DevTV: AutoCAD VBA to .NET Migration Basics

Complementary material

Augusto Gonçalves – Autodesk

Are you a VBA developer who needs to upgrade your code to .NET? It's a lot easier than you think using .NET COM Interop. Learn how quickly and easily you can convert your VBA projects to VB.NET, using the same ActiveX® Object Model you're familiar with from VBA.

About the Speaker:

Augusto Gonçalves is a DevTech engineer from Autodesk, based in São Paulo, Brazil. With five years of previous experience before joining Autodesk, he has started supporting developers on AutoCAD® and Civil 3D®-based technologies, especially in Latin America. Augusto holds a Master's in Computer Science and B.S. in Civil Engineering.
augusto.goncalves@autodesk.com

Introduction

In my role as an API consultant I have had opportunities to help ADN partners migrate their VBA projects to VB.NET. Although the migration can be challenging, once you see what needs to be done the task will not appear so difficult. This document is intended to be a supplement to the powerpoint slide and examples used during the session and will be a resource after the class. There are six main topics:

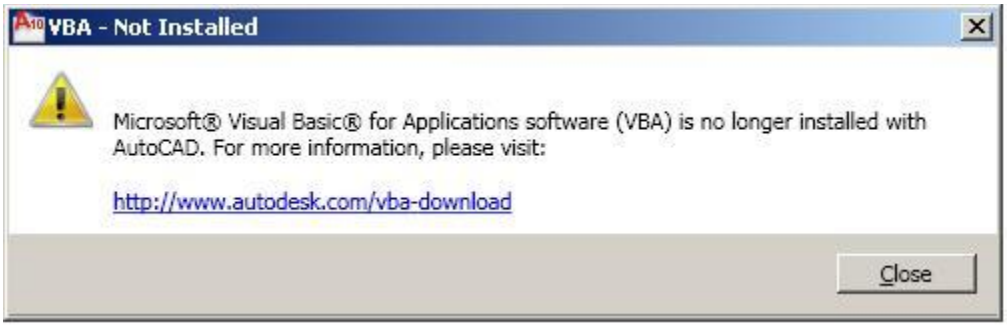
- Focus on basic concepts (why you will want to migrate, language comparison and the IDE).
- Hello world project.(Step by step)
- The migration process step-by-step.
- How use a helper macro to export VBA projects.
- Required corrections to work with events and forms.
- Troubleshooting tips

Note: The suggestions for migration that will be presented in this session should not deter you from learning VB.NET and the AutoCAD .NET API. The intent is to help you reuse, as much as possible, the programming logic already implemented in your ActiveX code being used in VBA.

Why migrate to VB.NET?

VBA is no longer being developed. Microsoft has announced the "Discontinuation of the VBA Licensing Program" (see http://msdn.microsoft.com/en-us/isv/bb190538.aspx) on July 2007. One major effect of this decision is that there is not a 64 bit version of VBA. (The VBA engine in AutoCAD 64 bit is the standard 32 bit VBA engine. It is implemented as a 32-bit component external to the AutoCAD process).

Also, as of AutoCAD 2010, it is necessary to download an installer to enable VBA in AutoCAD. We expect that VBA will not be included as a component AutoCAD for many more releases, so now is the time to start thinking about migrating your VBA to VB.NET. The tools and ideas presented in this class will help you quickly get up to speed on this process.



One benefit of migrating your VBA code is that once your ActiveX code is working in VB.NET, you gain access to many others APIs, including WinForms, ADO.NET and the AutoCAD.NET API. VB.NET is a true object-oriented language; which means that some unintuitive and inconsistent features like **GoSub**/**Return** and **DefInt** have been removed from the language.

Note: If your application uses Microsoft-based features, such as OLE Container Controls, Dynamic Data Exchange or DAO, take a look at the "Things to Consider Before Upgrading" in the MSDN library at this location: (http://msdn.microsoft.com/en-us/library/ywsayxak.aspx).

The AutoCAD ActiveX API

It is important to understand that VBA uses the AutoCAD ActiveX API, and the decision to discontinue VBA from Microsoft is only related to the VBA IDE and does not mean that the AutoCAD ActiveX API is going away. AutoCAD will continue to have this API and you can use the objects, methods and properties from other environments such as VB.NET.

This means that it is possible use the same object model you are comfortable using from VBA in your .NET code. This is possible because .NET can consume ActiveX objects as easily as pure .NET objects through its COM Interop functionality.

VBA / VB.NET comparison

From VBA you can only use ActiveX APIs. From VB.NET you can use the ActiveX API and the AutoCAD.NET API, including new .NET framework features. In fact, after your VBA code is migrated it will be called and run like a "pure" .NET project.

To run a VBA procedure you use the VBARUN command. AutoCAD .NET procedures are run like built-in commands. A .NET built project, (a .NET assembly) is loaded using the NETLOAD command. The dll can also be demand loaded (using registry keys). VB.NET code needs to be compiled with a separate compiler. (such as Visual Studio) Also embedding .NET assemblies in DWG file is not supported. The following table summarizes the main differences between VBA and VB.NET usage on AutoCAD.

	VBA	VB.NET
Compiler	Built-in (VBAIDE)	External tool
Load options	VBALOAD or embed in DWG	NETLOAD or using registry keys
Run options	VBARUN	Custom command on prompt

Visual Studio: the default .NET IDE

According to Microsoft, "the Visual Studio development system is a comprehensive suite of tools designed to help software developers create innovative, next-generation applications. It's the perfect work environment for application developers". There are different versions of this IDE, each one for each type of development environment and you can choose the one that best fit your needs.

Fortunately there is a free version called Express Edition. This version allows you to write and compile VB.NET project with almost all IDE resources available as the full version. It is important to remember that all versions of Visual Studio, including Express Edition, require permissions to install it on your machine. For more information and how to download, please visit the following link http://www.microsoft.com/express/vb/

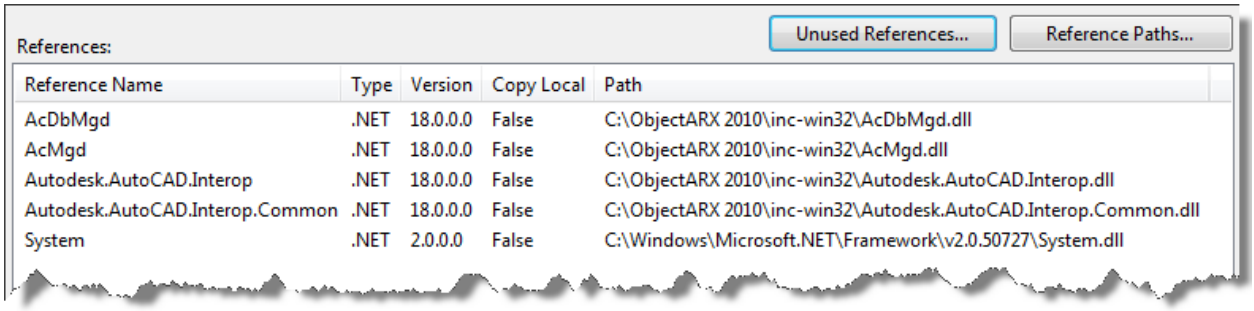
Note: Although .NET applications can run on both 32 bit and 64 bit platforms, the free version of Visual Studio does not allow you to compile for 64 bit by default. Visit the following link for more information: http://msdn.microsoft.com/library/we1f72fb.aspx. Later in this material we will present a workaround for this limitation.

Adding References for ActiveX

Like VB6, in VB.NET we need to add reference to both *AutoCAD XXXX Type Library* and *AutoCAD/Object Common XX.X Type Library*, where XXXX and XX.X stands for the version number. Projects created using the VBAIDE automatically add a reference to AutoCAD Type Library, which indirectly add reference to Object Common Type Library, so you do not have to do this manually.

A VB.NET project requires that we explicitly add a reference to the required assemblies. On a project created on Visual Studio, we can use three alternatives to add those references:

1. COM objects: this is the same as you would use in a VBA project.
2. TLB files: this is what the option number one actually does, but here you can manually browse and then add reference to **acax18enu.tlb** and **axdb18enu.tlb**, both located at *[Program Files folder]\Common Files\Autodesk Shared* folder (assuming default folder).
3. DLL files: here you can browse for the DLL version of AutoCAD COM Objects, which allows much more control over which version you are planning to support. These DLLs came with the ObjectARX SDK, which can be freely download from http://www.objectarx.com or, if you have ADN Subscription, you can download multiple versions of it from http://adn.autodesk.com. Once you have this SDK, browse for the following assemblies, **Autodesk.AutoCAD.Interop.dll** and **Autodesk.AutoCAD.Interop.Common.dll**, on *[ObjectARX folder]\inc-win32* or *\inc-x64*, to compile for 32 bits and 64 bits versions of AutoCAD. This is the recommended approach.



For example: suppose you have installed on your machine both AutoCAD 2008 and 2009. On this scenario, you can use COM objects or browse TLB files. If you plan to support both versions, it's recommended that you develop your application using 2008 references, but when you reference using options 1 and 2, your project will use the newer version, in this case 2009. To explicitly use the 2008 references, you should add references as described at option 3, using ObjectARX 2008.

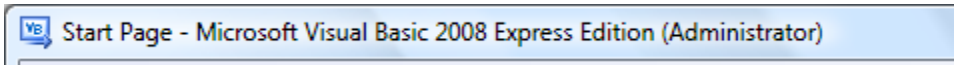
It is recommended that you compile your project with references to the oldest version you plan to support. For example: Let's say your application needs to work on AutoCAD 2008, 2009 and 2010. In this case, compile the code one time using the 2008 references for AutoCAD 2008 and 2009 and compile again using 2010 references for AutoCAD 2010. As result, you application will have two version, one for 2008/2009 and another to 2010. The "Unable to cast COM object" exception can be thrown in this case, see "Quick troubleshooting" section for more information.

Note: If you are planning to support 32 bits and 64 bits platforms, you must compile you project using the specific versions of the references. A project compiled with 32 bits version of references will not work on 64 bits because AutoCAD COM object have difference GUID on each platform.

