by building connections between these ports. 'Programs' are run by feeding data to some of the free parameter ports. Values put in a port are automatically broadcast to all objects connected to that port. When some object in the graph has sufficient input, it processes the data and stores the results in its result ports and so, passes it to the next object. This process continues and data flows through the graph.

5.1 Meta-level Design

The PortedObject meta-level design in CodA is done entirely via modifications to the following five meta-components; Send, Accept, Queue, Receive and Execution.

Send. At the inter-object level, PortedObjects cannot explicitly send messages. They can only store values in their logical output (result) ports. An object cannot tell whether or not storing a result will cause the value to be transmitted to some other object. The meta-level however, can detect the result setting operations and trigger the broadcasting of the new value to all objects connected to the modified port. So, while base-level PortedObjects have no explicit send operations, they implicitly use message sending in their implementation.

A PortedObject's Send behaviour is defined by a generic MultiSend object which provides infrastructure for multi-casting messages to a known set of receivers. For PortedObjects these receivers are represented by ports and connections.

Accept. PortedObjects use MultiAccepts whose behaviour differs from that of normal Accepts only in their support for manipulating ports and connections. This

consists mostly of add/remove and connect/disconnect methods in various forms. Also, as messages arrive (via accept:for:), the MultiAccept marks them with the port over which they came and queues them as per normal operation.

Queue. The PortedObject model uses MultiQueues for their Queue component. A MultiQueue supports the sorting of elements into one of many logical queues as defined by some discriminator, in this case, the arrival port. The default Queue interface is augmented with duplicate operations which take an additional parameter, a port identifier.

Receive. A PortedObject's Receive is concerned more with parameter coordination than ports and connections. Some PortedObjects require several inputs to be present before processing can take place. In some cases, processing only makes sense if some set of these parameters are reset from iteration to iteration. In others, a change of one parameter is cause for recalculation. To manage these constraints, PortedObjects use CoordinatedReceives.

When a CoordinatedReceive is asked to receiveFor: by an Execution, it produces the next available message which satisfies the current set of coordination constraints, cSet (see code below). Here cSet represents a very simple system of constraints based on a collection of port identifiers from which it is valid to take a value. As values arrive, their port is removed from cSet. When the set is empty, we know that we have received all the required values and so the object is ready for processing. That is, the receiver is *coordinated*. The initial values for the cSet are derived from information supplied by the programmer as part of the PortedObject definition scheme.

```
CoordinatedReceive>>receiveFor: base
    | message |
    message := base meta queue nextSatisfying: cSet for: base.
    cSet remove: message arrivalPort.
    ^message
```

Execution. Since PortedObjects do not have explicit message passing, we draw a distinction between the implementation receiving and executing a message, and the base-level object itself actually being evaluated. PortedObject evaluation can only happen when the object is coordinated. The messages handled by the Sends and Receives are infrastructure related and serve to transfer data (i.e., parameters and results) and determine coordination.

```
CoordinatedExecution>>process: message for: base
    | method |
    method := base meta protocol methodFor: message for: base.
    self execute: method with: message for: base.
    base meta receive isCoordinated ifTrue: [
        self evaluate: base.
        base meta receive resetCoordinationSet]
```

The main change in a PortedObject's Execution is highlighted by the modified process:for: method shown above. After executing an infrastructure message, the Execution tests for coordination. If the object is coordinated, it is evaluated. After evaluation, the coordination set is reset.

5.2 Compound PortedObjects

In complex PortedObject graphs we would like to be able to think of and manipulate a group of PortedObjects as one. The encapsulation should be completely transparent to objects both inside and outside the group. By taking a generic analysis object and reusing some of the meta-components already described, we can create a *compound* PortedObject as shown in Figure 4.

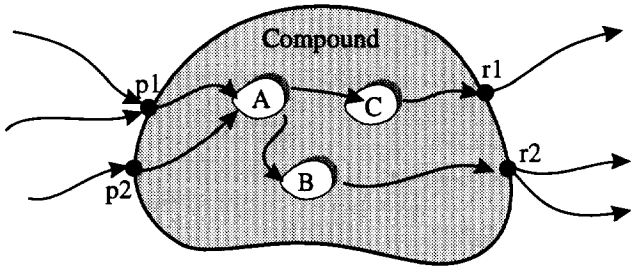


Fig. 4. Compound object example

In the diagram we see three objects (A, B and C) encapsulated in Compound. Compound is itself just a generic analysis object which by default has no ports or particular evaluation behaviour. We have added parameters p1 and p2, and results r1 and r2. The parameters and results are logically linked, as appropriate, to those of the contained objects.

