Composition of Before/After Metaclasses in SOM

Ira R. Forman
Scott Danforth
Hari Madduri

IBM Object Technology Products
11400 Burnet Road
Austin, Texas 78758

Abstract

In SOM, the IBM System Object Model, a class is a run–time object that defines the behavior of its instances by creating an instance method table. Because classes are objects, their behavior is defined by other classes (called metaclasses). For example, a "Before/After Metaclass" can be used to define the implementation of classes that, by suitable construction of their instance method tables, arrange for each invocation of a method to be preceded by execution of a "before method" and followed by execution of an "after method." This paper introduces and solves the problem of composing different Before/After Metaclasses in the context of SOM. An enabling element in the solution is SOM's concept of *derived metaclasses*, i.e., at run–time a SOM system derives the appropriate metaclass of a class based on the classes of its parents and an optional metaclass constraint.

Introduction

One interpretation of the history of programming is that progress is made by providing abstractions to ever larger entities and by ensuring the composability of those abstractions. In the beginning, assembly language instructions were gathered into control structures. Subsequently, control structures were gathered into procedures, and this was followed by the gathering of procedures into abstract data types. Now we have arrived at Object–Oriented Programming, where abstract data types are gathered into an inheritance hierarchy.

This paper addresses a further abstraction, Before/After Metaclasses, and solves the problem of their composition. A Before/After Metaclass has a *before method* and an *after method* that are executed before and after the methods of the instances of its instances (an instance of a metaclass is a class, which in turn has instances). Applications for Before/After Metaclasses abound, e.g., method tracing, invariant checking, path expression checking, object locking, etc. Given these opportunities, it is imperative that Before/After Metaclasses compose.

This paper solves the Before/After Metaclass Composition Problem in the context of the IBM System Object Model (SOM). SOM is an object–oriented model [7,17,22,23]. The SOM runtime supports the model and allows programs written in arbitrary languages to use the model via the SOM API. A binding is code that facilitates the use of a class or the implementation of a class. The SOMobjects Toolkit [24] provides both usage and implementation bindings for C and C++; also, language vendors for Smalltalk (Digitalk), Cobol (Mircofocus), and C++ (IBM, Metaware, and Borland) have announced support for SOM in their products.

The SOM Model

In SOM, classes are objects whose classes are called metaclasses. A class is different from an ordinary object because a class has (in its instance data) an instance method table defining the methods to which instances of the class respond. During the initialization of a class object, a method is invoked on it that informs the class of its parents. This allows the class to build an initial instance

Presented at OOPSLA '94 Conference, October 23–27, 1994

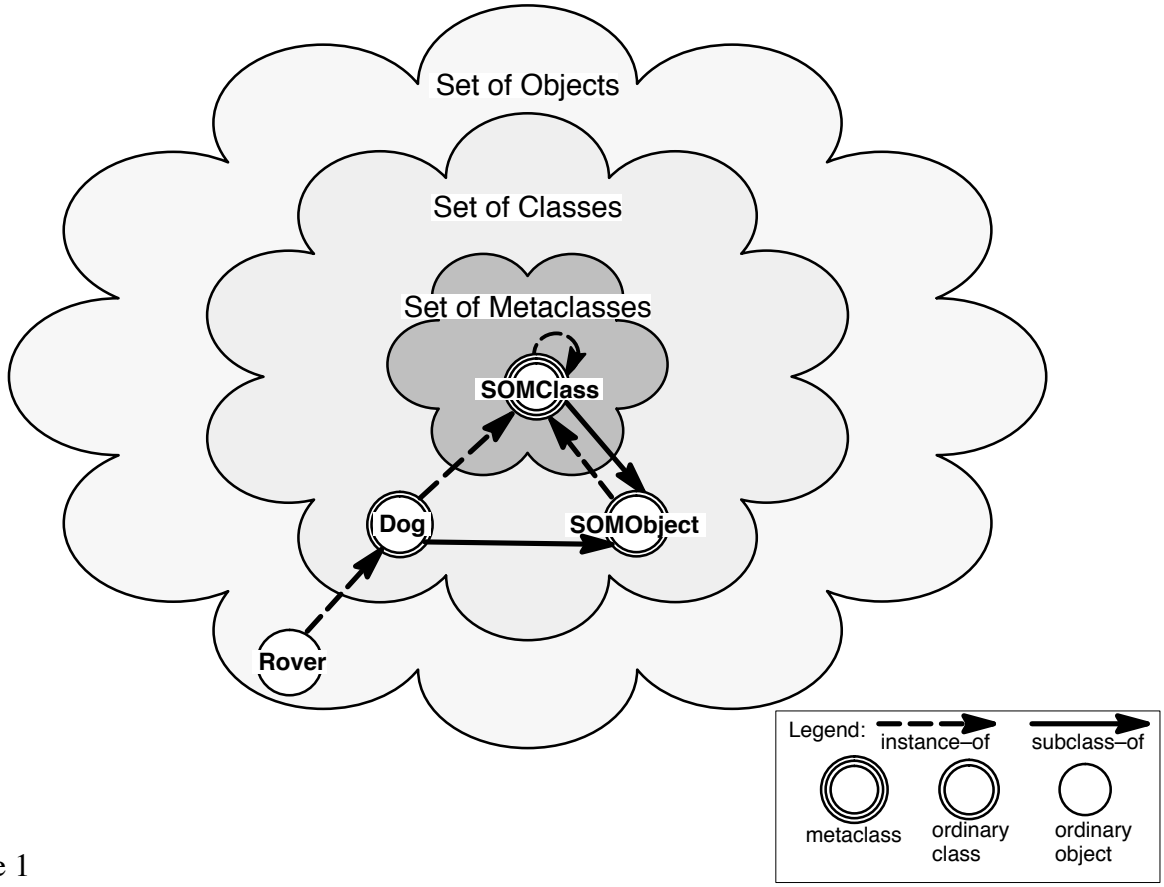

Figure 1

method table. Once this is done, other methods are invoked on the class to override inherited methods or add new instance methods.

