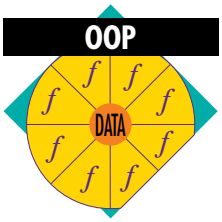
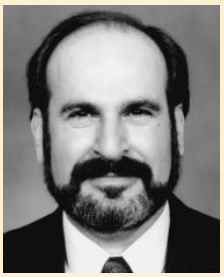


Cooperation Among Metaclasses in SOM

By Ira R. Forman and Scott H. Danforth

In SOM, metaclasses are independently programmed to impart properties to classes. As such, method table entries become resources over which metaclasses can conflict. SOM's metaclass framework implements a technique called metaclass cooperation for addressing this problem. This article describes metaclass cooperation. It also provides an example of the use of metaclass cooperation between Distributed SOM (DSOM) and Before/After Metaclasses in the SOMObjects Toolkit.


Ira R. Forman


Scott H. Danforth

In SOM, metaclasses determine the behavior of classes, which in turn determine the behavior of ordinary objects (objects that are not classes). Both SOM metaclasses and classes are defined by subclassing. As a result, metaclasses can be referred to by name in SOM, and can be explicitly used by programmers when defining new classes and metaclasses.

The SOMObjects Toolkit (Version 2.1) contains 15 metaclasses. When new metaclasses are defined by subclassing from these, the result correctly reflects the semantics implemented by the individual metaclasses that are combined. In other words, the implementations of the different abstractions provided by the Toolkit metaclasses do not interfere with each other when they are combined by subclassing. For this reason, we describe the Toolkit metaclasses as *cooperative*.

A previous paper[3] described SOM's notion of cooperation, which is summarized in the next section. This article contains an example of cooperation that combines two metaclasses from the Toolkit. The example shows that both individual Toolkit metaclasses and their combinations are useful. Cooperation between metaclasses enables this. In general, programmers of a cooperative metaclass do not need to know the other cooperative metaclasses with which it can be combined. Cooperative metaclasses are useful components for *open* object-oriented systems

The next section describes SOM's notion of cooperation (if you are not familiar with SOM, see the sidebar on page 6 before reading this article). The subsequent two sections describe useful metaclasses, followed by a section that presents a useful combination of these metaclasses based on cooperation.

Overview of Cooperation

Metaclass cooperation is the basis for creating metaclasses that can be composed. This implies that programs can be factored in new ways that lead to better modularity [5]. In addition, metaclass cooperation is essential to the proper operation of SOM, because the SOM runtime automatically derives new metaclasses as necessary to support the requirements of a new subclass. Figure 1 illustrates this based on the class declarations (in Figure 2) expressed using the SOM Interface Definition Language (IDL).

In this example, the programmer has declared class `C` as a subclass of both `A` and `B`. When the SOM runtime constructs the runtime class object `C`, it is necessary to determine the metaclass of which it will be an instance. As explained in "Reflections on Metaclass Programming in SOM[3]," the need for `C` to respond* to both the `foo` and `bar` class methods results in using a SOM-derived metaclass, denoted `DMC` in Figure 1.

When different metaclasses are automatically combined into a SOM-derived metaclass, as illustrated by Figure 1, methods or class variables

*An object responds to the methods supported by its class. The methods supported by a class are either the methods introduced by the definition of the class or the methods inherited from the class' parents.