In accordance to the PortedObject model, data values coming to A should come from some PortedObject's Send (e.g., a MultiSend). Compound's Send fits those requirements but it manages the *external* connections for Compound and has no facilities for managing a separate set of *internal* connections. The situation is similar for Compound's result ports and Accept.

An obvious solution is to implement new Send and Accept components which keep two connection lists, one internal and one external. But this would just be duplicating existing behaviour and adding special cases in connection management. An alternative is to use two PortedObjects instead of the single Compound. One would handle the group's parameters and one its results. This however goes against our goal of having the group act as one object.

We take a somewhat novel approach and extend Compound's meta-level to have two new roles, InternalSend and InternalAccept. These roles are actually filled

by normal MultiSend and MultiAccept components. Compound's original Accept and Send components remain unchanged and continue to handle all external connections while InternalSend and InternalAccept handle the internal connections. Figure 5 shows the configuration for the parameter side of Compound from Figure 4. Note that p1 and p2 in the two figures are the same.

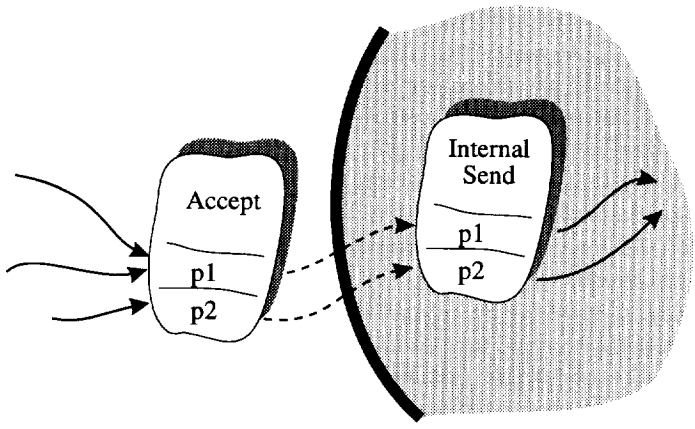


Fig. 5. Compound object parameter handling

Compound's Accept has two ports, p1 and p2, corresponding to its two parameters. Values arriving at those ports are tagged as described above and then passed, at the meta-level, to the corresponding port of Compound's InternalSend. From there they are, as per normal operations, broadcast over the appropriate port's connections to the objects contained in Compound. The structure of the result side is analogous though reversed.

This model is simple and appealing. From a porting and communication viewpoint, all objects have a consistent and uniform model. From a meta-level architecture point of view, it demonstrates how the meta-level is completely extensible and how meta-component defined behaviour is reused. In our original architecture design we never imagined a requirement for having multiple Send or Accept components. In this situation however, it is not only convenient and reusable but is aesthetically pleasing.

6 Putting the Pieces Together

As we have seen, in building object models, we may need to extend the set of roles that the meta-level plays. To do this, we generally create the role and develop at least two components; one which defines some default behaviour and

one which defines the new behaviour we specifically want to add to objects. The default component is used when an object's role is accessed but no behaviour (e.g., component) has been explicitly provided. Using this technique, as soon as it is added to the system, all objects have some defined behaviour for the role. Developers are then free to create variations on this default behaviour and substitute them for the default component on individual or groups of objects. For example, the DistributedObject model defines several new roles and we have shown that one of them, Replication, can be filled by three different components; DefaultReplication, MasterReplication and ClientReplication.

Until now we have only discussed how to build object models from scratch. We have not looked at composing new object models from others. The combination of object models in CodA is still largely a manual process but the architecture itself reduces considerably the work to be done.

Combining disjoint (i.e., non-overlapping) object models is straightforward. The new model simply contains the union of the non-default components from the originals. As long as all the components provide the standard CodA interfaces, the new model will run fine.

Combining overlapping object models requires programmer intervention. The general approach is to build new components which merge the behaviour of those being combined. CodA's fine-grained decomposition into objects helps in several ways here. First, the finer granularity gives a more precise indication of where the models collide. Second, the object-orientedness of the decomposition limits both the scope of the conflict and the spread of the change required for its resolution.

Objects also give us an abstraction of behaviour which is easier to use and reuse. Consider two object models X and Y which both redefine the Send component. If we wish to create a model XY (the combination of X and Y) we have to resolve the conflicts between XSend and YSend. If we assume that this resolution creates XYSend, a Send with the properties of both the X and Y models, then the conflict between XSend and YSend has been resolved and need never be resolved again. If the conflict is encountered in the combination of some other models, we can simply reuse XYSend. As we build a library of components, conflict resolution will become more a problem of identifying the existing component with the correct properties than of actually writing code.