When diagramming class hierarchies, this paper uses the convention that metaclasses are drawn with three concentric circles, ordinary classes (i.e., classes that are not metaclasses) are drawn with two concentric circles, and ordinary objects (i.e., objects that are not classes) are drawn with a single circle. The initial state of an example SOM program is depicted in Figure 1. There are four objects SOMObject (a class), SOMClass (a metaclass), Dog (an ordinary class), and Rover (an ordinary object). There are two relations among objects that one must understand.

First, there is the *instance of* relation between objects and classes depicted by the dashed arrow from an object to its class. When convenient the inverse relation, *class of*, is also used. SOMObject is an instance of SOMClass and SOMClass is the class of itself. An object's class is important because an object responds only to the methods that are supported by its class (that is, the methods that the class introduces or inherits).

Second, there is a relation between classes called the *subclass of* relation, which is depicted by the solid arrow from a class to each of its *parents*. SOMClass is a subclass of SOMObject . SOMObject has no parents.

SOMObject introduces the methods to which all SOM objects respond. In particular, SOMObject introduces the somDispatch method. This method provides a single, general dynamic dispatch mechanism for executing method calls on objects. Furthermore, a class can arrange its instance method table so that all method calls are routed through somDispatch. As a result, it is simple for SOM metaclass programmers to arrange for completely arbitrary processing in connection with method invocations on SOM objects.

As a subclass of SOMObject, SOMClass is an object but in addition introduces the methods to which all classes respond. For example, SOMClass introduces the somNew method, which creates instances of a class. Also, the methods responsible for creating and modifying instance method tables are introduced. All metaclasses in SOM are ultimately derived from SOMClass. (Similar arrangements of classes is also used in CLOS[12], ObjVlisp[5], Dylan[2], and Proteus[21].) With the SOM API, one can create new abstractions by programming metaclasses. In more general terms this has been referred to as a metaobject protocol [12,13] or computational reflection [15]. The strength of this general approach is that new abstractions can be created after the object model is implemented. That is, the Before/After Metaclasses are not part of the SOM kernel, rather they are part of a framework for programming metaclasses (see [9] for more information) that is built with the SOM API. Thus, by providing a metaobject protocol, we were able to a new abstraction to SOM.

Interfaces to SOM objects are described using IDL, an object interface definition language defined by the Common Object Request Broker Architecture (CORBA [16]) standard of the Object Management Group (OMG). SOM IDL is a CORBA–compliant version of IDL used to allow SOM class descriptions to be supplied in addition to object interface definitions. (That is, the interface to a class is described by the IDL alone, SOM IDL allows additional information about the implementation to be added.) The SOMobjects Toolkit has tools called emitters that translate SOM IDL into language–specific bindings for the corresponding classes of SOM objects (e.g., for C programmers this means that emitters produce header files for both the users of the class and the implementor of the class).

Below is the basic structure of an IDL definition for an object interface named Dog. At the same time, it is a SOM IDL description of a class Dog that supports this interface. The #ifdef and #endif (which, for simplicity, are omitted from subsequent examples) are part of the IDL language and are used to hide the SOM class implementation section from non–SOM IDL compilers.

```
interface Dog : SOMObject
 {
   method and attribute declarations here
  #ifdef  __SOMIDL__
  implementation
   {
      metaclass = SOMClass;
        instance variable declarations here
    };
   #endif
 };
```

In this example the interface Dog inherits from the SOMObject interface, and at the same time, the class Dog is declared to be a subclass of SOMObject . CORBA and SOM support multiple inheritance; additional parents of Dog can be listed alongside SOMObject in a comma–separated list. The actual methods and instance variables of Dog are not relevant to the current discussion.

As illustrated here, the implementation section can explicitly indicate a metaclass to be associated with the class of objects that support the interface being defined. This association is not necessarily direct, however. For reasons that will become clear, the actual class of the class described by any given SOM IDL is, in general, a subclass of the indicated metaclass.

Before/After Metaclasses

A *before method* is a behavior that precedes the action of some program construct. An *after method* is a behavior that succeeds the action of some program construct. Before and after methods are familiar to users of CLOS [12,19], where the granularity of application is the individual method. In the class–based object model SOM, the more natural granularity for before/after methods is the class, because there are many applications that fit this granularity (see next section). The SOMMBeforeAfter metaclass therefore introduces two methods BeforeMethod and AfterMethod that its instances (classes) arrange to run respectively before and after each instance method. By default, these two methods do nothing — to define a specialized before/after behavior, one creates a subclass of SOMMBeforeAfter and overrides the BeforeMethod and the AfterMethod with the desired behavior.

```
interface BarkingDog : Dog
{
   /* no method declarations
*/
  implementation
   {
      metaclass = Barking;
     /* no instance variables */
    };
};
```

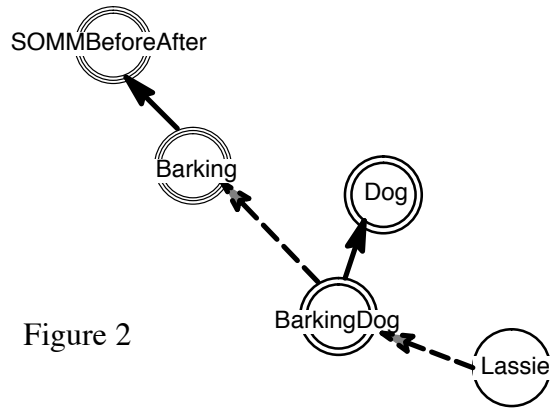

Figure 2

For example, consider the arrangement in Figure 2. The Barking metaclass overrides BeforeMethod and AfterMethod with a method that makes a "woof" sound when executed. As a result, all methods supported by the class BarkingDog (an instance of Barking) have this before/after behavior. That is, the object Lassie goes "woof" before and after each method invoked on it runs, because it is an instance of BarkingDog. The IDL for the BarkingDog class is given at the right of the figure. Note that the IDL is the only source code that needs to be written; the compiler of the SOMobjects Toolkit can generate from IDL all the necessary code to implement BarkingDog (in either C or C++).

