



Walt Disney World Swan and Dolphin Resort  
Orlando, Florida

## Moving Up to Autodesk Inventor® Add-Ins

Neil Munro - C-Cubed Technologies Ltd.

**MA13-3** You've got a handle on parameter equations, and your Autodesk Inventor VBA macros have increased productivity and are appreciated by all your users. So, what's next? In this session we'll examine why you might want to write an add-in with VB (or VB.NET) rather than sticking with a macro. We'll look at some common issues including debugging add-ins, creating toolbars and menus for your applications, add-in registration, and responding to events to make your add-ins as effective as possible.

### **About the Speaker:**

Neil is a faculty member at the British Columbia Institute of Technology (BCIT) in Burnaby, B.C., where he teaches CAD and programming courses in the Mechanical Technology program. He is actively involved in the Autodesk Inventor community. Along with a variety of tutorials and other training materials, he has written a number of popular Autodesk Inventor add-in programs.

**[nmunro@shaw.ca](mailto:nmunro@shaw.ca)**



## Introduction

You can use a variety of programming tools to customize and extend the capabilities of Autodesk Inventor®. These range from VBA functions in parameter expressions, to VBA macros, to external programs and add-ins written using VB6, C++, Delphi, or one of the .NET languages.

VBA (Visual Basic for Applications) is the embedded programming environment in Inventor. I hope everyone has realized that LISP will not be making an appearance as an embedded automation tool in Inventor...ever! VBA is embedded in a variety of Microsoft Windows® based software applications, including many Microsoft products, AutoCAD, and other CAD software applications. This wide distribution and the relative ease of learning VBA (as compared to say C++), has enabled many users to accomplish some degree of application automation. Although it is intended primarily as a tool for creating small, specific scripts (macros), VBA has been used to create complex automation programs in many different applications.

So, why would you want to abandon VBA and move to a dedicated programming tool? Hopefully this presentation will give you some information that may help you answer that question (Yes/No/Maybe/Sometimes). One of the common misconceptions of moving from VBA macros to add-ins written in VB or C++ is that they enable you to do much greater things with Inventor. There are a few features (for example, structured storage streams) that are available through C++ that are not accessible with VB add-ins or VBA macros, but for most of us “part time” programmers, there is not much difference in the access you have to Inventor.

## VBA vs Dedicated Programming Tools

So, what are the strengths and weaknesses of VBA?

### Strengths

- The biggest advantage of VBA is that it is included with, and embedded in Autodesk Inventor. Alt+F11 gets you to the development environment (no extra \$).
- You can automate Inventor from any other application (AutoCAD, Excel, and so on) that has VBA. Conversely, you can automate these applications from Autodesk Inventor macros.
- VBA applications run inside the Autodesk Inventor application memory space (in-process). This translates into speed.
- Because VBA is embedded inside a particular application, it supports shortcut references to Autodesk Inventor programming objects:
  - *ThisApplication* – refers to the top-level Inventor application object in the Inventor object model
  - *ThisDocument* – refers to the document (if the macro is embedded in a document)
- It is very easy to test code. Inventor is running and can run macros at any time.
- The learning curve for VBA is somewhat, to significantly less than, other programming tools.

## Weaknesses

- Code security is poor compared to compiled add-ins or external applications. VBA projects can be secured with password protection but this is not robust. Unprotected code can be accessed and changed by any user.
- The tools available to create VBA forms are weak compared to VB and other dedicated programming tools.
- Although you can have some macros that run automatically, many VBA projects need to be manually loaded before running macros embedded within them.
- Setting up Inventor to run macros can be a pain. Auto-run macros can only be defined inside document based projects. Expect performance issues if you have an assembly where many documents have embedded macros.
- Add-ins have the great advantage that they are loaded and activated automatically each time you start Inventor.

## Should you abandon VBA? (In a word, No)

- The VBA environment is a great tool for testing smaller portions of your program.
- It is a great environment to try out new areas (for you) of the API. This is especially true when developing .NET applications since F1 help can be unreliable, and information on object hierarchies (COM objects) is often not available.
- Inventor is always running. Developing add-ins requires you (without tricks) to start and stop Inventor on a regular basis. Add-ins run in debug mode are also out of process, so can be much slower than VBA code.
- The API is no different, you can test most things in VBA.

## What other options are there other than add-ins?

External programs (VB, VB.NET, C#, C++) using Inventor API

- EXE programs
- Run outside Inventor memory space (out of process), so slower.
- Can't customize Inventor user interface, not started with Inventor.

Apprentice Server Application

- In an add-in, your application is embedded inside the Inventor process. In an Apprentice Server application, a subset of Inventor is embedded in your application. These in-process applications are much faster than out of process applications.
- Subset of Inventor API functionality, mostly read-only access to data in Inventor documents.
- Properties and attributes are read/write, file references can be changed.
- Does not require Inventor to be installed on computer running the application ( + free download)
- Very useful for certain applications.

### Tools

What programming language should you use to write Inventor add-ins? If you are familiar with VBA, VB6 or VB.NET are probably your best choices. If you have a background in C, C# or C++ would probably be better choices. Delphi is also capable of generating Inventor add-ins. Regardless of your choice, you are programming with the COM based API exposed by Inventor. Since this is a macro to add-ins class, we'll limit the discussion to VB6 and VB.NET. (Also, I'm not very knowledgeable on matters of C, C#, and C++).

#### VB6

- Commercial sales discontinued but still available if you look. Still plenty of third party support, custom controls, web sites, and so on.
- Easiest transition from VBA, the syntax of VB6 is very similar to VBA.
- SP6 is the latest service pack
- Based on Component Object Model (COM), as is the Inventor API

#### VB.NET

- The future.
- VBA will eventually be replaced with Visual Studio Tools for Applications (VSTA): Office 12, AutoCAD 200?, Inventor ??
- Similar syntax, but requires a different approach to take advantage of many new features and the .NET framework, which provides much of the power of all .NET programming.
- Very good integration with programs written in any .NET language.
- Integration with Inventor is not as good as with VB6 (.NET is not COM), some tricks to get things to work.
- I've found it frustrating to go back to VB6 after using .NET.

