Reflections on Metaclass Programming in SOM

Scott Danforth
Ira R. Forman
IBM Object Technology Products
11400 Burnet Road
Austin, TX 78758

Abstract. This paper reports on the evolution of metaclass programming in SOM (the IBM System Object Model). Initially, SOM's use of explicit metaclasses introduced metaclass incompatibilities. This was cured by having SOM dynamically derive an appropriate metaclass by interpreting the "metaclass declaration" as a constraint. In effect, inheritance is given a new dimension, because the constraint is also inherited. The derived metaclass is the least solution to all these constraints. Subsequently, this cure led to the possibility of metaclasses conflicting over the need to assign meaning to a method. The cure for this problem is a framework that facilitates the programming of metaclasses that cooperate on the assignment of meaning to methods.

Introduction

The term "procedural reflection" was introduced by Smith [18,19] to describe a general theory and mechanism allowing computational systems to reason about their own operations and structures. Reflective capabilities have now been studied within a variety of different computational systems (CommonLoops [1]. 3KRS [16], Scheme [9,21], LISP [6], ObjVLisp [5,11], ABCL/R [22], Rosette [20], and CLOS [3,15]). This paper discusses some of the ways reflective capabilities are useful and important to SOM metaclass programmers. In a more general sense, the word reflection seems appropriate for describing the overall purpose of this paper, a thoughtful overview of the origin and historical development of mechanisms supporting metaclass programming in SOM.

First Principles

The benefits of using OOP for system-level support of component software development in systems like Smalltalk-80 and NextStep (based on Objective-C) are by now well-known and accepted. SOM's objective is to allow binary class libraries to provide these benefits independent of programming languages and compilers.

To provide language-neutrality, SOM defines a runtime API that is based on a few external procedures and simple data structures. This API is used by programmers (or by "language bindings" or by "DirectToSOM" compilers) to create and use SOM objects according to a traditional object-oriented model of computation. Within this context, SOM has two fundamental guiding principles:

1. If changes to the implementation of a class don't require changes to client source code (i.e., code that subclasses or uses instances of the class), it should be possible to replace the class's implementation (in a binary library) without requiring recompilation of client code.
2. The SOM API should be expressed as an object-oriented system composed of SOM objects available to the programmer.

The main reason for the first principle is to support component software. Once application binaries are delivered to users, any need for recompilation of source is best avoided -- even when aspects of the supporting system are reorganized or reimplemented. The first principle thus maximizes the flexibility available for supporting system evolution when OOP is used to implement and publish system interfaces. This issue cannot be taken lightly. Evolution of an OOP system often suggests refactorings of its class

Presented at the OOPSLA '94 Conference, October 23-27, 1994

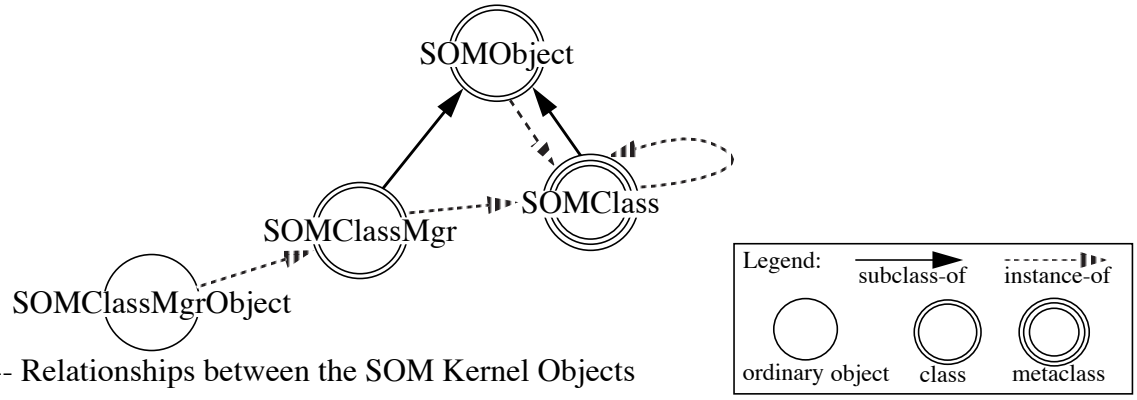

Figure 1 -- Relationships between the SOM Kernel Objects

hierarchy, even though this may not require source level changes in client code. From the perspective of metaclass programming, however, this principle represents a serious challenge [7]. The section on SOM-derived metaclasses, below, explains the unique mechanism provided by SOM for answering this challenge.

The second principle results in implementing classes as objects, which creates the very possibility of metaclass programming in SOM (if classes are objects, they must be instances of other classes called metaclasses). The second principle also relates directly to a fundamental requirement for reflection explained by [19]: reflection requires a model within which to express the operations, structures, and overall protocols that are used to organize and control computation. The SOM abstract machine expresses such a model for OOP, and a bootstrap procedure creates a concrete embodiment of this model as the initial SOM runtime environment.

In particular, the second principle results in a model of the operations, objects, and protocols used for implementing object-oriented computation -- i.e., for subclassing, inheriting methods, adding new methods, overriding inherited methods, and dispatching methods. Thus, in SOM, the fundamental requirement of reflection is met as a result of first principles -- the operations and objects used to implement OOP computation are reflected in concrete SOM objects immediately available to executing code.