The fine-granularity of our design is double edged however. Ad hoc groups of meta-components do not present as nice a package of object behaviour as a single 'meta-object'. Users are faced with a potentially large choice of possible components to fill a particular role. We address this in a number of ways. Relatively simple object models like ConcurrentObjects are represented by methods which configure meta-levels. To use that model, a user just applies the method to the object in question (see the example code below). This simply overwrites any preexisting behaviours.

```
configureAsConcurrentObject: base
    base meta queue: (StandardQueue for: base).
    base meta execution: (ConcurrentExecution for: base)
```

More sophisticated models use base-level language constructs (e.g., classes and metaclasses) as object model representatives. This has the benefit of being integrated with the environment but the drawback of still requiring user-written code.

Using the notion of *properties*, we address both the composition and combination problems. A property is a simple declarative token which points out one way in which a component is different from the default. For example, some of the PortedObject model components are *multi*. Comparing property lists allows us to even more precisely identify conflicts in components. Properties are also used in object model specification. Rather than hardcoding the use of particular components in a model, programmers declaratively specify that, for example, they want a Send with a certain set of properties. Whether a change is required and which actual Send is used is determined dynamically.

Properties, like many categorization systems, suffer from naming problems. Defining and guaranteeing the semantics of a particular property is difficult at best. So, while they do not solve the composition or combination problems, these operations, in a sufficiently rich and consistent component environment, are reduced to property constraint satisfaction.

A completely different approach is component generalization and parameterization. Looking at the models we have defined here, only ConcurrentObjects and PortedObjects overlap in the component domain. Initially we applied the above techniques with success. Then, as we developed other models which also overlapped, we found ourselves generalizing and parameterizing the various components to be more reusable. For example, Executions were changed to take a user supplied code block to define their execution activity. The result was a library of general components which can be setup in many different ways and so can be used in many different situations.

7 Related Work

There are several projects related to our work. We have already mentioned some and relate to a few more below. In general, most previous efforts have either a different focus or different approach with respect to the issues in object behaviour description.

RbCl [6] is similar to CodA in that explicitly supports active objects and factors the meta-level into objects. However, it factors out only a limited set of components and does not provide a framework for their composition and interaction.

The Apertos operating system [17, 18] differs mainly as a result of a different target domain than of the overall architecture. Apertos reifies aspects of object behaviour at the operating system level (e.g., memory management, page faults and device drivers). This level is mostly orthogonal to the current CodA meta-components. It would be interesting to combine the two domains in one framework to get a more complete and far-reaching reification of object behaviour.

Recent work in AL-1/D with distribution control at the meta-level [12] is also similar to ours. They focus on a set of meta-level concepts directly related to distribution requirements. We feel that in fact, in real systems, the issues related to distribution are more far-reaching. They involve heterogeneous state representations and update policies, and demand mechanisms for the control of the intra-object concurrency implicitly introduced by remote referencing. Furthermore, all of this should be possible on a per-use basis. As such, it is appropriate to use a meta-architecture with wider scope. It also appears that the CodA infrastructure and interface is more clearly defined.

Over the years we have been influenced by Actalk [3]. It provides a testbed for describing object behaviours in areas relating primarily to concurrent execution and message passing. Though recent versions are more and more component-based at the meta-level, it is still somewhat monolithic and code-based. Having said that, we are very interested in implementing in CodA, many of the object models available in Actalk.

8 Conclusions and Future Work

By treating the meta-level as "just another application" and applying typical software engineering practices, namely fine-grained decomposition into objects, we have created an extensible, uniform framework for object behaviour description. Object model definition by composition and extension was demonstrated through the development of three object models; ConcurrentObject, DistributedObject and PortedObject. These models show that in CodA, quite diverse object behaviours can be created with relatively minor changes to the meta-level and almost no changes to the base-level.

We have found that the decomposition of behaviour into objects as opposed to code gives meta-level users a higher-level abstraction in which to program. Responsibility for the definition of particular behaviours rests with identifiable, discrete objects rather than being spread through the meta-level code. Individual meta-components have a narrower domain and so are more easily reused. The unruliness of groups of meta-components (as opposed to single 'meta-objects') is addressed via the use of *properties* and the constructs (e.g., classes and meta-classes) available in the implementation environment.

While we have not solved the general object model composition and combination problem, the CodA architecture helps to identify and isolate points of conflict between models. Meta-components provide *firewalls* which limit both the scope of potential conflict and the spread of the changes resulting from its resolution. Components developed to resolve a particular conflict can be reused wherever that conflict occurs so the conflict need only be resolved once. In addition, we propose some simple but potentially useful concepts such as *properties* which ease the burden on meta-level users.