### Creating an Add-In

Like any good programmer, ~~steal~~ re-use as much as you can. The help files, SDK examples, and tools / example programs shipped with Autodesk Inventor are great "how to" resources. You can copy and modify from these to help you get started writing an add-in.

Web sites abound (VB6 and .NET) with discussion, tutorials, example code for many functions you might need outside the scope of Inventor programming.

#### Add-In Creation Wizards

Autodesk Inventor 10 includes a set of add-in creation wizards for VB6, VB.NET, C#, and C++. You can install wizards for all languages from:

*%Program Files%\Autodesk\Inventor 10\SDK\Tools\Developers\Wizards*

The VB6 wizard adds registration code generation and basic Inventor user interface creation tools.

**Note:** *To enable the Inventor registration and UI setup tools in the VB6 IDE you must complete the following steps (once only):*

- *Start a new Inventor Add-In project (the Inventor add-in toolbar is missing)*
- *Add a reference to: %Program File%\Microsoft Visual Studio\VB98\Wizards\InventorApplicationWizard.dll*
- *Close VB6 and reopen the project to show the toolbar*

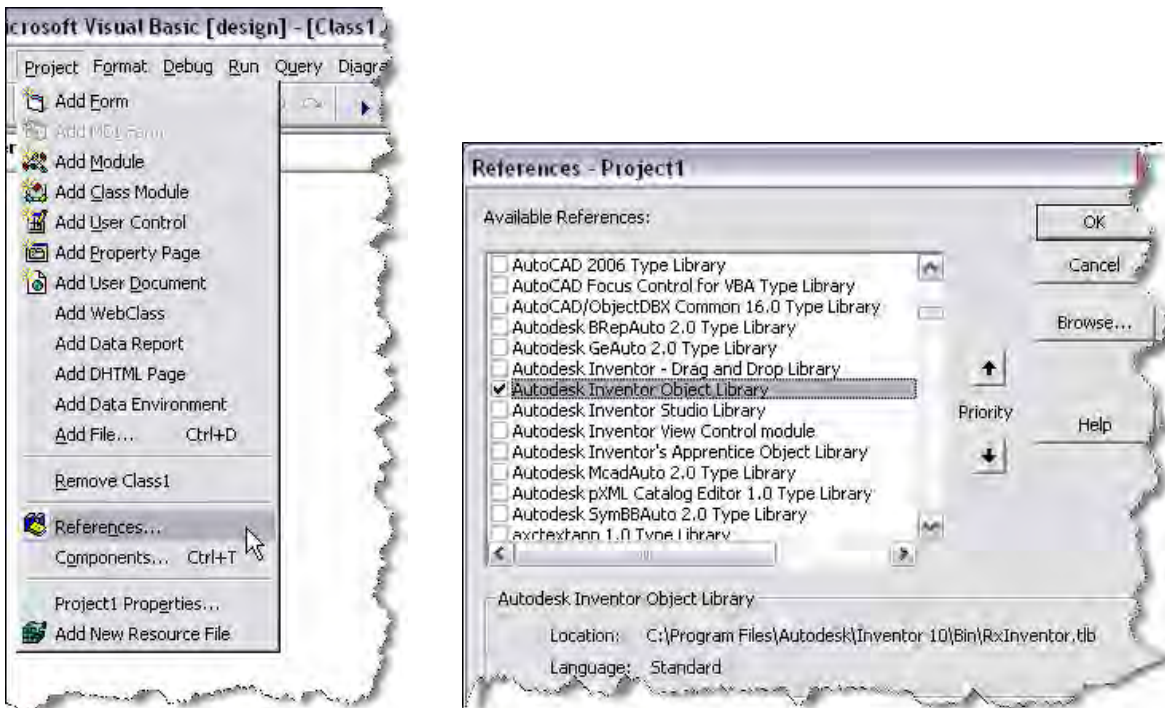
Another advantage of using the wizard is that the add-ins can register itself when it is started for debugging, which saves you having to manually do this (see registering your add-in).

## Add-In requirements

An add-in is (almost always) built as a dll (dynamic link library). Inventor recognizes your add-in during startup and loads it into its process. As shown above, Inventor 10 includes wizards that enable you to generate the outline of an add-in in a variety of programming languages. Another good approach is to develop your own template (or borrow one from somewhere and modify it to meet your needs). You can simply copy the template for each new add-in project.

The following steps outline how to start an add-in project from scratch in VB6.

1. Start a new Active X project (dll project). The project starts with a single class. The connection between your add-in and Inventor must be constructed in a class. Rename the project in the properties window. You might want to use a standard prefix for all of your add-ins (I use C3x as a prefix).
2. Rename the class from the properties window. Use a common naming standard for this add-in class. I use a format like **C3xProjectNameAddIn** where *ProjectName* matches the name of the project.
3. Add a reference to the Inventor object model (type library). This reference enables you to access Inventor objects (documents, geometry, and so on) from your application.



4. Implement Autodesk Inventor's *ApplicationAddInServer* interface.

This interface defines the required properties and methods you must implement in to have your add-in work with Autodesk Inventor. It provides methods that Autodesk Inventor calls during startup (where you can setup your add-in application), and when closing (you can close your application and clean up objects associated with it).

By implementing an interface, you agree to implement all the properties and methods defined in the interface. In addition to Activate and Deactivate, this interface also defines a property (Automation) that is used infrequently, and a method (ExecuteCommand) that is no longer used. You must place at least a comment in these two procedures or your add-in will not work. See the example in the help system for details.

### ApplicationAddInServer\_Activate method

As Autodesk Inventor loads, all properly installed and registered add-ins are located and loaded. When the add-in is loaded, the *ApplicationAddInServer\_Activate* method is executed. You perform setup for your add-in in this method. This usually includes:

- Setting a reference to the Autodesk Inventor application object.

This is the top level object exposed by the Inventor API (application programming interface). In comparison to VBA macros, this corresponds to the *ThisApplication* object.

