

[MSDN Home](#) > [MSDN Library Archive](#) >

---

Archived content. No warranty is made as to technical accuracy. Content may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist.

Visual Basic 4.0 Technical Articles

## Building Add-Ins for Visual Basic 4.0

Ken Lassesen

Microsoft Developer Network Technology Group

July 25, 1995

### Abstract

This article describes how to create 16- and 32-bit add-ins using Visual Basic® version 4.0. These add-ins will run in Microsoft® Windows® 3.1, Windows NT™, or Windows 95. The ADDINVB4 sample, which accompanies this article, shows several proof-of-concept add-ins, including a standardized menu bar and a library of modules. The reader is assumed to have passed Microsoft Certification in Microsoft Visual Basic version 3.0 and to be familiar with creating OLE servers in Visual Basic version 4.0.

### Introduction

Add-ins to the Visual Basic® Integrated Development Environment provide a way for Visual Basic programmers to rapidly create and refine development tools using Visual Basic, thus allowing them to use computer-assisted software engineering and other state-of-the-art software techniques. Using add-ins could potentially cut the time it takes you to develop a polished, robust application by an order of magnitude.

What is an add-in? Add-ins are special-purpose OLE servers that establish two-way communication between the add-in OLE server and the Visual Basic Integrated Development Environment, also an OLE server. Add-ins are a new feature of Visual Basic version 4.0 that will continue to evolve in future releases. The original designers intended the add-in model to be used for controlling source code and not for managing projects, but later developers modified the model to permit some degree of project management. The add-in model for Visual Basic 4.0 is so far incomplete; a few workarounds are needed, which this article will attempt to provide.

What can add-ins do? They can do almost anything you can imagine, for example:

- Routine task automation such as creating standard menus and standard toolbars or generating Help file templates and ToolTips
- Programming-standards automation
- Form and application wizards

In this article I will build an add-in that implements a few proofs-of-concept to show the potential of the add-in. This article will concentrate on the technique of building the add-in and not on what the add-in does. (Read the commented code for that information.) The add-in samples are intentionally incomplete.

What is the difference between a wizard and an add-in? An add-in has all the characteristics of a wizard, except that it has the intelligence to detect what the developer is doing and react appropriately. For example, an add-in can detect events in the VBIDE object. Imagine that you are creating a new form, and a dialog box automatically appears to ask if you want:

- To add the standard menu items (with code) to the new form?
- To have a form wizard create a set of controls from a database table?

Add-ins can buy better productivity, but they are not a silver-bullet cure-all. Add-ins encourage the discipline that developers do not always accept easily.

### Jump-start Tricks

I have described the vision of what's possible above, but reality is not always as clear. Creating an add-in can be a source of frustration and confusion. First, unless you work as an independent software vendor supplying tools to Visual Basic developers, you cannot expect to develop much in-depth knowledge or experience of the VBIDE object. Second, the implementation of the VBIDE object in Visual Basic 4.0 is neither complete nor intuitive. Third, add-ins are much more object-oriented than most Visual Basic coding.

I found three tricks that helped me in constructing add-ins:

- Use a map of the objects.
- Use nicknames for objects.
- Check the capabilities before writing the algorithm.

A map of the VBIDE object clarifies relationships easily and identifies limitations quickly. I created a map of the objects in the VBIDE object from the Object Browser. (This graphic in various formats is available in my article ["Mapping Visual Basic 4.0: The VBIDE Object."](#)) The VBIDE object contains objects whose names are confusing at the start. The common practice of having child objects taking the type name is not consistently done in the VBIDE object. For example, an Application.ActiveProject object is declared to be a ProjectTemplate object with no Application.ProjectTemplate object in the VBIDE object, while Application.FileControl object is declared to be a FileControl object.

In my Visual Basic code, I give objects nicknames to reduce complexities. To illustrate the difference between using nicknames and not doing so, consider the following lines of code:

```
'Example of a pedigreed object.
A$ = ThisInstance.ActiveProject.ActiveForm.SelectedControlTemplates(i%)
    .Properties(j%).Name
'Example of creating a nicknamed object.
Dim ThisProperty As VBIDE.Property
Set ThisProperty =
ThisInstance.ActiveProject.ActiveForm.SelectedControlTemplates(i%)
    .Properties(j%)

'Example of a nicknamed object
A$ = ThisProperty.Name
```

There is nothing wrong with using fully qualified objects, just as there is nothing wrong in describing the "begat" chapters in the King James Bible as classic literature—I just prefer simplicity.