The SOM capabilities based on these principles enable its users to *program at a higher level*. As this paper explains, the ability to program at a higher level unfolded in several stages. First was SOM's adoption of metaclass programming as its metaobject protocol. Second was the invention of the derived metaclass. Third was a framework that allows metaclasses to cooperate on the definition of method implementations. The result is that rather than merely programming classes anew or as extensions of previously written classes, SOM users will be able to factor functionality in new ways, which in turn leads to achieving programming goals by composing classes. Thus, SOM users can *program at a higher level* (as can any programmer with an available metaobject protocol).

The SOM Abstract Machine

The SOM abstract machine has changed in many ways since the introduction of SOM in OS/2 2.0, where SOM was used to implement the Workplace Shell (a class framework representing aspects of the OS/2 2.0 system). But, the top-level structure of the SOM machine remains the same. When the SOM runtime environment is created, the objects illustrated in Figure 1 are created and made available to the SOM user

Figure 1 contains the two primitive classes that are the basis for all subsequent classes:

- **SOMObject** - the root ancestor for all SOM classes
- **SOMClass** - the root ancestor of all SOM meta-classes.

All SOM objects are an instance of a SOM class, and all SOM classes are ultimately derived by subclassing from **SOMObject**. Thus, all SOM objects can execute the methods introduced by **SOMObject**. In a similar manner, all SOM classes are an instance of a SOM

metaclass, and all SOM metaclasses are ultimately derived from **SOMClass**. Thus, all classes can execute the methods introduced by **SOMClass**. Both **SOMObject** and **SOMClass** are instances of **SOMClass**.

In addition to **SOMObject** and **SOMClass**, the SOM kernel includes the class **SOMClassMgr** which implements a runtime class registry. This allows use of dynamically loaded libraries (DLLs) to define SOM classes. As illustrated in Figure 1, the SOM runtime environment initially contains the above three classes and a class manager (i.e., an object that is an instance of **SOMClassMgr**).

In SOM the class of an object defines its implementation. Therefore, in SOM, reflective code is method code that follows the instance-of link from "self" (the object on which the method is invoked), because SOM provides the method `somGetClass` for this purpose.

Operational use of the SOM API

Although Figure 1 shows the primitive objects used to implement SOM, an understanding of SOM includes knowing how these objects are used. For example, to create a new class in SOM, the following steps are followed:

- Choose a metaclass object (either **SOMClass** or some class derived from **SOMClass**) and create a new instance of the metaclass -- this will be the new class. **SOMClass** introduces a number of different methods for creating new objects, and, at this stage the new class we are creating is simply a new object that happens to be an instance of a metaclass.
- Inform the new class object of its parents by using the `somInitMIClass` method. When a class executes this method, it creates an initial *instance method table* by inheriting the contents of its parents' instance method tables. The instance method table of a class determines the behavior of its instances.
- Add new methods to the class and override inherited methods by using the methods provided by **SOMClass** for this purpose. These methods, when applied to the new class object, modify the class's instance method table that was created in the previous step.
- Use `somClassReady` to inform the class that its construction is complete. This method registers the new class with the **SOMClassMgrObject**, so that client code can access the class for further subclassing or instance creation.

Language bindings or DirectToSOM compilers automatically provide code that does the above steps according to static information provided by a class designer, thus removing these considerations from the direct concern of the programmer. But it is important to understand that all classes in SOM are runtime objects created by invoking methods on other objects as explained above. In general, SOM programmers can create classes according to dynamically computed requirements. This is of crucial importance, because supporting static class definition in SOM has required the dynamic derivation of metaclasses. To understand this surprising fact, it helps to review how metaclasses appear and are used in OIDL, the object interface definition language of SOM 1.0

Metaclasses in SOM 1.0

The original version of SOM was supported and implemented using C language bindings generated (by the SOM compiler) from class declarations expressed using OIDL (Object Interface Definition Language). OIDL provided two different ways of statically indicating the information necessary to create new metaclasses. Accordingly, metaclasses were either *explicit* or *implicit* metaclasses. An explicit metaclass is declared by explicit subclassing from some other metaclass (perhaps from **SOMClass**). Here is an example.

```
class: Counted;
parent: SOMClass;
data:
   long instanceCount;
methods:
   long getInstanceCount();
overrides:
   somInit;
   somInitMIClass;
   somNew;
```

Instances of the above metaclass include (in addition to inherited class variables) a class variable to record the number of instances of the class and (in addition to the inherited class methods) a class method to allow

users to access the current instance count. The new metaclass overrides the method `somInit` (introduced by **SOMObject**) and overrides the methods `somInitMIClass` and `somNew` (both introduced by **SOMClass**). The implementation of **Counted** must therefore provide four method procedures: one to execute the new method, `getInstanceCount`, and three others for the overridden methods.

In OIDL, a class designer indicates the use of an explicit metaclass when declaring a class, as illustrated here.

```
class: ClassExample;
parent: SOMObject;
metaclass: Counted;
...
```

On the other hand, an *implicit* metaclass is declared implicitly when declaring the class intended to be its instance, as illustrated by the following OIDL:

```
class: ClassImplicitExample;
parent: SOMObject;
data:
   long instanceCount, class;
methods:
   long getInstanceCOunt(), class;
overrides:
   somInit, class;
   somInitMIClass, class;
   somNew, class;
```

The above OIDL explicitly declares a class named **ClassImplicitExample** and also implicitly declares a metaclass that has the same fundamental characteristics as **Counted**, declared above. This is done by using the `class` modifier to indicate that either data or methods are to be attributes of the class being declared (as opposed to being attributes of instances of the class). But there is an important difference -- the implicit metaclass has no name. The functionality it provides, while generally useful, is available only on **ClassImplicitExample** and its descendants. In contrast, the functionality packaged by **Counted** is available, by name, to any class.