An interesting avenue of future work is to look for techniques which allow the dynamic *compression* and *expansion* of meta-levels. While the separation of meta-level roles into individual objects is logically efficient, it may be less than

optimal in implementation. By providing some sort of declarative description of each meta-component's behaviour and expected interactions (e.g., properties), we can automatically combine several components into one. This is analogous to some problems in typing, partial evaluation and code generation. Similarly, by remembering something of their original structure, compressed meta-structures can be expanded into their original form. Using a combination of expansion and compression, monolithic (e.g., compressed) structures can be made open. To change a monolithic meta-level structure, it is first expanded, then changed and finally re-compressed. We believe that this direction holds great promise in the battle to make systems more open and to make open systems more efficient.

9 Acknowledgements

We gratefully thank Jean-Pierre Briot, Pierre Cointe, Nick Edgar and Laurent Thomas for discussions and helpful comments on drafts of this paper.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. MIT Press, 1986.
2. J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.
3. Jean-Pierre Briot. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In S. Cook, editor, *Proceedings ECOOP '89*, pages 109–129, Nottingham, July 1989. Cambridge University Press.
4. Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings of OOPSLA '89*, pages 419–431, October 1989.
5. Pierre Cointe. CLOS and Smalltalk: A comparison. In Andreas Pæpcke, editor, *Object-oriented programming: The CLOS perspectives*, pages 215–250. MIT Press, 1993.
6. Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCl: A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings of the International Workshop on Reflection and Meta-level Architecture*, pages 24–35, November 1992. Tokyo, Japan.
7. Gregor Kiczales, Jim des Rivières, and Daniel Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Massachusetts, 1991.
8. Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 127–147, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.
9. Jeff McAffer. Meta-level architecture support for distributed objects. In preparation.
10. Jeff McAffer and John Duimovich. Actra - An industrial strength concurrent object-oriented programming system. *ACM SIGPLAN OOPS Messenger*, 2(2):82–85, April 1989. Proceedings of the ACM SIGPlan OOPSLA Workshop on Object-Based Concurrent Programming.

11. Robin Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*. Springer Verlag, 1992.
12. Hideaki Okamura and Yutaka Ishikawa. Object location control using meta-level programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 821, pages 299–319. Springer Verlag, July 1994.
13. Daniel A. Reed. *An overview of the Pablo performance analysis environment.* Department of Computer Science, University of Illinois, 1992.
14. Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot. SOS: An object-oriented operating system – Assesment and perspectives. *Computer Systems*, 2(4):287–337, Fall 1989.
15. Silicon Graphics Inc. *Explorer User's Guide*, 1992.
16. Stardent Computer Inc. *Application Visualization System, User's Guide*, 1989.
17. Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 414–434, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.
18. Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In S. Cook, editor, *Proceedings ECOOP '89*, pages 89–106, Nottingham, July 1989. Cambridge University Press.

A Meta-component Specification

The following is a description of the essential meta-components in CodA. Only the protocols needed for actual execution are specified. Others such as those required for configuration are not included as they will vary depending on the capabilities or properties of the component. All specifications are presented in terms of the Smalltalk implementation. The interface also supports a number of convenience methods which combine frequently used operations under one simpler protocol. These are not generally included in the specification. Readers are referred to the body of the paper for explanations of the component's roles.

Note that throughout the interface, the base-object (typically referred to as base) is explicitly specified as an argument. This is done for two main reasons: It allows for meta-components which have a one-to-many relationship with the base-level and it removes the requirement for implicit assumptions regarding the behaviour and arguments associated with an interface. For example, it may not be the case that the *sender* field of a message being sent is actually the object doing the send operation. As a general rule, if a meta-component maintains a one-to-one relationship to the base-level then the base argument is ignored.

A.1 MetaComponent

All meta-components respond to the following messages:

isDefaultBehaviour Answer true if the receiver represents the default behaviour for the role in which it was cast. Answer false if it has been explicitly set by the user.

A.2 Send

Fundamentally there are three different kinds of message sending: synchronous, asynchronous and future. These are realized as different protocols. Variations on the message argument introduce orthogonal concepts such as; express and system messages. Sends also explicitly support the transmission of reply messages as their requirements may be different from that of other message sends.

send: message for: base Send message for base. Defines the default sending behaviour and is typically, though not necessarily, mapped to one of the send operations given below.

send{Async/Sync/Future}: message for: base Send message for base. The sender and receiver are synchronized according to the specified mode (i.e., Async, Sync or Future).

reply: result to: message for: base Reply result to the reply destination listed in message for base. Replies are normal messages but may need to be treated differently to facilitate synchronization and other schemes.

