Have you ever wanted to program and customize AutoCAD Civil 3D but cannot seem to make the jump from VBA? We will walk through a real live project idea from start to deployment using Visual Basic® .NET and the AutoCAD Civil 3D .NET API. In the process, we will learn about proper referencing, object structure, transactions, and loading our built application into AutoCAD Civil 3D.

**Learning Objectives**
At the end of this class, you will be able to:
- Start/Create a .Net Project
- Build a .Net AutoCAD Command
- Interact with the Civil 3D .Net API
- Deploy the completed code

**About the Speaker**

Josh began working in construction in his teens giving him the opportunity to work on construction projects all throughout the southeast U.S. and even in the Caribbean. Because of his past, the natural transition was to Civil Engineering. Since graduating in 1996, Josh has had the privilege of working through differing stages - on hundreds of varied development projects around the world.

It was not long until Josh was employed as a CAD Manager. This became a true love for Josh - being able to help others use software to accomplish Civil Engineering Design - which showed as Josh won multiple national awards for the training articles written and customization provided.

Josh started with AutoCAD Release 12 and progressing through each release is now building .Net applications for AutoCAD Civil 3D 2013.

Currently, Josh serves as the Professional Services Manager for Advanced Technologies Solutions, an Autodesk Silver Partner and Autodesk Consulting Specialist.

joshua.modglin@gmail.com
Start/Create a .Net Project

Microsoft Visual Studio is provided in two different versions - Full and Express. Whereas Express is free and is limited in certain functionality, it does have many of the tools needed to begin creating very complex .Net add-ins for AutoCAD Civil 3D.

However, once you have chosen to use .Net the next big step is to choose a language that works **FOR YOU**. The great thing about .Net is that it takes whatever is written in the coding language and compiles it into a common language at runtime meaning for the coding part it makes no difference what language you choose from the capability option. Thus, the choice is left with what you are most comfortable with, what language has the most samples to learn from, and what other type of coding you anticipate doing in the future.

All AutoCAD Civil 3D .Net programming is going to be built on top of AutoCAD. Thus a first step would be downloading and installing the AutoCAD 2013 DotNet Wizard from [http://autodesk.com/developautocad](http://autodesk.com/developautocad). This allows us to choose to create an AutoCAD plug-in Project from the Project templates in Visual Studio. When creating a plug-in for AutoCAD 2013, make sure that the .Net Framework selected is 4.

Once you have named your plug-in, a wizard configurator opens and allows us to specify where the three required dlls are located for AutoCAD .Net applications. A best practice is to download the ObjectARX SDK (don’t worry about the name, we aren’t going to do any ObjectARX programming) from the same website as from above. The path to the required dlls to be entered in the Configurator would be the “inc” folder of where you installed the SDK.

Next, select the path to your AutoCAD Civil 3D’s acad.exe.
### Setting the Project’s Properties

In our Solution Explorer, if we right-click on our project’s name and select properties this will open a project’s properties tab.

![Screenshot of Solution Explorer with properties settings](image)

Notice that our assembly name (or the dll that will be made for our plug-in once everything is compiled) is the same name as our project. We can change the name of our output assembly but it is usually a good practice to keep it the same as our project.

#### Namespaces

Namespaces are what you might consider a directory structure (tree structure) for libraries. Thus, it is a good practice to assign namespaces that organize all your libraries under one major namespace (think Root directory).

The easiest way to make sure that all libraries we build are under the same root directory is to set the root directory to our company name.

For example, in our project the main root directory, or first namespace, will be “AU”.

Now since this project will include all functions, methods, etc that is specific to working with Civil 3D, we will add one more directory or namespace – “Civil3D”.

Thus, in our Root namespace we will have “AU.Civil3D”.

![Namespace example](image)
References
If we go down to the References option and click on this, we will note that two of the three required AutoCAD .Net references have been added for us. One of the references is new and the wizard does not pick it up. Thus we are going to add it manually using the Add... button. Using the Browse tab, go to the “inc” folder of the ObjectARX SDK that you installed previously and add the “AcCoreMgd.dll”. Finally, let’s add the two required references for the Civil 3D part of the work. These two assemblies (dll’s) are going to be in the same folder as your Civil 3D’s acad.exe. The five references are:

- acdbmgd.dll (in the inc folder of ObjectARX SDK. Required for AutoCAD)
- acmgd.dll (in the inc folder of ObjectARX SDK. Required for AutoCAD)
- accoremdg.dll (in the inc folder of ObjectARX SDK. Required for AutoCAD)
- AecBaseMgd.dll (in the Civil 3D acad.exe path. Required for AutoCAD Civil 3D project)
- AeccDbMgd.dll (in the Civil 3D acad.exe path. Required for AutoCAD Civil 3D project)

Since we are building an add-on, then when our software runs these references will already exist in and be called from AutoCAD. Thus, we need to set each of these references to not be copied with our project. We do this by selecting these newly added references and setting their “Copy Local” parameter to False in the properties window (View>Properties Window). When we are done our References should look similar to the figure below.

