**The World of Objects**

**littleDogs, Polymorphism, and Frameworks**

**BY ROGER SESSIONS**

he three pillars of object-ori­ ented programming are en­ capsulation, inheritance,

T

and polymorphism. Encapsulation and inheritance are well understood by most object-oriented program­ mers. However, many object-oriented programmers, even some with years of experience, are confused about poly­ morphism. Many do not even agree as to the meaning of the word.



What is especially interesting about this general state of confusion is that, of these three pillars, polymor­ phism is by far the most important. Encapsulation is nice, but hardly a major advance in the state of the art. Inheritance allows new functionality to be added to existing classes, but this is

rarely useful in real life. Polymorphism,

on the other hand, is the enabling tech­ nology for frameworks. Frameworks are one of the most important advances in code reusability since the



invention of procedures.

Whoever coined the word "polymorphism" made a bad choice. The reason most pro­ grammers do not understand polymorphism is because the WQrd itself is so intimidating. *It's* easy to relate to encapsu­ lation and inheritance, terms for which we have intuitive feelings. But who can relate to polymorphism?



In this column, let's

expore the concept of poly­ morphism. We could choose any number of object-orient­ ed programming languages for the discussion, becauseall support this concept, but let's use IBM's SOM technology with the C language bindings.

SOMis IBM's foundational object­ oriented strategy. It has long been asso­ ciated with OS/2, but is now becoming available on other IBM platforms such as MVS and AS400. I like SOM because it is language neutral. I can do my actu­ al programming in many different lan-

guages. SOM has other advantages, which I will discuss in future columns.

Everything you need to know·



about polymorphism can be summa­ rized in four words. If you can remem­ ber these four words, you will under­ stand polymorphism:

LittleDogs go woof woof

Let's see what this means. The word polymorphism comes from two Greek words, *polys,* meaning many, and *morpho,* meaning form. *It's* often used as an adjective, such as polymor­ phic method. It means that a class can have many forms of a given method, and the object run time can decide which will be used in a given situation. Orfali, Harkey, and Edwards, in their new book *The Essential Distributed Objects Survival Guide,* say "Polymor­ phism is a high-brow way of saying that the same method can do dif­

ferent things, depending on the class that implements it." Let's look at a very simple SOM class. Listing 1 shows a definition of a dog. Our dog is derived from SOMObject,

/because in SOM all objects are derived, directly or indirectly, from SOMObject. Our dog has two associated methods, print and bark. A release order is included to ensure upward binary compatibility. An **oidl** callstyle is used to avoid pass­ ing around an environment variable, which is a good pro­ gramming style, but superflu­

ous to this discussion.

Listing 2 shows the dog's

implementation. Most of the lines were generated by the



**46 OS/2 M A G A *Z* I N E** F E B R U A R Y 1 9 9 6



**The WoPid of Objects**



SOM precompiler. Two lines were added inside the print method and one line inside the bark method. The imple­ mentation of the bark method includes a standard SOM print statement and an invocation of the bark method. When invoking a SOM method in C, we pass the target object as the first parameter. The target object of the bark method is the same as the target object of the print method, thus we just pass through sam­ Self as the target parameter.

Notice that the invocation of the bark method inside pririt uses the form:

\_bark(somSelf)

I

I·

instead of the form in which the code is actually written:

barkCsomSelf)

I The underscore in front of the method

I, name shows how, in the SOM C bind­

ings, we invoke a method rather than a procedure.

When the print method is invoked, and it in turn invokes bark, what code will be called? The most logical guess would be the bark method that is imple­ mented in the same file. This guess

II appears to be validated by the client pro­

'I

gram (Listing 3).The program in Listing

3 instantiates a dog named Snoopie and asks him to print himself. The output that this program generates is:

My noise is generii dog noise

It's crystal clear that dog's print invokes dog's bark. The output is as expected, and no other candidate bark method exists.

Now let's slightly complicate the sit­ uation. Listing 4 defines bigDog. big­ Dog is derived from dog. It adds no new

**48 05/2 M A G A Z I N E** n B R U A R Y 1 9 9 6

methods, but overrides one of the dog methods: bark. Overriding means that it redefines what it means to bark for bigDog.The bigDog implementation of bark is shown in Listing S. You can see from the print statement that bigDogs bark quite differently than dogs.

Listing 6 defines yet another dog, a littleDog, also derived from dog, and also redefining the bark method. Its implementation is shown in Listing 7.



Now we have three versions of bark, one for dog, littleDog, and bigDog. How will the system sort this out?

In Listing 8 a client program that instantiates Snoopie (a dog), Toto (alittleDog), and Lassie (a big­ Dog) is shown. It invokes print on each of these ob­ jects. Looking back on Listing 2, inside the imple­ ment tion of dog's print we see a simple invo­ cation of bark. No visible branch



code exists. The invocation of bark is unconditional and unequivocal.

So what output do we expect from the program in Listing 8? With stan­ dard C procedural calls, we would expect the same version of bark to be invoked in each case. The most likely output from this program would be:

My noise is generic dog noise My noise is generic dog noise My noise is generic dog nbise

One of these lines would come from each of the three dog print invocations, using the target objects Snoopie, Toto, and Lassie, respectively. The actual out­ put this action generates is quite differ­ ent. It is:

My noise is generic dog noise

My noise is woof woof

My noise is WOOF WOOF WOOF WOOF

It's clear that the dog's print behaves very differently when invoked on each dog. When invoked on Snoopie, it invokes the dog's bark. When invoked on Toto, it invokes the littleDog's bark. When invoked on Lassie, it invokes the bigDog's bark. This response is true even though the exact same print (dog's) is invoked in all three cases. We know it's the same print method in all three cases because only one print method is defined (and implement­ ed)-dog's. This ability of print to auto­ matically route to different bark imple­ mentations based on the type of target object is called polymorphism.

