Composition of Before/After Metaclasses in SOM

By Ira R. Forman, Scott Danforth, and Hari Madduri

In the IBM System Object Model (SOM), 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. This article 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; that is, at runtime a SOM system derives the appropriate metaclass of a class based on the classes of its parents and an optional metaclass constraint.

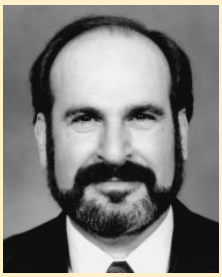
One way to view the history of programming is that progress is made by providing abstractions to ever larger entities and by ensuring the composability of those abstractions. In the beginning, assembly language instructions were gathered into control structures. Subsequently, control structures were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

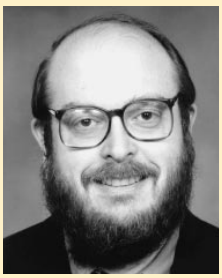
This article addresses a further abstraction—before/after metaclasses—and solves the problem of their composition. A before/after metaclass has a *before method* and an *after method* that are executed before and after the methods of the instances of its instances (an instance of a metaclass is a class, which in turn has instances). Applications for before/after metaclasses abound, including method tracing, invariant checking, path expression checking, and object locking. Given these opportunities, it is imperative that before/after metaclasses compose with one another.

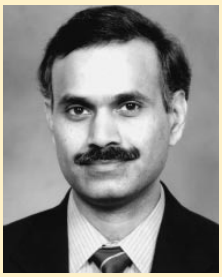
This article solves the before/after metaclass composition problem in the context of the IBM System Object Model (SOM). SOM is an object-oriented model [7,17,22,23]. The SOM runtime supports the model and allows programs written in arbitrary languages to use the model via the SOM Application Programming Interface (API). A binding is code that facilitates the use of a class or the implementation of a class. The SOMObject Toolkit[24] provides both usage and implementation bindings for C and C++. Language vendors for Smalltalk (Digitalk), COBOL (Micro Focus®), and C++ (IBM, Metaware, and Borland™) have also 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 method table. Once this is done, other methods are invoked on the class to override inherited methods or add new instance methods.


Ira R. Forman


Scott Danforth


Hari Madduri

© Copyright 1994. Association of Computing Machinery. Reprinted with permission from *OOPSLA '94 Conference Proceedings* (October 23–27, 1994).

