Mix-In It Up In Delphi

by David Baer

Issue 32 of *The Delphi Magazine* presented an article by Marco Cantù that explored the use of Delphi interfaces to effect multiple inheritance. Although the primary motivation of Inprise's provision of interfaces was to open Delphi up to work with Microsoft's Component Object Model (COM), Marco demonstrated a very useful alternative application of this feature.

We'll explore the topic further in this article. We'll look at grafting interfaces on to existing VCL components, and we'll see just how far away from COM we actually remove ourselves in the process.

Whereabouts IUnknown

Let's begin with a brief review of how we've arrived here and why this technology can be important outside the context of COM. C++, of course, has long supported multiple inheritance (MI) in its object model. But even some of the most zealous of C++ advocates have cautioned that the use of MI is generally *a bad thing*, a probable ticket to code maintenance hell.

However, one form of MI has been generally blessed. If all but the principal base class in an MI inheritance structure is completely abstract (ie contains no data and has only methods defined as the equivalent of Delphi abstract methods), the benefits of the considerable flexibility and power of MI are provided, but the pitfalls are avoided. This style of MI coding is often referred to as using 'mix-in' classes.

One of the goals of the designers of the Java language was to invent a medium of expression that had most of the power of C++, but which jettisoned those problematic language elements that rendered C++ code obtuse and error prone. To that end, mix-in class capability was provided in the form of (you guessed it) interfaces.

Meanwhile COM was evolving (cynics might say mutating) at Microsoft. In COM, objects are *only* accessed via interfaces. COM interfaces play much the same role as they do in Java. However, in COM, the rules for interacting with objects are quite complex and very different from object interaction in conventional OO languages. Programming to the COM specification at the actual API level is an extraordinarily challenging activity.

Although much has been made of the inadequacies of COM as an object model, it is nevertheless an extremely important and widely used technology. Inprise provided robust support for COM in Delphi 3 and they are to be congratulated on their solution, which manages to make COM integration almost painless, a rather incredible feat of software engineering. The Delphi approach involves the provision of compile-time services which hide many of the tedious details of COM interaction, and extensive runtime support from a number of VCL classes. Delphi 3 also defined a new syntactic construct called the `interface` to support Delphi/COM interaction, and that syntax is a very natural addition to the existing language. Although Delphi 3 interfaces are decidedly COM-centric, an interesting question arises as to how useful they might be outside the world of COM.

Don't Count On It

All interfaces in both Delphi and COM derive from a single standard ancestor interface, `IUnknown`. Even if we wish to avoid COM in our pursuit of mix-in capabilities, we can't avoid `IUnknown`. Delphi dictates that our interfaces *must* descend from it. When a class implements an interface, then it (or one of its ancestor classes) must provide an implementation of the `IUnknown` methods.

`IUnknown` defines three methods, two of which, `_AddRef` and `_Release`, concern themselves with referencing counting (we will get to the third one in a bit). COM classes always sport multiple interfaces, sometimes a fairly imposing number of them. COM objects are required to count the number of outstanding references to interfaces. As references to interfaces are obtained the count increments, as they are abandoned the count decrements. When it hits zero, the object may be destroyed.

However, programming COM at the low API level involves a good deal of very picky placement of `_AddRef` and `_Release` calls for things to work properly. This is where the Delphi compiler gets into the act. The compiler automatically generates calls to the counting routines as appropriate. When interfaces go out of scope, for example, reference counts are decremented. Referencing counting is a pervasive technique found in many places beside COM. Those familiar with the reference counting mechanisms of Delphi long strings will already understand how this game is played.

The Delphi VCL supplies a base class called `TInterfacedObject` (in the `System` unit) that inherits from `TObject` and includes the `IUnknown` interface. `TInterfacedObject` is of little use itself, but it serves as a base class for increasingly complex COM classes. Recall that interfaces include only the method definitions, but implement no code. Thus, `TInterfacedObject` must supply the implementation of the `IUnknown` methods, and it is instructive to examine these which are shown in Listing 1. As this has a fundamental placement in the class hierarchy, we can assume that it's a fairly 'boilerplate' implementation. This is also the first dilemma we encounter in trying to use interfaces for non-COM purposes.