The capabilities of the VBIDE object are not clear without careful reading. The VBIDE map in my article ["Mapping Visual Basic 4.0: The VBIDE Object"](#) shows the hierarchical structure of the VBIDE object, but does not illustrate its capabilities clearly. Table 1 shows the normal components of a Visual Basic project and the capabilities of the VBIDE object to manipulate these components. The VBIDE object at first appears deficient. Don't despair; this article builds a framework to correct this.

Table 1. The Capabilities of the VBIDE Object in Visual Basic 4.0 to Add, Remove, or Modify Project Components

Project components	Add	Remove	Modify
VBX	ProjectTemplate.AddToolboxVBX	No	No
Type library	ProjectTemplate.AddToolboxTypelib	No	No
Control	ControlTemplates.Add	No	Limited
Declarations or Procedure to Form	FormTemplate.InsertFile	No	No
Declarations or Procedure to Module	No	No	No
Declarations or Procedure to	No	No	No



- The Initialize event.
- The ConnectAddIn event.
- The Timer1\_Timer event.

## The Initialize event

When the VBIDE object creates an instance of the add-in, this instance may create many other instances as side effects. Figure 3 shows an example of the other instances created using my model.

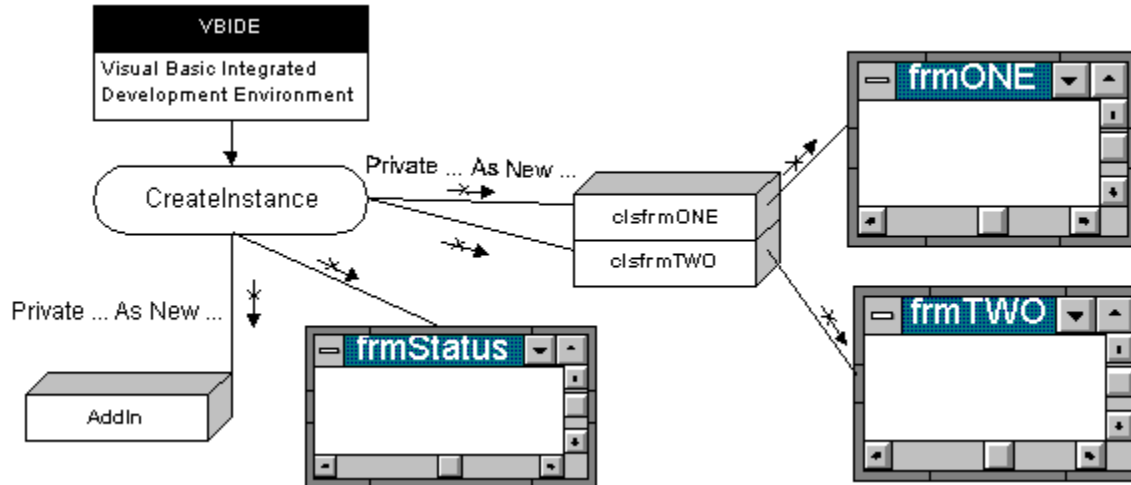


Figure 3. Cascading instance created with the add-in instance

This may seem a bit excessive. Unfortunately, an add-in needs this structure because the add-in may be used concurrently by many VBIDE instances. (The class module should have the `Instancing` property set to `Creatable MultiUse`). Using the `Creatable MultiUse` setting with multiple clients may require some reflection and may change your programming style. Thinking about the `Instancing` property has led me to suggest that you follow these guidelines:

- Global variables are not permitted.
- Module level variables in nonclass modules are not permitted.
- Static variables or procedures in nonclass modules are not permitted.
- The project creates all forms and class modules by using a `Dim ... As New ...` statement and uses only these new instances.

I leave the explanations to you for your enrichment and education.

The code used to create these instances is short. I create the child instances in the `Declarations` section of each file, using `Private` to enforce a strict hierarchy and to allow multiple instances with the same instance name to coexist but in different scopes. The child instances must disappear when the parent terminates. I implement this code in my `AddIn` class module. I create the other classes' instances and the `frmStatus` form instance with the following code:

```
'In Declarations of AddIn.Cls
Private ThisAddInMgr As New clsAddInMgr 'clsfrmONE
Private ThisTipsHelp As New clsTipsHelp 'clsfrmTWO
Private ThisForm As New frmStatus
```

I place the code to create a form instance in each child class module under the `Declarations` section of the class module or form:

```
'In another class module or form, i.e. frmONE.Cls
```

```
Private ThisForm As New frmAddInMgr 'frmONE
```

These few lines create all of the instances shown in Figure 3. Each time the VBIDE creates an instance of the add-in, the add-in instance in turn creates separate private instances of any child class or form.