The essence of the workings of SOMMBeforeAfter is a method that overrides the method somDispatch introduced by SOMObject. [1] The new dispatcher looks like this:

```
somDispatch ( self, primaryMethod, ... )
  BeforeMethod( class(self),
                      self,
                      primaryMethod, ... );
  retval := primaryMethod( self, ... );
  AfterMethod( class(self),
                    self,
                    primaryMethod,
                     retval, ... );
  return retval;
```

where primaryMethod is the method being invoked on a target object (e.g., Lassie) for which before/after behavior is desired. Note that in the psuedo–code used in this paper, the first parameter to a method invocation is always the target object. The ellipses represent all the other actual parameters to the method. As noted earlier, the metaclass of the object Lassie supports BeforeMethod; that is, the class BarkingDog responds to BeforeMethod. This is why, in the above pseudo–code, class(self) is the target object for the BeforeMethod and AfterMethod method invocations.

The Usefulness of Before/After Metaclasses

Before attacking the problem of composition of Before/After Metaclasses, let us pause to consider their usefulness. As mentioned earlier, CLOS has the notion of before/after methods, but our experience indicates that the more useful granularity for a class–based object model is the class. Foote and Johnson [10] advocate class–level granularity. So does Pascoe [20] (but his encapsulators apply to all method invocations on a class instance rather than a class). This is also the granularity used by the Demeter system [14] (which does the implementation by source code transformation).

Software engineering of classes has many examples of uses of before/after methods. Method tracing is a primary example of a useful software engineering tool that fits naturally into the before/after paradigm at the class granularity. Another example is invariant checking; one

[1] Those readers not familiar with SOM may skip this note. Actually, for the sake of efficiency, methods in SOM are usually invoked directly and the somDispatch method is not generally called. In this scheme the SOMMBeforeAfter metaclass arranges for somDispatch to be invoked by placing stubs in the method table to call somDispatch (this capability is part of the SOM API). The SOMMBeforeAfter metaclass also arranges that the contents of the original method table be saved so that somDispatch can invoke the primary method. In addition, the SOMMBeforeAfter metaclass ensures that somDispatch does not dispatch itself (which would cause a dispatch loop). The details of how this is done are very specific to the SOM API and beyond the scope of this paper. More information on the SOM API may be found in [24].

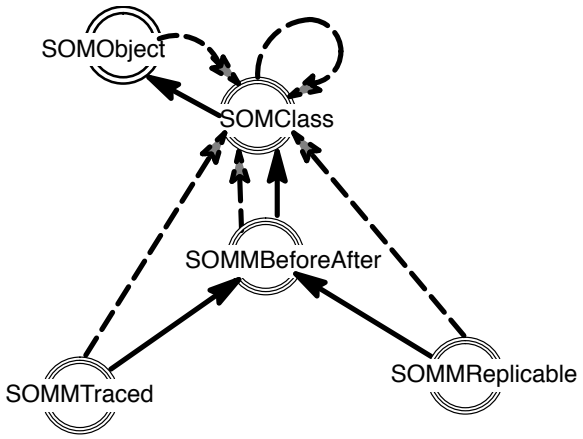

Figure 3

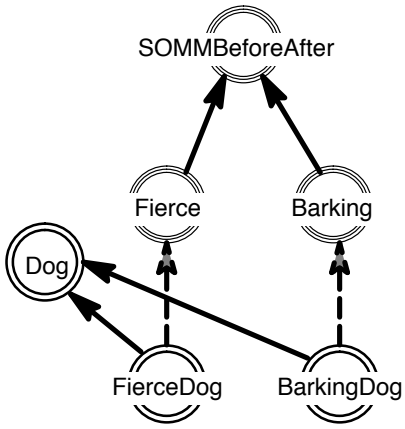

Figure 4

could imagine a metaclass which checks the invariant supplied by the class programmer as a method on the class. In addition to its reusability, such a metaclass would have the advantage of ensuring that the invariant is checked when new methods are added to the class. Other types of verification and monitoring are also feasible (for example, path expressions [4] or behavioral expressions [1]).

Concurrency yields other opportunities to use Before/After Metaclasses. For example, one can factor atomicity into a metaclass; the before method acquires a semaphore and the after method releases the same semaphore. In addition, we have found that Before/After Metaclasses provides a convenient way to offer framework capabilities to customers. The SOMObjects Toolkit contains a framework for creating replicated objects [24]; this framework has a set of rules for conveying the replicated property to a class. Basically the rules require the locking a set of replicas prior to an update, followed by the propagation and unlock after the update (the objective is to ensure one–copy serializability). The majority of the work required by this set of rules can done by a Before/After Metaclass. We have used SOMMBeforeAfter to construct metaclasses for both tracing and replication (Figure 3).

We have seen that other frameworks in the toolkit could similarly be aided. Elsewhere, a detailed example of how to use a metaclass in CLOS to make a class persistent is given in [18]. Suppose one wishes to provide a class library that has n classes. In addition, suppose there are p properties that must be included in all combinations for all classes. Potentially, the library must have $n2^p$ classes. Let us hypothesize that (fortunately) all these properties can be captured by before/after metaclasses, the size of the library is $n+p$. The user of the library need only produce those combinations necessary for his applications. This problem is one faced by users of some object–oriented databases and has arisen in the design of the OMG Persistence Standard [6]. When one considers this situation, one obvious conclusion is unavoidable: Before/After Metaclasses are not useful unless they compose, because if not, the use of one Before/After Metaclass would preclude the use of others. (Of what good is a trace metaclass, if it cannot be used to debug instances of the replicable metaclass?)