`_AddRef` couldn't be much simpler. It increments `FRefCount`, which is the private variable `TInterfacedObject` object's reference

```
function TInterfacedObject._AddRef: Integer;
begin
  Inc(FRefCount);
  Result := FRefCount;
end;
function TInterfacedObject._Release: Integer;
begin
  Dec(FRefCount);
  if FRefCount = 0 then begin
    Destroy;
    Result := 0;
    Exit;
  end;
  Result := FRefCount;
end;
```

➤ *Listing 1*

```
Type
  IAnInterface = interface(IUnknown)
    procedure AnInterfaceMethod;
  end;
  TMyClass = class(TAClass, IAnInterface)
  ...
var
  SomeObject: TMyClass;
procedure TMyForm.FormShow(Sender: TObject);
var
  IntRef: IAnInterface;
begin
  SomeObject := TMyClass.Create;
  IntRef := SomeObject;
  IntRef.AnInterfaceMethod;
end;
```

➤ *Listing 2*

```
function TMyObject._AddRef: Integer;
begin
  Result := 0;
end;
function TMyObject._Release: Integer;
begin
  Result := 0;
end;
function TMyObject.QueryInterface const IID: TGUID; out Obj): Integer;
begin
  Result := 0;
end;
```

➤ *Listing 3*

```
IUnknown = interface
  ['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;
```

➤ *Listing 4*

count for interfaces. But observe the logic in `_Release`. It decrements `FRefCount` and, when the value goes to zero, the object is destroyed. This may be perfectly fine if we're using COM object creation methodology, but if we instead wish to create our objects in the standard Delphi way, we've got a problem.

Consider the innocuous code in Listing 2. A problem arises because Delphi is observing that a reference count increment is needed when `IntRef` is assigned a value from `SomeObject`. Upon exit of the routine, `IntRef` goes out of scope and the compiler again helpfully generates an implicit call to `_Release`. Whoops! The code doesn't make it appear that we've freed our newly created `SomeObject`, but that is just what's happened. So here is our first departure from the boilerplate code.

We have two choices. The first is to fool the reference counting mechanism by calling `_AddRef` right in our constructor which allows us to always have a reference count of at least one. The second is to just ignore the whole business (there's no COM here, we don't need no stinkin' reference counts). This is a slightly cleaner solution, as the code in Listing 3 demonstrates.

GUID Riddance!

Now we get to the third method of `IUnknown`, `QueryInterface`. Unfortunately, there is no easy way to discuss `QueryInterface` without getting into some pretty complex aspects of COM. The good news is that we don't necessarily need to explain it. Once again, Delphi allows us to ignore the whole thing, if we are content to sacrifice a dash of polymorphic capability.

The COM landscape is scattered with unreadable names which look like this:

```
['{667337A0-D3C5-11D1-8914-
    4746535402305}']
```

These are known as GUIDs (Globally Unique Identifiers), which are claimed to be statistically unique across space and time (and as Julian Bucknall hasn't chosen to challenge this bold assertion, I'm certainly not about to). Every thing that can be named in COM has its own GUID, although more specific designations are usually used (CLSIDs being Class Identifiers and IIDs being Interface Identifiers, for example).

So, it can be no surprise that COM interfaces need identifiers, and the Delphi interface declaration syntax accommodates this. Listing 4 presents Delphi's own declaration of `IUnknown`. This is where things get a bit puzzling. The Delphi documentation clearly shows that the interface identifier is *optional*. But it is mute on the rules and consequences of not including one. Clearly, we can expect that such an omission would not be welcomed in the COM world. But for what do we need a GUID in our non-COM quest?

Before we get to that answer, we need to examine another aspect of the Delphi language implementation of interfaces. First of all, unlike

the invocation of mix-in class methods in C++, which is done via an object reference, Delphi interface methods may only be executed using an interface reference. A similar requirement exists in Java, by the way.

