



Nothing But .NET: Taking Advantage of the AutoCAD® Civil 3D® .NET API

Joshua Modglin – InMotion Consulting

CP430-2 This is the class for all those who ever wanted to learn how to program AutoCAD Civil 3D using the .NET API but have never had the time. 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.

About the Speaker:

Josh Modglin is a managing partner with InMotion Consulting. He began working in construction in his teens. Thus in school, the natural transition was to Civil Engineering. Since graduating in 1996, he has worked on hundreds of development projects around the world.

It has been a fun and bumpy ride - starting with AutoCAD Release 12, progressing to Eagle Point, dealing with AutoCAD Release 13, seeing the light with AutoCAD Release 14 and Softdesk 7, beginning to customize with Land Desktop 2000, losing hair on a production project using Civil3D 2005, and now building .Net applications for AutoCAD Civil3D 2011.

joshua.modglin@inmotioncon.com

STARTING A PROJECT

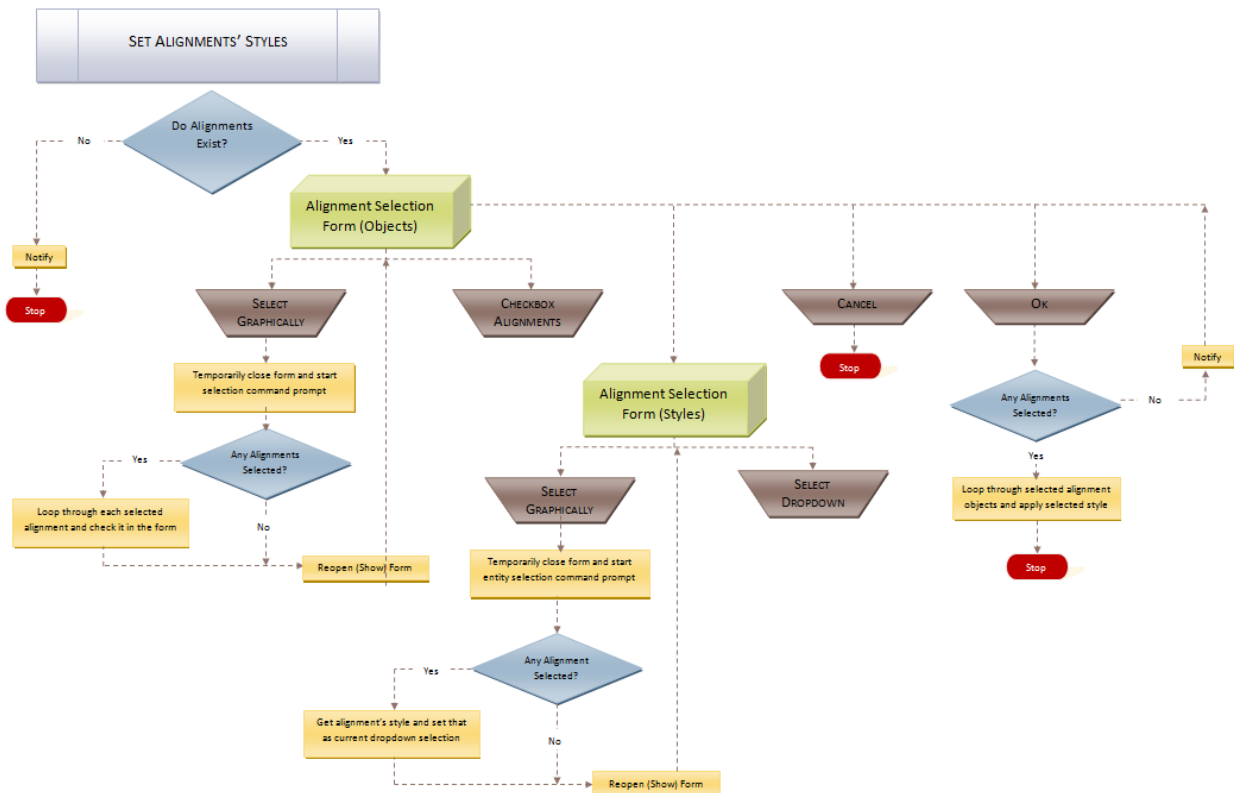
To start any type of project we have to not only know where are going (or the end result) but also how we are going to get there (steps involved). This is an oft forgotten truth of software projects and tools as well.

SOFTWARE REQUIREMENTS SPECIFICATION

For any software project – even the small ones – it is good to create a requirements specification. The basic idea is write out what the software is, the user interface, and functions needed to accomplish the task of the project. As there is for anything else, there is a standard called IEEE-STD-830-1998. However, whatever system you use just make sure you use one.

FUNCTION FLOW CHART

Another method that works better for the functions themselves is a flow chart. Let's do this for our main function within the project. We are going to create a function that allows us to select the alignments within the current drawing from a list or alternatively graphically and apply a selected alignment style to these selected alignments.



BREAKDOWN OF THE FLOW CHART

As we can see from the flow chart, we first check to see if there are any alignments in the drawing. Next we will open up a form (dialog box) which will have six options two for selection of the alignments, two for selection of the alignment style, and two general options (ok and cancel).

The options for selecting an alignment are either from a list or graphically. First, is a list with check boxes next to it. The user will then check the alignments he wants to apply the new style to. Alternatively, the user can select a button which will allow them to select the alignments graphically.