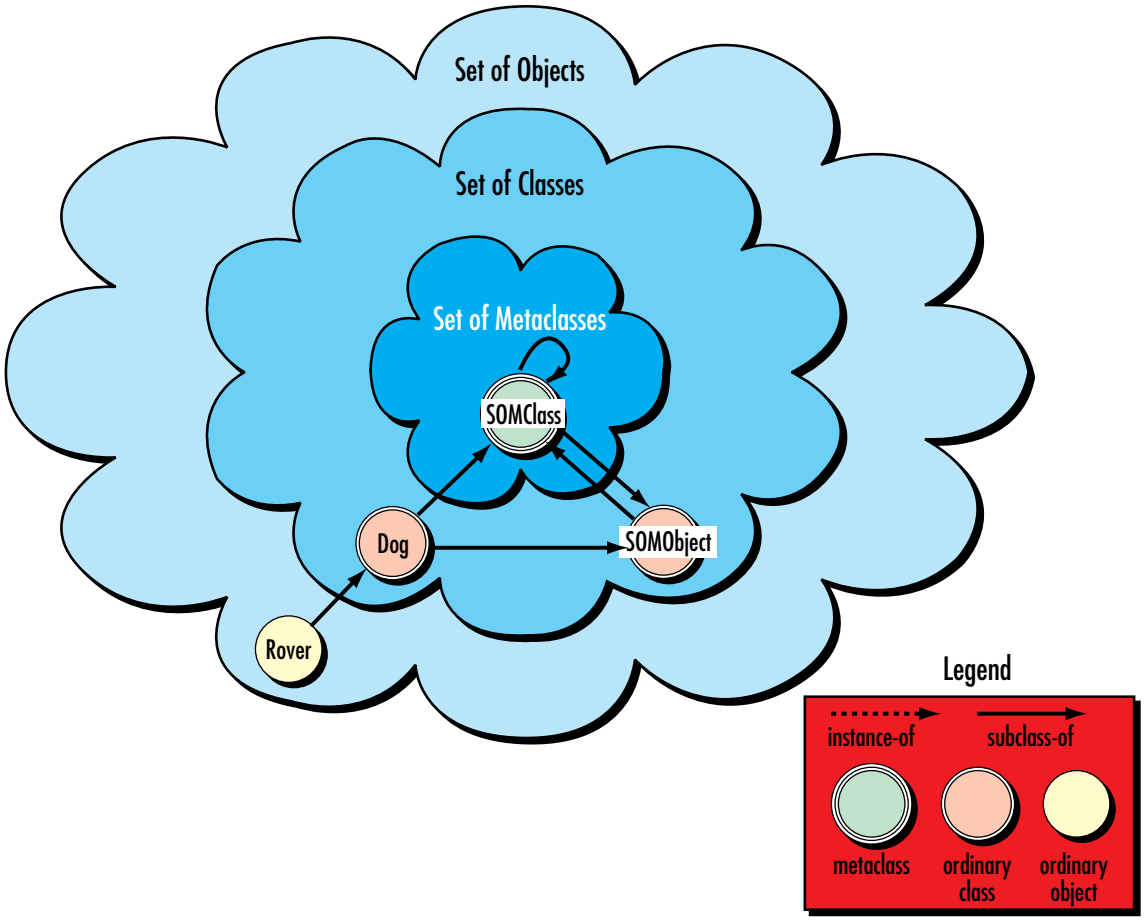


Figure 1. Structure of the SOM environment

When diagramming class hierarchies, this article uses the convention that metaclasses are drawn with three concentric circles; ordinary classes (classes that are not metaclasses) are drawn with two concentric circles; and ordinary objects (objects that are not classes) are drawn with a single circle. Figure 1 shows the initial state of an example SOM program. 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 must be understood:

1. 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).
2. 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. 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 it also introduces the methods to

which all classes respond. For example, SOMClass introduces the somNew method, which creates instances of a class. The methods responsible for creating and modifying instance method tables are also introduced. All metaclasses in SOM are ultimately derived from SOMClass. (Similar arrangements of classes are also used in CLOS[12], ObjVlisp[5], Dylan[2], and Proteus[21].) With the SOM API, you 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[9] that is built with the SOM API. Because SOM provides a metaobject protocol, we were able to add 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 (For C programmers, this means that emitters produce header files for both the users of the class and the implementor of the class).

Figure 2 shows 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 (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.

In Figure 2, the interface Dog inherits from the SOMObject interface; 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.

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

Figure 2. IDL definition for object interface Dog

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. For reasons that will become clear, the actual class of the class described by any given SOM IDL is generally 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 class-based object models such as 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 —arranged by its instances (classes) to run respectively before and after each instance method. By default, these two methods do nothing. To define a specialized before/after behavior, it is necessary to create a subclass of SOMMBeforeAfter and override the BeforeMethod and the AfterMethod with the desired behavior.

In Figure 3, 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. The IDL is the

