COM Corner: Automation Collections

by Steve Teixeira

Let's face it: we programmers are obsessed with bits of software code that serve as containers for other bits of software code. Think about it: whether it's an array, a `TList`, a `TCollection`, a template container class for the C++ folks, or a Java Vector, it seems that we're always in search of the proverbial better mousetrap for software objects that hold other software objects. If you consider the time invested over the years in this pursuit of the perfect container class, it is clear that this is an important problem in the minds of developers. And why not? This logical separation of container and contained entities helps us better organize our algorithms and maps to the real world rather nicely (a basket can contain eggs, a pocket, coins, a parking lot, autos, etc). It seems that, whenever I learn a new language or development model, I have to learn 'their way' of managing groups of entities. Which leads to my point: like any other software development model, COM also has its ways for managing these kinds of groups of entities, and to be an effective COM developer, we must learn how to master these things.

When we work with the `IDispatch` interface, COM specifies two primary methods by which we represent the notion of containership: arrays and collections. If you've done a bit of Automation or ActiveX control work in Delphi, you will probably already be familiar with arrays. You can easily create automation arrays in Delphi by adding an array property to your `IDispatch` descendant interface or dispinterface as shown in Listing 1.

➤ *Listing 1*

```
type
  IMyDisp = interface(IDispatch)
    function GetProp(Index: Integer): Integer; safecall;
    procedure SetProp(Index, Value: Integer); safecall;
    property Prop[Index: Integer]: Integer read GetProp write SetProp;
  end;
```

Arrays are useful in many circumstances, but they pose some limitations. For example, arrays make sense when you have data that can be accessed in a logical, fixed-index manner, such as the strings in an `IStrings`. However, if the nature of the data is such that individual items are frequently deleted, added, or moved, then an array is a poor container solution. The classic example is a group of active windows. Since windows are constantly being created, destroyed and changing z-order, there is no solid criterion for determining in what order the windows should appear in the array.

Collections are designed to solve this problem by allowing you to manipulate a series of elements in a manner that doesn't imply any particular order or number of items. Collections are unusual because there isn't really a collection object or interface, but a collection is instead represented as a custom `IDispatch` that follows a number of rules and guidelines. The following three rules must be adhered to in order for an `IDispatch` to qualify as a collection.

First, collections must contain a `_NewEnum` property that returns the `IUnkown` for an object that supports the `IEnumVARIANT` interface, which will be used to enumerate the items in the collection. Note that the name of this property must be preceded with an underscore, and this property must be marked as `restricted` in the type library. The dispid for the `_NewEnum` property must be `DISPID_NEWENUM (-4)`, and it will be defined as follows in the Delphi type library editor:

```
function _NewEnum: IUnknown
  [propget, dispid $FFFFFFFC,
  restricted]; safecall;
```

Languages such as Visual Basic that support the `For Each` construct will use this method to obtain the `IEnumVARIANT` interface needed to enumerate collection items. More on this later.

Secondly, collections must contain an `Item` method that returns an element from the collection based on the index. The dispid for this method must be 0, and it should be marked with the `default collection element` flag. If we were to implement a collection of `IFoo` interface pointers, the definition for this method in the type library editor might look something like this:

```
function Item(Index: Integer):
  IFoo [propget, dispid $00000000,
  defaultcollelem]; safecall;
```

Note that it is also acceptable for the `Index` parameter to be an `OleVariant` so that an integer, string, or some other type of value can index the item in question.

Thirdly, collections must contain a `Count` property that returns the number of items in the collection. This method would typically be defined in the type library editor as:

```
function Count: Integer [propget,
  dispid $00000001]; safecall;
```

In addition to the above mentioned rules, you should also follow the following guidelines when creating your own collection objects.

First, the property or method that returns a collection should be named with the plural of the name

of the items in the collection. For example, if you had a property that returned a collection of listview items, the property name would probably be `Items`, while the name of the item in the collection would be `Item`. Likewise, an item name called `Foot` would be contained in a collection property called `Feet`. In the rare case that the plural and singular of a word are the same (a collection of fish or deer, for example), the collection property name should be the name of the item with 'Collection' tacked on the end (`FishCollection` or `DeerCollection`).

Next, collections that support adding of items should do so using a method called `Add`. The parameters for this method vary depending on the implementation, but you may want to pass parameters that indicate the initial position of the new item within the collection. The `Add` method normally returns a reference to the item added to the collection.

Lastly, collections that support deleting of items should do so using a method called `Remove`. `Remove` should take one parameter that identifies the index of the item being deleted, and this index should behave semantically in the same manner as the `Item` method.