The Composition Problem

Now, consider Figure 4 in which there are before/after metaclasses Barking (as before) and Fierce, which has a BeforeMethod and AfterMethod that both growl. That is, both make a "grrrr" sound when executed. It should be clear that we can now create a FierceDog or a BarkingDog, but we have not yet addressed the question of how to compose the properties of fierce and barking. Composability means having the ability to easily create a FierceBarkingDog that goes "grrr woof woof grrr" when it responds to a method call, or a BarkingFierceDog that goes "woof grrr grrr woof" when it responds to a method call.

The problem of composing the properties of fierce and barking is complicated by the fact that there are several

ways in which one might express such compositions. Figure 5 depicts three techniques in which such a composition might naturally be indicated by a programmer. These are labelled Technique 1, Technique 2, and Technique 3, which create the FierceBarkingDog classes named FB–1, FB–2, and FB–3, respectively. The SOM IDL for each of these classes is given above a diagram that depicts the context in which the class description is given.

In Technique 1, a new metaclass (FierceBarking) is created with both Fierce and Barking as parents; an instance of this new metaclass (that is, FB–1) should be a FierceBarkingDog (if Dog is a parent).

In Technique 2, a new class is created that has parents that are instances of Fierce and Barking respectively; that is, FB–2 should be a FierceBarkingDog too (assuming FierceDog and BarkingDog do not further specialize Dog).

In Technique 3, FB–3, which should also be a FierceBarkingDog, is created by a declaring that its parent is a BarkingDog and that its explicit (syntactically de-

Technique 1

```
interface FB–1 : Dog
{
   /* no method declarations
*/
  implementation
   {
     metaclass = FierceBark-
ing;
     /* no instance variables */
    };
};
```

Technique 2

```
interface FB–2 : FierceDog,
                 BarkingDog
{
   /* no method declarations
*/
  implementation
   {
     /* no instance variables */
    };
};
```

Technique 3

```
interface FB–3 : Barking-
Dog
{
   /* no method declarations
*/
  implementation
   {
     metaclass = Fierce;
     /* no instance variables */
    };
};
```

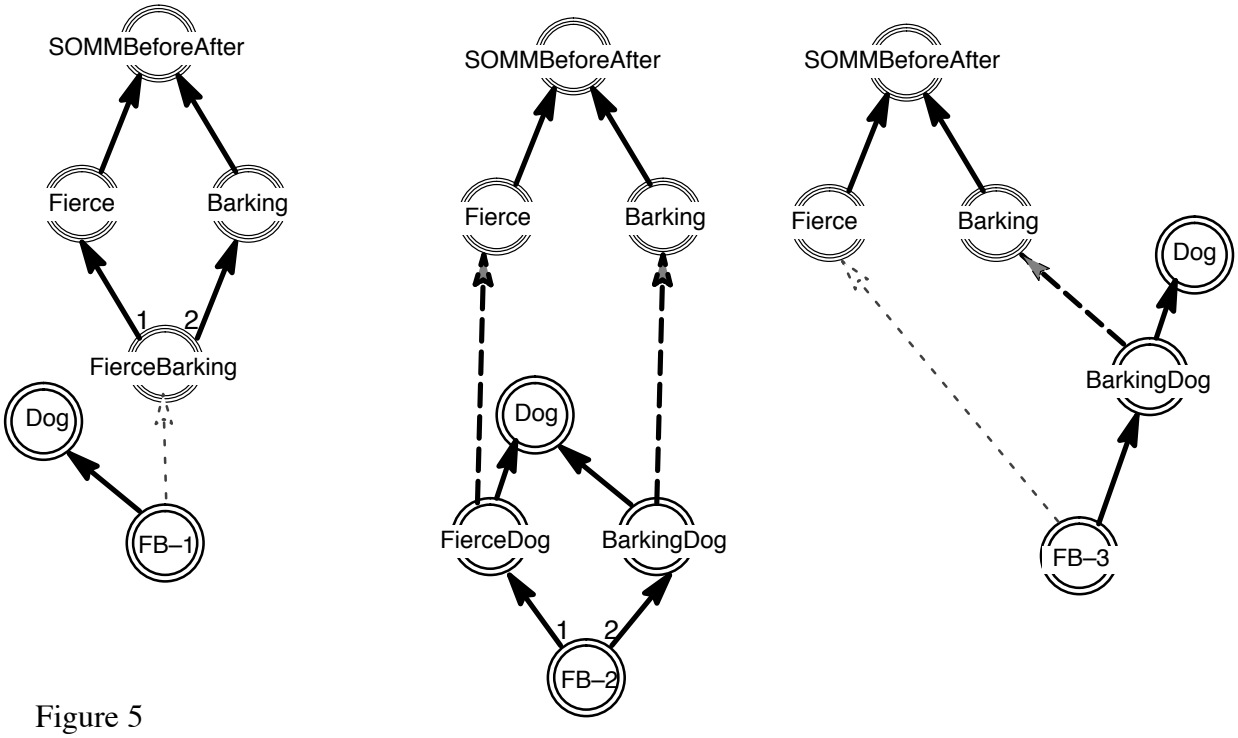

Figure 5

clared) metaclass (drawn with the light dashed arrow) is Fierce.

Looking at Figure 5, we ask the question: "should the three techniques produce the same result?" That is, should FB–1, FB–2, and FB–3 be equivalent classes (that is, behave the same and have instances that behave the same)? The answer must be "YES" because composition of metaclasses must be easily understood by the programmer. Non–equivalence of these three techniques would certainly lead to a system in which programming is complex and error–prone. This conclusion leads us to ask what common property these techniques have upon which an equivalence may be based. There is such a property, and, as described in the following section, SOM provides special support for it.

The Derived Metaclass in SOM

SOM allows and encourages the definition and explicit use of metaclasses. At the same time, however, SOM relieves programmers of the responsibility *for getting the metaclass right* when defining a new class. At first glance, this might seem to be merely a useful (though very important) convenience. But, in fact, it is absolutely essential in SOM. This is because SOM is predicated on binary compatibility with respect to changes in class implementations. Even though a programmer might, at one time, know the metaclasses of all classes above a new subclass, and, as a result, be able to explicitly derive an appropriate metaclass for the new class, SOM must guarantee that this new class still executes correctly when any of its ancestor class's implementations are changed (and this could include a choice of different metaclasses). Thus, a SOM programmer never needs to consider a newly defined class's ancestors' metaclasses. Instead, explicit metaclasses should only be used to *add in* desired behavior for a new class. Anything else that is needed is done automatically [12].