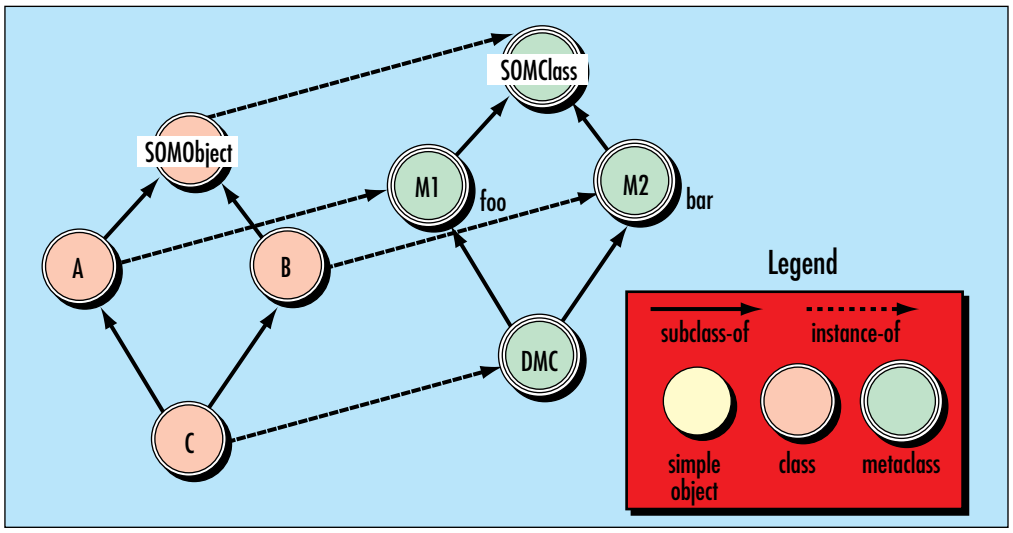


Figure 1. SOM-derived metaclasses

introduced by the different metaclasses cannot conflict with each other. This is because both the methods and instance data introduced by classes (in this case, metaclasses) are segregated into groups corresponding to their introduction of class when they are combined by using multiple inheritance. Thus, there can be no interference between `M1` and `M2` in instances of `DMC`.

In addition to introducing new class variables and methods, a metaclass programmer can also override inherited methods. Combined metaclasses can interfere with one another. For example, `somInitMIClass` is an important class method introduced by `SOMClass` (the base class for all metaclasses). This method determines the procedure pointers that are stored in the instance method table during initialization of class objects. Metaclass programmers can override `somInitMIClass` to store whatever is desired in the instance method table. But the power to explicitly load a class' instance method is a double-edged sword. How can metaclass programmers know that what they put in a class' instance method table (using a `somInitMIClass` method override) does not conflict with what another metaclass that overrides `somInitMIClass` puts there?

To ensure that a class is initialized to have the appropriate properties, SOM-derived metaclasses chain two special methods upwards to all parents to allow initialization of the class variables introduced by all ancestor metaclasses. These methods are `somInitMIClass` and `somClassReady`. This enables the composition of properties embodied by metaclasses and also causes the problem of interference over the definition of methods. If different, unrelated metaclasses are used as parents of an automatically SOM-derived metaclass, then different, generally unrelated `somInitMIClass` method procedures are executed in sequence.

```
#include <somcls.idl>
interface M1 : SOMClass {
    // declare a metaclass that
    // introduces the foo class method
     void foo();
};
interface M2 : SOMClass {
    // declare a metaclass that
    // introduces the bar class method
     void bar();
};
interface A : SOMObject {
    // declare a class that responds to
    // the foo method
     implementation { metaclass = M1; };
};
interface B : SOMObject {
    // declare a class that responds to
    // the bar method
     implementation { metaclass = M2; };
};
interface C : A, B {
    // declare a class that inherits
    // from A and B
};
```

Figure 2. Class declaration

Overview of SOM

In SOM, classes are objects whose classes are called *metaclasses*. A class differs 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, informing the class of its parents. This allows the class to build an initial instance method table. After this is done, other methods are invoked on the class to override inherited methods or to add new instance methods.

When diagramming class hierarchies, this article shows metaclasses drawn with three concentric circles, ordinary classes (classes that are not metaclasses) drawn with two concentric circles, and ordinary objects (objects that are not classes) drawn with a single circle. The initial state of an example SOM program is depicted in Figure A. There are four objects: `SOMObject` (a class), `SOMClass` (a metaclass), `Dog` (an ordinary class), and `Rover` (an ordinary object).

There are two relationships among objects that must be understood. 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—the methods that the class introduces or inherits.

Second, there is a relationship 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, which 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 it also 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 are also used in CLOS[7], ObjVlisp[2], Dylan[1], and Proteus[12].) The SOM API allows new abstractions to be created by programming metaclasses. In more general terms this has been called a *metaobject protocol*[7,8] or *computational reflection*[9]. 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, but they are part of a framework for programming metaclasses (see Reference 3 for more information) that is built with the SOM API. By providing a metaobject protocol, we

These unrelated `somInitMIClass` method procedures might then interfere with each other.

To solve this problem, SOM metaclass cooperation provides a framework for metaclass programmers to create method procedures that "cooperate" in executing inherited methods (instead of simply overriding these methods). This allows a cooperative metaclass to implement its desired semantics without interfering with the semantics implemented by other cooperative metaclasses with which it might be combined.