So, how do we get an interface reference? Rather simply, it turns out. One may declare interface references just as one declares object references. Assigning an object reference to this interface reference will provide the appropriate value (assuming that the types are compatible). This type compatibility is known at compile-time, and the compiler enforces it. Listing 5 illustrates this and some of the following points.

In COM, object clients never deal with direct object references, only interface references. The COM contract states that any interface of an object must be obtainable from any other interface supported by the object. That is the job of `QueryInterface`: to provide this service.

In Delphi, we need not call `QueryInterface`, however. This is where the compiler again offers assistance. Delphi utilizes the `as` operator for this purpose (again, see Listing 5). Delphi translates `I2 as I1` into a call to `QueryInterface`. In this case, the type compatibility is not known at compile-time. A type mismatch results in a runtime exception. But what happens if you've not supplied a GUID in your interface definition? The compiler produces the following not-so-helpful diagnostic: *Operator not applicable to this operand type*. It turns out that Delphi will allow use of the `as` operator *only* when GUIDs are specified. Mystery solved!

So, the final thing to attend to is the implementation of `QueryInterface`. If we omit the GUIDs in our interface declaration, the implementation can be anything, since the compiler won't allow `as` operators on the interfaces. As a result, implicit calls to `QueryInterface` will never be made. Listing 3 shows a minimal implementation. A slightly better one might include raising an exception, just in case another developer later decides to add GUIDs, but isn't aware of the original intention.

```
IInterface1 = interface(IUnknown)
  ['{7FB79D60-D6CB-11D1-8914-444553540000}']
  procedure Method1;
end;
IInterface2 = interface(IUnknown)
  ['{7FB79D61-D6CB-11D1-8914-444553540000}']
  procedure Method2;
end;
TSomeClass = class(TObject, IInterface1, IInterface2)
public
  procedure SomeMethod;
  ...
var
  SC:   TSomeClass;
  I1:  IInterface1;
  I2:  IInterface2;
begin
  SC := TSomeClass.Create;
  SC.Method1;     // compile error, object can't execute
                  // interface method
  I1 := SC;       // get IInterface1 reference from object
  I2 := SC;       // get IInterface2 reference from object
  SC.SomeMethod;  // no problem
  I1.Method1;     // no problem
  I2.Method2;     // no problem
  I2 := I1 as I2; // OK as long as GUIDs are present
                  //  otherwise compile error
...
```

➤ *Listing 5*

```
IKnowsObject = interface(IUnknown)
  function ObjectOfInterface: TObject;
end;
TSomeClass = class(TObject, IKnowsObject)
  ...
function TSomeClass.ObjectOfInterface: TObject;
begin
  Result := Self;
end;
```

➤ *Listing 6*

Naughty, Naughty

So, we've seen what the GUID buys us: access to the `as` operator on interfaces. Lack of this may be unacceptable in some circumstances, but will prove no great deficiency in many others. But there's one more thing we *can* do here, and it's something that would be no less than a criminal act in COM. We can have any of our interfaces produce an object reference. COM objects only reveal themselves through interfaces, but in a Delphi-only context, we needn't be bound by this restriction. Listing 6 shows an implementation of this technique.

Interfacing Reality

If the discussion thus far has failed to convince you that interfaces are a marvelous addition to the language, well, I'm not surprised. This has been a lot of rather dry explanation. But be patient. It's time to look at this capability in a somewhat realistic setting. Figure 1 shows the single form used for the demonstration project, which can found on this issue's companion disk. If you wish to take if for a test drive, there is nothing to install on the component palette. All items are classes, and you may compile and run the project with little ado.

Imagine that Object Pascal had some form of MI from its inception. Do you think the engineers devising the technology for data-aware components would have considered it? I confess I haven't done the analysis to suggest that use of interfaces would be superior to the solution we've grown to know and appreciate. But I believe that the alternative would have at least merited serious consideration in the design stage. To that end, the sample code used to demonstrate these concepts is a modest analogue to data sources and data-aware components.