A Delphi Implementation

If you've ever created ActiveX controls in Delphi, you may have noticed that there are fewer controls listed in the combobox in the ActiveX control wizard than there are on the IDE's component palette. This is because Inprise prevents some controls showing in the list using the `RegisterNonActiveX` function. One such control that is available on the palette but not in the wizard is the `TListView` control found on the `Win32` page of the palette. The reason the `TListView` control isn't shown in the wizard is because the wizard doesn't know what to do with its `Items` property, which is of type `TListItems`. Since the wizard doesn't know how to wrap this property type in an ActiveX control, the control is simply excluded from the wizard's list rather than allowing the user to create an utterly useless ActiveX control wrapper of a control.

However, in the case of `TListView`, `RegisterNonActiveX` is called with the `axrComponentOnly` flag, which means that a descendant of `TListView` will show up in the ActiveX control wizard's list. By taking the minor detour of creating a do-nothing descendant of `TListView` called `TListView2` and adding it to the palette, I can then create an ActiveX control that encapsulates the listview control. Of course, then I am faced with the same problem of the wizard not generating wrappers for the `Items` property and having a useless ActiveX control. Fortunately, ActiveX control writing doesn't have to stop at the code generated by the wizard and I am free to wrap the `Items` property myself at this point in order to make the control useful. As you might be beginning to suspect, a collection is the perfect way to encapsulate the `Items` property of the listview.

In order to implement this collection of listview items, I must create new objects representing the item and the collection, and add a new property to the ActiveX control default interface that returns a collection. I begin by defining the object representing an item, which I will call `ListItem`. The first step to creating the `ListItem` object is to create a new Automation Object using the icon found on the ActiveX page of the New Items dialog. After creating the object, I can fill out the properties and methods for this object in the type library editor.

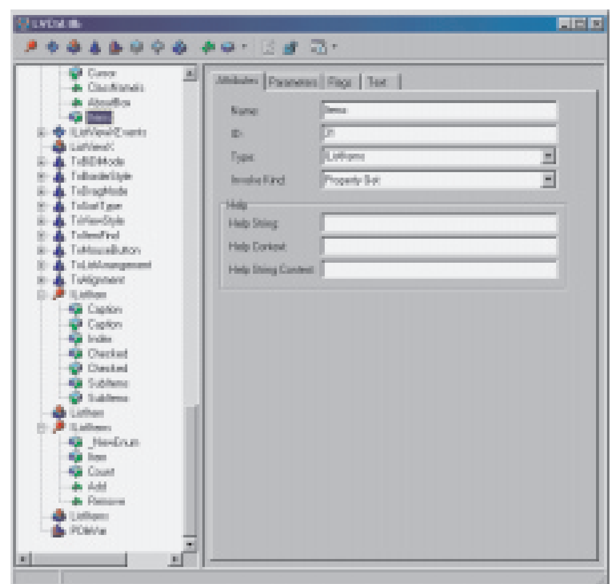
For the purposes of this demonstration, I will add properties for the `Caption`, `Index`, `Checked`, and `SubItems` properties of a listview item. Similarly, I create yet another new Automation object for the collection itself. I call this Automation object `ListItems`, and I provide it with the `_NewEnum`, `Item`, `Count`, `Add`, and `Remove` methods I mentioned earlier. Finally, I add a new property to the default interface of the ActiveX control called `Items` which returns a collection. Figure 1 shows these new interfaces, coclasses, properties and methods in the type library editor.

After my interfaces for `IListItem` and `IListItems` are completely defined in the type library editor, there is a little manual tweaking to be done in the implementation files generated for these objects. Specifically, the default parent class for a new automation object is `TAutoObject`, however, these objects will only be created internally (ie, not from a factory), so I manually change the ancestor to `TAutoInfObject`, which is more appropriate for internally-created automation objects. Also, since these objects won't be created from a factory, I remove from the units the initialization code that creates the factories because it is not needed.

Now that all of the infrastructure is properly set up, it is time to implement the `ListItem` and `ListItems` objects. The `ListItem` object is the most straightforward, since it is a pretty simple wrapper around a listview item. The code for the unit containing this object is shown in Listing 2.

Note that a `ComCtrls.TListItem` is being passed in to the constructor to serve as the listview item to