“Hello World” in VB.NET with COM Interop

The following is a step by step example that creates a simple project using the ActiveX API.

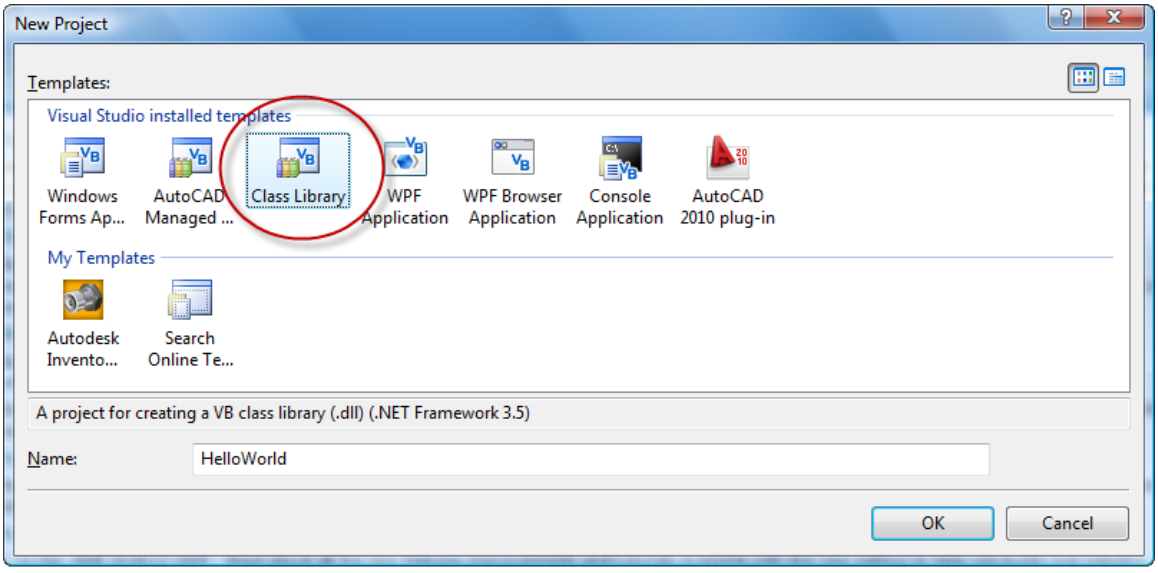
1. Start Microsoft Visual Basic Express Edition. Check if the application starts with *(Administrator)* on the title bar. If not, you may have to change your Windows Credentials to have this privilege. This is not mandatory, just recommended, see more info at MSDN (http://msdn.microsoft.com/en-us/library/ms165100.aspx).

2. Go to menu ***File***, and then select ***New Project***. Here you can select two kinds of projects: Windows Forms Application or Class Library. The first type will produce a stand alone .exe file that allows your application to automate AutoCAD. The second type will produce a .dll file that you can load inside AutoCAD and register new commands.

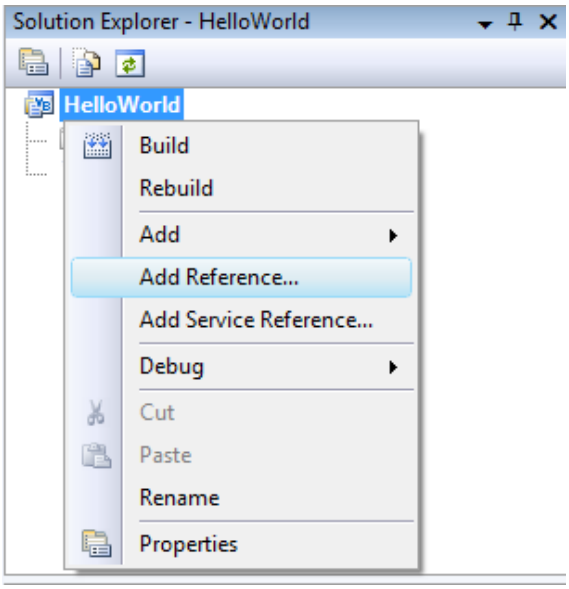
   As we are focusing on VBA migration used inside AutoCAD, let’s choose Class Library as project type like in the image below (this will allow us to load the dll). Type the desired name, in this case *HelloWorld*, and finally press “Ok” button. Visual Studio will then create a new project.

   Tip: the image below also shows two others interesting types of projects to extend or automate Autodesk products, *AutoCAD 2010 plu-ing* and *Autodesk Inventor Addin*, which are template projects for AutoCAD.NET API (also called AutoCAD Managed API) and Inventor COM API respectively. We can develop for both using the same IDE and language. Very cool, isn’t it? You may need to install these additional wizards separately, the AutoCAD wizard can be found at http://www.autodesk.com/developautocad, but works for pure .NET project and will not be used in this handout.

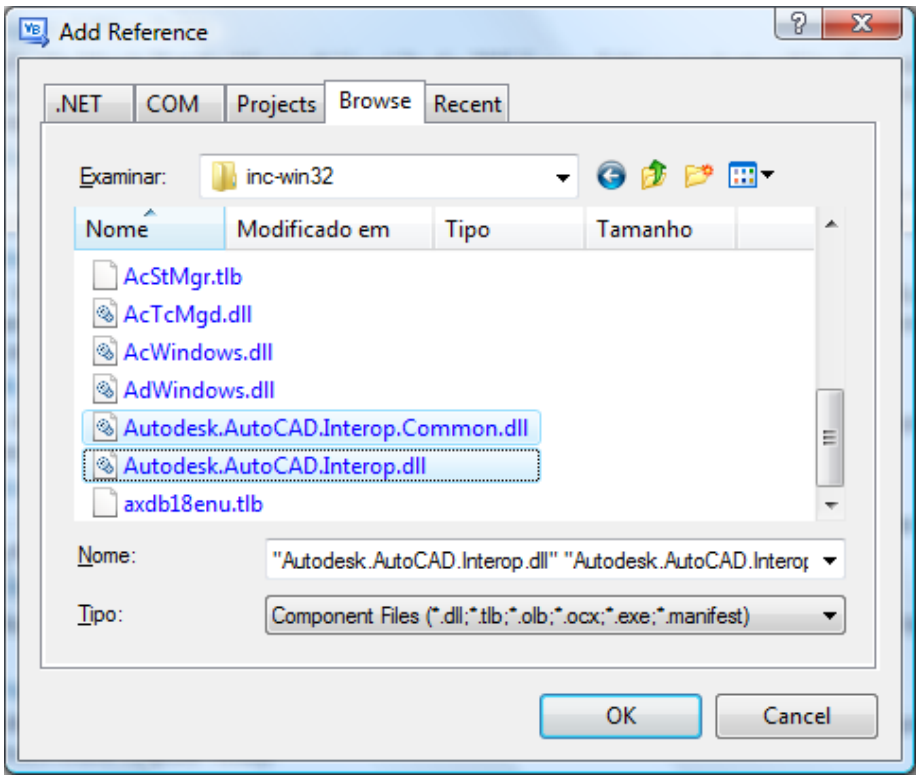
   

3. Now, at the *Solution Explorer* window, right click on **HelloWorld** project and then select ***Add Reference***. It is also possible go to project properties, select the “References” tab and click on “Add” and select “Reference”.

Tip: If this window does not appear on your VS IDE, go to menu **View**, and select **Solution Explorer**, or simply press Ctrl+Alt+L.

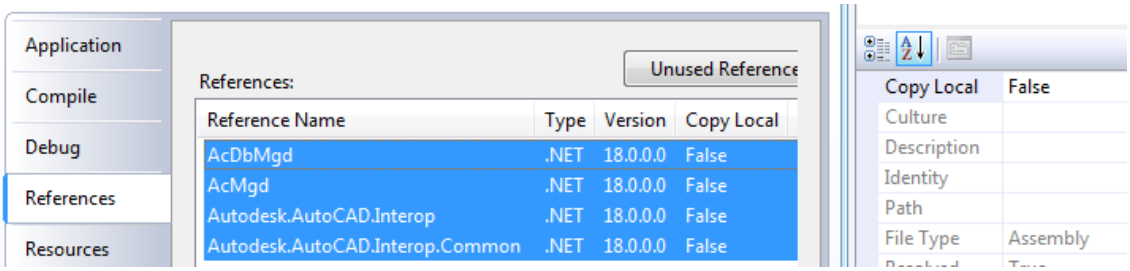


Following the recommendation described at section “Adding References for ActiveX on VB.NET”, add references to both assemblies available at ObjectARX folder, like the image below. Additionally, as we will use some AutoCAD.NET features to hook our project inside AutoCAD, so add references to **AcMdg.dll** and **AcDbMgd.dll** in the same folder.



Note: if your code is going to run outside of AutoCAD (external application) do not add a reference to AcMgd and AcDbMgd assemblies. Please refer to this section - "What about Automation from external application?"

Make sure all AutoCAD related references are set to Copy Local equals false. This is important to avoid duplicating these references and future debug problems. To set Copy Local to false, right-click on the project name **HelloWorld** and go to "Properties", then select the reference and go to the "Properties" window and change the Copy Local property. The final result should be like in the following image.



4. In .NET we need to import the namespace we want to use. This will dramatically reduce the amount of code we need to type and, as we are migrating from VBA and will allow your code to look very similar to the way it does in VBA. After importing the namespaces, the code inside *class1.vb* will look like the following code:

```
'import AutoCAD Type Libray namespace
Imports Autodesk.AutoCAD.Interop
'import AutoCAD/Object Common Type Library namespace
Imports Autodesk.AutoCAD.Interop.Common

Public Class Class1

End Class
```

5. VB.NET assemblies can define AutoCAD commands which will run sub routines. To "mark" a sub procedure as a command we need to add an attribute that defines the command name, like in the following code:

```
Public Class Class1
    <Autodesk.AutoCAD.Runtime.CommandMethod("commandName")> _
    Public Sub subName()

    End Sub
End Class
```

The *commandName* is what the user enters on the AutoCAD command prompt to call your *subName* routine. The *commandName* and the *subName* do not need to be the same.