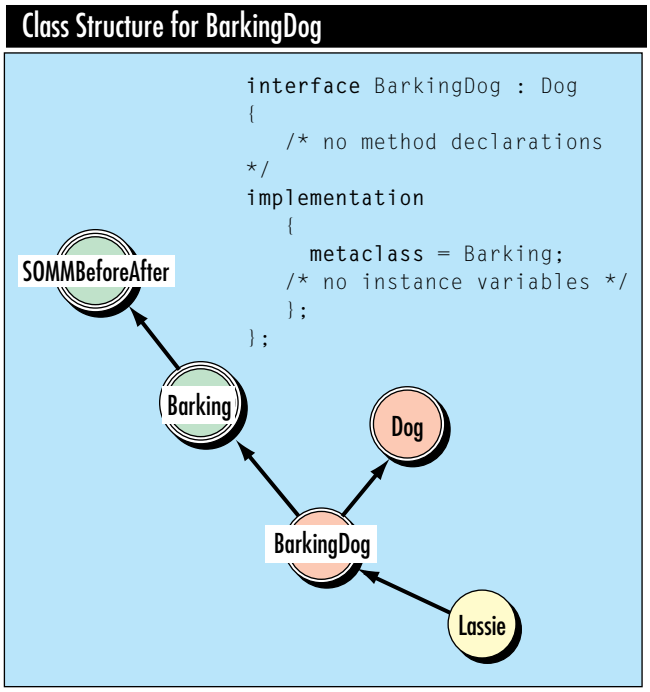


Figure 3. BarkingDog class structure

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

Figure 4. New dispatcher for SOMObject

only source code that needs to be written; the compiler of the SOMObjects Toolkit can generate all the necessary code from IDL 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`*. The new dispatcher looks like Figure 4.

In Figure 4, `primaryMethod` is the method being invoked on a target object, such as `Lassie`, for which before/after behavior is desired. In the pseudo-code used in this article, 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.

Usefulness of Before/After Metaclasses

Before attacking the composition problem of before/after metaclasses, 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. Pascoe[20] does also, 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[14] system, 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. Imagine a metaclass that checks the invariant supplied by the class programmer as a method on the class. In addition to its reusability, this type of 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, such as path expressions[4] or behavioral expressions[1], are also feasible.

Concurrency yields other opportunities to use before/after metaclasses. For example, atomicity can be factored 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 provide 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

* Those readers not familiar with SOM may skip this note. 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 article. See reference 24 for more information SOM API.

for conveying the replicated property to a class. The rules require locking a set of replicas before 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 be done by a before/after metaclass. We have used `SOMMBeforeAfter` to construct metaclasses for both tracing and replication (see Figure 5).

Other frameworks in our toolkit could similarly be aided. Elsewhere, Paepcke[18] provides a detailed example of how to use a metaclass in CLOS to make a class persistent. Suppose you wish 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.

If we hypothesize that (fortunately) all these properties can be captured by before/after metaclasses, the size of the library is `n+p`. The users of the library need only produce those combinations necessary for their applications. This problem is one faced by users of some object-oriented databases and has also arisen in the design of the OMG Persistence Standard[6]. In 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. (What good is a trace metaclass if it cannot be used to debug instances of the replicable metaclass?)

The Composition Problem

Suppose there are before/after metaclasses `Barking` (as before) and `Fierce` (shown in Figure 6), which has a `BeforeMethod` and `AfterMethod` that both "growl"—both make a "grrrr" sound when executed. We can now create a `FierceDog` or a `BarkingDog`, but we still have not 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.

Composing the properties of fierce and barking is complicated because there are several ways in which these compositions can be expressed. Figure 7 shows three techniques in which a programmer might naturally indicate such a composition. These are labeled 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 class is given above a diagram that depicts the context in which the class description is given.

