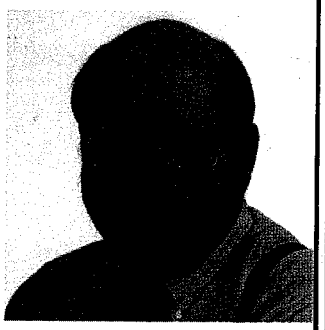
Metaclass and the Dogs of Shakespeare



BY ROGER SESSIONS

In my February column on polymorphism I explained why littleDogs go "woof woof" and bigDogs go "Woof Woof Woof Woof" (*OS/2 Magazine*, p. 46). Implicit in this discussion was that all dogs bark when asked, and the only difference between the types of dogs is the nature of that bark. I showed how the code:

```
_bark(Lassie, ev)
```

generates the bigDog bark ("Woof Woof Woof Woof"), while the code:

```
_bark(Toto, ev)
```

generates the littleDog bark ("woof woof").

However, I was recently reading Shakespeare, and it occurred to me that I made an error in that column. I believe in admitting my mistakes, so here is an attempt to set the record straight.

The line that made me rethink my analysis of littleDogs was this line, spoken by King Lear: "The little dogs and all, Tray, Blanch, and Sweet-heart, see, they bark at me." In order to see why this line caused me such vexation, we must translate it from Shakespearean English into a more familiar language, namely SOM with the C bindings. The SOM C translation of it is:

```
King KingLear;
littleDog Tray, Blanch,
    Sweetheart;
/* ... */
_enter(KingLear, ev);
_lookAt(Tray, ev);
_lookAt(Blanch, ev);
_lookAt(Sweetheart, ev);
```

resulting in the following output:

"The little dogs and all, Tray, Blanch, and Sweet-heart, see, they bark at me."

```
woof woof
woof woof
woof woof
```

The problem is that at no point does King Lear *ask* any of the littleDogs to bark! This fact is quite at odds with the code I showed in February, which would have permitted the littleDogs to bark only if their monarch so requested. With my polymorphic littleDogs, Shakespeare would have had to include these lines in his play to get his desired effect:

```
_bark(Tray, ev);
_bark(Blanch, ev);
_bark(Sweetheart, ev);
```

So, in this column I am going to reimplement littleDogs as they would have been programmed by the Bard himself.

Let's analyze the situation further. Just what did King Lear do to make the dogs bark? Perhaps he *looked* at the dogs? I believe Shakespeare's intention was that looking at the dogs was merely one example of what King Lear might have done to set off the barking fit. I submit that *anything* King Lear did to the dogs would have had the same result. They would have barked even if King Lear had asked them to roll over!

In other words, these dogs always bark, in addition to doing whatever it is they do. Barking, then, is not associated with the *implementation* of a method, but rather is associated with the *invocation* of a method. How do we program this in SOM?

The answer is to slip into a new mode of programming, one that Ira Forman describes as *metaclass programming*. Ira is one of the SOM developers and great champions of metaclass programming, which he describes as the next major advance in programming languages. As you can see, Ira, similar to all great champions, sometimes gets carried away, but he does raise some interesting issues. I, along with many others, am indebted to him for first explaining the concepts of metaclass programming.

Those of you who have heard Ira's talks will recognize my reimplemented barking dogs as an adaptation of Ira's growling dogs example (which, of course, is an adaptation of my original barking dog, so all's fair).

We can briefly describe metaclass programming as programming not at the object level, but at the class level. It turns out that we are still programming objects, but these are now very special objects—class objects. Lets look more closely at these class objects.

All SOM objects are associated with some class. Knowing that Toto, for example, is instantiated as a littleDog

tells us that the Toto object is associated with the littleDog class.

Every class that is available to a given SOM program has an associated object called the *class object*. If littleDog is derived from dog, and dog from SOMObject (the root of all SOM objects), then a program with instantiated littleDogs will have class objects in its address space for SOMObject, dog, and littleDog.

These class objects are similar to other SOM objects. They must be instantiated, they are associated with a class, they are defined by IDL, and they have associated methods. Many SOM programmers aren't aware of these class objects, because they are often automatically instantiated. When one executes the statement:

```
Toto = littleDogNew();
```

under the covers, SOM checks to make sure that class objects have been instantiated for littleDog and all of littleDog's base classes. If any class objects haven't already been instantiated, they will be instantiated as part of the execution of littleDogNew.