**Note:** You may choose to declare a variable of Public scope in the class that will be assigned a reference to the application object. This makes the application object available from any code in your add-in.

```
Public ivApp As Inventor.Application
```

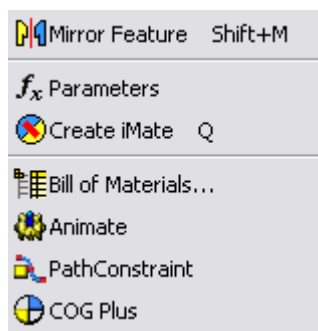
A more strict approach to programming would have you declare this variable private to the class (available to code within the class, not available outside the class).

```
Private ivApp As Inventor.Application
```

You will then need to pass this variable to other sections (classes, forms) of your add-in, as required. For example, if your add-in displays a form, you can pass the variable to a property declared in the form.

- Creating tool and/or menu button definitions.

These are tools that enable the user to activate or control your add-in. You must create these button definitions each time the add-in is activated (every time Autodesk Inventor starts).



- Creating any required toolbars or menus, and adding your tool buttons to these and/or other command bars (once only).

You are only required to create the toolbars and menus the first time the add-in executes, or when you change the user interface (for example, moving from a menu interface to a toolbar). Autodesk Inventor maintains the composition, position, and visibility of command bars (toolbars, panel bars, and menus) between sessions.

**Note:** *It is good practice to handle the event that fires when the user resets all tools to their default setup. In the add-in class, declare a variable as follows:*

```
Private WithEvents uiEvents As Inventor.UserInterfaceEvents
```

*Then handle the uiEvents.OnResetEnvironments; reset your command bars to their original configurations, and add buttons as required to the Inventor command bars.*



For new add-in writers, creation and set-up of tool buttons and command bars is often the biggest stumbling block. The good news is once you have something that works, or you have...err...umm... borrowed a working add-in, you can reuse and edit the basic code for future add-ins.

### ApplicationAddInServer\_Deactivate method

This method executes when Autodesk Inventor shuts down, or the user manually unloads your add-in. Add code here to clean up your add-in, and release any references to objects that would prevent your add-in from unloading. It is good programming practice to set all class level object variables (from this class) to nothing before the add-in closes.

**Tip:** *As you test and debug your add-in, you will undoubtedly run into the odd crash of your application. Even if the crash does not crash Inventor, it can often prevent Inventor from closing properly. Keep the Task Manager running on the task bar so you can quickly check for and shut down extra Inventor instances.*

You might also choose to save settings for your add-in to the registry (VB6) or a config file (.NET) before it closes.

The Autodesk Inventor programming help includes an example with the basic requirements for each of the ApplicationAddInServer methods and properties.

## ButtonDefinitions and Command Bars

Add-ins are intended to integrate as seamlessly as possible into Autodesk Inventor. The ability to add toolbars, panels, and menus to any Inventor environment (e.g. 2D Sketch, Part Features, Assembly, Sheet Metal), and populate them with a variety of user defined controls is an important step when you create an add-in. The Autodesk Inventor programming help files have good information on how to create and manage buttons and command bars (tool containers), but it is spread out over a number of overviews. In addition, adding buttons in .NET is not nearly as straight forward as in VB6.

The following process outlines my process for creating and adding buttons to a VB6 add-in.

1. Create and save any button images as icon (.ico) files.
  - VS.NET has great tools for creating icons of any size.
  - There are a number of freeware or shareware icon creators.
  - *IrfanView* is a well respected image translator that can save other image types as icon files.
  - Don't be afraid to "borrow" Inventor icons as the starting point for your icon (use a screen capture program such as Snag-It). Also, try to use the same colors as native Inventor icons, yellow for existing geometry, blue for new geometry. Create 16w x 15h small icons, and 24x22 large icons

**Tip:** *Inventor will scale your small icons up if you don't care to create two versions.*

2. Use the Resource editor included in VB6 to load the icons into a resource file.
  - Add-Ins > Add-In Manager > (Load the VB6 Resource Editor)
  - Tools > Resource Editor
  - Add icons and then save the resource file (same folder as your add-in)
3. In the class that implements the ApplicationAddInServer interface, declare ButtonDefinition variables at the class level (above the first subroutine in the class).

In VB6, you must declare these variables with the WithEvents keyword. This enables your add-in to recognize and respond when the user clicks the button.

```
Private WithEvents myIvButton As Inventor.ButtonDefinition
```

In the ApplicationAddInServer\_Activate method of the class:

4. Load the Icons from the resource file.

```
Dim myIcon as IPictureDisp
Set myIcon = LoadResPicture(101, vbResIcon)
```

The icon is declared as an *IPictureDisp* interface, and the *LoadResPicture* method loads the icon indicated by the index and type arguments (from the project's resource file).

5. Set a reference to the inventor application object. This top level API object is available from the AddInSiteObject parameter of the Activate method.

```
Set ivApp = AddInSiteObject.Application
```

6. Add the button definition(s) to the ControlDefinitions collection. This collection is accessed through the CommandManager object below the application object.

```
With ivApp.CommandManager.ControlDefinitions
  Set myIvButton = .AddButtonDefintion(argument list)
End With
```

The AddButtonDefintion method adds a simple menu or toolbar button. Other methods are available to add combo box and macro control definitions. Split buttons are added to the CommandBars collection.

The argument list in the above code is long, and can be the source of a number of errors. You can copy the code for each button definition and make edits, or consider creating a function or class method to return a button (the code is not much shorter, but follows good programming practice).

```
.AddButtonDefinition ( _
  "My button display name", _           `display name
  "C3xMyButton", _                     `internal name
  kShapeEditCmdType, _                 `command type
  "{10E015F9-8528-44FC-B588-087600209208}", _ `your add-in ID
  "Description of button action", _     `button description
  "Tool Tip Text", _                   `button tool tip
  myIcon, _                             `small icon 16x15
  myIcon, _                             `large icon 24x22
  kDisplayTextInLearningMode)          `button text display
```

Note the following in the above statement.