SOM 2.1 Metaclass Framework

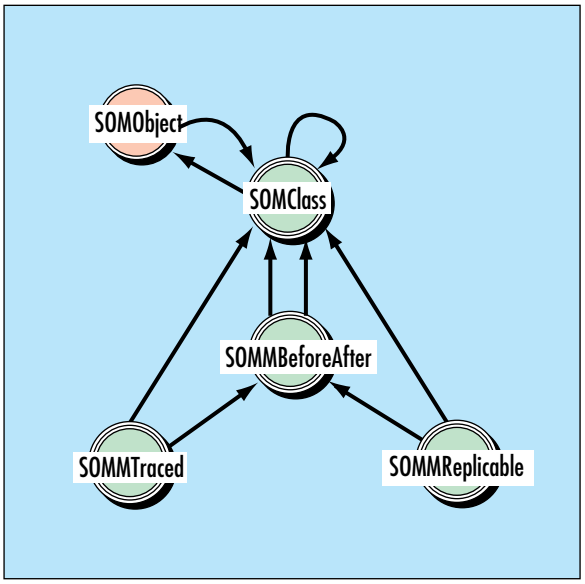


Figure 5. SOM 2.1 metaclass framework

Fierce and Barking Metaclasses

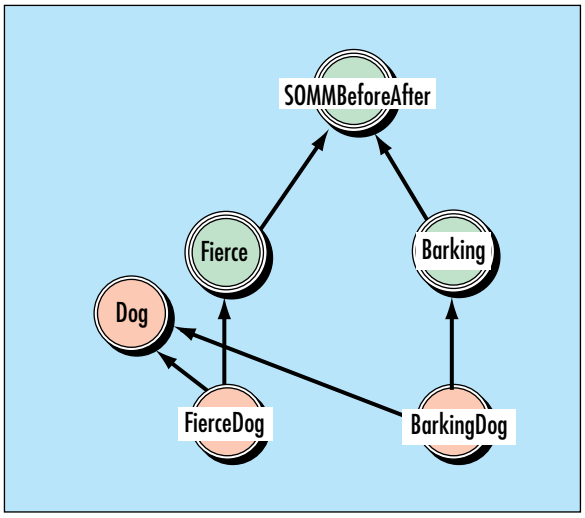


Figure 6. Fierce and Barking metaclasses

- **Technique 1**: A new metaclass (`FierceBarking`) is created with both `Fierce` and `Barking` as parents; an instance of this new metaclass (`FB-1`) should be a `FierceBarkingDog` (if `Dog` is a parent).