To understand this better, consider the simple single–inheritance example illustrated by Figure 6. In this figure, A is an instance of AMeta; we assume that AMeta supports a method bar and that A supports a method foo that uses the expression bar(class(self)). That is, the method foo invokes a method on the class of the object on which foo is operating. Now consider what happens when A is subclassed by B, a class that has an explicit metaclass declared in its SOM IDL as in Figure 6. If the class hierarchy were to be formed as in Figure 6, then an invocation of foo on an instance of B would fail because BMeta does not support bar. This situation is referred to the as metaclass incompatibility. SOM does not allow hierarchies with metaclass incompatibilities. Instead, SOM builds derived metaclasses that prevent this problem from occurring. The actual SOM class hierarchy that results for B is depicted in Figure 7, where SOM has

```
interface B:A {
   ...
 implementation {
   metaclass = BMeta;
  ...
  };
};
```

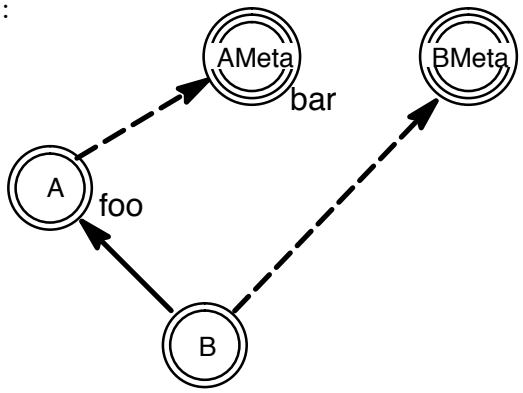

Figure 6

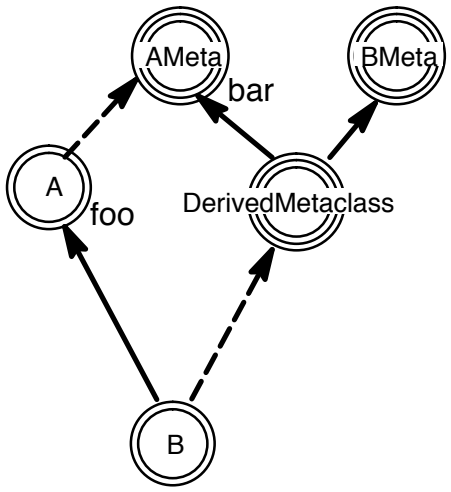

Figure 7

automatically built the metaclass DerivedMetaclass; this ensures that the invocation of foo on instances of B do not fail. This example shows that the metaclass statement in the SOM IDL is treated as a constraint on the actual metaclass. The derived metaclass can be viewed as the minimal metaclass supporting the constraints of metaclass compatibility.

Several papers [3,11] have called this situation "the metaclass compatibility problem," but none go beyond a characterization of the compatibility condition required on the metaclass statement. In SOM there is no such problem; in a situation where the explicitly declared metaclass is not compatible with the parents of the class, an appropriate metaclass is constructed — this is the derived metaclass. Because class construction is a dynamic activity in SOM, this derivation is actually accomplished at run–time with no need for prior description in IDL. [2]

The Composition Solution

We now return to an examination of the techniques in Figure 5. Here, the derived metaclass constructed by SOM for FB–3 will be FierceBarking, not Fierce as indicated in the SOM IDL. Now look at the diagram for Technique 2 of Figure 5; here also, although the metaclass of FB–2 is not explicitly declared, the derived metaclass provided by SOM for FB–2 will be FierceBarking.

Figure 8 combines the diagrams in Figure 5 and shows the actual class relationships (which are established when the class objects are instantiated). Note that the explicit metaclass in the SOM IDL of FB–1 is its derived class FierceBarking; the derived metaclass of FB–2 is also FierceBarking. However, the derived metaclass of FB–3 is not the explicit metaclass in the SOM IDL, rather it too is FierceBarking. The solution to the composition problem jumps out at us. The common element among the three techniques for expressing before/after composition is the metaclass FierceBarking, which is the class of FB–1, FB–2, and FB–3 (in the case of FB–1 the relationship is explicit and in the cases of FB–2 and FB–3 the relationship is derived). Therefore we conclude: composition can be based on the "completed" metaclass hierarchy that results from the use of derived metaclasses in SOM.

Now that the general nature of a common solution has been presented, it remains to discuss the details of this solution. By the nature of before/after, the main dispatch function is changed to look like this:

```
somDispatch ( self, primaryMethod, ... )
  BeforeMethodDispatch( class( class ( self )),
                                    self, ... );
  retval := primaryMethod( self, ... );
  AfterMethodDispatch( class( class ( self )),
                                  self, ... );
  return retval;
```

where primaryMethod is the identifier of the method being invoked by the application and acquiring before/after behavior. To produce an appropriate composed order of before method invocations, BeforeMethodDispatch does a preorder traversal of the metaclass hierarchy towards SOMClass looking for the first metaclass on each path that defines BeforeMethod; each such BeforeMethod is then invoked on the client object. The BeforeMethodDispatch implementation thus looks like this:

```
BeforeMethodDispatch( aMetaclass,
                                  clientObject, ... )
  if  aMetaclass defines BeforeMethod
      BeforeMethod( clientObject, ... )
  else
      for all parents of aMetaclass
                  that support BeforeMethod
          BeforeMethodDispatch( aParent,
                                  clientObject,
                                      ... )
```

The search restriction to metaclasses that support BeforeMethod is based on the fact that a before/after metaclass may have parents that are not before/after metaclasses (i.e., parents that are not descendents of SOMMBeforeAfter). Note that if the metaclass of the client ob-

[2] CLOS does not allow the construction of metaclass incompatibilities. When a class is constructed validate–superclasses is called to ensure that all superclasses have the same metaclass as the class being constructed (see [19] page 84 or [12] pages 240–241); if this condition is not true, an error is signalled.