- o The second argument (InternalName) is prefaced with a unique string ID used for all button definitions in add-ins I create (**C3x**). Have a prefix of your own.
- o The command type should correspond to the action undertaken when the user clicks the button. See the programming help for descriptions of command type classifications (search for *AddButtonDefinition*).
- o The single icon is used for both small and large icons

**Tip:** Replace all hard coded strings and numbers with constants.

The above button definition code will fail when you run your application from the VB6 IDE (debugging). When you run your add-in from the VB6 IDE, the add-in is run "out-of-process", or outside the memory space of Inventor. Icons cannot be passed between processes with the AddButtonDefinition method, so an error is generated. Instead of editing your button code to run the application in debug mode, catch the error and create a text only button:

```

On Error Resume Next
With ivApp.CommandManager.ControlDefinitions
  Set myIvButton = .AddButtonDefintion(argument list)
  If Err.Number = -2147418113 Then 'replace with constant
    Err.Clear
    Set myIvButton =.AddButtonDefinition ( _
      "My button display name", _
      "C3xMyButton", _
      kShapeEditCmdType, _
      "{10E015F9-8528-44FC-B588-087600209208}", _
      "Description of button action", _
      "Tool Tip Text", _
      , _ 'no small icon specified
      , _ 'no large icon specified
      kDisplayTextInLearningMode)
  End If
End With

```

7. Create environments and command bars as required, and add your buttons to the command bars. Also, save your buttons in a command category so users have access to them for UI customization.
  - These actions are required only when the add-in starts the first time, or if you are changing the UI for accessing your add-in. For example, you want buttons on different command bars, or you are adding new button definitions.
  - Command bars are containers for buttons. They are used to build menus, toolbars, and panel bars.
  - The *FirstTime* parameter of the Activate method enables you to restrict UI setup to times when it is required. If FirstTime is True, it indicates to the add-in that this is the first time it is being loaded and to take some specific action. Typically, when the flag is True, the add-in proceeds to create all of the objects under the UserInterfaceManager that it needs - Environments, CommandBars and Controls. These objects are persistent, but if this is the first time the add-in is loaded, they need to be created from scratch. You want to avoid setting CommandBars and Environments each time your add-in loads, as you would overwrite any UI customization performed by the current user. Button definitions must be created each time your add-in is loaded.
  - You can create new environments for your application (e.g. a FEA environment), but you would more commonly add a custom menu or toolbar for your buttons to an existing environment. You add your CommandBar(s) to the Environment(s) where the tools on the CommandBar(s) are applicable. You would typically use a Pop Up command bar (toolbar or menu), but specialized command bars for flyouts and split buttons are also available. See the programming help file for more information on the type of command bars you can create.

- Add a command category to the CommandCategories collection. Identify the command category with the name of your add-in, and add you button definitions to the category. Your add-in users will see your category on the Commands tab of the Customize dialog box, and can easily find your button definitions in order to add them to other command bars.

For example, to create a toolbar that will show up as one of the optional panel bars in the Part environment:

```
Dim C3xPartCmdBar As CommandBar
Dim uiMgr As UserManager
Dim partEnv As Environment

If FirstTime Then
    Set uiMgr = ivApp.UserInterfaceManager

    'create the command bar
    Set C3xPartCmdBar = uiMgr.CommandBars.Add("C-Cubed", _
                                                "C3xPartTools", _
                                                kButtonPopupCommandBar, _
                                                kAddInID)

    'add your button definition to the command bar
    Call C3xPartCmdBar.Controls.AddButton(myIvButton)

    'get the part environment
    Set partEnv = uiMgr.Environments.Item("PMxPartEnvironment")

    'add the command bar to the part environment panel bar
    Call partEnv.PanelBar.CommandBarList.Add(C3xPartCmdBar)
End If
```

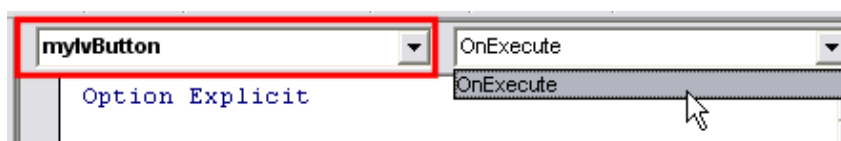
You are not limited to adding your buttons to your own command bars. You can add them to any of the built in command bars in Autodesk Inventor.

To force the "FirstTime" code to execute in subsequent Autodesk Inventor sessions, you can do one of the following:

- Increment the Version setting in the registration file (see registration section).
- Delete the following registry key:  
*HKCU\Software\Autodesk\Inventor\RegistryVersion10.0\UI Customization V3*

Deleting the registry key removes all Inventor UI customization, so incrementing the version setting is the preferable solution when you distribute your add-in.

8. Finally, in your add-in class, generate the event handler for each button's *OnExecute* event. To generate the event handler, select the ButtonDefinition from the object list, and the *OnExecute* event from the event list. Add code to perform your add-in specific tasks in the event handler subroutine. If your add-in is form based, you would load the form here.



### Add-Ins with Forms

Not all add-ins will have a form based UI, the complete user interface may consist of tool buttons or menu items. If your add-in does use a form, here are a few things to consider.

1. If you are converting your VBA application to an add-in, bite the bullet and rebuild the form using the controls in VB6. You can import the VBA form as a Designer, but the forms and controls are not editable in VB6. Also, there are a large number of built in controls available in VB6. You can copy the code from VBA event handlers and tweak it to work with similar VB6 control events. If you are converting to a VB.NET add-in, forms must be rebuilt and the code will also require more tweaks.
2. Modeless forms in VBA stay in front of the Inventor window when you interact with Inventor. Modeless add-in forms will disappear behind the Inventor window without additional code. Use the *SetWindowLong* API call (Windows API) to keep the add-in form visible when the user switches over to interact with Autodesk Inventor. See the VB6 add-in code in the course resources for an example. You must remember to reset the parent window when your form unloads.
3. If you declared a private variable for the Inventor.Application object in your add-in class, pass a reference to the object to a public property of your form (from the OnExecute event of the button that displays the add-in form). Depending on the needs of your program, the property can be read/write (accessible to code outside the form), or write-only (accessible to code inside the form).

