White Paper: CA-Visual Objects and the Jasmine API

Revision 1.0: April 1998
Paul Piko and Peter Fallon

Introduction

Whenever the API is discussed in the Jasmine documentation, C source code examples are not far away. But of course you are not restricted to using C to manipulate a Jasmine database via the API. CA-Visual Objects, with its understanding of C data types and calling conventions, can be easily used in conjunction with the API.

As such, we thought it would benefit if we provided some examples of raw Jasmine API use in CA-Visual Objects. We hope that this gives you an idea of the API's capabilities by being able to walk through the steps of a working program.

Why use the Jasmine API at all?

If you have read our white paper entitled *Object Databases and Jasmine Development Options*, you will know something of the API in context with the other Jasmine development options. The API provides you complete access to the Jasmine engine, via the ODQL language itself. Since it operates directly with Jasmine, there is no middle layer of ActiveX code to slow things down, or limit what you can do.

One thing should be apparent in this example. The client API consists almost entirely of functions devoted to putting data into, and getting data out of ODQL variables at runtime. Only a handful of functions actually perform any actions – and most of those indirectly. The real action comes from sending ODQL commands to the runtime engine for processing. I mentioned the word "*client*" above deliberately, as there are a large number of functions that relate solely to server side execution, that is, methods written in C. Writing database methods is something we cannot do in VO at this point because of the way Jasmine compiles methods.

(A quick note for those of you who dabble in Visual Basic 5.0. There is a clash between the VB 5.0 IDE and the Jasmine API. A VB program using the API will fail inside the IDE with errors – but an EXE program apparently works. This makes VB development with the API very difficult.)

Sample Application: BOUTIQUE.EXE

The documentation supplied with Jasmine includes a tutorial that covers the basics of creating Jasmine applications using various tools. You can find an electronic version of the tutorial on the Jasmine installation CD -- if you have the Workgroup or Enterprise Edition of Jasmine the tutorial also comes as a printed book.

Chapter 5 of the tutorial provides an overview of developing an application using the Jasmine C API, and includes C code for a sample application. In this paper we will produce a similar sample application, but we will use CA-Visual Objects as the programming language. We will follow the same approach taken in the chapter so you can see where the similarities and differences occur. Later, when you find other examples given in C, you should be able to understand how to approach a conversion into the CA-VO equivalent.

This example assumes a working knowledge of CA-Visual Objects programming techniques and a familiarity with calling DLL functions. The full source code is supplied with this paper in the file Boutique.AEF.

The steps listed here correspond with the steps presented in the tutorial. You should read these notes in conjunction with the tutorial – some information has been omitted to avoid repetition.

Preliminary Information in the Program

In order to use the Jasmine API from CA-Visual Objects we need to have prototypes, constants and structures defined for all the API features that we want to use. Unfortunately these are not supplied with Jasmine, however we have collected together the ones necessary for this sample application and included them in separate modules within the application.

Error Checking

Throughout the code the status of calls to the Jasmine API is checked. If at any time an error occurs (indicated by a status that is **not** ODB_NORMAL), a text box will display the error:

```
FUNCTION  ShowMessage (pSession)
   LOCAL  pMsg AS PTR
   LOCAL  liStatus AS LONG
   LOCAL  liLen AS LONG

   // Go get the information from Jasmine
   pMsg :=  MemAlloc(512)
   odbGetError(pSession,@liStatus,pMsg,512,@liLen)
   TextBox {,"Information",Mem2String(  pMsg,liLen),BOXICONASTERISK}:show()
   MemFree (pMsg)
```

Entry Point for Starting the Program

The complete code for this sample will exist within the Start() method of the application:

```
METHOD start CLASS app
```

Variable Declaration

The following code specifies the data types of the variables used within BOUTIQUE.EXE. The **odbData** structure mentioned in the tutorial, is used to switch between Jasmine's internal data formats and that of the programming language in use. However, it is an internal structure that is not intended for direct access by the programmer, so has been defined here as a pointer (see the variables **data** and **pPhoto**).

```
   LOCAL  pSession AS  odbSessionHandle
   LOCAL  liStatus AS LONG

   LOCAL  cVirtualNode AS STRING
   LOCAL  cUserId AS STRING
   LOCAL  cPassword AS STRING
   LOCAL  cEnvFile AS STRING
   LOCAL  cDBName AS STRING

   LOCAL DIM  data[10] AS PTR

   LOCAL  i AS DWORD
   LOCAL  liScanId AS LONG
   LOCAL  liActual AS LONG
   LOCAL  cName AS STRING
   LOCAL  pPhoto AS PTR

   LOCAL  cFile AS STRING
```