The following sections describe two important class frameworks enabled by cooperative metaclasses in the SOMObjects Toolkit (Version 2.1).

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[7, 11] where the granularity of application is the individual method. In the class-based object model SOM, the more natural granularity for before/after

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[10]) standard of the Object Management Group (OMG). SOM IDL is a CORBA-compliant version of IDL that allows SOM class descriptions to be supplied in addition to object interface definitions. (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 example, to C and C++ programmers this means that emitters produce header files for both the users and the implementer of the class.

The following example shows the basic structure of an IDL definition for an object interface named `Dog`.

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

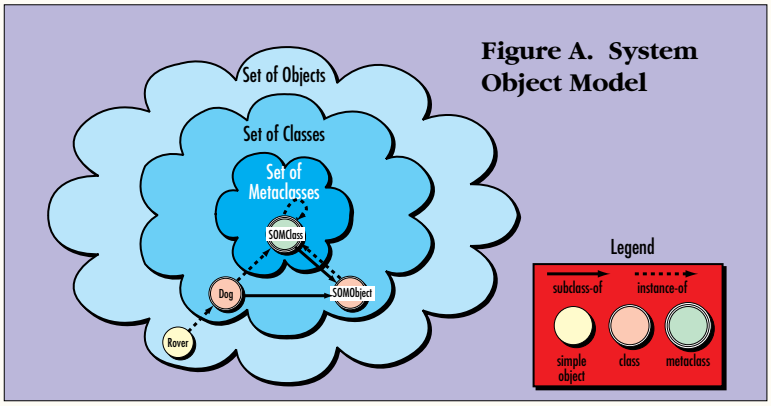


Figure A. System Object Model

At the same time, it is a SOM IDL description of a class `Dog` that supports this interface. The `#ifdef` and `#endif` —part of the CORBA IDL language—are used to hide the SOM class implementation section from non-SOM IDL compilers.

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 supports the interface being defined. This association is not necessarily direct. For reasons described in this article, the actual class of the class described by any given SOM IDL is, in general, a subclass of the indicated metaclass.

methods is the class, because there are many applications that fit this granularity[5]. Therefore, the `SOMMBeforeAfter` metaclass introduces two methods—`sommBeforeMethod` and `sommAfterMethod`—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, divide `SOMMBeforeAfter` into subclasses and override the `sommBeforeMethod` and the `sommAfterMethod` with the desired behavior.

For example, consider the classes in Figure 3. The `SOMMTraced` metaclass overrides `sommBeforeMethod` with code that prints the method name and the calling parameters, and overrides `sommAfterMethod` with code that prints a message indicating that the method has finished execution and also the returned value. As a result, all methods supported by the class `TracedDog` (an instance of `SOMMTraced` and a subclass of `Dog`) have this before/after behavior. That is, for all methods invoked on the object `Lassie`, trace information is printed before and after each method invocation because `Lassie` is an instance of `TracedDog`. The IDL for the `TracedDog` class, shown at the left of the figure, is the only source

A Traced Object — Lassie

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

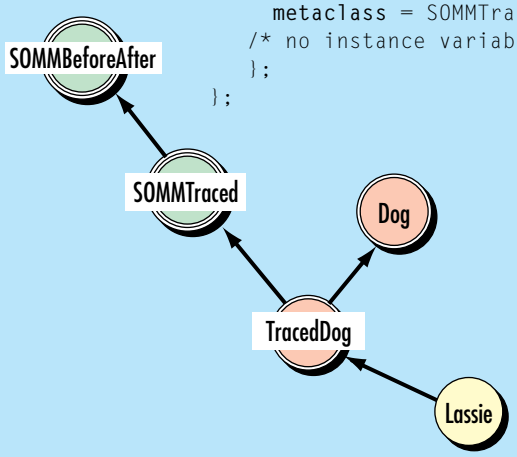


Figure 3. Example of a traced object—Lassie

```
somDispatch ( self, primaryMethod, ... )
    1. BeforeMethod( class(self), self,
                     primaryMethod, ... )
    2. retval := primaryMethod( self, ... )
    3. AfterMethod( class(self), self,
                    primaryMethod, retval, ... )
    4. return retval
```

Figure 4. Before/After dispatcher

code that needs to be written to implement `TracedDog`. The compiler of the SOMObjects Toolkit can generate from IDL all the necessary code to implement `TracedDog` (in either C or C++).