### General Add-In Considerations

The move to an add-in usually implies that you are taking a more serious approach to automating Autodesk Inventor. If you are planning to distribute the add-in, either internally to other Inventor users, or as a commercial product to external users, strive to emulate the user interface of built in Autodesk Inventor dialogs, and mimic the minimal interference of most Autodesk Inventor commands.

- Small things such as red text as a visual indicator for invalid textbox entries.
- Consider replacing a dialog interface with context menu items for commands that do not require value entries.
- A somewhat dated set of guidelines for add-in development is available at:  
*%Program Files%\Autodesk\Inventor 10\SDK\Docs\Guidelines\Third-Party Design Guidelines.doc*  
Sections 3 and 4 in this document outline design guidelines for dialogs and other commands.
- Store setting for your add-in (last form position, most recently used values, other settings) in the registry (VB6) or a .config file (VB.NET). Users really appreciate this, and it is an easy way to add a professional touch to your application. Write the settings from the form's Unload event and/or the add-in class' Deactivate method. Read in the settings from the form load event and/or the add-in class' Activate method. Example "save settings" classes for both VB6 and VB.NET are included in the course resources download.
- Learn and use more advanced Autodesk Inventor API techniques. (see SDK examples)
  - Make use of advanced interaction events. Use interaction and client graphics for previews.
  - Learn how to use transactions and the ChangeProcessor to group a series of actions into logical undo/redo steps from the user's point of view.
  - Use reference keys to create and use persistent links to specific objects in an Inventor document.
- Consider creating your own browser pane(s) and adding them to the appropriate documents. You can add an Active-X control (third party, or one you create yourself) to custom browser panes. See the programming help overviews for information on adding custom browser panes.

## Working with Autodesk Inventor Events

Since your add-in can be loaded when Inventor starts, you can take advantage of application and document level Inventor events without user interaction. You can trap Inventor events in macros as well, but add-ins are more suited to supporting events.

Events are divided into a number of general categories:

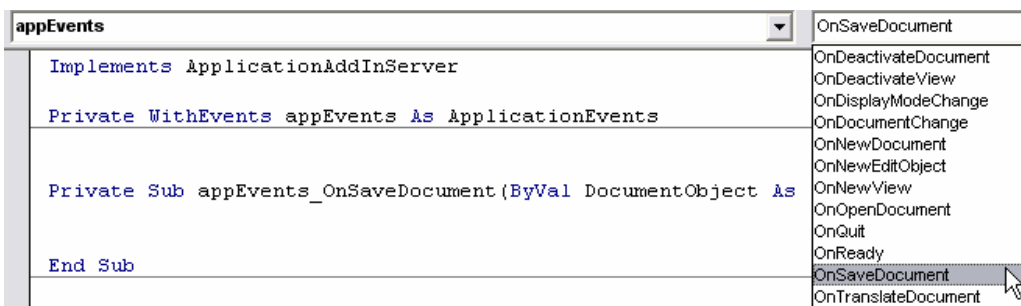
- Document Level Events:
  - General events associated with a particular document, such as OnChange, or OnSave.
  - Part, assembly, and drawing documents support events specific to the document type.
  - Drawing document sub-objects such as views,
- Application Level Events:
  - General events that are not specific to a particular document type.
  - Some application events are similar to document events, such as OnDocumentSave.
    - When a document is saved, both the document and application level events are fired.
  - Other events such as OnNewEditObject are only supported at the application level.
  - You can respond to OnReady (Inventor started) and OnQuit (Inventor closing) to perform tasks when no documents are active

**Tip:** Respond to Application Events rather than document events if possible (better performance).

- Interaction Events:
  - If used, your add-in must set-up and start Interaction events
  - Your add-in can capture user interaction with Inventor objects (via mouse and keyboard)
  - You turn off the capture of these events after your user has fulfilled your interaction requirements
- A number of other categories, such as FileUIEvents are provided for specific tasks.

A detailed discussion of working with Inventor events is beyond the scope of this presentation, but the general procedure for responding to Inventor events in a VB6 add-in is:

- Declare a form or class level variable using the  *WithEvents*  keyword.
  - To capture the events only when a form is active, declare the variable within the form.
  - To setup capture of events at all times (perhaps application level events), declare the variable in the add-in class.
- Assign the appropriate events object to the variable during initialization of your class or form
  - Form Load event of a form
  - ApplicationAddInServer\_Activate event for the add in class
- Select the specific event to handle and add code in the generated event handler.



## Registering Your Add-In

How does Autodesk Inventor know your add-in exists? How does it find your add-in so it can load it when Inventor loads?

As with all COM components, information about your add-in is stored in the system registry. You must add the correct information to the registry in order for Inventor to find and load your add-in. As always, there are a number of methods to get this information into the registry:

- Manually create a text based .REG file., or copy an existing .REG file and edit the values for your add-in. You then “merge” this file into the registry. Users of your add-in must also merge this data into the registry on their machine.
- If you create an add-in using the Inventor Add-In wizard (VB6), the wizard includes a form where you can enter the required information. The wizard can automatically merge the data into the registry, and it can output a .REG file you can distribute with your add-in.
- Add-ins created in a .NET language can include self registration code in the add-in. This code can be called during installation on other machines, and during debugging as you develop the add-in. The .NET templates included with Autodesk Inventor 10 all include self-registration code. You can make changes to the actual settings just as you can in a .REG file.