Connecting to the Jasmine Database

In order to make the connection code clearer we have split it up into parts. In order to establish a Jasmine session we need to pass across the name of the database we want to access, and an appropriate user name and password.

```
   cVirtualNode := ""        // The name given to the connection
   cUserId := ""             // use * for an installation password
   cPassword := ""
   cEnvFile := ""            // environment setting file
```

```
// If no virtual node is specified, Jasmine will    try to use local database
IF ! Empty( cVirtualNode)
      cDBName :=  cVirtualNode+"::jasmine/jasmine"
ELSE
      cDBName := "jasmine/jasmine"
ENDIF

liStatus :=  odbSesStart( @ pSession, PSZ( cDBName), PSZ( cUserId), ;
                          PSZ( cPassword), PSZ( cEnvFile) )
```

odbSesStart() and odbSesEnd() Functions

odbSesStart() needs to be called to establish a Jasmine session and allow communication with the database. When you have finished using Jasmine you need to make a call to odbSesEnd() to terminate the session (Note: the C code listed in the tutorial is missing the call to odbSesEnd())

Session Handle

The local variable **`pSession`** holds the session handle created by the odbSesStart() function. All functions that need to communicate with the Jasmine server require the session handle as an argument.

Checking the Return Code and Showing an Error Message

The following code performs error checking, uses the odbExecODQL() function to provide direct access to the Jasmine database for executing ODQL statements, then begins a transaction with Transaction.start.

```
IF ! ( liStatus :=  odbExecODQL( pSession,;
                                 PSZ("Transaction.start();"),;
                                 0,NULL_PTR ) ) == ODB_NORMAL
      ShowMessage (pSession)
ENDIF
```

The Default Class Family

The following code specifies the default class family (CAStore).

```
IF ! ( liStatus :=  odbExecODQL( pSession,;
                                 PSZ("defaultCF  CAStore;"),;
                                 0,NULL_PTR ) ) == ODB_NORMAL
      ShowMessage (pSession)
ENDIF
```

Declaring Variables

The following code declares ODQL variables for use later in a query. Note that the class that **`photo`** belongs to is CABitmap – the tutorial incorrectly lists it as Bitmap.

```
IF ! ( liStatus :=  odbExecODQL( pSession,;
            PSZ("T[String name,   mediaCF::CABitmap photo] set   ts;"),;
                                 0,NULL_PTR ) ) == ODB_NORMAL
      ShowMessage (pSession)
ENDIF
```

Executing the Query

This code executes the query to create a set of objects belonging to the Top class that have a property of “hot” that is greateer than zero.

```
IF ! ( liStatus :=  odbExecODQL( pSession,;
            PSZ("ts = [x.name,  x.photo] from Top x where   x.hot > 0;"),;
                                 0,NULL_PTR ) ) == ODB_NORMAL
      ShowMessage (pSession)
ENDIF
```

Allocating Space for Fetching Data

We need to allocate some space for the results of our query by making calls to odbAllocData():

```
FOR i := 1 TO 10
      data[ i] := odbAllocData()
NEXT
```

Opening the Result Set

The following code opens a scan on the set named **ts**.

```
IF ! ( liStatus :=  odbScanOpen( pSession,;
                                 PSZ(" ts"),;
                                 @liScanId,0) ) == ODB_NORMAL
      ShowMessage (pSession)
ENDIF
```