➤ *Figure 1*



```
unit LVItem;
interface
uses
  ComObj, ActiveX, ComCtrls, LVCtrl_TLB, StdVcl, AxCtrls;
type
  TListItem = class(TAutoIntfObject, IListItem)
  private
    FListItem: ComCtrls.TListItem;
  protected
    function Get_Caption: WideString; safecall;
    function Get_Index: Integer; safecall;
    function Get_SubItems: IStrings; safecall;
    procedure Set_Caption(const Value: WideString);
      safecall;
    procedure Set_SubItems(const Value: IStrings); safecall;
    function Get_Checked: WordBool; safecall;
    procedure Set_Checked(Value: WordBool); safecall;
  public
    constructor Create(AOwner: ComCtrls.TListItem);
  end;
implementation
uses ComServ;
constructor TListItem.Create(AOwner: ComCtrls.TListItem);
begin
  inherited Create(ComServer.TypeLib, IListItem);
  FListItem := AOwner;
end;
function TListItem.Get_Caption: WideString;
begin
  Result := FListItem.Caption;
end;
function TListItem.Get_Index: Integer;
begin
  Result := FListItem.Index;
end;
function TListItem.Get_SubItems: IStrings;
begin
  GetOleStrings(FListItem.SubItems, Result);
end;
procedure TListItem.Set_Caption(const Value: WideString);
begin
  FListItem.Caption := Value;
end;
procedure TListItem.Set_SubItems(const Value: IStrings);
begin
  SetOleStrings(FListItem.SubItems, Value);
end;
function TListItem.Get_Checked: WordBool;
begin
  Result := FListItem.Checked;
end;
procedure TListItem.Set_Checked(Value: WordBool);
begin
  FListItem.Checked := Value;
end;
end.
```

➤ *Listing 2*

be manipulated by this Automation object.

The implementation of the `ListItems` collection object is a little more complex. First, because the object must be able to provide an object supporting `IEnumVARIANT` in order to implement the `_NewEnum` property, I chose to support `IEnumVARIANT` directly in this object. Therefore, my `TListItems` class supports both `IListItems` and `IEnumVARIANT`. `IEnumVARIANT` contains 4 methods (see Table 1).

The source code for the unit containing the `ListItems` object is shown in Listing 3. The only method in this unit with a non-trivial implementation is the `Next` method. The `celt` parameter of the `Next` method indicates how many items should be retrieved. The `elt` parameter contains an array of `TVarArgs` with at least `elt` elements. Upon return, `pceltFetched` (if non-nil) should hold the actual number of items fetched. This method returns `S_OK` when the number of items returned is the same as the number requested or `S_FALSE` otherwise. The logic for this method iterates over the array in `elt` and assigns a `TVarArg` representing a collection item to an element of the array. Note the little trick I'm performing to clear out the `OleVariant` after assigning it to the array. This ensures that the array will not be garbage collected. Were I not to do this, the contents of `elt` could potentially become stale if the objects referenced by V are freed when the Variant is finalized.

Method	Purpose
Next	Retrieve the next n number of items in the collection.
Skip	Skip over n items in the collection.
Reset	Go back to the first element in the collection.
Clone	Create a copy of this IEnumVARIANT.

➤ *Table 1*

Similar to `TListItem`, the constructor for `TListItems` takes a `ComCtrls.TListItems` as a parameter and manipulates that object in the implementation of its methods.

Finally, I complete the implementation of the ActiveX control by adding the logic to manage the `Items` property. First, I must add a field to the object to hold the collection.

```
type
  TListViewX = class(
    TActiveXControl,IListViewX)
  private
    ...
    FItems: IListItems;
end;
```

Then I assign `FItems` to a new `TListItems` in the `InitializeControl` method:

```
FItems :=
  LVItems.TListItems.Create(
    FDelphiControl.Items);
```

Lastly, the `Get_Items` method can be implemented to simply return `FItems`:

```
function TListViewX.Get_Items:
  IListItems;
begin
  Result := FItems;
end;
```

The real test to see whether my collection works is to load the control in Visual Basic 6 and try to use the `For Each` construct with the collection. Figure 2 shows my simple VB test application running.

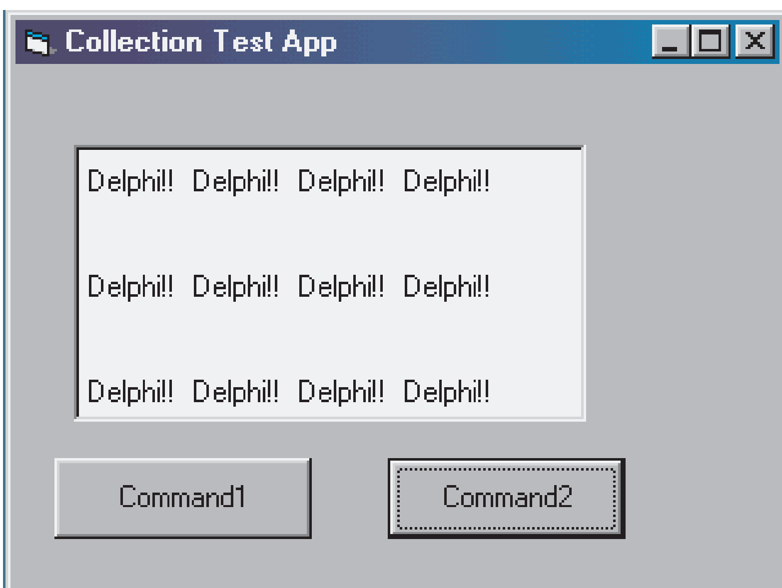
Of the two command buttons you see in Figure 2, `Command1` adds items to the listview, while `Command2` iterates over all of the items in the listview using `For Each` and adds exclamation points to each `Caption`. The code for these methods is shown in Listing 4.