The contents of a typical add-in .REG file are shown in the following image.

```

REGEDIT4

1 [HKEY_CLASSES_ROOT\CLSID\6F292480-66CF-4AEF-95C6-44439729105A] Add-In GUID
@="C3_Overlay"

2 [HKEY_CLASSES_ROOT\CLSID\6F292480-66CF-4AEF-95C6-44439729105A\Description]
@="Addin to manage positional rep overlay views"

3 [HKEY_CLASSES_ROOT\CLSID\6F292480-66CF-4AEF-95C6-44439729105A\Implemented Categories\{39AD2B5C-7A29-11D6-8E0A-0010B541CAA8}]
4 [HKEY_CLASSES_ROOT\CLSID\6F292480-66CF-4AEF-95C6-44439729105A\Required Categories]
5 [HKEY_CLASSES_ROOT\CLSID\6F292480-66CF-4AEF-95C6-44439729105A\Required Categories\{39AD2B5C-7A29-11D6-8E0A-0010B541CAA8}]
6 [HKEY_CLASSES_ROOT\CLSID\6F292480-66CF-4AEF-95C6-44439729105A\Settings] ← Add-In Settings
"LoadOnStartUp"="1"
"Type"="Standard"
"SupportedSoftwareVersionGreaterThan" = "9.."
"Version"=" dword:1
    
```

If you are using a manually created or copied .REG file, you need to:

1. Compile your add-in. (File > Make *ProjectName.dll*)
2. Search the registry for the GUID that was assigned to your add-in. You search for the ProgID assigned to the add-in class. The format of the ProgID is *ProjectName.ClassName*. For example, for this example add-in, I would search for *C3xMickey.C3xMickeyAddIn*.
3. Copy the GUID from the registry key, and add or replace the six (6) GUID instances in the .REG file. Do not replace the two GUID strings at the end of the Implemented Categories and Required Categories keys, these are separate identifiers that tie the add-in to Autodesk Inventor.
4. Merge the .REG file into the registry. (Double-click the file to merge).

**Tip:** Place a shortcut to the .REG file on your desktop to speed registration.

Your add-in should theoretically work at this point. However, you will most likely need to test and debug it, starting your add-in from the VB IDE, and then starting Inventor. Each time you compile your add-in, or start the add-in in the VB IDE, the registry keys from your .REG file are removed from the registry. You must re-merge the .REG file before starting Autodesk Inventor, or your add-in will not be loaded. See the *Debugging Your Add-In* section for more details.

**Warning:** VB6 has a bad habit of generating a new GUID for your project each time you compile. This can be extremely frustrating as you would need to change the .REG file after each compilation. To stop the compiler from generating a new GUID, set the Version Compatibility for your project to Project Compatibility.

From the menu, select **Project > ProjectName Properties | Component tab**

**Note:** Do not delete the dll after activating this setting. Overwrite the dll on each compile.

## Add-in Settings

Most of the keys under the Settings registry key are self explanatory. See the programming help files for an overview of the settings. The following are settings that are not covered adequately in the help or may cause confusion.

### Versioning

There are two settings that deal with versioning. The programming help has a good overview of the setting that controls which Autodesk Inventor versions (releases) will be able to load your add-in. You can limit the availability of your add-in to a specific Inventor version, to versions newer or earlier than a specific value, or to any version other than the one specified.

For example, to limit the add-in to be loaded only in Inventor 9 and later. (Note the extra period)

**"SupportedSoftwareVersionGreaterThan" = "8.5."**

The "Version" key refers to the UI version of your add-in. This setting determines the value of the FirstTime parameter in the ApplicationAddInServer\_Activate event. If you make changes to how your add-in integrates into the Inventor user interface, increment this value in the .REG file. The next time the add-in loads, the FirstTime flag is set to True, and your environment and command bar setup code will be executed. The format of this setting differs from the other settings, its data type is DWORD (number).

**"Version"=dword:1** (increment the number after the colon)

Two other keys are available to control user interaction with your add-in

- You can hide your add-in in the Add-Ins dialog (lame, because the user can show hidden add-ins)  
**"Hidden" = "0" or "1"** ("1" = Hidden, "0" = Visible)
- You can also prevent the user from unloading your add-in through the Add-Ins dialog.  
**"UserUnloadable" = "0" or "1"**

There is nothing preventing you from adding your own keys below the Settings key for your add-in. How you use them is up to you, Inventor ignores them when the add-in is loaded.

## Debugging Your Add-In

If you follow the Autodesk Inventor discussion groups, the rant level on software bugs ebbs and flows between versions and service packs. Variations on the following are common *“I’m sure it is only a matter of adding or changing a few lines of code to implement or fix feature x”*. Most macro and add-in writers are not professional programmers, but I’m sure you recognize these misguided statements from your own experiences. Debugging your program is an integral part of the development process and sometimes it can be difficult to stomp them all.

The use of breakpoints, watches, and other debugging tools is almost identical in VB6 and the VBA environment in Inventor. The primary issue you need to overcome when debugging a VB6 based add-in is setting up the debugging session so that your breakpoints will be recognized.

To debug an add in written in VB6:

- Complete the following once:
  - From the menu, select Project > *ProjectName* Properties
  - On the Debugging tab, under When this project starts, click “Wait for components to be created”. This tells your add-in to wait until Autodesk Inventor loads it before executing any code.
  - Ensure Autodesk Inventor is closed and no stray instances are still active (task manager)
  - Compile the dll
  - Find the class GUID in the registry as outlined above, and create a .REG file
  - Merge the .REG file
  
- Complete the following each time you start a new debugging session:
  - In the VB6 IDE, place one or more break points in your code. Start with a break in the ApplicationAddInServer\_Activate event handler
  - Start your add-in
  - Merge the .REG file again (*Tip: Place a shortcut to the .REG file on your desktop*)
  - Start Autodesk Inventor. If you have a break point in the Activate event, Inventor will break in your add-in before the main Inventor window is activated.

**Note:** *If you create an add-in with the Inventor Add-In Wizard, you have the option of starting Inventor when your add-in starts. The wizard performs the registration tasks prior to starting Inventor.*

To stop the debugging session without leaving Autodesk Inventor hanging:

- Close Autodesk Inventor
- Once Inventor has completely shut down, stop your add-in from the VB6 IDE

**Note:** *If your add-in crashes and you can’t recover:*

- *Stop the add-in (if it isn’t already)*
- *Close Autodesk Inventor*
- *In task manager, check to see if an Inventor instance is still active. If so, end the process.*

## Distributing Your Add-In

Of course you will want to share your brilliant add-ins with co-workers or other users, or maybe even try your hand at making some money from them. Getting your add-in to work on other machines opens up a whole new set of possible problems.

Again, sticking with VB6 add-ins, the requirements for installing an add-in on another machine are:

- The dll file must be copied to the target machine
- Any supporting files (other than the VB6 runtime files) must be copied to the target machine
  - For example, if you add a VB6 Progress Bar or VB6 Tree View control to a form in your project, you must include the *mcomctl.ocx* file with your project.
- All dll and ocx files described above must be registered on the target machine
  - The .REG file must be merged into the registry on the target machine and supporting .DLL and .OCX files must be registered using regsvr32.exe (we won't even discuss dll versioning).

