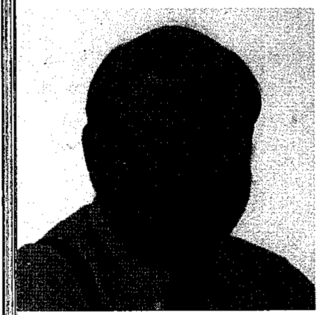
OS/2 Developer

The World of Objects



Encapsulation and the EPIC Nature of Dogs

BY ROGER SESSIONS

If we know that Lassie is a dog, then we know something about Lassie. If we know what dogs in general do and don't do, then we know what Lassie will and won't do. If we know that dogs bark when we ring the door bell, then we know that Lassie will bark when we ring the door bell. If we know dogs can't fly, then we know Lassie can't fly.

But knowing that Lassie is a dog doesn't tell us everything about Lassie. It tells us what she does and under what circumstances, but not how. Unless we are an expert on the implementation of dogs, we have no idea how Lassie actually goes about the business of barking. We don't know how the millions of neurons in her brain fire in the correct sequence, what muscles are involved, or how she makes her vocal chords vibrate. All we know is that given an appropriate stimulus, Lassie goes "Woof Woof."

In fact, we can't even prove that Lassie is really doing the barking. For all we know, Lassie is using a tape recorder. Or Lassie might be nothing more than an elaborate puppet and her bark coming from a good ventriloquist. Or maybe Lassie calls in a barking specialist whenever she hears the doorbell, and Lassie acts as nothing more than a noise broker.

It is said that the truth shall set us free. In this case at least, it is our ignorance that sets us free. Or more precisely, sets Lassie free. The less we know about Lassie, the more freedom Lassie has in her implementation. As long as she can figure out some way to make a whole bunch of noise when the doorbell rings, we will be happy, feed her, and not ask silly questions.

In object-oriented programming, we call this general principal *encapsulation*. Encapsulation says that we ask objects only what they do, not how they do it. An object, say Lassie, is encapsulated if its interactions with its client are determined only by its interface and not by its implementation.

Comparison of three languages

Lets look at the concept of interface from the perspective of three different programming languages, all supported on OS/2. These languages are C++, SOM's IDL (Interface Definition Language), and good old reliable C.

We describe objects by their interfaces. Loosely speaking, we can think of an interface as describing the behaviors or the methods that a class supports.

Our dog class will support two methods, a **setBark** method, which is used to tell the dog what its bark is, and a **bark** method, which is used to tell the dog that the time has come to bark. Since Lassie is a dog, we know she will support these methods.

In C++, using for example IBM's VisualAge C++ on OS/2, we could describe our dog interface as:

```
class dog {
 public:
   void setBark(char *newBark);
   void bark();
 private:
   char *myBark;
};
```

C++ breaks a class definition into two sections, a public and a private section. The public one is the class's interface. The private section is intended to be of interest only to the programmer implementing the interface and can change without warning.

IDL (Interface Definition Language) is defined by the Object Management Group (OMG). It's unique in that the choice of the language used in the implementation of the class is not constrained by the language used to define the interface. In contrast to IDL, if we define our interface in C++, we can use only C++ for implementing our class. If we define our interface in IDL, we can use C++, C, COBOL, or any other language that supports IDL.

IBM considers IDL strategic and is supporting it, or planning on supporting it, with all of its languages on all of its platforms. The IBM implementation of IDL is called SOM, for the System Object Model. SOM has been available on OS/2 for a number of years now.

In SOM IDL we can describe our dog interface like this:

```
#include <somobj.idl>
interface dog : SOMObject {
 void setBark(
   in char *newBark);
 void bark();
 implementation {
   callstyle = "oidl";
 };
};
```

This dog interface contains a great deal of information, more than one might expect by looking at this deceptively simple definition. However, we are only interested in encapsulation and from this perspective the IDL dog interface defines the same two methods as the C++ definition, **setBark** and **bark**.

This IDL definition, similar to its C++ counterpart, tells us that the **setBark** method takes a single string parameter and that the **bark** method returns a string. Like C++, it doesn't tell us how the dog stores that string. Extending the idea of encapsulation beyond C++, the IDL definition doesn't tell us in what language the code implementing **setBark** and **bark** is written. Also, IDL does not support the concept of public and private sections. If it's not part of the public interface, it shouldn't be part of the IDL. At least, it shouldn't be part of the IDL you allow clients to see.