Fetching 10 Items From the Database

The following code fetches up to 10 elements into the array **data**. Values are then retrieved for each element – the photo bitmap is extracted to a file, and it's name and the name of the Top are displayed in a text box.

```
IF ! ( liStatus :=  odbScanFetch(pSession,liScanId,10,;
                                 @liActual ,@data) ) == ODB_NORMAL .and. ;
                                 ! liStatus == ODB_SCAN_END
      ShowMessage (pSession)
ENDIF

FOR i := 1 TO  liActual
      cName :=  odbGetStr( odbFieldAt(data[ i],0))

      pPhoto :=  odbFieldAt(data[ i],1)
      cFile := "D:\HOT"+NTrim(  i)+".BMP"
      IF ! ( liStatus :=  odbGetMMDataToFile(  pSession,pPhoto,;
                                               PSZ( cFile)) ) == ODB_NORMAL
            ShowMessage (pSession)
      ELSE
            TextBox {,"Information","File "+  cFile+;
                        " has been created for photo property of "+;
                        cName}:show()
      ENDIF

NEXT
```

Closing the Result Set

The following code closes the scan on the set **ts**.

```
odbScanClose( pSession,liScanId)
```

Ending the Transaction

The following code ends the transaction and deallocates the Jasmine related data:

```
odbExecODQL( pSession,PSZ(" Transaction.end();"),0,NULL_PTR)

FOR i := 1 TO 10
      odbFreeData(data[ i])
NEXT
```

"Day of the JACL™" – A Preview

As indicated in previous announcements, we are working on a set of classes for CA-VO that completely encapsulates the Jasmine API. Using our classes, the above example from the API tutorial can be rewritten something like this:

```
METHOD start CLASS app

   LOCAL oSession AS  jSession

   LOCAL cVirtualNode AS STRING
   LOCAL cUserId AS STRING
   LOCAL cPassword AS STRING
   LOCAL cEnvFile AS STRING

   LOCAL oDemoQuery as  jQuery
   LOCAL oPhoto as  jMMedia

   LOCAL cName AS STRING
   LOCAL cFile AS STRING

   cVirtualNode := ""       // The na me given to the connection
   cUserId := ""            // use * for an installation password
   cPassword := ""
   cEnvFile := ""           // environment setting file

   oSession :=  jSession{ cVirtualNode,cUserId,cPassword,cEnvFile}

   //session object can be configured to display e   rrors
   oSession :showErrors := TRUE

   IF oSession :Connect()
         TextBox {,"Information","Jasmine session established", ;
                  BOXICONASTERISK} :show()

         oDemoQuery :=  jQuery{, "T[String name,   mediaCF::CABitmap photo]",;
                                 jCOLLECTION _SET }

         oDemoQuery :QueryExpression := "Top x where   x.hot > 0"
         IF oDemoQuery :RunQueryExpression()

               FOR i := 1 TO  oDemoQuery:Count()

                     cName :=  oDemoQuery:Item:name
                     oPhoto :=  oDemoQuery:Item:photo

                     cFile := "D:\HOT"+NTrim(  i)+".BMP"
                     IF oPhoto :GetToFile( cFile)
                           TextBox {,"Information","File " +   cFile + ;
                              " has been created for photo property of " + ;
                              cName}:show()
                     ENDIF

                     oDemoQuery :ScanNext()
               NEXT
         ENDIF

         oSession :Disconnect()

   ENDIF
```

We feel that is a big improvement, on readability, maintainability while retaining the speed benefit!

The **J**asmine **A**PI **C**lass **L**ibrary, or the **JACL**™, consists of 4 basic layers:

Layer 1 – Base

This layer includes all the relevant declarations, prototypes and structures to deal with the API from CA-VO. A subset of the base layer was used in the first Boutique example above. You can program using the direct Jasmine API calls, as long as you are happy allocating and freeing memory. It's useful to at least understand this layer, as it makes reading the Jasmine sample programs and API documentation easier to follow. As we showed, you can write a VO program that is almost a 1:1 match for any C program that uses the API.

Layer 2 – Low level Calls

This layer takes the base layer and wraps it in a CA-VO blanket. All of the calls correspond to the API functions but with minor exceptions you are handling CA-VO data types instead of C ones. This makes using the API more manageable as it looks after conversions from the more common CA-VO data types to the required C data types.

Using this layer, you can perform any client API task. You can use this layer to create your own set of classes, or extend ours! Naming conventions have been kept close to the original API calls deliberately so you can still use the Jasmine API documentation.

