Inheritance of Metaclass Constraints in SOM

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

Contact: ira@austin.ibm.com

Abstract

Reflective programming techniques are useful and arise naturally in those forms of Object-Oriented Programming where classes are objects. Within this context, an obvious step is to encapsulate generalized reflective behavior within class frameworks to enable reuse of this behavior through inheritance. But, to accomplish this, it is essential that classes (as objects) respond to all methods responded to by their ancestors. Otherwise, inherited reflective code is not necessarily valid for subclass instances, which can result in method resolution failures. This potential problem has been called metaclass incompatibility.

In SOM, we are led inescapably to the conclusion that metaclass declarations for classes must be treated as behavioral constraints, and that subclassing should maintain these constraints. We believe this is the only reasonable approach to preventing metaclass incompatibility and enabling inheritance of reflective code. We believe this approach is further supported by the utility of the computational metaphors it enables. This paper addresses both these points.

Introduction

Class-based object-oriented programming synthesizes programming with knowledge representation. When classes are objects, the level of programming is further raised with operations on classes themselves. To the extent that these operations mediate or support the construction of new classes, programmers may even control the semantics of subclassing. Subclassing is an important composition operation in Object-Oriented Programming, so this illustrates an interesting aspect of reflection: programmers can influence the semantics of their program composition operations.

For example, two operations that might be associated with subclassing are *class join* (that is, using multiple inheritance to form a union of instance methods and instance variables with some discrimination scheme to handle conflicts), and *specialization* (adding new instance methods and instance variables, and overriding inherited functionality with new implementations). Specialization itself is often augmented with operations (such as “parent method call” or “call-next-method”) that enable incremental extension of an inherited implementation.

In SOM, there are indeed methods on class objects that mediate all of these operations [4]. But, we believe the power this brings is most naturally employed within a wider context in which programmers develop additional conceptual operations related to classes, and introduce new methods to support them. Thus, our objective is to have an open implementation provided by reflection [13], in which operations on classes can be created and inherited as needed.

We have demonstrated this capability in SOM by using explicit metaclasses to capture general object properties not associated with any particular class. Examples of such metaclasses are Persistent and ThreadSafe. A useful way to view such metaclasses is that they are mappings of the set of class objects into itself. That is, if Dog is a class, then so is Persistent(Dog).

A metaobject protocol [12] is important to the efficacy of this approach, because information about the implementation of an arbitrary class object at runtime is required to implement general mappings of this kind. Furthermore, in our experience, implementing such mappings can feed requirements back to the metaobject protocol. For example, this process resulted in the SOM metaclass cooperation framework [4].

In any case, given the ability to create a set of useful "generic" mappings, we believe it is essential that they be composable. That is, a programmer should be able create the Dog class and then create the thread-safe, persistent dog class with a composition like ThreadSafe(Persistent(Dog)).[1]

In SOM we have achieved this objective. Because it is based on inheritance of metaclass constraints, our metaclass composition approach is automatically integrated with subclassing and enjoys a number of appealing algebraic properties. For example, inheritance of metaclass constraints leads to the composition of two metaclass mappings being equivalent to the mapping defined by the join of the two metaclasses. Also, application of a metaclass distributes over class join. These properties provide important aid for maintaining intellectual control over a complex environment of classes and metaclasses.

We believe our approach is essential if one is to encapsulate reflective solutions within explicit metaclasses. Without inheritance of metaclass constraints, subclassing becomes an overwhelming problem for the programmer due to the burden of "getting the metaclass right." With the right support, however, automatic inheritance of functionality provides an elegant affirmation for both reflection and OOP.

A Overview of SOM

Let us introduce the notation that is to be used to describe our object model. First there are the graphical symbols for depicting a set of objects and its interrelationships. These are shown in Figure 1. When a class object defines an instance method (either by introducing a new instance method or overriding an inherited instance method), the name of the method is written to the lower right of the class object. Overrides are parenthesized.

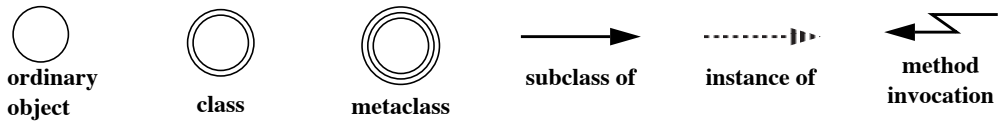

Figure 1. Graphical symbols.

The state of an example SOM environment is depicted in Figure 2. There are four objects SOMObject (a class), SOMClass (a metaclass), Dog (an ordinary class), and Rover (an ordinary object). There are two relations among objects that one must understand.

1. If they were not composable, then a combinatorial argument (given p properties, there are 2^p combinations) asserts that little has been accomplished, because one must do special programming to create the composition.

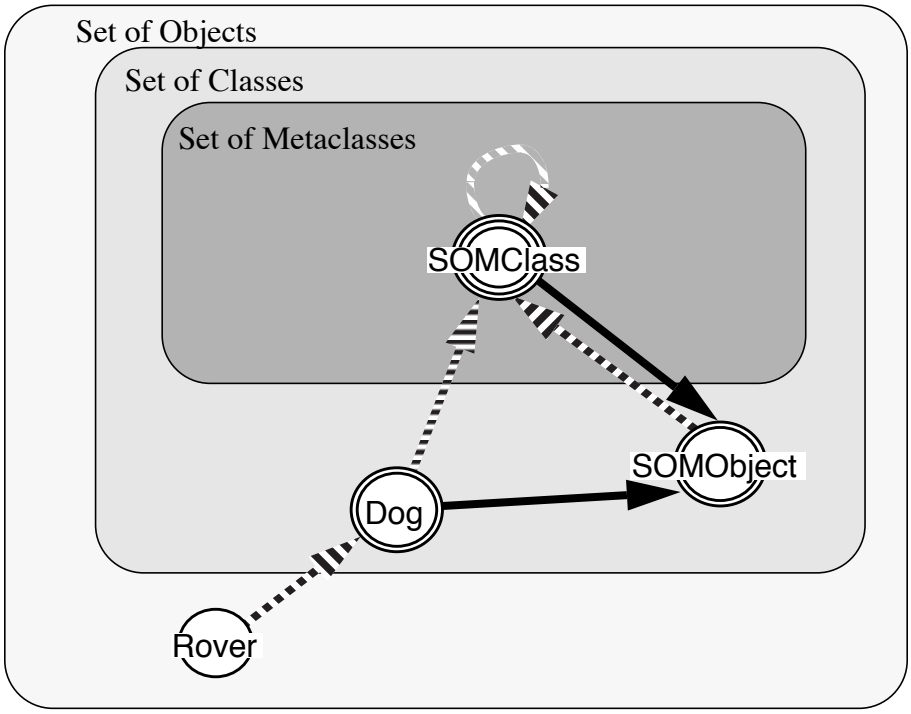

Figure 2. Example of various SOM objects.