The `SOMMBeforeAfter` method overrides the method `somDispatch`, which is introduced by `SOMObject`. The new dispatcher looks like Figure 4.

In this case, `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 `sommBeforeMethod`; that is, the class `TracedDog` responds to `sommBeforeMethod`. This is why, in the pseudo-code in Figure 4, `class(self)` is the target object for the `sommBeforeMethod` and `sommAfterMethod` method invocations.

For efficiency in the actual implementation, methods in SOM are usually invoked directly, and the `somDispatch` method is not generally called. To route all method invocations through `somDispatch`, the `SOMMBeforeAfter` metaclass places redispatch stubs in the method table to call `somDispatch`.*

A redispatch stub is a small piece of code that routes a method invocation through `somDispatch`. The `SOMMBeforeAfter` metaclass also arranges for the contents of the original method table to 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. More information on the SOM API can be found in Reference 6. In other words, the `SOMMBeforeAfter` metaclass needs some control over the method dispatch and overrides all method table entries with a piece of code (the redispatch stub) that ensures that control passes through `somDispatch`.

Distributed SOM

A second important framework that requires control of the method dispatch definition is DSOM, which implements remote method invocation as depicted in Figure 5. The specification is simple: given an object reference, you must be able to invoke methods on the remote object (identified by the object reference). In the usual convention, the invoking process is called the *Client*, while the process that holds the receiving object is called the *Server*.

*This is done with a method called `somOverrideMtab`, which is part of the SOM Application Programming Interface (API) that has been deprecated. *Deprecation* of a method means that the documentation of the method has been withdrawn from the Toolkit, and use of the method is being discouraged. Because of SOM's promise of release-to-release binary compatibility, a deprecated method's implementation is still available. Most of the deprecated methods (such as `somOverrideMtab`) are methods of `SOMClass`. The intent of these methods was to allow Toolkit users to build classes dynamically and to do metaclass programming. Classes can be dynamically built with `somBuildClass`. Metaclass programming is done by subclassing from the Toolkit's metaclass framework.

DSOM implements this according to the Object Management Group's Common Object Request Broker Architecture (CORBA) specification[10], which states that this is done through a level called the Object Request Broker (ORB), shown in Figure 6. The ORB must provide the following services: marshaling the actual parameters of the invocation, forwarding the invocation to the appropriate process, invoking the method at the server process, and finally returning any computed values.

The DSOM implementation of an ORB is based on implementing an object reference as a proxy object for the remotely located object. The way to invoke a method on a remote object is by invoking the method on the proxy, as shown in Figure 7. (The relationship between the CORBA specification and DSOM is more complex than we have shown. Object references can exist as an object or in a linear form; there is automatic conversion in the DSOM implementation. The *SOM User's Guide* section 6.8 contains a more thorough description of this relationship.)

Proxies are constructed as shown in Figure 8. As an object, a proxy has a class `ProxyFor_A` (in SOM 2.1, the name is actually `A_ _Proxy`, but we use `ProxyFor_A` for readability) that is created by joining the class of the remote object (`A`) with the DSOM framework class `SOMDClientProxy`. The metaclass of `SOMDClientProxy` is `SOMDMetaProxy`; because of inheritance of metaclass constraints in SOM, `SOMDMetaProxy` is also the metaclass of `ProxyFor_A`.

The key to implementing the DSOM proxy is redefining `somDispatch` in the class `SOMDClientProxy`. This redefinition interfaces with the ORB to forward a method invocation to the remote target object. Statically declared methods are invoked directly through the method table, but the method invocation is routed through `somDispatch` when its method table entry is replaced with a redispatch stub (a small piece of code that redirects the direct invocation through `somDispatch`). A dynamically constructed DSOM proxy class has all method table entries for methods inherited from the target class (`A` in Figure 8) replaced with redispatch stubs. In addition, some (but not all) methods that are inherited from `SOMDClientProxy` are also forwarded to the target object (by an explicit invocation of `somDispatch` within the method's implementation).

