Using Interfaces with PowerBuilder

Implementing in the real world

PART 2 OF 2

WRITTEN BY
ROLAND MÜHLBERGER

Wouldn't it be nice if we could use DataWindows, DataStores, and DataWindowChild objects in the same way. This article shows you how.

In Part 1 (***PBDJ***, Vol. 9, issue 11) I described a way to implement Java's and C#'s interface concept into PB. We did the following:

- Created a wrapper class (the "interface") that has a reference to a PowerObject as an instance variable
- Created a method within the wrapper class for every method of the interface
- Called the actual method per dynamic call
- Encapsulated the call with a TRY/CATCH-block in PB 8 and above
- At runtime, created the interface object, set its base object, and used it

This time we move away from concepts and switch over to using interfaces in the "real programmer's world." This article shows you how to write algorithms (and also the pitfalls along the way) that don't distinguish between DataWindows, DataStores, and DataWindowChild objects (if you come across the term "DataWindowChildren", don't worry, this is my personal abbreviation for DataWindowChild objects).

Knowing Your Way

Before starting our journey, we need to know why we should take it. First, what's our goal? We want to treat DataWindows, DataStores, and DataWindowChildren the same. Why? These classes are essential for PowerBuilder programmers as a good programmer frequently uses them. Why can't we just create one single variable and let it point to the appropriate object at runtime? Figure 1 sheds some light on this.

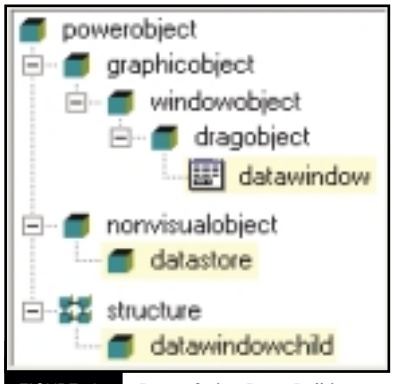

FIGURE 1 Part of the PowerBuilder class hierarchy

The common base class for all three classes is PowerObject, which is limited in the number of methods it understands. DataWindows and DataStores do have lots of methods in common, but because they are on completely different branches of the inheritance tree, you can't use a single variable to point to a DataStore or a DataWindow and use their common methods.

With this in mind, it's not easy to write an algorithm that works on any of those objects; you need lots of CASE statements. Sometimes using ShareData will help, but this is limited to buffer manipulations. Let me show you where it won't matter if the object you work on is a DataStore, DataWindow, or a DataWindowChild. Sounds promising, doesn't it?

Get Going

As already mentioned, the three classes have many methods in common. Just think of all the GetItem statements, Sorting, Filtering... Our first step is to determine the common methods (see Figure 2).

The green rectangle represents all the DataWindow methods, the yellow section all the DataStore methods, and the pink one all the DataWindowChild methods. The orange segment is all the methods that DataWindows and DataStores have in common, and the blue is all the methods shared by all three classes.

In Figure 2, the DataWindow has all the methods of the DataWindowChild, and the DataStore has some methods the DataWindow doesn't. In fact, only one method, CreateFrom, goes into the yellow part.

Table 1 shows you some prominent examples of methods from each section.

Moving Along

In the table the blue column is the most important; it holds all the methods that all three classes understand. Implement an interface called IDwCore ("Interface DataWindow Core") for it, then inherit two classes from it: the interface class IDwDs ("interface DataWindow + DataStore") for the orange part and IDwDwc ("interface DataWindow + DataWindowChild").

Figure 3 shows the resulting class hierarchy. For the base class use IBaseAs, the base interface class. (I constructed and explained the base class in Part 1. For more information, visit www.romu.com/interfaces.)