Despite the feelings I know some of the Delphi faithful have toward VB, we must remember that VB is the primary consumer of ActiveX controls, and it's very important to

```
unit LVItems;
interface
uses
  ComObj, Windows, ActiveX, ComCtrls, LVCtrl_TLB;
type
  TListItems = class(TAutoIntfObject,
    IListItems, IEnumVARIANT)
  private
    FListItems: ComCtrls.TListItems;
    FEnumPos: Integer;
  protected
    { IListItems methods }
    function Add: IListItem; safecall;
    function Get_Count: Integer; safecall;
    function Get_Item(Index: Integer): IListItem; safecall;
    procedure Remove(Index: Integer); safecall;
    function Get__NewEnum: IUnknown; safecall;
    { IEnumVariant methods }
    function Next(celt: Longint; out elt;  pceltFetched:
      PLongint): HResult; stdcall;
    function Skip(celt: Longint): HResult; stdcall;
    function Reset: HResult; stdcall;
    function Clone(out Enum: IEnumVariant): HResult;
      stdcall;
  public
    constructor Create(AOwner: ComCtrls.TListItems);
  end;
implementation
uses ComServ, LVItem;
constructor TListItems.Create(AOwner: ComCtrls.TListItems);
begin
  inherited Create(ComServer.TypeLib, IListItems);
  FListItems := AOwner;
end;
function TListItems.Add: IListItem;
begin
  Result := LVItem.TListItem.Create(FListItems.Add);
end;
function TListItems.Get__NewEnum: IUnknown;
begin
  Result := Self;
end;
function TListItems.Get_Count: Integer;
begin
  Result := FListItems.Count;
end;
function TListItems.Get_Item(Index: Integer): IListItem;
begin
  Result := LVItem.TListItem.Create(FListItems[Index]);
end;
procedure TListItems.Remove(Index: Integer);
begin
  FListItems.Delete(Index);
end;
function TListItems.Clone(out Enum: IEnumVariant): HResult;
begin
  Enum := nil;
  Result := S_OK;
  try
    Enum := TListItems.Create(FListItems);
  except
    Result := E_OUTOFMEMORY;
  end;
end;
function TListItems.Next(celt: Integer; out elt;
  pceltFetched: PLongint): HResult;
var
  V: OleVariant;
  I: Integer;
begin
  Result := S_FALSE;
  try
    if pceltFetched <> nil then
      pceltFetched^ := 0;
    for I := 0 to celt - 1 do begin
      if FEnumPos >= FListItems.Count then
        Exit;
      V := Get_Item(FEnumPos);
      TVariantArgList(elt)[I] := TVariantArg(V);
      // trick to prevent variant from being garbage
      // collected, since it needs to stay alive because
      // it is party of the elt array
      TVarData(V).VType := varEmpty;
      TVarData(V).VInteger := 0;
      Inc(FEnumPos);
      if pceltFetched <> nil then
        Inc(pceltFetched^);
    end;
  except
  end;
  if (pceltFetched = nil) or ((pceltFetched <> nil) and
    (pceltFetched^ = celt)) then
    Result := S_OK;
end;
function TListItems.Reset: HResult;
begin
  FEnumPos := 0;
  Result := S_OK;
end;
function TListItems.Skip(celt: Integer): HResult;
begin
  Inc(FEnumPos, celt);
  Result := S_OK;
end;
end.
```

➤ *Above: Listing 3* ➤ *Below: Listing 4*

```
Private Sub Command1_Click()
  ListViewX1.Items.Add.Caption = "Delphi"
End Sub
Private Sub Command2_Click()
  Dim Item As ListItem
  Set Items = ListViewX1.Items
  For Each Item In Items
   Item.Caption = Item.Caption + "!!"
  Next
End Sub
```



➤ *Figure 2*

ensure that our controls function properly in that environment.

Summary

That about covers it, from concept to implementation, to testing. Collections provide powerful functionality that can enable your controls and Automation servers to function more smoothly in the world of COM. I think I also demonstrated that collections are terribly difficult to implement, so it's worth your while to get in the habit of using them when appropriate. If your luck is like mine, once you become comfortable with collections, it's very likely that someone will soon come along and create yet a newer and better container object for COM.

Steve Teixeira is the Director of Software Development at DeVries Data Systems, Inc. If you have questions or ideas for new *COM Corner* articles, please email Steve at steve@dvdata.com