Developing Advanced ActiveX Controls with Visual Basic 5.0

by Daniel Appleman

Portions of this paper are excerpted from the book "Dan Appleman's Developing ActiveX Components with Visual Basic 5.0: A Guide to the Perplexed" published by Ziff-Davis Press. ISBN: 1-56276-510-8.

Author's bio:

Daniel Appleman is the president of Desaware Inc., a developer of add-on products for Microsoft Visual Basic including SpyWorks, VersionStamper, and StorageTools. He is also the author of "Dan Appleman's Visual Basic 5.0 Programmer's Guide to the Win32 API" and "Dan Appleman's Developing ActiveX Components with Visual Basic 5.0: A Guide to the Perplexed"

Introduction

If you can create a Visual Basic form, you can create a Visual Basic ActiveX control—The steps and skills necessary to create controls with VB 5.0 are practically identical to those used to create a form. What's more, control creation possibilities are virtually unlimited. This paper is intended to help Visual Basic programmers learn how to take full advantage of the capability of Visual Basic 5.0 to support creation of ActiveX controls. Why do I say "learn how to take full advantage" instead of just "take full advantage"? Because Visual Basic supports the creation of extremely sophisticated controls, especially when you consider the possibilities of adopting Win32 API techniques and using third party extensions.

It would take a book to even begin to teach all of the advanced techniques that are applicable to control development (in fact, it did). So the intent of this paper is to point you in the right direction, both in terms of understanding the nature of Visual Basic ActiveX controls, and in terms of pointing out the things that you should learn in order to create advanced controls.

When should you create an ActiveX control?

Think of an ActiveX control first, as a reusable User Interface component. It is, of course, possible to create invisible controls that provide reusable functionality that has nothing to do with a user interface, but before you make a final decision to solve a problem using a control, you should be aware of other new features in Visual Basic 5.0 that may provide a better fit for your particular situation. For instance, one of the major new features in Visual Basic 5 is that ActiveX components can raise events. This applies to ActiveX DLL components and ActiveX EXE components. As a result, many tasks that previously required ActiveX controls can now be implemented using ActiveX components.

The following is a brief summary of the major characteristics of the two technologies that should prove helpful in deciding which you should use for a given task:

- ActiveX components (EXE or DLL servers): These are generally used to encapsulate business rules, algorithms, and data services. They provide objects with associated functionality to other components or applications - even ActiveX controls. ActiveX components can contain forms for a user interface, but are not designed to be seamlessly placed on containers in the same way as ActiveX controls. One of the major new features of ActiveX components is Multi-threading which can provide significant performance improvements in many situations associated with larger enterprise systems and multithreaded applications.
- ActiveX controls: These are generally reusable User Interface components that work in Visual Basic, a web page or any other ActiveX host. When used in a web page, they are a component in the web page

and are controlled by VBScript or JavaScript. ActiveX controls have the ability to be sited on a container, and Support design time programming characteristics including:

- Support for property pages.
- The ability to store property values set at design time in a location specified by the container.
- The ability to interact with the container's property browser if one exists.

Starting out with ActiveX controls

Beginning anything new can be intimidating. The suggestions in this section should help guide you in your development efforts.

Design your control

Visual Basic's ease of use has often tempted programmers to go directly to coding, without bothering to design their application. The same will probably be true with ActiveX controls. Before you even open a new control project, you should decide on the following:

- Which control model to use (more on this in a moment).
- Which standard properties and methods will you implement.
- Which standard events will you implement.
- The characteristics of the user interface for your control (design time and runtime).

A large part of creating a quality ActiveX control lies in the details - verifying the design and runtime characteristics of every property, method, event and user input. Defining these ahead of time will go a long way towards helping you succeed.

The Four Control Models

The Visual Basic 5.0 documentation suggests that there are three control models. For the purposes of this paper, there are actually four:

Enhance an existing control

In this model your control contains a single constituent control. Most of the properties, methods and events of the constituent control are mapped to public properties, methods and events of your control. Your control can add new properties or events, or otherwise modify the behavior of the standard control. For example: you could implement a list box that has built in search capabilities.

The biggest advantage of enhancing an existing control is that it is extremely easy to do. The ActiveX control interface wizard can do much of the work of mapping public properties, methods and events to those of constituent controls or the UserControl object.