The `SOMDClientProxy` ensures that the `somDispatch` of all proxies properly interfaces with the client side of an ORB, marshals the

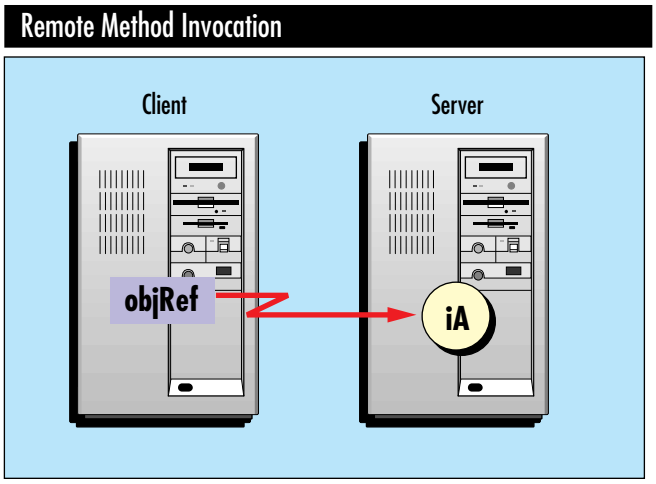


Figure 5. Remote method invocation

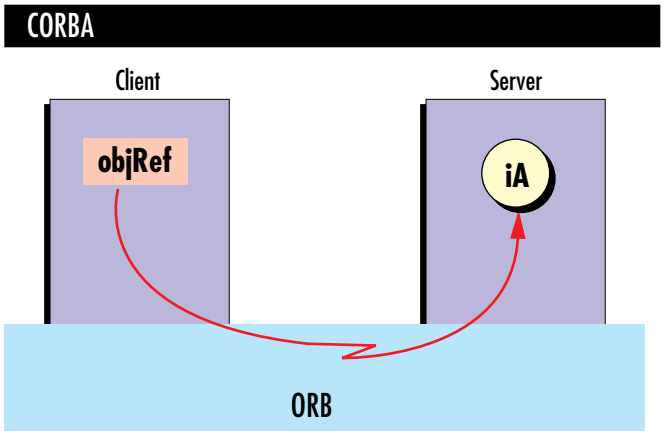


Figure 6. The Common Object Request Broker Architecture

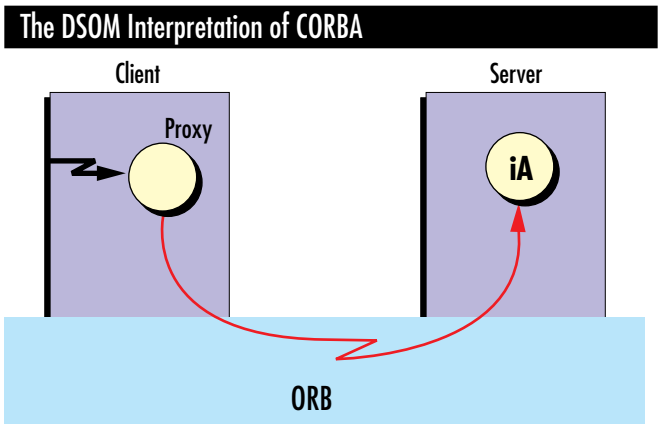


Figure 7. DSOM interpretation of CORBA

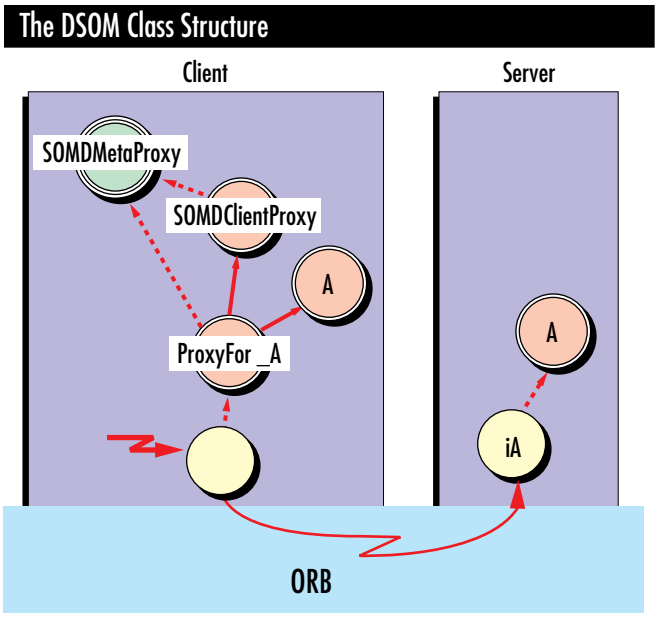


Figure 8. The DSOM class structure

```
somDispatch ( somSelf, primaryMethod, ... )
     1. Marshal the actual parameters.
     2. By using ORB services, send the
        invocation to the server.
     3. Bring back the values returned by the
        remote invocation.
     4. If the returned value is a CORBA
        object reference, convert it to a
        proxy.
```