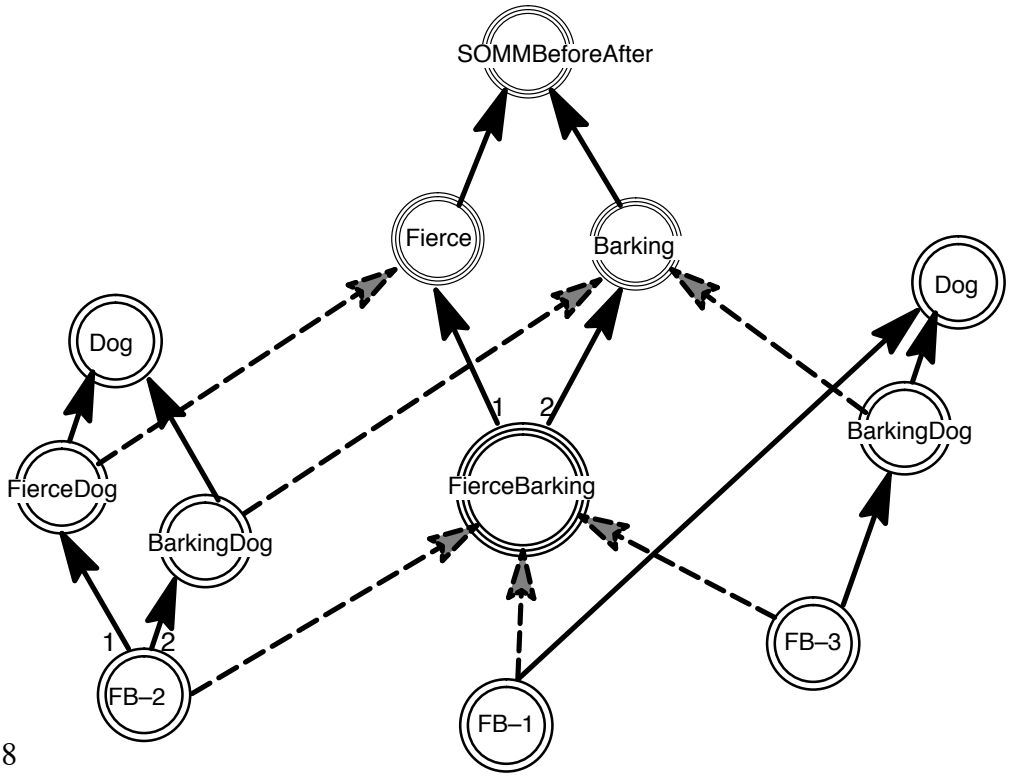

Figure 8

ject defines a BeforeMethod, the search succeeds and only one BeforeMethod is called. The rationale for this is simple: the creator of a before/after metaclass knows the parents of this metaclass, and, when overriding BeforeMethod, can be expected to explicitly invoke any inherited functionality that is necessary (using parent method calls).

The AfterDispatchMethod is quite similar except that the parents are traversed in the reversed order. The AfterDispatchMethod implementation looks like this:

```
AfterMethodDispatch( aMetaclass,
                              clientObject, ... )
  if  aMetaclass defines AfterMethod
      AfterMethod(  clientObject, ... )
  else
      for all parents of aMetaclass
                that support AfterMethod
               (in reverse order)
          AfterMethodDispatch( aParent,
                                  clientObject,
                                      ... )
```

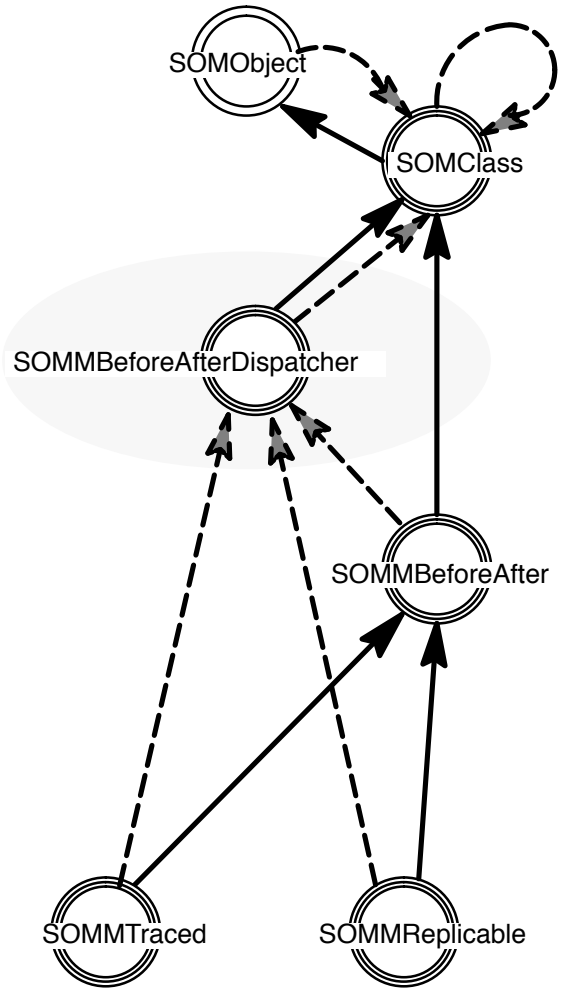

Figure 9

One final issue remains to be addressed: What class introduces the methods BeforeMethodDispatch and AfterMethodDispatch? These are methods to which all subclasses of SOMMBeforeAfter respond [3]; thus the answer is that they must belong to the class of SOMMBeforeAfter. Therefore, as shown in Figure 9, the class of SOMMBeforeAfter becomes the metaclass SOMMBeforeAfterDispatcher. One might call SOMMBeforeAfterDispatcher a *meta–metaclass* because its instances are metaclasses. Figure 9 thus replaces Figure 3, in terms of an overall class design.