First, there is the *instance-of* relation between objects and classes depicted by the dashed arrow from an object to its class. When convenient, the inverse relation *class-of* is also used. SOMObject is an instance of SOMClass and SOMClass is the class of itself. An object's class is important because an object responds only to the methods that are supported by its class (that is, the instance methods that the class introduces or inherits). The metaclass of any object is reached by two traversals of the instance-of relation. An object's metaclass is important because reflective code will invoke methods on the object's class, which responds only to the methods support by the object's metaclass.

Second, there is the *subclass-of* relation between classes, 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.

When new subclasses are declared in SOM, a metaclass *for* the new class can be indicated. This is called an explicit metaclass. Not providing an explicit metaclass is equivalent to declaring SOMClass as the explicit metaclass. In the absence of metaclass incompatibility, the explicit metaclass for a class will be the metaclass of the class's instances. However, SOM's solution to avoiding metaclass incompatibility means that sometimes the explicit metaclass for a new class may not actually be used as the class of the new class. Instead, the explicit metaclass may be an ancestor of a derived metaclass automatically created to avoid metaclass incompatibility. This will be explained below, in the following section.

Within this overall context, SOMObject introduces the methods to which all SOM objects respond. As a subclass of SOMObject, SOMClass supports methods available on all objects, and in addition introduces new 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. SOMClass and SOMObject are the two most important classes of the SOM kernel.

Similar arrangements of classes are also used in CLOS[12], ObjVlisp[2], Dylan[1],, and Proteus[16]. The reason for the similarity is the following theorem (see [11] page 200).

> A finite acyclic directed graph has at least one node of out-degree zero.

By considering the directed graph of the instance-of relation, this theorem yields insight into three distinct approaches to object models. First, there is the approach taken by certain object-oriented database projects [14] in which metaclasses are not objects that have classes (rendering the theorem inapplicable). Alternatively, one has models in which all objects have a class and all classes (including metaclasses) are objects. In this case, the theorem implies that either the graph of the instance-of relation must be infinite or there must be a cycle. Corresponding to the first possibility, there is the infinite tower of computational reflection [17]; in this approach although the graph is infinite, only the needed levels are generated. Finally there is the approach we are taking in which the graph has a cycle. This approach was introduced in Smalltalk-80, where the cycle actually contains two objects (see [9] page 271).

As this overview suggests SOM relies heavily on twenty years of work on Object-Oriented Programming; SOM does, however, move down a yet untrodden path as we see in the next section.

Inheritance of Metaclass Constraints

Metaclass compatibility [10] can be understood as follows. Suppose a class A introduces a method foo that determines the class of its target and then invokes bar on this class (where bar is a method introduced by the metaclass AMeta, an explicit metaclass for A). All subclasses of A inherit foo; therefore, in order to avoid method resolution errors, every subclass of A must be an instance of a metaclass that supports bar. The only way to enforce this condition is for the programming system to adhere to the following invariant:

> The class of every class must be a descendant (with respect to inheritance)
> of the classes of each of its parent classes

or equivalently

> If X is an ancestor of Y, then classof(X) is an ancestor of classof(Y)

This second characterization makes it easy to understand why the general solution for SOM is recursive, terminating in SOMClass. More will be said about this later.

The effect of this invariant is that when using multiple inheritance to define a new subclass, the class of each parent class becomes a constraint on the class of the new class being created. When an explicit metaclass is declared, this can be handled as an additional constraint. In SOM, when necessary, a new metaclass is created to satisfy these constraints; we call this a *derived metaclass* [3], because multiple inheritance is used to solve the constraints.

This approach has a very important consequence. It allows explicit metaclasses to capture object properties without associating them to any particular class. Examples of such metaclasses are Persistent and ThreadSafe; let us assume that neither is an ancestor of the other. A useful way to view such metaclasses is that they are mappings of the set of class objects into itself. That is, if

Dog is a class, then so is Persistent(Dog); let us view such an expression as an abstraction for the syntax in any programming language that states "declare a class that has Dog as a parent and Persistent as an explicit metaclass." Consider the situation in Figure 3 where PersistentDog=Persistent(Dog). Now consider

ThreadSafePersistentDog = ThreadSafe(PersistentDog)

and ask what must be true of its metaclass Z. By our invariant, Z must be a descendant of Persistent. This means that Z cannot be the metaclass ThreadSafe. On the other hand, Z must be able to impart the thread-safe property to its instances (otherwise, the explicit metaclass declaration has no meaning). Inheritance is the most direct way for Z to obtain the ability to impart a property; so, we conclude that Z must be a descendant of ThreadSafe. If Z is a descendant of both ThreadSafe and Persistent and there are no other considerations to be taken into account, then the simplest solution is to conclude that Z must be equivalent to the join of ThreadSafe and Persistent.

This informal argument makes a very strong claim: in models in which metaclass constraints are maintained by subclassing, composition of metaclass (viewed as mappings of classes into classes) will be equivalent to the multiple inheritance join of the metaclasses.

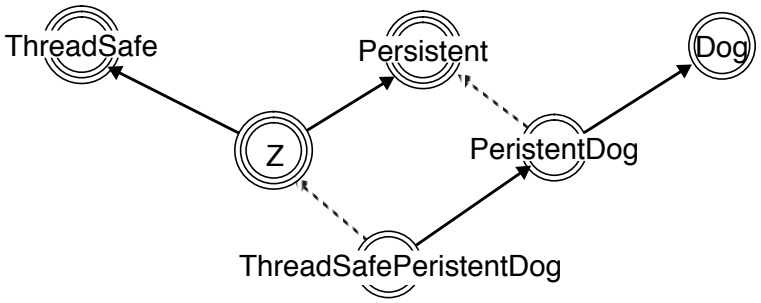

Figure 3. Z must the class that is both ThreadSafe and Persistent.

Thus, we have been led from a simple desire (elimination of metaclass incompatibilities) to an important result: composition of properties expressed as metaclasses can be equated to multiple inheritance in the metaclass hierarchy). If ⊔ represents multiple inheritance join, then

F(G(X)) = (F⊔G)(X)

where X is an arbitrary class and F and G are arbitrary metaclasses.[1]

A similar argument yields another desirable result:

(F⊔G)(X) = F(X) ⊔ G(X)

that is, application of metaclass mappings distributes over the join operation.

