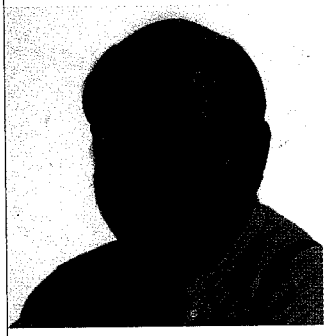
Dynamic Frameworks



BY ROGER SESSIONS

I recently went to Edinburgh to teach a class in SOM (System Object Model). My client had asked me to focus the class on the nondistributed aspects of SOM.

I must admit, I was at first perplexed by this request. I think of object distribution as the *raison d'etre* for SOM. What, I wondered, could my client's interest in SOM be, if not object distribution? Language neutrality? Release-to-release binary compatibility?

It turned out my client was interested in all of these features. The group was even interested in object distribution. But the single greatest reason for using SOM was dynamic class loading.

Dynamic vs. nondynamic class loading

I have used dynamic class loading, but mostly for building generic DSOM servers. The class in Edinburgh gave me the opportunity to think about using this feature to develop truly dynamic frameworks.

Dynamic class loading allows programs to make use of classes that are unknown at compile and build time. Dynamic class loading allows arbitrary classes to be loaded at run time on an as-needed basis.

Let's consider a program that instantiates a dog object as defined by **dog.idl** shown in Listing 1. This program could be written in several ways, one of which is shown as version 1 of **test.c** (Listing 2). Let's briefly review the purpose of the more important lines of **test.c**.

Line 7 declares an object named **classObject** of type **SOMClass**. **SOMClass** is the default type of a class object. Class objects know, among other things, how to instantiate objects of a given class (dog, in this case). I discussed class objects in more detail back in my May 1996 column ("Metaclass and the Dogs of Shakespeare," pp. 51-55).

Line 11 initializes the dog class. It does many things, among which is the instantiation of the dog class object.

Line 12 assigns the dog class object to the variable **classObject**. We get the dog class object from the macro **_dog**. All SOM classes have an associated macro with which one can get the class object for that class. The macro takes the form **_<classname>**; **_dog**, for example.

Line 14 uses the **somClass** factory method **somNew** to create a new dog object. This method is defined in **SOMClass** and is supported by all class objects. It returns a new object of the type for which the receiving object is the class object. In this case, the receiving object, **classObject**, is the class object for dogs, and, therefore, this method returns a new dog.

The rest of the program is standard stuff. Line 16 asks our new dog to bark. Line 18 deinstantiates the object. Line 19 frees our environment variable.

Version 1 of the program does not need dynamic class loading. It can build everything it needs directly into the test executable. If it does place the dog code into a DLL, it does so as a convenience.

Let's consider a variation on this program. Suppose we want this program to handle not only dogs, but arbitrary classes derived from dog. To make this program as general as possible, we will modify it to ask the user what kind of dog she wants and then create a class object for that particular type of dog.

Our existing program needs some modifications. We can't use **_dog** to find our class object because we won't know what class we are going to load until the user tells us.

For this type of a program, we have a new set of requirements. First, we need to map arbitrary strings to class objects. Second, we need a way of loading in object DLLs that we didn't know about when our executable was built.

SOM provides an object called the **SOMClassMgrObject**. This object knows how to map between strings and class objects and also how to load in object DLLs at run time. **SOMClassMgrObject** is actually a global object reference to an object of type **SOMClassMgr**. The **SOMClassMgrObject** can be instantiated explicitly, by calling the **SOMEnvironmentNew**, or implicitly, through any of several calls to the SOM run time; for example, **dogNew**.

The **SOMClassMgr** class and, therefore, the **SOMClassMgrObject** object, support many methods. The method we are most concerned with here is **somFindClass**. This method is the one which, given a string, returns a class object and, if necessary, dynamically loads in the DLL containing that class's implementation.

Based on our method description, we can predict that **somFindClass** will take several parameters. First, we expect it to take an **Environment** parameter, our standard parameter for returning error information. Second, we expect it to take a string

Listing 1: dog.idl.

```
#include <somobj.idl>

interface dog : SOMObject {
  void bark();
  implementation {
    dllname = "dog1.dll";
    releaseorder: bark;
  };
};
```

parameter, the name of the class we want dynamically loaded and whose class object we want returned.

Oddly, neither of these two expectations is met. The **Environment** is not passed, because **SOMClassMgr** is one of those pre–SOM 2.0 classes that was created before SOM adopted the CORBA standard and, therefore, the **Environment** parameter. Also, the method does not take a string, although it does take a parameter that is closely related.

Instead of a string, **SOMFindClass** takes a parameter type called a **somId**. It's not worth going into the internals of a **somId** type. The most important thing to understand is that you can convert freely back and forth between a **somId** and a string, using the paired functions **somIdFromString** and **somStringFromId**.

Now let's look at the second version of **test.c**, modified to use the **somFindClass** method (Listing 3).

The significant changes in Listing 3 are:

- Line 9 declares a local variable of type **somId**.
- Line 10 declares a character array to hold the requested class name.
- Lines 15 to 17 ask the user to type in the requested class and stores the result.
- Line 19 converts the string containing the class name into a **somId**.
- Line 20 asks the **SOMClassMgrObject** to find the requested class, dynamically loading whatever DLL is necessary.
- Lines 23 to 26 print an error message if the requested class object couldn't be found.