Implementing the interface methods is fairly straightforward, and the code is pretty much the same for all of them. The following code is an exam-

Color	Classes	Methods
	Only DataWindow	Copy, paste, show, hide
	DataWindowChild and DataWindow, but not DataStore	ScrollToRow, SetRedraw, SetRowFocusIndicator
	All three classes	SetItem, GetItemX, Find, Filter, Sort, SaveAs
	DataWindows and DataStores, but not DataWindowChild	Create, print, all graph methods
	DataStore only	CreateFrom

TABLE 1 Examples for methods

ple of the interface method for AcceptText:

```
Integer i_ret

// ---

TRY
   i_ret = ipo_base. DYNAMIC
Accepttext()
CATCH (RuntimeError rte)
   IF rte.number = 65 THEN
      // R0065 - dynamic function not
found
      MethodNotImplemented(ipo_base,
'Accepttext', &
   'Integer Accepttext()', this)
   ELSE
      // some other error - throw
again
      THROW rte
   END IF
END TRY

RETURN i_ret
```

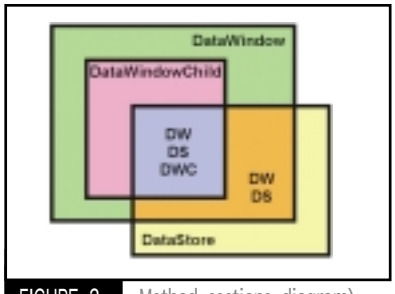

FIGURE 2 Method sections diagram)

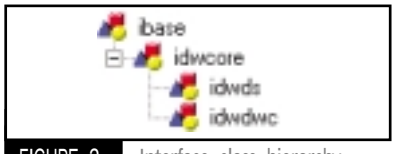

FIGURE 3 Interface class hierarchy

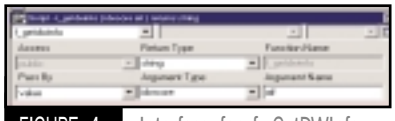

FIGURE 4 Interface for f_GetDWInfo

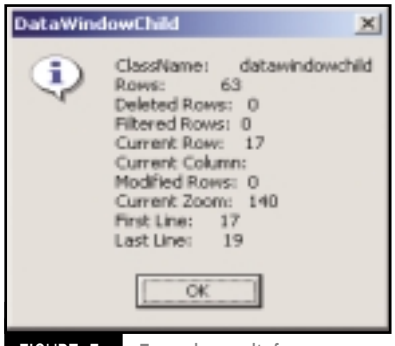

FIGURE 5 Example result for f_GetDWInfo

As mentioned in Part 1, I use SmartPaste, a PowerBuilder documentation tool, for this task (www.romu.com/smartpaste).

Be Careful Not to Trip

What we've done so far sounds easy: for each method that's in all three classes implement an interface method in IDwCore; for the ones that only DataWindow and DataStore have, implement the method in IDwDs; and methods of DataWindows and DataWindowChild go into IDwDwc. (You can download the code from www.sys-con.com/pbdj/sourcec.cfm or www.romu.com/interfaces.) The result of implementing all interface methods for IDwCore is an interface class with 126 methods. The inherited interfaces IDwDs and IDwDwc have 84 and 25 methods, respectively.

There are several methods that need some explanation.

VARIABLE PARAMETER LISTS

Some methods, like ImportClipBoard, ImportFile, ImportString, and Retrieve, have a variable list of parameters. (PowerBuilder doesn't allow us to implement such methods or override them, but you can see them in the PowerBuilder browser; they have an ellipsis (…) in the parameter list.)

For those methods we implement a whole range of interface methods in order to mimic a "variable parameter list". For instance, for Retrieve implement an interface method without a parameter, one with one parameter, one with two. If you want to use even more parameters, you need to enhance the interface class and implement a method with even more parameters

INSTANCE VARIABLES

DataWindows and DataStores have two identical instance variables that are commonly used and should also go into the interface IDwDs, object and dataObject. To achieve this, write interface methods called GetDWObject and GetDataObject (together with SetDWObject and SetDataObject) that access the instance variables by doing a type cast and then accessing the property. This is the only place in the interface classes where you'll find CASE statements. (Every interface method of IDwCore could be written that way, but this works only for our special combination of DataWindow, DataStore, and DataWindowChild. A big advantage of interfaces is that you usually do know which classes will implement the interface, and can therefore do this little trick.)