The options for selecting the alignment style are either from a combo box or graphically. First is a combo box with a list of all the alignment styles within the drawing. Alternatively, the user can select a button which will allow them to select the alignment style by selecting an alignment object.

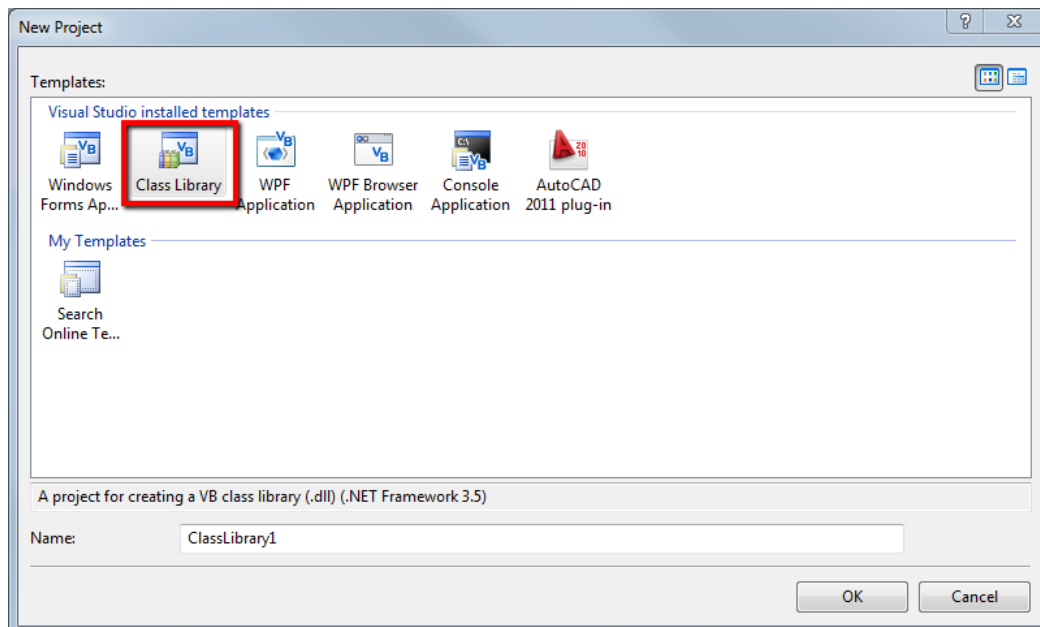
Once the user has his selections he selects an ok button and the function runs through the selected alignments and applies the selected style to it.

Because we have created a flow chart, we understand, not only the function but also, the steps to accomplish the function and what traps may be required.

CREATING THE VISUAL STUDIO PROJECT

There are many .Net languages that can be used and currently all Visual Studio packages (2003-2010) can be used to build add-ons to AutoCAD Civil 3D 2011. However, only 2008 is truly supported. Thus, we are going to use Visual Studio 2008 Express – Visual Basic Edition to accomplish our project.

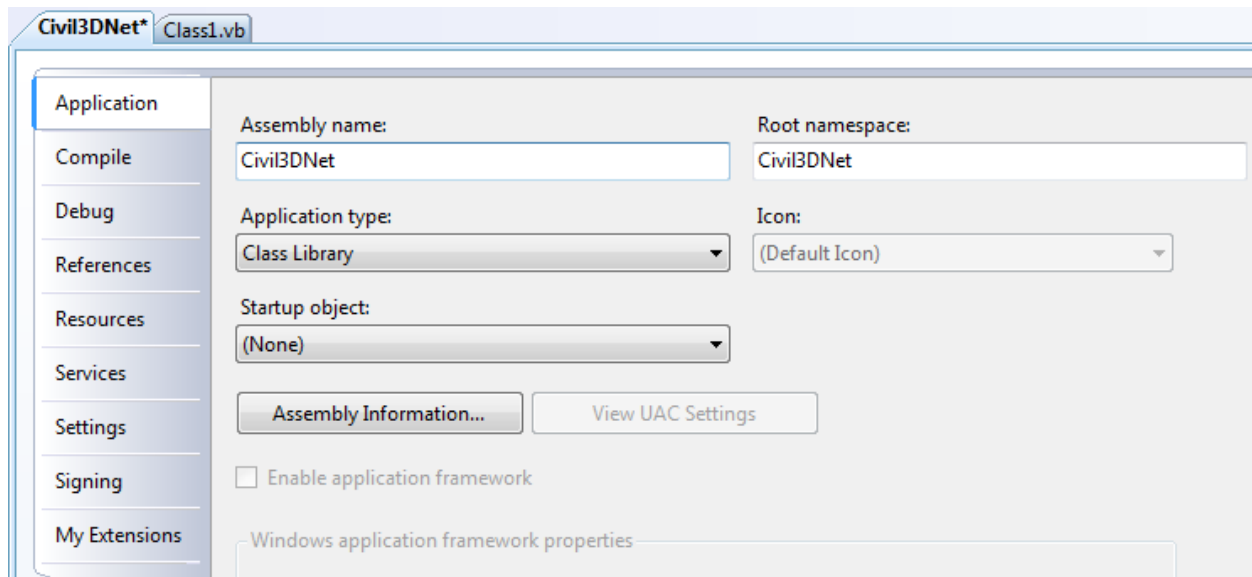
First we open VS and start a new “Class Library” project template (We could use the plug-in but just in case the plug in is not available or does not work, we are going to build one from scratch).