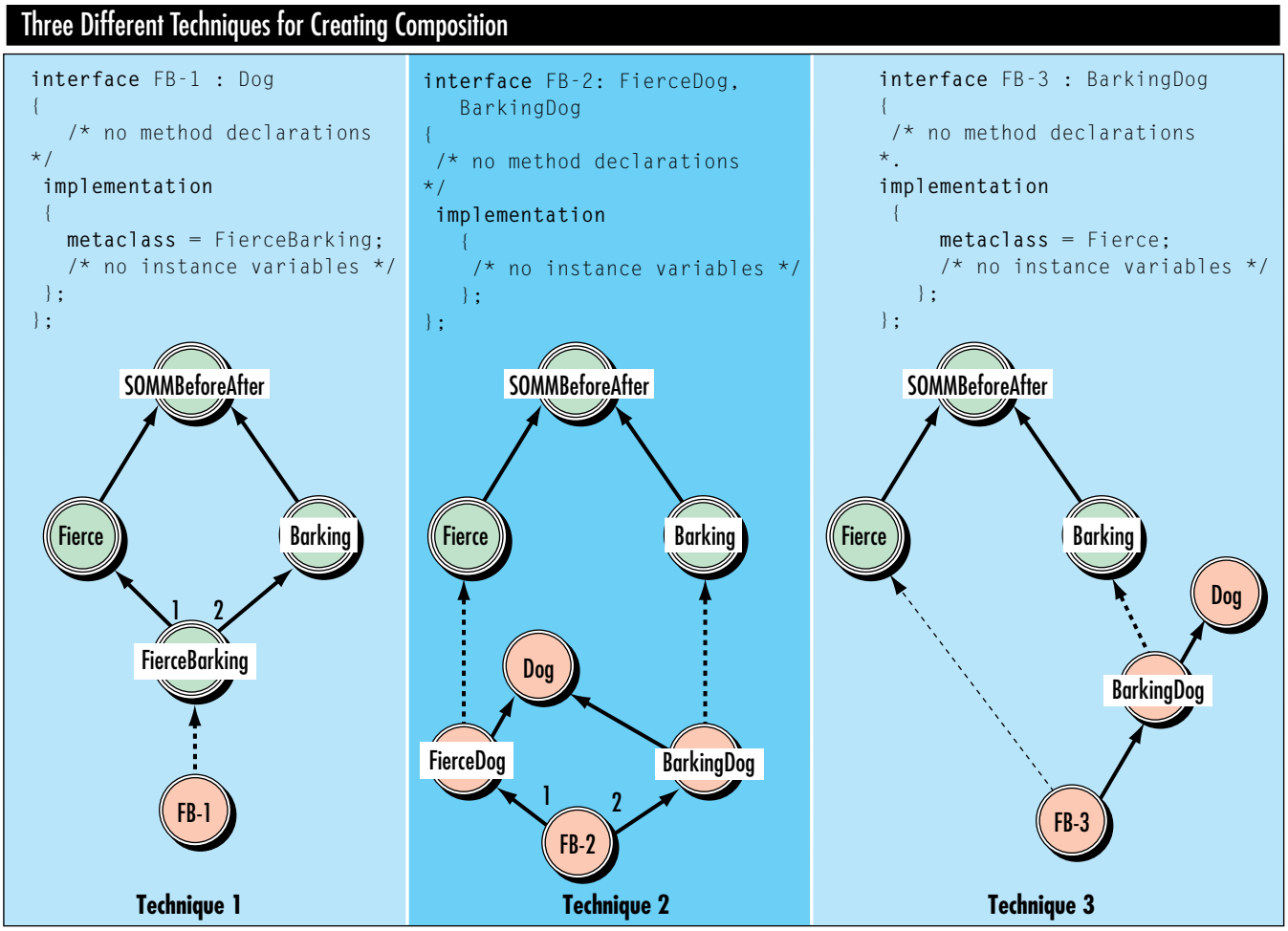


Figure 7. Three different techniques for creating composition

- **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`).
- **Technique 3**: `FB-3`, which should also be a `FierceBarkingDog`, is created by declaring that its parent is a `BarkingDog` and that its explicit (syntactically declared) metaclass (drawn with the light dashed arrow) is `Fierce`.

These three techniques should produce the same result; that is, `FB-1`, `FB-2`, and `FB-3` should be equivalent classes—behave the same and have instances that behave the same. That is because composition of metaclasses must be easily understood by programmers. 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 can be based. There is such a property and 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 programmers 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' implementations are changed (and this could include a choice of different metaclasses). SOM programmers never need to consider a newly defined class' ancestors' metaclasses. Instead, explicit metaclasses should be used only to *add in* desired behavior for a new class. Anything else that is needed is done automatically [12].

Figure 8 shows a simple single-inheritance example. `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. If the class hierarchy were to be formed as in Figure 8, an invocation of `foo` on an instance of `B` would fail because `BMeta` does not support bar. This situation is called *metaclass incompatibility*. Since SOM does not allow hierarchies with metaclass incompatibilities, it builds derived metaclasses that prevent this problem from occurring.

The actual SOM class hierarchy that results for `B` is depicted in Figure 9, where SOM has 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.

Although other articles[3,11] have called this situation the *metaclass compatibility problem*, 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 runtime with no need for prior description in IDL*.

The Composition Solution

In Figure 7, the derived metaclass constructed by SOM for `FB-3` will be `FierceBarking`, not `Fierce`