For the server (data source), I've defined the class `TStringArray`. It's much like a `TStringList` except that

it has, at any given time, a fixed dimension it maintains, a current row index and it supports an Assign function compatible with any TStrings class. But most significantly, it serves components which implement the IStringArrayClient interface. For clients, I've defined classes based on three stock Delphi components: TEdit, TListBox and TTrackBar, each of which implement the IStringArrayClient interface. Each also has a property, StringArray, used to specify the server.

The TStringArray is somewhat like a data source in that it can serve a number of diverse clients at one time. The beauty of using interfaces to effect this solution is that we may add additional client types at any time without either TStringArray or other existing client classes needing to be modified. Listing 7 presents the declaration of TStringArray. TStringArray has three properties, none of them too mysterious. The Count property is read/write and specifies the number of elements. Increasing Count causes additional rows with empty string values to be added, decreasing it discards any strings beyond the upper bound. The Current read/write property is the current 'row' index. If Count is zero, Current will be -1, otherwise it will always be between 0 and Count -1. Finally, Strings, predictably the default read/write property, gives access to the contents of the array.

```
TSAChangeType = (sacSingleItemChange,
      sacCurrentChange,
      sacMajorChange,
      sacClosingDown);
IKnowsObject = interface(IUnknown)
  function ObjectOfInterface: TObject;
end;
ISt*ringArrayClient = interface(IKnowsObject)
  procedure StringArrayChange(ChangeType: TSAChangeType);
end;
TStringArray = class(TPersistent)
private
  FCurrent:         Integer;
  ClientList:       TList;
  StrList:          TStringList;
  UpdateCount:      Integer;
private
  function GetCount: Integer;
  function Get(Index: Integer): String;
  procedure NotifyClients(ChangeType: TSAChangeType);
  procedure Put(Index: Integer; const Value: String);
  procedure SetCount(Value: Integer);
  procedure SetCurrent(Index: Integer);
public
  constructor Create;
  destructor Destroy; override;
  procedure Assign(Source: TPersistent); override;
  procedure AssignTo(Dest: TPersistent); override;
  procedure BeginUpdate;
  procedure Clear;
  procedure EndUpdate;
  procedure RegisterClient(C: IStringArrayClient);
  procedure UnregisterClient(C: IStringArrayClient);
public
  property Current: Integer read FCurrent write SetCurrent;
  property Count: Integer read GetCount write SetCount;
  property Strings[Index: Integer]: String read Get
    write Put; default;
end;
```