1. We have been somewhat cavalier in our use of symbols like F and G to denote both metaclass objects and metaclass mappings. We could formalize this argument by creating a method on SOMClass called somEndow(F,X), which returns the class object that is an instance of F and subclass of X. The metaclass mapping associated with F is the curried function defined by somEndow(F,X). This formalization is left for a future paper.

Inheritance of Metaclass Constraints From the Programmer's Point of View

Now let us examine what inheritance of the metaclass constraint means to programmers Aaron (the programmer of class A) and Beth (the programmer of class B, which is a subclass of A).

Consider the example depicted in Figure 4, where Beth wishes to create class B as a single-inheritance subclass of A (which is an instance of the metaclass F). Without inheritance of the metaclass constraint, Beth is faced with the problem of determining the metaclass of which B will be an instance.

In this example, things are simple due to the absence of multiple inheritance and an explicit metaclass for B. The right answer is F. So, maybe it wouldn't be too unreasonable in this case to require this information from Beth. In a case like this, Smalltalk provides a precedent for doing otherwise; Smalltalk automatically creates a subclass of F as the metaclass for B without requiring any declarations from the programmer [9].

Furthermore, when multiple inheritance and explicit metaclasses are available, arriving at the right answer is not so simple as in Smalltalk (the general solution requires recursion to determine the class of the metaclass, and its class, and so on [3]). Thus, in SOM, from the standpoint of programmer convenience, it is even more important to provide an automatic solution as part of the semantics of subclassing.

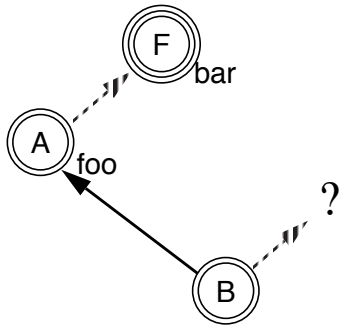

Figure 4. Programmer convenience example.

Let us pursue this idea further by considering what happens if Beth wishes B objects to have a property that is implemented by the explicit metaclass G (see Figure 5). We argued above that a composition of G with F is necessary in this case.

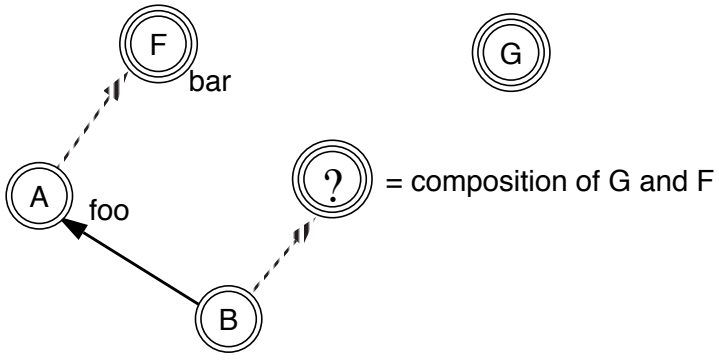

Figure 5. Programmer needs to include property implemented by G.

But is this really necessary? Suppose Beth wishes to avoid the overhead of composing G and F.

What must she know to do so safely? Among other things, she must know that no method of A is implemented reflectively, in terms of any method of F. But, this is information that Beth shouldn't have in general; subclasses shouldn't need to know how ancestors' methods are implemented. And, as explained, our interest is in allowing inheritance of reflective code. So, Beth must compose G and F. It is the only safe thing to do when reflective code can be used to implement methods.

Therefore, from Beth's point of view (as the programmer of B), the programming system should provide inheritance of metaclass constraints and automatic composition. The composition needs to be done, it can be done automatically, and, because of the complexity of the solution in general, there seems nothing to gain and much to lose if the programmer is required to do it explicitly.

Now let us consider Aaron's point of view. Suppose Aaron is considering a change to A. Figure 6 depicts a change in which A is to be reimplemented with the explicit metaclass F', a new subclass of F. If inheritance of metaclass constraints is not included in the semantics of subclassing, then Aaron cannot make this change because B (and all other existing descendants of A) may then suffer from metaclass incompatibility.

Ultimately, the implications of this (for maintaining consistency within class libraries and between different class libraries) are simply unacceptable. One reason why OOP is successful is that it makes sense in terms of its real-world benefits for software engineering. Changes in the implementation of an ancestor class must not require explicit source changes to the declaration of descendant classes. This topic is addressed in [8] where we enumerate the safe transformations that are supported in SOM. Specifically a programmer should be able to migrate the metaclass of a class downward in the metaclass inheritance hierarchy. Asserting that this transformation must preserve the correctness of descendants implies that the metaclass constraint must be inherited. That is, on the right side of Figure 6, the programming environment must make F' the metaclass of B when that class object is created.

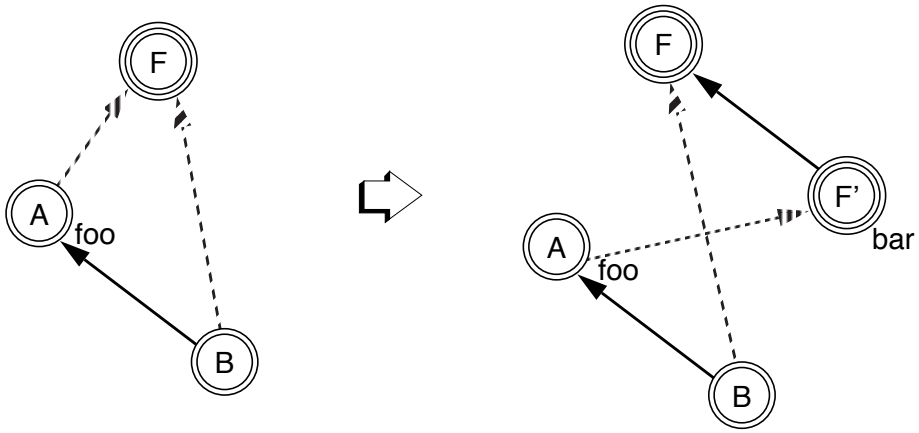

Figure 6. Evolution of a class.

Note that we have been giving the impression that Aaron and Beth are different people; the problems presented are just as troublesome if there is one person involved. Although our examples have shown B to be a direct subclass of A, all of the problems are present when B is some distant descendant of A. Thus, given enough distance between the classes and time between their development, no programmer can be expected to manage the problem of inferring that the metaclass of B needs to be changed.

The SOM Metaclass Framework

In spite of the above examples, it may still be possible to deceive oneself into thinking that a programmer should be the one to determine the actual metaclass for a class (i.e., that an explicit metaclass declaration for a class should be treated as an imperative rather than a constraint). But once one has built even a small library of reusable metaclasses, the problem of avoiding metaclass incompatibilities can become quite arduous.