Tip: Like we did for the COM namespace, you can also use the *Imports* keyword and import the Autodesk.AutoCAD.Runtime namespace to reduce the length of the commandMethod attribute above. <CommandMethod("commandName")> _

Note: AutoCAD commands do not return values, if you use a *function* instead of a *sub*, your assembly will throw "Error binding to target method" exception.

6. The AutoCAD.NET API does not have the *ThisDrawing* variable available in VBA. It does represent the active document inside AutoCAD, so we can instantiate a variable and then use the *Utility* to prompt a message, like in the following code.

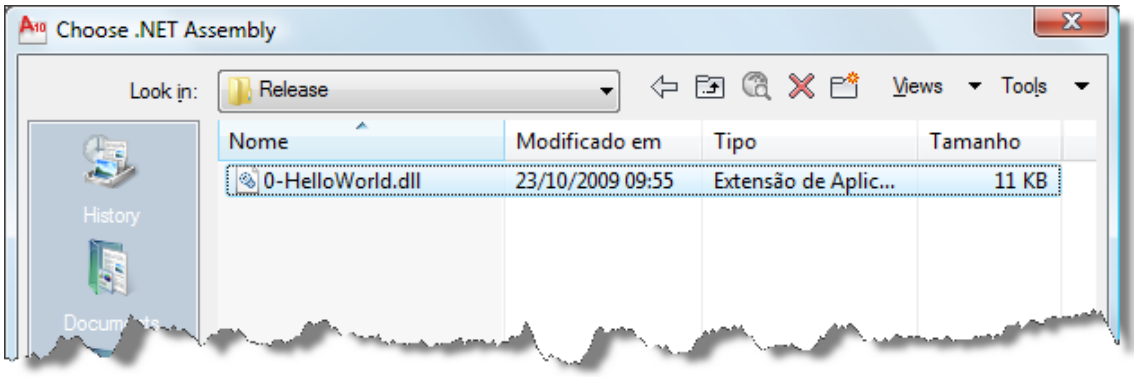
```
Public Class Class1
    <Autodesk.AutoCAD.Runtime.CommandMethod("commandName")> _
    Public Sub subName()
        'create an AcadDocument variable
        Dim ThisDrawing As AcadDocument

        'and get the Active Document
        ThisDrawing = Autodesk.AutoCAD.ApplicationServices. _
            Application.DocumentManager.MdiActiveDocument.AcadDocument

        'let's print "Hello World" on the prompt
        ThisDrawing.Utility.Prompt("Hello World from VB.NET")
    End Sub
End Class
```

7. Compile you project by selecting menu *Build, Build HelloWorld*. You should receive a “Build succeeded” message on left-bottom of Visual Studio status bar.

8. Start AutoCAD, call NETLOAD command. At the “Choose .NET Assembly” dialog select the *HelloWorld.dll* file on the output directory, usually at *[Project folder]\bin\Release* folder. Click “Open”.



Finally type the command name at the prompt. The AutoCAD prompt should look like the following:

Command: NETLOAD
Command: commandName
Hello World from VB.NET

Improving the ThisDrawing variable

One interesting feature of .NET languages is the ability to encapsulate the get and set methods of a variable’s property inside a single method, known as *property*, instead to separated *let* and *get* methods. So why not create a *ThisDrawing* property to reuse it across our code? This will increase the similarity of VBA and VB.NET code.

The following code shows a modified version of the **HelloWorld** step-by-step sample showed earlier. Notice we make the variable a *ReadOnly* property as it shouldn't be changed. This property returns an instance of AcadDocument type.

```
Public Class Class1
   'return the Active Document
   Public ReadOnly Property ThisDrawing() As AcadDocument
      Get
         Return Autodesk.AutoCAD. _
            ApplicationServices.Application. _
            DocumentManager.MdiActiveDocument.AcadDocument
      End Get
   End Property

   <Autodesk.AutoCAD.Runtime.CommandMethod("commandName")> _
   Public Sub subName()
      'let's print "Hello World" on the prompt
      ThisDrawing.Utility.Prompt("Hello World from VB.NET")
   End Sub
End Class
```

Note: You may not want to store the value of ThisDrawing as a variable. AutoCAD is a MDI (*Multiple Document Interface*) environment and the active document object will change every time the user changes between drawings. Later in this material we will cover how deal with events and how keep track of ThisDrawing variable.

Calling VBA code from VB.NET

During your migration process, you may need to run both VBA and VB.NET code side-by-side. It is possible call VBA routines from VB.NET using the following piece of code.

```
Public Sub runMacro()
   'create an AcadApplication variable
   Dim acadApp As AcadApplication
   'get the AcadApplication
   acadApp = Autodesk.AutoCAD.ApplicationServices.Application.AcadApplication
   'and run the macro
   acadApp.RunMacro("C:\ProjectFile.dvb!ThisDrawing.MyRoutine")
End Sub
```

The *RunMacro* method is well known on VBA, but here it was migrated to .NET. Note that if your project has the ThisDrawing property or variable, you can simply get the ThisDrawing.Application property and then call RunMacro.

Tip: As a .NET project defines a command, you can call them from VBA code using SendCommand.

Enabling Debug on Visual Basic Express

Developing an AutoCAD addin without being able to debug it is difficult. The Visual Studio Express editions do not enable this option through UI, but we can manually edit the .vbproj file to enable it. If you are editing a working project, make sure you have a backup of this file and Visual Studio is not running.

For this hello world sample, open the **HelloWorld.vbproj** file on a text editor, such as Notepad. Note that you can apply this to other Project files, such as .csproj for C# projects. Find the following piece of XML code and append the bold marked part, **StartAction** and **StartProgram** (with the desired AutoCAD version).

```
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
  <DebugSymbols>true</DebugSymbols>
  <DebugType>full</DebugType>
  <DefineDebug>true</DefineDebug>
  <DefineTrace>true</DefineTrace>
  <OutputPath>bin\Debug\</OutputPath>
  <DocumentationFile>HelloWorld.xml</DocumentationFile>
  <NoWarn>42016,41999,42017,42018,42019,42032,42036,42020,42021,42022</NoWarn>
  <StartAction>Program</StartAction>
  <StartProgram>C:\Program Files\AutoCAD 2010\acad.exe</StartProgram>
</PropertyGroup>
```

After the above steps, the F5 key or the menu *Debug, Start Debugging,* should work without problems. As a friendly reminder, debug features may not work if the application references are set to Copy Local equals true, please refer to step 3 of the previous “Simple Hello World” section.

What about Automation from external application?

First, if your application automates AutoCAD from an external application, also known as out-of-process, you cannot add reference to AcMgd and AcDbMgd assemblies. In fact, this is not necessary as this kind of application cannot define a command, but you can become tempted to use in-process features on an out-of-process application.

Tip: In case you want to automate AutoCAD using in-process .NET features, there is an interesting blog post about how create new COM interfaces using .NET and access them at the following link (http://through-the-interface.typepad.com/through_the_interface/2009/05/interfacing-an-external-com-application-with-a-net-module-in-process-to-autocad.html).

If your external application automates AutoCAD, then it is probably a VB6 project, not a VBA project. If so, in the following sections of this material we will cover some migration steps, and you may want to skip the export from VBA to VB6 step.

Step-by-step migration

In the following sections we will cover the basic steps required to migrate your VBA to the new VB.NET. These steps are generic, so you may need additional work depending on how your code was designed.

After understanding these steps, you will be more confident when using the VBA to VB6 utility and addressing more complex issues, such as migrating events. Before starting, make sure to create a **Backup Copy of Everything** you are planning to migrate.

First step: Prepare you code for migration

As will be discussed further in this text, the biggest part of migration process is performed by Visual Basic Upgrade Wizard, but there are some “Programming Recommendations” that will dramatically improve the final result.

Use Early-Binding instead Late-Binding

This is the most important recommendation you should consider, but what is early and late-binding? Every time you define variables as Object and/or Variant you are using late-biding, which means that your code will locate the executing method only on run time. To better understand what this is and how it works, please visit the following link.

Using early binding and late binding in Automation
http://support.microsoft.com/kb/245115

Now, suppose you are using late-binding to declare variables like below. This code will change the caption of the Label. Note that, during development time, there is no information on the type of *myLabel* variable, and only during run time the variable will know its type.

```
Dim myLabel As Object
Set myLabel = Me.Label1
myLabel.Caption = "SomeText"
```

Like many others, the caption property name has changed on VB.NET, and as the Wizard does not know the actual type of *myLabel*, it cannot migrate the property name properly. If we use early-binding, like below, the Wizard will migrate the property correctly.

```
Dim myLabel As Label
Set myLabel = Me.Label1
myLabel.Caption = "SomeText"
```

The same idea applies to Variant declarations and especially the use of CXxxx functions, such as CInt, CStr, etc. The following links show information you should consider before start your migration. I would highly recommend reading the topics from the first link. It can bring good improvements to your final result.

Further reading

The following links show information you should consider before start your migration, specially the first link, which can bring great improvements to your final result.