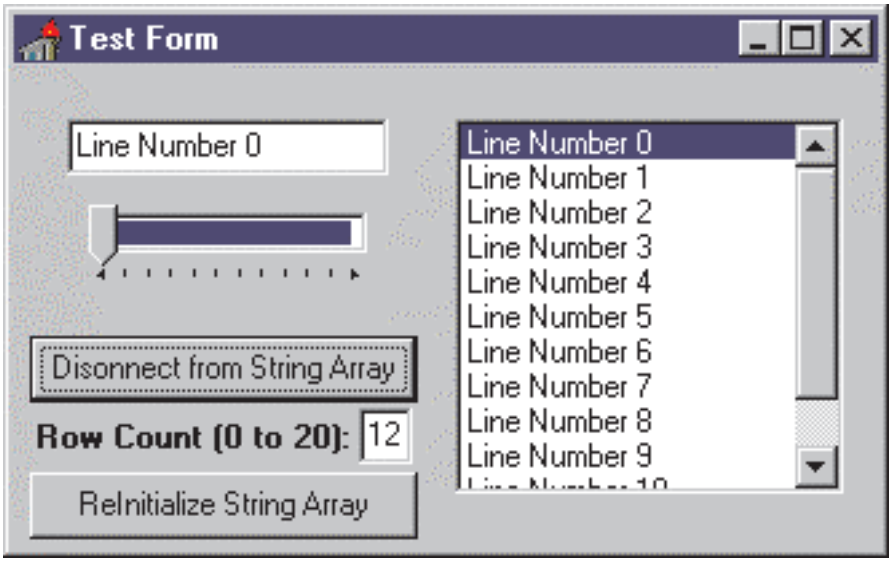
➤ *Listing 7*

TStringArray is derived from TPersistant, not because it is intended to be streamable, but for another important reason (which is off subject, but well worth taking a short detour to mention). By using TPersistant as the base class, we have the Assign/AssignTo turnaround of TPersistant at our disposal. This permits TStringArray and TStrings objects to interact with via the Assign method. Space does not permit a full explanation, but many of you will be able to discover what's going on by examining the code in the short TPersistant class definition (in the Classes unit). For now, let me just offer the opinion that there has never been a more compelling, yet economical, demonstration of the power of polymorphism. For my money, this is as close to poetry as program code can get.

For further information, I heartily recommend reading *Delphi Component Design* by Danny Thorpe of Inprise.

But back to the business at hand. It's time to discuss client interaction responsibilities of TStringArray. Client controls may be totally passive, simply reflecting content or position changes in the server. However, they may also be agents of change to either of those things. For example, a read-only edit control client attached to a string array will simply present the contents of the current row (or no text, if Count is 0). But if the edit control is not read-only, then changes in its text must be communicated to the string array. When a client is an agent of change, it's the responsibility of that client to appropriately communicate modifications to the server. When this occurs, all of the other clients must be notified that something has happened as well. This becomes the responsibility of the server.

➤ *Figure 1*



Our server, being a generally helpful one, gives clients a bit more information than 'something's different'. There are three state changes communicated, not all of which will be of interest to all clients: `Current` has changed, the contents of one element has changed, or something major has happened. Our track bar client, for example, will have no interest in the fact that the contents of an element of the array have changed. It's interested in the number of elements and the current element index. All clients will be interested in a major change, which happens when multiple element values are changed or when `Count` changes. Finally, we add one additional state change: the server's shutting down.

Service With A Smile

Let's look at how the state changes are communicated.

It's really quite simple. `Clients` must implement the `IStringArrayClient` interface, which has but one method: `StringArrayChange`. Its sole parameter is the type of change `TStringArrayChangeType`. But before we focus on state changes, let's briefly examine how clients connect and disconnect.

The implementation of `TStringArray` is presented in Listing 8. Two methods, `RegisterClient` and `UnregisterClient`, take care of client connection/disconnection. Both methods have a single parameter of type `IStringArrayClient`, which is an interface reference. These simply add or delete the interface reference from an internal `TList`. Note that the parameter variables need to be