Loose Ends

The facility described in this paper has been available inside of IBM since August, 1993; for example, the traced metaclass is used in parts of the SOMobjects Toolkit (version 2.0). The facility will have general availability as part of SOMobjects Toolkit (version 2.1). Before concluding, there are several issues to address, which for the sake of ease of presentation have been ignored until this point.

First, our solution to the composition of Before/After Metaclasses has a pleasant property; composition is associative. Figure 10 shows three Before/After Metaclasses A, B, and C that introduce three before methods respectively (Before1, Before2, and Before3). Metaclass F represents the composition (AB)C and G represents the composition A(BC). Both compositions lead to the same sequence of before method invocations (that is, Before1;Before2;Before3). Of course, composition is not commutative, e.g., a FierceBarkingDog is not the same as BarkingFierceDog (which goes "woof grrr grrr woof"). The order of the metaclasses depends on the search order which is determined by the order of the parents.

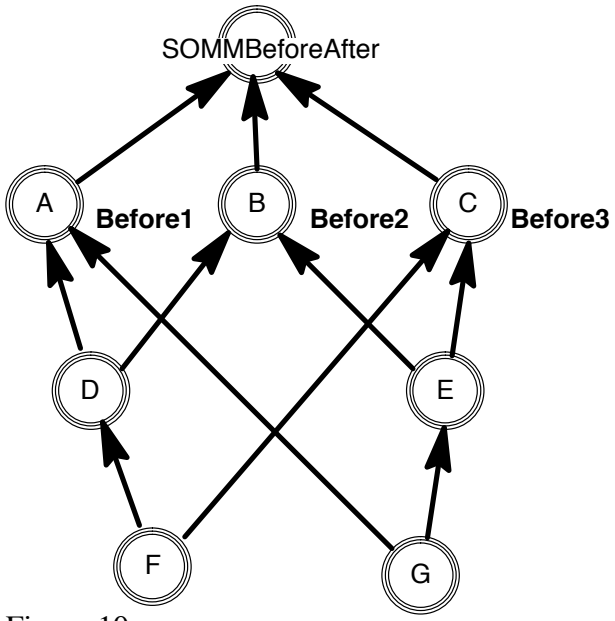

Figure 10

Second, there is the issue of exempting a method from the before/after behavior. We have seen situations for different before/after methods that range from exempting a particular primary method to exempting a particular kind of method (e.g., read–only methods) to exempt-

3 We say that an object responds to a method. Thus, all objects respond to the methods supported by its class. If the object is a class, the class inherits the methods supported by its parents, but responds to the methods supported by its metaclass.

ing all methods inherited from a particular parent. We have found that the simplest solution is to have the designer of the particular Before/After Metaclass determine the scheme for method exemptions. Usually this comprises of the introduction (by the metaclass designer) of a predicate method to be called by the BeforeMethod and the AfterMethod; if the predicate returns true, both wrapper methods return without doing anything.

Third, for the sake of simplicity of discourse, the before (after) dispatch method (presented above) does not check whether the before (after) method has already been executed during the search of the metaclass hierarchy. (For example, if in Figure 10 one adds an H as a subclass of both F and G, all the before methods are executed twice.) There are a number of techniques to solve this problem. So far, no case has arisen where it is desirable to have multiple executions of before/after methods.

Another issue is having the before method determine that the primary method should not be run. To handle this case, the before method is extended to return a boolean to the dispatcher, which contains true if the primary method is to be run. If the primary method is not run, neither is the corresponding after method. Because the before and after methods belong to the same metaclass, the before method can do any tidying up the after method might have been required to perform (this obviates the need to communicate to the after method that the primary method was not invoked).

Before/After Metaclasses in CLOS

Although CLOS has multi–methods (and is not a language in which classes are thought of as containing methods), a comparison is instructive. The language–defined granularity of application for before/after behavior in CLOS is the method. That is, one may define a method with a qualifier (either :before or :after); such qualified methods are invoked before and after primary methods of similar applicability.

This granularity has its uses, but does not qualify as defining an operation on classes, which is one of the objectives of this paper. Our position is that before/after methods come in pairs that are applied over all methods of a class. Of course, this comparison is too superficial, because the advantage of having a metaobject protocol in CLOS lies in the ability to define a similar operation to the one we have defined here using SOM. The two features of CLOS that are most relevant are the call–next–method method and the apply–methods function.

In CLOS, all ancestor classes are kept on a list, called the *class precedence list*. One uses call–next–method to invoke the method with the same name (as the currently executing method) from the next entry in the class precedence list. This is the approach used by CLOS for calling parent methods in a multiple inheritance model. The well–known problem with this is that the ordering of the class precedence list is fixed even though different methods may require different execution orderings for parent's implementations. In SOM, much as in C++, the programmer identifies the specific parent(s) whose code should be invoked.

In the CLOS metaobject protocol, all method invocations are dispatched by the apply–methods function (see [12] page 43). This function invokes the :before methods, then the primary methods, and finally the :after methods. Here :before and :after are the method qualifiers that indicate the position of the method in the dispatch order with respect to the primary methods. Note that there is one apply–methods function for the entire program (where as in SOM, each class has its own somDispatch method).

These differences between SOM and CLOS mean that the natural solutions to implementing before/after class behavior in CLOS looks different than the natural solution for SOM that is presented in this paper. One "CLOS–natural" approach to having before/after class behavior might look as follows.

- At class instantiation time, one could use the current mechanism of CLOS to "apply" the class's before/after method to all primary methods of the class. This leaves open the problem of how the class–level before/after methods should be specified, but let's not worry about it. Composition is achieved if the before/after pairs are applied in the same order, because one of the CLOS functions that controls method dispatch (apply–methods)

reverses the order of execution of the after methods. If one desires to have preemption (as described in the previous section), apply–methods would have to be further modified. An additional design issue in this solution concept is that of ensuring that dynamically–added methods (methods added after class instantiation) receive the before/after methods.