Implicit metaclasses can know and make use of the methods inherited and introduced by their instances. A class (for example **Y** in Figure 2) might implement an instance method by invoking a class method on itself (reached by first using `somGetClass` on the target instance), passing the target instance as an argument to the class method, and then use an implicit metaclass **I** to implement the class method using instance methods introduced by **Y**. In contrast, an explicit metaclass is normally designed to be of use to any class, and therefore doesn't make any assumptions concerning the methods inherited or introduced by its instances (other than that these inherited methods must include those introduced by **SOMObject**). It would probably seem more reasonable to package functionality for providing instance counts using an explicit metaclass such as **Counted**. Note that Smalltalk was the first language to have class as objects [14], but it provides only implicit metaclasses.

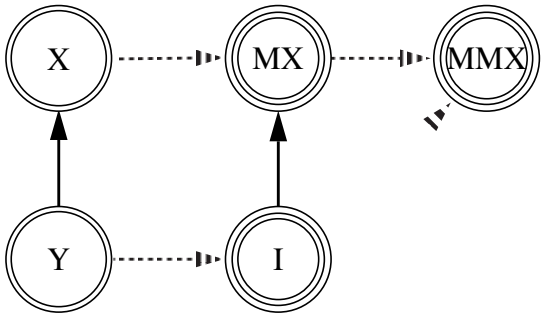

Figure 2 -- Semantics of Implicit Metaclasses

Areas of Concern for SOM 1.0

How, specifically, are implicit metaclasses derived? What are their parents, and of which classes are they instances? Figure 2 illustrates the general situation for SOM 1.0 implicit metaclasses. The class **Y** has been derived by subclassing **X**, and the class **I** is the implicit metaclass declared by **Y**'s designer. **I** is derived by subclassing from **MX** (**X**'s class) and is created as an instance of **MMX** (**MX**'s class).

This semantics supports SOM classes as polymorphic objects useful through the interfaces of all ancestor classes. For example, recall the example scenario suggested above for reflective programming with implicit metaclasses. It is imperative that class methods introduced by **MX** be available on **Y**, because the implementation of **Y**'s methods is inherited from **X**, and these methods may access the class of the instance target and invoke methods introduced by **MX** on this class. The above derivation of **I** guarantees this

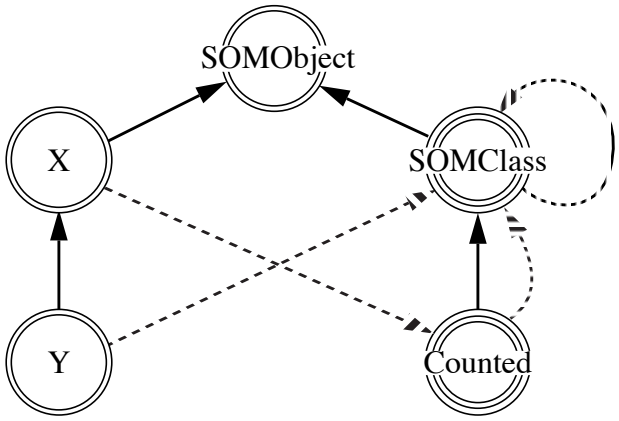

Figure 3 -- The Problem with Explicit Metaclasses in SOM 1.0

result in general, for all implicit metaclasses.

Unfortunately, SOM 1.0 does not provide a similarly pleasing result in the case of explicit metaclasses. For example, using the explicit metaclass **Counted** declared earlier, it is possible to construct the following example, in which the class of **Y** does not support an appropriate interface:

```
class: X;
parent: SOMObject
metaclass: Counted;

class Y;
parent: X;
metaclass: SOMClass;
```

Figure 3 illustrates the semantics of these declarations in SOM 1.0, and the following code, expressed using the C bindings of SOM 1.0, illustrates the problem with the class **Y** in Figure 3. Execution of this code creates a method resolution error because the class of **Y**, **SOMClass**, doesn't support the method `numInstances` on **Y**.

```
#include <Y.h>

void printCount(X *x)
{/* This code is typesafe on Xs */
 printf("%d\n",
        _numInstances(_somGetClass(x)));
}

main()
{Y *yInstance = YNew();
 /*  But this call with a subclass
     instance fails */
 _printCount(yInstance);
}
```

This kind of situation was identified by Nicolas Graube [12], who characterized the problem in terms of *metaclass compatibility*. Put simply, **SOMClass** is not compatible with the requirements placed on **Y**'s class to support the **Counted** interface. However, SOM 2.0 doesn't construct class hierarchies with metaclass incompatibilities. Instead, SOM 2.0 automatically builds new metaclasses that are compatible with their requirements, dynamically subclassing from existing metaclasses whenever this is necessary.

