Go Clone Yourself

Mark Hunter

Too bad "Business Objects" is already used as a product name by a vendor with the same name. Business objects are a flexible and robust approach to class design in SAL, and their value will continue to increase as companies increase component-oriented development. To me, a "business object" in SAL is a user-defined variable (UDV) that represents an entity in the developer's data model. It's based on a class definition that encapsulates one database table or a group of related database tables. In the sample application we'll examine later on, we have class definitions for Customer and for Invoice, and the Customer class includes a set of Invoice objects.

More sophisticated OOP features in SQLWindows are far away, behind more pressing enhancements like COM servers. But you can still do a lot today with SAL's OOP features, both 16-bit and 32-bit. Here's an unusual but effective technique for working with user-defined variables (UDVs).

There are definite advantages to business objects. They make it easy to move to a three-tier or n-tier application architecture. Database access and business rules are located in the business objects themselves. The user interface running on the client simply requests data items from the business object and sends items back, through a function call interface. Thus, the user interface client can be relatively "thin."

SQLWindows

16 / 32

In today's world, with the lack of COM server functionality in CTD, the business object and user interface must reside in the same executable, logically separate but physically joined. The only way around this constraint is to use special add-ons like the Remote Objects capability in Pivato Consulting's XSal. But that will change soon—much sooner than you might think—and true distributed applications will be available in CTD. When that happens, the developer who has been using business objects will be far ahead of the pack, able to quickly put those business objects into a server application. And for those who are still using 16-bit SQLWindows, business objects will work just fine, and will be a major advantage after the move to 32-bit.

Limitations

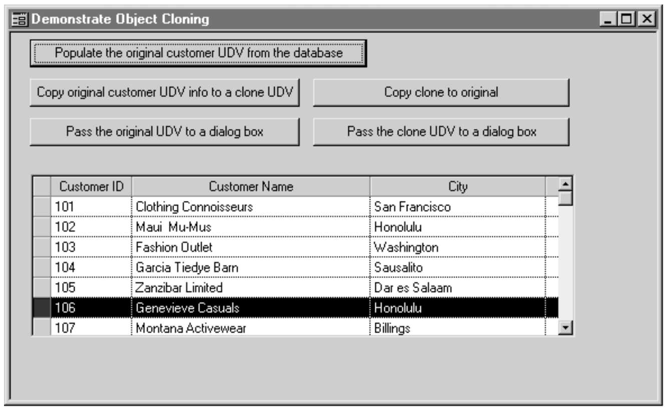
So, business objects are effective today and are truly superior tomorrow. What's the downside of using them? For one thing, it can be a little difficult to manipulate UDVs in SAL. Here are some of things that can't be done:

```
◆ Function: fDoSomething
  ◆ Returns
    ◇ Boolean:
  ◆ Parameters
    ◇ Receive fcSomeEntity: Entity ! can't declare a
      UDV as a receive parameter

◆ Function: fDoSomethingElse
  ◆ Returns
    ◇ fcSomeEntity:  ! can't return a UDV, only a
      primitive datatype
  ◆ Parameters
    Number: nDatabaseKey

◆ Function: fDoMoreStuff
  ◆ Returns
    ◇ Boolean:
  ◆ Parameters
    fcSomeEntity: Entity
  ◆ Local Variables
    ◇ FcSomeEntity: AnotherEntity
  ◆ Actions
    ◇ Set AnotherEntity = Entity ! can't assign a
      UDV to another UDV
```

Workarounds

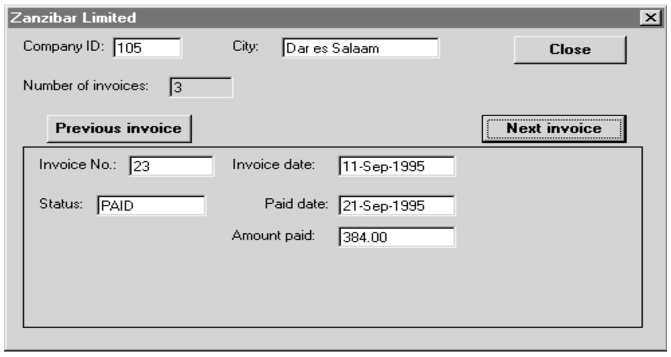
Gee, sounds like a lot of problems. Is it still worth basing your application architecture on UDVs? Yes, it is. Let's look at these issues more closely.