➤ *Listing 8*

```
procedure TStringArray.Assign(Source: TPersistent);
var
  I:   Integer;
  Max: Integer;
begin
  if (Source is TStrings) or
    (Source is TStringArray) then begin
    BeginUpdate;
    try
      Max := TStrings(Source).Count;
      if StrList.Count < Max then
        Max := StrList.Count;
      for I := 0 to (Max - 1) do
        if Source is TStrings then
          StrList[I] := TStrings(Source)[I]
        else
          StrList[I] := TStringArray(Source)[I];
      for I := Max to (StrList.Count - 1) do
        StrList[I] := '';
    finally
      EndUpdate; // which calls NotifyClients
    end;
  end else
    inherited Assign(Source);
end;
procedure TStringArray.AssignTo(Dest: TPersistent);
begin
  if Dest is TStrings then begin
    Dest.Assign(StrList);
    Exit;
  end;
  inherited AssignTo(Dest);
end;
procedure TStringArray.BeginUpdate;
begin
  Inc(UpdateCount);
end;
procedure TStringArray.Clear;
begin
  if StrList.Count <> 0 then begin
    StrList.Clear;
    FCurrent := -1;
    NotifyClients(sacMajorChange);
  end;
end;
constructor TStringArray.Create;
begin
  inherited Create;
  ClientList := TList.Create;
  StrList := TStringList.Create;
  FCurrent := -1;
end;
destructor TStringArray.Destroy;
var
  I:  Integer;
begin
  for I := ClientList.Count - 1 downto 0 do
    IStringArrayClient(
      ClientList[I]).StringArrayChange(sacClosingDown);
  ClientList.Free;
  StrList.Free;
  inherited Destroy;
end;
procedure TStringArray.EndUpdate;
begin
  Dec(UpdateCount);
  if UpdateCount = 0 then
    NotifyClients(sacMajorChange);
end;
function TStringArray.GetCount: Integer;
begin
  Result := StrList.Count;
end;
function TStringArray.Get(Index: Integer): String;
begin
  Result := StrList[Index];
end;
procedure TStringArray.NotifyClients(ChangeType:
  TSAChangeType);
var I : Integer;
begin
  if UpdateCount = 0 then
    for I := 0 to (ClientList.Count - 1) do
      IStringArrayClient(
        ClientList[I]).StringArrayChange(ChangeType);
end;
procedure TStringArray.Put(Index: Integer;
  const Value: String);
begin
  if StrList[Index] <> Value then begin
    StrList[Index] := Value;
    NotifyClients(sacSingleItemChange);
  end;
end;
procedure TStringArray.RegisterClient(
  C : IStringArrayClient);
begin
  { ShowMessage('Connecting ' +
    C.ObjectOfInterface.ClassName); }
  ClientList.Add(Pointer(C));
  C.StringArrayChange(sacMajorChange);
end;
procedure TStringArray.SetCount(Value: Integer);
var
  I: Integer;
begin
  if Value <> StrList.Count then begin
    if Value < StrList.Count then
      for I := (StrList.Count - 1) downto Value do
        StrList.Delete(I)
    else
      while (StrList.Count < Value) do
        StrList.Add('');
    if (FCurrent = -1) and (Value > 0) then
      FCurrent := 0
    else if (FCurrent <> -1) and (Value = 0) then
      FCurrent := -1
    else if FCurrent > (Value - 1) then
      FCurrent := Value - 1;
    NotifyClients(sacMajorChange);
  end;
end;
procedure TStringArray.SetCurrent(Index: Integer);
begin
  if Index <> FCurrent then begin
    if Index < 0 then
      Index := 0
    else if Index > (StrList.Count - 1) then
      Index := StrList.Count - 1;
    FCurrent := Index;
    NotifyClients(sacCurrentChange);
  end;
end;
procedure TStringArray.UnregisterClient(
  C: IStringArrayClient);
begin
  { ShowMessage('Disconnecting ' +
    C.ObjectOfInterface.ClassName); }
  ClientList.Remove(Pointer(C));
end;
```

cast as Pointer. Although Delphi will allow an object reference to be a parameter to a TList.Add, an interface reference is disallowed. But the cast removes the compiler's objection, and everything works satisfactorily.

It's expected that the clients themselves will call these methods when they are ready to connect or disconnect (we'll see how later). But the server can do one thing helpful to a newly connected client. As its first service, it can invoke its interface's method StringArrayChange with a change type of sacMajorChange. This lets the client react in its normal fashion to a major change incident.

Other incidents that require a change notification to be sent are a change in any of the three properties. For example, setting a Strings value triggers execution of StringArrayChange with change type sacSingleItemChange. This can be seen in method Put. The notification is done by calling NotifyClients, which does so for each client in the list. When Count changes, then this warrants a NotifyClients with change type sacMajorChange. A new value assigned to Current gets the clients a sacCurrentChange notification. Finally, if the server is about to destruct, it notifies any clients of that fact, so that they may disconnect.

One last detail: notice the BeginUpdate and EndUpdate methods. These serve the same function as like-named methods in a number of other VCL classes. They allow notifications to clients to be suspended during a period of upheaval. A listbox client does not need to be informed of every element change during a Clear operation, but only at the end. Like other Begin/EndUpdates, these are nestable. Naturally, when an EndUpdate returns the nesting level to 0, the server appropriately issues a notification with type sacMajorChange.

