littleDogs, Polymorphism, and Frameworks

BY ROGER SESSIONS

The three pillars of object-oriented programming are encapsulation, inheritance, and polymorphism. Encapsulation and inheritance are well understood by most object-oriented programmers. However, many object-oriented programmers, even some with years of experience, are confused about polymorphism. 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, polymorphism 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 technology 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 programmers do not understand polymorphism is because the word itself is so intimidating. It's easy to relate to encapsulation and inheritance, terms for which we have intuitive feelings. But who can relate to polymorphism?

In this column, let's expore the concept of polymorphism. We could choose any number of object-oriented programming languages for the discussion, because all support this concept, but let's use IBM's SOM technology with the C language bindings.

Listing 1: Dog definition.

```
#include <somobj.idl>
interface dog : SOMObject {
 void print();
 void bark();
 implementation {
  releaseorder: print, bark;
  callstyle = oidl;
 };
};
```

SOM is IBM's foundational object-oriented strategy. It has long been associated 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 actual programming in many different languages. SOM has other advantages, which I will discuss in future columns.

Everything you need to know about polymorphism can be summarized in four words. If you can remember these four words, you will understand 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 polymorphic 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 "Polymorphism is a high-brow way of saying that the same method can do different 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 passing around an environment variable, which is a good programming style, but superfluous to this discussion.

Listing 2 shows the dog's implementation. Most of the lines were generated by the

Listing 2: Dog implementation.

```
#ifndef SOM_Module_dog_Source
#define SOM_Module_dog_Source
#endif
#define dog_Class_Source
#include "dog.ih"
SOM_Scope void SOMLINK print(dog somSelf)
{
  somPrintf("My noise is ");
  _bark(somSelf);
}
SOM_Scope void SOMLINK bark(dog somSelf)
{
  somPrintf("generic dog noise\n");
}
```

Listing 3: Dog client.

```
#include <dog.h>
main()
{
  dog Snoopie;
  Snoopie = dogNew();
  _print(Snoopie);
}
```

SOM precompiler. Two lines were added inside the print method and one line inside the bark method. The implementation 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 **somSelf** as the target parameter.

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

```
_bark(somSelf)
```

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

```
bark(somSelf)
```

The underscore in front of the method name shows how, in the SOM C bindings, 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 implemented in the same file. This guess appears to be validated by the client program (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 generic 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 situation. Listing 4 defines bigDog. bigDog is derived from dog. It adds no new 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 5. 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.

Listing 4: bigDog definition.

```
#include <dog.idl>
interface bigDog : dog {
implementation {
  override: bark;
};
};
```

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 (a littleDog), and Lassie (a bigDog) is shown. It invokes print on each of these objects. Looking back on Listing 2, inside the implementation of dog's print we see a simple invocation 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 standard 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 noise
```

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 output this action generates is quite different. 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 implemented)—dog's. This ability of print to automatically route to different bark implementations based on the type of target object is called polymorphism.

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

Listing 5: bigDog implementation.

```
#ifndef SOM_Module_bdog_Source
#define SOM_Module_bdog_Source
#endif
#define bigDog_Class_Source
#include "bdog.ih"
SOM_Scope void SOMLINK bark(bigDog somSelf)
{
  somPrintf("WOOF WOOF WOOF WOOF");
}
```

Listing 6: littleDog definition.

```
#include <dog.idl>
interface littleDog : dog {
implementation {
  override: bark;
};
};
```

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 resolution. By removing the underscore, 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 frameworks? Frameworks are architectural contexts within which objects interact. Orfali, Harkey, and Edwards write, "a framework provides an organized environment for running a collection of objects."

Frameworks are important 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 specialize.

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 framework 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 **repaintYourself**, which takes as parameters a screen area in which to repaint.

This framework might not have any idea what graphical objects will eventually be created. All it knows is that whatever those types are, they will be derived from **graphicObject** and will support the **repaintYourself** method.

Users of this GUI framework now have great technology for creating graphical 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 framework. 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-

Listing 7: littleDog implementation.

```
#ifndef SOM_Module_ldog_Source
#define SOM_Module_ldog_Source
#endif
#define littleDog_Class_Source
#include "ldog.ih"
SOM_Scope void SOMLINK bark(littleDog somSelf)
{
  somPrintf("woof woof\n");
}
```

Listing 8. littleDog and bigDog client.

```
#include <dog.h>
#include <bdog.h>
#include <ldog.h>
main()
{
  dog Snoopie;
  littleDog Toto;
  bigDog Lassie;
  Snoopie = dogNew();
  Toto = littleDogNew();
  Lassie = bigDogNew();
  _print(Snoopie);
  _print(Toto);
  _print(Lassie);
}
```

Listing 9: Phone definition.

```
#include <somobj.h>
interface phone : SOMObject {
void determinePhoneNumber();
string getNumberToCall();
void dial(in string number);
boolean connectionAccepted(in string number);
void transmit();
boolean connectionRequested();
implementation {
/* ... */
};
};
```

Listing 10: Calling code.

```
number =
_getNumberToCall(myPhone);
_dial(myPhone, number);
if (_connectionAccepted(myPhone,
number)) {
  _transmit(myPhone);
```

Listing 11: Answering code.

```
if(_connectionRequested(myPhone))
{
  _transmit(myPhone);
}
```

nate the activity of many different types of telephones. The company needs this program to be highly modifiable. They know the phones they currently support, 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 company. 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 **getNumberToCall**, but does not depend on how those phones provide such support.

A generic phone type might support the **getNumberToCall** method by reading 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 perspective, 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 resolution.

Framework technology is exciting. It provides a highly adaptable mechanism for developing and reusing code. Polymorphism is basic to frameworks. The same mechanism that allows Toto to say "woof woof" also allows a million-line framework, such as Taligent, to coordinate the activity of thousands of different types of objects. OS/2

Roger Sessions is president of ObjectWatch Inc. a company specializing in training and consulting in the use of SOM, DSOM, and related OO technologies. 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.*