A.3 Accept

accept: message for: base Determine if message can be accepted by base. To accept a message is to promise to consider performing computation based on its contents. It is not an implicit guarantee that the message will be processed but rather that the message has arrived at the destination. The act of accepting a message also involves a preliminary determination of what is to be done with the message. For example, if the message is marked as *express* then it should be considered for immediate execution.

acceptReply: message for: base Replies are normal messages but may need to be treated specially to facilitate synchronization and other schemes.

A.4 Queue

There are a great variety of possible queuing policies and factors in determining in which queue a message should be put and where it should be placed. These policies and factors are generally established via setup parameters on the Queue. The Queue protocol supports methods for enqueuing and dequeuing messages and various forms of message retrieval.

dequeue: message for: base Remove message from the receiver.

enqueue: message for: base Add message to the receiver.

nextFor: base Remove and answer the next available message from the receiver. This defines the default dequeuing behaviour and is typically, though not necessarily, mapped to one of the next operations given below.

blockingNextFor: base Remove and answer the next available message from the receiver. An answer is not given until a message is available.

nonBlockingNextFor: base Remove and answer the next available message or nil if none is available. An answer is always returned immediately.

nextSatisfying: constraints for: base Remove and answer the next available message from the receiver which satisfies the constraints. An answer is not given until such a message is available.

peekFor: base Answer the next available message from the queue or nil if none are available. No messages are removed from the receiver. An answer is always returned immediately.

A.5 Receive

receiveFor: base Answer the next available queued message. This defines the default receiving behaviour and is typically, though not necessarily, mapped to one of the receive operations given below.

nonBlockingReceiveFor: base Answer the next available queued message or nil if none are available. Subsequent calls will not return the same message. An answer is always returned immediately.

blockingReceiveFor: base Answer the next available queued message. Subsequent calls will not return the same message. An answer is not given until a message is available.

A.6 Protocol

methodFor: message for: base Answer the method best suited to processing message. If a method cannot be found then answer some method which will handle the error condition.

A.7 Execution

Executions describe the basic processing activity of an object. How and when they receive, lookup and execute messages. For passive objects this is determined largely by the external thread of control and when other objects send messages to the Execution's base-object(s).

For active objects, the Execution has complete control over these aspects. It must also define what the object does when it is not processing some received message as well as how the object's execution maps onto physical computational resources (e.g., processes and processors). In short, the Execution provides an encapsulation of processing power for the exclusive use of its base-level object(s).

execute: method with: arguments for: base Execute method with arguments on receiver base.

process: message for: base A convenience protocol which combines message to method mapping and method execution. message is processed by first sending methodFor:for: to the relevant Protocol and then execute:with:for: to the receiver.

processImmediately: message for: base Similar to process:for: but the any normal execution currently underway is interrupted with the processing of message.

activityFor: base *Answer an evaluable description of base's activity loop.*

A.8 State

States describe the physical storage and structure of objects. It is important to note that they do not actually contain the base-level state but simply know how to store and retrieve it. State slots can be named or numbered.

at: id for: base Answer the current value of slot id in base.
at: id put: value for: base Store value in slot id of base.
slotIdsFor: base Answer a list of all the ids for the slots available in base.

B Default Meta-component code

The following are the default implementations for many of the methods mentioned in the body of the paper. They are given as a point of reference so readers can judge the amount of change required to effect the behaviours described.

```
DefaultSend>>send: message for: base
    ^message receiver meta accept
        accept: message for: message receiver

DefaultSend>>reply: result to: message for: base
    | reply |
    reply := message asReply.
    reply arguments: (Array with: result).
    ^reply receiver meta accept
        acceptReply: reply for: reply receiver

DefaultAccept>>accept: message for: base
    ^base meta queue enqueue: message for: base

DefaultAccept>>acceptReply: message for: base
    ^base meta execution processImmediately: message for: base

DefaultQueue>>enqueue: message for: base
    ^base meta execution process: message for: base

DefaultQueue>>nextFor: base
    ^nil

DefaultReceive>>receiveFor: base
    ^base meta queue nextFor: base

DefaultProtocol>>methodFor: message for: base
    ^self lookupTable at: message selector

DefaultExecution>>execute: method with: arguments for: base
    ^method executeFor: base withArguments: arguments
```

```
DefaultExecution>>process: message for: base
    | method |
    method := base meta protocol methodFor: message for: base.
    ^self execute: method with: message args for: base

DefaultExecution>>processImmediately: message for: base
    ^self process: message for: base

DefaultState>>at: id for: base
    ^self slots at: id

DefaultState>>at: id put: value for: base
    ^self slots at: id put: value
```