Furthermore, in addition to the problems we illustrated with examples above, there is an additional difficulty that has been alluded to. Metaclasses are classes, so when solving the metaclass constraints, our new invariant must apply to the derived metaclass as well. Thus, the constraint solving procedure must be invoked recursively. This recursion terminates because the class structure is well-founded with all chains of the *instance-of* relation ending at SOMClass.

To illustrate this, we present an overview of part of the SOM Metaclass Framework. This will serve two objectives. First, it will provide evidence that we are composing interesting metaclasses. Second, it will show that the burden of getting the metaclass right (eliminating method resolution failures due to metaclass incompatibilities) is best borne by computers -- not programmers.

We start our overview by showing in Figure 7 the private metaclasses in the SOM kernel that support the notion of derived metaclasses. These are called Derived and DerivedMeta (which is a meta-metaclass). A derived metaclass always has Derived as its last parent. The SOM kernel uses this to ensure that a derived metaclass has an implementation for initialization methods (see [4]) that will invoke the initialization code of the derived metaclass parents. Thus, if the initializations compose, the metaclasses as mappings compose. The requirements for metaclass composition are:

1. Initializations must be idempotent, and
2. Initializations cannot conflict over method table entries (this seemingly strong requirement is mitigated by metaclass cooperation -- see below).

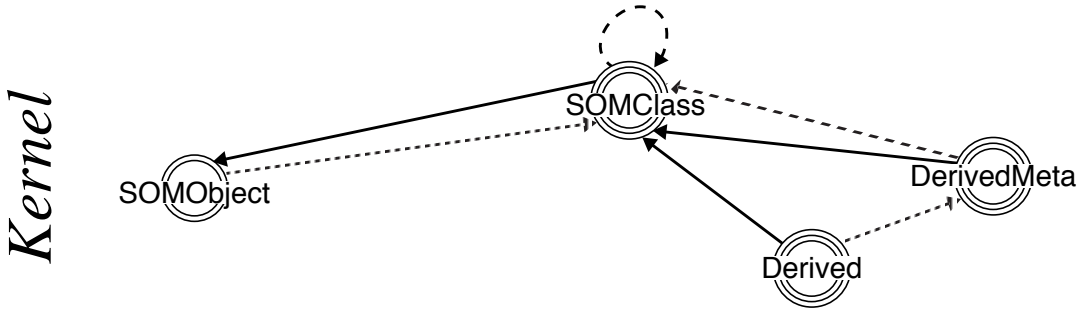

Figure 7. Additional SOM kernel metaclasses required for SOM derived metaclasses.

Metaclass programming is difficult for the following reasons.

1. A metaclass programmer must deal with two levels of indirection. That is, a metaclass programmer must understand how to mutate a class object so that its instances will have a desired property. (This difficulty would be present in any metaobject protocol.)
2. A metaclass programmer must ensure that each new metaclass does compose with other metaclasses (including metaclasses that have not yet been programmed).

The SOM Metaclass Framework is the part of our product that addresses these difficulties. The first difficulty is addressed by having metaclasses that offer specific services that ease the burden of metaclass programming. Examples are Before/After Metaclasses and the creation of proxy classes; both of these are described below. The second difficulty is addressed by the cooperation metaclasses (drawn in Figure 8).

Cooperative	This metaclass is the base for all cooperative metaclasses; it redefines the concept of a method table entry corresponding to a set of implementations that all get executed.
MetaCooperative	This metametaclass introduces data into cooperative metaclass objects.
CooperativeSistered	This metaclass ensures that its instances have a shadow copy, called a sister, which preserves the original method table.
CooperativeRedispatched	Methods in SOM are generally invoke with offset dispatching. However, one can arrange that all method invocations pass through a dispatch method; instances of this metaclass are so modified. Doing so, places "redispatch stubs" in the method table (thus, the need to preserve the original method table in the sister).

The Cooperative metaclass mitigates the requirement that composable metaclasses not conflict over the definition of a method table entry by shifting the granularity of the conflicts. Because method table entries of cooperative metaclasses correspond to a set of implementations, the conflict is shifted to which implementation gets executed first. As long as at most one metaclass (among a set of metaclass constraints) claims the right to execute first for a particular method, there is no conflict -- and the metaclasses can be composed using multiple inheritance.

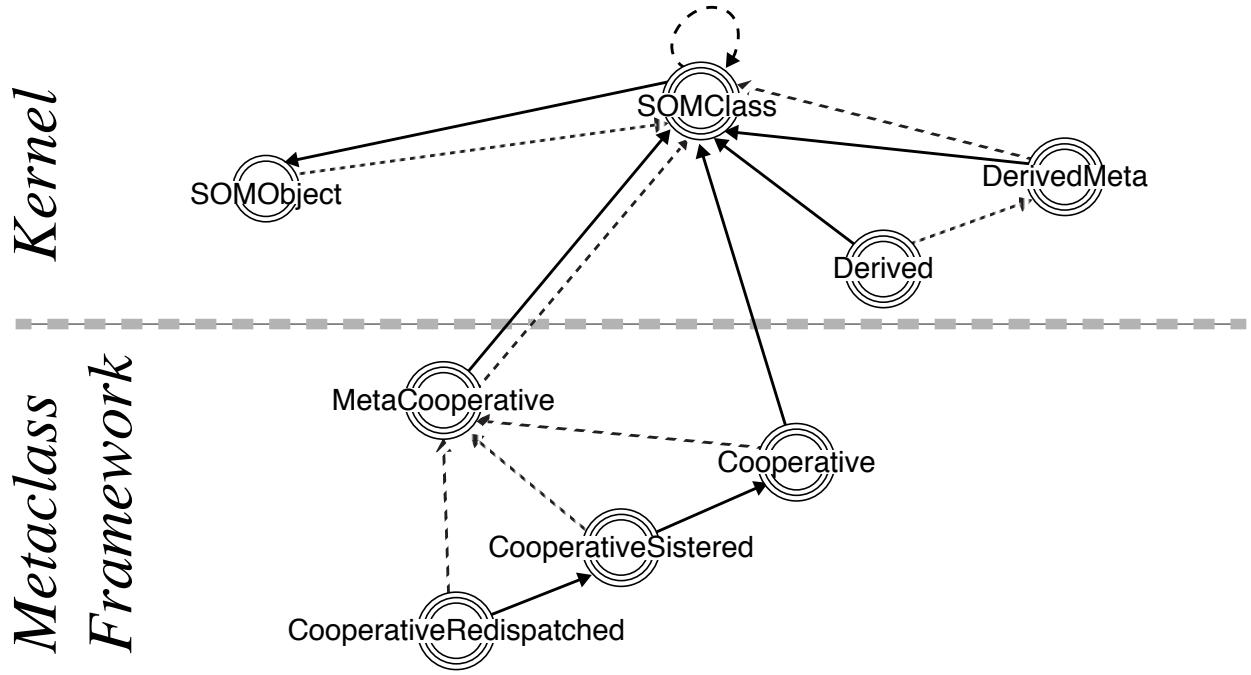

Figure 8. The metaclasses of the subframework for metaclass cooperation.