Preparing Your Visual Basic 6.0 Applications for the Upgrade to Visual Basic .NET
http://msdn.microsoft.com/en-us/library/aa260644.aspx

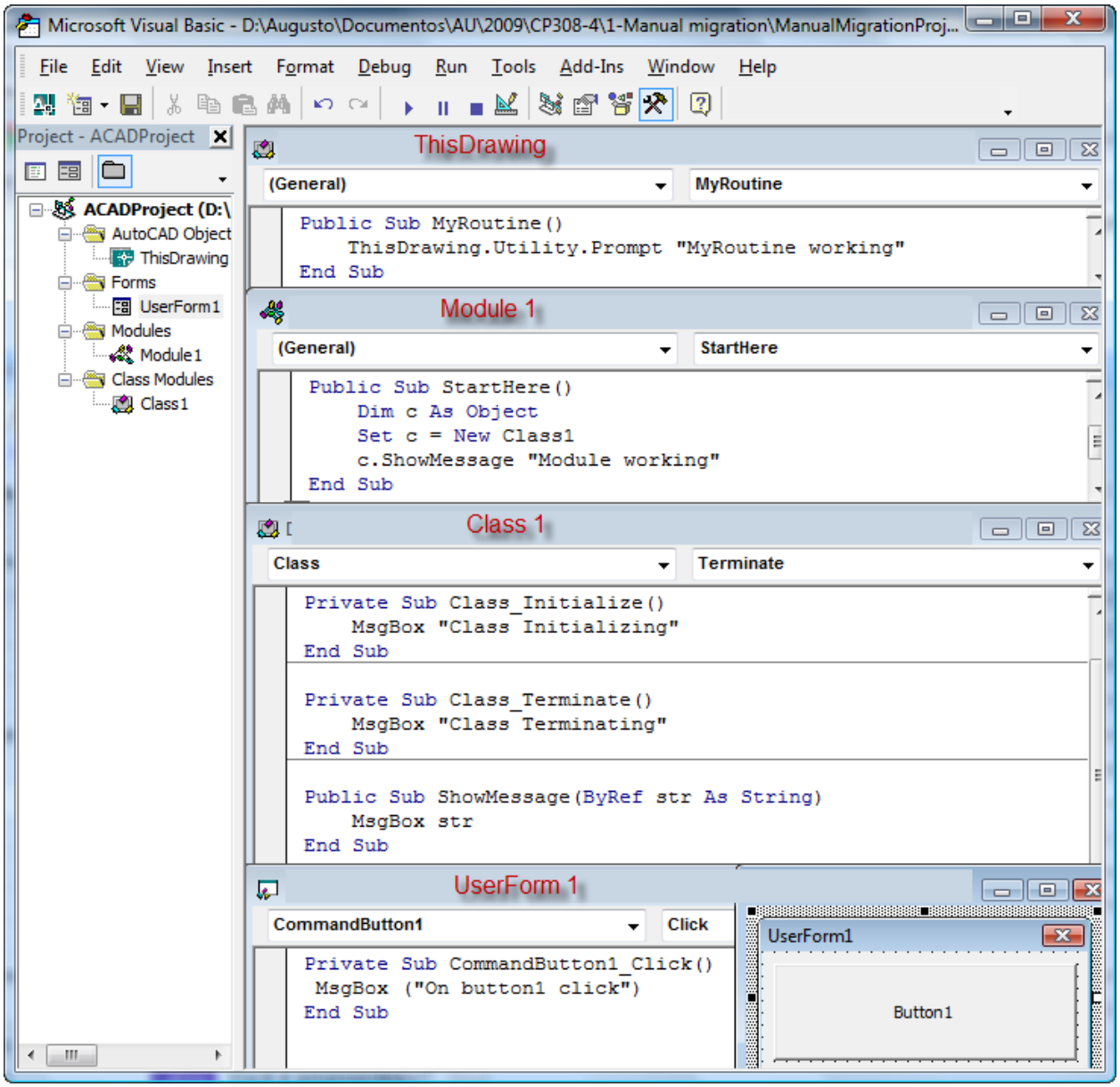
Preparing a Visual Basic 6.0 Application for Upgrading
http://msdn.microsoft.com/en-us/library/14w905kc.aspx

Convert VBA Code to Visual Basic When Migrating to Visual Studio 2005 Tools for Office
http://msdn.microsoft.com/en-us/library/aa537180(office.11).aspx

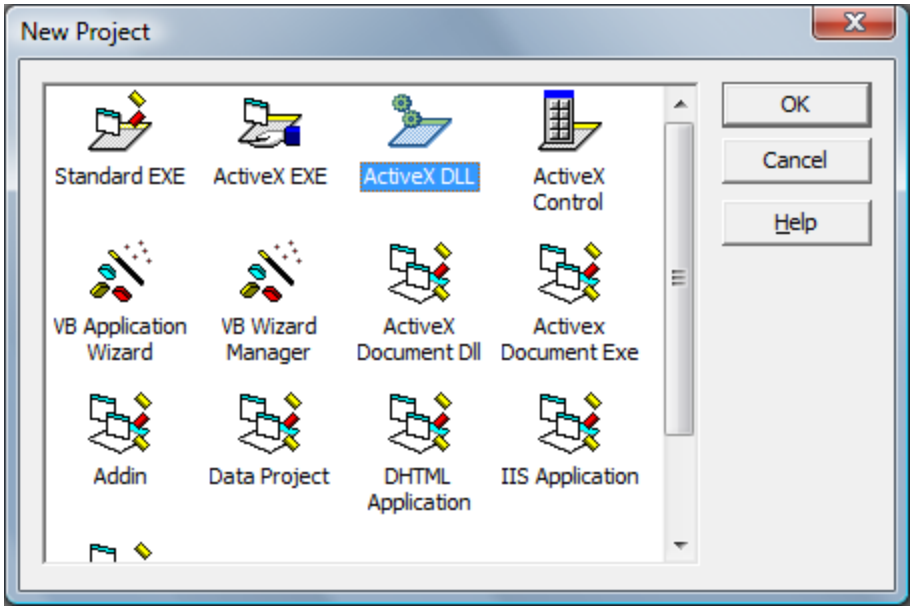
Second Step: Export your VBA project to VB6

The Visual Basic Upgrade Wizard migrates VB6 projects to VB.NET, not VBA projects. You will need to migrate your VBA project to VB6 to use the .NET wizard. Also, the process is manual because we need to export each item from VBA and import them inside a VB6 project. At this step you may face the big problem of manual migration – you need to install VB6. A utility to avoid using VB6 is discussed in a later section.

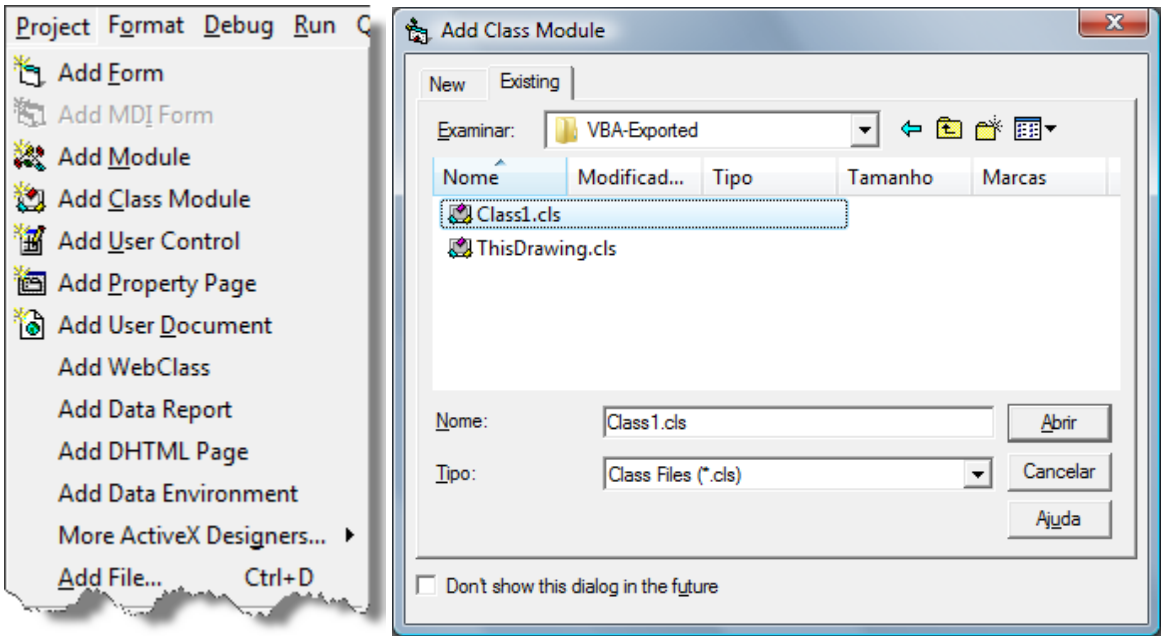
Suppose a simple VBA project that has one Module, one Class and one UserForm and uses the ThisDrawing shortcut like in the image below. Now for each of these items, we need to select menu *File, Export File* in order to export them. In this case, the result will be a set of .bas and .cls files.



Now start *Microsoft Visual Basic 6.0* and create a new project by selecting *File, New Project*, then a new **ActiveX DLL** project and click Ok like in the image below. A new blank project will be created and you don't need to configure its properties, we will configure it later on VB.NET.

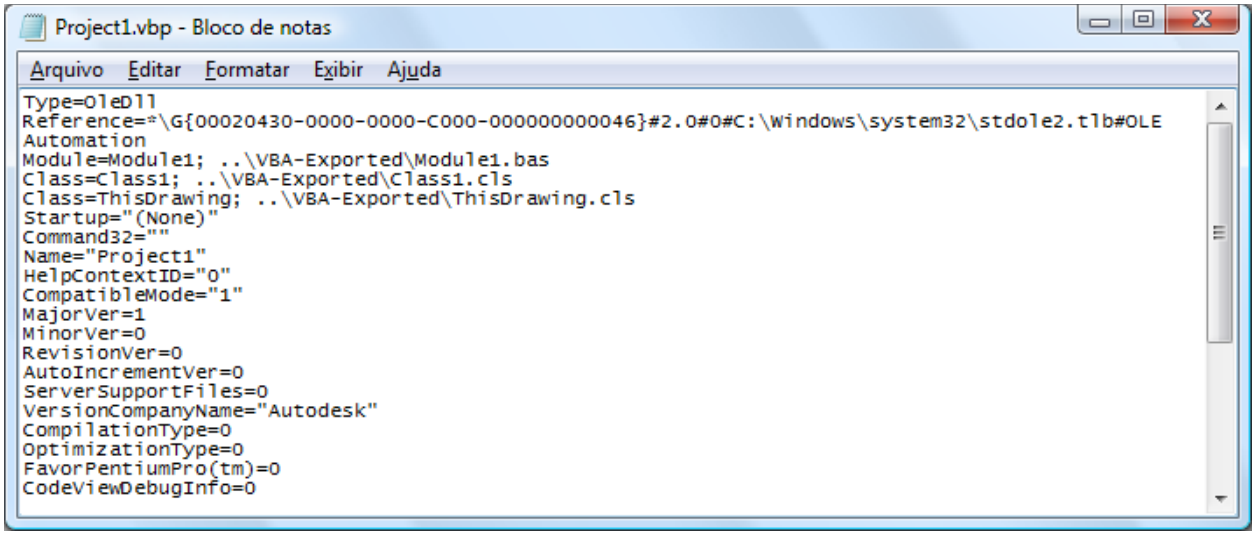


With the new project open on VB6, go to menu *Project,* and select *Add Class* for each class exported from VBA, *Add Module* for each module and *Add Form* for each user form. Inside the opening dialog, instead create new, select *Existing* tab like in the image below and locate the file.