Listing 2: Version 1 of test.c.

```
1. #include <dog.h>
2. #include <stdio.h>
3.
4. void main()
5. {
6.   dog snoopy;
7.   SOMClass classObject;
8.   Environment *ev;
9.   ev = SOM_CreateLocalEnvironment();
10.
11.  dogNewClass(0,0);
12.  classObject = _dog;
13.
14.  snoopy = _somNew(classObject);
15.  printf("Snoopy says:\n");
16.  _bark(snoopy, ev);
17.
18.  _somFree(snoopy);
19.  SOM_DestroyLocalEnvironment(ev);
20. }
```

Listing 3: Version 2 of test.c.

```
1. #include <dog.h>
2. #include <stdio.h>
3. 4. void main()
5. {
6.  dog snoopy;
7.  SOMClass classObject;
8.  Environment *ev;
9.  somId nameId;
10. char dogType [100];
11.
12. somEnvironmentNew();
13. ev = SOM_CreateLocalEnvironment();
14.
15. printf("Snoopy's Type: ");
16. fflush(stdout);
17. scanf("%s", dogType);
18.
19. nameId = somIdFromString(&dogType[0]);
20. classObject = SOMClassMgr_somFindClass
21.  (SOMClassMgrObject, nameId, 0, 0);
22.
23. if (!classObject) {
24.   printf ("Sorry... Can't load %s\n", dogType);
25.   exit(0);
26. }
27.
28. snoopy = _somNew(classObject);
29. printf("Snoopy says:\n");
30. _bark(snoopy, ev);
31.
32. _somFree(snoopy);
33. SOM_DestroyLocalEnvironment(ev);
34. }
```

In line 21, you may notice two extra parameters on the **somFindClass** invocation: the two zeros. These methods are used to specify major and minor version requirements, a topic we won't cover here. Using zeros for these parameters tells the method we aren't interested in versions.

To dynamically load in the DLL corresponding to a class, the **SOMClassMgrObject** must be able to determine which DLLs implement

which classes. It finds this information by using the interface repository, a run-time–accessible database that contains information about classes, including the name of the DLL in which the class implementation lives.

When we finish building our program and the dog class library, we will have both a **test.exe** and a **dog1.dll** created. Here's a typical run of our test program, with user input in bold:

```
Snoopy's Type: dog
Snoopy says: Generic dog noise
```

Of course, there are other possible outcomes. Consider this run:

```
Snoopy's Type: littleDog
Sorry... Can't load littleDog
```

The difference between these two program runs occurs when **somFindClass** attempts to locate the requested class in the interface repository. In the first case, **dog** is found in the interface repository. In the second case, **littleDog** isn't.

Now, this particular demonstration may not be very convincing. After all, **test.c** contained **dog.h**. How can we be sure that the **test.exe** was really using dynamic loading and wasn't merely calling code that had been preloaded in the **dog.h** file?

Returning from Edinburgh, it occurred to me that frameworks are a lot like travelers.

The best proof that **somFindClass** works as advertised is to use it to load a class that the executable couldn't possibly have known about at compile or build time. Therefore, we'll build a brand new DLL, one containing a littleDog. Its IDL definition will look similar to that of dog, with an override of the **bark** method. We will implement the littleDog **bark** method to type "woof woof".

The important point is that **test.c** knows nothing about littleDogs or their DLLs. If we rerun this program without recompiling or rebuilding, we get a completely different result by virtue of the newly available littleDog DLL. The new output looks like:

```
Snoopy's Type: littleDog
Snoopy says: woof woof
```

Of course, our test program will still fail if we ask it to load a class we haven't yet implemented (say, **bird**). But at least now we know how to solve that problem.

Importance of dynamic loading

Most people will not use dynamic loading to write test programs that

will load on-demand dog classes. Most people will use this capability to create frameworks that can manipulate classes that were unknown at the time the framework was built. This trick is very useful.

For example, consider the problem my client is trying to solve. These folks were developing a large framework to manipulate business objects. The resulting framework executable will be distributed throughout the company. Other groups in the company will create their own specialized business objects and use the framework executable to manipulate them.

My client's code won't necessarily require a user to enter in the class of a business object. Other ways of letting the framework know about newly available object classes are possible. Configuration files are one other common solution.

Without dynamic loading capability, framework distribution would have been very difficult. At the very least, the framework would have had to have been distributed as compiled *.obj* files, and framework programmers would have had to rebuild the product each time they wanted to add a new business object. Building a large, complex framework is by no means a trivial task.

Frameworks and travelers

Returning from Edinburgh, it occurred to me that frameworks are a lot like travelers.

There are those frameworks that are static. They are hardwired at compile time to specific classes and work well as long as they aren't expected to deal with any others. These frameworks do not require dynamic class loading. They are like the traveler that wants to plan a trip's every detail before taking the first step.

Then there are those frameworks that are dynamic. These frameworks require dynamic class loading to ensure that *any* object can be manipulated, even those unknown at compile time. These frameworks are like the traveler that will take in any new sight, any new experience, at the drop of a hat.

I am grateful to Edinburgh for reminding me how much I enjoy runtime coordination of both my travels and my frameworks.

The code for this article, including the makefiles and files not shown here, can be downloaded from the OS/2 Objects Home Page. Go to *http://www.fc.net/~roger/owatch.htm* and look for a link to code for *OS/2 Magazine* articles. OS/2

Roger teaches and consults on OS/2 Object technology. His books include the recently published Object Persistence—Beyond Object-Oriented Databases. *Roger also publishes the* OS/2 Objects Home Page *(http://www.fc.net/~roger/owatch.htm). He can be contacted at roger@fc.net.*