We are going to name this project Civil3DNet. In our Solution Explorer (you can open from View>Solution Explorer), we see that our project (and solution) named Civil3DNet is created and one class is already created called “Class1.vb”.

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.



Notice that our assembly name (or dll) 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

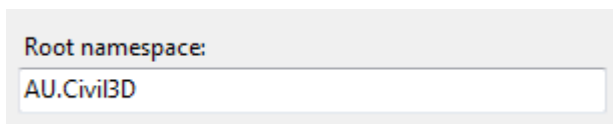
Namespaces are what you might consider a directory structure (tree structure) for libraries. Thus, it is a good practice to assign namespaces that organize all your libraries under one major namespace (think Root directory).

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.

In this exercise 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”.

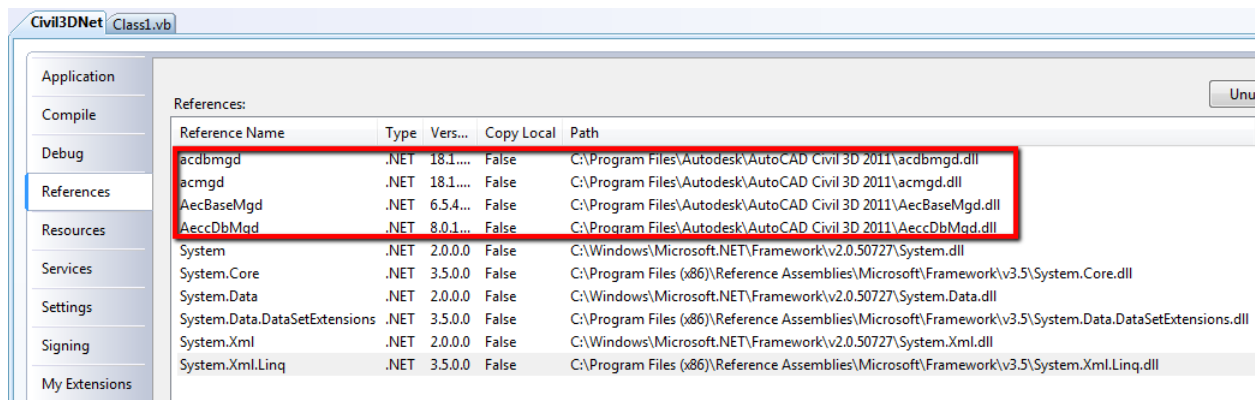


REFERENCES

If we go down to the References option and click on this, we are going to add four references. Each one of these is located in “C:\Program Files\Autodesk\AutoCAD Civil 3D 2011” (It is best to use references from the ObjectARX SDK where possible). To add the references we click on the add button about mid-right of the references section. The four references we are looking for are:

- acdbmgd.dll
- acmgd.dll
- AecBaseMgd.dll
- AeccDbMgd.dll

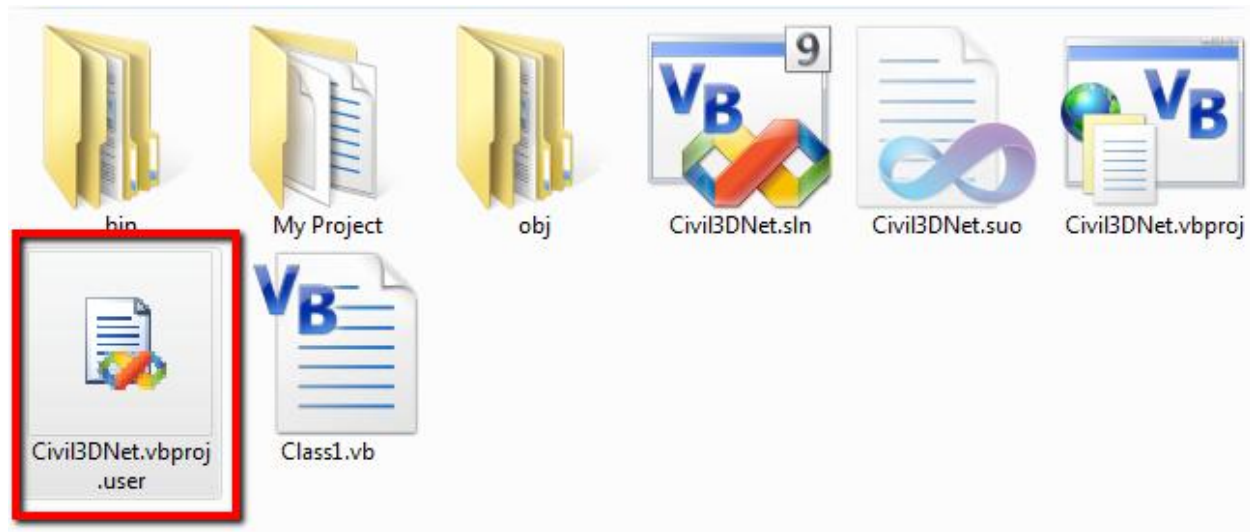
Since we are building an add-on, then when our software runs these references will already exist in and be called from Autodesk. Thus, we need to set each of these references to not be copied with our project. We do this by selecting these newly added references and setting their “Copy Local” parameter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



DEBUGGING IN EXPRESS

We will want to test and debug our software but VS Express does not give us a way to set an application to run for debugging in our project properties. Thus we have to do a little more manual legwork.

First, let's save the project that we have so far. Now close the VS Express application and, using Windows Explorer, go to the project directory. Select the Civil3DNet.vbproj.user file and open the file using Notepad.