Build a .Net AutoCAD Command
Let’s focus now on that class in our Solutions Explorer. The Wizard creates two classes for us as samples. However, we are going to create our own classes so let’s delete those two class files (myCommands.vb and myPlugin.vb). Now, we are going to create a new class. We do this by right-clicking on our project name in Solutions Explorer and choosing to Add>Class... Name the class “Commands”.

Since this class is where we are going to store our commands there are certain libraries that we will be using quite extensively in this class. Thus to reduce typing, let’s Import these namespaces.

ABOVE where it says “Public Class Commands”, we want to add:

```vbnet
Imports Autodesk.AutoCAD.Runtime
Imports Autodesk.AutoCAD.EditorInput
Imports Autodesk.AutoCAD.DatabaseServices
Imports Autodesk.AutoCAD.ApplicationServices
```
Imports Autodesk.Civil.DatabaseServices

This reduces the need to type all of those namespaces (or directories) every time we want to use
something that resides in that namespace library.

Also so that Autodesk is aware and loads this class faster, we want to inform it that this class contains
commands. To do this we add the following line below the imports but above the class.

<Assembly: CommandClass(GetType(AU.Civil3D.Commands))>

So what we have so far is:

Imports Autodesk.AutoCAD.Runtime
Imports Autodesk.AutoCAD.EditorInput
Imports Autodesk.AutoCAD.DatabaseServices
Imports Autodesk.AutoCAD.ApplicationServices
Imports Autodesk.Civil.Land.DatabaseServices

<Assembly: CommandClass(GetType(AU.Civil3D.Commands))>

Public Class Commands
End Class

Just to do some clarification, the file (“Commands.vb”) is often referred to as a class. However, this is
just an envelope to hold an object or objects. So a file could contain multiple classes.

The Command

Now that we have our class file setup, let’s actually build the command. The command is what will be
typed into the AutoCAD command line to fire an associated method. We are going to name our
command “ChangeAlignmentStyles”. To build our command, we will write the following inside the
“Commands” class:

Public Class Commands
    <CommandMethod("ChangeAlignmentStyles")> _
    Public Sub cmdChgAlignStyles()
        End Sub
End Class

So we have a command name of “ChangeAlignmentStyles” assigned to a public sub named
“cmdChgAlignStyles”. So when we have our library (dll) loaded into AutoCAD and a user types the
command name, the sub routine will fire.

So now that we have our command built we need to add meat to the command.

We will be interacting with the user through the command line so first we will get the editor which
allows us to input to the command line and also get input from the user at the command line. We get
the editor of the current document using the following:
Dim ed As Editor = Application.DocumentManager.MdiActiveDocument.Editor

The first step in our function is to check to see if there are any alignments in the drawing to begin with. We do this by first getting the current CivilDocument object and checking to see if the document contains any alignment ObjectId's. An ObjectId is an AutoCAD unique id assigned to an object when the document (drawing) is opened. We get this using the following code:

```vbcsharp
Dim alignIds As ObjectIdCollection = civDoc.GetAlignmentIds
```

Since the alignIds object is a collection, all we have to do is check to see if there are any objects in the collection or if it is an empty collection. If it is empty we tell the user that we are ending, why, and exit the method.

```vbcsharp
If alignIds.Count = 0 Then
   ed.WriteMessage(vbLf & "No alignments in drawing.")
   Exit Sub
End If
```

Notice that we call a method called WriteMessage which is in the Editor. This writes to the command line. By adding the “vblf”, we make sure that our message is on a new line.

---

**Interact with the Civil 3D .Net API**

Now we know there are alignments in the drawing if we are passed this point in our code. We simply have to get the alignment names.

---

**Transactions**

To get anything from the document’s database, you need to query it. To query you need to start a Transaction which ‘opens’, or connects you to the drawing database. The tendency is to leave the connection open for however long you need to complete your function. However, since you may not be the only one working with the database even when you are in your function, you need to do what you are there for only and get out. Once you are out, the objects obtained while you were in the connection are no longer valid.