Finally, save the VB6 project as a .vpb file. If you open this file with Notepad you will note that it is actually a text file, and it contains references to our .bas and .cls files. This is interesting because text files are easily editable and this will be the basic idea of the VBA to VB6 converter discussed in a later section.

And now the project is ready for migration, let's move forward.



Third Step: Use the Visual Basic Upgrade Wizard

We will use the Visual Basic Upgrade Wizard, but there are other tools and you can search on the web. You can use the default values for the wizard, such as “DLL/Custom Control Library” project type - just consider changing the destination folder if you like.

The upgrade wizard will generate one file for each VB6 item and three extra files:

- **AssemblyInfo:** according to MSDN, this file “provides properties for getting the information about the application, such as the version number, description, loaded assemblies”, see more at (http://msdn.microsoft.com/library/microsoft.visualbasic.applicationservices.assemblyinfo.aspx).
- **app.config:** is the application configuration file that contains settings specific to the application. You can find more information at (http://msdn.microsoft.com/en-us/library/ms229689.aspx).
- **_UpgradeReport**: contains important information about the migration process and, in this case, as presented in the image below, describe some warnings, which will be discussed further. Bottom line: you should consider analyzing this report before moving forward.

Upgrade Summary

Types of Issues Found	Learn More	Count
⊟ Warnings	There were some differences found that may causes unexpected results when your application is run. You should test your application thoroughly to verify that it functions as desired.	1
⊟ Some late-bound default property references could not be resolved	Late-bound default property reference could not be resolved	1
⊟ ThisDrawing.cls		1

#	Location	Object Type	Object Name	Property	Description
1	MyRoutine				Couldn't resolve default property of object ThisDrawing.Utility.

Total	1

First we will look at the code for Module1. We left a VBA late binded object “c” on purpose, to demonstrate the resultant code. Note that VB.NET did not know the type and makes type an Object. (And shows warnings whenever it is used). The upgrade is running “blind”, and does not know if the c object really contains a ShowMessage method.

This rest was migrated without relevant problems and therefore the result in the code below is pretty similar to the original, except for some syntax changes. (Such as the removal of the *Set* keyword and the use of parenthesis (…) for method calls.

```
Module Module1
      Public Sub StartHere()
            Dim c As Object
            c = New Class1
            'UPGRADE_WARNING: Couldn't resolve default property of object
            'c.ShowMessage.
            c.ShowMessage("Module working")
      End Sub
End Module
```

The upgrade wizard made some changes to the Initialize and Terminate methods. (now called New and Finalize). The Upgrade Wizard renamed those old methods appending the *_Renamed* suffix, and then calls them from the New and Finalize methods.

Advanced tip: As .NET uses a Garbage Collector mechanism, the exact time when the Finalize executes is undefined. Resources are not guaranteed to be released at any specific time.

Every .NET objects derives, implicitly or explicitly, from **System.Object**, that’s why **Class1** should call *MyBase.New* and *MyBase.Finalize*. According to MSDN library, the Object class “supports all classes in the .NET Framework class hierarchy and provides low-level services to derived classes. This is the ultimate base class of all classes in the .NET Framework.” (http://msdn.microsoft.com/en-us/library/system.object.aspx).

Tip: There will be several UPGRADE_NOTE comments all over the resultant code, which can help you understand what the upgrade wizard has done. These comments also include links to more information on MSDN website, but they were removed from this text for brevity.

```
Friend Class Class1
    'UPGRADE_NOTE: Class_Initialize was upgraded to Class_Initialize_Renamed.
    Private Sub Class_Initialize_Renamed()
        MsgBox("Class Initializing")
    End Sub
    Public Sub New()
        MyBase.New()
        Class_Initialize_Renamed()
    End Sub
    'UPGRADE_NOTE: Class_Terminate was upgraded to Class_Terminate_Renamed.
    Private Sub Class_Terminate_Renamed()
        MsgBox("Class Terminating")
    End Sub
    Protected Overrides Sub Finalize()
        Class_Terminate_Renamed()
        MyBase.Finalize()
    End Sub
```

```
    'UPGRADE_NOTE: str was upgraded to str_Renamed.
    Public Sub ShowMessage(ByRef str_Renamed As String)
        MsgBox(str_Renamed)
    End Sub
End Class
```

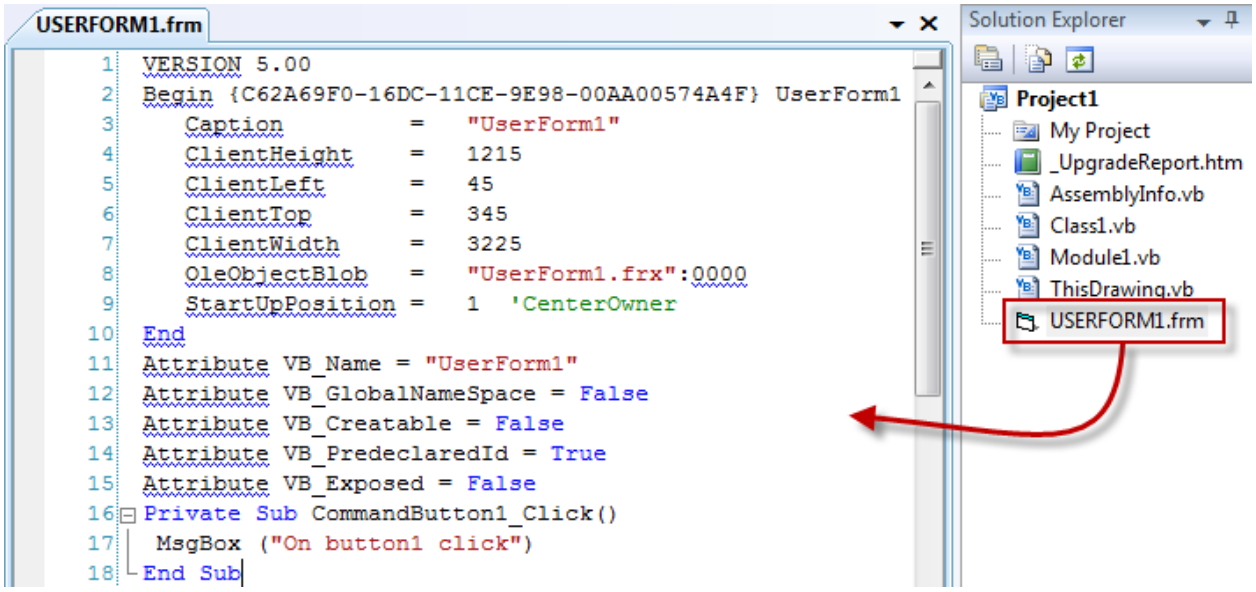
The VBA *ThisDrawing* object only exists inside VBA, therefore the Upgrade Wizard does not understand and shows an UPGRADE_WARNING comment in the code. This warning also appears at the _UpgradeReport file shown before.

Trying to solve this problem, the wizard creates to unnecessary lines which were strikeout in the following piece of code. We can safely cleanup these two lines. After removing them, you will get a compiler error which requires some adjustments.

```
<System.Runtime.InteropServices.ProgId("ThisDrawing_NET.ThisDrawing")> Public
Class ThisDrawing
    'UPGRADE_NOTE: ThisDrawing was upgraded to ThisDrawing_Renamed.
    Public Sub MyRoutine()
        Dim ThisDrawing_Renamed As Object
        'UPGRADE_WARNING: Couldn't resolve default property of object
        'ThisDrawing.Utility.
        ThisDrawing.Utility.Prompt("MyRoutine working")
    End Sub
End Class
```

Note: while the UPGRADE_NOTE contains interesting information, the UPGRADE_WARNING contains very important information that usually cause compiler errors and you should carefully review all of them. You can use the upgrade report to locate them.

Visual Basic 6 cannot handle user forms, therefore the Upgrade Wizard do not migrate them and then the resultant VB.NET project simply show the old VB6 code for user forms like in the image below. This is one of the main issues when migrating VBA projects to VB.NET. Fortunately the VBA to VB6 Converter, which will be presented later on this material, can help us on that.



Fourth Step: Adjusting the result

The required corrections were presented in "Hello World" section. First add reference to the two Interop .dll files, **AutoCAD.Autodesk.Interop** and **AutoCAD.Autodesk.Interop.Common**. Next, add reference to **AcMgd** and **AcDbMdg** to enable AutoCAD.NET support.

Tip: While adding reference to AutoCAD.NET assemblies, Visual Basic may prompt a "requires a later version of .NET framework" message. If so, please refer to the "*AutoCAD.NET assemblies (AcDbMgd or AcMdg) requires a later version of the .NET Framework*" topic in the "Quick troubleshooting" section.