Once open replace all the text with the following:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <StartAction>Program</StartAction>
    <StartProgram>C:\Program Files\Autodesk\AutoCAD Civil 3D 2011\acad.exe</StartProgram>
  </PropertyGroup>
</Project>
```

Save and close the file and reopen the project in VS Express. The project is now ready for debugging.

BUILDING OUR FUNCTION

Let's focus now on that class in our Solutions Explorer. We want to right-click on the "Class1.vb" file and rename it to "Commands.vb". Next, let's open this class.

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:

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

Imports Autodesk.Civil.Land.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 a envelope to hold an object or objects. So a file could contain multiple classes.

THE COMMAND

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

```
Public Class Commands
    <CommandMethod("SetAlignmentStyles")> _
    Public Sub cmdSetAlignStyles()

        End Sub
End Class
```

So we have a command name of “SetAlignmentStyles” assigned to a public sub named “cmdSetAlignStyles”. 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 was 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:

```
Dim civDoc As Autodesk.Civil.ApplicationServices.CivilDocument
civDoc = Autodesk.Civil.ApplicationServices.CivilApplication.ActiveDocument
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 and why and exit the function.

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

So what we have so far is:

```
<CommandMethod("SetAlignmentStyles")> _
Public Sub cmdSetAlignStyles()
    Dim ed As Editor = Application.DocumentManager.MdiActiveDocument.Editor

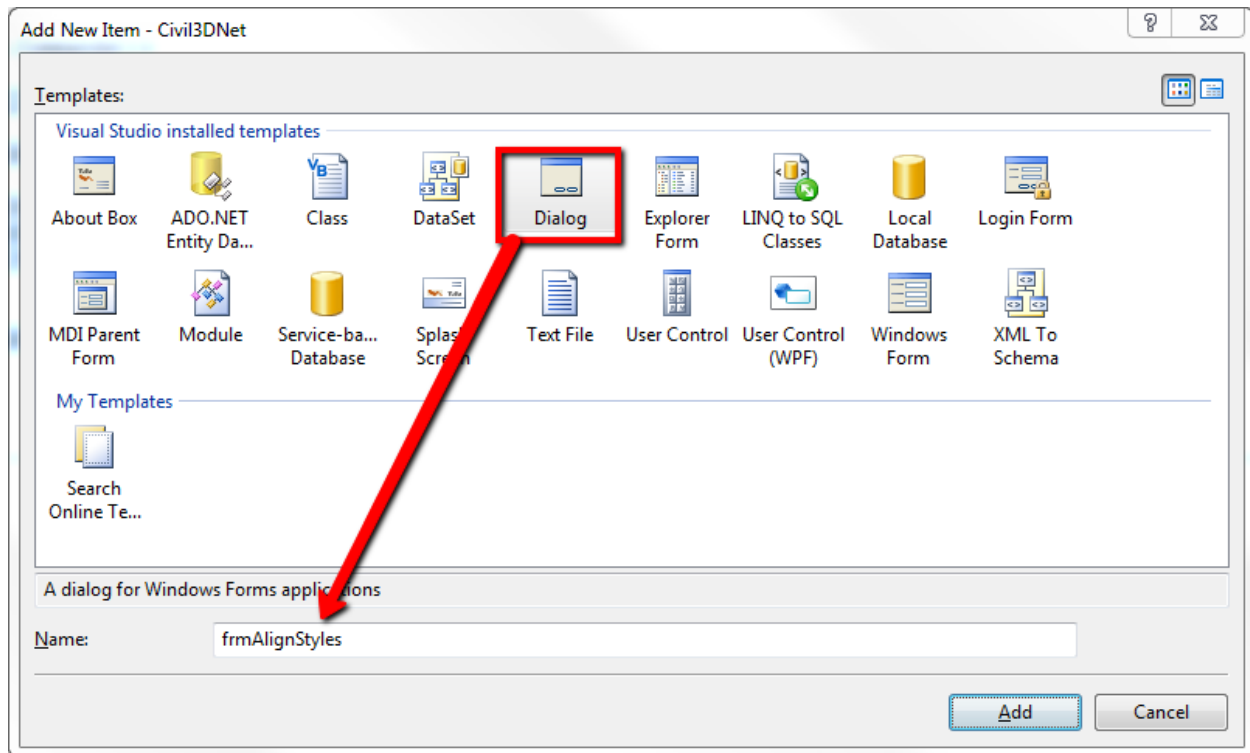
    Dim civDoc As Autodesk.Civil.ApplicationServices.CivilDocument
    civDoc = Autodesk.Civil.ApplicationServices.CivilApplication.ActiveDocument
    Dim alignIds As ObjectIdCollection = civDoc.GetAlignmentIds

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

Notice that we added 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.

CREATING THE FORM

Okay, so we have created the command and built a check to see if there are any alignments in the drawing. If there are alignments then we are going to show a form with the alignments in a list. Let’s create a new dialog form by right clicking on our project in the Solutions Explorer and choosing “Add New Item”. From the New Item dialog, we want to select a *Dialog* item and name it *frmAlignStyles*.



A dialog is just a form with a few parameters preset for us and the ok and cancel buttons already added. We now have our base form created for us. Let's change the text for our form title to read "Set Alignment Styles". Other parameters to set for the form:

- FormBorderStyle= Sizable
- MinimumSize=436,330
- ShowIcon=False
- Size=436,330

First let's add a *Button* from the Toolbox (View>Toolbox). The *Button* will have the following parameters:

- Name= btnAlignPick
- Anchor=Top,Right
- Location=384,8
- Size=38,23
- Text=Pick

Now let's add a *DataGridView* object. We need to set the following parameters for the *DataGridView* object.

- Name = dgvAlign
- AllowUserToAddRows=False

- AllowUserToDeleteRows=False
- Anchor=Top,Bottom,Right,Left
- AutoSizeColumnMode=Fill
- Columns=
 - Checked
 - HeaderText=" "
 - DataPropertyName=Selected
 - SortMode=NotSortable
 - ColumnType=CheckBox
 - FillWeight=15
 - cName
 - HeaderText="Name"
 - DataPropertyName=Name
 - SortMode=Automatic
 - ColumnType=TextBox
 - FillWeight=65
 - Desc
 - HeaderText="Description"
 - DataPropertyName=Description
 - SortMode=NotSortable
 - ColumnType=TextBox
 - FillWeight=100
 - cId
 - HeaderText="Id"
 - DataPropertyName=Id
 - Visible=False
 - SortMode= NotSortable
 - ColumnType=TextBox
 - FillWeight=1
- Location=8,8
- MultiSelect=True
- RowHeadersVisible=False
- Size=370,184

- SelectionMode=FullRowSelect

Now add a *Label* and line it up under the *DataGridView* but far enough that the user sees that they are not connected. The *Label* should have the following parameters:

- Text=Select Alignment Style
- Anchor=Bottom,Left
- Location=5,205

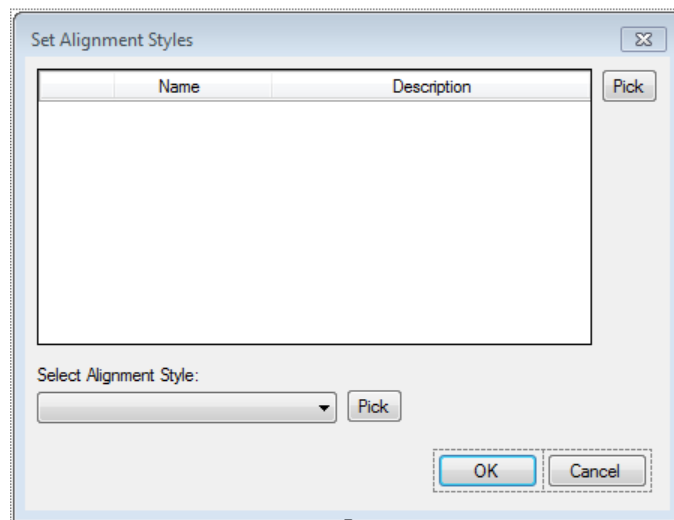
Next add a *ComboBox* object right under the label. The parameters for the *ComboBox* object should be:

- Name=cboStyle
- Anchor=Bottom,Left
- DropDownStyle=DropDownList
- Location=8,223
- Size=200,21
- Sorted=False

Finally copy the Pick button, paste and align it just right of the *ComboBox* object. The new *Button* should have the following parameters:

- Name=btnStyle
- Anchor=Bottom,Left
- Location=214,221

When you are done, the form should look similar to the figure below.



POPULATING THE FORM

To fill the form we need to get the collection of alignment objects and alignment styles. To do this we have to understand AutoCAD *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 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 SPECIAL CLASSES

We are going to create two special *Classes* – one for styles and one for objects. To do this, we are going to create another *Class* which will contain functions for alignments.

First let's create a Civil 3D Object *Class*. To do this we are going to right-click on the Solutions Explorer and add an Objects class file. However, we are going to create a new *Namespace* in this file called Objects and build both the styles and object *Class* in this one file. Hence, open the new file and above the *Class* name add "Namespace Objects" and rename the *Class* to "BaseObject".

You should have the following:

```
Namespace Objects
    Public Class BaseObject

    End Class
End Namespace
```

Now we are just going to add a few *Properties* to the *Class*. *Properties* are a simple way of storing data which can be called and even changed. These *Properties* will be stored inside the object through private variables. The *Properties* will be Name, Description, Selected, and Id. So our *Class* will look like the following:

```
Public Class BaseObject
    Private objName As String = Nothing
    Public Property Name() As String
        Get
            Name = objName
        End Get
        Set(ByVal value As String)
            objName = value
        End Set
    End Property
    Private objDesc As String = Nothing
    Public Property Description() As String
```

```

    Get
        Description = objDesc
    End Get
    Set(ByVal value As String)
        objDesc = value
    End Set
End Property
Private objId As Autodesk.AutoCAD.DatabaseServices.ObjectId = Nothing
Public Property Id() As Autodesk.AutoCAD.DatabaseServices.ObjectId
    Get
        Id = objId
    End Get
    Set(ByVal value As Autodesk.AutoCAD.DatabaseServices.ObjectId)
        objId = value
    End Set
End Property
Private objSelected As Boolean = False
Public Property Selected() As Boolean
    Get
        Selected = objSelected
    End Get
    Set(ByVal value As Boolean)
        objSelected = value
    End Set
End Property
End Class

```

Next, we are going to create another *Class* for the styles. However, we again want to store this in another *Namespace*. We are planning for the future where we may need to grow (or inherit) from these *Classes*. Consequently, right below the Objects namespace we are going to add the Styles *Namespace*. Inside the Styles *Namespace*, we are going to add the BaseStyle *Class*. It will also have similar properties to the base object (This is just an exercise, remember. Styles though have very different properties). We will have a name and id. Your BaseStyles *Class* will look like:

```

Namespace Styles
    Public Class BaseStyle
        Private objName As String = Nothing
        Public Property Name() As String
            Get
                Name = objName
            End Get
            Set(ByVal value As String)
                objName = value
            End Set
        End Property

        Private objId As Autodesk.AutoCAD.DatabaseServices.ObjectId = Nothing
        Public Property Id() As Autodesk.AutoCAD.DatabaseServices.ObjectId
            Get
                Id = objId
            End Get
            Set(ByVal value As Autodesk.AutoCAD.DatabaseServices.ObjectId)
                objId = value
            End Set
        End Property
    End Class
End Namespace

```

```

        End Set
    End Property
End Class
End Namespace

```

CREATING THE FUNCTIONS

Now that we have the base *Classes* built, let create some functions that will create a list of these *Classes* 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 Alignments. It will belong in the Functions *Namespace*.

Our first function will return a list of BaseObject *Class* objects that contain data about all the alignments in the drawing. We will include a *Parameter* called “IncludeDrefs” which will allow the function to filter the list to include only alignments that reside in the drawing or all including referenced alignments. We also set our function to be shared so that we can call it without creating an instance of the *Class*. What we have so far is:

```

Namespace Functions
    Public Class Alignments
        Public Shared Function GetAlignments(ByVal IncludeDrefs As Boolean) _
                                            As List(Of Objects.BaseObject)

        End Function
    End Class
End Namespace

```

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.

```

Public Shared Function GetAlignments(ByVal IncludeDrefs As Boolean) As _
    List(Of Objects.BaseObject)

    Dim civDoc As Autodesk.Civil.ApplicationServices.CivilDocument
    civDoc = _
    Autodesk.Civil.ApplicationServices.CivilApplication.ActiveDocument

    Dim alignIds As ObjectIdCollection = civDoc.GetAlignmentIds

    If alignIds.Count = 0 Then
        Return Nothing
    End If

End Function

```

Finally, lets create a *Transaction* and iterate through our Id collection to create a list (or collection) of BaseObjects 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.

```

Dim aligns As New List(Of Objects.BaseObject)
Dim aBobj As Objects.BaseObject
Using trans As Transaction = _

```

```
HostApplicationServices.WorkingDatabase.TransactionManager.StartTransaction
```

```
End Using
```

What we have done is called on the working database to start (create) a new *Transaction* and assigned this *Transaction* to the “Trans” variable. The *Transaction* will be disposed of at the *End Using* statment.

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

```
Public Shared Function GetAlignments(ByVal IncludeDrefs As Boolean) As _
    List(Of Objects.BaseObject)
    Dim civDoc As Autodesk.Civil.ApplicationServices.CivilDocument
    civDoc = Autodesk.Civil.ApplicationServices.CivilApplication.ActiveDocument
    Dim alignIds As ObjectIdCollection = civDoc.GetAlignmentIds

    If alignIds.Count = 0 Then
        Return Nothing
    End If

    Dim aligns As New List(Of Objects.BaseObject)
    Dim aBobj As Objects.BaseObject
    Using trans As Transaction = _
        HostApplicationServices.WorkingDatabase.TransactionManager.StartTransaction
        Dim align As Autodesk.Civil.Land.DatabaseServices.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 Objects.BaseObject
                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
```

Whew!! Now let’s create a function which will create a list of the alignment *STYLES*. Since we have created the hard one first, this will go a little easier. There are no parameters to the function and there will always be an alignment style so we don’t have to check to see if there are any. We can go straight to the *Transaction*.

```
Public Shared Function GetAlignmentStyles() As List(Of Styles.BaseStyle)
    Dim civDoc As Autodesk.Civil.ApplicationServices.CivilDocument
```

```

civDoc = Autodesk.Civil.ApplicationServices.CivilApplication.ActiveDocument

Dim aStyles As New List(Of Styles.BaseStyle)
Dim aStyleObj As Styles.BaseStyle
Using trans As Transaction = _
    HostApplicationServices.WorkingDatabase.TransactionManager.StartTransaction
    Dim aStyle As Autodesk.Civil.Land.DatabaseServices.Styles.AlignmentStyle
    For Each id As ObjectId In civDoc.Styles.AlignmentStyles
        Try
            aStyle = trans.GetObject(id, OpenMode.ForRead)
            aStyleObj = New Styles.BaseStyle
            aStyleObj.Id = aStyle.Id
            aStyleObj.Name = aStyle.Name
            aStyles.Add(aStyleObj)
        Catch
        End Try
    Next
End Using

Return aStyles
End Function

```

Now that we have the functions built we can go back to our form and add calls to these functions to populate the form.

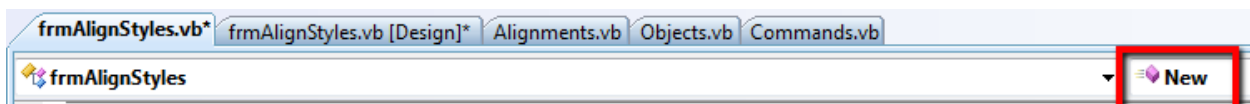
We need to go to the code behind the form. We want to add two *Friend* variables – one for the alignments list and one for the styles list. Because they are declared as *Friends*, we will be able to see them in the rest of our project.