A client that understands the contract implied by the dog definition can write code against that contract. One example of C++ client code using this dog looks like the following:

```
#include "dog.hpp"
int main()
{
 dog *Lassie;
 Lassie = new dog;
 Lassie->setBark(
   "Woof Woof");
 Lassie->bark();
}
```

On OS/2, this C++ source code is compatible with a dog defined in IDL and implemented in either C++ or C. It is also compatible with a dog both defined and implemented in C++.

A C client can also use this dog if it is defined in IDL. This client looks like:

```
#include <dog.h>

int main()
{
 dog Lassie;
 Lassie = dogNew();
 _setBark(Lassie,
   "Woof Woof");
 _bark(Lassie);
 return(0);
}
```

This C code is compatible with a dog defined in IDL and implemented in either C or C++. It is not compatible with a dog defined and implemented in C++, because C++ does not support the use of its classes by other languages.

We can even define and implement our encapsulated dog completely in C. An equivalent C definition for a dog is:

```
struct dogType {
 char *myBark;
};
typedef struct dogType *dog;

void _setBark(dog thisDog,
char *newBark);
void _bark(dog thisDog);
dog dogNew(void);
```

and a C implementation of this dog is:

```
#include "dog.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void _setBark(
 dog thisDog,
 char *newBark)
{
 thisDog->myBark =
  (char *) malloc(strlen
  (newBark)+1);
 strcpy(thisDog->myBark,
   newBark);
}
void _bark(dog thisDog)
{
 printf("%s\n",
   thisDog->myBark);
}
dog dogNew()
{
 return (dog)
   malloc(sizeof(dog));
}
```

This C implementation is compatible with the C client that we looked at earlier, even though that client was written to use an IDL implementation.

We say that these objects are all encapsulated, because the client code using these objects has no dependencies on the objects' implementations. All of the client interactions with these objects are defined by the objects' interface.

Writing objects that are nonencapsulated

Just as we saw that we can write encapsulated objects in nonobject-oriented languages (such as C), we can also write poorly encapsulated objects in state-of-the-art object-oriented languages.

For example, we know our dog has to store its bark string. A nonencapsulated implementation could store the bark string in global memory allocated by the client. In order to use this dog, the client must allocate a buffer, declare a global variable with a particular name, and set that variable to the previously allocated buffer.

Because this implementation of dog requires the client to know quite a bit about how the dog manages its string storage (information that is not part of the interface), we say this dog implementation is not encapsulated. And we can write this sorry code in SOM, C++, or C.

EPIC objects

Encapsulated objects have what I call EPIC characteristics. EPIC stands for Exchangeable, Protectable, Isolatable, and Confidential. These characteristics are so important that even products like Microsoft's OLE, which rejects the other key ideas of object-oriented programming, accepts the importance of encapsulation.

Lets consider each of these in turn.

Exchangeable

The E in EPIC stands for Exchangeable. Different implementations of well-encapsulated objects can be exchanged for each other without impacting their clients.

Lets consider two possible SOM implementations of our IDL dogs, both in C. The first stores the bark string in an internal character buffer as shown in Listing 1.

The second implementation stores the bark string in a file as shown in Listing 2.

Which is the right implementation? Both. Either works fine for our immediate needs. Because the dog is a well-encapsulated object, these two implementations can be exchanged for each other without requiring source code changes to our client. In fact, by using SOM on OS/2, these changes can even be made at run time by a simple DLL replacement.

Our nonencapsulated dog, the one using the client-allocated global buffer, is not exchangeable with these two encapsulated versions. Without the client changing its source to allocate that buffer, that dog will not bark.

Exchangeability gives the object implementor considerable flexibility. For example, it's a common practice to prototype interfaces with a simple implementation and then add more robust, better performing, or less limited code later in the development cycle.

Exchangeability also gives flexibility to the client, who can write code with one implementation of dog and then find, write, or purchase a better implementation at any time.

Protectable

The P in EPIC stands for Protectable. Encapsulated objects are protected from odd behavior on the part of their clients.

With encapsulated objects, clients interact with objects only through approved methods. These methods can be guarded with code that checks and rejects invocations that would otherwise cause catastrophic failure, such as calling the dog's **setBark** with a string that contains unprintable characters.

Code using protected objects is very robust. It is almost impossible to break a well-protected object.

Nonencapsulated objects do not support this level of protection. For example, it would be very easy for the client of the nonencapsulated dog to place corrupted values in the global memory used by the dog object. The next time that dog tries to bark, watch out!