Add the two *Imports* lines and the *ThisVariable* property. Finally, append the CommandMethod attribute in order to mark the *MyRoutine* method as a custom command. The result should look like the following (upgrade comments were removed for brevity).

```
'Import these namespaces
Imports Autodesk.AutoCAD.Interop
Imports Autodesk.AutoCAD.Interop.Common

'Add the ThisDrawing property
Public Class ThisDrawing
    Public ReadOnly Property ThisDrawing() As AcadDocument
        Get
            Return Autodesk.AutoCAD. _
                ApplicationServices.Application. _
                DocumentManager.MdiActiveDocument.AcadDocument
        End Get
    End Property
    'Add this attribute to mark this sub routine as a command
    <Autodesk.AutoCAD.Runtime.CommandMethod("myRoutine")> _
    Public Sub MyRoutine()
        ThisDrawing.Utility.Prompt("MyRoutine working")
    End Sub
End Class
```

Tip: In some cases your code may use the ThisDrawing variable from other modules, which will throw a compiler error, for example *'Application' is not a member of 'ThisDrawing'*. There are many ways to address this, but you can create a property with the appropriate name and returning type inside the above ThisDrawing class, like the following for Application.

```
Imports Autodesk.AutoCAD.Interop

Public Class ThisDrawing
    '...
    'the rest of your ThisDrawing class
    '...
    Public Shared ReadOnly Property _
    Application() As AcadApplication
        Get
            Return Autodesk.AutoCAD. _
            ApplicationServices. _
            Application.AcadApplication
        End Get
```

```
    End Property
End Class

'...
'somewhere else in your project
Public Module Module1
    Public Sub someMethod()
        'call the ThisDrawing.Application
        ThisDrawing.Application.Visible = True
    End Sub
End Module
```

Additionally, the module is upgraded without the *Public* keyword, therefore its commands are not properly recognized by AutoCAD. We should set it as public if we want to run it as a command. To run methods from Modules as commands, you must also append the CommandMethod attribute to all methods you want to use.

```
Public Module Module1
    <Autodesk.AutoCAD.Runtime.CommandMethod("StartHere")> _
    Public Sub StartHere()
        <<The rest of the code have not changed>>
```

The VBA to VB6 Converter: A "magic" helper tool

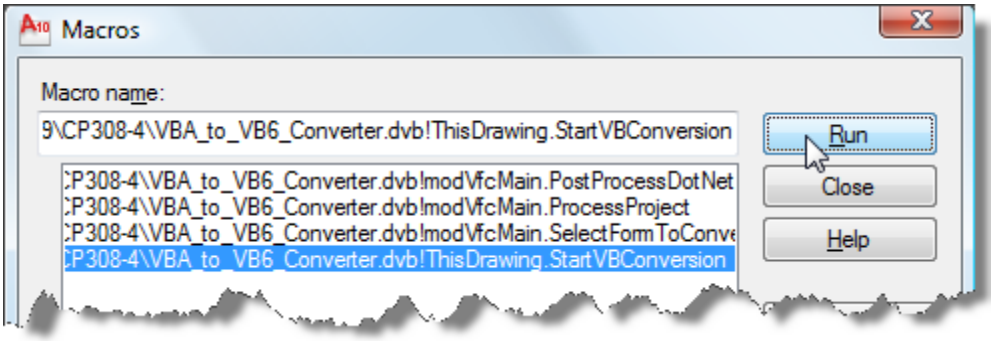
The **VBA → VB6 Converter** is a macro that exports VBA projects to VB6 format and make them ready for Visual Basic Upgrade Wizard without using VB6 IDE – no need to install VB6. It takes advantage of the fact that these project files are actually text files and can be edited by external tools. This is handy as export each VBA file and then import into a new VB6 project automatically, which save us from manually perform the "Second Step" section. This tool was originally written by Leslie Lowe (mookiha@hotmail.com) and here is presented with some new features. You can find the original version at http://discussion.autodesk.com/forums/thread.jspa?threadID=410953.

Additionally, the biggest benefit of this tool is converting User Form into VB6 forms. This allows the upgrade wizard to migrate them to new (and similar) VB.NET forms and controls, known as WinForms. The converter works for a limited number of supported controls, which are one of the mostly used. Here is the list of supported controls:

- MSForms.Label
- MSForms.TextBox
- MSForms.CheckBox
- MSForms.OptionButton
- MSForms.CommandButton
- MSForms.ToggleButton
- MSForms.Image
- MSForms.ListBox
- MSForms.ComboBox
- MSForms.ScrollBar
- MSForms.Frame

Note: The converter does not work for the MSForms controls *TabStrip*, *Multipage*, *SpinButton*, or any other external control. You may need to manually migrate them.

To use the converter, first VBALOAD the project you want to migrate, then VBALOAD the **VBA_to_VB6_Converter.dvd** file. VBARUN the **StartVBConversion** routine, as shown in the following image, which will start the migration dialog.



The converter form has 3 areas, one for each step, see image below:

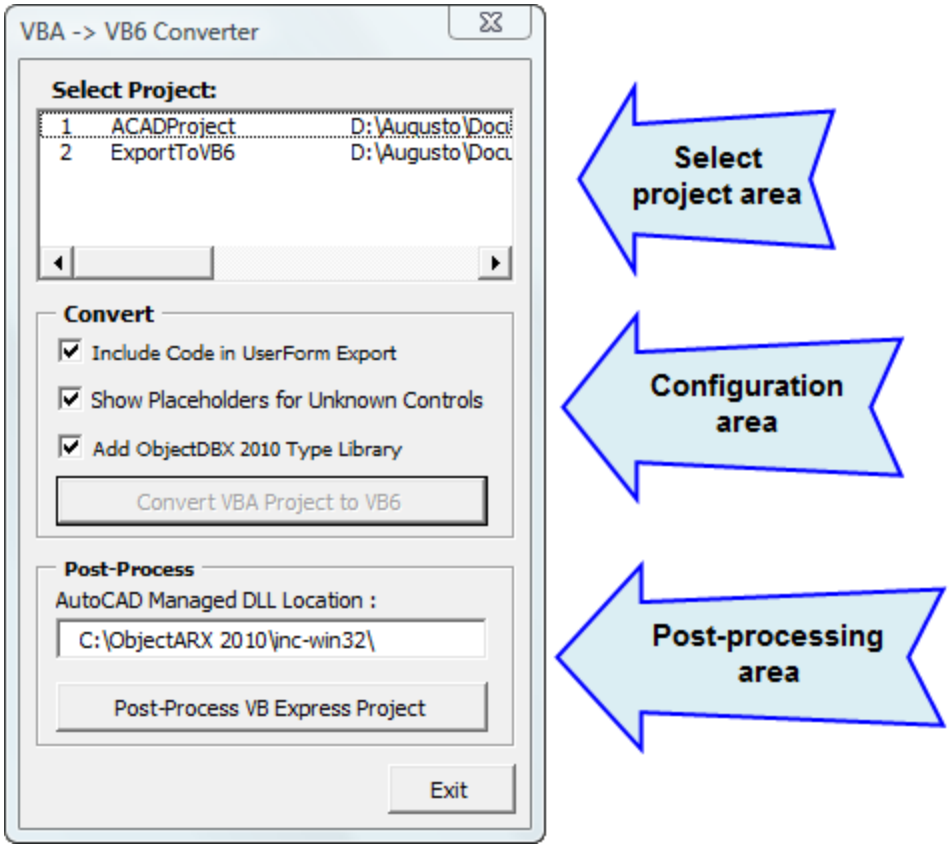
- First select the project you want to convert, which should be VBALOADed before running the converter macro. Also note that this list also contains the *ExportToVB6* project, which is the converter macro itself.

- The second allows us to change some settings:
  - Whether to include UserForms in the migration process, e.g. in case you have many unsupported controls you may not want to try converting them using this tool.
  - Include a place holder marking controls that the macro was not able to convert.
  - Add reference to ObjectDBX type library in case you use objects of this reference. (You probably do).

  After configuring all the settings above according to your project, click on “Convert VBA project to VB6” button. The resultant project will always placed at *[VBA project folder]\VB6Conv* folder.

- After using the upgrade wizard, the third area helps us configure the VB.NET project settings. To use it, first close the VB.NET project, then click the post-process button and select the .vbproj file of the upgraded project. It will make the following changes:
  - Set the target .NET Framework to 3.5, which avoid a warning message described at “Quick troubleshooting” section.
  - Enable debug and F5 shortcut on express edition (as shown on “Enabling Debug on Visual Basic Express” section).
  - Set the target platform to any CPU which removes the 32 bit limitation of express edition.
  - Add reference to required managed assemblies located at ObjectARX SDK folder. This will require the correct location of the SDK, so change the location textbox field if required.

  You can edit those references later and, for example, change the Interop versions. Please refer to section “Adding References for ActiveX on VB.NET” for more information.



Working with Events

Handling events on VB.NET is quite similar to VB6 and VBA. When a variable is declared as **WithEvents**, it means that a method can handle its events, but now the method that receives the event should be post-fixed by the **Handles** keyword. In general, the upgrade wizard should work well for variables that handle events, mostly because the syntax is similar, but does not migrate ThisDrawing events.

The ThisDrawing variable has a special characteristic inside VBA - it automatically changes its context according to the current document. Basically, the runtime environment automatically updates the variable and prepares its events for the new document. Let's see a quick example: suppose the following VBA code inside ThisDrawing module. It will run for every document without problem.

```
Private Sub AcadDocument_BeginCommand(ByVal CommandName As String)
   ThisDrawing.Utility.Prompt "(Begin command " & CommandName & ")"
End Sub
Private Sub AcadDocument_EndCommand(ByVal CommandName As String)
   ThisDrawing.Utility.Prompt "(End command " & CommandName & ")"
End Sub
```

After exporting it using the VBA to VB6 converter, upgrading using Visual Basic Upgrade Wizard and perform the "ThisDrawing cleanup" , the above VBA code becomes the following VB.NET code. To work with events properly we need more than a simple ThisDrawing property, so we cannot use the property we have been using in the previous examples.

```
Public Class ThisDrawing
    'UPGRADE_NOTE: ThisDrawing was upgraded to ThisDrawing_Renamed.
    Private Sub AcadDocument_BeginCommand(ByVal CommandName As String)
        'UPGRADE_WARNING: Couldn't resolve default property of object
        'ThisDrawing.Utility.
        ThisDrawing.Utility.Prompt("(Begin command " & CommandName & ")")
    End Sub

    'UPGRADE_NOTE: ThisDrawing was upgraded to ThisDrawing_Renamed.
    Private Sub AcadDocument_EndCommand(ByVal CommandName As String)
        'UPGRADE_WARNING: Couldn't resolve default property of object
        'ThisDrawing.Utility.
        ThisDrawing.Utility.Prompt("(End command " & CommandName & ")")
    End Sub
End Class
```

To adjust the above code will require a few steps. Check the following steps at the code below.