Even though this is the easiest, There are reasons why you might want to use another approach:

- Because each constituent control defines its own behavior and drawing characteristics, there are limits to what you can do to modify the control. There are occasions where you can use advanced subclassing techniques to perform some modifications to the behavior of the constituent controls, but this is a rather advanced technique that requires a good understanding of the Win32 API and, depending on the

control, some trial and error as you attempt to figure out how the control responds to different Windows messages.

- Constituent controls are always in run mode when your control is active, making it impossible to set the control's design time properties even though the control's container is in design mode. For example: if you want to create an enhanced list box that supported both single and multiple selection modes, you would need to place two separate list box controls into your control - one set to single selection, and the other to multiple selection. You would then show one of these controls depending on your control's configuration. This approach also requires you to manage properties carefully - when the developer using your control sets a property, you need to be sure to set the property for the visible control. This technique breaks down when you have more than one or two design time properties to deal with, as the number of constituent controls you need doubles with each one. There are, however, other solutions to this problem, as you will see shortly.
- As soon as you go beyond the built in controls or redistributable controls provided by Microsoft, you run into licensing and copyright issues. In order for your control to work in Visual Basic's design environment, each of the constituent controls must be properly licensed.

Build a complex control from constituent controls

This model is a superset of the Enhanced control model. The major differences are as follows:

Instead of mapping your public properties and events to a single control, you map them to any or all of the constituent controls. It is also possible to map the same property to more than one control.

You can map multiple constituent events to one event in your control. For example: in a control consisting of two command buttons, the click event from both constituent controls can be mapped to a single parameterized click event as follows:

```
Event Click(ByVal ButtonNum%)
Private Sub Command1_Click()
   RaiseEvent Click(1)
End Sub

Private Sub Command2_Click()
   RaiseEvent Click(2)
End Sub

Private Sub UserControl_Click()
   RaiseEvent Click(0)
End Sub
```

You can map a single class property to multiple controls as shown here with the Font property:

```
Public Property Get Font() As Font
   Set Font = Command1.Font
End Property

Public Property Set Font(ByVal New_Font As Font)
   Set Command1.Font = New_Font
   Set Command2.Font = New_Font
   PropertyChanged "Font"
End Property
```

Why do we return the font property from the Command1 object in the Property Get statement? In truth, it doesn't matter. Both of the constituent controls reference the same Font object, so you can retrieve a reference from any of them and it will work.

This model of control creation suffers from exactly the same advantages and disadvantages as the Enhanced control approach. It also has the characteristic that the UserControl itself cannot receive the focus - only the constituent controls receive focus.

User drawn controls

The two prior approaches have the advantage of being incredibly easy to use. Now let's look at an approach whose major advantages is incredible flexibility - it allows you to do almost anything.

User drawn controls represent one of the most exciting approaches for control creation, though it also represents a jump in complexity. With this type of control you work primarily with properties and events of the UserControl object. You can include invisible constituent controls if you wish, but as soon as you add a visible constituent control, the behavior of the control changes in that the UserControl object can no longer receive the focus. You will usually use the UserControl_Paint event to draw to the control. All of the appearance characteristics and behavior of the control is defined by your code.

When implementing a user drawn control, you must define the following:

- Design time appearance. It is possible to create a control that a developer can interact with at design time by setting the UserControl object's EditAtDesignTime property to True, but this is relatively uncommon.
- Run time appearance and behavior when the control has the input focus.
- Run time appearance and behavior when the control does not have the input focus.

The biggest advantage of user drawn controls is their flexibility. Because you determine the behavior and appearance of the control at all points in its life, there are few limits on what your control can do.

The biggest disadvantage of user drawn controls derives from this fact. Not only can you determine the behavior and appearance of the control - you MUST do so. This can rapidly get very complex.

Keep in mind that when you are implementing a user drawn control, you are not limited to the functionality inherent in Windows - you have access to all of the Win32 API functions as well - and the Win32 API includes some extremely powerful graphic and text output functions that go far beyond what is implemented by Visual Basic (and are faster besides).

Also watch for third party products that include commercial quality controls with source code. They not only offer controls that you can use in your application, but a great way to learn advanced control creation techniques.

The fourth control model: Create Your Own Window

The Windows operating system defines standard classes of windows such as listboxes, text boxes, outline windows, common dialogs and so on.

Many ActiveX controls and Visual Basic intrinsic controls are actually superclasses of these standard windows. For example: The Visual Basic list box control actually creates a Windows listbox window. It intercepts the underlying messages for the window and maps them into control properties and events. This allows a relatively simple control to take full advantage of controls that are built into the operating system.