The contrast between polymorphic resolution and standard procedural res­ olution is seen by changing a single

**The World of Objects**

character in our program. By removing the underscore in front of the bark invocation within the print implementation in Listing 2,



the same client program gives a very different result.

My noise is generic dog noise

My noise is generic dog

noise

My noise is generic dog noise

Using the underscore in front of bark tells SOM to use polymorphic method resolu­ tion. By removing the under­ score, SOM assumes we are making a standard procedure

call. Standard procedure resolution says that all calls to a given procedure name route to the same code location. In this case, that location is the dog's bark.

Obviously polymorphic resolution must be accomplished at run time, not compile time. At compile time, print only knows about dogs. It has no way to know that it will someday be invoked on a littleDog. In fact, it doesn't even know that such things as littleDogs exist. Only at run time can we look at the actual type of a target object and route accordingly.

Without polymorphic resolution, Toto would be forever constrained to making a "generic dog noise." It is

only through the run-time magic of

polymorphic resolution that Toto can say "woof woof." That's all

you need to know about polymorphism.

**Frameworks**

What does all this stuff have to do with frame­ works? Frameworks are architectural contexts with- · in which objects interact. Orfali, Harkey, and Edwards write, "a framework pro­ vides an organized environ­ ment for running a collec­ tion of objects."

Frameworks are impor­ tant because they offer massive opportunities for code reuse. They take advantage of the fact that

nearly all of the interactions between objects can be defined generically.

However, in order for a framework to be useful, it must be extensible.

Extensibility means working with a wide range of objects, including objects

that the framework doesn't

even know about. How can a framework function with unknown types of objects? The answer is through the use of polymorphic method resolution. The framework says, "I can work with any type of object, as long as it supports these methods." The framework assumes all objects are derived from some framework-provided base type, and each object type overrides the methods

it needs to special ze.

The program in Listing 8 can be

thought of as a very simple framework; one that coordinates the bark activity of various dogs. You can add any dog type you want, as long as that type is derived directly or indirectly from dog and overrides the bark method.

An example of a more serious frame­ work might be a GUI framework that allows objects to be dragged, resized, or hidden. The framework might provide a base type called graphicObject. The graphicObject could support a re­ paintYourself, which takes as parame­ ters a screen area in which to repaint.

This framework might not have any idea what graphical objects will even­

tually be created. All it knows is that

whatever those types are, they will be derived from graphicOb­ ject and will support the repaintYourself method.

Users of this GUI frame­ work now have great tech­ I).Ology for creating graphi­

,cal objects. Suddenly they can have dogs, littleDogs, bigDogs, and animals of all types that support the very complex algorithms of drag, resize, and hide­ courtesy of the GUI frame­ work. All the objects have to know is how to repaint themselves.

Let's look at another

example. Consider a phone company that needs to write a program to coordi-

**05/2 M A G A Z I N E** F E B R U A R Y 1 9 9 6 49

**The World of Objects**



nate the activity of many different types of telephones. The company needs this program to be highly modifiable. They know the phones they currently sup­ port, but not what new phone services will be offered in the future. They need to be easily flexible to be competitive.

A framework is an ideal mechanism for this compa­ ny. The company defines the basic expectations of a phone type (Listing 9).It then writes a framework that defines how phone objects will interact. The framework code, which defines one phone calling another, for example, might look similar to the example in Listing 10. This code depends on all phones supporting get­ NumberToCall, but does not depend on how those phones provide such support.

A generic phone type might support the getNum­ berToCall method by read­ ing digits pressed on a touch pad. A new service, which supports speed dialing,



might override this method and define a version that looks up numbers in a local database based on a two-digit key. Yet another system might use voice recognition technology to determine the calling number. The important point, from the framework's perspec-

tive, is not how getNumberToCall will be implemented, but only that it will be implemented as an override to the base method getNumberToCall, thus enabling polymorphic method resolu­ tion.

Framework technology is exciting.It

provides a highly adaptable mechanism for developing and reusing code. Poly­ morphism is basic to frameworks. The same mechanism that allows Toto to say "woof woof" also allows a million-line framework, such as Taligent, to coordi­ nate the activity of thousands of differ­ ent types of objects. , [!1W

*Roger Sessions* is *presidentofObjectWatch Inc. a company specializing* in *training and consulting* in *the use ofSOM, DSOM, and related* 00 *technologies. He has spo­ ken at over 30 conferences and has written extensively.*His *books include* Object Per­ sistence: Beyond Object-Oriented Data­ bases, Class Construction inC and *C++i* Object-Oriented Fundamentals, *and* Reusable Data Structures for C. *Roger also publishes an Internet Newsletter called* ObjectWatch on SOM, *and can be con­ tacted via e-mail at roger@{c.net.*

Deployment Insurance For Your Most Critical Applications



Concerned about the reliability of your client;server application? The Automated Test Facility delivers the kinds of testing today's mission-critical applications need. And that includes business workflow testing, which is vital for key information systems where there are user interdependencies. Only ATF, with its unique architecture, can simulate user behavior and test for these interdependencies so that your system is

J

! I tested under actual production conditions.



! Think of ATF as deployment insurance for

:I

I' your most critical applications.

ATF tests applications under Windows, Windows NT, Windows 95 and all flavors of OS/2.



'I '

Reader Service No. 21

50 OS/2 M A G A Z IN E F E B R U A R Y 1 9 9 6

For information about ATF contact: Softbridge, Inc.

125 CambridgePark Drive

Cambridge, MA 02140

Phone: (617) 576-2257

Fax: (617) 864-7747 [market@sbridge.com](mailto:market@sbridge.com)

Reader Service No. 22