- It's not necessary to declare a UDV as a receive parameter to a function. All UDVs are always receive parameters when used as function parameters. That is, any change to a UDV's instance variables or class variables made inside the function


Demonstrate Object Cloning
Populate the original customer UDV from the database
Copy original customer UDV info to a clone UDV
Copy clone to original
Pass the original UDV to a dialog box
Pass the clone UDV to a dialog box

Customer ID	Customer Name	City
101	Clothing Connoisseurs	San Francisco
102	Maui Mu-Mus	Honolulu
103	Fashion Outlet	Washington
104	Garcia Tiedye Barn	Sausalito
105	Zanzibar Limited	Dar es Salaam
106	Genevieve Casuals	Honolulu
107	Montana Activewear	Billings

Figure 1. The sample application, CLONE.APP, showing data from the ISLAND database.



Company ID: 105
City: Dar es Salaam
Number of invoices: 3
Invoice No.: 23
Invoice date: 11-Sep-1995
Status: PAID
Paid date: 21-Sep-1995
Amount paid: 384.00

Figure 2. All the values shown in this dialog are contained in a single UDV of type fcCustomer.

will persist when the function ends. All UDVs are "passed by value." It's only primitive datatypes that are capable of being passed by reference (Receive Number: nParm) or by value (Number: nParm). My thanks to reader Randy Newburn for pointing this out. His comments inspired me to pursue the techniques in this article.

- Functions can't return a UDV; they can only return primitive datatypes. As we'll see shortly, the techniques presented in this article provide a good workaround for this limitation.

- In many object-oriented languages, it's perfectly legal to say "Object1=Object2". In SAL, you can't assign one UDV to another UDV. But you can come awfully close, using these techniques.

Some examples

Before we get too deep into the techniques, let's watch them in action. Start up CLONE.APP, shown in **Figure 1**.

This app populates a table window with the rows from the COMPANY table in the ISLAND database. When you click "Populate the original customer UDV from the database," the app uses the customer ID from the first column of the table window and tells UDV Customer to populate itself. The UDV fetches data from the COMPANY table and related sets of data from the INVOICE table.

When you click "Copy original customer UDV info to a clone UDV," the values in UDV Customer are copied into a second UDV, CustomerCopy. This is done with a single function call, which we'll examine below.

The next two buttons, "Pass the original UDV to a dialog box" and "Pass the clone UDV to a dialog box," cause a dialog box to appear. The dialog box takes one parameter of type fcCustomer. All the values displayed (see **Figure 2**) come from querying that UDV parameter's instance variables.

The last button, "Copy clone to original," causes values in CustomerCopy to overlay those in Customer, again using a single function call.

The "City" field in the dialog has been left editable. When the dialog ends, it copies the value in that field back to the CITY variable in the UDV, as shown:

```
◆ Pushbutton: pbClose
  ◆ Message Actions
    ◆ On SAM_Click
      ◇ Set parmCustomer.CITY=dfCity
      ◇ Call SalEndDialog(hWndForm,1)
```

Then the logic that called the dialog moves that CITY variable value into the City column of the table window. If you run the sample app, bring up the dialog, and change the City field, you'll see that change propagated to the table window. This was done through a UDV that was passed to the dialog as an ordinary parameter, not a Receive parameter:

```
◆ Pushbutton: pbShowOriginal
  ◆ Message Actions
    ◆ On SAM_Click
      ◇ Call SalModalDialog(dlg1,hWndForm,Customer)
      ◇ Set tbl1.colCity=Customer.CITY

◆ Dialog Box: dlg1
  ◆ Parameters
    ◇ fcCustomer: parmCustomer
```

So we see that UDVs passed as function parameters always behave as if they were Receive parameters.

And functions can't use class definitions as return datatypes, so functions can't return UDVs. That's too bad. In our thin client of the future, we might very well ask our middle tier to give us a customer where COMPANY_ID=103, for example. But something quite similar is possible in SAL. Let's see what really happens when you click "Populate the original customer UDV from the database" in the sample app. First, the main form defines a couple of UDVs: Customer and CustomerCopy. They're based on class definition fcCustomer:

```
◆ Functional Class: fcCustomer
  ◇ Description:
  ◇ Class Variables
  ◆ Instance Variables
    ◇ Number: COMPANY_ID
    ◇ String: COMPANY_NAME
    ◇ String: ADDRESS
    ◇ String: CITY
    ◇ String: STATE
    ◇ String: ZIP
    ◇ String: COUNTRY
    ◇ String: PHONE
    ◇ String: FAX
    ◇ String: TERMS
    ◇ Number: LINE
    ◇ String: CORPORATE_URL
    ◇ fcInvoice: Invoice[*]
    ◇ Number: nInvoiceCount
  ◆ Functions
    ◇ Function: fCloneYourself
    ◇ Function: fPopulateYourself
```