Metaclasses in SOM 2.0

While supporting previously existing binaries, SOM 2.0 added multiple inheritance and complete support for OMG's CORBA (Common Object Request Broker Architecture) [17]. Although OIDL is still supported by the SOM compiler, the preferred language used to declare SOM classes is now CORBA IDL. The following is an IDL declaration for the metaclass **Counted**:

```
interface Counted : SOMClass {
  readonly attribute long instanceCount;
  #ifdef __SOMIDL__
  implementation {
    somInit: override;
    somInitMIClass: override;
    somNew: override;
  };
  #endif
};
```

CORBA IDL was designed to support interfaces to objects, not their implementations. The SOM IDL implementation section (guarded with an `#ifdef`) provides additional information used by the SOM compiler to create language bindings that assist in implementing SOM classes whose objects support the declared interface. For brevity, the `#ifdef` is omitted in following illustrations. Here is the IDL for **X** and **Y** of the previous example:

```
interface X : SOMObject {
 implementation { metaclass = Counted;};
};

interface Y : X {
 implementation { metaclass= SOMClass;};
};
```

IDL doesn't provide implicit metaclasses, but explicit

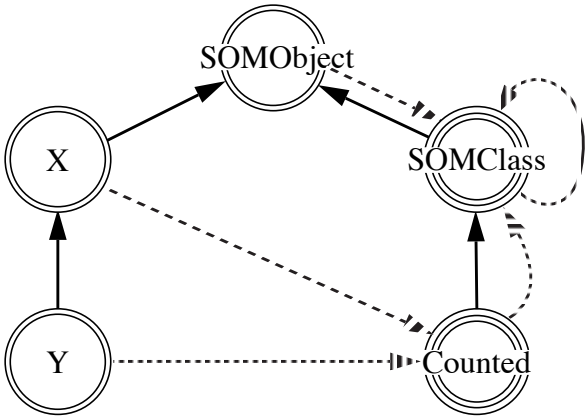

Figure 4 -- SOM 2.0 supports classes as polymorphic objects

metaclasses can serve the same purposes. A close coupling between the implementations of a metaclass and its instances is supported from IDL by using the IDL pre-processor to make such metaclasses statically visible only to a restricted set of class declarations (normally, a single class). Details of this are not important to this paper. But, the fact that all metaclasses in IDL are explicit underscores the importance of providing an improved semantics for explicit metaclasses in SOM 2.0.

SOM-derived Metaclasses

In SOM 2.0, if a metaclass is indicated in a class declaration, the new class is constructed (at runtime) either as an instance of the indicated metaclass or as an instance of some class derived from the indicated metaclass. In the second case, there are two possibilities: the metaclass may already exist as a user-defined metaclass, or, if necessary, SOM will *derive* it dynamically in the process of creating the new class object.

Why isn't a SOM 2.0 class always simply an instance of the metaclass indicated in its declaration (as was the case in SOM 1.0)? The answer is that SOM allows unconstrained class declarations -- even those such as illustrated by the problematic class **Y** in the above example -- while also supporting classes as polymorphic objects. For example, Figure 4 illustrates the SOM 2.0 semantics of the above problematic OIDL and IDL declarations. As shown, SOM 2.0 simply uses **Counted** as the class of **Y**. An example

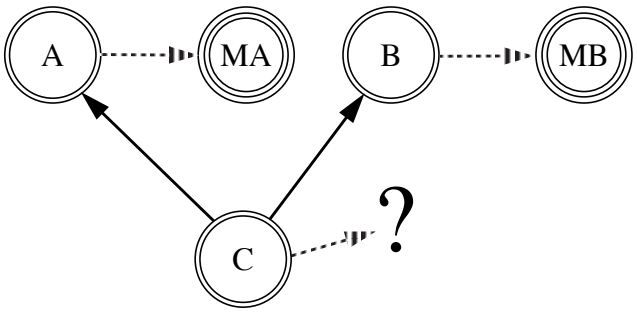

Figure 5 -- Of what class should C be an instance?

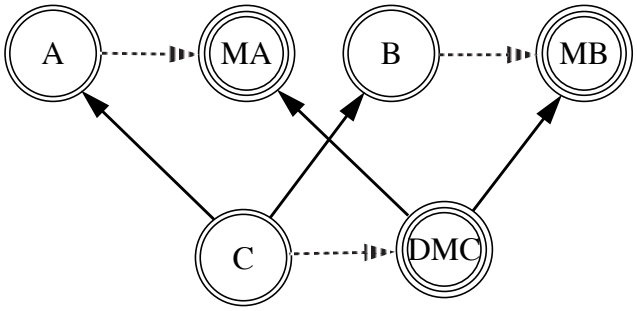

Figure 6 — The solution: C's metaclass must be derived from A's and B's metaclasses

that requires SOM to actually derive a new metaclass is provided by Figure 5. In Figure 5, a new class **C** is declared using multiple inheritance from two other classes, **A** and **B**, whose classes are, respectively, **MA** and **MB**. The question in Figure 5: What should be the class of **C**? The solution must guarantee that **C** (i.e., the class object itself) responds to the interfaces of both the **A** and **B** class objects. As illustrated in Figure 6, SOM guarantees this by deriving the class of **C** (named **DMC**, for "Derived Metaclass") from the classes of **C**'s parents.

