Using Interfaces with PowerBuilder

Implementing the concept

PLEASE
NEED
PHOTO
HI RES

WRITTEN BY
Roland Mühlberger

Interfaces are one of the most important concepts that have found their way into modern software engineering in the past couple of years. Java, COM, CORBA, C# – all those languages/concepts support interfaces. PowerBuilder does not. This article shows you how to sneak the interface concept into PB by using some of its special features.

So Many Interfaces – What This Article Is Not About

The term *interface* is a common one in software engineering. In this article, I define interface to mean "named collection of method definitions." We especially don't deal with:

- The PBNI, the PowerBuilder native interface, which will be a part of PB9 and allows C++ classes to call into PB
- User interfaces – all kinds of GUIs like Swing, the Windows GUI...

What Are Interfaces?

If you don't know interfaces, the next part should shed some light on the concept. In short, interfaces define a set of functions. If a certain class implements all those functions, it is said to implement the interface. What can we do with those classes? All classes that implement an interface can be treated alike: we can make a list of them and call the methods defined in the interface. Or we can define functions that take objects of a specific interface as arguments and use the methods that are defined for the interface in these functions.

Let's look at implementing interfaces in C#, the successor of C++ for the .NET platform:

```
public interface IPrintable {
  void Print();
}
```

This simply defines an interface called IPrintable with one method called Print. We can have two nonrelated classes in the class hierarchy that both implement this interface (see Listing 1).

Both classes, Person and Account, implement the interface IPrintable. We can implement a function that only works on objects of IPrintable:

```
public static void Print(IPrintable
  ip) {
 ip.Print()
}

{
// main program.
// ...

// create new object and call static
 function Print
Person p = new Person("John","Test");
Print(p);

Account a = new Account("Savings", -
 123.45);
Print(a);
```

We have created two completely different objects but have used the same method to print them out. What's the good thing about this? Interfaces don't bother with inheritance; they only look at which methods a class implements. You can use polymorphism without inheritance. If you don't know what polymorphism is, look it up in any object-oriented programming primer (in short, it's the possibility of having references point to objects of different classes at runtime. The short program is a good example of polymorphism).

Interfaces are somehow related to inheritance, but there are a lot of differences. Table 1 shows you some.

Implementing Interfaces in PB

Lets get to the PB part of it all. How can we get interfaces into PB? It doesn't support them natively, so we have to find a different approach. PB has one feature, called "DYNAMIC", that allows us to call functions or events on variables that don't support those methods at compile time. Here's how it works:

```
PowerObject po
po = this.of_GiveMeAnObject()
po. DYNAMIC Print()
```

The short script shows us a local variable of static type PowerObject that gets assigned an object reference. We can't see which object po points to after this call. It might be an object of class PowerObject; it might also be an object of class DataStore (which is inherited from PowerObject). If we didn't code the "DYNAMIC", PB would react with "unknown function name Print()" at compile time. But, by stating this call to be dynamic, we tell PB: "Trust me, during runtime po will point to an object that has the function Print."

We use this dynamic call to implement an interface "IPrintable". Since we want to make a lot of interfaces, we should have created a base class for all interfaces called "IBase" first. IBase is inherited from PowerObject, which you can do in PB by simply creating a new "custom class object." We inherit IPrintable from IBase and then we implement the interface methods. Our interface will be simple as we have only one function called "Print" that we

INHERITANCE	INTERFACE
Classes have exactly one base class (apart from multiple inheritance)	Classes can have multiple interfaces
Classes are part of the class hierarchy	Interfaces are not related to the class hierarchy
Classes implement methods	Interfaces only define methods
Classes can have any name	Interfaces usually start with an uppercase "I"
Classes consist of methods and attributes	Interfaces define of methods and attributes

TABLE 1 ?????????????????

need to implement. We create that public function, giving it a return type of Integer with no parameters. If you look at the source code (available on at www.sys-con.com/pbdj/source.cfm), the function will be defined like this:

```
public function integer print ();
```

How do you implement this function? Because we were talking about dynamic calls, the content should be something like:

```
RETURN baseObj. DYNAMIC Print()
```

What is this baseObj? It's the reference to the object that implements the interface. We need a place to define this base object, and we use our base class IBase to define the according methods: SetObject and GetObject:

1. IBase gets a new private instance variable called ipo_base of class PowerObject:

```
Private:
PowerObject ipo_base
```

2. Base gets a new function called SetObject that takes a PowerObject called apo_base as an argument and assigns it to ipo_base:

```
ipo_base = apo_base
```