```

Friend aligns As List(Of Objects.BaseObject)
Friend aStyles As List(Of Styles.BaseStyle)

```

We then have to get the sub “New” which we can get from the pulldown in the form’s code tab.



In this sub, we will add calls to the functions we just created and populate the variables from above.

```

aligns = Functions.Alignments.GetAlignments(True)
aStyles = Functions.Alignments.GetAlignmentStyles

```

Next we link this data to our objects. We do this by assigning the *DataSource* for the objects and also assigning the member to use for displaying data. The *DataGridView* data to display for each row was already assigned in the *DataPropertyName* parameter. Thus, we only have to assign the display member for the combo box. However, ultimately we will use the *ObjectId* to get the style so we can also assign this as the value of the selection. One value (Name) is displayed to the user while another value (ObjectId) is provided to us.

```

Me.dgvAlign.DataSource = aligns

Me.cboStyle.DataSource = aStyles
Me.cboStyle.DisplayMember = "Name"

```



```
Me.cboStyle.ValueMember = "Id"
```

SETTING UP THE GRAPHICAL SELECTIONS

In our form, we have created an option where they can graphically select alignments and can also select an alignment to get the style selection from. Let's build the code for this.

To do this we need to create an *Event*. Visual Studio makes this somewhat effortless. If we go back to the form's design and double click on the *btnAlignPick* object, VS will create a new sub routine which is linked (or handles) the button click event.

```
Private Sub btnAlignPick_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnAlignPick.Click

End Sub
```

Now we want to add a command line selection prompt. We also want to build a filter that selects only alignments. First, we need to get the current document's *Editor*. Then we need to build the options for the selection prompt. We create a new *PromptSelectionOptions* to do this.

```
Dim ssOpts As New PromptSelectionOptions
With ssOpts
    .AllowDuplicates = False
    .AllowSubSelections = False
    .MessageForAdding = vbLf & "Select alignments: "
    .MessageForRemoval = vbLf & "Remove alignments: "
    .RejectObjectsFromNonCurrentSpace = True
    .RejectObjectsOnLockedLayers = False
End With
```

Now we create a selection filter using an array of *TypedValues*. The array only contains a filter for the object type but we could add all kinds of filters here (layers, linetypes, etc.). Finally, we create a *PromptSelectionResult* variable and call the *GetSelection* method from the *Editor*.

```
Dim Values() As TypedValue = {New TypedValue(DxfCode.Start, _
    "AECC_ALIGNMENT")}
Dim fil As SelectionFilter = New SelectionFilter(Values)
Dim ssRes As PromptSelectionResult = ed.GetSelection(ssOpts, fil)
```

Now if the selection does not return ok then we have nothing to do and we can just exit the routine. However, if it does return ok then it contains some selected objects. We want to iterate the selected objects and mark these as selected in the *DataGridView*. We already have the *ObjectIds* of the objects and this is one of the things available in the *PromptSelectionResult.SelectedObjects* collection. Let's use this to our advantage by building another function in our *Functions.Alignment Class*. This function will search through a list of *BaseObjects* and return the first one in the list that has the same *ObjectId* as what is fed to the function. (This function should reside in a *Functions.General Class* but for sake of this example, we will place it in the *Functions.Alignment Class*).

```
Public Shared Function FindAlignmentById(ByVal AlignmentId As ObjectId, _
    ByVal AlignmentsList As List(Of Objects.BaseObject)) As Objects.BaseObject
```

```

If AlignmentsList Is Nothing Then Return Nothing
If AlignmentsList.Count = 0 Then Return Nothing

For Each align As Objects.BaseObject In AlignmentsList
    If align.Id = AlignmentId Then Return align
Next
Return Nothing
End Function

```

As we can see the power of our little base classes allow us to avoid *Transactions* a lot of time. Don't confuse the POWER of a *Transaction* with it being a BAD thing. However, misunderstanding and/or abuse of that power is often what causes the most crashes in custom add-ons. Even if we use *Transactions* correctly, can we guarantee that other add-ons that reside on a user's computer are using *Transactions* correctly? And since add-ons can fire using events, other add-ons may be working at the same time as us. Thus, we use the POWER of *Transactions* when we need to only.

All right let's get back to our selection results. All we do now is loop through the selected objects and find the associated BaseObject using the new function above. We then change the *Selected Property* of that BaseObject to True. Since our *DataGridView* is linked to the data of our list of BaseObjects the *DataGridView* automatically updates.

```

Dim aObj As Objects.BaseObject
For Each selObj As SelectedObject In ssRes.Value
    aObj = Functions.Alignments.FindAlignmentById(selObj.ObjectId, Me.aligns)
    If aObj Is Nothing Then Continue For
    aObj.Selected = True
Next

```

So our finished routine for graphically selecting and adding alignments to our checked list is:

```

Private Sub btnAlignPick_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnAlignPick.Click
    Dim ed As Editor = _
Autodesk.AutoCAD.ApplicationServices.Application.DocumentManager.MdiActiveDoc
ument.Editor

    Dim ssOpts As New PromptSelectionOptions
    With ssOpts
        .AllowDuplicates = False
        .AllowSubSelections = False
        .MessageForAdding = vbLf & "Select alignments: "
        .MessageForRemoval = vbLf & "Remove alignments: "
        .RejectObjectsFromNonCurrentSpace = True
        .RejectObjectsOnLockedLayers = False
    End With
    Dim Values() As TypedValue = {New TypedValue(DxfCode.Start, _
        "AECC_ALIGNMENT")}
    Dim fil As SelectionFilter = New SelectionFilter(Values)
    Dim ssRes As PromptSelectionResult = ed.GetSelection(ssOpts, fil)
    If ssRes.Status <> PromptStatus.OK Then Exit Sub