READ-ONLY ARGUMENTS

Some built-in methods have arguments that are passed by read-only. PowerBuilder does not allow you to use those arguments in a dynamic call, so you need to assign it a local variable first.

Having Reached the Goal

Now that we've implemented the interface classes, we're ready to use them. The usual way is:

1. Create a service class that operates on objects of the interface class. You can use all the methods the interface provides.
2. During runtime, create the interface for a class using CreateInterface. This function creates the interface wrapper and sets the reference that the interface object encapsulates.
3. Call the service class with the newly created interface object.

Our example will return information about the object: how many rows, the current row, the visible part, and lots more.

For the sake of simplicity implement this as a global function f_GetDWInfo, which takes an IDwCore-Object as a parameter and returns the information as a string. Figure 4 shows the interface.

Listing 1 shows the implementation. Use "if" for the interface type prefix. The next part shows how to call the function with any of the three classes.

DATAWINDOW

```
IDWCore if_core

if_core = CreateInterface(dw_test,
'IDWCore')
MessageBox('DataWindow',
f_GetDWInfo(if_core))
```

DATAWINDOWCHILD

```
IDWCore           if_core
DataWindowChild  dwc
dw_test.GetChild('state', dwc)
if_core = CreateInterface(dwc,
'IDWCore')
MessageBox('DataWindowChild',
f_GetDWInfo(if_core))
```

DATASTORE

```
DataStore ds_test
IDwDs     if_core

ds_test = CREATE DataStore

if_core = CreateInterface(ds_test, 'IDwDs')
if_core.SetDataObject('d_customer')

MessageBox('DataStore', f_GetDWInfo(if_core))
```

It works! We can call f_GetDWInfo with a DataWindow, a DataStore, or a DataWindowChild. Figure 5 shows a possible example for a DataWindowChild.

The source code is from an example that's available for download. I've chosen a simple one, but it still shows the power of interfaces.

Summing It Up

Using the implementation style for interfaces proposed in Part 1, I designed interfaces for common functions of DataWindows, DataStores, and DataWindowChild objects. Using those interfaces enabled me to write pieces of code that work with any of the three objects. ▼

Author Bio

Roland Mühlberger works as a PowerBuilder class librarian and software engineer for the Austrian company ecosys. In addition, he runs his own business (ROMU Software) as an independent consultant. His special interest (besides mountain climbing) are programming tools; he's the author of SmartPaste, a tool for documenting PB source code.

aricr@arconsultinginc.com

Listing 1

```
String s_ret

s_ret  = 'ClassName:       ' + aif.ClassName() + '~r~n'
s_ret += 'Rows:            ' + String(aif.RowCount()) +
'~r~n'
s_ret += 'Deleted Rows:    ' +
String(aif.DeletedCount()) + '~r~n'
s_ret += 'Filtered Rows:   ' +
String(aif.FilteredCount()) + '~r~n'
s_ret += 'Current Row:     ' + String(aif.GetRow()) +
'~r~n'
s_ret += 'Current Column: ' +
String(aif.GetColumnName()) + '~r~n'
s_ret += 'Modified Rows:   ' +
String(aif.ModifiedCount()) + '~r~n'
s_ret += 'Current Zoom:    ' + aif.Describe('datawin-
dow.zoom') + '~r~n'
s_ret += 'First Line:      ' + aif.Describe('datawin-
dow.firstRowOnPage') + '~r~n'
s_ret += 'Last Line:       ' + aif.Describe('datawin-
dow.lastRowOnPage') + '~r~n'

RETURN s_ret
```