Layer 3 – VO Classes

As you saw in the second example program, we have created a set of classes such as jSession{} and jQuery{} that encapsulate the fundamental data types and operations performed with the API. Gone is the need to worry about memory allocation, pointers, or the finer points of transaction management in ODQL statements.

The classes we have created cover all of the ODQL data types. Several CA-VO classes encompass metaclass information such as Jasmine class structures and class families. We have taken care of many common operations and dependencies such as wrapping query statements inside a transaction, so that you can deal with the basic data handling issues. We are providing comprehensive error handling and the ability to trace and record ODQL command execution, so that any problems you have can be handled either at a high level, as the example above shows, or as discretely as you like.

The most important aspect of these classes is that Jasmine objects will look, act and feel like VO objects. Property access and method execution will be normal VO code.

Layer 4 – ODQL in VO

The final layer is what will set the library apart. The big complexity with using the API is that everything is done through ExecODQL() functions, which just run OQDL statements like macros or code blocks. This means that you have to be very familiar with ODQL to use the API.

Our solution is to create classes that encompass most of the ODQL language, so you will rarely have to pull up the ODQL reference if you just want to manually start a transaction, or see if one is open, or query a class's structure, or any other operation. Additionally, most of the ODQL data types are classes themselves, with class methods and properties available for the taking (eg: the collection classes have methods for sorting, adding and removing items). This layer will provide those as properties and methods of the respective "j" classes constructed by Layer 3.

Our goal here is simple. If you are creating Jasmine databases, and writing ODQL methods (and of course know CA-VO), we want it to be a no-brainer to extend that knowledge to use our classes to write API enabled programs.

Availability

The JACL is currently under development and will be taking into account the features to be included in CA-VO 2.x. However, to get the most out of the Jasmine API interface you have to be adept at ODQL programming, as that is what drives all API operations. So while our class library will take care of the nitty gritty declarations, memory management, and Jasmine/C/VO data type conversions, you still need to know ODQL to get the most out of the library.

So we are also working on our next set of training materials:

Jasmine Database Construction and ODQL Programming

> These course notes take a look at all the steps involved in creating a Jasmine database. We start from scratch, and build a Store, Class Family and several classes and methods and populate it with some data. As each piece is assembled, we discuss the issues involved, the decisions you are faced with and the options Jasmine allows.

But this course is not restricted to learning how to use the Database Administrator. We take a solid look at Object Data Query Language (ODQL), how it is put together, what you can do with it, how to write methods, how to use it for ad-hoc database work, and how it is used in the C API.

Jasmine API Programming in CA-Visual Objects

Sometimes if you want to apply real power to a problem, you have to build your solution with the finest and most detailed parts. The Jasmine C API gives you complete access not only to Jasmine, but to the ODQL engine, so there isn't anything you cannot do, and no problem you cannot approach from several different directions.

The course provides the VO prototypes for all the client API functions, walks you through building queries, retrieving data, working with ODQL, issues with API programming and insights on how you should encapsulate these functions inside your own classes. This course assumes a good working knowledge of VO and Jasmine, especially ODQL.

So, why tell you about this now?

We felt it was important to ensure that no one was discouraged by the bias towards C in Jasmine's API documentation. Almost everything that is listed there can be done with VO, with a little effort, spit, polish, and a few calls to CA-Support (the only things that cannot be done are server-side calls).

One final important item to note: ODQL is **case sensitive!!!** The language is very insistent on everything being exactly, syntactically and grammatically correct, and takes no prisoners. If you are experimenting and can't work out why you are getting loads of error messages from a single, simple expression – check the help files and make sure your use of case is exactly correct.

Authors:

If you've got any more questions, or would like to know about Jasmine training, course notes and tools (Jasmine and CA-Visual Objects) please contact either of us. We are:…

Paul Piko
Piko Computing Consultants
8 Hedline Place, Macleod, VIC, 3085
Australia
Ph: (+61 3) 9432 1222
Fax: (+61 3) 9432 1255
piko@compuserve.com
http://ourworld.compuserve.com/homepages/piko

Peter Fallon
Castle Software Australia
3/8 Reid St Ashwood, VIC, 3147
Australia
Ph: (+61 3) 9885 5184
Fax (+61 3) 9886 0100
pfallon@compuserve.com

Copyright Notice

Trademark Acknowledgements