For example, you start a Transaction and get an alignment object. When you close the Transaction, you really can’t work with the alignment object. Now since we are just querying and not editing the database, we probably could leave the Transaction open but it is just not good practice so how do we get the data and use it OUTSIDE of the Transaction?
Creating Classes
To accomplish this, we are going to create classes which will store the information for us.

What is a Class?
Oftentimes we refer to the file holding a class object as a class. However, classes in .Net are actually objects. For example, you have in Civil 3D an Alignment object. This alignment object INHERITS the traits and properties of a base AutoCAD object. You can create your own special object in .Net with its own properties, methods, and functions by creating a new class. Then you can use these for presenting user interfaces to the user and accomplish quicker coding.

Creating a Base Class
For example, in our project we need a user interface where the user can select a new alignment style and from a group of alignments the user can use check boxes to select which alignments they want to change styles to match the style selection.

So we are going to create a base class and populate this base class using functions. We do this by right-clicking on our project name in Solutions Explorer and choosing to Add>Class... Name the class “BaseC3DObject”.

This class is going to have two properties - Name and Id. We add properties using the following:

```vbnet
Private oId As ObjectId = Nothing
Public Property Id As ObjectId
    Get
        Return oId
    End Get
    Set(value As ObjectId)
        oId = value
    End Set
End Property

Private oName As String = ""
Public Property Name As String
    Get
        Return oName
    End Get
    Set(value As String)
        oName = value
    End Set
End Property
```

We have a variable defined which is only available within the class (declared Private) set to something as default. Then we create a property which is available everywhere (declared Public) and store the value for use.

Can you think you think of another property that belongs to all C3D objects? Maybe a description?

Finally, we will add a fourth property called IsSelected. This property will be a Boolean (or a True/False).
Creating Functions

Now that we have a BaseC3DObject class built, let create some functions that will create a list of these that we can use. To do this we are going to create a third class file (Solutions Explorer>right-click on Project and Add Class). The class will be called Functions.

Our first function will return a list of BaseC3DObject class objects that contain data about all the alignments in the drawing. We will include an Argument called “IncludeDrefs” which will allow the function to filter the list to include only alignments that reside in the drawing or also include referenced alignments. We also set our function to be shared so that we can call it without creating an instance of the functions class (remember this is an object). What we have so far is:

```csharp
Public Shared Function GetAlignments(IncludeDrefs As Boolean) As List(Of BaseC3DObject)
End Function
```

We are first going to do a check similar to what we did with our command at the beginning. We are going to see if there are any alignments in the drawing to begin with. If not, we will return nothing.

```csharp
Dim alignIds As ObjectIdCollection = civDoc.GetAlignmentIds
If alignIds.Count = 0 Then Return Nothing
```

Finally, let's create a Transaction and iterate through our ObjectId collection to create a list (or collection) of BaseC3DObjects that will contain the data of the alignments.

Since we want the Transaction to be disposed of immediately after our completion with it, we are going to call it through the Using statement.

```csharp
Dim aligns As New List(Of BaseC3DObject)
Dim aBobj As BaseC3DObject
Using trans As Transaction = _
    HostApplicationServices.WorkingDatabase.TransactionManager.StartTransaction
End Using
```

What we have done above is called on the working database (or the guts of the current drawing) to start (create) a new Transaction and assigned this Transaction to the “trans” variable. The Transaction will be disposed of at the End Using statement.

Now we will create a loop and taking the id’s, we will get the alignment objects. From them we will create a new BaseC3DObject and add it to our BaseC3DObject’s collection. Of course, don’t forget we have to check against the whole Dref argument. Finally, we return the collection of BaseC3DObjects.

Don’t forget to include the imports to the namespaces in these class files!

It’s always a good practice to ‘wrap’ the inside of your using statement with a Try/Catch/End Try statement for your transaction. This way, we know the transaction will always complete cleanly.
Dim align As Alignment
For Each id As ObjectId In alignIds
  Try
    align = trans.GetObject(id, OpenMode.ForRead)
    If IncludeDrefs = False Then
      If align.IsReferenceObject = True Then Continue For
    End If
    aBobj = New BaseC3DObject
    aBobj.Id = align.Id
    aBobj.Name = align.Name
    aBobj.Description = align.Description
    aligns.Add(aBobj)
  Catch
  End Try
Next
Return aligns

We will create another function in this class to do the same for collecting all the alignment styles within the drawing.