Figure 9 depicts two additional subframeworks of the Metaclass Framework. On the left are the Before/After Metaclasses [6], which are used to wrap all methods of a class with a before method and an after method. On the right is the proxy subframework, which creates proxies for objects in the same address space and is used by Distributed SOM to create proxies for remote objects [7].

BeforeAfter	This metaclass introduces the methods BeforeMethod and AfterMethod. In addition, it provides the dispatch algorithm for method invocations.
BeforeAfterDispatcher	The paper [6] describes the dispatch algorithm of before/after methods as doing a depth-first search of the metaclass hierarchy at method invocation time. The hierarchy above a metaclass does not change after the class is created. The actual implementation does the search at class creation time; it compiles a list of before/after method into each Before/After Metaclass.
Traced	This is a utility metaclass in which the before/after methods provide method tracing (this metaclass was built both because it is a good test of BeforeAfter and it is a useful utility).
ProxyFor	This metaclass supports a method for creating a proxy class for any class.
ProxyForObject	This is base class for all proxy classes.

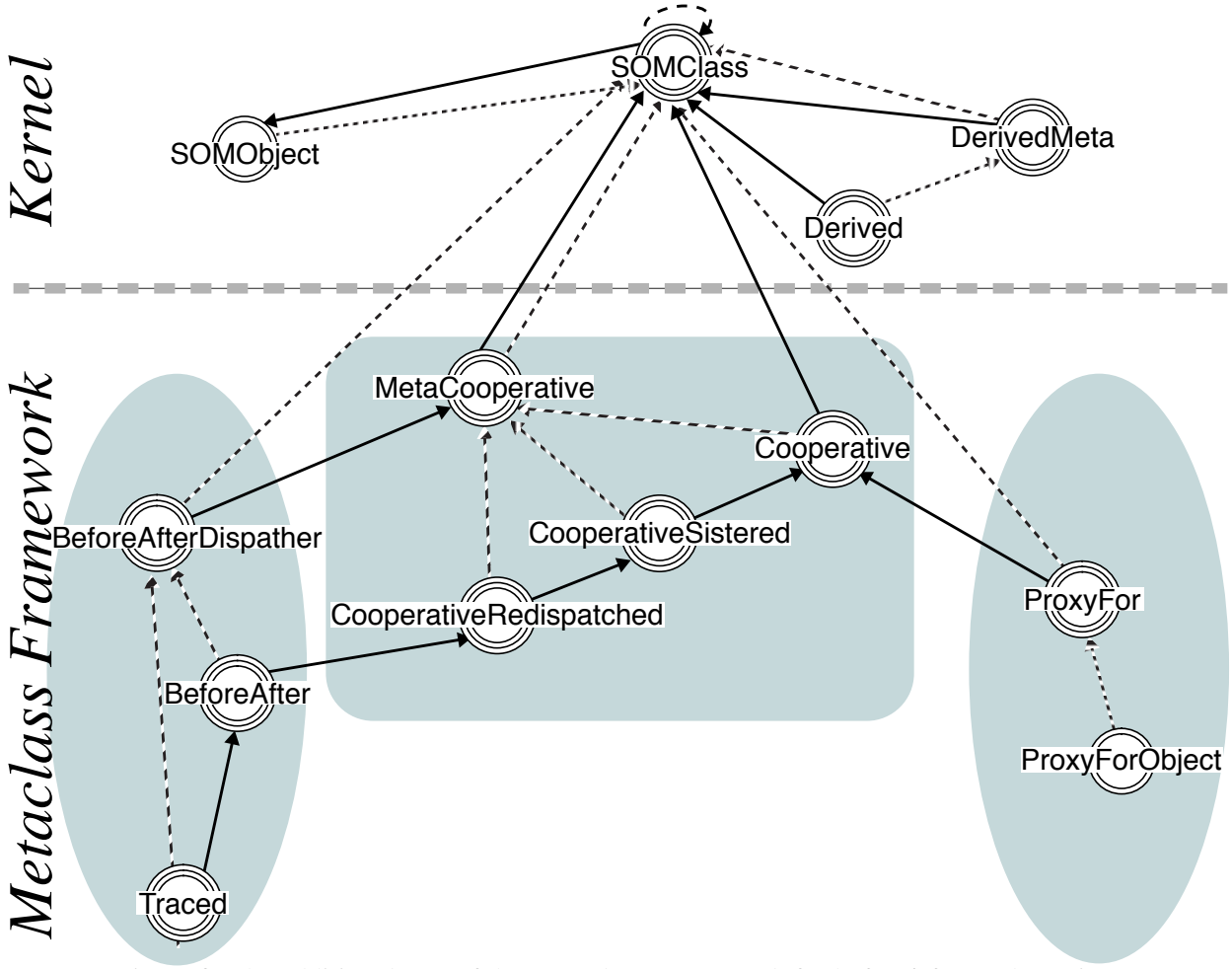

Figure 9. The additional part of the Metaclass Framework for before/after and proxies.

An Example of Recursion in Solving Metaclass Constraints

We have introduced the Metaclass Framework as preparation for the following example. Suppose one wishes to trace the proxies for instances of the Dog class. This can be done quite easily in SOM, simply by creating a class with parents Dog and ProxyForObject and with Traced as an explicit metaclass. Figure 10 shows the class structure that results. Note that the desired class, TracedProxyForDog, is an instance of a derived metaclass (TracedProxyForDog_Derived), which in turn is an instance of a derived metaclass (TracedProxyForDog_Derived_Derived).

This example is not contrived in any way. Runtime class hierarchies such as this are common occurrences in SOM reflective programs. Yet, providing inheritance of metaclass constraints results in simple compositional solutions from the programmer's perspective. In the above example, the programmer writes no method code at all, but simply defines the new class. This is what reusability is all about.

The point of this example is that getting the metaclass right can be a difficult problem for a programmer to solve. The solution requires full knowledge of the class structure above the explicit parents and explicit metaclass (that is, all classes reachable by either parent or instance links). Even if such knowledge were made available to the programmer, requiring the programmer to get explicit metaclasses "right" would be problematic. Considering the attendant need to redefine descendant classes when ancestor implementation change, programmers would demand computer support and derived metaclasses would be the result.