Figure 9. SOMDClientProxy Dispatcher

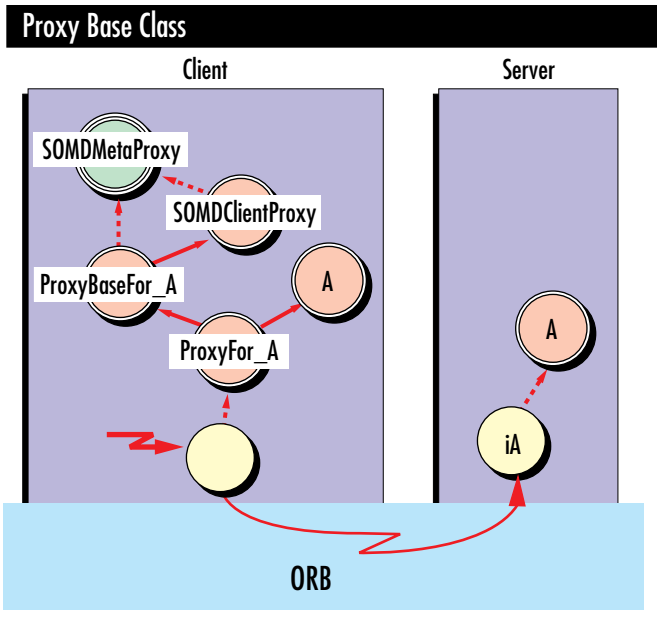


Figure 10. User-created proxy base class

actual parameters of the invocation, forwards the invocation to the appropriate process, and returns computed answers (see Figure 9).

`SOMDClientProxy` is the default base class for dynamically created proxies. In addition, `SOMDClientProxy` can be subclassed to create specialized base classes for proxies, as in the following example:

```
interface ProxyBaseFor_A :
SOMDClientProxy {
     ...
};
```

To use this proxy base class, it must be named in the IDL of the target class with the `baseproxyclass` SOM IDL modifier. Figure 10 depicts a proxy class with a specialized base proxy class. The advent of the Before/After Metaclasses in SOMObjects Toolkit 2.1 has generated good reasons to create specialized base proxies, as shown in the next section.

The Need for Cooperation

Each of the two frameworks—Before/After Metaclasses and DSOM—has a metaclass that requires the ability to define `somDispatch` to impart its property to ordinary objects. Without the ability to cooperate, composition of these two frameworks is not possible. The technology for cooperation[3] is based on reinterpreting the content of a method table entry as the sequence of methods, all of which might be invoked. The rationale for this is simple: from the metaclass programming viewpoint, metaclasses compete over resources such as a method table entry. By reinterpreting the method table entries as a sequence, you change the granularity of the resource conflict. Metaclasses can request the first, last, or arbitrary position in the sequence corresponding to an individual method table entry.

There are two areas of cooperation in which to compose DSOM proxies with Before/After Metaclasses. First, each framework must cooperate on the definition of the `somDispatch` method. In this case, the Before/After Metaclass requests the first method on the cooperation chain. `SOMDClientProxy` provides the last method on the cooperation chain.

Secondly, each framework must agree as to where to place redispatch stubs in the method table. The Before/After Metaclasses require that redispatch stubs be placed in all entries, while the DSOM proxy places redispatch stubs in most

(but not all) entries. In this case, DSOM's requirements take precedence. The proxy metaclass, SOMDMetaProxy, arranges this by requesting the first position in the cooperation chain for the method somOverrideMtab and ending the execution of the method without overriding all of the method table entries.

The result of this cooperation is that when Before/After Metaclasses compose with DSOM proxies, the before/after behavior occurs only for the method invocations that are forwarded from the proxy to the target object. This might not be what is desired by application programmers in all cases. That is, there are cases where before/after behavior is needed on a method, regardless of whether it is forwarded. However, Before/After Metaclasses cooperate on somDispatch, and DSOM uses somDispatch only to forward methods from the proxy to the target. A future version of the DSOM proxy will allow either to be easily programmed.

There is an additional question to address: Even if composition is possible, will it be used? That is, are there real examples that require the kind of cooperation presented here? This question is affirmatively answered by the following familiar example: transaction processing with logging—keeping an audit trail or journal for transactions.