3. IBase gets another function called GetObject that returns PowerObject. The implementation is simply:

```
RETURN ipo_base
```

Having done that, we can implement our Print function in IPrintable as follows:

```
RETURN ipo_base. DYNAMIC Print()
```

A big part of what we wanted to achieve is already done; however, it's still far from perfect. We can now do something similar to the following:

```
Iprintable ip
DataWindow dw
DataStore ds
dw = this.of_GetDW()
ds = this.of_GetDS()
ip = CREATE IPrintable
ip.SetObject(dw)
ip.Print()
ip.SetObject(ds)
ip.Print()
DELETE ip
```

In addition, we can create classes that take references to IPrintable objects and use their Print Method.

What could be improved:

1. Creating the interface with create is somehow strange. I don't want to create anything; I simply want to use the interface. We need to make it easier to use interfaces.
2. What if our base object doesn't implement the functions stated in the interface? We need to make our code more robust.

To deal with the first issue, we implement a global function CreateInterface. This function creates an object of a given interface class and assigns the base object to it. The function takes two arguments, the reference to a PowerObject called apo_base and a string called is_interface. It yields an object of class IBase, our base interface class:

```
global function ibase createinterface
(powerobject apo_base, readonly string
as_interface);
IBase if_new
// ---
if_new = CREATE USING as_interface
if_new.SetObject(apo_base)
RETURN if_new
end function
```

CreateInterface creates a new object of the given interface class and sets the base object to the object given as an argument to the function. With this enhancement, we can use interfaces similar to:

```
Iprintable ip
DataWindow dw
DataStore ds
dw = this.of_GetDW()
ds = this.of_GetDS()
ip = CreateInterface(dw, 'IPrintable')
ip.Print()
ip = CreateInterface(ds, 'IPrintable')
ip.Print()
```

That doesn't look much nicer than the part before, does it? You can see the enhancement if you imagine a global function called PrintMe that takes one argument of class IPrintable:

```
global function integer printme
(iprintable aif_print);
RETURN aif_print.Print()
end function
```

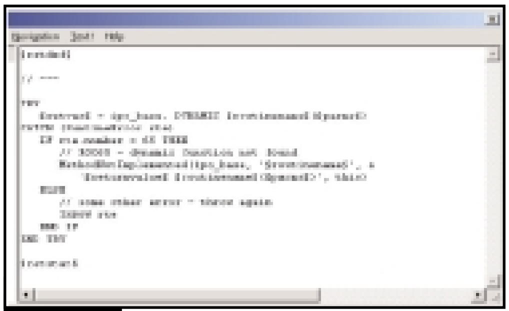
With such a function we could code our example as:

```
DataWindow dw
DataStore ds
dw = this.of_GetDW()
ds = this.of_GetDS()
PrintMe(CreateInterface(dw,
'IPrintable'))
PrintMe(CreateInterface(ds,
'IPrintable'))
// we leave the interface objects for
garbage collection...
```

As you can see, there's no need for a local variable of the interface class anymore. In this case, we can't destroy the interface objects; we leave them for the PB garbage collector.

If all these examples still don't convince you, the following provides an example of polymorphism with the use of interfaces – define a new class PrintableList and implement the two methods shown in Listing 2.

PrintableList takes objects of class (or should we say interface?) IPrintable and manages them in an open array. Now, if you call PrintAll, the according method Print of all the objects added to the list gets called. The objects encapsulated in the interface objects don't need to have any inheritance relationship (for instance, DataStore and DataWindow and also DataWindowChild). You can fill the list with interfaces to any objects that implement the function Print and it will work!



```
[illegible]

// ---

TRY
   [illegible] = ipo_base. DYNAMIC [illegible]
CATCH ([illegible])
   IF rte.number = 65 THEN
      // ERROR - dynamic function not found
      MethodNotImplemented(ipo_base, '[illegible]', &
         '[illegible]', this)
   ELSE
      // some other error - throw again
      THROW rte
   END IF
END TRY

[illegible]
```

FIGURE 1 An interface implementation SmartPaste macro

I mentioned a second issue that could be improved. The Print method is not robust enough – if we call it on an object that doesn't implement the method, PB simply gives runtime error 65, which means dynamic function not found.

By using the exception handling introduced in PB8, we can alleviate the problem. Listing 3 provides a more robust implementation of the Print function within our interface IPrintable.