In fact, the information necessary to solve the above problem is not completely available to a programmer. Many of the classes in Figure 9 are deliberately private. The public view of SOM metaclasses is shown in Figure 11.

Information hiding is an important consideration for software engineering. Certainly, the job of the programmer is greatly simplified when there is less to know, but, perhaps even more importantly, information hiding enables change. This was the primary reason for making many of the classes in the SOM Metaclass Framework private. If all aspects of a class hierarchy must be seen by subclass programmers in order to avoid metaclass incompatibility, this ability is severely compromised.

It is very important to us that SOM should have the freedom to change and improve its implementation for many of the aspects of metaclass support discussed in this paper. Even after all of the interfaces are perfected, there will be no need to complicate the programmer's life by exhibiting the three meta-metaclasses DerivedMeta, MetaCooperative, and BeforeAfterDispatcher. Indeed, recent developments suggest that Derived, DerivedMeta, and MetaCooperative will actually vanish in the near future. Because programmers' code doesn't refer to these classes, and because metaclass constraints are solved dynamically by SOM as part of subclassing, all existing code will continue to run when these changes are made. The only thing that will change is the runtime collection of classes that are constructed when a user's code runs (i.e., Figure 10).

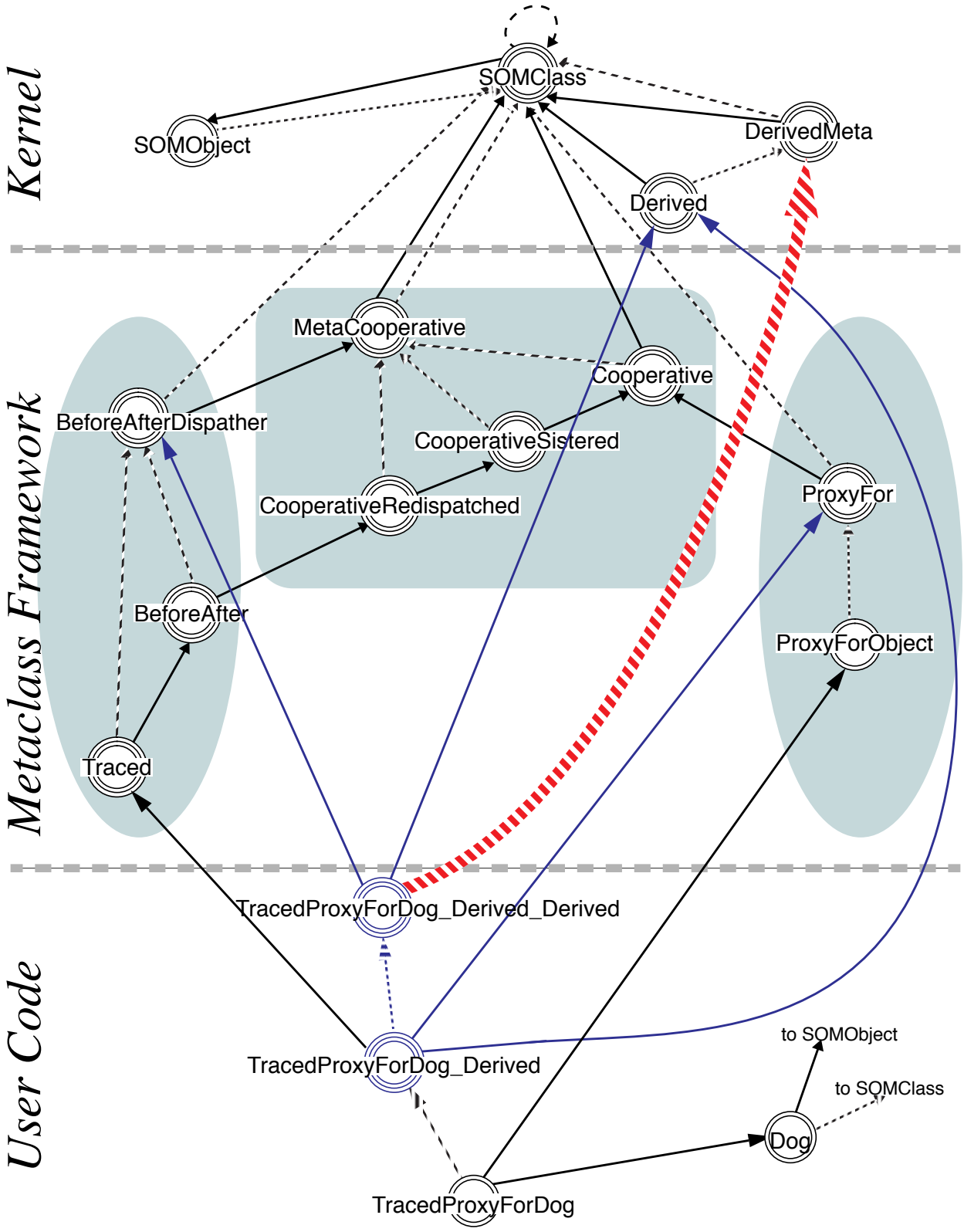

Figure 10. The complexity of creating a traced proxy for Dog.

On a related note, the ability to migrate metaclass constraint downwards (as in the example of Figure 6) has also been important to us. BeforeAfterDispatcher was originally a subclass of SOMClass (because BeforeAfterDispatcher preceded the invention of the cooperation metaclasses). Later, when BeforeAfter was made a subclass of CooperativeRedispatched, a corresponding change to the parent of BeforeAfterDispatcher was overlooked. This oversight merely led to the creation of an additional derived metaclasses (for example, the class of BeforeAfter would be derived from BeforeAfterDispatcher and MetaCooperative), but all existing code continued to work. Subsequently, this oversight was corrected, and, again, there was no loss of release-to-release binary compatibility [8].

For obvious reasons, we value the ability to evolve our implementations over time, and we believe that the more complex a software system is, the more likely it is that future evolution will be desired. Certainly, we find reflective programming a complex endeavor. So, we consider it very important that programmers don't need complete knowledge of our private metaclasses to safely inherit and use reflective code provided by our public metaclasses.