Several ways exist for getting hold of the class object of a given class. One of the most common is to use the SOM-provided macro _<class>. In the SOM-generated header file for any class, say, littleDog, is a macro of the form _<class>. For the littleDog class, this macro would look like **_littleDog**. This macro returns the class object. (In the littleDog case, the class object returned is the one associated with the littleDog class.)

As with all SOM objects, these class objects are derived ultimately from SOMObject. Therefore, it's safe to call any of the SOMObject methods on class objects. One of the SOMObject methods is **_getClassName**, a method that returns the class of an object. For example, if we invoke **_getClassName** on Toto, we'll have the string "littleDog" returned. If we invoke this method on **_littleDog**, we will, by default, get the string "SOMClass."

The default class of all class objects is "SOMClass." In fact, the only distinguishing characteristic of class objects is that their class is always either SOMClass or some class derived from SOMClass. As do all classes, SOMClass has a defining IDL with various method declarations. The SOMClass IDL can be found in the SOM-include directory.

Class objects have a lot of interesting behaviors. One of these behaviors is the logic controlling how methods are invoked on objects of their class. It can be modified by changing the implementation of the class object's class.

Listing 1: barkingClass definition.

```
#include <sombacls.idl>

interface barkingClass : SOMMBeforeAfter {
  implementation {
    override : sommBeforeMethod;
  };
};
```

Listing 2: barkingClass implementation.

```
#ifndef SOM_Module_dogcls_Source
#define SOM_Module_dogcls_Source
#endif
#define barkingClass_Class_Source

#include "dogcls.ih"
SOM_Scope boolean  SOMLINK sommBeforeMethod(
barkingClass somSelf,
Environment *ev,
SOMObject object,
somId methodId, va_list ap)
{
   barkingClassMethodDebug("barkingClass","sommBeforeMethod");
   somPrintf("woof woof\n");
   return
(barkingClass_parent_SOMMBeforeAfter_sommBeforeMethod(somSelf,
         ev, object, methodId, ap));
}
```

Listing 3: littleDog definition.

```
#include <somobj.idl>
#include <dogcls.idl>

interface littleDog : SOMObject {
  void lookAt();
  void rollOver();
  implementation {
    releaseorder: lookAt, rollOver;
    metaclass = barkingClass;
  };
};
```

Listing Guide

Listing 1 shows the definition of barkingClass. Notice we have included the SOM-provided file (*sombacls*), which defines barkingClass's base class, **SOMMBeforeAfter**. Although **SOMMBeforeAfter** defines both a **sommBeforeMethod** and a **sommAfterMethod**, we are only overriding the sommBeforeMethod.

Listing 2 shows the implementation of barkingClass. Most of this file was automatically generated by the SOM precompiler. The only thing we added was the **somPrintf** call which actually does the barking.

Listing 3 shows the definition of littleDog. This definition looks similar to a typical dog definition, except for the addition of the metaclass directive and the inclusion of the file defining the metaclass (*dogcls.idl*). Notice we define two methods, **lookAt** and **rollOver**.

Listing 4 shows the implementation of dog. It's an absolutely standard, run-of-the-mill dog implementation. Notice that in no place in the implementation of these methods is there any barking behavior.

Listing 5 shows the test program. In no place in the test program do we ask Toto to bark.

Listing 6 shows the output, with lines numbered for reference. The first of Toto's barks (line 1) comes as a result of an invisible method call—the **somInit**—resulting from Toto's instantiation. The second bark (line 4) comes because we looked at Toto. The third bark (line 6) comes because we asked Toto to roll over.

Listing 4: littleDog implementation.

```
#ifndef SOM_Module_dog_Source
#define SOM_Module_dog_Source
#endif
#define littleDog_Class_Source
#define littleDogClass_Class_Source

#include "dog.ih"


SOM_Scope void  SOMLINK lookAt(littleDog somSelf,  Environment *ev)
{
  somPrintf("Who is looking at me?\n");
}

SOM_Scope void  SOMLINK rollOver(littleDog somSelf,  Environment *ev)
{
  somPrintf("Watch me roll over.\n");
}
```

Listing 5: Test program.

```
#include <dog.h>
void main()
{
  littleDog toto;
  Environment *ev;

  toto = littleDogNew();
  somPrintf("Toto instantiated\n\n");

  _lookAt(toto, ev);
  _rollOver(toto, ev);
}
```

Listing 6: Program output.

```
1. woof woof
2. Toto instantiated
3.
4. woof woof
5. Who is looking at me?
6. woof woof
7. Watch me roll over.
```

You can see that the class of a class object is very important. Among other things, the class of a class object controls both method invocation and instantiation. Their invocation implementation is what we need to investigate to implement a Shakespearean version of littleDog.