The problem with this approach is that your control can suffer from limitations that are introduced at either level of the hierarchy. Not every ActiveX control implements all of the functionality of the underlying Windows class. For example: even the Visual Basic ListBox control does not allow you to set tab locations in a list box, a feature that is built into the operating system's implementation. With Visual Basic 5.0 authored ActiveX controls you are also limited in that any design time only properties cannot be changed at runtime.

Why is this?

Because with a LISTBOX class window, certain characteristics such as whether the control can handle multiple selections, must be determined as the window is created. In order to change this characteristic, it is necessary to destroy and recreate the window. Visual Basic 5.0 created controls cannot arbitrarily destroy and recreate a constituent control - something that is necessary to change characteristics such as support for multiple selections.

But there is a solution.

Instead of using the Visual Basic listbox, you can go directly to the windows class itself and create your own LISTBOX class window on top of the UserControl. Because you created the window yourself, you can destroy it and recreate it at will - allowing you to change these "design time" characteristics any time you wish - even at the container's runtime if you so choose.

The advantage of this approach is that it allows you to take full advantage of window classes that are defined by the operating system.

But there are a number of disadvantages:

- This approach requires a good understanding of Windows API techniques and the operation of the windows class.
- This approach can involve quite a bit of work - especially for complex controls.

But there is one factor to keep in mind. If you are creating a control for your own use, you may not need to expose all of the functionality offered by the operating system for a particular window class. A commercial developer will usually expose as many of the features of the underlying control as possible to make the control useful to as many programmers as possible. But when you take this approach for your own applications, you can concentrate on those features that you need, and not waste time implementing and testing others.

So this model may be more practical than it seems at first glance.

See the wizard

The ActiveX control interface wizard can be very helpful in taking some of the grunt work out of implementing your controls. It is especially useful with regards to supporting standard properties, methods and events in your control. It is also an excellent learning tool, since it allows you to see Microsoft's recommended way to handle certain types of properties.

The wizard begins with a screen that asks you which standard properties will be supported in your control. If your control is intended to be invisible at runtime, you should remove all of the standard properties from the control at this time. Otherwise, you should review the list to see if it matches the design specification that you should have created before running the wizard. The default list suggested by the wizard is a good starting point. I would suggest that you add the hWnd property as well, as it will make it easier to use your control with more advanced applications that use Win32 API calls. Do not arbitrarily add every available property to the list - many of them are intended for internal use by the control and should not be exposed.

The next screen prompts you for custom properties, methods and events. Remember that you can add members to your control at any time by editing the code for the control. It is entirely up to you as to whether you find the wizard more convenient. This is one area in which Visual Basic control development differs significantly from Visual C++. The Visual C++ class wizard edits code in several files including the header file, the source file and the ODL (Object Description Language) file, performing tasks that are difficult for even experienced programmers to do correctly. The Visual Basic ActiveX control interface wizard produces relatively little code, and that code is so simple that even beginning programmers can understand it. It is thus a very reasonable approach to add control members by just editing the control's code module.

The next wizard screen maps the control's public members to those of the constituent controls or the UserControl object. The code produced by the wizard to map members in this manner is very educational, though it may not always match the functionality that you desire.

The final wizard screen lets you specify the characteristics of those members that are not mapped. The first thing that you should do is choose a property type other than the default variant type. While there are some situations where variant properties are appropriate, by and large they serve only to impose additional testing complexity in return for a loss in performance. Not a good deal by any measure. You can also set the member description and read/write characteristics. Once again, these are all settings that can be made quite easily without the wizard if you prefer to do so. You should also use the Tools-Procedure Attributes menu command to bring up the Procedure Attributes dialog box and make sure that the procedure ID is correct for each standard property.

Regardless of whether you choose to use the wizard or write your code directly, it is imperative that you understand how the code that the wizard produces works. This is because it is unlikely that the code produced by the wizard will match your needs exactly, especially in a sophisticated control or one that is made up of multiple constituent controls where the members of the control do not map directly to those of constituent controls.

Remember that the wizard does not provide any range checking or validation on your properties - something which you should always do.

Learn the Fundamentals

There are several subjects relating to control development that you do not need to understand well in order to muddle through a simple control. But a thorough understanding of them will help a great deal when designing and implementing sophisticated controls.

Key Objects

There are four main objects that you will work with when creating ActiveX controls using Visual Basic:

The UserControl object is the heart and soul of every VB ActiveX control. It is the object that the container interacts with directly. All user interaction is through this object, and its events drive the behavior of your control.