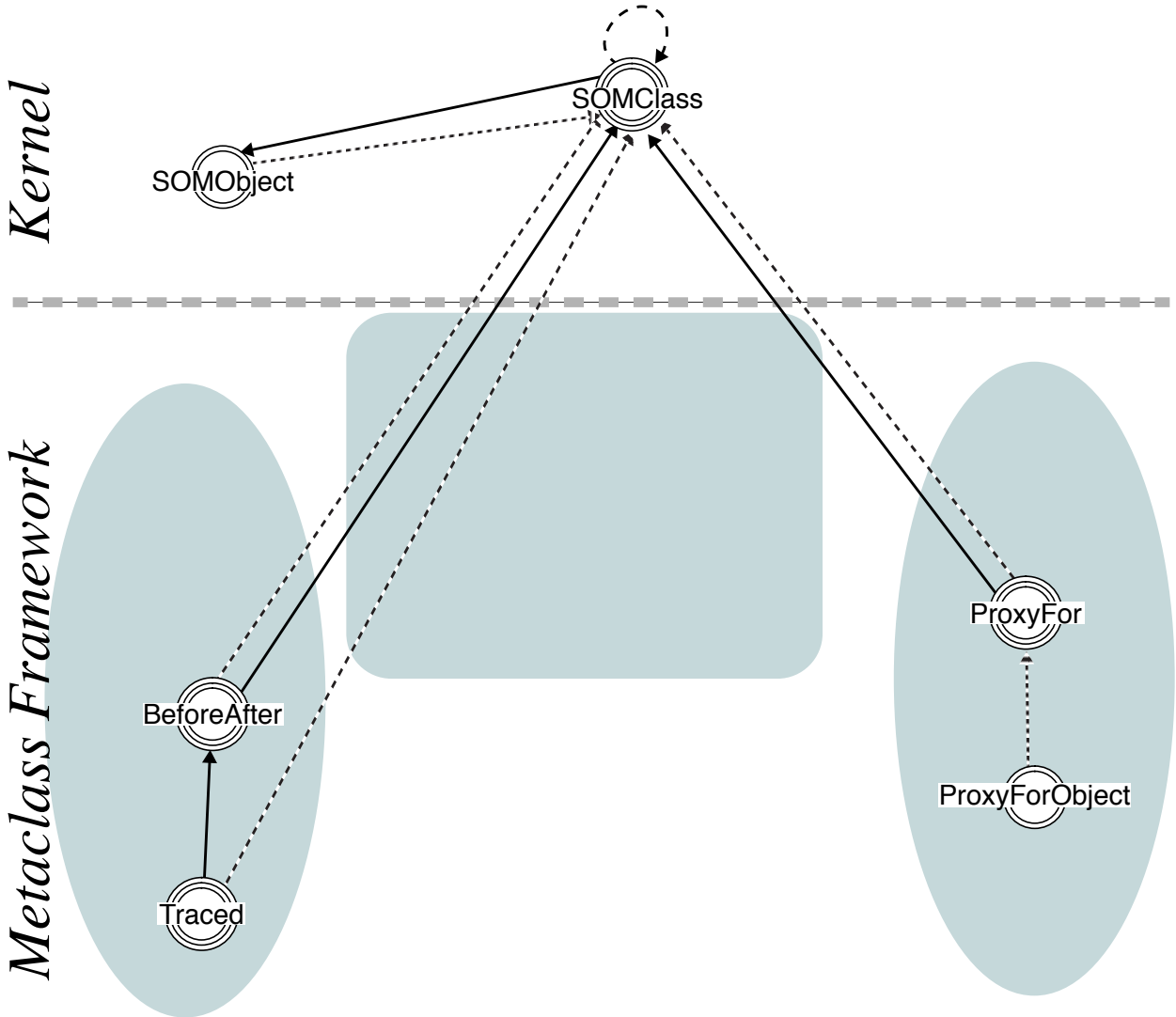

Figure 11. The public view of the Metaclass Framework.

An Epistemological Argument for Inheritance of Metaclass Constraints

In addition to its support for encapsulation and code reuse, an important contribution of object oriented programming to software engineering is that "OOP is a synthesis of programming and knowledge representation." In standard Object Oriented Programming, a class corresponds to a noun representing a set of objects with some common properties. The introduction of the abstractions from knowledge representation (such as inheritance of object properties) into programming languages greatly narrows the gap between problem analysis and software implementation, between customer and software provider, between model and program.

Roughly speaking, in the 1950's programming language design was concerned with the modeling of verbs (in COBOL procedures are called "verbs"). In the 1970's, the concern shifted to the modeling of nouns (with the invention of abstract data types and object-oriented programming). Now, based on our experience with SOM, we perceive the possibility of further strengthening the connection with knowledge representation by using metaclasses to model adjectives.

A metaclass can be used to modify a class so as to impart a property to its objects (instances). Consider the situation depicted in Figure 12. The metaclass illustrated here modifies a class object to give its instances a thread-safe property. This is most easily done using Before/After Metaclasses as described in [6], but the use of Before/After Metaclasses is not necessary to our argument -- it just grounds our claim that such metaclasses can be built. Continuing with Figure 12, we ask what is X? The answer is that X is the class of thread-safe dogs.

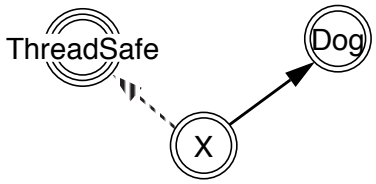

Figure 12. Modifying a class with a Metaclass.

Now let us take this one step further. Consider Figure 13 and ask what is Y? If X is the class of thread-safe dogs, then Y must be the class of secure, thread-safe dogs. This argument declares that if the classes and their instances are to form a proper "isa" hierarchy, an instance of Y must be a proper instance of X. This again implies that the metaclass constraint must be inherited. Furthermore, Figure 13 may express the desire that Y have the secure property, but the figure is not properly drawn because the dashed arrow is supposed to represent the actual instance relation (not the explicit metaclass) and we know that Y must be an instance of the composition of Secure and ThreadSafe.

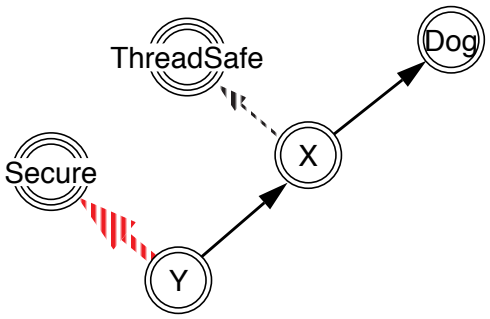

Figure 13. Implicitly Composed Metaclasses.

Let us reexamine the phrase "Y must be the class of secure, thread-safe dogs." This means that instances of Y are secure and thread-safe. Now, that "and" corresponds to a union of properties when we intend that classes represent a set of objects. In class structures, multiple inheritance is used to express such unions. This must also hold on the metaclass level, because metaclass are classes. Therefore, when composing metaclasses it is proper to use multiple inheritance on the metaclasses; in this context "proper" refers to the viewpoint of knowledge representation. Therefore, Figure 14 is the appropriate way to represent that Y is a secure and thread-safe dog class, where M is the metaclass whose instances are classes that are both secure and thread-safe.