Since it is getting the styles, the function will not need an IncludeDrefs argument. Also instead of a collection of alignment object ids, we will get the AlignmentStylesCollection using the following:

Dim alignStyles As Styles.AlignmentStyleCollection = civDoc.Styles.AlignmentStyles
Since a C3D drawing MUST always have at least one alignment style, you don't have to check if the count is greater than 0. Finally, the object type that we will work with in our transaction is AlignmentStyle instead of Alignment. Otherwise, the code is almost identical.

Public Shared Function GetAlignmentStyles() As List(Of BaseC3DObject)
Dim alignStyles As Styles.AlignmentStyleCollection = civDoc.Styles.AlignmentStyles
Dim aligns As New List(Of BaseC3DObject)
Dim aBobj As BaseC3DObject
Using trans As Transaction = _
  HostApplicationServices.WorkingDatabase.TransactionManager.StartTransaction
  Dim align As Styles.AlignmentStyle
  For Each id As ObjectId In alignStyles
    Try
      align = trans.GetObject(id, OpenMode.ForRead)
      aBobj = New BaseC3DObject
      aBobj.Id = align.Id
      aBobj.Name = align.Name
      aBobj.Description = align.Description
      aligns.Add(aBobj)
    Catch
    End Try
  Next
End Using
Return aligns
End Function
Creating the User Interface

We have all the means to populate our user interface so let’s go ahead and create it. We are going to use Microsoft’s WPF style to do this. Similar to adding a new class, we are going to add a new Window. We are going to name this window “winAlignStyles”. WPF (Windows Presentation Foundation) uses a format similar to XML to create the interface. The idea is the interface directly connects to the data oftentimes through a view model. We won’t spend time in the handout talking about the interface. However, the dataset provided with this handout will show how to connect the data to the interface.

Using the Functions

Now that the functions are created, we need to call the functions within our command in our Commands class. We create two lists - one for styles and for the alignment objects.

```
Dim aligns As List(Of BaseC3DObject) = Functions.GetAlignments(True)
If aligns Is Nothing Then Exit Sub

Dim alignStyles As List(Of BaseC3DObject) = Functions.GetAlignmentStyles
If alignStyles Is Nothing Then Exit Sub
```

We will create a new instance of the window object, winAlignStyles, we created above. When creating a new instance, we have programmed it in such a way that you must provide two lists as arguments - one for alignments and one for styles. Now we will call the method to show the dialog to the user. If the result is that the user clicked anything else but the okay button to close the window, then the command will end.

```
Dim win As New winAlignStyles(aligns, alignStyles)
If win.ShowDialog <> True Then Exit Sub
```

Building the Actual Code

It is amazing that the code to actually do the work is minimal compared to interacting with the user. However, the key here is that we have interacted cleanly with the user, AutoCAD, and Civil 3D. Now let’s go back to the command routine and actually adjust our selected alignments.

Since we know the window returned okay because we are still in our routine and did not exit, we will fire a Transaction and adjust the alignments. However, this is a special Transaction since we are actually doing more than querying the database but making changes to it. Consequently, we need to lock out anyone else from making changes at the same time. Also, since we are making changes we need to commit these changes to the database at the end of the Transaction.

```
Try
    Using trans As Transaction = _
        HostApplicationServices.WorkingDatabase.TransactionManager.StartTransaction
        Try
            Dim align As Alignment
            For Each bCo As BaseC3DObject In win.Aligns
                If bCo.IsSelected = False Then Continue For
```
align = trans.GetObject(bCo.Id, OpenMode.ForWrite)  
align.StyleId = win.cboStyle.SelectedItem.Id  
Next  
trans.Commit()  
Catch  
End Try  
End Using  
Catch  
End Try  
End Using

That is IT! We have done it!

**Deploy the Completed Code**

Of course, chances are you want to share what you have done with others and want to make this as automatic as possible. Autodesk came up with an improvement to automatic deployment through the use of the application plug-ins folder and package contents file.

All you have to do is copy your dll to the `C:\Program Files\Autodesk\ApplicationPlugins` directory in a special folder for your project. For example, `myproject.bundle` folder. The two files required to exist in this folder is your project assembly (the dll) and a `PackageContents.xml` file. A sample file exists in the dataset with this handout.

More information about the package contents file is located at: http://docs.autodesk.com/ACD/2013/ENU/files/GUID-5E50A846-C80B-4FFD-8DD3-C20B22098008.htm

The folder must contain “.bundle” at the end of the name.