A DSOM remote method invocation can capture the notion of a transaction (assuming the DSOM server is acting in single-thread mode). Transactions should be logged at the requesting sites.[4] Each proxy could be required to be individually crafted to log its activity. This necessitates much duplication of effort, especially considering the fact that logging is very much like tracing. Assume that we have a Logging metaclass that is a subclass of SOMMBeforeAfter and is much like the SOMMTraced metaclass.

Figure 11 shows how to arrive at the desired structure. First, create a general (and reusable) LoggingProxy with the following IDL:

```
interface LoggingProxy : SOMDClientProxy
{
     implementation {
        metaclass = Logging;
     };
};
```

This class is used as the base class for deriving a LoggingProxy class for any class. For class A in Figure 11, the IDL for the target class looks like the following.

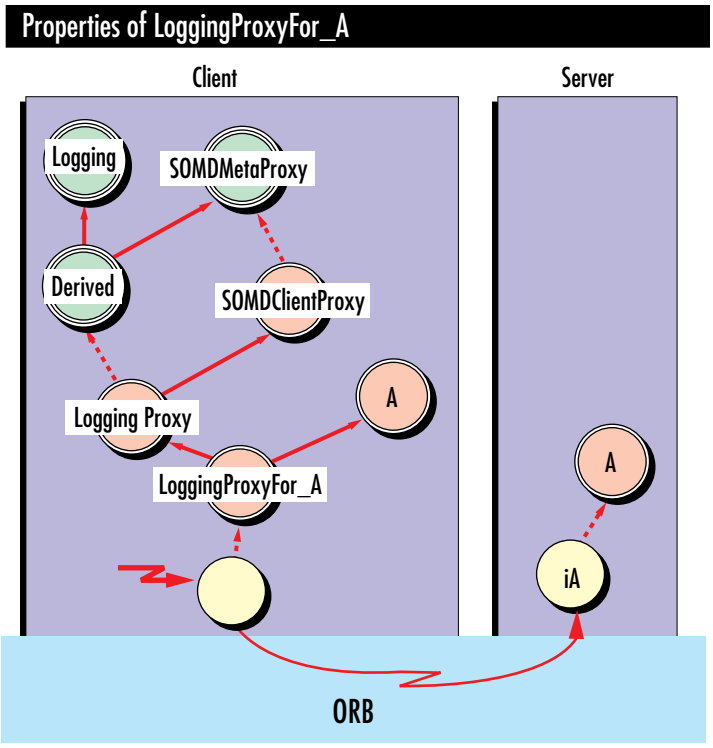


Figure 11. LoggingProxyFor_A has properties imparted by both Logging and SOMDMetaProxy

```
interface A {
   implementation {
      baseproxyclass = LoggingProxy
      ...
   };
};
```

The advantage of this arrangement (over the usual object-oriented abstractions) is better separation of concerns. There are three basic concerns that have been separated into independently programmed modules:

- Functionality of the transaction (located in the method being invoked)
- Remote invocation (inherited from SOMDClientProxy)
- Audit trail (located in the Logging metaclass)

Each of these concerns can be independently programmed only because Logging cooperates on somDispatch rather than overriding it. Without the ability to cooperate, DSOM would have to be modified to provide logging. Such a provision (probably implemented by subclassing SOMDClientProxy and overriding somDispatch) would not be reusable.