The Ambient object provides you with information about your control's container. The most important ambient property is the UserMode property which tells you whether your control's container is in design mode or run mode (for those containers that support this features).

The Extender object is the object that a developer using your control will use to communicate with your control. It contains not only the public members of your control, but additional properties that are added by the container. For example: The Left and Top property describes the location of the upper left corner of your control. You do not need to implement these properties because the container adds them to your control's extender object.

The control object itself contains the members that you define for the control. These are the public properties, methods and events that a developer will use to work with your control.

The meaning of design time and run-time

One of the most confusing issues that a new control developer faces when using Visual Basic to create controls is the difference between the control's design time and runtime, and the container's design time and runtime. The following table summarizes the possible control states:

Control \ Container	Design time	Runtime
Design time (Designer open)	Control disabled (hatched lines)	Control disabled (hatched lines)
Runtime (Designer closed)	Control design mode	Control run mode

When the control is in design mode, the control is disabled and appears with hatched lines in any container that is referencing the control. When the control's designer is closed, the control is running - regardless of the mode of the container. You will need to define both the design time and runtime characteristics of your control.

COM

The Component Object Model is the underlying technology on which all ActiveX components are based. While you can accomplish a great deal using Visual Basic without understanding COM, learning at least the principles of how it works can help you take full advantages of all of the features of Visual Basic. It will certainly help you understand how those features work.

Error Handling

Regardless of whether you use the ActiveX control interface wizard or code support for properties yourself, you should always review your control's members to incorporate validation to make sure that they cannot be set to illegal values. You should review the list of trappable Visual Basic errors, since you can, and should, use these standard error codes where appropriate. This will help ensure that your control is fully compatible with containers that expect certain standard errors to be raised in particular situations. For example: you should raise error 380 if your control detects an attempt to set a property to an invalid value.

Building Advanced Controls

There is a saying that the most important part of solving any difficult problem is knowing that an answer exists. The intent in this section is not to try to teach you advanced techniques, but rather to point out some of the techniques that are possible so that when you reach the point of needing them, you'll no that a solution does exist.

Persistence can go beyond properties

An ActiveX control can save and load design time properties by using the PropBag object during the UserControl's WriteProperties and ReadProperties events.

Both the ActiveX control interface wizard and the typical control samples tend to show individual properties being saved in this manner. But you should be aware that there is absolutely no relation between the property saving mechanism and the properties themselves. The labels that you use in the PropBag functions do not need to be names of actual properties. In other words, you can save any data that you wish using any label that you wish.

This means that the property persistence mechanism for controls should be thought of as a way to save the design time characteristics of a control - not just a way to save properties.

Of course, if you are saving properties, you should use the actual property name as a label for the PropBag functions in order to improve the readability of both your control code and the forms that are created using your control.

DataBinding can go beyond standard control databinding

The easy way to databind a property in a control is to let Visual Basic do it for you. This is accomplished by defining a property and setting it's attributes using the Tools-Procedure Attributes dialog box. There are four available selections for databinding:

- Property is data bound. Check this box to cause the property to be bound.
- This property binds to the DataField. This property becomes the "default" bound property if the control has more than one bound property.
- Show in DataBindings collection at design time. Allows the developer using your control to edit the data binding at design time.
- Property will call CanPropertyChange before changing. This options informs the container that your control will call CanPropertyChange before changing bound property values.

To understand these better, you need to have an idea of what happens when you set a property to be bound. Visual Basic adds four properties to the control's extender object:

1. The DataSource property is used by the developer to select a data control to which your control will be bound.
2. The DataField property is used by the developer to select a field in the database to which one of your control's properties will be bound (the one that has its "This property binds to Datafield" check box set. You should always check this box for at least one property).
3. The DataBindings property appears in the VB window to allow the developer to bind other bound properties in your control to various fields in the database. All bound controls whose procedure attribute specifies "Show in DataBindings collection at design time" will appear in the DataBindings dialog. The control's Extender object also exposes a DataBindings collection which contains a DataBinding object for each bound property. This object contains the binding information for the property.
4. The DataChanged property indicates if any of the bound properties have been changed.

One important thing to keep in mind is that databinding is supported by the container. Containers are not required to support databinding, and not all that do will support binding of more than one property in a control.