Isolatable

The I in EPIC stands for Isolatable. Encapsulated objects can be written, tested, and debugged in full isolation of the code that will be eventually using them.

This ability to isolate the object implementation from other development activity is a great advantage. It offers a natural mechanism for dividing large projects into a series of small well-defined subprojects, all of which can be worked on in parallel.

The goal of isolatability was shared by another historic programming methodology, structured programming. Encapsulation, though, goes much further than structured programming ever did. Structured programming only offered techniques for partitioning a program's logic. It offered nothing for dealing with the much more complex issue of a program's data. Encapsulation shows us how to partition both logic and data.

Confidential

The C in EPIC stands for Confidential. Encapsulated objects can keep their secrets.

This issue can be significant for classes that will be marketed. When we sell classes, we want to ship nothing but binary object files in the form of libraries and DLLs and interfaces definitions in the form of text files. We don't want to sell our source code. We consider our source code proprietary.

The names of private methods and the design of our data structures can also give important clues to the nature of our algorithms. By showing our clients nothing but the public interface, we can keep our algorithms confidential.

Of the three languages we have discussed, only SOM's IDL has first-rate support for confidentiality. C++ and C both require that private algorithm-revealing information be openly displayed along with the interface.

C++ actually has an even more serious problem in this area. C++ has very limited support for shipping class code in anything other than source code. This limited support is because of, among other reasons, a general lack of agreement among C++ compiler ven-

Listing 1: Dog with memory buffer.

```
#include "dog.ih"
#include <stdlib.h>
#include <stdio.h>

SOM_Scope void SOMLINK setBark(
dog     somSelf,
char*    newBark)
{
  dogData *somThis = dogGetData(somSelf);

  _myBark = (string)
   SOMMalloc(strlen(newBark)+1);
  strcpy(_myBark, newBark);

}
SOM_Scope void SOMLINK bark(
dog somSelf)
{
  dogData *somThis = dogGetData(somSelf);
  printf("%s\n", _myBark);
}
```

Listing 2: Dog with file.

```
#include "dog.ih"
#include <stdlib.h>
#include <stdio.h>

#define dogFile "dog.dat"

SOM_Scope void SOMLINK setBark(
dog     somSelf,
char*    newBark)
{
  FILE *f;
  f = fopen(dogFile, "w");
  fprintf(f, "%s", newBark);
  fclose(f);
}
SOM_Scope void SOMLINK bark(
dog somSelf)
{
  FILE *f;
  int n = '\0';
  char buff[100];

  f = fopen(dogFile, "r");
  for (n=0; n!='\n'; n++) {
    fscanf(f, "%c", &buff[n]);
  }
  buff[n-1] = '\0';
  fclose(f);

  printf("%s\n", buff);
}
```

dors as to the nature of the run-time object model. In this regard, C is actually more advanced than C++.

Conclusion

We have looked at three different OS/2 technologies, C++, SOM, and C. We have seen that all of these technologies support encapsulation. Encapsulation is important because it allows the development of objects that are Exchangeable, Protectable, Isolatable, and Confidential. These are the EPIC characteristics of encapsulated objects.

The best, but not the only, support for encapsulation comes from object-oriented technology, and this is one of the factors driving its rapid adoption. All of these OS/2 technologies, both object-oriented and nonobject-oriented have good support for Exchangeability, Protectability, and Isolatability. Confidentiality is mainly an issue for companies selling class libraries. In this area, SOM offers the best support, followed by C, followed by C++.

Remember, that we can get poorly written nonencapsulated code using both C++ and SOM. We can also get well-written highly encapsulated code using C. This is as much a programmer issue as it is a technology issue.

To keep all these options in perspective, just keep the following in mind. We want Lassie to be a good dog. The better encapsulated we make her, the more EPIC her character will be.

News from the object front

At the end of 1995, IBM made an important announcement. They have decided to license the Java technology for use in OS/2. Java is a new language which allows compiled objects to run on any hardware or software platform.

This technology could work well with SOM. SOM allows methods to be invoked on objects living on other machines, but has no way of downloading the code necessary to implement those methods. Java offers the ability to download an object's implementation to any machine, but no way of allowing methods to be remotely invoked on that downloaded object.

These two technologies offer a perfect complement to each other. However, the current information from IBM is sketchy. No information is available from the IBM announcements or on the IBM World Wide Web pages, which claim to give the most recent Java information, to indicate that IBM understands the important relationship between Java and SOM. My friends within IBM assure me that this relationship is being investigated. I will be watching this topic very carefully and reporting on developments as they unfold.

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