## The ConnectAddIn event

The ConnectAddIn event passes a pointer to the instance of the VBIDE that created it. The add-in instance needs this pointer so that it can manipulate the right instance of the VBIDE. The add-in instance uses the ConnectAddIn event to add menu items to the VBIDE menu bar.

The VBIDE model for menu items is simple: Each menu choice sends a click event without arguments to a class instance that has an AfterClick method. The procedure receives no information, except that the AfterClick event fired. I must attach each menu item to a different class instance. This allows each menu item to fire a different AfterClick event in each class module.

**Note** Users may start actions by using a form or a menu item added to the Visual Basic Integrated Development Environment. I prefer to use only the menu items to toggle forms to visible or invisible. This allows users to access add-in features by a single click on a form instead of navigating a menu with two or three clicks.

The class instance needs to receive a pointer to the instance of the VBIDE object and a pointer to the menu item that fires it. This class passes these items to the child form so that the form may manipulate the VBIDE object or the menu item controlling it. I never try to trace my way up an object's hierarchy using the Parent property. I find that it is easier and simpler to pass the needed pointers down the hierarchy at the creation of each child instance instead. The code to do this in our project is very simple.

```
Set ThisInstance = VBInstance 'Keep a reference.  
Set ThisSubMenu = ThisInstance.AddInMenu.MenuItems.AddMenu("&My Addin")  
Set ThisMenuItem(2) = ThisSubMenu.MenuItems.Add("&Standard Code")  
hThisMenuItem(2) = ThisMenuItem(2).ConnectEvents(ThisAddInMgr)  
ThisAddInMgr.MenuLine = ThisMenuItem(2)  
ThisAddInMgr.VBIDE = ThisInstance
```

The first line saves a pointer back to the calling VBIDE for this instance. The second line adds a submenu to the VBIDE where I will place other menu items for the add-in. The next line adds a menu item under our submenu and then directs the AfterClick events to a specific class instance. The last two lines pass the pointers to the instance and the menu item to the child class instance. The child class instance saves these values and then passes them on to any child class instances or form instances.

```
Public Property Let VBIDE(vNewValue)  
Set ThisInstance = vNewValue 'Keep a reference locally.  
ThisForm.VBIDE = vNewValue 'Pass reference to child.  
End Property
```

At this point you may be a bit confused over another ThisInstance. I always use This as a prefix to Private variables and objects and My as a prefix to application-scope variables and objects. With my add-ins, every class and every instance has its own Private ThisInstance; the Instancing property's setting of Creatable MultiUse prevents the use of MyInstance.

## The Timer1\_Timer event

This last event may confuse you a little. Okay, a lot. One problem with doing OLE Automation is time-outs. I prefer to do asynchronous automation using a timer. With asynchronous automation, the OLE Automation call sets an argument's value, enables a timer, and then returns to the client. The timer executes the method after the OLE Automation call finishes and prevents a time-out. For example, when an instance creates a form instance, Form\_Load would fire, but if this event takes a long time to finish, a time-out may occur. Also, the user perceives faster performance because of this quicker return.

In my implementation, the `Form_Load` procedure is always empty. I use a `DelayedForm_Load` procedure and a timer set to an arbitrary 3/10 of a second. To ensure that the instance executes `DelayedForm_Load` procedure once, the `Timer1_Timer` procedure uses a flag.

```
Private Sub Timer1_Timer()  
    'This allows return to occur fast!  
    Static fFormLoad As Long 'A flag to prevent multiple execution to occur fast!  
    If Not fFormLoad Then  
        fFormLoad = True 'Set flag and disable timer.  
        Timer1.Enabled = False 'BEFORE calling DelayedForm_Load.  
        DelayedForm_Load 'Otherwise the timer may fire AGAIN during it.  
    End If  
    'Other uses of timer may be added here.
```

## Manipulating the VBIDE Project from Forms

The way I created the instances above allows easy manipulation of the VBIDE object from an add-in form. I have local copies of all the important objects that I need to access. I do not have to think about the add-in, but can change the scope of my thinking to the single form I am in and the VBIDE project.

In this section I will look at some manipulations of the VBIDE object and its associated project:

- Extracting information from the active VBIDE project.
- Modifying a file.

The form uses the pointers `ThisMenuLine` or `ThisInstance` to do these manipulations in my model.