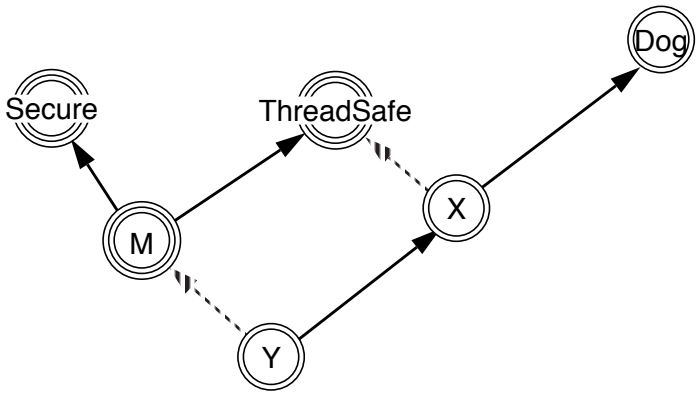

Figure 14. Explicit Composition of Metaclasses.

Not all metaclasses correspond to adjectives; some metaclasses are not meant to be instantiated (for example, SOMMBeforeAfter [6]). But those that impart properties to instances of ordinary classes are normally amenable to being named with adjectives. A minor issue arises here as to whether metaclasses should be named with adjectives alone or adjectives prepended to the word "Class." Using the word "Class" at the end of every metaclass does distinguish classes from metaclasses, but using adjectives alone is useful to indicate that the metaclass is intended to be instantiated.

Conversely, not all adjectives make good metaclasses. Inheritance of the metaclasses constraint implies that metaclasses should add properties rather than take them away. It is observed in [15] that if one uses a metaclass to implement the "must be abstract" class property (i.e., no class instances can exist), then inheritance of the metaclass constraint means that the property passes to all descendant classes.

Clearly, when metaclass constraints are inherited, there is very little utility to such a metaclass. But, in contrast, a metaclass that introduces the "can be abstract" class property *would* be useful. For example, in SOM, SOMClass introduces this capability because it is a well-known, potentially useful aspect of classes. As a result, all classes in SOM inherit the option of being abstract. The decision as to whether a given class will actually be abstract (i.e., refuse to construct instances) is then made by class designers, on a class by class basis.

So, we're not convinced by the objections raised in [15]. In our opinion, inheritance of the metaclass constraint and the use of multiple inheritance as the composition mechanism has such great value (when compared with the alternatives) that the lack of special support for metaclasses that subtract properties seems of little consequence. Certainly there are good examples of the utility of non-monotonic logics in knowledge representation (particularly as regards exceptions to inherit-

ance of properties) [16]. But our feeling is that support for non-monotonic capabilities is not as important as the ability to evolve an implementation.

Conclusion

In this paper, we have shown how SOM implicitly supports inheritance of metaclass constraints through use of derived metaclasses, and have explained why we believe this is the only reasonable approach in OOP models that support multiple inheritance and explicit metaclasses. Our examples focused on considerations that relate to practical requirements of software engineering in the real world, and also illustrated a strengthened relationship between OOP and knowledge representation.

Ultimately, we expect that the most important aspect of our approach may relate to the benefits offered by information hiding. The fewer things about ancestor classes that subclass designers need to know about and depend on the better. We prefer not to include metaclasses in the list of things that subclass programmers need to explicitly consider. As we have shown, this would be difficult enough in static situations, but real software is not static. It grows and evolves. As we work with the SOM class library, we are continually reminded how important it is to be able to make major changes without breaking clients. If reflective programming is to be useful in this context (and to us, it certainly seems to be), requiring subclasses to be explicitly modified when ancestor classes use different metaclasses is simply not a workable solution.

Acknowledgments

We thank Govind Balakrishnan, Michael Cheng, and Mike Heytens for comments on this paper.

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) 156-165.
3. Danforth, S.H. and I.R. Forman “Derived Metaclasses in SOM” *Proceedings of the 1994 Conference on Technology of Object-Oriented Languages and Systems* Versailles, France (April, 1994).
4. Danforth, S.H. and I.R. Forman “Reflections on Metaclass Programming in SOM” *Proceedings of OOPSLA’94*, Portland, Oregon (October 23-26, 1994).
5. Danforth, S., Koenen, P. and Tate, B. *Objects for OS/2* Van Nostrand Reinhold (1994).
6. Forman, I.R., S.H. Danforth, and H.H. Madduri “Composition of Before/After Metaclasses in SOM” *OOPSLA’94 Conference Proceedings*, Portland, Oregon (October 23-26, 1994).
7. Forman, I.R. and S.H. Danforth “Cooperation Among Metaclasses in SOM” *AIXpert* (August 1995) 4-29.
8. Forman, I.R., M.H. Conner, S.H. Danforth, and L.K. Raper “Release-to-Release Binary Compatibility in SOM” *OOPSLA’95 Conference Proceedings* Austin, Texas (October 16-19, 1995).
9. Goldberg, A. and D. Robson *Smalltalk-80 The Language and Its Implementation* Addison Wesley (1983).

10. Graube, N. “Metaclass Compatibility” *OOPSLA’89 Conference Proceedings* (1989).
11. Harary, Frank *Graph Theory* Addison-Wesley 1972.
12. Kiczales, G., des Rivieres, J., and Bobrow, D. G. *The Art of the Metaobject Protocol* The MIT Press, Cambridge, Massachusetts (1991).
13. Kiczales, G. and Paepcke, A. *Open Implementations and Metaobject Protocols* The MIT Press, Cambridge, Massachusetts (1994).
14. Klas, W., G. Fischer, and K. Aberer “Integrating Relational and Object-Oriented Database Systems using a Metaclass Concept” *Journal of Systems Integration* 4 (1994) 341-372.
15. Mulet, P., Malenfant, J., and Cointe, P. “Towards a Methodology for Explicit Composition of MetaObjects” *OOPSLA’95 Conference Proceedings* (October 16-19, 1995) 316-330.
16. Russinoff, D.M. “Proteus: A Frame-Based Nonmontonic Inference System” *Object-Oriented Concepts, Databases, and Applications* Kim, W. and Lochovsky, F.H. (ed.) ACM Press, New York (1989) 127-150.
17. Wand, M. and Friedman, D. “The Mystery of the Tower Revealed” *Proc. 1986 LISP and Functional Programming Conference*, 1986.