**The World of Objects**

**Dynamic**

**Frameworks**

BY ROGER SESSIONS



'

'I :

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

I must admit, I was at first per­ plexed 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 neu­ trality? Release-to-release binary

compatibility?

It turned out my client was inter­

ested 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 Edin­ burgh gave me the opportunity to think about using this feature to develop truly dynamic frameworks.

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

Let's consider a program that in­ stantiates a dog object as defined by dog.idl shown in Listing 1. This pro­ gram could be written in several ways, one of which is shown as ver­ sion 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. SOM­ Class is the default type of a class

**34 05/2 M A G A Z I N E** D E C E M B E R 1 9 9 6

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, there­

fore, this method returns a new dog.

The rest of the program is stan­

dard 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 pro­ gram to handle not only dogs, but arbitrary classes derived from dog. To make this program as general as pos-

sible, we will modify it to ask the user what kind of dog she wants and then create a class object for that particu­ lar 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 exe­

cutable 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. SOM ClassMgrObject is actually a global object reference to an object of type SOMClassMgr. The SOMClassMgr­ Object can be instantiated explicitly, by calling the SOMEnvironment­ New, 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 con­ taining 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. Sec­ ond, we expect it to take a string

**The World of Objects**



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

Oddly, neither of these two expec­ tations 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 COREA standard and, therefore, the Environment parameter. Also, the method does not take a string, al­ though it does take a parameter that is closely related.

Instead of a string, SOMFindClass takes. a parameter type called a som­ Id. It's not worth going into the in­ ternals of a somld type. The most important thing to understand is that you can convert freely back and forth between a somld and a string, using the paired functions somldFrom­ String and somStringFromld.

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

The significant changes in Listing

3 are:

Listing 2: Version 1 of test. c.



Listing 3: Version 2 of test. c.



• Line 9 declares a local variable of type somld.



• 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 con­ taining the class name into a somld.

• Line 20 asks the SOMClassMgr­ Object to find the requested class, dynamically loading whatever DLL is necessary.

• Lines 23 to 26 print an error mes­ sage if the requested class object couldn't be found.

In line 21, you may notice two extra parameters on the somFind­ Class 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 SOM­ ClassMgrObject must be able to determine which DLLs implement

**05/2 M A G A Z I N E** D E C E M B E R 1 9 9 6 35



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

When we finish building our pro­ gram and the dog class library, we will have both a test.exe and a dogl. 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 somFind­ Class attempts to locate the request­ ed class in the interface repository. In the first case, dog is found in the in­ terface repository. In the second case, littleDog isn't.

**The World of Objects**

Now, this particular demonstra­ tion 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 over­ ride 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

Can Your Client/Server

Test Tool Tackle The Entire

I 1

I

Testing Process?

**ATF** can.

If you want to learn about

ATF, fax this back to us at

6 1 7-8 6 4- 7 4 7

I

I

The Automated Test Facility tests applications under

Windows NT, Windows 95, Windows, and 05/2.

Name: \_ Company:: \_ Address: \_

City: \_

State: Zip: \_

Tel: -------------------------------

***SQFTBRIDGE***

*AutomatedTestFadfty*

*125 CambridgePark Drive*

*Cambridge, MA 02140*

*Phone: 617-576-2257*

*E-mail:* [*market@sbridge.com*](mailto:market@sbridge.com) [*www.sbridge.com*](http://www.sbridge.com/)

Reader Service No. 12

36 05/2 M A G A *Z* IN E D E C E M B E R 1 9 9 6



*Now Get*

**In Charge!**

**In Charge!** is a full function personal and small

business finance system for OS/2.

**In Charge!** supports:

• Multiple sets of financial books

• All types of accounts, from checking to stock margin

• Multiple currencies

• Securities portfolio management

• Powerful check printing facility

• CheckFree electronit bill payment

• Graphical reports

• Special small business functions

*And much more!*

**In Charge!** is available through dealers, or directly from Spitfire Software for only



$79

+Shipping

(404) 257-0187 • Fax: (404) 255-8032

Reader Service No. 13

-/



**The World of Objects**

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

For example, consider the prob­ lem my client is trying to solve. These folks were developing a large framework to manipulate business objects. The resulting framework exe­ cutable will be distributed through­ out the company. Other groups in the company will create their own specialized business objects and use the framework executable to manip­ ulate them.

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

Without dynamic loading capabil­ ity, 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 program­ mers 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 oc­ curred to me that frameworks are a lot like travelers.

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

tail before taking the first step.

Then there are those frameworks

that are dynamic. These frameworks

require dynamic class loading to en­ sure that *any* object can be manipu­ lated, even those unknown at com­ pile 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 re­ minding me how much I enjoy run­ time 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*](http://www.fc.net/-roger/owatch.htm)

and look for a link to code for *OS/2*

*Magazine* articles. **[i]fiJ**

*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/*](http://www.fc.net/-roger/) *owatch.htm). He can be contacted at roger@(c.net.*

ll



**For Free!** For more information order a free full function demo packet.

Fax or mail this coupon to Alaska. **Fax: +49/6196/95 72-22**

Company

Reader Service No. 14

**05/2 M A G A Z I N E** D EC E M B E R 1 9 9 6 **37**