## Extracting information from the active VBIDE project

To illustrate the extraction of information from `ThisMenuLine` and `ThisInstance`, I changed the caption of each form belonging to my add-in so that the user can identify the project attached to the form instance. The code is a one-liner:

```
Me.Caption = ThisMenuLine.Caption + " : " + ThisInstance.ActiveProject.FileName
```

Although this does get the job done, I use a nickname for clarity:

```
Private ThisProject As VBIDE.ProjectTemplate  
....  
Set ThisProject = ThisInstance.Active  
...  
Me.Caption = ThisMenuLine.Caption + " : " + ThisProject.FileName
```

**Note** A word of caution in declaring objects from references: I always qualify the class type by the library type—for example, `As VBIDE.ProjectTemplate` instead of `As ProjectTemplate`. This extra qualification identifies the type's library and prevents problems when different libraries use the same property, method, or object name.

## Adding a file

Although I could blindly copy files from other projects, I would prefer to see a menu of files, click the item I want, and have these files added to my project. I create a file containing the code and then add it. The natural depository for these files is a database using a memo field or other equivalent field.

Because most Visual Basic files very rarely exceed the capacity of a Visual Basic string, I place the code I obtained from the database into a string and then call the procedure below. The procedure creates the full path from the filename so that this new file is in the same directory as the project file, checks for the existence of this file, and then adds it to the project. [Note that in the code below, several lines that would normally be typed as one line have been chopped up with returns for online readability.]

```

Public Sub Project_AddFile(ByVal FileCode$, ByVal FileName$, Project
    As VBIDE.ActiveProject)
Dim FullPath$
FullPath$ = ExtractPath(Project.FileName) + FileName$
If Len(Dir$(FullPath$)) > 0 Then
    MsgBox "File [" + FileName$ + " Already exists. Please delete and try
        again.", vbCritical, "ERROR"
    Exit Sub
End If
fno% = FreeFile
Open FullPath$ For Output As #fno%
Print #fno%, FileCode$
Close fno%
FileType$ = Project.AddFile(FullPath$)
MsgBox "The " & FileType$ + " file [" + FullPath$ + "] has been added",
    vbInformation, "Add File"
End Sub

```

This style of add-in allows reuse of standard forms (log-on, System Information), modules (INI functions, registry functions) or class modules. What if I just want to toss in a bunch of procedures into an existing file? The issue becomes a bit more complex because some of the procedures may already exist in the file.

## Modifying existing files

When I started to write this section, I had planned to describe `ControlTemplates`, show how you can add menus and controls, and then move on to my own hacks. I also had planned to show how `Component.InsertFile` works. Upon discovering the limitations of the available methods and the relative slowness of OLE communication, I decided to skip the available methods and go directly to the hacks. Why? Because the hacks work for everything and reduce what you need to learn.

My basic hack is simple:

1. Save the file (form, class module, module, project).
2. Read the file in as a string.
3. Manipulate the string.
4. Rewrite the file.
5. Reload the file into the project.

This approach allows me to check for existing modules, modify items, and delete items. I can implement an algorithm such as "Rename all command buttons to 'pb' followed by a concatenation of its name." I can implement find and replace functionality, a method not currently in the VBIDE object. The bottom line is I have complete control, limited only by my vision and ability to code and not the object's weaknesses. The add-in manipulates the project using Visual Basic code and uses only part of the VBIDE object and its methods.

**Note** Some limitations of the VBIDE object: The `Component.InsertFile` method applies only to a form, and fails when the selected file is a module or class module. The `Component.InsertFile` method cannot add menu items or controls to a form. The `ControlTemplates.Add` method can add a control or menu, but then the procedure must call `ControlTemplate.Property.Name` and `ControlTemplate.Property.Value` to set each property. There is no ability to remove procedures, controls, or menu items.

So how do I modify existing files? The essential code is below. The function called in the third line changes for different uses.

```

FName$ = RemoveSelectedFile(ThisInstance, VBName$) 'VBIDE interaction
FText$ = ReadFileAsString$(FName$) 'File I/O
FText$ = This_AddMenu2File(FText$, lstMenu) 'Custom manipulation
WriteStringAsFile FName$, FText$ 'File I/O
ThisInstance.ActiveProject.AddFile FName$ 'VBIDE interaction

```

The RemoveSelectedFile procedure walks through the ProjectTemplate.Components to identify the file to save and to temporarily remove from the project. The ReadFileAsString procedure opens the file and reads the code in one gulp, instead of slowly doing Line Input and concatenating the strings. The This\_AddMenu2File procedure does the desired manipulation. The WriteFileAsString procedure writes the code in one gulp. I then add the file back in by calling the ProjectTemplate.AddFile method to place the modified copy back into the project.