You could have your users perform all this manually, but they wouldn't think much of you. Fortunately there are numerous tools available to automate these tasks.

- VB6 includes a Package and Deployment wizard that helps you package all required files and creates a setup file (Setup.exe) your users can run to install the add-in on their machine. The setup also creates an uninstaller so the user can remove the add-in cleanly (but why would they ever want to do that?).
- ***Tips for using the Package and Deployment Wizard to Distribute an Add-In:***
  - *Do not include the Inventor.tlb file in the package (Inventor type library)*
  - *Do not include the VB6 runtime files in the package*
  - *Add the .REG file, your help files, and any other support files (including special Inventor parts, assemblies, and so on) to the package (any third party .OCX files are usually automatically included)*
  - *Specify that the .REG file be merged on installation*
- Microsoft also supplies tools (not included in VB6, but possibly free?) to create Microsoft Windows Installer files (MSI). Powerful but comes with a steep learning curve.
- Commercial installers such as InstallShield and Wise Installer. Depending on how much you want to pay, various flavors of password protected trial versions can be created. They even provide web based services for commercial software distribution / licensing. Steep learning curve for advanced functions.
- A number of free or low cost installation scripting applications are also available. The learning curve is generally reasonable, and they offer a great deal of customization and flexibility. Registration information can be built into the install script, eliminating the need for a separate .REG file).
  - *Inno Setup* - <http://www.jrsoftware.org/isinfo.htm>
  - *NSIS Installer* - <http://www.nullsoft.com/free/nsis>  
(VB6 and VB.NET NSIS install scripts included with course resources)

### VB.NET / C# Considerations (Ok, not much C#)

What's different about creating an add-in in a .NET language? Well, quite a lot, but in true programmer style you can find much of what you need to get started in examples and samples.

A few warnings before you decide to go this route.

1. .NET Add-Ins eat up ~100MB of memory space (not each I hope) that is not available for Autodesk Inventor documents. If your users are struggling with large assemblies and memory issues, this might not be the best choice.
2. When you compile your .DLL file in VB.NET, the result is intermediate language code called MSIL. This code is recompiled into machine specific binary code just before it is run the first time. There are a number of tools that can disassemble this intermediate code into VB.NET or C# code. There are also tools to help you, obfuscation and encryption can help prevent decompilers from making sense of your code.
3. Autodesk Inventor 10 works much better with .NET than Autodesk Inventor 9.
4. Once you get comfortable with VB.NET, its not much fun going back to VB6

**Tip #1:** *The SDK samples do not include a VB.NET add-in. However, Autodesk Inventor 10 ships with an Assembly Tools add-in that is written in VB.NET. You can install it to provide additional tools in the assembly environment, but the real bonus is that the source code is also available after installation. A great resource for learning the ins and outs of VB.NET Inventor add-ins.*

**Install from:** `%Program Files%\Autodesk\Inventor 10\SDK\Tools\Users\AssemblyTools`

### VB.NET and Inventor Tips

If you plan on using VB.NET, you can write programs using a similar approach to VB6. However, to use the full power of VB.NET you need become familiar with classes, inheritance, other object oriented programming concepts, Try/Catch exception handling, and some of the vast functionality built into the .NET framework.

The issues that I've run into over the last year of trying to get a VB.NET add-in to fly include:

- Specifying a reference to the Autodesk Inventor object model. If you examine the files in the \bin folder below the Inventor installation folder, you will find the COM based Inventor Object Library (RxInventor.tlb) and also a .NET interop assembly (Interop.Inventor.dll). When you add a reference to Inventor in your VB.NET add-in, always select the Autodesk Inventor Object Library (RxInventor.tlb) from the COM tab in the Add Reference dialog box. Don't browse to and select the existing interop assembly. A new interop assembly is generated for your project, with correctly set property values.
- VB.NET 2003 does not support edit and continue. You cannot edit code while your add-in is running. This is partially offset in that .NET add-ins can self-register and start Inventor automatically when you run the add-in in the IDE.
- Icon and image issues in button definitions (see section below, it's still not perfect).

- Data type issues when passing arguments to Inventor API methods (COM) from an add-in (.NET). The issue here is that all data types (integers, doubles, strings, and so on) in VB.NET are defined in the .NET framework (this is one of the great strengths of .NET). The Autodesk Inventor API is a COM based API, which uses its own data types. For example, a COM Integer data type is 16 bits long, a VB.NET Integer is 32 bits. In theory, the translator between .NET and COM (called a wrapper) is supposed to handle all of the conversions (marshalling) between the two environments. See the section below for a few tips.
- Poor links to Inventor help, and limited information on watched Inventor objects. F1 does not get you to the Inventor programming help. The COM to .NET interface often prevents you from seeing much detail about a COM based object in a watch window.
- Installation issues on client machines. Native .NET applications (.EXE and .DLL) do not require that information be written to the registry. Since an add-in works with Inventor's API (COM), it has to be registered in a similar fashion to a VB6 (COM) add-in. To register a .NET dll you must use REGASM.exe (register assembly) rather than REGSRV32.exe that is used to register COM based components.

**RegAsm.exe myIvAddIn.dll /codebase**

The */codebase* switch is required to enable Inventor to find the add-in in its installed folder.

- For yet undiscovered reasons, distributing and installing .NET add-in on other machines can be a hit and miss affair. My first .NET add-in worked on a few machines, but on most the add-in would install without error, show up as loaded in the Add-Ins dialog box, but was unavailable (in fact, it was never really activated).

To guarantee that a VB.NET add-in will be correctly registered and activated, you must install it to a folder below the *Autodesk Inventor 10\bin* folder.

**Note:** *If you then uninstall it from this location, it can be installed in any folder and the add-in will load and operate correctly. There is no requirement to do so, this just points to the fact there is a registration issue with some .NET add-ins.*

The exception is for machines that have at least one of the VS.NET programming tools installed.