    Dim aObj As Objects.BaseObject
    For Each selObj As SelectedObject In ssRes.Value

```

```

aObj = Functions.Alignments.FindAlignmentById(selObj.ObjectId, Me.aligns)
If aObj Is Nothing Then Continue For
aObj.Selected = True
Next
End Sub

```

Our last support function we have to build in our form is for the user to graphically select an alignment to get the selected alignment style. The big difference in our interacting with the user in this case is that we want to allow only a selection of one entity and not a selection set. So we use different tools in the *EditorInput* called *PromptEntityOptions* and *PromptEntityResult*. Then we will actually have to fire a *Transaction* and query the alignment object to get the alignment style's *ObjectId*. From there all we will have to do is set the *ComboBox*'s *SelectedValue* to the alignment style's *ObjectId*.

The button handler for the style selection should look similar to the following:

```

Private Sub btnStyle_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnStyle.Click
    Dim ed As Editor = _
Autodesk.AutoCAD.ApplicationServices.Application.DocumentManager.MdiActiveDoc
ument.Editor

    Dim entOpts As New PromptEntityOptions(vbLf & _
        "Select alignment to get style from or <return to form>: ")
    With entOpts
        .SetRejectMessage(vbLf & "Object is not a Civil 3D alignment")
        .AddAllowedClass( _
            GetType(Autodesk.Civil.Land.DatabaseServices.Alignment), True)
        .AllowNone = True
        .AllowObjectOnLockedLayer = True
    End With
    Dim entRes As PromptEntityResult = ed.GetEntity(entOpts)
    If entRes.Status <> PromptStatus.OK Then Exit Sub

    Using trans As Transaction = _
HostApplicationServices.WorkingDatabase.TransactionManager.StartTransaction
        Try
            Dim align As Autodesk.Civil.Land.DatabaseServices.Alignment
            align = trans.GetObject(entRes.ObjectId, OpenMode.ForRead)
            Me.cboStyle.SelectedValue = align.StyleId
        Catch
        End Try
    End Using
End Sub

```

Notice that I always add a Try..Catch inside all of my *Transactions*. This way if anything crashes while in a transaction, I still dispose of my transaction.

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, learn how to show our form, and actually adjust our selected alignments.

First to show our form, it is best to use the Autodesk method to show the form as part of the AutoCAD application. Create a new instance of the form and then show it using the *Autodesk.ApplicationServices.Application.ShowDialog* method. This will return a dialog result which we can check.

```
Dim frm As New frmAlignStyles
If Application.ShowDialog(frm) <> _
Windows.Forms.DialogResult.OK Then Exit Sub
```

Since we know it returned okay because we are still in our sub 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*.

```
Using lck As DocumentLock = _
    Application.DocumentManager.MdiActiveDocument.LockDocument
Using trans As Transaction = _
    HostApplicationServices.WorkingDatabase.TransactionManager.StartTransaction
Try
    Dim align As Alignment
    For Each aObj As Objects.BaseObject In frm.aligns
        If aObj.Selected = False Then Continue For
        align = trans.GetObject(aObj.Id, OpenMode.ForWrite)
        align.StyleId = frm.cboStyle.SelectedValue
    Next
Catch
End Try
trans.Commit()
End Using
End Using
```

That is IT! We have done it!

DEPLOYING OUR SOFTWARE

Once we build our project DLL, we need to deploy it. To have it automatically load into AutoCAD Civil 3D we will need to write to the registry. The best way to do this is use an Installer. Unfortunately, Microsoft's Visual Studio Setup Projects don't really give you the ability to check for an existing installed application such as whether AutoCAD Civil 3D 2011 is installed and, if so, where. Consequently, you have to use another Installer software. Personally, I LOVE the Shareware application called InnoSetup (<http://www.jrsoftware.org/isinfo.php>). A sample InnoSetup project is included in the documentation. The solution is to create an application in the Applications Key of the AutoCAD Civil 3D registry. For example, in *HKCU(HKLM)/Software/Autodesk/AutoCAD/R18.1/ACAD-9000:409/Applications* we add a new *Key* called *AUCivil* which has the following values:

- String Value named DESCRIPTION
 - This is simply a description of our software so we will set the value to “AU Sample”.
- A DWord(32 bit) Value named LOADCTRLS
 - This tells AutoCAD when and how to load the application. By setting it to “2”, we will load our application at startup.
- A String Value named LOADER
 - This is the path to our DLL file and tells AutoCAD where to load it from. Set this to the path that the DLL was installed in the deployment process.
- A DWord(32 bit) Value named MANAGED
 - This tells AutoCAD that this is a managed .Net dll (versus an ObjectARX application). We set this to “1” because we are a managed application.

OTHER RESOURCES

Please feel free to contact me at joshua.modglin@inmotioncon.com . You can also check out some of the other online resources to learn more about Civil 3D .Net Customization.

<http://inmotioncon.com>

<http://forums.autodesk.com/t5/AutoCAD-Civil-3D-Customization/bd-p/190>

<http://www.civil3d.com/category/civil3dcom/api/>

<http://www.civil3d.com/category/civil3dcom/programming/>

See you next Year!!