1. Import the required namespaces for interop. As showed earlier, this will make the VB.NET code more similar to the VBA equivalent and reduce manual changes. Additionally, to reduce the amount of code typing in this particular case, also import the ApplicationServices namespace.

2. As the ThisDrawing variable represents the active document, create the variable as AcadDocument type and set its value using the .NET Application object. Previously I have recommended to not store it as a variable, but using the read-only property instead. Here, to use events like VBA code, we need declare it using the **WithEvents** keyword, which cannot be applied to properties. This is why we will use a variable.

3. The active document variable should change every time the user switches between documents. To listen to these changes, we can get access to the collection of documents using the .NET object DocumentCollection and its value from the .NET Application object.

4. Using the above DocumentCollection object, create an event handler for DocumentActivated. Inside this event, just update the value of the ThisDrawing variable.

5. The upgrade wizard does not recognize the ThisDrawing variable, so it does not append the Handles keyword after events, so append it manually.

```
'<< Step 1 >>
'Import these namespaces
Imports Autodesk.AutoCAD.Interop
Imports Autodesk.AutoCAD.Interop.Common
'Another namespace - recommended here
Imports Autodesk.AutoCAD.ApplicationServices

Public Class ThisDrawing

    '<< Step 2 >>
    'Declare a ThisDrawing variable WithEvents
    Private WithEvents ThisDrawing As AcadDocument = _
        Application.DocumentManager.MdiActiveDocument.AcadDocument

    '<< Step 3 >>
    'We also need to keep track of the active document, so let's
    'get the DocumentManager, which control all opened documents
    Private WithEvents Docs As DocumentCollection = _
       Application.DocumentManager

    '<< Step 4 >>
    'every time the active document change, store it at thisDrawing
    Private Sub Docs_DocumentActivated(ByVal sender As Object, _
                                       ByVal e As DocumentCollectionEventArgs) _
                                       Handles Docs.DocumentActivated
        ThisDrawing = e.Document.AcadDocument
    End Sub

    '<< Step 5 >>
    'Append the Handles keyword for the migrated code
    Private Sub AcadDocument_BeginCommand(ByVal CommandName As String) _
                                          Handles ThisDrawing.BeginCommand
        ThisDrawing.Utility.Prompt("(Begin command " & CommandName & ")")
    End Sub
    Private Sub AcadDocument_EndCommand(ByVal CommandName As String) _
                                        Handles ThisDrawing.EndCommand
        ThisDrawing.Utility.Prompt("(End command " & CommandName & ")")
    End Sub
End Class
```

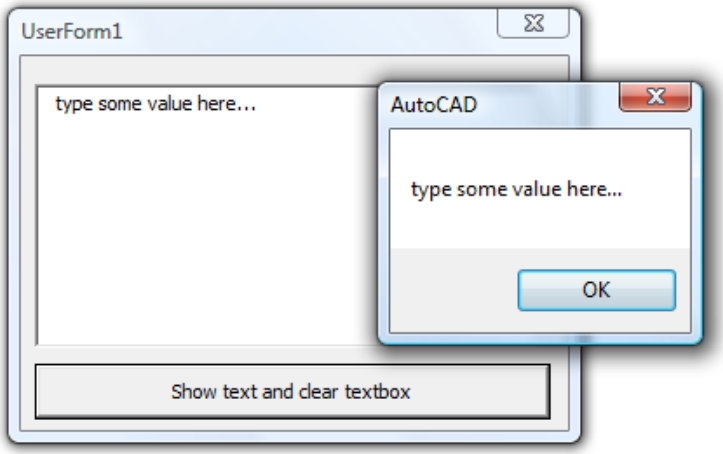
Advanced tip: the above code has a special requirement – it should start handling events for begin and end of commands when it is loaded, like its VBA equivalent, but this may not happen. Why? Because the ThisDrawing variable must be initialized, which in this case will not happen until the class is instantiated. If the class contains a command, then it will initialize when your user runs the command by the first time, but if it doesn't? On a .NET assembly, the class that implements the IExtensionApplication interface will run when it the assembly loads. You can either implement it at the above class or create another class that initializes all other objects you need. The following code snippet shows the class signature with the interface implementation. You must declare the Initialize and Terminate, but for this demonstration they do not require additional coding.

```
Public Class ThisDrawing
    Implements Autodesk.AutoCAD.Runtime.IExtensionApplication

    'Required when implementing the IExtensionApplication interface
    Public Sub Initialize() Implements _
        Autodesk.AutoCAD.Runtime.IExtensionApplication.Initialize
    End Sub
    Public Sub Terminate() Implements _
        Autodesk.AutoCAD.Runtime.IExtensionApplication.Terminate
    End Sub
    '...
    'the rest of the code . . .
    '...
```

Migration of UserForms

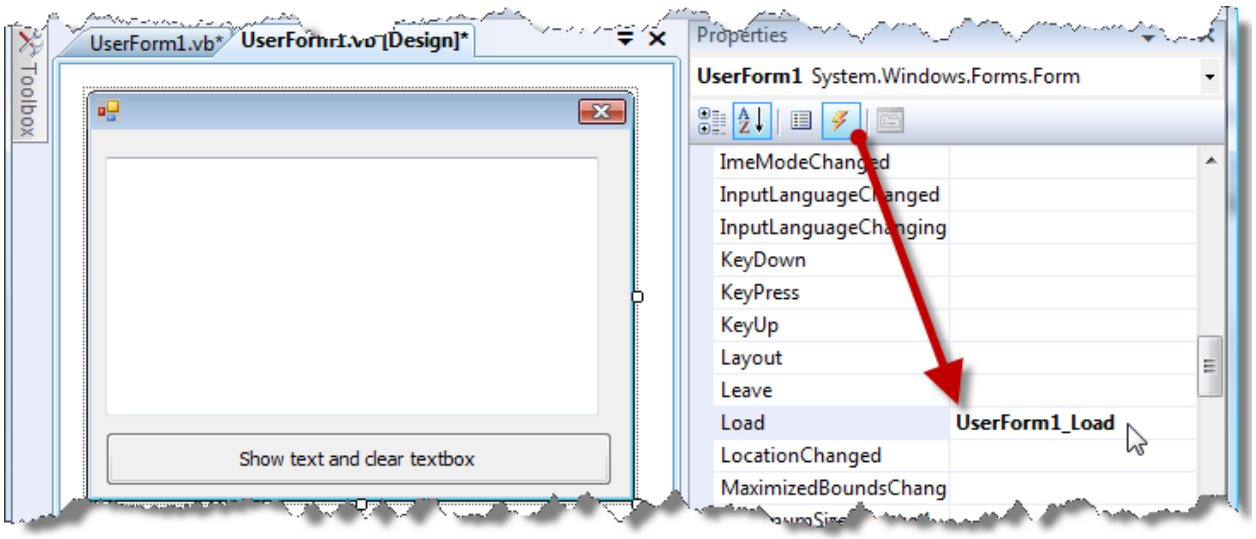
The VBA to VB6 converter works pretty smooth for projects with UserForms, but there is some additional information you may need. Consider a simple userform with a textbox field and a button. When the form initializes, it fills the textbox with an initial value and when the user clicks on the button, the textbox text is printed and cleared. The following image and code show it. For brevity there is no caller code.




```
Private Sub CommandButton1_Click()
   Call showTextboxText(TextBox1)
   TextBox1.Text = ""
End Sub
Private Sub showTextboxText(tbox As TextBox)
   MsgBox tbox.Text
End Sub
Private Sub UserForm_Initialize()
   TextBox1.Text = "The initial value"
End Sub
```

After running the VBA to VB6 converter and running the Visual Basic Upgrade Wizard, the .NET WinForm looks very similar to the original, but we need to make some corrections to the form events.

Visual Basic has a special way to handle events for forms and controls. To use it, select the form or the control you want, then at the Properties window click at the event icon ( ) to list all possible events, use mouse double click on **Load**, like in the following image. A new method to handle the event will be created at the code.



In the code you can either call the old initialize method, or copy and paste the original code inside the new method. The result will look like the following. Note that the **Handles** keyword was added to the button click event. For the specific case of the form, as the event is declared inside the base class, so use the **MyBase** keyword.

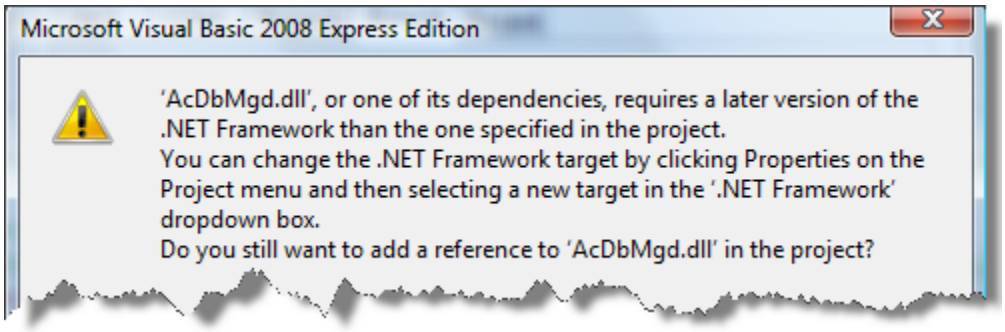
```
Friend Class UserForm1
    Inherits System.Windows.Forms.Form
    Private Sub CommandButton1_Click(ByVal eventSender As System.Object, _
                                     ByVal eventArgs As System.EventArgs) _
                                     Handles CommandButton1.Click
        Call showTextboxText(TextBox1)
        TextBox1.Text = ""
    End Sub
    Private Sub showTextboxText(ByRef tbox As System.Windows.Forms.TextBox)
        MsgBox(tbox.Text)
    End Sub
    'This initialize will not work, we need to create this event again
    Private Sub UserForm_Initialize()
        TextBox1.Text = "The initial value"
    End Sub
    'Here is the new Load event that replaces the Initialize
    Private Sub UserForm1_Load(ByVal sender As System.Object, _
                               ByVal e As System.EventArgs) _
                               Handles MyBase.Load
        'call the old initialize (or copy/paste the code here)
        UserForm_Initialize()
    End Sub
End Class
```

Tip: To ensure your new VB.NET project is handling events properly, you may want to check if all of them are implementing the **Handles** keyword.