The RemoveSelectedFile procedure identifies the file by name only. I do not know what type of file it is, so I cannot determine if the requested action is appropriate. Because the procedure is in a shared module, I pass the pointer of the VBIDE instance to it and receive the full filename when it returns. If the form is new (not saved), I must call the Component.SaveAs method before I call the ProjectTemplate.RemoveComponent method.

```
Public Function RemoveSelectedFile$(ThisInstance As VBIDE.Application,
    ByVal VBName$)
Dim i%, ProjFiles As VBIDE.Components
Set ProjFiles = ThisInstance.ActiveProject.Components
For i% = 0 To ProjFiles.Count - 1
    If StrComp(ProjFiles(i%).Name, VBName, 1) = 0 Then
        RemoveSelectedFile$ = ProjFiles(i%).FileNames(0)
        'SaveAs needed if file just created
        ProjFiles(i%).SaveAs ProjFiles(i%).FileNames(0)
        ThisInstance.ActiveProject.RemoveComponent ProjFiles(i%), True
        Exit Function
    End If
Next i%
MsgBox "Unable to save file -- not found.", vbCritical, "Error:" + VBName$
Exit Function
End Function
```

The ReadFileAsString procedure speeds up the reading of the text file by avoiding Line Input. I thought I should show you the code because not many people are aware of this trick.

```
Function ReadFileAsString$(ByVal FileName$)
Dim fno%, TMP$, flen&
flen& = FileLen(FileName$)
TMP$ = Space(flen&)
fno% = FreeFile
Open FileName$ For Binary As #fno%
Get #fno%, 1, TMP$ 'The easy way!
Close #fno%
ReadFileAsString$ = TMP$
End Function
```

The WriteFileAsString procedure uses the same approach as shown below:

```
Public Function WriteStringAsFile(ByVal FName$, ByVal FText$)
Dim fno%
fno% = FreeFile
Open FName$ For Output As #fno%
Print #fno%, FText$
Close #fno%
Exit Function
```

The This\_AddMenu2File procedure makes the changes to the file's code. The code does whatever I want it to do—it is not an add-in issue. I assume that you are skilled with this type of manipulation and will skip the code.

I have covered the essentials for coding add-ins. I leave the analysis of the manipulation and how to implement the code to you. A review of compiler construction texts or the YACC and LEXX utilities should help you with the code because the manipulations often require a grammar and parsing.

## A Quick Checklist

The following checklist enumerates the common steps in creating an add-in:

1. Create an OLE server project.
2. Add a class module called "AddIn" to serve as the entry point for the add-in.
3. In the References dialog box, check the "Microsoft Visual Basic 4.0 Development Environment" item.
4. Press F4 in each class module's code view and modify the following properties:
  - Instancing should be set to Creatable MultiUse.
  - Public should be True.
5. Create the ConnectAddIn and DisconnectAddIn procedures.
6. Create a Class\_Initialize procedure to do initialization, if needed.
7. From the Tools menu, choose Options to designate the project name.

The code in the procedures and module can follow my model or your own.

A few general suggestions about coding:

- Use ConnectAddIn to do initialization routines. I rarely use Class\_Initialize. It fires before ConnectAddIn and does not have access to the pointer to the VBIDE object.
- Add only one menu item under Add-Ins for each server. I add additional items as submenu items.
- If you use any databases, open the database for shared, nonexclusive access. Remember that multiple clients use the tables concurrently.
- Do not put message boxes, modal forms, or lengthy processing in the events fired by the VBIDE object. The OLE server will wait past its time-out period if a prompt return to the client does not occur.
- Use classes to connect forms to the VBIDE object. Send events to the class instance that owns the form. (Forms may not receive events.)
- Avoid using As Object. I always declare items as their correct data type for better readability and for better performance. For example, I always declare the pointer to the VBIDE object as a VBIDE.Application. This data type is part of the "Microsoft Visual Basic Development Environment" reference. The example below shows the difference:

```
Public Sub ConnectAddIn(VBInstance As VBIDE.Application) 'Use
Public Sub ConnectAddIn(VBInstance As Object) 'AVOID
```

## Conclusion

Add-ins are a windfall for the developer who wishes to produce more applications in less time while increasing robustness and the number of features. The time required to learn to create add-ins, to decide what to do, and to implement and test them is a short-term pain that will produce long-term gain if done intelligently. The potential is awesome; the vision is unlimited; the profit for commercial versions is considerable! Have fun!

---

[Manage Your Profile](#) | [Legal](#) | [Contact Us](#) | [MSDN Flash Newsletter](#)

© 2007 Microsoft Corporation. All rights reserved. [Terms of Use](#) | [Trademarks](#) | [Privacy Statement](#)

