**The World of Objects**



**Metaclass and the**

**Dogs of Shakespeare**

**BY ROGER SESSIONS**



n my February column on poly­ morphism I explained why little­ Dogs go "woof woof" and bigDogs

I

go "Woof Woof Woof Woof" *(OS/2*

*Magazine,* p. 46). Implicit in this discus­ sion 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 be­ lieve in admitting my mistakes, so here is an attempt to set the record straight.

The line that made me rethink my analysis oflittleDogs was this line, spo­ ken 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;

*I\** .•• *\*I*

\_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 iact 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 *invoca­ tion* of a method. How do we program thisinSOM?

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 YO)I who have heard Ira's talks will reco\_inize 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

**05/2 M A G A Z I N E** M A Y 1 9 9 6 51



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.* IflittleDog is derived from dog, and dog from SOMObject (the root of all SOM

objects), then a program with instanti­

ated 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



52 **OS/2 M A G A Z I N E** M A Y 1 9 9 6



**The World of Objects**

programmers aren't aware of these class objects, because they are often auto­ matically instantiated. When one exe­ cutes the statement:

Toto= LittleDogNew();

under the covers, SOM checks to make sure that class objects have been instan­ tiated 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, little­

Dog, is a macro of the form \_<class>. For the lit­ tleDog class, this macro would look like \_little­ Dog. This macro returns the class object. (In the lit-

tleDog case, the class .object returned is the one associated with the littleDog class.)

As with all SOM objects, these class objects are derived ulti­ mately from SOMObject. There­ fore, 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 "SQMClass."

The ,default class of all class objects is "SOMClass." In fact, the only distin­ guishing characteristic of class objects is that their class is always either SOM­

Class or some class derived from SOM­ Class. As do all classes, SOMClass has a defining IDL with various method dec­ larations. 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 imple­ mentation of the class object's class.

**The WoPid of Objects**



You can see that the class of a class object is very important. Among other things, the class of a class object con­ trols both method invocation and instantiation. Their invocation imple­ mentation is what we need to investi­ gate to implement a Shakespearean ver­ sion 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 con­ trolled by the class ofToto's class object, but who understands that? So instead, we shorten it by saying that the class of

54 **05/2 M A G A Z I N E** M A Y 1 9 9 6

Toto's class object is defined to be Toto's *metaclass.* This expla­ nation gives us a much easier statement to contemplate: The invocation of Toto's methods is controlled by Toto's metaclass.



To create our desired little­ Dog behavior, we need to modi­ fy 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 behav­ ior 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-pro­ vided class that defines two methods: sommBeforeMethod and sommAfter­ Method. It also redefines the method invocation behavior to use the follow­ ing pseudo-coded algorithm:

invokeCmethodName, targetObject)

{

sommBeforeMethod

CtargetObject);

\_methodName(targetObject);

sommAfterMethod

CtargetObject);

}

In other words, objects whose meta­ class is SOMMBeforeAfter automati­ cally have the \_sommBeforeMethod method invoked before every method

invocation and the \_sommAfter­ Method 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 defin­ ing a new class, say, barkingClass, which is derived from SOMM­ BeforeAfter and overrides sommBe­ foreMethod. Our override imple­ mentation will include barking behavior.



Now we only have one conceptu­ al 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:

Ill Deriving a new class from either SOMClass or some class derived from SOMClass.

Ill 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 cre­ ated a barking littleDog. We followed these specific steps:

Ill We derived a new class, barking­ Class, from SOMMBeforeAfter, a SOM-provided class derived from SOMClass.

Ill We overrode the SOMMBeforeAfter

method, sommBeforeMethod to add barking behavior.

Ill We used the metaclass directive to telllittleDog its metaclass is now barkingClass.

Ill This action changed the class of the littleDog class object to a barking­ Class rather than a SOMClass, which it would have been by default. In other words, the metaclass of little­ Dogs is changed to barkingClass, rather than SOMClass.

Ill This change resulted in the barking behavior automatically being in-

The World of Objects

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 lit­ tleDogs 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

**Lock and Load!**

*Create, Deliver and Secure* OS/2 *Desktops over any LAN!* •



• Store/Manage Desktops Centrally! Users get their

desktop whenever and wherever log on!

• Works with any LAN!

• Easily associate Desktops with User ID's!

• Avoid difficult REXX maintenance or INI nightmares!

• Authenticate users Desktops with your LAN Security!

Desktop

Commander Desktop

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 persis­ tence 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. m:Jil

A *Complete* Solution To OS/2 *Desktop* Control

• Easily take a picture of uses desktops and store it centrally!

• Standardize any group of workstations or allow

users to have their own Desktop wherever they log in!

• Restrict right mouse button options!

• Restore lost or changed desktops instantly!

• Security upgrade available!

*A {\ .*

Observatory

A *Complete* Solution To OS/2 *Desktop Security*

• Same benefits of the Desktop Commander and more!

• Password protect objects and applications...even the

Launch Pad!

• Take background tasks off the Window List!

• Create Security/Audit Logs!

• Drag and Drop File Encryption!

• Prevent Cnti-Break and Alt-F1 access!

• Inhibit unauthorized file access!

• Prevent clever users from building unauthorized objects!

Information 800.525.1650.

References

For more information on metaclass pro­

gramming, 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.

© 1995 Pinnacle Technology, Inc. • PO Box 128, Kirklin, IN 46050 .• 317.279.5157

OS/2,OS/2 Ready!,OS/2 WARP and Ready for OS/2 WARP are trademarks of the IBM Corporation. © 1995 Pinnacle Technology, Inc. All Rights Reserved.

Reoder Service No. 28

-60,000 Power Users and Corporate Buyers

,.,.,

*Roger Sessions is president of Object­ Watch Inc., a company specializing in training and consulting in the use of COREA 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 Construc­ tion inC 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@(c.net.*



MARKET PLAUE

•OS/2, Windows, DOS Software

•High-end Computer Systems

•Powerful Per)pherals

•Memory, B9'ard and Upgrade Products

•Storage & Backup Solutions

•Networking Products & Services

•Enterprise Computing

•Multimedia

•Services & Consulting

Advertise in the OS/2 Magazine

OS/2 MARKETPLACE

Calltoday to place an ad!

Gordon Peery at (603) 924-9141 [gpeery@ mfi.com](mailto:gpeery@mfi.com)

05/2 M A G A Z I N E M A Y 1 9 9 6 55