The Client

We'll undertake a thorough examination of only one of the three client classes. The TSAListBox class gets involved in everything except a value change to a string array element, so it provides us with a serviceable specimen to dissect. Study of the other two client classes is left as an exercise to the interested reader. First, let's dispense with the code common to all three (and to any additional client class types added later). These are the implementations of the three methods of IUnknown, the single method of IKnowsObject, and the method that's involved in connecting and disconnecting to the string array. Listing 9 presents these methods.

```
procedure TSAListBox.SetStringArray(SA: TStringArray);
begin
  if SA <> FStringArray then begin
    if SA = nil then begin
      FStringArray.UnregisterClient(Self);
      FStringArray := nil;
      Clear;
    end else begin
      if FStringArray <> nil then
        FStringArray.UnregisterClient(Self);
      FStringArray := SA;
      FStringArray.RegisterClient(Self);
    end;
  end;
end;
function TSAListBox._AddRef: Integer;
begin
  Result := 0;
end;
function TSAListBox._Release: Integer;
begin
  Result := 0;
end;
function TSAListBox.QueryInterface(const IID: TGUID; out Obj):
  Integer;
begin
  Result := 0;
end;
function TSAListBox.ObjectOfInterface: TObject;
begin
  Result := Self;
end;
```

➤ *Listing 9*

As promised earlier, _AddRef, _Release and QueryInterface are minimal placeholders. However, forgetting to include them is a mistake. But (you say) how could one forget? Surely the compiler will notice their absence and flag the omission. It certainly would do so except for an easily overlooked detail. TComponent implements these methods, in spite of the fact that neither TComponent nor its two ancestors includes the IUnknown interface in their declarations. They are there to assist Delphi in working its COM integration magic, but they are definitely not what we want here. We must hide them with our own implementations, or nasty things will happen at runtime. Also, ObjectOfInterface is like the example shown earlier. The demonstration code does not actually use the method except in several blocked lines of code which may be uncommented by the truly curious (or skeptical).

The final common method is SetStringArray, which is the write method for the class's StringArray property. Most of the code here will be common to all string array clients. But the processing required when the property is set to nil will vary according to what's appropriate for the control type. For the listbox, clearing the contents is all that is needed. The remainder of the code is involved in calling RegisterClient and UnregisterClient on the string array.

Now we come to the StringArrayChange method. This needs to reflect the particular qualities of the client. In the case of the listbox, we need to update a single line for notification type sacSingleItemChange. For a sacCurrentChange, we need to set the ItemIndex property. For sacMajorChange, we just reload the contents with an Assign of the StringArray, and then set the ItemIndex. For sacClosingDown, we set the control's own StringArray property to Nil, and that will take care of the disconnect details.

The client listbox must do one more thing. When a new item is selected, the string array must be notified that there is a new Current

value. This is done in a message handler for `CN_COMMAND`.

Wrapping Up

That's about it. The three client classes would require a bit of improvement if they were to be used in a production setting. The track bar, for example, could use some refinement in adjusting the number of ticks for larger numbers of string array elements. Certainly, all three classes would need to be upgraded to full component status. But the basics are all there, and I hope you've been able to appreciate the possibilities.

Interfaces are powerful and elegant, and they make a marvelous addition to Delphi's language. I, for one, wish Inprise would make them a bit more independent of COM. It would be wonderful if there were a way to specify that an interface is intended as Delphi-only (or perhaps letting the absence of a GUID serve this purpose). Such interfaces could descend from an empty interface, and there would be no need to provide dummy implementations of `_AddRef` *et alia*, In fact, the compiler would not even generate implicit calls to these. Additionally, the compiler could handle the `as` operator, since it has all the information needed to compile such casts into native code, and throw the `is` operator back into the works as well. Hmmm.

Dear Santa...

David Baer is Chief Software Architect at Spear Technologies in San Francisco, CA. Having been writing software for over 30 years, he credits Delphi with helping him to avoid the traditional male mid-life crisis. He may be reached at dbaer@speartechnologies.com