When the app starts, all the instance variables in these UDVs are empty. But when the pushbutton is clicked . . .

```
◆ Pushbutton: pbPopulate
  ◆ Message Actions
    ◆ On SAM_Click
      ◇ Call Customer.fPopulateYourself(hSql,
        tbl1.colCompanyId)
```

. . . the main form tells that empty UDV to populate itself and passes the Customer ID column from the table window to tell it which COMPANY row is desired. fPopulateYourself isn't too complicated. The only thing to note is that fcCustomer is a complex object. It not only contains several variables with primitive datatypes; it also contains an array of type fcInvoice. fcInvoice is the class definition that describes data from the INVOICE table. Each customer has a set of invoices that belongs to it, thus class fcCustomer uses array Invoice to hold this information. During fcPopulateYourself( ), the primitive datatypes are fetched from table COMPANY. Then a loop is set up to fetch all related invoices:

```
◇ ! Set up an array of invoice objects and
  fetch database values into each element of the array.
◇ Set nInvoiceCount=0
◇ Set nIndex=0
◇ Call SqlPrepareAndExecute(hSql,
    'select INVOICE_NO
    from INVOICE
    into :nInvoiceNo
    where COMPANY_ID=:nParmCompanyId')
◆ While SqlFetchNext(hSql,nReturn)
  ◇ Set nInvoiceCount=nInvoiceCount+1
  ◇ Set nIndex=nInvoiceCount-1
  ◇ Call Invoice[nIndex].fPopulateYourself
    (nInvoiceNo)
```

What's so interesting about this? Well, I decided that class fcInvoice knows a lot more about how to get invoice information from the database than class fcCustomer does. So you'll note that fcCustomer.fPopulateYourself( ) doesn't run full database queries against table INVOICE. It runs a query to get only the invoice keys related to the current customer. Then it simply points to each object in the Invoice array, hands it the key value, and tells it to populate itself. This reduces the complexity of fcCustomer, and keeps the database access logic for table INVOICE over in class fcInvoice, where it belongs.

So, this complex class fcCustomer has an array of objects of type fcInvoice, which populate themselves when asked. In actuality, each INVOICE in database ISLAND has a set of INVOICE_ITEMs, and those items are related to PRODUCT rows, etc. So it would appear that a realistic business object for a customer could be quite complex. (We won't go there now—two levels of encapsulation is plenty for demonstration purposes!) But by using properly encapsulated class definitions, the complexity is largely hidden from the developer. After all, the complexity of an entire set of invoice information is confined to just a single array and a few lines of code in the populate function, in fcCustomer. Even if I had put an additional array in fcInvoice to hold INVOICE_ITEM information, that wouldn't increase the apparent complexity of fcCustomer.

So, although SAL won't let a function return a UDV, we can define an empty UDV, pass it along to a function, and let the function do the work. Great, now our Customer UDV is crammed full of information. Let's click "Pass the original UDV to a dialog box." Looking back at the first code listing, we see that Customer is passed as a parameter to the dialog. Inside the dialog, that UDV is used as the source of values to be placed in the data fields:

```
◆ On SAM_CreateComplete
  ◇ Call SalSetWindowText(hWndForm,
  parmCustomer.COMPANY_NAME)
  ◇ Set dfInvoices=parmCustomer.nInvoiceCount
  ◇ Set dfCity=parmCustomer.CITY
  ◇ Set dfCompanyId=parmCustomer.COMPANY_ID
  ◇ Call fSetInvoiceData()
```

If I were a good OOP citizen, I wouldn't refer directly to the instance variables like this—I would have Get and Set functions for each variable and would use those functions instead. Oh, well, maybe when the deadlines aren't so pressing, I'll do it right. Don't worry; my production apps do it that way! If you want a tool to make Get and Set coding faster, see the July 1998 issue of *Centura Pro*, which offers David Burke's utility, GETSET.ZIP.

The dialog box has no need to go to the database, because the UDV has all the information it needs, including a complete set of invoice information. And we didn't need dozens of parameters to accomplish this;

just one. Changes made to the information (like editing the City data field) are written back to the UDV, and when the dialog ends, the UDV contains all changes.