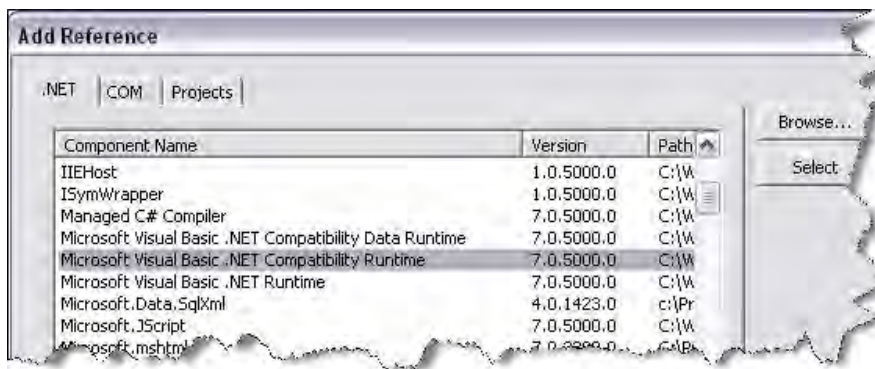
See the course resources for installation script examples for both VB6 and VB.NET add-ins

## Adding Button Definitions to a VB.NET Add-In

As mentioned earlier, adding buttons to a VB.NET add-in is more complicated than in VB6. The *LoadResPicture* method has no direct replacement in VB.NET (that I've been able to find). Inventor's *AddButtonDefintion* method requires that all icons be passed as *IPictureDisp* objects. There are two (and probably more) methods to overcome this in VB.NET.

**Note:** Examples of both techniques are included in the course resources.

1. Add the old VB6 functionality to your VB.NET add-in. The course resources include a working example. The general steps are.
  - Create your icons and add them to your add-in assembly
  - In the properties window, change the Build Action of the icon(s) to *Embedded Resource*
  - In your project, add a reference to the VB6 compatibility runtime dll.



- Import the *Microsoft.VisualBasic.Compatibility.VB6* namespace into your add-in class (reduces typing)
- Extract the icon resource from your add-in assembly
- Use the *IconToIPicture* method from the *Support* class and pass the resulting *IPictureDisp* interface object to the *AddButtonDefinition* method.

**Note:** Microsoft does not support the use of the VB6 Compatibility library in .NET projects. The library may be discontinued in the future. In addition, if you use this library you must copy and register the associated dll if you distribute your add-in.

2. Use the *System.Windows.Forms.AxHost* class to convert an image to an *IPictureDisp* interface object. This method is used in the VB.NET add-in example (Assembly Tools) shipped with Autodesk Inventor 10. This is not a perfect solution either. You must use bitmap files (or convert your icons to bitmaps) which eliminates transparency. In addition, the .NET help indicates that this method is for internal .NET conversions and should not be used in user developed projects.

**Note:** When you debug your add-in, your *AddButtonDefintion* method must pass **Nothing** for each of the two *Icon* arguments. In VB6 you just leave out these optional arguments.

## Passing Arguments to Inventor Methods

Although data types in .NET and COM are different, you can almost always use the same data type declaration in VB.NET as you would in VB6. The Inventor.Interop assembly that is generated when you make a reference to the Inventor object library handles the translation from .NET to COM. So, if an Inventor method required a double value (32 bit variable under COM), you can pass a double (64 bit variable under .NET) from your .NET code, and the Interop assembly will handle the translation.

There are a few places that I've stumbled on where this does not work. If your add-in throws an exception when passing arguments to an Inventor method, it is likely the data type conversion has gone astray. If this happens to you, try changing the data type of your .NET variable.

There are a couple of items you must take into consideration:

### String Variables:

- A string variable in .NET is a reference data type. Without going into details, this means that when you declare a string variable, its initial value is **Nothing**.
- In VB6 (COM), a string is assigned an initial value of an empty string (""), and all Inventor methods that take string arguments expect at least an empty string.
- Therefore, you must assign a value to any .NET string variable prior to passing it to an Inventor method.
- In .NET, you can assign a value to a variable at declaration:
  - `Dim myText As String = ""`

### Arrays

- An array in .NET is also a reference data type
- Many Inventor methods require arrays (Matrices, Vectors, and so on)
- If you declare a variable to hold an Inventor array type, you must assign default values to the array elements prior to passing the variable to an Inventor method.

```
Dim Origin As Inventor.Point, XAxis As Inventor.Vector
Dim YAxis As Inventor.Vector, ZAxis As Inventor.Vector
```

```
Dim oTG As Inventor.TransientGeometry
oTG = ivApp.TransientGeometry
```

```
'can't pass objects that = nothing from .NET
'to IV, so assign default values to array elements
Origin = oTG.CreatePoint(0, 0, 0)
XAxis = oTG.CreateVector(0, 0, 0)
YAxis = oTG.CreateVector(0, 0, 0)
ZAxis = oTG.CreateVector(0, 0, 0)
```

Try

```
'get the coordinate system of the occurrence
Call myMatrix.GetCoordinateSystem(Origin, XAxis, YAxis, ZAxis)
```

### Summary:

Add-ins are a natural progression from Autodesk Inventor macros. The convenience of having tools loaded and ready to go when Inventor opens, the power of the programming tools compared to VBA, and the challenge of learning some new skills are enticements to moving on. The move from VBA to VB6 is not a large leap, but consider moving to VB.NET or one of the other .NET languages. .NET is the Inventor programming environment for the near future.

With some up-front work to create a good template, you can create an add-in in almost the same time it takes to create a VBA macro!

Thanks for attending!

### Resources:

If you are creating commercial add-ins or providing automation tools for a number of Inventor users, consider joining the Autodesk Developers Network (ADN). For a reasonable fee you get direct support for problems, access to API tips and tricks, and much more.

<http://usa.autodesk.com/adsk/servlet/section?id=472012&siteID=123112>

The discussion group for Autodesk Inventor programming is a great free resource. Along with your peers, a number of Autodesk API gurus (employees) frequently answer questions.

<http://discussion.autodesk.com/forum.jspa?forumID=120>