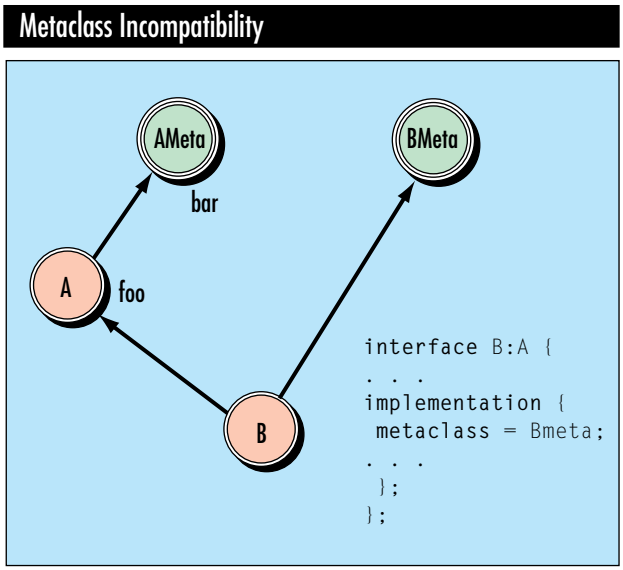


Figure 8. Example of a metaclass incompatibility

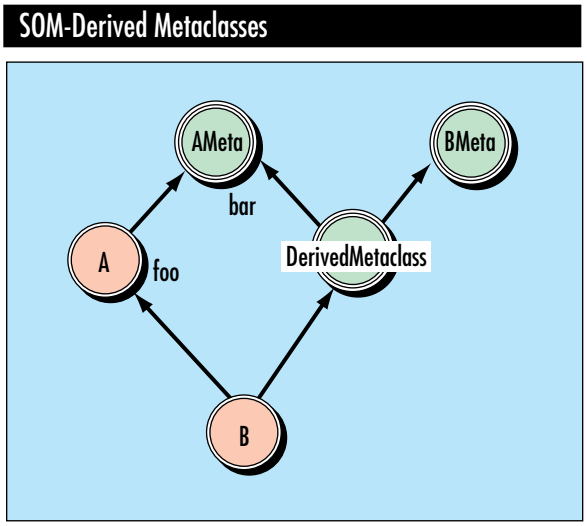


Figure 9. SOM-derived metaclasses prevent incompatibilities

as indicated in the SOM IDL. Although the metaclass of `FB-2` in Technique 2 is not explicitly declared, the derived metaclass provided by SOM for `FB-2` will be `FierceBarking`.

Figure 10 combines the diagrams in Figure 7 and shows the actual class relationships, which are established when the class objects are instantiated. 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

* 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 references 19 and 12, pages 84 and 240-241 respectively); if this condition is not true, an error is signaled.