We enclose the call to Print with a TRY-CATCH block. If a runtime error occurs, it's caught. If the error code is 65, we use the global function MethodNotImplemented to show the error, otherwise the exception is thrown again (that means an error occurred within Print).

As I mentioned earlier, this works for PB from version 8 up. Unfortunately, there's no way to "catch" this error in earlier versions of PB, because PB stops execution if it comes across runtime error 65 (even trying to handle the error in the application SystemError does not work).One alternative is to use runtime type information by querying the classDefinition property of the base object, but this is more complex than the version presented and is rather slow, so I won't go into more detail about it here.

Smart Work

Implementing numerous interface functions can become quite tedious, and because the basic layout is the same for all functions, you can use templates or predefined scripts to help with your work. Another possibility is to use SmartPaste with a macro defined as in Figure 1.

For functions that have the signature:

```
public function integer moveto (inte-
ger ai_x, integer ai_y)
```

SmartPaste will produce the code in Listing 4 by using the above macro.

If you don't know SmartPaste, it's a shareware tool usually used to create header comments. It can be downloaded for free from www.romu.com/smartpaste.

More Enhancements

There is one method that can make using interfaces less error-prone: for every class that implements the methods of an interface, implement a new event that's called the same way an interface is. That event doesn't need arguments or a return value. Then we can use the enhanced version of the global function CreateInterface, called CreateCheckedInterface, to create an interface:

```
IBase if_new
// ---
IF IsValid(apo_base) THEN
  IF apo_base.TriggerEvent(as_inter
   face) = 1 THEN
    if_new = CREATE USING as_inter
     face
    if_new.SetObject(apo_base)
  ELSE
    InterfaceNotImplemented(apo_base,
     as_interface)
  END IF
END IF
RETURN if_new
```

This piece of PowerScript relies on the fact that TriggerEvent returns 1 if the event provided as an argument is implemented (apart from executing the code of the event, of course). If the event does not exist, the interface is apparently not implemented, therefore a function to deal with this issue is triggered.

When working with interfaces it's a good idea to know whether a certain object is of a class that implements the interface. With the event implementation stated earlier, we can write an interface test HasInterface:

```
global function boolean hasinterface
(powerobject apo, readonly string
as_interface);
Boolean b_ret
// ---
IF IsValid(apo) THEN
b_ret = (apo.TriggerEvent(as_inter-
face) = 1)
END IF
RETURN b_ret
end function
```

Summing It Up

There is a way to implement the concept of interfaces in PB. The main idea behind it is to use a wrapper interface class and call the interface methods with a dynamic call. In PB 8 and above the encapsulation of the call can be made robust by using exception handling. The proposed concept doesn't support instance variables as part of the interface; they need to be encapsulated by functions.

Note: In the case of an error, PB8 does not behave the same within the IDE and during runtime. It shows its own messagebox with the system error when starting the program within the IDE, and in the executable it doesn't. PB9 programs (if you're a beta participant) work okay when started from within the IDE. They still show the messagebox when debugging.

All the source code for this article can be downloaded from www.romu.com/interfaces. ▼

office@romu.com

Author Bio

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

Listing 1

```
public class Person: IPrintable {
  public string firstName;
  public string lastName;
  // … any other needed methods
  public void Print() {
    Console.WriteLine(lastName + ", " + firstName);
  }
}
public class Account: IPrintable {
  public string acctName;
  public double currentAmount;
  // … any other needed methods

  public void Print() {
    Console.WriteLine("Account " + acctName + ":" +
     currentAmount);
  }
}
```

Listing 2

```
public subroutine of_printall ();
Long l_cur, l_count
l_count = UpperBound(if_printables[])
FOR l_cur = 1 TO l_count
   if_printables[l_cur].Print()
NEXT
end subroutine

public function long of_addobject (iprintable aif_print);
Long l_count
l_count = UpperBound(if_printables[])
l_count ++
if_printables[l_count] = aif_print
RETURN l_count
end function
```

Listing 3

```
Integer retVal
// ---
TRY
   i_ret = ipo_base. DYNAMIC Print()
CATCH (RuntimeError rte)
   IF rte.number = 65 THEN
     // R0065 - dynamic function not found
     MethodNotImplemented(ipo_base, 'print', 'integer
       print()', this)
     i_ret = -1
   ELSE
     // some other error - throw again
     THROW rte
   END IF
END TRY
RETURN retVal
```

Listing 4

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

Download the Code!

The code listing for this article can also be located at www.sys-con.com/pbdj/