Keep in mind, however, that your ActiveX control has full access to the Microsoft Data Access Library, and all the database features that are available to any Visual Basic control or component. As such, it is possible to create more sophisticated models of databinding - for example: you can create a list box that is bound to a field, where each entry in the list box corresponds to a different record. You can have multiple list boxes in your control, each of which is bound to a different data source.

To do this you will have to define your own properties to allow the user to set the sources for the data, which can be tricky and is definitely harder than letting Visual Basic do the work for you. However, if the level of data binding that you need goes beyond that supported with the built in databinding features, the extra effort can make even the most sophisticated design possible.

Interface Extensions

Even this early in the life of Visual Basic 5.0, I have already heard people make the statement that you can't write serious ActiveX controls using Visual Basic, and that to do so you must go to tools such as Visual C++.

This is absurd.

Visual Basic's great strength has always been that encapsulates a significant amount of functionality in the language itself, then allows you to accomplish virtually anything by selectively using Win32 API functionality or third party products. This is definitely the case with ActiveX controls.

Visual Basic created ActiveX controls can handle many types of controls without going beyond the core language. In fact, in one area - the ability to use constituent controls, Visual Basic is far superior to Visual C++. The following techniques are available for extending Visual Basic when it comes to creating ActiveX controls:

Win32 API Functionality

The Win32 API provides thousands of functions that cover every aspect of Windows, and the vast majority of them can be called safely from your ActiveX control project. The most important of these for most control developers are likely to be those relating to drawing, as the Win32 API provides drawing functionality that is vastly greater than that provided by Visual Basic alone.

Subclassing and Private Windows

Visual Basic now provides native support for private windows and in-process subclassing, though you should use it with great caution. You should, in any case, perform these operations in a separate ActiveX DLL component and not in your control code. This is because adding these features to your control code makes it nearly impossible to debug your control. This is because when your control enters break mode, no Visual Basic code runs in your control - including the code that implements the private window or subclassing operation. This is a case where using a commercial subclasser or window management tool may save you time and effort.

Overriding Standard Interfaces

Every ActiveX control works by supporting standard interfaces within the control, regardless of the language used to implement the control. These interfaces are part of the ActiveX specification. Visual Basic's standard implementation of these interfaces may not expose the full functionality of the interface to you as a control developer. However, you can use a third party tool such as Desaware's SpyWorks to override the behavior of a standard interface. For example: the IPerPropertyBrowsing interface is used internally to control the display of property information and enumeration lists in the VB property window. By overriding this interface you can display anything you want in the property window entry for a given property, and you can customize the drop down list with entries that do not correspond to an enumeration. Plus, you can use drop down lists for non enumerated properties.

Adding New Interfaces

You should be aware that the Implements statement allows you to add interfaces to your Visual Basic classes and objects. This includes your ActiveX control object.

Under Visual Basic 5.0, the Implements statement only supports a kind of interface called an "automation interface", one that is based on an interfaced called IDispatch. However there are certain non-automation interfaces defined by the ActiveX specification that can be used to extend the functionality of your ActiveX control. This is another task that can be accomplished using third party products. The IObjectSafety interface is a good example of this. This interface allows a control to report to a container that it is safe for initialization or safe for scripting - an important feature for any control that is intended to be downloaded through the Internet or a corporate intranet. The Visual Basic setup wizard allows you to mark a control as safe by adding an entry into the registry. However, if you implement IObjectSafety in your control, not only can you avoid modifying the registry (and the overhead of a registry lookup), but you can create a control that supports both a safe and unsafe mode. On an Internet download, the browser can request that the control enter safe mode. You can detect this request and disable any features in your control that could harm

the client user's system. However, a developer using the control within an application would be able to take advantage of all of the control's features, since Visual Basic does not request that a control enter safe mode.

Conclusion

Visual Basic is not only the easiest way to create ActiveX controls, I expect it will soon become the most popular way to create all types of controls, from simple downloadable controls to complex controls intended primarily for application developers.

If you take the time to understand the underlying technology, and are prepared to learn and use all of the tools available for extending Visual Basic, there will be few tasks, if any, that you cannot accomplish in Visual Basic when it comes to creating advanced ActiveX controls.

My company, Desaware, specializes in advanced tools and information for Visual Basic programmers. Our web site, www.desaware.com, contains additional information related to ActiveX control development including:

- Learning to create ActiveX controls, ActiveX code components and ActiveX documents.
- Use of the Win32 API
- Subclassing and private window management techniques.
- Overriding standard interfaces
- Implementing new interfaces

Last Page