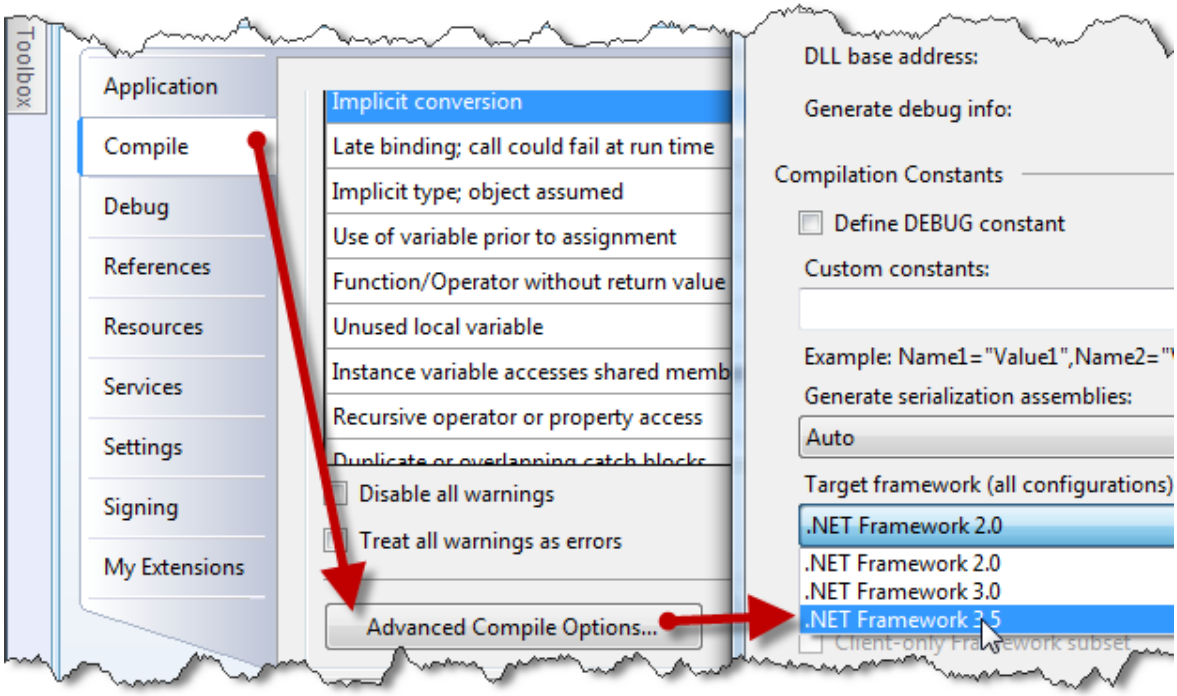
Quick troubleshooting

AutoCAD.NET assemblies (AcDbMgd or AcMdg) requires a later version of the .NET Framework

After run the Upgrade Wizard, while adding reference to AutoCAD.NET assemblies, you may receive the following message box. This warning is shows because, even with the wizard running on Visual 2008 for .NET framework, the migrated project is created for .NET framework 2.0, but AutoCAD uses the newer 3.5. You can simply select "Yes".



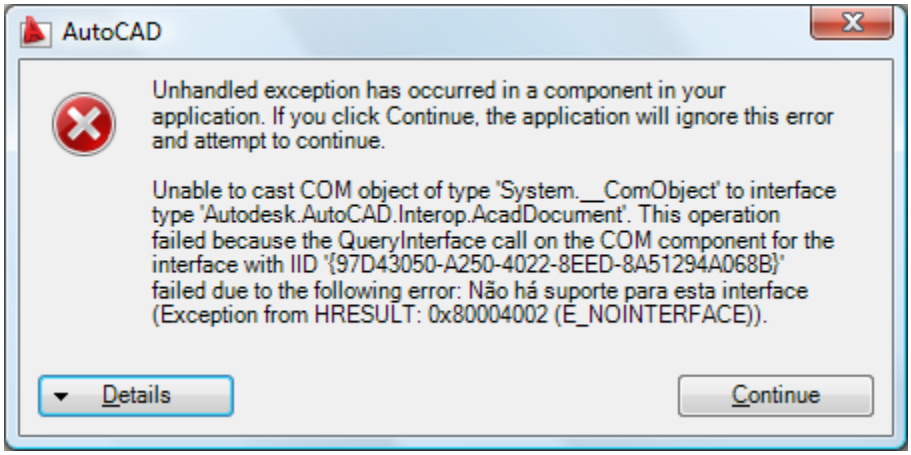
At the *Solution Explorer,* use mouse right click on project name, then go to Project properties. Like in the following image, select *Compile* tab, click on *Advanced Compile Options…* and change the "Target framework" to ".NET Framework 3.0", then select OK. Visual Basic will prompt you to save, close and reopen the project.



Unable to cast COM object exception

The exception shown in the image below is very usual when using AutoCAD COM objects from .NET. AutoCAD keep compatibility in cycles, so if your code was compiles using major version R17 (2007/2008/2009) of COM and you try running it on R18 (2010), it will throw this exception. A similar exception will occur if you try run a code compiled with 32 bits COM on 64 bits AutoCAD, or vice-versa.

To solve this exception, compile your project with the appropriate version of COM references, for each major version and platform.

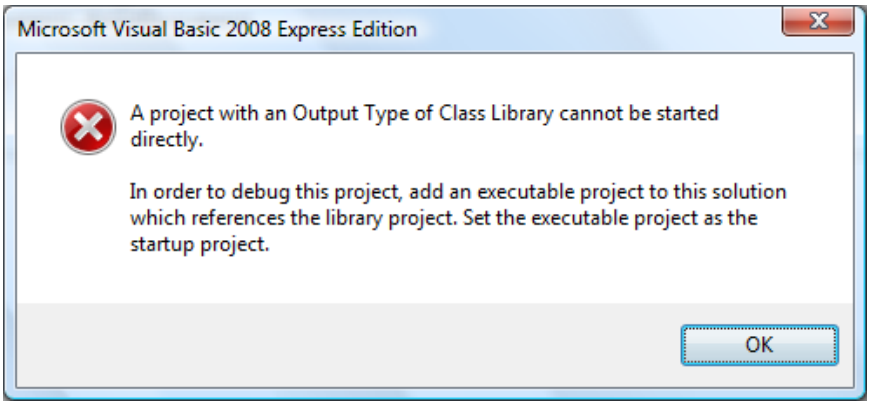


Project cannot be started directly

When you go to menu *Debug, Start debugging* or press F5, you receive an error message like in the image below. AutoCAD .NET add-ins must output a Class Library (i.e. a .dll file) and it cannot start by itself, you need an .exe file, in this case the **acad.exe**.

On the non-Express Visual Studio versions you can set an external program to run your application, but the Express Edition do not allow this configuration. Additionally, this limitation prevents us debugging our project. Fortunately, it is possible to directly edit the .vbproj file and enable debugging. Please refer to "Enabling Debug on Visual Basic Express" section or use the post-processing of the VBA to V6 converter tool.

For more information about the required changes, please visit the following blog post: http://through-the-interface.typepad.com/through_the_interface/2006/07/debugging_using.html



Conclusion

Thank you for attending this session on migration VBA to VB.NET using COM Interop. I hope you found the class enjoyable and valuable. In this handout I have presented information that will be very helpful when migrating your code.

You have seen a simple “Hello World” sample that demonstrates what is necessary to create a VB.NET assembly using COM Interop. We covered the basic migration process, using a step-by-step approach, including the VBA to VB6 converter which also provides a great improvement when dealing with UserForms. Finally we discussed how to handle events in the special case of the ThisDrawing variable.

This text has many “tips” and “important” information strategically located where make more sense. Also, when you start migrating your code, you may encounter one of the common issues listed at the troubleshooting section. For more information, consider visiting the links listed in the next section.

I am glad you are considering migrating your VBA code to this new great technology. Good luck and success on your work.

Further Reading

Through the Interface blog - http://through-the-interface.typepad.com
Kean Walmsley’s .NET focused blog includes several example codes for .NET

AutoCAD.NET Developer’s Guide - http://www.autodesk.com/autocad-net-developers-guide
.NET documentation with .NET (C# and VB.NET) samples with the equivalent VBA code

AutoCAD Developers Center – http://www.autodesk.com/developautocad
Training material, recorded presentations, and our AutoCAD .NET Wizards.

ADN DevTV: AutoCAD VBA to .NET Migration Basics –
http://download.autodesk.com/media/adn/VBA_Migration/DevTV_Recording.zip

Discussion Groups - http://discussion.autodesk.com/forums/category.jspa?categoryID=8

Information about the Autodesk Developer Network – http://www.autodesk.com/joinadn
ADN members can ask unlimited API questions through our DevHelp Online interface

API Training – http://www.autodesk.com/apitraining
Information about upcoming training classes and webcasts, also download of webcasts

Watch out for our regular ADN DevLab events. DevLab is a programmers’ workshop (free to ADN and non-ADN members) where you can come and discuss your AutoCAD programming problems with the ADN DevTech team.

The IDE Visual Basic Express - http://www.microsoft.com/express/vb