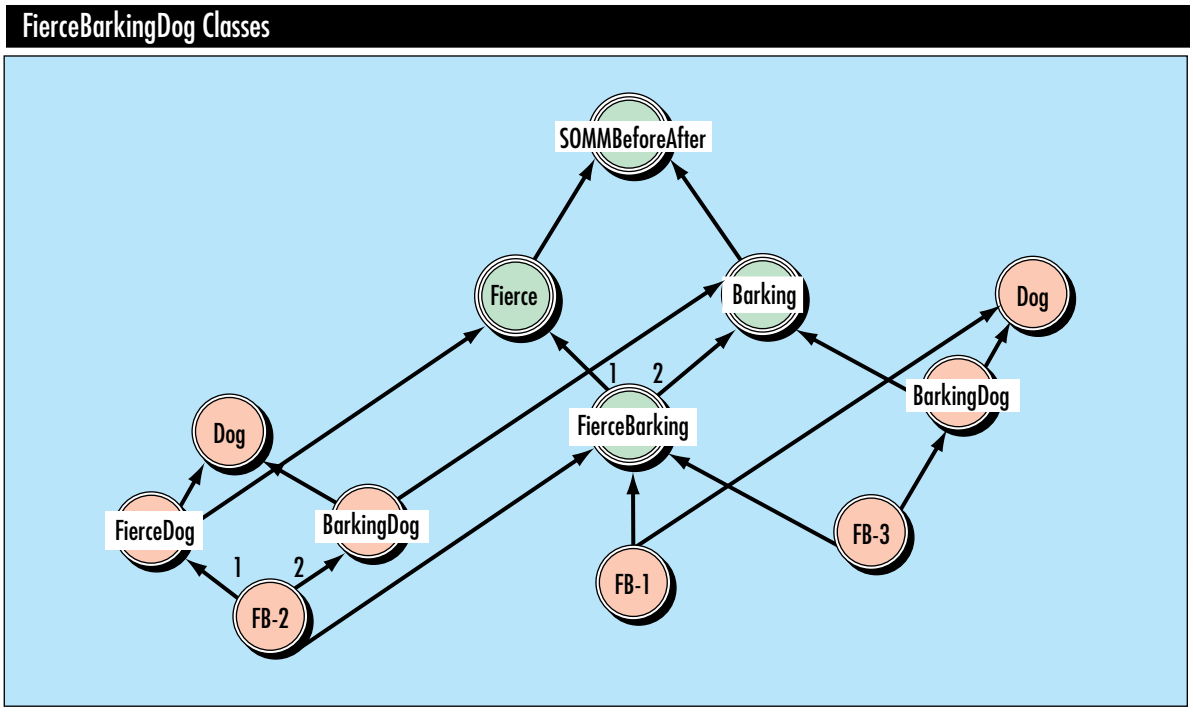


Figure 10. FierceBarkingDog classes have equivalent metaclasses

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

Figure 11. Main dispatch function

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

Figure 12. BeforeMethodDispatch implementation

the explicit metaclass in the SOM IDL. It too is `FierceBarking`.

The solution to the composition problem is apparent. 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 `FB-1`, the relationship is explicit and in `FB-2` and `FB-3` the relationship is derived. Therefore, we conclude that composition can be based on the "completed" metaclass hierarchy that results from using derived metaclasses in SOM.

By the nature of before/after, the main dispatch function now looks like Figure 11.

The `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 looks like Figure 12.

The search restriction to metaclasses that support `BeforeMethod` is based on the fact that a before/after metaclass can have parents that are not before/after metaclasses (parents that are not