Of course, a general solution must deal with any number of parents and an explicit metaclass (when one is indicated) Figure 7 presents the general case using IDL and illustrates the resulting SOM-derived metaclass, **DMC**. This shows how SOM uses multiple inheritance to derive a new metaclass whose instance's interfaces are compatible with both (1) the requirements indicated by the programmer that indicates a metaclass when subclassing and (2) the requirements implied by the need to support the newly-defined class as a polymorphic object with respect the class of each parent. This has the effect of treating the metaclass as a constraint rather than an

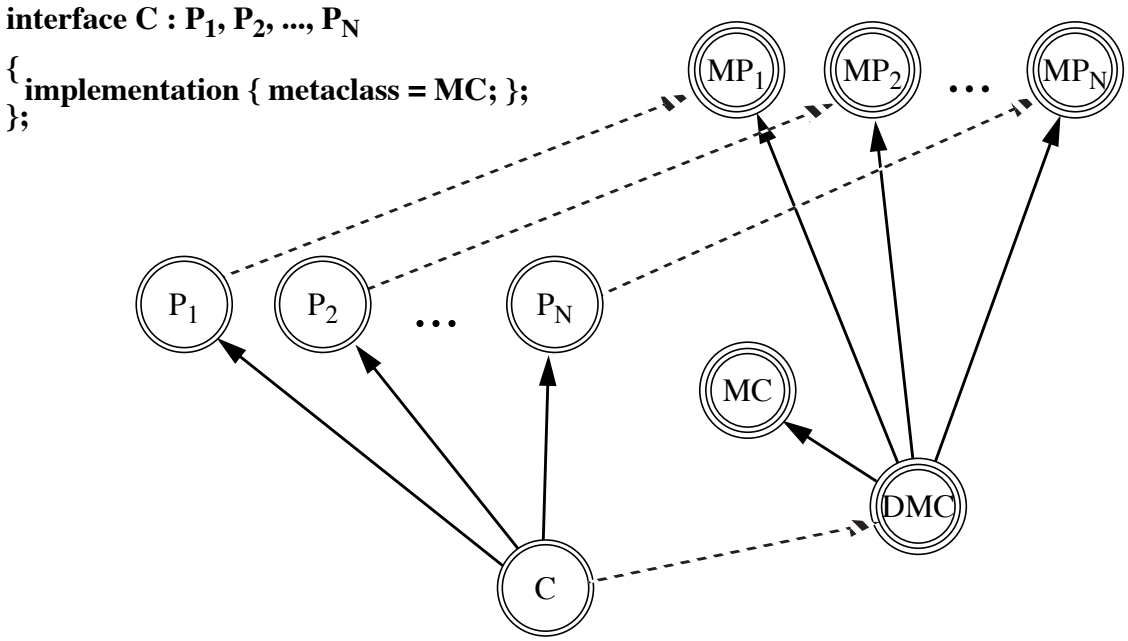

Figure 7— A General Solution for Interface Compatibility

imperative declaration. In effect, inheritance is given a new dimension, because the metaclass constraint is also inherited. The derived metaclass is the least solution to all these constraints.

Dealing with Object State

The approach illustrated in Figure 7 guarantees metaclass compatibility by assuring the existence of the necessary interfaces and class variables. But class variables must be initialized. In SOM, a class's state includes:

- an instance method table (introduced by **SOMClass**)
- a variety of other class variables (often introduced by subclasses of **SOMClass** -- i.e., other metaclasses)

And, of course, a class' state must be initialized before the class is used. As suggested earlier, the SOM API provides methods whose purpose is to provide for initialization (and uninitialization):

- `SOMObject::somInit`
  default variable initialization
- `SOMClass::somInitMIClass`
  create and initialize instance method table
- `SOMClass::somClassReady`
  register class
- `SOMObject::somUninit`
  free allocated resources

In general, a metaclass designer overrides each of these methods to perform class-specific initialization (or uninitialization). And, each of the resulting method procedures used by a metaclass implementation normally makes parent method calls to cooperatively invoke similar functionality implemented by the parents of the metaclass. To correctly initialize (uninitialize) all the variables containing a class's state, then, a SOM-derived metaclass overrides the above methods with code that makes the necessary parent method calls on all of its metaclass parents. This assures that all appropriate initialization code is executed. This is done only for these four, special initialization methods. Further details are provided elsewhere [7].

It is interesting to note the conditions under which the problem solved by derived metaclasses can arise. Metaclass incompatibility can arise in the case of single inheritance models that allow explicit metaclasses and in the case of multiple inheritance models (with either explicit or implicit metaclasses). Smalltalk has implicit metaclasses but doesn't support multiple inheritance. C++ allows multiple inheritance but doesn't allow metaclasses. Thus, neither of these

object models needs to worry about metaclass incompatibility. On the other hand, object models such as ObjVLisp [4] and CLOS [2] can encounter the problem [12]. None of these systems provide the benefits offered by SOM-derived metaclasses.

SOM is unique in that it relieves programmers of the responsibility for avoiding metaclass incompatibility when defining a new class. At first glance, this might seem to be merely a useful convenience. But, in fact, it is essential because SOM must provide backwards binary compatibility with respect to changes in class implementations. A programmer might, at one point in time, know the classes of all the ancestor classes of a new subclass (and so on, recursively), and, as result, be able to explicitly derive an appropriate class for a new subclass using the approach we have described above. But, we doubt that a system based on requiring this would ever be successful. And, in any case, SOM must guarantee that a class implementation continues to execute and function correctly when its ancestor class's implementations are changed without retracting existing interfaces. This includes specifying different parents or different metaclasses, so requiring a static solution (on the part of either a programmer or a compiler) is simply not acceptable in SOM.

SOM-derived metaclasses enable functionality packaged as SOM classes to be combined into a large number of different configurations, thus supporting code reuse. Software reuse is one promise of OOP technology, and it has been gratifying to find that many useful kinds of functionality are automatically composed by SOM-derived metaclasses. Examples include aspects of DSOM (a SOM class framework allowing transparent distribution of objects according to the CORBA model [10]), Replication (a SOM class framework providing single-copy serializability for distributed, replicated objects [10]), and "before/after" metaclasses [8]. Another example is provided by the metaclass cooperation framework, described in the following section.