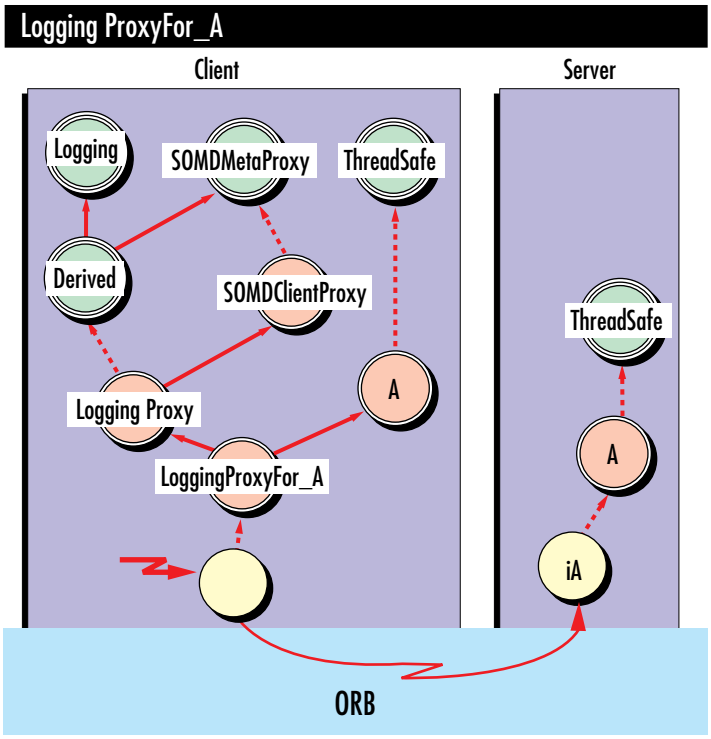


Figure 12. LoggingProxyFor_A is not thread safe

There is another aspect of cooperation. By design, DSOM does not provide concurrency control when the server operates in multithreaded mode. The reason is that there are many schemes for concurrency control[4], and it is not feasible for DSOM to support them all. Instead, DSOM customers can use the Before/After Metaclasses to provide concurrency control as shown in Figure 12. The metaclass `ThreadSafe` on the server can provide concurrency control, because `ThreadSafe` can acquire locks before a method is invoked and release locks afterward. Note that although `ThreadSafe` also appears on the client side (as a metaclass of `A`), `ThreadSafe` has no effect on the proxies to `A` instances. This is because DSOM does not allow inheritance of the metaclass constraint from the class being proxied.

Conclusion

A metaclass can embody a property (such as thread safety) independently of the objects that possess the property. The metaclass can impart such properties to the class at runtime without modification of the class, which leads to improved modularity. But the key to reuse is the ability to compose the metaclasses so that the properties are naturally composed. The standard object-oriented notion of method override leads to metaclasses that compete over the definition of methods and, thus, interfere with each other's operation (that of imparting properties). SOM uses cooperative metaclasses that add to the function of a method and avoid interference. This article shows that such cooperation is essential to a highly desirable solution to a very practical problem (that is, independently programming transaction logging and remote method invocation).*

Acknowledgments

The authors wish to thank Liane Acker for her comments on this article.

Presented at the Eighth IBM International Conference on Object Technology in San Francisco, June 13-16, 1995

References

1. Apple Computer. *Dylan: An Object-Oriented Dynamic Language* (1992).
2. Cointe, P. "Metaclasses Are First Class: the ObjVlisp Model." *OOPSLA '87 Conference Proceedings* (October 4-8, 1987) p. 156-165.
3. Danforth, S.H. and Forman, I.R. "Reflections on Metaclass Programming in SOM." *Proceedings of OOPSLA '94*. Portland, Oregon. (October 23-26, 1994).
4. Date, C.J. *Database: A Primer*. Addison-Wesley (1983).
5. Forman, I.R., Danforth, S.H., and Madduri, H.H. "Composition of Before/After Metaclasses in SOM." *Proceedings of OOPSLA '94*, Portland, Oregon. (October 23-26, 1994).
6. *IBM SOMobjects Developer Toolkit User's Guide*. Version 2.1 (October, 1994).
7. Kiczales, G., des Rivieres, J., and Bobrow, D. G. *The Art of the Metaobject Protocol*. Cambridge, Massachusetts: The MIT Press. 1991.
8. Kiczales, G. and Paepcke, A. *Open Implementations and Metaobject Protocols*. Cambridge, Massachusetts: The MIT Press. 1994.

*Currently, in the SOMObjects Toolkit (Version 2.1), the interface to metaclass cooperation (as described in Reference 1) is not public. It is experimental and available on request. In the future, we (the authors) expect it to be available in the form of an operation on a class, much the same way as the Before/After Metaclass is an operation on a class.

9. Maes, P. "Concepts and Experiments in Computational Reflection." *OOPSLA '87 Conference Proceedings* (October 4-8, 1987) p. 147-155.
10. Object Management Group. *The Common Object Request Broker: Architecture and Specification*. OMG Document Number 91.12.1. 1991.
11. Paepcke, A. (ed.) *Object-Oriented Programming: The CLOS Perspective*. Cambridge, Massachusetts: The MIT Press. 1993.
12. 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.

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 H. Danforth, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Danforth is a member of the Object Technology Products group. He 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.