descendents of SOMMBeforeAfter). If the metaclass of the client object 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 very similar, except that the parents are traversed in the reversed order. Figure 13 shows the AfterDispatchMethod implementation.

```
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 13. AfterDispatchMethod implementation

Loose Ends

The before/after metaclass facility described in this article has been available inside IBM since August 1993; for example, the traced metaclass is used in parts of the SOMObjects Toolkit (Version 2.0). The facility became generally available as part of SOMObjects Toolkit (Version 2.1) in November 1994. Before concluding, there are several issues to address, which for 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 14 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 (Before1; Before2; Before3). Of course, composition is not commutative; for example, 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.

Second 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, such as read-only, to exempting 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 the metaclass designer introduces a predicate method to be called by the BeforeMethod and the AfterMethod. If the

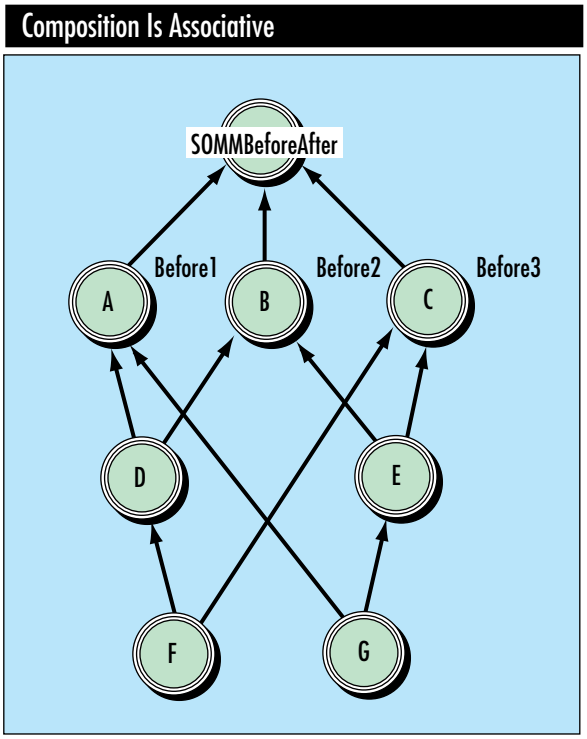


Figure 14. Composition is associative

predicate returns true, both wrapper methods return without doing anything.

Third, for the sake of simplicity, 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 14, an H is added as a subclass of both F and G, all the before methods are executed twice. Several techniques can 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 eliminates the need to communicate to the after method that the primary method was not invoked).

Conclusions

The central issue addressed by this article 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 Connor 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. Andy Martin did an outstanding job implementing the first SOM compiler. We wish to thank Liane Acker, Gregor Kiczales, and Harold Ossher for their comments on earlier drafts of this article. 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, Inc. *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). p. 419-431.
4. Campbell, R.H. and Habermann, A.N. "The Specification of Process Synchronisation by Path Expressions." *Lecture Notes in Computer Science* (16). Springer-Verlag, 1974. p. 89-102.
5. Cointe, P. "Metaclasses are First Class: the ObjVlisp Model." *OOPSLA '87 Conference Proceedings* (October 4-8, 1987). p. 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). p. 327-335.
11. Graube, N. "Metaclass Compatibility." *OOPSLA '89 Conference Proceedings* (October 1-6, 1989). p. 305-316.
12. Kiczales, G., des Rivieres, J., and Bobrow, D. G. *The Art of the Metaobject Protocol*. Cambridge, Massachusetts: The MIT Press, 1991.
13. Kiczales, G. and Paepcke, A. *Open Implementations and Metaobject Protocols*. Cambridge, Massachusetts: The MIT Press, 1994.
14. Lieberherr, K.J. and Xiao, C. "Object-Oriented Software Evolution." *IEEE Transactions on Software Engineering* 19 (April 1993). p. 313-343.
15. Maes, P. "Concepts and Experiments in Computational Reflection." *OOPSLA '87 Conference Proceedings* (October 4-8, 1987). p. 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* (S10G6309). Armonk, New York: IBM Corporation, 1991.
18. Paepcke, A. "PCLOS: A Critical Review." *OOPSLA '89 Conference Proceedings* (October 1-6, 1989). p. 221-237.

19. Paepcke, A. (ed.). *Object-Oriented Programming: The CLOS Perspective.* Cambridge, Massachusetts: The MIT Press, 1993.
20. Pascoe, G.A. "Encapsulators: A New Software Paradigm in Smalltalk-80." *OOPSLA '86 Conference Proceedings* (September 29 - October 2, 1986). p. 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.) New York: ACM Press, 1989. p. 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.* Armonk, New York: IBM Corporation, 1993.

Ira R. Forman, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Forman is a member of the Object Technology Products Group, where he specializes in object-oriented distributed systems and object composition. He has a PhD in Computer Science from the University of Maryland.

Scott Danforth, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Danforth is currently developing language-neutral object technology for binary class libraries and system-level frameworks for OOP. He has a PhD in Computer Science from the University of North Carolina at Chapel Hill.

Hari Madduri, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Madduri, the lead designer of Object Replication Framework is currently the architect/designer of distributed SOM. Mr. Madduri has a PhD in Computer Science from the University of Wisconsin in Madison.

You're Invited to be an Innovator!

Qualified commercial or in-house internal-use developers with plans to port products to, or develop and market products for IBM Power Personal Systems are invited to participate in the IBM Power Personal Developer's ToolBox program. This program offers to the growing and increasingly diverse community of developers the opportunity to purchase specific PowerPC-based development products that include IBM Power Personal hardware with selectable components, two operating systems (IBM AIX 4.1 for Clients, and Microsoft® Windows NT™), software development tools, and compilers.

Comprehensive technical support is available to assist in the development of hardware and software products.

As an enrolled participant in the program, you will be provided a unique developer's identification number. You will also be given a special toll-free telephone number to place an order with IBM to create a customized system configuration that meets your needs. You may place multiple orders for systems and receive a special developer's price on the hardware and software. Exciting new changes are being made to the Developer's ToolBox Program, including lower developer prices. For details, call 1-800-627-8363.

Porting Centers for Power Personal Systems

Solution Partnership Center
San Mateo, CA
1-800-678-4249
415-312-0240

Solution Partnership Center
Boston, MA
(to open in Summer, 1995)

Solution Developer Technical Support Center (SDTSC)
Dallas, TX
1-800-553-1623
214-280-5981

IBM Kirkland Programming Center
Kirkland, WA
1-800-803-0110
206-889-9011