Metaclasses in SOM 2.1

SOM-derived metaclasses do not solve all the problems encountered by SOM metaclass programmers. In particular, the dynamic behavior of different metaclasses combined into a derived metaclass may result in "interference" between these metaclasses. This problem does not manifest itself as a lack of polymorphism (the problem identified in [12]), but as an operational conflict between different metaclasses' behavior. For example, this problem could arise when two unrelated metaclasses combined into a derived metaclass want different method procedures to execute when a new object is created with `somNew`. If the two metaclasses both use their `somInitMIClass` code to override the object creation method `somNew`, then the last override to execute "wins," interfering with the other metaclass, whose method procedure not execute.

The question is how to cooperatively associate a multiplicity of method procedures with a single method. When method procedures are contributed by classes related by inheritance, parent method calls serve this purpose. But, when a method procedure is contributed by a metaclass, parent method calls cannot do the job of enabling cooperation. Remember that SOM-derived metaclasses are determined dynamically; those metaclasses combined into a SOM-derived metaclass do not have any static inheritance relations among themselves (otherwise there would be no reason to derive a new metaclass).

We address this problem with a "metaclass cooperation framework." This provides a programming model in which metaclasses achieve their objectives cooperatively by combining different method procedures into a "cooperation chain." Interference (when it would otherwise occur) is identified as conflicting requirements for ordering method procedures in this chain. This maximizes the opportunity for cooperation between metaclasses (because, for most purposes, ordering doesn't matter) and it guarantees that metaclasses never mysteriously cease to operate correctly as a result of interference with other metaclasses.

Next we re-implement **Counted** to provide supporting details. The following examples are expressed according to the cooperation framework provided by ESOM, a current research prototype for SOM 2.1. ESOM is not a product, so these examples are for illustration only. The APIs provided by the SOM 2.1 product may vary from those shown here.

We start with a metaclass that implements an instance count without using the cooperation framework, and then we show how the same objective can be achieved with the framework. The IDL for an "uncooperative" metaclass might appear as follows.

```
interface Counted : SOMClass {
 readonly attribute long instanceCount;
 implementation {
   somMethodProc* doFree;
    // a class variable explained below
   somInit: override;
    // to initialize instanceCount
   somNew: override;
    // to increment instanceCount
   somInitMIClass: override;
    // explained below
 };
};
```

First note the above **Counted** cannot be reliably combined into a SOM-derived metaclass with any other metaclasses that overrides `somNew`. As explained above, if some other metaclass overrides `somNew`, and this metaclass (call it **MC**) and **Counted** happen to be automatically combined into a SOM-derived metaclass during subclassing, then it would use either **MC**'s `somNew` or **Counted**'s. Despite this potential problem, below is an implementation for **Counted** using DirectToSOM C++ (i.e., C++ compiled to the SOM API, as provided by the MetaWare AIX and OS/2 C++ compilers). The file `Counted.hh` is a C++ header file produced from `Counted.idl` by a SOMObjects Toolkit compiler

Although the above solution is not cooperative (due to its override of `somNew`) the handling of the instance method table entry for `somFree` is similar to the way that the cooperation framework is designed.

```
#include <Counted.hh>

Counted::Counted()
{instanceCount = 0; }

Counted::somNew()
{instanceCount++;
 return SOMClass::somNew(); }

void somFree(SOMObject& obj) // a function used below
{
 obj.somGetClass().instanceCount--;
 obj.doFree();
 /* doFree is set in somInitMIClass, below */
}

Counted::somInitMIClass(long inherit_vars,
                        string className,
                        SOMClassSequence* parentClasses,
                        long dataSize,
                        long dataAlignment,
                        long maxStaticMethods,
                        long majorVersion,
                        long minorVersion)
{
 /* Do parent method call to chain somInitMIClass upwards.*/
 SOMClass::somInitMIClass(...);
 /* Record instance mtab entry for somFree in a class variable.*/
 doFree = somClassResolve(this,SOMObjectClassData.somFree);
 /* Replace original somFree entry with the above function. */
 somOverrideSMethod( "SOMObject::somFree", somFree );
}
```

`Counted::somInitMIClass` remembers the initial content of the `somFree` entry of the instance method table, and this is later called by the locally-registered routine for `somFree`, after decrementing the class's instance count. This is very similar to a parent method call, but is not based directly on inheritance. Rather, it is simply based on whatever the content of the instance method table is when `Counted::somInitMIClass` saves the `somFree` entry.

If the technique illustrated above for `somFree` were all that a metaclass programmer needed to avoid interference, then there would be little need for a cooperation framework. Metaclass programmers could simply use this technique to achieve the desired results. But, complications arise from providing control over the cooperation chain ordering and, also, from handling parent method calls correctly. As a result, the methods introduced by the cooperation framework are extremely important -- they solve a number of difficult problems and they offer a simple-to-use interface for metaclass programmers. These methods are now described using IDL.

```
somMethodProc**
sommAddCooperativeInstanceMethod(
          in somId methodId,
          in somMethodProc* coopProc);
```

This method installs a cooperative override in the receiver's instance method table and is the cooperation framework analogy to the technique illustrated in the above example for handling `somFree`. The returned result is the location of the method procedure pointer that must be invoked by `coopProc` to continue the cooperation chain. This location is maintained and used by a class to support cooperation chain ordering.

```
somMethodProc**
sommAddCooperativeClassMethod(
          in somId methodId,
          in somMethodProc* coopProc);
```

This method installs a cooperative override in the instance method table of its receiver's class. In other words, this method allows a class to change its own behavior (as opposed to the behavior of its instances) by modifying the instance method table of the class of which it is an instance. This is how cooperation on class methods (such as `somNew`) is achieved, and provides an interesting use of reflection within the cooperation framework.