Two other possible CLOS approaches suggest themselves.

- Because all method invocations in CLOS are dispatched through apply–methods , before/after methods for classes could be implemented in CLOS by changing apply–methods to invoke the BeforeMethodDispatch before apply–methods and the AfterMethodDispatch afterwards. Of course, this requires that derived metaclasses be adopted by CLOS in order to use the metaclass hierarchy to determine the before/after behavior.

- Because of the way call–next–method works in CLOS, another scheme is possible. Suppose for the moment that each CLOS class has a dispatch method, CLOSDispatch, i.e., all method invocations for a class first call the CLOSDispatch method which in turn invokes the primary method. This could also be effected with a change to apply–methods to call CLOSDispatch to invoke the primary method. Based on this change, one could implement composition of Before/After Metaclasses in CLOS by overriding CLOSDispatch with

```
(defun CLOSDispatch ( ... )
        (BeforeMethod ...)
        (call–next–method)
        (AfterMethod ...))
```

  This scheme allows the before/after behavior to be inherited from classes that override CLOSDispatch; composition is attained from the way call–next–method chains its way up the class hierarchy. This approach has the disadvantage of not allowing the overriding of either before methods or after methods.

Although neither of these approaches could be called "CLOS–natural" (because they adopt CLOS to use features of SOM), they isolate two ways in which the SOM solution differs from a CLOS solution.

Conclusions

The central issue addressed by this paper is raising the level of programming by composing metaclasses. The standard notion of inheritance–based subclassing represents a union–like operation for composition of class implementations. There is no reason to believe that one such operation is sufficient for all the possible compositions that need to be performed. With the approach described here, we believe that significant object properties can be implemented using before/after metaclasses and that these properties can be subsequently composed. Linguistically, a property is often represented by an adjective while a class is represented by a noun. Composition of metaclasses should be as easy as putting a sequence of adjectives in front of a noun when we speak.

Acknowledgements

Mike Conner and Larry Raper are the designers of the SOM model and API; their insight in providing SOM with metaclasses provides the basis upon which we worked. We wish to thank Liane Acker, Gregor Kiczales, and Harold Ossher for their comments on earlier drafts of this paper. In addition, the comments of the OOPSLA referees were very valuable and greatly appreciated.

References

1. Atkinson, C. *Object–Oriented Reuse, Concurrency and Distribution: An Ada–based Approach* Addison–Wesley (1991).
2. Apple Computer *Dylan: An object–oriented dynamic language* (1992).
3. Briot, J.–P. and Cointe, P. Programming with Explicit Metaclasses in Smalltalk–80 *OOPSLA '89 Conference Proceedings* (October 1–6, 1989) 419–431.
4. Campbell, R.H. and Habermann, A.N. The Specification of Process Synchronisation by Path Expressions *Lecture Notes in Computer Science* Volume 16, Springer–Verlag (1974) pp. 89–102.
5. Cointe, P. Metaclasses are First Class: the ObjVlisp Model *OOPSLA '87 Conference Proceedings* (October 4–8, 1987) 156–165.

6. Copeland, G. private communication 1994.
7. Danforth, S. A Bird's Eye View of SOM *IBM OS/2 Developer* (Winter 1992).
8. Danforth, S. and Forman, I.R. Derived Metaclasses in SOM *Proceedings of the 1994 Conference on Technology of Object–Oriented Languages and Systems* Versailles, France (April 1994)
9. Danforth, S. and Forman, I.R. Reflections on Metaclass Programming in SOM *OOPSLA '94 Conference Proceedings* (October 23–27, 1994).
10. Foote, B. and Johnson, R.E. Reflective Facilities in Smalltalk–80 *OOPSLA '89 Conference Proceedings* (October 1–6, 1989) 327–335.
11. Graube, N. Metaclass Compatibility *OOPSLA '89 Conference Proceedings* (October 1–6, 1989) 305–316.
12. Kiczales, G., des Rivieres, J., and Bobrow, D. G. *The Art of the Metaobject Protocol* The MIT Press, Cambridge, Massachusetts (1991).
13. Kiczales, G. and Paepcke, A. *Open Implementations and Metaobject Protocols* The MIT Press, Cambridge, Massachusetts (1994).
14. Lieberherr, K.J. and Xiao, C. Object–Oriented Software Evolution *IEEE Trans. on Software Engineering* **19**(4) (April 1993) 313–343.
15. Maes, P. Concepts and Experiments in Computational Reflection *OOPSLA '87 Conference Proceedings* (October 4–8, 1987) 147–155.
16. Object Management Group *The Common Object Request Broker: Architecture and Specification* OMG Document Number 91.12.1 Revision 1.1.
17. *OS/2 Technical Library System Object Model Guide and Reference* Document S10G6309, IBM Corp., Armonk, N.Y. (1991).
18. Paepcke, A. PCLOS: A Critical Review *OOPSLA '89 Conference Proceedings* (October 1–6, 1989) 221–237.
19. Paepcke, A. (ed.) *Object–Oriented Programming: The CLOS Perspective* The MIT Press, Cambridge, Massachusetts (1993).
20. Pascoe, G.A. Encapsulators: A New Software Paradigm in Smalltalk–80 *OOPSLA '86 Conference Proceedings* (September 29 – October 2, 1986) 341–346.
21. Russinoff, D.M. Proteus: A Frame–Based Nonmontonic Inference System *Object–Oriented Concepts, Databases, and Applications* Kim, W. and Lochovsky, F.H. (ed.) ACM Press, New York (1989) 127–150.
22. Sessions, R. and Coskun, N. Object–Oriented Programming in OS/2 2.0 *IBM Personal Systems Developer* (Winter 1992).
23. Sessions, R. and Coskun, N. Class Objects in SOM *IBM OS/2 Developer* (Summer 1992).
24. *SOMobjects Developers Toolkit – User's Guide and Reference Manual* IBM Corp., Armonk, N.Y. (1993).