When we start talking about class objects, language quickly gets in our way. For example, we might say that the invocation of Toto's methods is controlled by the class of Toto's class object, but who understands that? So instead, we shorten it by saying that the class of Toto's class object is defined to be Toto's *metaclass*. This explanation gives us a much easier statement to contemplate: The invocation of Toto's methods is controlled by Toto's metaclass.

To create our desired littleDog behavior, we need to modify the behavior defined by Toto's metaclass so as to tack in a little something else when invoking a method.

Unfortunately, overriding the method invocation behavior defined by SOMClass is beyond most people's programming ability. But fortunately, SOM has a class derived from SOMClass that provides exactly the hooks we need. This class is called **SOMMBeforeAfter**.

SOMMBeforeAfter is a SOM-provided class that defines two methods: **sommBeforeMethod** and **sommAfterMethod**. It also redefines the method invocation behavior to use the following pseudo-coded algorithm:

```
invoke(methodName,
       targetObject)
{
  _sommBeforeMethod
       (targetObject);
  _methodName(targetObject);
  _sommAfterMethod
       (targetObject);
}
```

In other words, objects whose metaclass is **SOMMBeforeAfter** automatically have the **_sommBeforeMethod** method invoked before every method invocation and the **_sommAfterMethod** method invoked after every method invocation. All we need to do now is stick our bark behavior into the **sommBeforeMethod** method. Then, our littleDogs will bark before any method invocation.

We accomplish this task by defining a new class, say, barkingClass, which is derived from **SOMMBeforeAfter** and overrides **sommBeforeMethod**. Our override implementation will include barking behavior.

Now we only have one conceptual problem left. We said that the default metaclass for all objects is SOMClass. How do we tell Toto that his metaclass is our newly defined barkingClass? By using a special directive in the dog IDL. This directive is the *metaclass directive*.

Let's summarize our discussion and then look at some sample code.

Every SOM object has an associated class. Every class has an associated class object. We can change the behavior of a whole class of objects by modifying the behavior of the class object. We do this modification by following these steps:

- Deriving a new class from either SOMClass or some class derived from SOMClass.
- Use the metaclass directive to tell the original class (for example, littleDog) that the new class is its metaclass.

As a specific example of this, we created a barking littleDog. We followed these specific steps:

- We derived a new class, barkingClass, from **SOMMBeforeAfter**, a SOM-provided class derived from SOMClass.
- We overrode the **SOMMBeforeAfter** method, **sommBeforeMethod** to add barking behavior.
- We used the metaclass directive to tell littleDog its metaclass is now barkingClass.
- This action changed the class of the littleDog class object to a barkingClass rather than a SOMClass, which it would have been by default. In other words, the metaclass of littleDogs is changed to barkingClass, rather than SOMClass.
- This change resulted in the barking behavior automatically being in-

voked before any method is called on a littleDog object.

Summary

Shakespeare would be very happy with this implementation of littleDog. No matter what King Lear tells these littleDogs to do, they are going to bark first.

We can "meta-program" regardless of which SOM language bindings we use or how we create our SOM objects. It works just as well with distributed objects as with local ones.

We have many possible uses of metaclass programming. We could use it to create a garbage collection scheme that keeps track of which objects are in use. We could use it to tie into a persistence framework that checks if object data needs to be read in from a disk before methods are invoked on the object. We could also use it as a basis for caching objects across address spaces.

Try it, you'll like it. Shakespeare would have. OS/2

References

For more information on metaclass programming, refer to these articles:

Ira R. Forman, Scott H. Danforth, and Hari H. Madduri, 1994. Composition of Before/After Metaclasses in SOM. *OOPSLA '94 Conference Proceedings*, October 23-27 Portland, Oregon.

Scott H. Danforth and Ira R. Forman, 1994. Reflections on Metaclass Programming in SOM. *OOPSLA '94 Conference Proceedings*, October 23-27, Portland, Oregon.

Roger Sessions is president of Object-Watch Inc., a company specializing in training and consulting in the use of CORBA technologies on IBM platforms. He has spoken at over 30 conferences and has written extensively. His books include Object Persistence: Beyond Object-Oriented Databases, Class Construction in C and C++; Object-Oriented Fundamentals, *and* Reusable Data Structures for C. *Roger also publishes an Internet newsletter called* ObjectWatch on SOM *and can be contacted via e-mail at roger@fc.net.*