The above two methods can be used by metaclasses without any possibility of interference. In contrast, metaclasses using methods that request a particular position in the cooperation chain (specifically, the first or last position) may interfere with each other. This possibility is handled by allowing each metaclass to build up a request block by making requests, and then asking to have the request block satisfied.

```
boolean sommSatisfyRequests();
```

When this method is invoked, the class's current request block is checked to see if any new requests conflict with previously-granted requests. If so, none of the new requests are granted and FALSE is returned. Otherwise all the new requests are granted. Instead of returning a result, the request methods themselves all accept an extra output argument that is the address of a variable that the caller wants loaded with the location of its cooperation chain method pointer (if the request is satisfied upon later use of `sommSatisfyRequests`).

```
void
sommRequestFirstCooperativeInstanceMethodCall(
  in somId methodId,
  in somMethodProc* coopProc,
  out somMethodProc** chainProcAddrAddr);
```

This method is similar to `sommAddCooperativeInstanceMethod,` but requests that `coopProc` be the first cooperation chain method procedure that is called when the indicated method is invoked on an instance of the class being initialized..

```
void
sommRequestFirstCooperativeClassMethodCall(
  in somId methodId,
  in somMethodProc* coopProc,
  out somMethodProc** chainProcAddrAddr);
```

This method is similar to `sommAddCooperativeClassMethod`, but requests that `coopProc` be the first cooperation chain method procedure that is called when the

indicated method is invoked on the class being initialized.

```
void
sommRequestFinalClassMethodCall(
      in somId methodId,
      in somMethodProc* methodProc);
```

This method *requests* that the indicated `methodProc` be called to provide the "final" semantics for the indicated class method. Note that no output argument is used to support cooperation -- the final method simply returns a result. The last two methods are both reflective -- a class object invokes these methods on itself in order to change its future behavior.

The notion of a method name corresponding to a set of implementations is also employed in Subject Oriented Programming [13].

Using the Cooperation Framework

Using the methods described above, a metaclass `CoopCounted` can cooperate on the class method `somNew` and the instance method `somFree` as illustrated below. Note that no special ordering of cooperation chain methods is required.

```
interface CoopCounted : SOMMCooperative
{
 readonly attribute long instanceCount;
 implementation {
   somMethodProc**  doFree;
   somMethodProc**  doNew;
   somInit: override;
        // to initialize instanceCount
   somInitMIClass: override
        // to register cooperation
 };
};

#include <CoopCounted.hh>

Counted::Counted()
{instanceCount = 0; }

CC_somFree(SOMObject& obj)
{
 obj.somGetClass().instanceCount--;
 *doFree(obj); /*cooperate on somFree*/
}

CC_somNew(CoopCounted& somSelf)
{
 somSelf.instanceCount++;
 return *doNew(somSelf);
      /*cooperate on somNew*/
}

CoopCounted::somInitMIClass(...)
{SOMClass::somInitMIClass(...);
 doFree =
   sommAddCooperativeInstanceMethod(
     "SOMObject::somFree",CC_somFree);
 doNew = sommAddCooperativeClassMethod(
          "SOMClass::somNew",CC_somNew);
}
```

The above example provides a simple illustration of ideas and techniques used for metaclass programming in ESOM.

Comparison with CLOS

In comparison with many other OOP models, SOM 2.0 provides enhanced opportunities for using classes to encapsulate useful functionality, and therefore enhances code reuse. To guarantee metaclass compatibility, the SOM 2.0 runtime uses multiple inheritance to derive metaclasses from which polymorphic class objects can be instantiated. Due to

the complexity of correctly supporting classes as polymorphic objects, it seems unlikely that the power of explicit metaclasses would be generally useful without this support. Finally, to aid in preventing interference between different metaclasses combined into a derived metaclass, ESOM provides a metaclass cooperation framework that allows metaclasses to achieve their objectives cooperatively by creating cooperation chains for both instance and class methods.

It is interesting ask to whether CLOS could also do these things, and, if so, how. By default, CLOS requires a subclass to have the same metaclass as its parent(s). While this prevents metaclass incompatibility, it also removes most of the benefit of explicit metaclasses. But, experienced CLOS users have indicated to the authors that this policy could be changed on a per-application basis by suitable use of the CLOS Metaobject Protocol, and that derived metaclasses could thereby be integrated into the overall semantics of CLOS class definition by automatically creating appropriate metaclasses (as in SOM) whenever necessary.

In CLOS, multiple inheritance is supported by linearizing ancestor classes into a *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 would allow the necessary method chaining as required for initialization of derived metaclass' instances' state -- likely via the CLOS `initialize-instance` method.

Creation and use of "cooperation chains" for methods also seems possible in CLOS. However, just as the parent-method call paradigm in SOM is too limited in flexibility, so too would be use of `call-next-method` through the class precedence list. The right way to view the cooperation chain for a method is that it is orthogonal to parent calls. For each different method, the chain is built up dynamically, as different classes' initialization code (defined by the different metaclasses combined into a derived metaclass) is executed.