Actually, that's important to remember. In this case, we wanted to make changes to the UDV. But any time you use UDVs as function parameters, your UDVs aren't protected from changes made inside functions, intentional or otherwise. How could you protect them? Well, you could make a copy of the UDV first…but SAL won't let you, right?

Something different

That's right. But I used a workaround. Take a look at the function fCloneYourself( ) in fcCustomer:

```
◆ Function: fCloneYourself
  ◇ Description:
  ◇ Returns
  ◆ Parameters
    ◇ fcCustomer: parmCustomer
  ◇ Static Variables
  ◆ Local variables
    ◇ Number: nIndex
  ◆ Actions
    ◇ Set parmCustomer.COMPANY_ID=COMPANY_ID
    ◇ Set parmCustomer.COMPANY_NAME=COMPANY_NAME
    ◇ Set parmCustomer.ADDRESS=ADDRESS
    ◇ Set parmCustomer.CITY=CITY
    ◇ Set parmCustomer.STATE=STATE
    ◇ Set parmCustomer.ZIP=ZIP
    ◇ Set parmCustomer.COUNTRY=COUNTRY
    ◇ Set parmCustomer.PHONE=PHONE
    ◇ Set parmCustomer.FAX=FAX
    ◇ Set parmCustomer.TERMS=TERMS
    ◇ Set parmCustomer.LINE=LINE
    ◇ Set parmCustomer.CORPORATE_URL=CORPORATE_URL
    ◇ Set parmCustomer.nInvoiceCount=nInvoiceCount
    ◇ Set nIndex=0
    ◆ While nIndex<nInvoiceCount
      ◇ Call Invoice[nIndex].fCloneYourself(parmCustomer.Invoice[nIndex])
      ◇ Set nIndex=nIndex+1
```

At the beginning of the article I promised you an "unusual" technique, and here it is. Remember, this function is inside class definition fcCustomer. Note that it takes a single parameter of type….fcCustomer! It uses its own containing class definition as a parameter. This means that I can call this function in an fcCustomer UDV, and pass it a second, empty UDV as a parameter. All the values from the first UDV will get copied to the second UDV! This is exactly what happens when you click "Copy original customer UDV info to a clone UDV":

```
◆ Pushbutton: pbCloneTheCustIomer
  ◆ Message Actions
    ◆ On SAM_Click
      ◇ Call Customer.fCloneYourself(CustomerCopy)
```

It's just a one-line function call. Almost as compact as a real OOP language, eh? It works in both directions; when you click "Copy clone to original":

```
◆ Pushbutton: pbCopyCloneToOriginal
  ◆ Message Actions
    ◆ On SAM_Click
      ◇ Call CustomerCopy.fCloneYourself(Customer)
```

This is, indeed, unusual, but it's very effective. The cost to you as a developer is the time it takes to write the fCloneYourself( ) function for each of your business object class definitions. It's reasonably easy, and the benefits are very significant. In CLONE.APP, you can see that I maintain two copies of the same customer's information in two separate UDVs. You can change information (such as City) in one but not the other. In a production application you could store the original information in one UDV while you let the user edit the other UDV. When the user is finished, you could do comparisons between old and new values to enforce business rules, calculate net change, and accomplish other tasks, all without going to the database yet. If the user decides to start fresh, you don't have to go back to the database; you simply clone the original UDV's values to the copy again. There are many possibilities, so go ahead—go clone yourself! **CP**

Download Clone.zip from this issue's Table of Contents at www.ProPublishing.com or find it on this month's Companion Disk.

Mark Hunter, aside from being editor for this very publication, also works his Centura consulting, training, and mentoring magic from Hunter Software, Inc., located in Southern California, but serving a worldwide clientele.

Buy or Build?

As noted in the article, the object cloning techniques require you to write one function, fCloneYourself, in each class definition that will be subject to cloning. Although this function isn't very complicated or time-consuming, you may still prefer a solution that doesn't require you to code new functions. If so, check out XSal2 at www.pivato.com. This third-party add-on for SQLWindows has very fast, effective functions for cloning objects. Further, XSal2 can serialize objects: read or write them into large strings, with actual instance variable values intact, that can be stored in files or databases, or passed to other applications. This turns SQLWindows into a 4GL with more advanced object-handling capabilities than any other. In addition to these capabilities, XSal2 has a large number of other features.—*M.H.*