Arranging for appropriate interaction between parent calls for a method and the calls contained within a cooperation chain for the method presented a challenge in SOM, and the solution was encapsulated using classes (two public metaclasses make up the cooperation framework). It seems likely that CLOS could also create and encapsulate a similar solution. This might be done using the CLOS ability to support multiple primary methods. In between invocation of `:before` and `:after` methods in CLOS, the `apply-methods` function orders the execution of any number of primary methods. As a result, metaclasses in CLOS might simply be able to add new primary methods. A remaining detail to consider would be the need for a metaclass to request a particular position among the primary methods.

Clearly, the mechanisms provided by CLOS and SOM are somewhat different. Yet it seems clear that the ideas incorporated in SOM for support of explicit metaclasses have general applicability for other systems in which classes are first class objects and explicit metaclasses are available.

Conclusion

SOM 1.0 allowed explicit metaclasses, but really only provided reliable support for implicit metaclasses. SOM 2.0 added a unique form of support for explicit metaclasses that enables their reliable use in evolving OOP systems. SOM version 2.1 then builds on this foundation to provide a metaclass cooperation framework. Experience has shown the necessity for evolution in software systems, and, clearly, SOM has been no exception. Yet, while offering greatly enhanced capabilities in comparison with the original SOM 1.0, current versions of SOM continue to support the original Workplace Shell and all its associated applications. Thus, although SOM's objective was to support evolution of class libraries in general, the second principle (which used SOM to implement SOM) has resulted in similar support for SOM's evolution.

This report focused on aspects of SOM's evolution that relate to metaclass programming. This evolution has been influenced by our experiences constructing useful metaclasses (a few of which were mentioned here), and by the addition of multiple inheritance. There are a host of other areas in which SOM has evolved, but these are topics for other papers.

Acknowledgments

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 Gregor Kiczales, Ralph Johnson, and the anonymous OOPSLA referees for their valuable and greatly appreciated comments.

References

1. Bobrow, D.G., Kahn, K., Kiczales, G., Masiner, L., Stefik, M., and Zdybel, F. "CommonLoops --- Merging Lisp and Object-Oriented Programming," *OOPSLA '86 Conference Proceedings*, 1986.
2. Bobrow. D. G. and Kiczales, G. "The Common Lisp Object System Metaobject Kernel: A Status Report," *Proceedings ACM Conference on Lisp and Functional Programming*, July, 1988.
3. Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G. and Moon, D.A., "Common Lisp Object System Specification" *Sigplan Notices*,Vol. 23 (September 1988).
4. Briot, J.-P. and Cointe, P. "A Uniform Model for Object-Oriented Languages Using the Class Abstraction," *IJCAI* Vol. 1, August, 1987.
5. Cointe, P. "The ObjVlisp Kernel: A Reflexive Lisp Architecture to Define a Uniform Object-Oriented System," in *Meta-Level Architectures and Reflection* Pattie Maes and Daniele Nardi (ed.)North-Holland, 1987.
6. Danvy, O. and Malmkjaer, K.. "Intentions and Extensions in a Reflective Tower," *Proceedings ACM Lisp and FP Conference*, 1988.
7. Danforth, S. and Forman, I.R. "Derived Metaclasses in SOM," *Proceedings TOOLS Europe '94*, 1994.
8. Forman, I.R., Danforth, S. and Madduri, H. "Composition of Before/After Metaclasses in SOM," *OOPSLA '94 Conference Proceedings*, 1994.
9. Friedman, D. and Wand, M. "Reification: Reflection without Metaphysics," *Proceedings ACM Lisp and FP Conference*, 1994.
10. S*OMObjects Developer ToolKit, Users Guide*, IBM, June, 1993.
11. Graube, N. "Reflexive Architecture: From ObjV-Lisp to CLOS," *Proceedings ECOOP '88*, Springer Verlag LNCS Vol. 322, 1988.
12. Graube, N. "Metaclass Compatibility," *OOPSLA '89 Conference Proceedings*, 1989.
13. Harrison, W. and Ossher, H. "Subject-Oriented Programming (A Critique of Pure Objects)" *OOPSLA '93 Conference Proceedings*, 1993
14. Ingalls H.H. "The Evolution of the Smalltalk Virtual Machine," in *Smalltalk-80 Bits of Wisdom Words* of Advice G. Krasner (ed.) Addison-Wesley 1983.
15. Kiczales, G., des Rivieres, J. and Bobrow, D. G. *The Art of the Metaobject Protocol*, MIT Press, Cambridge MA, 1991.
16. Maes, P.*Computational Reflection* Ph. D. Thesis, Artificial Intelligence Laboratory, Vrije Universiteit, Brussel, 1987
17. T*he Common Object Request Broker: Architecture and Specification,* Revision 1.1, Object Management Group and X/Open, 1993.
18. Smith, B.C. *Reflection and Semantics in a Procedural Language* Ph.D.Thesis, Laboratory for Computer Science, MIT, 1982.
19. Smith, B. "Reflection and Semantics in Lisp," *Proceedings ACM Lisp and FP Conference*, 1993
20. Tomlinson, C. and Singh, V. "Inheritance and Synchronization with Enabled Sets," *OOPSLA '89 Conference Proceedings*, 1989.
21. Wand, M. and Friedman, D. "The Mystery of the Tower Revealed," *Proceedings ACM Lisp and FP Conference*, 1986.
22. Watanabe. T. and Yonezawa, A. "Reflection in an Object-Oriented Concurrent Language," *OOPSLA '88 Conference Proceedings*, 1988.